RHEINISCHE FRIEDRICH-WILHELMS-UNIVERSITÄT BONN

Component-Based Tailorability

Dissertation

zur Erlangung des Doktorgrades (Dr. rer. nat.)

der

Mathematisch-Naturwissenschaftlichen Fakultät

der

Universität Bonn

vorgelegt von

Oliver Stiemerling

aus

Bonn

Bonn, im Mai 2000

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Rheinischen Friedrich-Wilhelms-Universität Bonn

1. Referent: Prof. Dr. Armin B. Cremers

2. Referent: Prof. Dr. Rainer Manthey

Tag der Promotion: 3.7.2000

Abstract

This dissertation addresses the question of how to use software components to build tailorable software systems. Tailorable software systems can be adapted to changing or diversified requirements after initial development and deployment. Traditionally, software components are used in the development phase of a software system in order to reduce costs and increase quality. Here, novel principles and methods are developed and evaluated that permit the maintenance and manipulation of a system's compositional structure during its use.

A first exploratory experiment using existing component technologies shows that the approach of component-based tailorability raises two main technical problems. First, one has to design an appropriate *component model* that permits the decomposition of a software system according to its expected evolution. The experiment shows that the appropriateness of a component model is primarily determined by the forms of component interaction it supports. If the interactions required by a given decomposition cannot be implemented, the approach fails to provide the necessary range of tailorability. Second, one needs a *runtime and tailoring environment* that manages the system's component structures. The environment should distinguish between a component plan and its instances, thus permitting the sharing of plans and their subsequent changes. Furthermore, in the case of shared component plans, a method is needed for restricting the effect of a change to subset of instances.

Both problems are first addressed on a formal level. Within the framework of data space theory, a mathematical model of component interaction is developed. Based on this model, a theorem shows that the combination of two forms of basic interaction (events and shared variables) permits the high-quality implementation of interactions resulting from arbitrary system decompositions. The second problem is addressed by a formal model of a runtime and tailoring environment, defining its central elements and methods, in particular the runtime tailoring operations.

As proof-of-concept, both solutions are implemented in the JAVA programming language. The FLEXIBEANS component model extends the existing JAVABEANS model with the shared variable interaction style demanded by the theoretical results. The EVOLVE platform is a runtime and tailoring environment for component-based multi-user systems that are distributed over the Internet. In combination, FLEXIBEANS and EVOLVE are employed to provide a number of example systems with the property of tailorability. This shows the applicability of the component-based tailorability approach. The final discussion points to directions of future research.

to my parents

Acknowledgements

Many people have provided invaluable help during the completion of this dissertation and the work it presents. First of all, thanks are due to my supervisor, Professor Dr. Armin B. Cremers, for guiding me through three and a half years of work and providing decisive support at critical points of the way. Professor Dr. Rainer Manthey kindly agreed to be the second referee.

Next, I want to thank my colleagues and friends in ProSEC and the Software Technology Group for providing multi-faceted feedback on many aspects of my work. In particular, Günter Kniesel can claim full responsibility for its strong JAVA flavor. Tom Arbuckle helped with the language (Naturally, all remaining mistakes are my own).

During the practical phases of the work, I had the pleasure to cooperate with an exceptional team of Masters students. The component-based search tool was implemented by Markus Won. Ralph Hinken deserves credit for the implementation of the EVOLVE platform's object-oriented architecture. Michael Hallenberger made the concepts of component-based tailorability visible and understandable with his 3D tailoring interface. Gunnar Stevens and Christian Bohning developed applications using the FLEXIBEANS component model and thus strengthened the empirical basis of the approach. Thanks to all of you!

From other institutions, Jean-Yves Vion-Dury (Xerox Research Center Europe), Marc Spielmann (RWTH Aachen), Gunnar Teege and Michael Koch (TH München) also provided valuable feedback on parts of the work.

Since life is not only work, I also thank all my friends and in particular Frédérique and my family for their support during this time.

Table of Contents

Chapter 1

Introduction

1.1 Topic and Motivation

This dissertation addresses the question of how to use components to build tailorable software systems. Traditionally, the concept of software components is employed in the development phase of a software system's lifecycle in order to gain the benefits of reusing existing parts: decreased time-to-market, increased quality and reduced costs. The work presented here investigates the use of software components *after* development in order to additionally gain the benefit of tailorability – that is the possibility of adapting an already deployed system to changing or diversified requirements.

Tailorability is an important property of software systems employed to support commercial enterprises. Rapidly changing markets force companies to continuously adapt their products and processes to meet the demands of their customers. As far as these commercial activities are supported by IT systems, rigidity in system design poses a serious threat to the survival of a company.

Apart from dynamic requirements, diversity is another driving factor for tailorability. Today, completely custom-built systems are the exception, because it is much more economical to tailor off-the-shelf products to an organization's specific requirements. On a smaller scale, even each user in an organization can have different requirements for a particular software system, owing to his or her specific tasks or preferences. From the vendor's perspective, a tailorable software system promises to reach a larger market segment than a rigid one.

Finally, as software systems become more complex and increasingly have implications for people's lifestyles and working environments, exact requirements are often uncertain or even controversial. An organization's management might have rather different opinions on the tasks and functionalities of a software system as opposed to those of the employees. Requirements emerge or change as the consequences of the system's introduction become apparent for everybody involved. Owing to the intertwined nature of work organization and system design, requirements are often the theme of an ongoing negotiation process between the powers-that-be in an organization (or even between organizations, e.g. in a IT supported supplier-customer relationship).

Tailorability is a means to deal with dynamic, diversified and uncertain requirements. However, implementing tailorability is a distinctively non-trivial task due to the multi-faceted nature of the problem. A software developer has to determine the aspects of the software system that are supposed to be tailorable, build a software architecture which provides the required flexibility, and finally has to implement tailoring mechanisms which control and perform the changes during the use of the system. On one end of the spectrum, these mechanisms can have the form of user interfaces for tailoring, which place the full flexibility in the hands of the users and system administrators (*adaptability*), or – on the other end – they can resemble intelligent agents which tailor the system automatically (*adaptivity*). In the first case – adaptability – the question is raised, who performs the tailoring operations and how the often-collaborative process of tailoring is supported – either by technical or organizational means or both.

Thus the software developer is confronted with several challenges requiring expertise in a number of rather different areas. Furthermore, these challenges arise anew for each application he or she develops. While the first question – concerning the tailorable aspects of a system – is application specific, there is no apparent reason why the architecture and tailoring mechanisms have to be re-invented and re-implemented each time.

The idea pursued in this work is to employ a generic – that is an application-independent – architectural concept to form the basis of an approach to tailorable system design. Such an approach promises concepts and implementations for tailoring architectures and mechanisms that can be re-used in many development projects, thus facilitating the work of the software developer. The ultimate objective is to create an approach that permits the developer to focus completely on the central question specific to the application he is designing: *which aspects are supposed to be tailorable*?

As a first step on the way towards this objective, the work presented here investigates the concept of *software components* as a basis for the implementation of re-usable tailoring architectures and mechanisms and develops the approach of *component-based tailorability*.

1.2 Component-Based Tailorability

In contrast to the traditional use of software components during development (figure 1.1-a), *component-based tailorability* maintains the component structure after initial composition, permitting re-composition of an application during its use (figure 1.1-b).

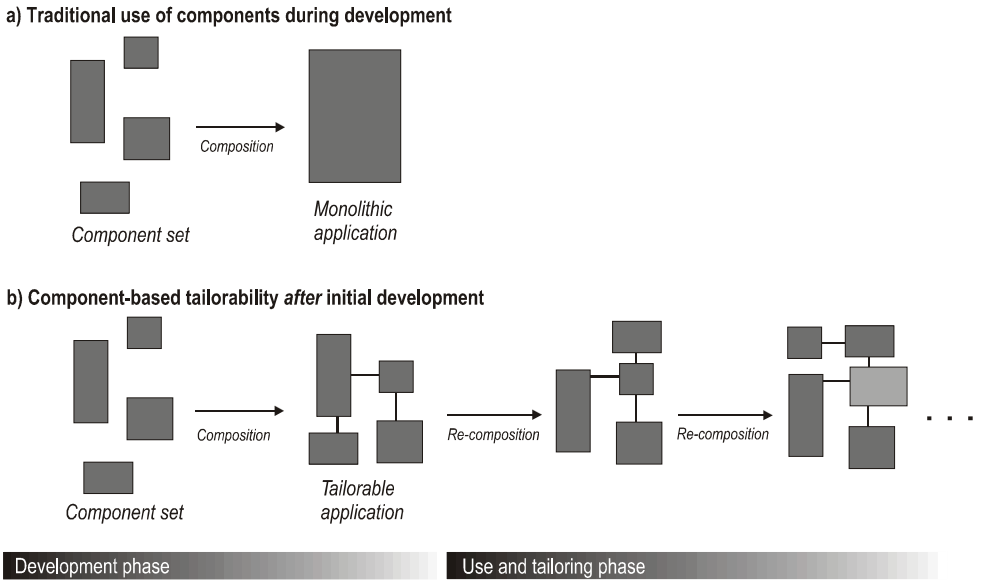


Figure 1.1: Component-based tailorability in contrast to traditional use of components

By re-composing the application, it is changed in order to meet different requirements. Thus in the context of the component-based tailorability approach, *tailoring an application* means *changing its composition*.

Why use software components as the fundamental concept for an approach to generic tailorability?

Foremost, the concept of a software component is independent of any specific application domain[1]. Components can be, and are, applied in almost any type of software system. (Szyperski 1998) even goes as far as postulating the use of components as a "*law of nature*" in software engineering (p. xiii). Consequently, one can expect that an approach to tailorable systems, building on the concept of software components, is widely applicable. The functionality of the software system and the functionality for tailoring the system are treated as separate concerns. Thus components appear to be a good choice as a fundamental architectural concept for tailorability.

Furthermore, software components promise a number of benefits with respect to tailorable system design.

First, the notion of composition is hierarchical. Thus it is possible to inspect and manipulate the system at *different levels of complexity and abstraction*, as depicted in figure 1.2:

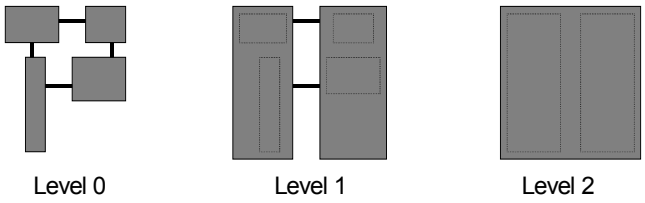


Figure 1.2: Different levels of a hierarchically composed system.

In figure 1.2, level 0 only shows instances of atomic components. Level 1 depicts the same system as a composition of two complex components (the underlying atomic components of level 0 are denoted by the dotted rectangles). On level 2, the system is regarded as monolithic.

Tailorability on different levels of abstraction and complexity is considered useful for *adaptability* approaches, because it supports a gradual approach to tailoring (MacLean et al. 1990, Bentley and Dourish 1995). If a system supports hierarchical component-based tailorability, a user can start with simple parameterization of the whole system (in figure 1.2, this is level 2) and later perform compositional tailoring operations (e.g. substituting the left component on level 1). A more experienced user such as a system administrator might tailor the system on level 0. When going to a lower level, the supported tailoring operations remain the same. Only the granularity and functionality of the components change. This determines the complexity but also the power of tailoring operations on that level. For similar reasons, hierarchical composition is also useful in *adaptivity* approaches. The same automatic tailoring mechanisms (for example in the form of intelligent agents) can be applied to different parts and levels of the system.

[1] While the basic concept of software components is application-independent, technical details obviously depend on application specific factors such as the need for distributed execution, programming languages, supported hardware platforms, required real-time capabilities etc. Consequently, an implementation of generic tailoring functionality will certainly not be applicable to every domain, but restricted to a certain – if large – class of applications. The implementation of the EVOLVE platform presented in chapter 8 is designed for JAVA-based multi-user client-server applications distributed over a TCP/IP network. While chapter 9 demonstrates that the EVOLVE system is applicable to a range of different domains, it should certainly not be employed to design, for instance, aircraft control systems.)

The second expected benefit is the support of the *sharing and exchange of complex components* that represent tailored parts of the system. (Mackay 1990) observed that tailoring is often a cooperative activity that involves sharing and exchange of successful system adaptations. She noted, for instance, that experienced users of a UNIX graphical desktop environment provided beginners with tailored initialization files defining the appearance of the desktop environment. Complex components can be described, stored and distributed as files. They can be attached to an email or deposited in a shared workspace. Furthermore, since complex components can have several instances in the running system, changes to the composition of one complex component can be shared by all instances.

The third benefit is the inherent *extensibility* of component-based software systems. Components can be developed independently as long as they adhere to a common specification (the component model). Consequently, the tailorability of a system does not depend on the initial set of components. The set can be extended with new components in order to meet unexpected requirements. If the programming language supports dynamic class loading (as JAVA does), it is even possible to add newly developed components to a running system.

These expected benefits were the initial motivation to develop and investigate the component-based tailorability approach. The approach raises a number of technical problems that are elaborated in section 1.4. However, before going into technical detail, the following section gives the reader a first impression of the exemplary application of the component-based tailorability approach in the EVOLVE system.

1.3 The EVOLVE Platform

The EVOLVE platform is designed to support the deployment and subsequent tailoring of arbitrary distributed component-based multi-user applications. It is independent from domain specific functionality and can thus be used to provide many different software systems with the property of tailorability. Figure 1.3 depicts a simple example for a component-based application made tailorable by the EVOLVE platform – a *shared to-do list* employed by two users to coordinate their tasks (this application is described in more detail in chapter 7):

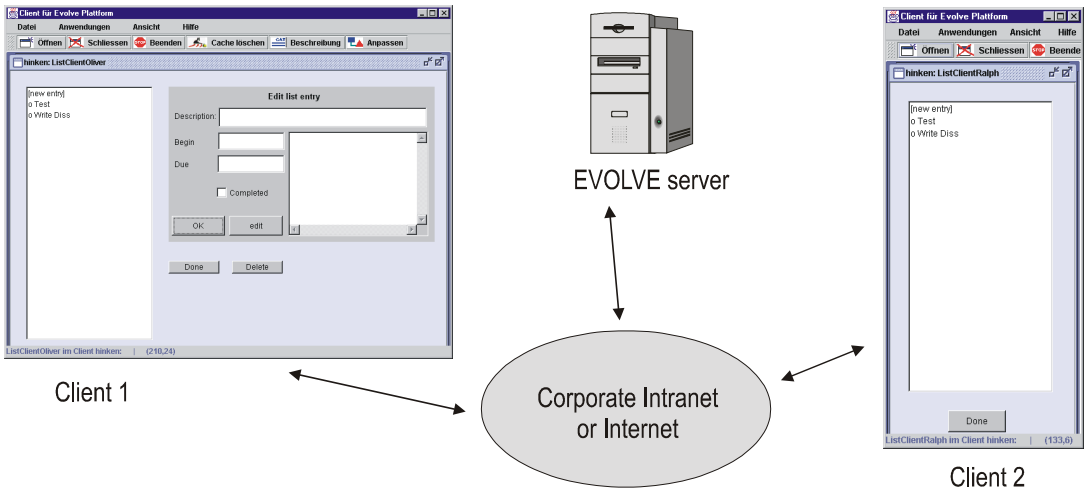


Figure 1.3: The EVOLVE platform supports multi-user client-server applications distributed over TCP/IP networks (here the *user perspective*, showing two clients of a shared to-do list application)

Client 1 belongs to a manager, client 2 to a subordinate. The actual data of the shared to-do list is stored on the EVOLVE server (in the middle of figure 1.3) employing a (invisible) server component. The clients are tailored to meet the requirements of their respective owners. While the subordinate may only see the contents of the list and mark entries as

"done", the manager can actually add new entries to the list and delete them. The distributed application is built from a set of components (visible ones like the buttons, the list and the editor; and invisible ones like the data storage component on the server).

In a traditional system, the composition would be static after development and deployment. The EVOLVE platform, however, maintains and permits the manipulation of the system's component structure. During the use of the system, a system administrator, outside consultant or even an end user can switch to the *tailoring mode* (figure 1.4) in which he can inspect and manipulate the entire distributed application (if he or she has the right to do so).

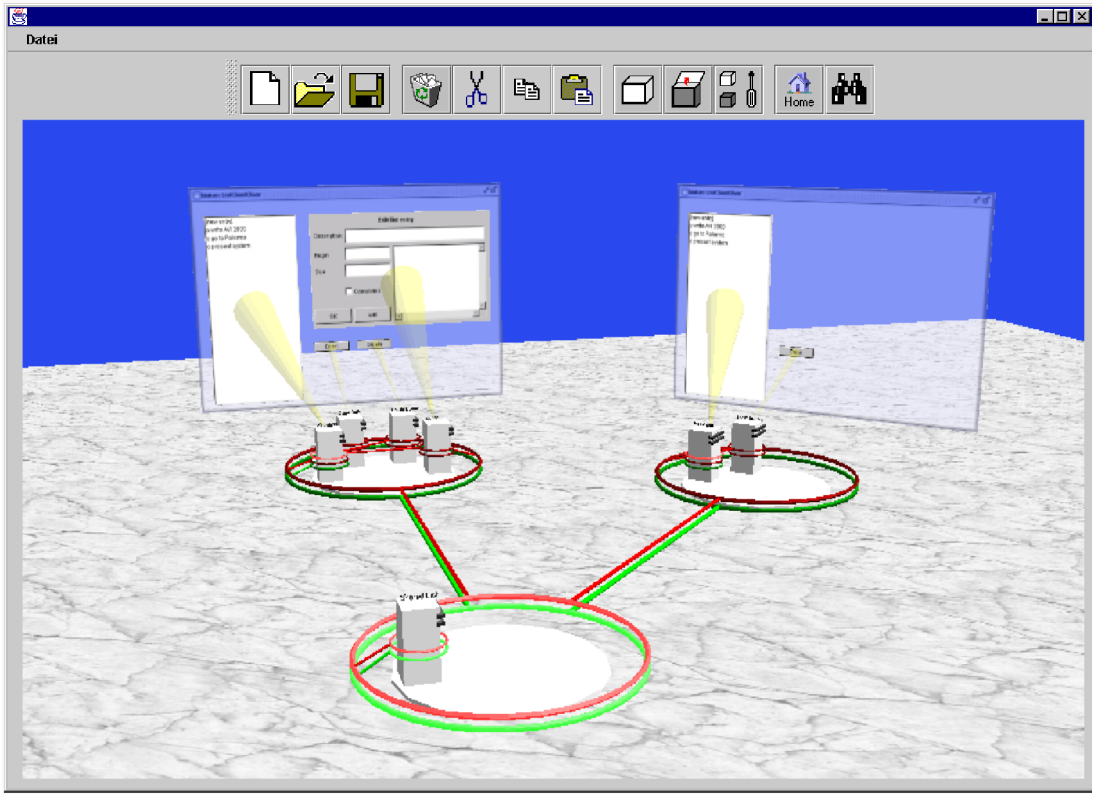


Figure 1.4: Screenshot of the 3D tailoring interface (showing the *tailoring perspective* of the application depicted in figure 1.3)

Figure 1.4 depicts a 3D interface for component-based tailoring that accesses the flexibility provided by the EVOLVE platform (One can also attach other types of tailoring mechanisms – e.g. for automatic tailoring. Component-based tailorability is not restricted to adaptivity approaches.) The two circles in the background represent the two clients, together with the virtual screens onto which appearance and position of the visible components are projected. All components – visible and invisible – are represented as boxes depicting the compositional structure of the application. The circle in the foreground represents the server that contains the invisible component for storing the contents of the shared to-do list.

The tailor can move around the 3D component scene and perform manipulations, e.g. concerning the positioning of the visible components on the screen or the connections of the components. He can remove component instances or add new ones from a repository (not shown in figure 1.4). The parameterization of component instances can also be manipulated. In short, the tailor has full control over every aspect of the application's composition, whenever the requirements of the supported group change.

In the shared to-do list example, the manager's trust in the subordinate might have increased enough to permit the subordinate to directly delete items from the shared to-do list – instead of only marking them as "done" for later control and deletion by the manager. In this case, the tailor would add a delete button component to the client of the subordinate.

While the example of adding a button is admittedly a rather simple tailoring operation, the EVOLVE platform permits the application of re-composition (and re-parameterization) operations to every part of a distributed software system. Furthermore,

- *tailoring operations can be performed during runtime*. In extremis, a system administrator can add the delete button, while the user is working with the shared to-do list. The system does not have to be shut down and the state of all other components is maintained.
- *tailoring operations can be performed remotely*. The whole system can be tailored from any workstation in the network. This feature supports models of centralized and decentralized system management.
- *the effect of tailoring operations can be shared among many users*. In the example, if other users share the definition of the subordinate client, the effect of the tailoring operations is propagated to all running instances of that definition.
- *the effect of tailoring operations can be restricted to subgroups of users*. This feature permits the accommodation of individual differences. In the example, there could, for instance, be a need to exclude a user who would misuse the delete button.
- *tailoring operations can be applied to every level of the compositional hierarchy*. A system administrator might inspect and manipulate the system on a very fine-grained level, while an end user might prefer a more high level view.

The EVOLVE platform and some more complex applications are described in detail in chapters 8 and 9. The presentation in this section is designed to give the reader only an introduction to the application of component-based tailorability. The next section is devoted to the fundamental questions that are raised when components are employed in this fashion.

1.4 Problem Areas and Research Approach

While tailorability raises questions in many different areas, this work focuses on the *technological questions* that have to be addressed in order to realize component-based tailorability. These questions are motivated by an *exploratory experiment*, during which component-based tailorability was applied to a "real world" tailoring problem (the design of the search tool in the POLITeam project, Cremers et al. 1998). The goal of the experiment (which is described in chapter 4) was to verify the fundamental feasibility of the approach. Furthermore, its analysis identifies the various shortcomings of the state-of-the-art software technologies employed with respect to the expected benefits presented in section 1.2. The resulting questions can be grouped into two problem areas (each consisting of two questions on different levels of generality). The following two subsections give an overview of these problem areas and how they are approached in this dissertation.

1.4.1 Problem Area 1: Component Interaction

A component model usually prescribes a limited set of possible component interaction styles (interaction primitives). The exploratory experiment demonstrates that the event interaction style prescribed by the JAVABEANS component model (which was used in the experiment) severely restricts the developer in implementing the desired set of components. These restrictions manifest themselves in inappropriate decompositions and in overly synchronized interactions. While the latter merely lead to inefficient applications, the former seriously undermine the objectives of the component-based tailorability approach. The decomposition of an application's functionality into a set of components determines the ease with which an application can be tailored and evolved over time. Thus, the first challenge in this problem area is to find a set of interaction primitives that permits the implementation of arbitrary decompositions.

Due to its fundamental nature, this question merits a treatment on a theoretical level. Within the context of an algebraic framework (data space theory) for modeling interacting computational entities, different interaction primitives are described and compared. The main result of these efforts is a theorem, which states that the problems encountered in the exploratory case study could be solved by a component model providing both event- and shared variable-like interaction styles.

The second question raised in this problem area is less fundamental and more programming language specific: How should direct remote interaction and named interaction ports, both of which are missing in the JAVABEANS component model, be implemented? In order to provide these missing features, the FLEXIBEANS component model is developed. Additionally, it provides the missing shared variable (object) interaction style motivated by the theoretical investigation.

Summarizing, the first problem area addressed in this work concerns the question of which notion of a component is appropriate for component-based tailorability.

1.4.2 Problem Area 2: Component Management

In contrast to approaches of component-based development, component-based tailorability obviously requires the maintenance of the compositional structures in a manipulatable form during the use of the system. Since several parts of the system can be structurally equivalent (e.g. GUIs for users with similar tasks), it is useful to distinguish *plans* for compositions and their *instances*. One plan can be shared by several instances, thus permitting changes to a single plan to take effect consistently in all its instances. One the other hand, sometimes the effect of changes to the plan is supposed to be restricted to a subset of instances (e.g. GUIs of users with special preferences). The review of related work and the exploratory experiment reveals that state-of-the-art approaches to component management are not sufficient for component-based tailorability, because they do not distinguish plans and instances.

Analogously to the problem of component interaction, component management is addressed on two levels. Based on set theory and first order predicate logic, an appropriate model of component management is developed. Its novel features are the explicit distinction between component plan and instance, and a scope control operation that restricts the effect of changes to specific subsets of plan instances. The model abstracts away from any programming language or platform and is thus widely applicable.

On the second, more practical level, the mathematically defined model is implemented as the EVOLVE system to support JAVA-based multi-user client/server applications distributed over a TCP/IP network. This implementation serves as a basis for the empirical evaluation of the techniques developed to solve the problems described above.

Summarizing, the second problem area concerns the question of how to manage the compositional structures of deployed software systems.

1.4.3 Evaluation

The objective of this work is to develop an approach that permits application developers to focus on the question which aspects of the application are supposed to be tailorable. Employing component-based tailorability, a developer's primary task is to design a set of components that can be composed in different ways to meet the various requirements. By deploying these components on top of the EVOLVE platform instead of monolithically composing them in a traditional development environment, the application becomes instantly tailorable without any additional effort by the developer.

In order to evaluate how close the work has come to achieving this ultimate objective, the obvious strategy is to apply the approach to software systems whose requirements are highly dynamic, diversified and uncertain. Owing to the volatile nature of cooperative work,

groupware systems are suitable candidates for the evaluation of the approach. Thus the applicability of the approach is demonstrated by implementing a number of groupware systems, two of which are discussed here.

Apart from the general applicability of the approach, the implementations are analyzed to empirically verify the theoretical results concerning component interaction. Furthermore, the scalability of the approach is investigated by measuring the performance of the EVOLVE platform in a variety of scenarios.

1.5 Contributions to the State-of-the-Art

On the most general level, this dissertation demonstrates how software components can be employed *after* development to build applications that are tailorable to meet dynamic and diverse requirements. Therefore, it builds a bridge between the technology- and the user-centered perspectives on system design. The component-based tailorability approach is a *technical* basis for building software systems that are *usable* in different environments for different tasks. Concrete contributions to the field of *Software Technology* are: –

- a conceptual model of a tailorable software system that identifies the fundamental elements of such systems and can be used to describe and classify existing approaches to tailorable system design;
- the analysis of an exploratory experiment demonstrating the concrete problems with the current state of the art when employing software components for tailorability *after* application deployment;
- a model of a data space executed in a multithreaded environment permitting the investigation of systems of interacting components which are traversed by threads of control;
- the application of data space theory to model component interaction, in particular to the comparison of different interaction primitives. The comparison concerns the problem of implementing interactions required by arbitrary application decomposition;
- the proof that a combination of events and shared variables permits the implementation of component interaction in a way which obviates the necessity for overly tight synchronization;
- the extension of the JAVABEANS component model with the features of shared variable interaction, named ports, and direct remote interaction – resulting in the FLEXIBEANS component model;
- a novel model for component management after application deployment permitting the sharing of the tailoring operations' effect between several instances of one compositional plan;
- a scope control operation that permits the restriction of the tailoring operations' effect to arbitrary subsets of instances of a compositional plan.

Owing to the extensive evaluation of the EVOLVE platform's use for the implementation of groupware systems, the work also contributes to the field of *Computer Supported Cooperative Work*: –

- a platform which permits the development of highly tailorable groupware systems.
- the basic technology for supporting synchronous and asynchronous cooperative tailoring of complex distributed applications on the internet.

1.6 Dissertation Structure and Reader's Guide

Chapter 2 defines the **fundamental terminology and concepts** used in this work. Readers who are familiar with the notion of components, tailorability and groupware can forego these sections. The chapter also presents the basic conceptual model of a tailorable software system that underlies much of the approach pursued in this work.

Chapter 3 gives an extensive overview of **related work** and of approaches and technologies that are fundamental for the dissertation. In particular, the chapter contains an analysis of the many different perspective on the problem of designing tailorable software systems. Concerning fundamental technologies, the description of the JAVABEANS component model is essential for the understanding of chapters 4, 5 and 7. The description of JAVA RMI is essential for chapter 7 and useful for chapter 8.

Chapter 4 is devoted to the analysis of the **exploratory case study** concerning the search tool in the POLITeam project. The analysis motivates and exemplifies the problem areas described above and culminates in the questions that are addressed in the main body of this work. Consequently, it is essential for the understanding of the following chapters.

Chapter 5 addresses questions concerning the appropriate interaction primitives for implementing **component interaction**. For a reader who is interested in formally verifying properties of a specific component model, the chapter provides a complete description of the mathematical framework employed for comparing arbitrary interaction primitives. The main result presented here is the theorem that constitutes the mathematical justification for the design of the FLEXIBEANS component model presented in chapter 7.

Chapter 6 develops the model for **component management** after deployment. It defines the necessary data structures and algorithms in a mathematical fashion independent of programming language. This chapter is essential for readers who intend to implement a runtime- and tailoring environment, because it defines the relevant concepts with sufficient precision. If the reader is only interested in getting an overview and is familiar with object-oriented programming techniques and UML (Unified Modeling Language), the description of the EVOLVE platform in chapter 8 is probably a better starting point.

Chapter 7 presents the **FLEXIBEANS component model** and builds on the theoretical results presented in chapter 5. However, the results are briefly explained in a non-mathematical fashion, so that an understanding of the theory is not necessary for a reader who simply wants to implement applications with the FLEXIBEANS component model.

Chapter 8 gives an overview of the implementation of the **EVOLVE platform**. Like chapter 7, this chapter is essential for readers who want to implement applications for the EVOLVE platform because it explains how EVOLVE applications can be structured and distributed over a network.

Chapter 9 is devoted to the **evaluation** of the component-based tailorability approach. The description of the implementation of two groupware applications and a non-groupware scenario are intended to permit the reader a judgment of the applicability of the approach. The chapter also summarizes the results of a number of quantitative performance tests.

Finally, chapter 10 summarizes the results of the dissertation and gives an outlook on **future work**.

Chapter 2

Elementary Concepts and Terminology

2.1 Introduction

The purpose of this chapter is the definition of the elementary concepts and terminology. Building on the notion of a *software system* as a computing and interacting entity situated in a specific environment, the concepts *tailorable software system* and *software component* are introduced. These central terms are then related to definitions in the literature. Based on these concepts, the topic of this research – *component-based tailorability* – is defined. Furthermore, the term *groupware system* is introduced, because the motivation and empirical evaluation of this work draw on experiences from this type of software systems.

2.2 Software Systems

A **software system** (here called a system) is something that performs computations and interacts with its **environment**, i.e. other software systems, physical objects, or people. The computations of a software system and its interactions with the environment are mutually dependent. One or several physical or virtual processors execute the software system.

This definition of a software system is influenced by recent work of (Wegner 1997) who proposes that – in addition to *computation* – *interaction* is the other fundamental characteristic activity of software systems. He argues that the traditional abstraction of a software system as a computational entity (e.g. a Turing Machine) does not adequately capture the essence of today's systems. Instead, he claims that the continuous interaction between computational systems (e.g. as supported by the Internet) or between systems and people forms the basis of the benefits derived from these systems. The following uses the widely known example of a (single user) word processor in the definitions' explanations.

A text editor is executed on a microprocessor, directly interacts with the operating (software) system and indirectly – via the operating system and the physical input and output devices attached to the processor – with the human user. Its computations concern text documents, e.g. spell-checking or formatting.

A software system performs a certain **role** within its environment, i.e. its computations and interactions follow a meaningful pattern in the context of that environment. Given a specific environment, the role of a software system induces a set of **requirements** that are specifications of system properties. A **property** is a testable quality of the system's computations and interactions and is invariant for a specific role in a specific environment. If a system exhibits the properties specified for a certain role and environment, it can perform the role in that environment.

General properties of a software system performing the role of a word processor in an office environment are that it permits creating, editing, saving, and printing documents. A more specific property might concern the interaction protocol used when interacting with the office printer. The latter property specializes the more general property of permitting the printing of documents. Similarly, roles can be general or specific. The role *editor* is rather general, while the role *JAVA source code editor* is more specific (and might thus induce requirements concerning more specific properties like, for instance, the coloring of JAVA keywords).

2.3 Tailorability

Today, most software systems are not designed to perform exactly one role in one specific environment. According to (Trigg 1992), the need to support a range of different roles or environments is the consequence of *fluidity* (changes in role and/or environment), *diversity* (deployment of the software system in different roles and/or environments) and *uncertainty* (lack of specific information about role and/or environment).

A software system is **versatile**, if it exhibits a static set of properties that meets the requirements induced by a range of different roles or environments. For example, software systems like word processors – that are targeted at large markets – are often delivered with a cornucopia of functions in order to be useful to every segment of these markets.

However, versatility cannot always be achieved, because the requirements induced by the targeted range might specify conflicting properties. A set of properties is **conflicting**, if a software system cannot exhibit all of them at the same time. A word processor, for instance, usually cannot show its overabundance of functionality at the user interface and at the same time claim the property "easy to use".

A software system is **tailorable**, if its set of properties can be changed. Designing a software system with this meta-property (i.e. a property which "is about" other properties) of tailorability addresses the problem of conflicting system properties.

Again considering the word processor example, tailorability is often applied to the visibility of functionality at the user interface, e.g. in form of button bars which can be changed according to a specific role and environment.

Tailorability also addresses the problem of requirements unknown at design time. Regard the example of a word processor that permits defining keyboard shortcuts or abbreviations for often used lengthy terms. The designers of the word processor cannot anticipate the kind of terms and preferred abbreviations in a specific environment and for a specific role. Thus making the system tailorable can be a solution to the problem.

2.3.1 Definitions of Tailorability in the Literature

The way tailorability is defined depends on one's point of view. In the literature, one can distinguish several perspectives on the problem that are elaborated in chapter 3 ("Related Work and Technologies"). For the purpose of this chapter, two distinct points of view are of interest:

- The user perspective
- The system perspective

The user perspective is characterized by regarding the system as a "black box" with certain properties that make this "box" useful for the user. The central characterizing element of definitions stemming from this point of view is *stability*. A state change of the system is considered to be tailoring, if properties of the system are changed that are stable during regular use. Regard, for instance, the definition of (Henderson and Kyng 1991, p. 223):

"*... we tailor when we change stable aspects of an artifact*."

The definition by (Oberquelle 1994, p. 34 – translated from German by the author) is quite similar:

"[tailorability is the] *... modifiability of aspects of a tool* [...] *that are stable during the execution of regular tasks. These stable aspects are changed as reaction to locally discovered needs during the use of the tool.*"

Both definitions obviously rely on a notion of "regular use" in order to define tailorability via the concept of stability. Since a software system can be used in different ways, these definitions always have to be interpreted relative to a particular fashion of using the system. Consequently, it is not possible to define tailorability as a purely technical concept. (Henderson and Kyng 1991) provide two further characteristics to distinguish between using and tailoring a system: First, they suggest to find out, whether a system's state change concerns the tool itself (e.g. the text editor) or the subject matter (in a text editor this would be the text document). If the subject matter is changed, then the state change can be interpreted as using, if the tool itself is changed, than it is tailoring. Second, they use the point in time when a change impacts upon system-supported tasks as a criterion. If the effect is immediate, then it is using, otherwise it is tailoring.

The system perspective is characterized by regarding the system's environment as a "black box" which poses certain requirements on the system. Tailorability (in this context often referred to as "adaptability") is usually defined as the system's ability to change (or rather: be changed) whenever there are shifts in these outside requirements. Regard, for instance, the definition by (Mens et al. 1996, p. 37):

"*Adaptable systems are systems that can easily be adapted to a steady change of various requirements*."

The system perspective's definition is also not purely technical. The dynamics of the system's environment are acknowledged by including its requirements in the definition.

This dissertation's definition of tailorability (see above) takes into account both perspectives. The environment has certain requirements on a software system performing a specific role in that environment. In order to meet these requirements, the system needs a certain set of properties. Properties are stable (invariant), while the system performs one role in one environment. A system is tailorable, if its set of properties can be changed in order to perform a different role in a different environment. Thus, both the notion of stability of system properties and the notion of system external requirements are included in the definition.

Even though this work's focus lies on technical aspects of tailorability, at this point it is important to acknowledge the multi-faceted – and in particular: not purely technical – nature of tailorability. This point is elaborated in chapter 3. The following conceptual model of the architecture of a tailorable software system serves as a basis for the more technical discussion pursued here.

2.3.2 A Conceptual Model of a Tailorable System's Architecture

From the technical perspective the basic constituents of a tailorable system are of interest. Without subscribing to any particular style of architecture, one can identify three necessary elements of a tailorable software system: The variable properties of the system have to be represented internally as data; there has to be functionality for changing these data; and changes to the data representing a property have to cause the actual property exhibited by the system to change. These three elements: –

- **representation of system properties,**
- **functionality to change the representation,** and
- **connection between representation and system properties**

constitute a basic conceptual model of the architecture of a tailorable system. The tailorability of a system is characterized by the design choices made for the three elements. The following three examples are designed to demonstrate that even in rather different tailorable systems, the three elements can always be identified.

Regard, for instance, the *initialization file* of a word processor that contains a parameter determining the window background color. In this case, the variable system property is the background color, its representation is a numerical parameter value in the initialization file; the manipulation functionality is provided by a simple text editor; and the connection between the representation and the actual property is realized by reading the initialization file during the start-up of the software and coloring the background accordingly.

Another example is an *interpreted scripting language* for automating tasks in a word processor. Here, the variable property is the system behavior when executing a script. The behavior is represented by a text-file adhering to the syntactic conventions of the scripting language. Again, the manipulation functionality is provided by a text editor. The connection between the representation and the behavior is realized by interpreting the script.

The most extensive changes to a system obviously can be made, if the complete *source code* and a compiler are available. In this case, all system properties are variable, because the whole system is represented in the source code and can, for instance, be manipulated in a text editor. The connection between the actual properties and their representation is realized by compiling the source code and re-starting the system, i.e. loading and linking the resulting binary files.

These three examples demonstrate how the tailorability of a system is determined by the design choices made concerning the three elements of the conceptual model:

First, the *representation scheme* obviously determines the extent of the changes that can be made. In the first example, only a single parameter of the system – the background color – can be changed within the permitted range of values. In the second example, the possible space of changes is spanned by the specification of the (word processor specific) scripting language, including every syntactically correct script. In the third example, the space of possible changes includes every system that can be specified in the source code language (e.g. a general purpose language like JAVA).

Second, the choice of *manipulation functionality* determines how and also who controls the changes during the lifetime of a software system. All three examples permit manipulation of the (textual) representation of system properties by humans using a simple text editor. Alternatively, the background color value in the initialization file could be changed via a special-purpose user interface; or a script could be "recorded" by letting the system "watch" the user executing a sequence of tasks (programming by example – see Nardi 1993). In such cases, i.e. if the manipulation functionality provides for direct and intentional changes of the representation by humans, the system is called **user tailorable** or **adaptable**. A person performing tailoring activities is called a **tailor**. If the changes are made automatically and do not (or only indirectly) depend on human actions, the system is called **adaptive**. A simple example of adaptivity is a variable menu that contains the last documents edited in a word processor. This feature permits quick access to work in progress, without having to search for the relevant documents. If a new document is edited, the system automatically changes the menu items accordingly, i.e. the new document is added to the menu and the oldest document is removed. While these changes depend on user actions (loading and editing a document), they do not have to be explicitly intended and executed by the user. More complex adaptive systems maintain models of their changing environment and attempt to predict and implement the specific properties required. The distinction between user tailorable and adaptive systems can be gradual, if only parts or distinct phases of the tailoring activity are automated. (Kühme et al. 1992) distinguish the four phases *initiative*, *suggestion*, *decision* and *execution* of a system change and classify the manipulation functionality by determining which phases are automated and which are manual.

Third, the choice of the *connection between representation and actual system properties* determines the point in time when the changes become effective in the system. In the first example the initialization file is evaluated at start-up time. For changes to the background color parameter to become effective, the system has to be shut down and started up again. Changes to the script in the second example become effective when the changed part of the script is interpreted the next time. The system does not have to be shut down. In order for changes to the source code in the third example to become effective, one first has to shut down, re-compile and then re-load the whole system. Obviously, the choice made for this element of a tailoring architecture has a large influence on how disruptive system changes are for the operation of the system.

The notions of *representation*, *manipulation functionality,* and *connection* between the representation and properties will be used as a basic framework for description and comparison in this dissertation.

2.4 Software Components

Collins Cobuild Dictionary defines a component as "*...one of the parts or features from which something is created, made or produced.*" (Collins 1987, p. 285). Consequently, on the most general level, a **software component** (here called a component) is a part or feature of a software system. Due to the non-physical nature of software systems, one has to further distinguish between the plan of a part or feature and its instance in the running (executed) system (compare Szyperski 1998). A component can be instantiated several times within the running system. In contrast to a component (plan), a component instance can have a state.

Here, if the distinction is irrelevant or obvious from the context, component instances are simply referred to as components.

(Teege 2000) further distinguishes components as parts and components as features. Part instances are explicitly connected to other part instances in order to form a software system. Features are not explicitly connected but added – in any order – to a set of features that together make up a software system. In this dissertation components are understood as parts.

A **composition** is a plan that describes a network of connected component instances and their initial states (parameterization). A composition can again be regarded as a component. Thus the notion of composition is hierarchical. A language in which a composition can be expressed is called a **composition language**. A **component-based software system** is a software system that can be represented by a composition. A composition specifying only one instance that is not the composition of other instances is an **atomic component**. A non-trivial composition is called a **complex component**. An atomic component is implemented in a programming language (e.g. JAVA) and is deployed in binary, i.e. compiled and machine-readable form. It offers facilities: –

- to be instantiated (if necessary more than once);
- to remove instances (to be "de-instantiated");
- to tailor single component instances by parameterization (i.e. changing their initial state);
- to reveal its specific capabilities to interact with other component instances and its parameters;
- to be connected to other component instances;
- to be disconnected.

The ways in which component instances can interact and how to implement the above facilities in a particular programming language (or in some cases only on the binary level) are specified by a **component model**. The JAVABEANS component model, for instance, prescribes how to implement components in the JAVA programming language. (Since the prototypical implementation of the concepts developed in this work was performed in JAVA, the following uses the JAVABEANS component model as example. However, the definitions also hold for other components models as presented in chapter 3.)

Components can only participate in an **interaction**, i.e. a dependency or influence on each other's state, if they are connected in a composition. An interaction follows a protocol that specifies the possible sequences of actions executed by the components taking part in the interaction. A **port** is a part of a component and indicates the component's ability to participate in an interaction. The **port type** determines the protocol of the interaction and the **port role** indicates the part played by the component in this interaction. If there are only two different port roles required by an interaction, the port role is also called **port polarity**. Two components can be connected in order to participate in a specific (two role) interaction, only if each has a port of the port type given by the interaction and the two ports' polarities differ. An interaction is called **symmetric**, if all required port roles are the same; otherwise it is called **asymmetric**.

The JAVABEANS component model permits component interaction via an event protocol. JAVA events originate from a component playing the part of *source*. They are consumed by a component playing the part of *listener*. In the terminology of this dissertation, the event source component has a port of the port type *event* (which in JAVA is further refined by the type of the event object which is passed from event source to event listener) and of the port role (or polarity) *source*. The event listener component has a port the port type *event* and the port role (or polarity) *listener*. The event protocol specifies an asymmetric interaction. In chapter 3 "Related Work" the JAVABEANS component model is described in more detail.

A **port name** distinguishes a port from other ports of the same component that have the same port type and play the same port role. The JAVABEANS component model does not

support the notion of port names. Thus a JAVABEANS component cannot have two ports listening for (or sending) events of the same type.

Throughout this dissertation a graphical representation of components and compositions will be used. It is similar to the graphical representation of the DARWIN composition language (see Magee et al. 1995). Figure 2.1 shows the basic elements of the representation:

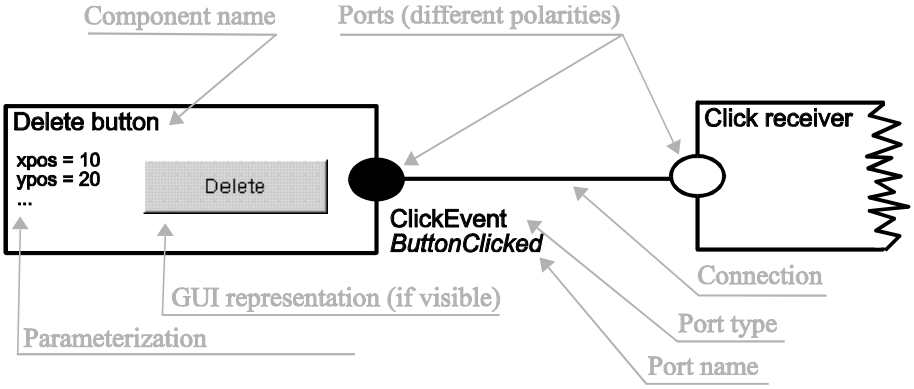


Figure 2.1: Graphical representation of components and composition

A component is represented by a rectangle. Ports are represented by filled or empty circles denoting the polarity of the port (filled = source, empty = listener). Connections between components are drawn as lines between the ports taking part in the interaction. If relevant, the parameterization of the component as an instance is shown within the rectangle.

2.4.1 Definitions of Software Components in the Literature

The definition of software component given above – as *a part or feature of a software system* – is rather general. However, this generality is necessary at this point, because most other definitions in the literature are too restraining by explicitly stating what component are to be used for – that is for software development and not for tailoring after development as intended here. Regard, for instance the definition by (Jacobsen et al. 1992) in his book *Object-Oriented Software Engineering*:

"*By components we mean already implemented units that we use to enhance the programming language constructs. These are used during programming and correspond to the components in the building industry*."

Jacobsen also relates the notion of software components to components in another non-software industry. Other definitions imply the ways in which components are to be used, as well. (Szyperski 1998) writes:

"*A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties*."

By referring to "*third parties*", Szyperski takes into account the existence of component markets, which are an important phenomenon when using software components for programming. Both definitions cover the notion of components in the process of component-based software development.

However, for the purpose of this work, a more technically focused definition is needed which abstracts from the ways in which components are traditionally used. The definition by (Nierstrasz and Tsichritzis 1995) probably comes closest to how component are understood here:

"*A software component is a static abstraction with plugs*."

This definition only refers to technical aspects of components. The notion of plugs is intended to mean the same compositional mechanism as the ports defined above.

2.5 Component-Based Tailorability

A software system has the (meta-) property of **component-based tailorability**, if the representation element of its tailoring architecture (as defined in the conceptual model) is a description of its composition. Consequently, an approach to designing software systems with this property includes the following elements: –

- a component model;
- a composition language;
- functionality for manipulating the composition;
- a realization of the connection between the representation of composition in the composition language and the actual instantiated composition which makes up the running software system.

Component-based tailorability is the topic of this work. However, its motivation (chapter 4) stems from experiences made in the area of groupware design and its empirical evaluation (chapter 9) consists of the implementation of a number of groupware systems. Therefore, the following section defines groupware and gives a brief overview of the types of functionalities that are considered to be characteristic for these systems.

2.6 Groupware Systems

A **groupware system** is a software system which is deployed in an environment consisting of a group of people and which performs the general role of supporting the cooperation within this group. People are **cooperating**, if they work together to achieve a common goal.

The probably best-known and most successful example of a groupware system is email. An email system supports asynchronous text-based communication between group members – a functionality that is useful in a wide variety of environments.

The list of further examples for groupware systems in table 2.1 demonstrates that a wide variety of systems can be employed to support cooperation. Their only commonality lies in the fact that they are designed to perform the rather general role of cooperation support in a group environment. Many commercial systems integrate several of these basic group support functionalities. LOTUS NOTES (see Dierker and Sander 1997), for instance, includes email, group scheduling, shared workspaces and basic workflow functionality – similar to DEC's LINKWORKS (see DEC 1997) or MICROSOFT EXCHANGE (see Microsoft 1998).

Example	Description of cooperation support functionality
Teleconferencing systems	Teleconferencing systems provide synchronous media-streams (usually audio and video) between group members at remote locations.
Chat systems	Chat systems support synchronous text-based communication. The communication takes the form a dialogue to which all participants can contribute.
Shared workspace systems	Shared workspace systems permit the asynchronous joint use of work artifacts like text documents or spreadsheets. These systems are often enhanced with mechanisms for access control, promoting awareness of other user's activities, or searching within the workspace.
Workflow systems	Workflow systems support structured asynchronous cooperative work by coordinating the information and document flow within a group according to a system's internal representation of the group's work processes.
Group decision support systems	These systems support certain phases in group decision-making processes. They help to collect (sometimes anonymously) contributions to a discussion, to organize and categorize them and finally they support voting procedures.
Group scheduling systems	A group scheduling system permits the sharing of individual and joint calendars in order to coordinate meetings and allocate resources like conference rooms and equipment.

Table 2.1: Examples of basic groupware functionality (taken from Ellis et al. 1991)

The functionalities provided by today's commercial groupware systems have been shaped over a number of successive system versions. The scientific community in parallel still struggles to understand and predict the specific role of such systems in group environments. A rigorous scientific treatment of groupware systems is rather difficult, because of the diverse natures of the two central elements of the research topic: computers and groups of human beings. On one side, computers can be described and predicted quite accurately with the help of formal methods – on the other side, aspects of groups which appear to influence the way its members cooperate are quite elusive and too complex and unpredictable to be expressed in useful models. Consequently, it is almost impossible to predict exactly what will happen, if a groupware system is introduced into a specific group environment. How will the system be used? Will it be used at all? Will it change the style in which the group members cooperate? What will be the exact role of the system in that cooperation?

The field of CSCW (Computer Supported Cooperative Work) has taken up the challenge of identifying and addressing the questions raised by the design and introduction of groupware (for an overview see e.g. Ellis et al. 1991). Due to the above-mentioned diverse nature of the research subject, the field is highly interdisciplinary with contributions – apart from computer science – from sociology, psychology, anthropology, ethnography and several other disciplines. A lot of work is empirical, studying groupware systems in use and attempting to elicit requirements for the design of such systems. While the field is still missing a comprehensive theoretical base, there are a number of themes emerging from the large body of empirical work.

Prominent among these themes is the importance of tailorability as a property of groupware, which is observed by several authors (Greenberg 1991, Malone et al. 1995, Bentley and Dourish 1995, Simone and Schmidt 1998). The importance of tailorability is attributed to the dynamics and diversity of cooperative work. Furthermore, the above-mentioned uncertainty about the exact role a groupware system will play in a specific group environment adds to the need to design for tailorability. Consequently, the field of CSCW has produced quite a variety of research prototypes of tailorable groupware systems that are reviewed in chapter 3.

The empirical parts of this work are rooted in the field of CSCW. In the initial case study (chapter 4), a part of a groupware system was made tailorable using the concept of component-based tailorability. The implementation of the EVOLVE runtime and tailoring environment (chapter 8) was oriented at the requirements raised by groupware systems distributed over the Internet.

Chapter 3

Related Work and Technologies

3.1 Introduction

This chapter presents work related to component-based tailorability and technologies necessary to implement the approach. The first section introduces tailorability and gives an overview of the multifaceted discussion of this topic in different fields of research beyond the technical scope of this work. The second section discusses general technical foundations of tailorability. Then the third section introduces the central concept of the approach developed in this work: software components. After revisiting its origins and discussing its constituent ideas, a number of current technologies that claim the label "component" are discussed and related (component models, distributed component interaction, heterogeneous component interaction). Finally, the fourth section focuses on tailorability of groupware and presents a number of current research and commercial groupware systems employing different techniques to achieve tailorability.

3.2 Tailorability

The last chapter defined tailorability as a meta-property of a software system, which permits the system to play different roles in different environments. Not surprisingly, text editors were prominent among the first applications with this property, due to their use in a wide variety of environments and roles. As early as 1981, (Stallmann 1981) describes the design of the EMACS text editor that already incorporates a number of tailorability features. While the base system is implemented in C, the higher functions of the editor are programmed in EMACS Lisp. Existing functions can be replaced and new functions can be added and bound to keys. Additionally, a number of these functions can be parameterized (persistently in initialization files, permitting minor adaptations of the editor by non-programmers) as depicted in figure 3.1:

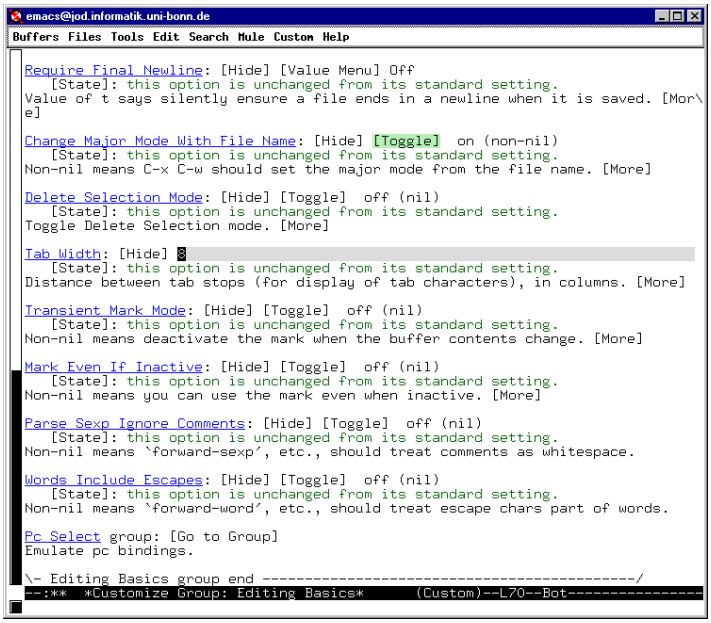


Figure 3.1: Parameterization of the EMACS text editor

The Lisp system inside EMACS is an interpreter. This realization of the *connection between the system properties and their representation* (as Lisp code) makes it possible to effectively change and add code during runtime. The fact that almost all higher functions (e.g. for basic text editing) are accessible in Lisp code provides for extensive tailorability of the system. However, changing Lisp code is a viable option primarily for programmers. The parameterization of the functions (which Stallman calls “customization”) can be regarded as a second level of tailorability, which is easier to handle, but less powerful. The parameters can be arranged in conceptually coherent groups (so called “customization buffers”) which – in addition to providing access to the parameters – can also include further explanations of the sometimes quite cryptic parameter names (see figure 3.1).

The EMACS example indicates that designing tailorable software systems is a multifaceted problem, which poses a number of questions for the software developer and requires contributions from a number of different fields within computer science (see Stiemerling et al. 1997).

First of all, the developer has to know, resp. anticipate *which properties of the system are likely to be subject to diverse or dynamic requirements*. In a text editor like EMACS, for instance, an important area for individualization is the way a user controls the application, e.g. the key-bindings to certain functions. Different users might have different prior experiences or predilections concerning this point. Eliciting and documenting requirements is the focus of the field **Requirements Engineering**. Subsection 3.2.1 discusses the contributions of this field to the problem of tailorable system design.

Second, a *suitable software architecture* has to be found which represents the tailorable properties as data and connects this representation to the actually exhibited properties of the running system. In a text editor like EMACS this might be achieved simply by storing key-bindings to functions in a table that is interpreted during user interaction, i.e. whenever the user presses a key or combination of keys. The field of **Software Technology** investigates the basic technologies for building software – always aiming at certain "desirable" engineering properties like robustness, correctness or even elegance. Since software technology is the focus of this work, the contributions from this field are dealt with more elaborately in the next section (3.3).

Third, the developer has to decide how the tailorability provided by the architecture is to be controlled, i.e. he has to determine the *manipulation functionality*. This decision requires knowledge about when, by whom and under which circumstances the system will be tailored. The table of key-bindings in a text editor might be stored in a text file, thus permitting manipulation by a user with the editor itself. Alternatively, the user could be able to select from a list of whole alternative tables (e.g. "*MAC-like bindings*", "*PC-like bindings*", "*Unix-like bindings*"), perhaps in form of a menu. More technically inclined users might favor the text file alternative, while the list selection alternative is probably more suitable for beginning users. The field **Human Computer Interaction** deals with the design of interfaces between man and machine. Subsection 3.2.2 gives an overview of the discussion concerning interfaces for tailorability in this field.

Fourth, in many cases the developer has to take into account *the cooperative nature of tailoring activities*, with more experienced users (also system administrators or third party consultants) tailoring the system for and with the actual end users (see Mackay 1990). Again regarding the text editor example, it might be a useful feature to permit more experienced users to supply beginning users with a pre-tailored version of the key-binding table. It could appear in the list of alternative tables for later easy selection by the beginning user. In the field **Computer Supported Cooperative Work** (compare chapter 2) the cooperative nature of tailoring has been discussed for some time now. Subsection 3.2.3 presents a few (technical as well as non-technical) approaches for supporting tailoring as a cooperative activity.

Even though aspects of tailorability surface in different fields of research, they are nevertheless interdependent. For example, the user interface presentation of the (system internal) representation of the tailorable properties to the tailor is certainly constrained and influenced by the format of that internal representation. Indeed, at the heart of tailorable system design lies the task of building the system in a way that a (well-chosen) part of the implementation can be opened up and presented to the tailor (or another software system) for manipulation. Thus design for tailorability on a technical level is essentially the creation of a manipulatable, partial view on the system's implementation.

The following subsections aim at giving the reader an impression of the manifold questions and challenges posed by tailorability and serve to position the technical work presented here within the whole picture.

3.2.1 The Process Perspective: Design Methodologies for Tailorable Systems

The field of Requirements Engineering provides techniques for requirements elicitation and documentation. Requirements elicitation is essentially a communication process between designers and users. It can vary considerably in terms of duration and effort depending on the type of software system to be developed, its environment and obviously project resources. On one end of the spectrum, elicitation techniques can take the form of long-term ethnographic studies of application domains which involve designers closely observing or even taking part in the everyday work of the end users (Grønbæk et al. 1995). Since many commercial development project lack time and funding for such studies, (Hughes et al. 1994) discuss the use of "quick-and-dirty" studies in form of short-term work place visits,

observations and workshops. Less involved methods include short end user interviews or questionnaires.

Rapid prototyping approaches (see e.g. Greenbaum and Kyng 1991) employ a cyclical or iterative project organization with several intertwined phases of development and user evaluation. The end users' "hands-on" experience with early versions of the software system is supposed to facilitate the dialogue between designers and end users, which is necessary for requirements elicitation. The choice of elicitation technique for a specific project depends on a number of factors (e.g. type of organization, schedules of end users, etc.) and there is certainly no one best way to do this.

This also holds for requirements documentation. There are a number of documentation techniques with different levels of detail and biased towards different types of requirements. A well-known approach is the use-case methodology (Jacobsen et al. 1992), revolving around the use-case construct that is a script-like documentation of user-system interaction in a specific situation. Use-cases are employed to document system requirements from a user perspective. Usually other documentation techniques like domain object models (see Rumbaugh et al. 1991) or state-charts (see Harel 1988) are used in conjunction with use cases in order to capture other relevant requirements. Over the last years, a number of modeling techniques have been combined and standardized in the UML (Unified Modeling Language, see e.g. Fowler and Scott 1997).

3.2.1.1 Documenting Diverse and Dynamic Requirements

These techniques for elicitation and documentation (and many more which have not been mentioned here) are already applied – supposedly successfully – in industrial practice. However, the question remains, how they support the elicitation and documentation of *diverse and dynamic requirements*, which is essential for the design of tailorable software systems. The participants of the ECOOP '96 workshop on adaptable software came to the following conclusion (Tekinerdogan and Aksit 1996, p.10):

> "*The conventional design processes are problematic because they are based on subsequent elimination of alternatives and early commitments which reduce the adaptability.*"

This bias in existing requirements engineering techniques surfaces both in the documentation and elicitation methods. A review of the current literature discloses the almost complete lack of work on methods taking into account diverse and dynamic requirements. In the following, a sole exception is discussed.

(Ecklund et al. 1996) introduce the *change case* extension of the above-mentioned use-case methodology. Change cases introduce the notion of possible requirement changes as explicit part of the requirements model. According to the schema depicted in figure 3.2, a *change* entity connects certain *use cases* (which describe the requirements before the change) with *change cases* (which are specialized use cases describing the requirements after the change):

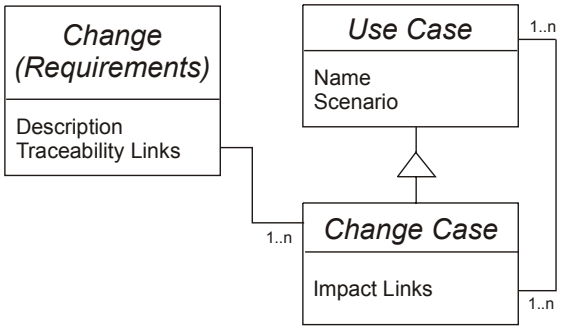


Figure 3.2: Change cases (Ecklund et al. 1996, p. 344)

While change cases appear to be a possible documentation technique for dynamic (and also diverse) requirements, the authors employ their extended requirements model to investigate

the notion of *robustness* for a fixed object-oriented system design, i.e. they are interested in the ratio of the number of changes in the requirements model to the number of necessary subsequent changes in the object-oriented design of the final application. They do not discuss the use of their model to support the design of a tailorable system. However, the change case method is the only approach mentioned in the literature that explicitly includes the notion of change in the requirements model.

3.2.1.2 Eliciting Diverse and Dynamic Requirements

(Stiemerling et al. 1997) discuss the use of participatory design techniques (also see e.g. Grønbæk et al. 1995) for eliciting diverse and dynamic requirements. They suggest to guide the selection of end users for participation in requirements elicitation (e.g. in form of interviews and workshops) according to presumed differences in their requirements for the software to be developed. For example, according to (Grudin 1994) such differences are often found between subordinates and managers. Differences in requirements can also stem from different work tasks, personal predilections, etc. The authors propose that by letting many different stakeholders participate in the requirements elicitation process, differences in requirements come into focus and can either be resolved or explicitly taken into account by means of a tailorable system design.

The approaches presented above can only be a first step on the way to a design methodology that explicitly takes into account design for tailorability during each step of the process. Concerning this point there exists a gap in the state-of-the-art that requires future work.

3.2.2 The Human Perspective: Tailoring For and By the Users

The activity of tailoring is characterized by the fact that the system is changed not in the context of development but during the use of the system (see Henderson and Kyng 1991). This implies that the tailors are not the original developers and probably not even programmers at all. Ideally, the people who know most about the environment and the work the system is supposed to support should be able to define and adapt its structure and behavior. The field of *Human Computer Interaction* has taken up the challenge of designing interfaces that permit normal users (and non-developers in general) to tailor. This section gives a brief overview of the questions addressed and concepts developed in this area.

3.2.2.1 Adaptability or Adaptivity?

One of the most fundamental questions in this area – which has been the subject of intense controversy – is the question whether it makes sense to *automate* the activity of tailoring (a current offspring of this discussion can be found in Shneiderman and Maes 1997).

Automation of system adaptation (adaptivity) promises the advantage that end users can reap all the benefits of a tailorable system without actually having to spend the time to learn a tailoring interface and then to tailor. However, the other side – in favor of user tailoring – argues that it is already difficult enough to predict which aspects of a system are supposed to be variable (see preceding section about design methodologies) and that it is therefore impractical to build a system which *additionally* decides in its own, *in which situation which alternative for such a variable aspect is appropriate*. Furthermore, they argue that an interactive system which changes itself – however appropriate this change might be – would confuse the user who gains the impression of losing control over the application and has difficulties in predicting what the system will do in a certain situation (for an overview of the pros and cons see e.g. Friedrich 1990).

While the discussion is often conducted in a rather polarizing fashion, one can also approach the question in the more analytical fashion of (Kühme et al. 1992). As already indicated in chapter 2, (Kühme et al. 1992) open up the design space between the two extremes *adaptivity* and *adaptability* by dividing the activity of tailoring in four steps: *initiative*, *suggestion*, *decision* and *execution* which are alternatively placed within the purview of the user or the system. *Initiative* means the discovery of the need for tailoring the system. If this step is automated, the system might monitor the user and – if it, for instance, discovers that

the users often uses a functionality which is "hidden" behind a five mouse-click dialogue sequence – initiates the tailoring process. In the next (automated) step the system generates a number of *suggestions*, e.g. to define keyboard short cuts to the functionality. Now, assuming the *decision* step is not automated, the user chooses his or her preferred short cut and then the system takes over again to execute the change by *implementing* the chosen short-cut in the system.

The decision to automate parts of the tailoring process is strongly influenced by the practicalities and impracticalities of the nature of the tailoring problem at hand. If one can define a clear relationship between (machine detectable) user-system interaction patterns and tailoring options then adaptivity is a possibility. Otherwise, one cannot be sure that the result of the tailoring will be consistent with the needs and preferences of the user.

3.2.2.2 Presenting the "World Under the Hood" to Users

Involving the end user in the tailoring process has the obvious consequence that at least parts of a system's implementation have to be opened up and presented – in manipulatable form – to the user (compare: open implementation, Kiczales 1996). At this point the fields of Software Technology (ST) and Human Computer Interaction (HCI) touch. The challenge for HCI is to find an appropriate interface presentation of the essential concepts of a certain implementation architecture, the choice of which in turn forces ST to take into account the intrinsic cognitive complexity of certain architectural concepts. While a tailoring "architecture" like the INI-file is essentially based on simple selection from a limited set of anticipated alternatives (Henderson and Kyng 1991), the full option space provided by a full-fledged interpreted imperative programming language is hard to present in a form which is conceptually easier to grasp than by simply permitting the user to edit a text file containing the code. The user still has to understand the concept of, for instance, a while loop or an if-then statement to realize the whole potential of the language. Nevertheless, it is always possible to design an interface that is easier to use by limiting the range of tailorability provided by the underlying architecture. This limitation, however, is obviously not in the interest of the developer of the system, who has chosen a particular system architecture in order to provide the appropriate degree of system tailorability.

Consequently, many HCI efforts focus on making the interface presentation of specific system implementation concepts palatable for non-programmers (often with the nice side-effect of making life easier for programmers as well). (Nardi 1993) discusses several approaches to what she calls "end user programming". Among those are: –

- *Form-Based Interfaces* (similar to the EMACS parameterization interface shown in figure 3.1. Also compare Stallmann 1981) for architectures that provide the flexibility to chose between anticipated alternatives. These types of tailoring interfaces are found in most commercial applications (like word processor or web-browsers) as menu items like "preferences", "options", etc.
- *Macro-Recorders,* which permit end users to "program" macros for automating certain repetitive tasks by simply letting the recorder watch there actions in one instance and record them in form of a macro. The underlying architecture obviously has to permit access its functionality via a scripting language (see also Cypher 1993).
- *Visual Programming Languages,* which some regard as one way to permit even non-programmers to realize the full power of a complete programming language (see also Myers 1990). The multi-dimensional representation is supposed to give the "programmer" a clearer picture of the relationships within program code. A metaphorical representation of program elements and semantics (e.g. visualizing statements as jigsaw-pieces which fit together only in certain ways) is assumed to permit an easier conceptual access to the programming language.

Other work investigates interfaces for systems whose behavior is partially defined by a rule base. Typically such architectures are appropriate for environments in which the intended system behavior can be expressed in terms of "if-then" statements:

(Terveen and Murray 1996), for instance, describe the design of a user programmable software agent whose job it is to process emails according to the habits and preferences of its "master". The user can formulate email filtering rules which the agent checks for inconsistencies in relation to other rules and suggests changes (compare Kühme et al. 1992) to remedy these inconsistencies.

(Stiemerling 1996) develops and applies the idea of rule-based tailorability to the problem of formulating and changing complex access control policies in the context of groupware systems. In contrast to (Terveen and Murray 1996), here the system lets the user check the rule base by stating hypothetical access situations like "What happens if my boss attempts to read my private electronic diary?" which the system then evaluates.

All the approaches presented here have in common that they embrace a technical principle or paradigm of system implementation and attempt to present it in an appropriate form to the tailor. It is obvious that due to the nature of the problem of designing tailorable system – i.e. opening up part of a system's implementation – it is not possible to completely hide implementation details from the user. Consequently, even when approaching the problem from the technical side – as this research does – one cannot ignore requirements stemming from the more user-oriented perspective. The issue discussed in the following section is an example for such a requirement.

3.2.2.3 Multiple Tailoring Interfaces: Different Strokes for Different Folks

Many of the techniques presented above work well for certain groups of users and tailoring problems. However, quite often a system has to be tailorable in more ways than one architecture and tailoring interface can provide and also by more than one type of user. Consequently, there are efforts that focus on developing architectures and interfaces that support a large number of tailoring techniques.

(MacLean et al. 1990) present the BUTTONS system that revolves – not surprisingly – around the concept of buttons that encapsulate a certain functionality. The behavior of the button is implemented in LISP and can be controlled by a number of parameters that – apart from the behavior – also define the appearance and the label of a button. Furthermore, a button can be moved around the screen, copied and sent via email. MacLean et al. argue that based on these properties the user can employ a large number of tailoring techniques – nine – that can be ordered according to the "*skill required for tailoring*" (p. 181). They arrive at this "*large number*" of techniques by declaring modifying LISP Code and programming in LISP Code as two distinct difficulty levels and by differentiating between changing the appearance parameters and changing generic behavioral parameters. These distinctions neither concern the underlying architecture nor the interface for tailoring, but solely the way the one interface for accessing LISP code and the one interface for changing parameters are used.

(Bentley and Dourish 1995) have coined the phase "*customization gulf*" (p. 139) for the steep incline in skill necessary for performing tailoring activities that go beyond changing parameters. When shifting to a different level of tailoring, a user usually has to learn a completely new language for tailoring (e.g. from parameter editing to Lisp programming, as in the BUTTONS system). They identify this shift in level and language as a major barrier for users wanting to tailor their system. They identify as a serious challenge the design of a system that permits a beginning user to move from a lower (and thus less powerful) level of tailoring to the next higher level with only moderate learning efforts.

Consequently, a system architecture is desirable, which supports the user to view and manipulate the system at many different levels of complexity without having to learn a completely new language. Especially the increase of necessary skill between the levels should be as gradual as possible.

3.2.2.4 Other Issues and Approaches

(Mackay 1991) empirically investigates triggers and barriers for users to tailor software. Especially the barriers are of interest for the HCI community, as they obviously indicate the areas where the design of the tailorable system has to be improved. While the reason cited

most often for not tailoring the system was "*lack of time*", other causes appear to be less difficult to rectify. First of all, many users claimed that the system was "*too hard to modify*". The last two subsections discussed approaches for making tailoring easier for non-programmers. Secondly, the users complained about "*poor documentation*" and the fear to break something in the system.

(Mørch 1997) has suggested including into the tailorable system an explicit representation of the *design rational* (the *why* of the initial design and already applied tailoring actions) in order to enhance the user's understanding of the system design and his options for tailoring it. Mørch uses the example of a drawing software package that is tailored to support the drawing of kitchen layouts. The changes concern for instance the scale-command (change size) for rectangles that is adapted to the restricted layout requirements of the "kitchen-planning domain". Kitchen appliances mainly have a fixed depth (56 cm) and limited width (20-100 cm). The scale-command is adapted to respect these restrictions. The rational that is added during the tailoring activity explains these reasons and helps the user to understand the domain background of these design decisions that would be rather hard to deduct from the adapted code in the programming language.

The user's fear to break something in the system has inspired the idea to provide him with a safety net for the case that the changes do not have the desired effect. The *undo-command* present in many systems can serve this purpose. In groupware systems (and multi-user applications in general) one faces the additional problem that the effects of a change might not even be visible to the tailoring user. Thus it is necessary to permit a user to safely explore the effects of a change without actually impacting upon the work of others. The above-mentioned hypothetical access situations described by (Stiemerling 1996) can serve this purpose.

3.2.3 The Group Perspective: Tailoring as a Collaborative Activity

Tailoring is not only investigated as a single user activity performed by non-programmers as described in the last section; there is also work identifying tailoring as a collaborative activity, involving a number of people in an organization, from the actual end-users to the system administrator, or even outside consultants. (Mackay 1990) for instance describes how users exchange customization files for desktop environments, with more experienced users sharing their successful adaptations with beginners. While this work draws part of its motivation from the observed need of tailorability for groupware, the position taken in this section reverses that theme and gives an overview of the collaborative aspects of tailoring, including approaches for technically supporting joint tailoring, or groupware for tailorability.

3.2.3.1 Organizational Support for Tailoring as a Collaborative Activity

(Carter and Henderson 1990) observe that in the context of an organization, in order for tailoring activities to succeed, the (user) task of "*changing the system*" has to be acknowledged and even actively supported by the organization. End users have to adopt the idea that "making the system work for them" is part of their job and cannot be fully delegated to the system support personnel. Sometimes, the users even have to be made aware of that it is possible to change the system at all and that they are allowed to do so by themselves. Tailoring is supposed to be regarded as "*something good*". Carter and Henderson – based on their experience in the BUTTONS project at Xerox PARC – propose the concept of a *tailoring culture* as prerequisite for tailoring in an organization. The notion of tailoring culture encompasses all the non-technical aspects that aim at supporting the change of an organization's technical systems.

An important part of a tailoring culture are people within the organization who support the other users in learning how to tailor (e.g. via tutorials), directly support tailoring activities and also perform certain more complex tailoring tasks for the end users. However, the primary aim of these "*handymen*" or "*transactors*" is to involve the users in the ongoing design process of the system and to change their attitude towards the system. The authors report the following (p. 109):

> "*We have seen that a large part of the success of the BUTTONS project arose from a change in the users' culture. They moved from being passive and feeling unable to control their electronic environment to being active managers of it. They viewed their workstations, and the buttons in particular, as being changeable objects.*"

(Gantt and Nardi 1992) have investigated the collaboration between CAD users who tailor and have come to similar conclusion as (Carter and Henderson 1990). In their terminology, the people who support and encourage tailoring are "*gardeners*", "*gurus*" or simply "*local developers*". In contrast to (Carter and Henderson 1990), they state the need for these people to be genuine domain experts and not necessarily part of the development team or support staff. (Gantt and Nardi 1992) also report, that in some departments of the organization they investigated, the local developers were given official recognition and also resources to pursue their activities.

3.2.3.2 Technical Support for Tailoring as a Collaborative Activity

The issues discussed above concern only the organizational aspects of supporting tailoring as a collaborative activity, especially recognition of the role of local supporters. Since many collaborative tailoring activities are based on *sharing* of adaptations (e.g. in form of customization files, see Mackay 1990), there are a number of requirements the technical system has to meet in order to support the exchange of the results of these activities.

First, it has to be possible to isolate specific adaptation from the rest of the system and bring them in storable and/or shareable form. (Henderson and Kyng 1991, p. 232) call this "*objectification of changes*". A configuration- or INI-file is a well-known form of objectification. It can be stored in the operation system's file system, moved and copied.

Second, there has to be a means to transfer the change from one instance of the system to another. In the BUTTONS system described by (MacLean et al. 1990) a button can be attached to an email message or shared via a common folder in the operation system. Naturally, the same holds for INI-files.

Third, ends users receiving adaptations have to have the opportunity to understand how a specific change is intended and what it is good for. One approach to achieving this is given by the integration of the design rational (see previous subsection and Mørch 1997).

(Kahler et al. 1999) present a prototypical system that permits collaborative tailoring of a text editor (MICROSOFT WORD). The adaptations supported by the prototype concern the button bars that are used to access the functionality of the editor. They can be saved, annotated (with a name and a description) and sent to specific groups of other users. These can preview the button bars and then decide whether they want to integrate the adaptation in their own system or not.

3.3 Software Technical Foundations for Tailorability

This section describes a number of software technologies that are fundamental for tailorability. The section assumes a basic knowledge of object-oriented concepts (for an overview see e.g. Nierstrasz 1989).

3.3.1 Meta- and Reflective Systems

In chapter 2, tailorability was characterized as a *meta-property* of a software system, i.e. a property that "is about" other properties. Software systems that are – at least partially – about other software systems are often called *meta-system.* Systems that are about themselves are called *reflective systems* (see e.g. Smith 1982, Maes 1987, Ferber 1989, Kiczales et al. 1991). This section relates these concepts to the problem of building tailorable software systems.

3.3.1.1 Meta-Systems, Causal Connection and Reflection

A *meta-system* is a software system whose data represent aspects of another software system; the meta-system "is about" the other system, which is also called its *object-system*. Regard, for instance, an interpreter whose data concern the code, data and execution state of the interpreted object-system. During interpretation, the computations of the meta-system emulate the computations of the object-system. Another example for a meta-system is a help-system, containing a – human readable – description of its object-system which it can display in various forms and which it perhaps even permits the user to change, e.g. by annotation. Figure 3.3 depicts the relation between a meta-system and its object-system:

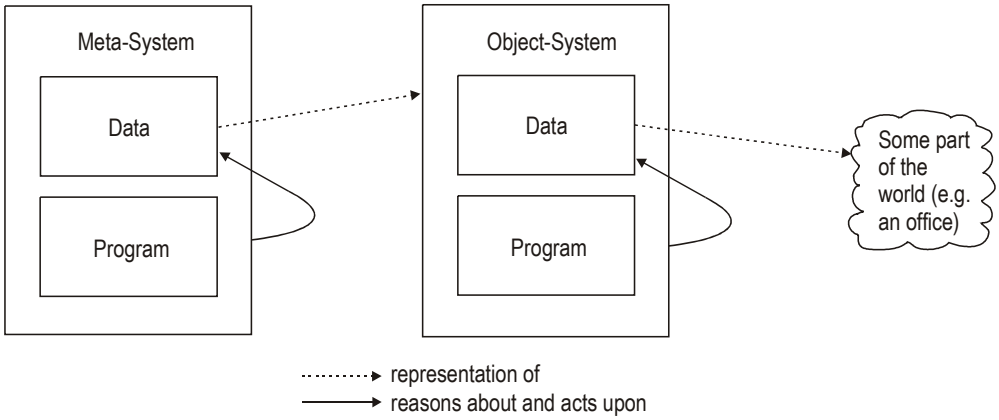


Figure 3.3: Meta-system

A meta-system can reason about and act upon the data representing its object-system. However, there is a fundamental difference between the interpreter example and the help-system example. In the former, changes to the data (e.g. the location of a program pointer of the object-system) impact upon the object-system. In the latter, if the user annotates the description of the object-system, i.e. if he changes its representation, this does not impact upon the object-system itself. Concerning this difference, one says that the interpreter is *causally connected* to its object-system, while the help-system is not. Thus the causal connection relation implies that if the meta-system changes the data representing aspects of its object-system, these changes actually "cause" the object-system to change, as well. Causal connection also implies that if for some reason the object-system changes, the data representing the object-system changes accordingly. In the interpreter example this direction of the causal connection relation is not relevant, because the interpreter (the meta-system) is – by virtue of interpreting the source code – the only entity directly influencing the object-system. However, for a tool that visualizes the runtime behavior of a software system, this other direction of the causal connection relation plays the primary role. Figure 3.4 depicts the idea of a meta-system that is causally connected with its object-system:

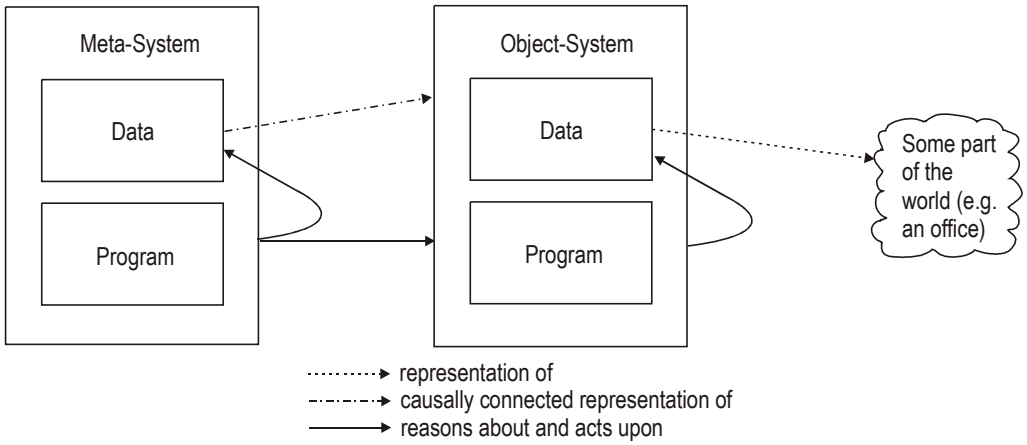


Figure 3.4: Causally connected meta-system

By being causally connected to its object-system, the meta-system is able to reason and act not only upon the data representing the object-system but also directly about the object-system itself. (Arguably, assuming a faithful representation of the object-system as data, a non-causally connected meta-system can also reason about its object-system, but it relies on the user to keep the representation and the state of the object-system "in synch". Compare (Maes 1987).

The special case of a causally connected meta-system that has as object-system itself is called a *reflective system* (this definition is adapted from (Maes 1987) who points out that there are other interpretations of the notion of reflection – these, however, are outside the scope of the discussion here). Apart from performing computations about some other part of the world, a reflective system can actually reason and act upon itself. Figure 3.5 depicts a reflective system:

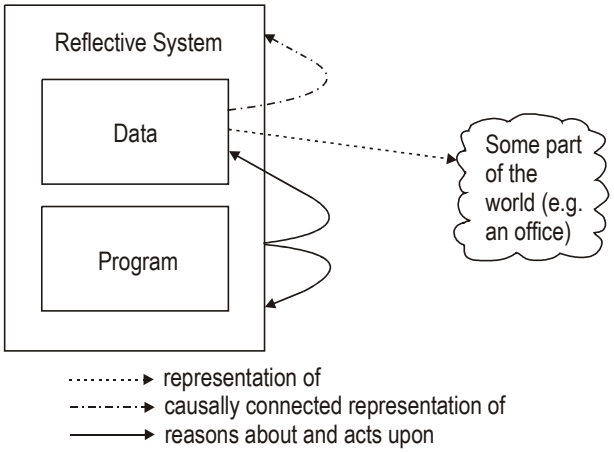


Figure 3.5: Reflective system

A system is said to perform *reflective computations* (or execute a process of *reflection*), if it evaluates or manipulates data representing – causally connected – aspects of itself. The faculty to perform reflective computations usually has to be supported by the programming environment (language and runtime system) and can take a variety of forms. Some languages simply permit the evaluation of a string of characters as a command (see e.g. the `eval` function in TCL/TK), while others permit the implementation of complex protocols for runtime inspection and manipulation of program structures and even execution states. The following discusses metaobject protocols that introduce reflective techniques in the design of object-oriented languages.

3.3.1.2 Metaobject Protocols

In the area of object-oriented programming languages, (Kiczales et al. 1991) have coined the term *metaobject protocol* for a representation of aspects of the implementation of a programming language, specifically including aspects of the static structure (the classes) of an object-oriented software system implemented in that language. In their implementation of the language CLOS (Common Lisp Object System) it is possible to inspect the internals of the implementation of the language and the structure of software systems written in that language. One says that the metaobject protocol permits *introspection* of an object-oriented software system. This is achieved by adding functions concerning the internal structure and behavior of the system to the documented language that operate on a (object-oriented) representation of the system. Classes are represented by objects of the class `standard-class`. Objects that thus represent aspects of the language are called *metaobjects*. Regard, for instance, the function:

`class-of` <object> .

This function returns the metaobject with class `standard-class` that represents the class of <object>. The metaobject offers a number of accessor functions that permit the inspection of the structure of objects instantiated from the class represented by the metaobject. Regard, for example:

`class-name` <class> ,

`class-direct-methods` <class> , or

`class-direct-superclasses` <class> .

Similar functions permit a user to request from within the system, for instance, a list of all classes used in his program.

The work of (Kiczales et al. 1991) even goes one step further by permitting the object structure of a system not only to be inspected but also to be manipulated. In this case the protocol is called *intercessory* in addition to introspective. The basic technique for extending the language is to derive subclasses from class `standard-class` and then override methods. (Kiczales et al. 1991, p. 72) give as a simple example a class that counts its instances. They derive a new subclass `counted-class`:

```
(defclass counted-class (standard-class)
     ((counter :initform 0)))
```

and then override the `make-instance` method:

```
(defmethod make-instance :after ((class counted-class) &key)
     (incf (slot-value class 'counter)))
```

The new implementation increases the `counter` of the class whenever a new object of that class is created (the methods decreasing the counter are not shown here). Employing these basic techniques of *intercession* and *introspection* a programmer has a certain degree of control over the way the language is implemented, ranging from simple issues like class instance counting to more complex undertakings like adding persistency features to the language (Paepcke 1993).

The programming language JAVA that was used for the prototypical implementation of the concepts developed in this work provides an introspective metaobject protocol by representing *classes*, *methods*, and *fields* as metaobjects. The classes of these (meta-) objects, however, cannot be used as superclasses (they are declared as `final`). Consequently, there is no protocol for intercession (in version 2). However, the introspective reflection facilities of the JAVA language are quite useful when dealing with situations in which a system has to integrate code (in form of newly written classes) that did not exist at the system's compile-

time. The following piece of code[2] demonstrates how – in principle – JAVA permits handling such a situation:

```
NewClass = ClassLoader.loadClass("NewClass", true);
Methods = NewClass.getMethods();
NewObject = NewClass().newInstance();
Methods[i].invoke(NewObject,parameters);
```

`NewClass` is loaded into the running system and by introspection the program request an array of the class methods from which it then invokes method `i` with `parameters` on a `NewObject`. Similarly, one can inspect and access fields of the new class. This technique is used in industrial composition environments like IBM Visual Age (see IBM 1998) for handling JAVABEANS components (which are – as pointed out in chapter 2 – essentially JAVA classes).

3.3.1.3 Tailorability and the Meta-Level

The astute reader already has noticed the rather striking similarities between the conceptual model of the architecture of a tailorable system presented in chapter 2 and the basic ideas of meta- and even reflective systems: a tailorable system can be regarded as the (causally connected) object-system of a meta-system which has as its data the *representation* of the tailorable properties and provides *functionality for the manipulation* these data. The *connection* between representation and actual system properties is identified as the causal connection between the data of the meta-system and the object-system. This relationship between a tailorable system and the concept of a meta-system is depicted in figure 3.6 (compare with figure 3.4):

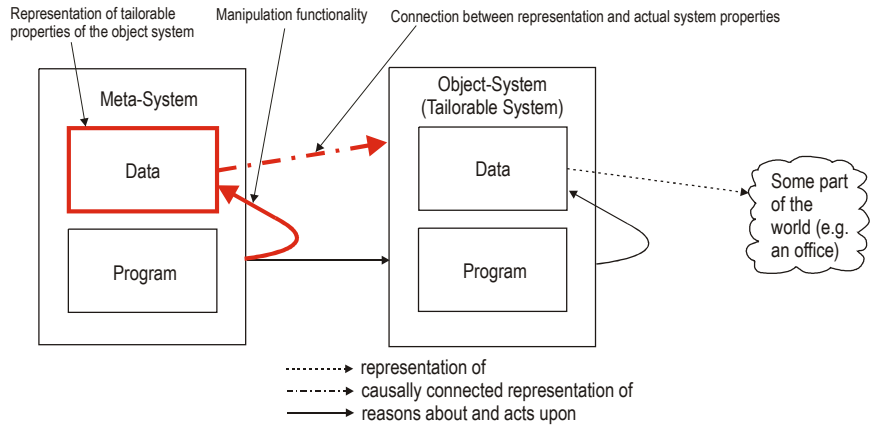


Figure 3.6: A tailorable system as object-system of a meta-system

If the representation data and the manipulation functionality is part of the tailorable system, one could go as far as identifying a tailorable system as a reflective system. However, this is not always the case, as demonstrated in the example of chapter 2, where the representation data is kept in an INI-file and is manipulated via the text editor (which is external to the tailorable system).

It is useful to draw the parallel between tailorable systems and meta- or even reflective systems, because this permits building the bridge between concepts from the area of programming language design and the questions raised in this work. In particular, the questions concerning the design of the component-model and the runtime and tailoring environment benefit from a solid understanding of the requirements posed upon a programming language to be employed in this endeavor.

---

[2] The code is simplified for demonstration purposes and thus not executable. For an executable example look at the JavaSoft web pages at `www.javasoft.com`.

Since this work aims at developing an application-independent approach that – as part of tailorability – stresses the dynamic extensibility of the system, the meta-system, i.e. the implementation of the tailoring functionality has to have a high degree of control over many aspects of the object-system. Since this is obviously difficult to achieve from the outside of a tailorable system, it seems natural to implement the meta-system of a "deeply" tailorable system in the same language as the object-system. Even if another programming language is used for the meta-system, the object-system still has to provide access to its interior. Consequently, the availability of (at least introspective) reflection facilities is almost a *sine qua non* prerequisite for the programming language used in the implementation of any general approach to extensive tailorability with the property of extensibility.

3.3.2 Design Patterns

Another approach that supports building software "designed for change" is the notion of *software design patterns* (Gamma et al. 1995). A design pattern is supposed to capture the essential idea behind a solution to a recurring design problem in object-oriented programming with the intention of re-using this idea. It takes the form of an object-oriented model of the pattern (the original book by the "gang of four" uses OMT (Rumbaugh et al. 1991) and interaction diagrams (Jacobsen et al. 1992). Today UML (Fowler and Scott 1997) – which combines these approaches – is the technique of choice, together with textual descriptions of the intent, motivation, applicability, and example code of the pattern.) Even though the notion of design patterns is not necessarily tied to object-oriented programming, the object-oriented models play an important role in visualizing and communicating the pattern.

3.3.2.1 Tailorability and Design Patterns

While the objectives of re-use and design for change might on first sight appear not to be that closely related, they imply the same design principles for system structure: namely localizing, i.e. isolating, the impact of requirement changes. By identifying a recurring solution to a problem, a design pattern – by definition – identifies aspects that are expected to remain static and define their relationship to the variable aspects of the problem. (Gamma et al. 1995, p. 23) describe the relationship between design patterns and design for change:

> "*Design patterns help you [...] by ensuring that a system can change in specific ways. Each design pattern lets some aspect of system structure vary independently of other aspects, thereby making a system more robust to a particular kind of change.*"

Consequently, a design pattern can be regarded as a solution (on the technical level) for a specific tailoring problem, providing a specific variability, the need for which arises from anticipated requirement changes. (Gamma et al. 1995, p. 30) discuss for each design pattern, which kind of change – i.e. which kind of tailoring problem – each pattern described in their book supports.

While design patterns are primarily aimed at supporting changes by the developers, i.e. decreasing the re-design effort in case of changing requirements, many of them can easily be employed to accommodate the less design- and more use-oriented nature of changes in the context of tailorability. This point is elaborated in the following by demonstrating how a specific design pattern can be used to implement a specific kind of tailorability.

3.3.2.2 Example: Using the Mediator Design Pattern for Tailorability

The aspect made variable by the Mediator design pattern is the way a number of objects (the "colleagues") communicate with each other. The idea of the design pattern is to route the communication through a central control object (the "mediator") that encapsulates the variable aspects of the communication between the colleagues. Figure 3.7 depicts the structure of this pattern (see Gamma et al. 1995, p. 276):

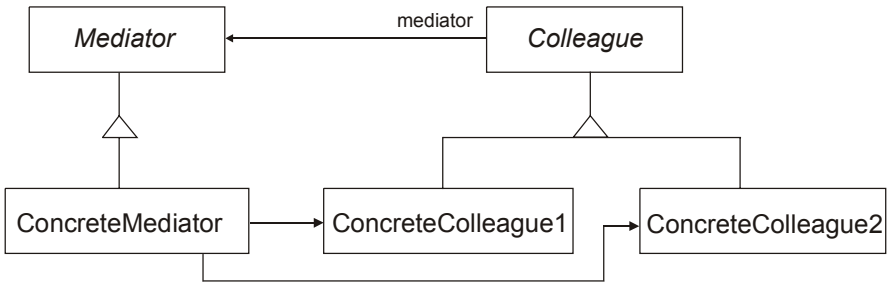


Figure 3.7: Structure of the Mediator design pattern

(Syri 1997) employs this design pattern to flexibly add CSCW functionality to a common workspace implemented by a container object. The "colleagues" from the design pattern are implemented as enabler objects encapsulating different aspects of cooperative work (e.g. access control, notification mechanisms, creation of shared objects etc.). The mediator serves as access point to the container object and "mediates" this access according to the functionality provided by the attached enabler objects. If, for instance, the event notification enabler is attached to the mediator, every access to the contents of the container object – e.g. deleting a document object within the container – leads to an event notification for interested parties. Figure 3.8 depicts this use of the design pattern to provide the container object with access control and notification functionality:

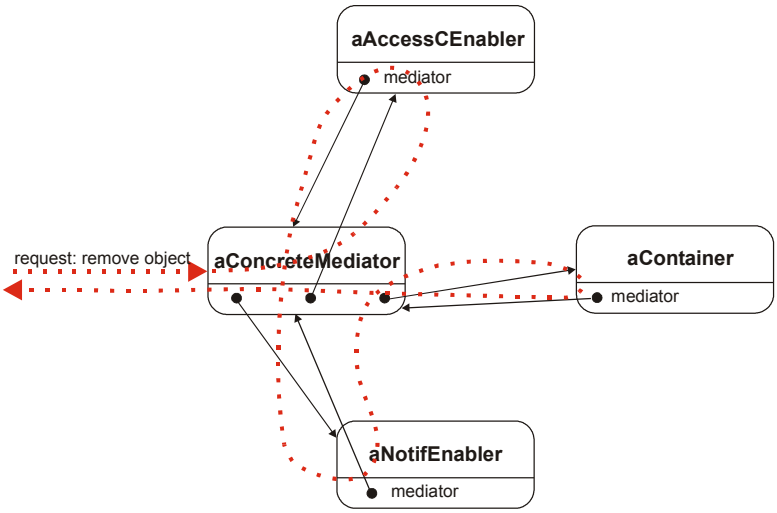


Figure 3.8: Use of the Mediator pattern to design tailorable CSCW objects

The dotted line depicts the flow of control when a request to the container object is processed (not shown here: it is also possible that the enablers are involved after the request is processed by the container object). Now, one can tailor the specific document container object by flexibly attaching or detaching enablers. The mediator "knows" about the interaction between the enablers and the container object (and possibly the interaction of the enablers among each other, e.g. when an "access denied" decision by the access enabler is supposed to lead to a notification of the owner of the container object) and can thus flexibly manage the joint behavior of the whole collection of objects.

The use of the mediator pattern by (Syri 1997) is an example for how a design pattern can be used to solve a particular tailoring problem, i.e. provide a particular kind of tailorability on the technical level. The implementation of the EVOLVE tailoring platform described in chapter 8 makes use of the *Proxy* (Gamma et al. 1995, p. 207) and the *Composite* (p. 163) design patterns together with the use of the reflective programming features of JAVA in order to provide (hierarchical) component-based tailorability in an application-independent way.

3.3.3 Explicit Domain Modeling

The technologies discussed in the preceding subsections permit the design of software systems that can change some of their properties (e.g. the behavior of the container object in the Mediator example). However, these technologies only indirectly reflect aspects of the environment. The fact that a notification enabler is connected to the Mediator of a container object might give the observer the vague idea that awareness of what happens in the container objects is somehow relevant for the environment (e.g. for the other users working with the documents in the container object). But it does not directly reflect any structures or other properties of the environment. One might call this view the "*software-as-a-tool*" perspective which implies that a software system is a part of the whole environment and – while not resembling its environment – performs a certain role in it (similar to a cheese knife which does not resemble a piece of cheese, but – due to its design – performs quite adequately in the environment of a *plateau de fromage*).

Another perspective regards "*software-as-a-model*". In this perspective, the software system derives at least part of its functionality from the fact that it contains an explicit model of its environment (or domain) and that certain information processing tasks concerning the modeled part of the domain can be automated (or at least supported) by the system. The following examples are systems that contain such an explicit model of environmental aspects that are stable during regular use of the system, i.e. changes of which can be regarded as tailoring:

- *Workflow Management Systems*. These systems usually contain explicit models of the work processes to be supported. During regular use, the process models usually remain stable and the flow of documents through the organization is guided automatically according the model (see e.g. WFMC 1995). If the supported processes change, their models have to be adapted accordingly.
- Systems with *Discretionary Access Control*. Systems in which users (or other systems) are not supposed to read or manipulate some the objects usually contain a manipulatable (at the *discretion* of the user) internal representation of the relevant access policy. Traditionally, these representations take the form a matrix or a list attached to each object or user (see e.g. Lampson 1974). More complex access policies can be modeled as sets of production rules (see e.g. Stiemerling 1996) or by including additional inheritance hierarchies over the user and object dimension of the matrix (see e.g. Shen and Dewan 1992). Whenever the access policy – which is obviously determined by many factors external to the system – changes, the system model of the policy has to be adapted, as well.
- *Mailing Lists*. Most mail clients permit the user to define mailing lists, i.e. groups of users that are often addressed as a whole. These lists can also be regarded as a (rather simple) model of certain relevant group structures in the "real world". Obviously, if the real groups change, the mailing lists have to updated to keep the behavior of the system "in synch" with the current situation.

The decision to include explicit representations of environmental aspect – which are stable during regular use – in a software system always involves a trade-off between the time one saves by automating certain tasks and the time one loses by updating the model every time the real world changes. Regard the mailing list example: this functionality saves users a lot of time, because they do not have to manually compile the addresses each time they want to send an email to an often addressed large group. Judging by the popularity of this feature, the time saved appears to outweigh the burden of keeping one's mailing lists up to date.

Integrating an explicit domain model into a system requires the choice of an appropriate data structure for the internal representation (and also the connection and manipulation functionality). In a simple case this might be nothing more than a list of access rights (see above) that is changed via a simple interface and evaluated each time an object access is attempted. In more complex cases – like for example equipping an autonomous robot with an

adaptive internal representation of its physical environment (see e.g. Fox et al. 1998) – developing the right kind of representation and manipulation/evaluation functionality constitutes a major effort.

While the work on component-based tailorability presented here focuses on the perspective "*software-as-a-tool*", it should be noted that situations could arise in which hybrid approaches are the technique of choice. Regard, for instance, the Mediator example of the last subsection. While the general functionality provided by the access control enabler is added to the container object in a compositional fashion, the fine-tuning (i.e. representing of the concrete access policy) will probably require an explicit model of the access policy as part of the state of the enabler object.

3.4 Software Components and Component Technologies

There are many technologies and products that claim the label "*component*". This subsection is designed to give the reader an overview of the meaning(s) attributed to the term in the related literature. Furthermore, it presents a number of component technologies that are fundamental for this research.

Doug McIlroy first used the term *component* in his presentation "*Mass Produced Software Components*" at a NATO Software Engineering conference (McIlroy 1968). In his talk he proposed the thesis that the software industry should – in order to achieve the same level of reuse, quality, and cost-efficiency as the hardware industry – establish a software component subindustry:

> "*When we undertake to write a compiler, we begin by saying* 'What table mechanisms shall we build?*' Not,* 'What mechanisms shall we use?*' I claim that we have done enough of this to start taking such things off the shelf.*"

His words convey the initial motivation for software components rather well. During the process of component-oriented software development – instead of building them from scratch – complex software systems are assembled from pre-existing building blocks; similar to the way electronic hardware is assembled from standardized components like transistors, resistors and multi-purpose microchips, or mechanical hardware from standardized cogs, screws etc. In transferring this practice of design *reuse* to software engineering, McIlroy expected the following benefits:

- *Higher software quality*. Using components that have already been employed (and thus been tested and matured) in a number of other software projects is likely to increase the quality of the final software product. Furthermore, component subindustries that are specialized, for instance, on audio-processing components can be expected to produce high quality software in their sector of expertise.
- *Faster software development.* Simply adding a pre-existing component to the new system is expected to be faster than writing the required functionality from scratch.
- *Cheaper software.* Counting the effort which goes into implementing certain functionalities, multiple reuse of the same component in different projects is prone to decrease to cost per unit. Furthermore, the decrease in development time is also expected to impact upon cost.

Additionally, (Parnas 1972) – in his 1972 discussion of how to decompose a software system into components (or modules) – demonstrated that the developer's effort to adapt the code of an application in the face of changing requirements could be significantly reduced by an appropriate modular structure. Thus the cost of producing a number of versions of the same software systems is likely to decrease when using components. However, Parnas also showed that an inappropriate decomposition could have the opposite effect. This can be seen as a first indicator, that the idea of component software – while principally sound – is a

distinctively non-trivial concept. This point is supported by the subsequent evolution of component technologies: while the above stated expectations of 1968 appear to be fully justified and reasonable, they only began to slowly materialize in the early 1990 with the advent of the real first component subindustries – more than 20 years after the initial idea was born (see discussion of the COM component model in the following).

The main reasons for this delay were the technological gaps that had to be filled. These gaps mainly concerned *standardization*. Without well-documented standards, it appears to be almost impossible to reuse software written by somebody else. Object-oriented mechanisms like inheritance and subclassing can help when reusing code within the same application. However, reuse of the same code in another application and by another programmer requires intimate knowledge of the design and implementation decisions. Only those object-oriented class-libraries or frameworks (Pree 1997) that provide functionality required by a very large number of applications are generally known to be successful examples for reusable software (e.g. the GUI-libraries provided by many operation systems). When reusing more specific pieces of software, the effort to understand and integrate them appears to outweigh the expected reuse benefits. Standards – similar to design patterns – provide a common ground for software developers and thus reduce the effort necessary for integrating pre-existing software into new applications. Looking at the component standardization "landscape" today, one can identify three broad areas in which standards appear to play a major role:

- *Component models*. Component models determine which requirements a piece of software has to adhere to in order for it to be a component. In particular, they specify how component instances interact, how the interfaces for these interactions are defined and what assumptions a developer using a pre-existing component with a specific interface can make about that component.
- *Distributed component interaction*. Many commercial applications today are distributed over a network (the Internet being the most prominent example). Consequently, interacting component instances can reside on different physical machines, thus giving raise to the need for a standardized distributed interaction protocol.
- *Interaction of heterogeneous components*. When integrating systems, developers are often not only confronted with distributed components, but also components written in different programming languages. In order to overcome this obstacle, standardization efforts also focus on language-independent protocols of interaction and bridges between different component standards.

The following gives an overview of some of the standards and "de-facto standard" technologies in these three areas, going into detail on those which are fundamental for this work (The component model JAVABEANS and distributed interaction technology JAVA RMI). For a more "balanced" discussion see e.g. (Szyperski 1998). In addition to the three areas of standardization, this subsection also gives an overview of approaches that deal with describing compositional structures. Such *composition languages* are an important prerequisite for component-based tailorability.

3.4.1 Component Models

The two predominant component models today are probably JAVABEANS and Microsoft's COM architecture (COM = component object model). While both aim at the same objectives of reuse and interoperability there are fundamental technological differences stemming from the commercially induced paradigmatic gulf between the two worlds: while JAVABEANS are based on a specific programming language (JAVA, obviously), COM is a binary standard, i.e. platform specific but programming language independent (However, note that while third party vendors are porting COM to other platforms like UNIX, this does not mean that a compiled binary component has the same platform independence like a JAVABEANS component).

Recently, the specification of the CCM (CORBA Component Model) has been voted on by the OMG (Object Management Group) as part of CORBA 3.0. This component model is independent from programming language and platform. Since so far there are no real applications, an analysis of it is referred to future work. However, as part of CORBA it is bound to achieve significance in the area of distributed system design. Note that, in the pre-CORBA 3.0 literature, quite often CORBA itself is referred to as a component model. However, in the terminology of this dissertation, the suite of standards provided by the older CORBA versions does not include a component model.

3.4.1.1 JAVABEANS

JAVABEANS (see JavaSoft 1997) supports component interaction via JAVA events that are essentially void method calls parameterized with a reference to an event object whose class constitutes the type of the event. While an interaction following this style only permits two roles (event source and event listener), an event source can be connected to more than one listener (multicast) and a listener can be connected to more than one source (fan-in). The JAVABEANS specification does not require any particular order in which the event source calls the event handling methods of the listeners in the case of an occurring event. Listener components and source components are connected by registering the listener with the source. Furthermore, JAVABEANS have properties (parameters) that can be set in order to customize a component instance (e.g. the size of its GUI representation). Figure 3.9 shows the delete button example from chapter 2. Employing this example the following gives the reader an overview of how JAVABEANS are implemented

Figure 3.9: Delete button example

The event object class `ClickEvent` is declared by extending the base class for all events `java.util.EventObject` (the code examples given here are simplified in order to highlight the relevant aspect. Complete and executable examples can be found in (JavaSoft 1997)):

```
public class ClickEvent extends java.util.EventObject {}; ...
```

The click receiver component on the right (which for this simple example has the sole purpose of receiving click events) announces its ability to receive click events by implementing the Java interface `ClickEventListener` which also has to be declared beforehand:

```
interface ClickEventListener extends java.beans.EventListener {
   void handleClickEvent(ClickEvent e);
}
```

Implementing the `ClickEventListener` interface requires the class `clickReceiver` to implement the event handling method `handleClickEvent`:

```
public class clickReceiver extends java.lang.Object implements
   ClickEventListener {}; ...

   public void handleClickEvent(ClickEvent e);
```

By implementing the interface, the click receiver component implements a port of the port type `ClickEvent` and the port polarity *listener* in the terminology of this work. In figure 3.9, this port is represented by the little empty circle. It was already pointed out in chapter 2 that the JAVABEANS component model does not support the concept of port names. All events of type `ClickEvent` will be handled by the `handleClickEvent` method. Thus, it is

not possible to distinguish – on the compositional level – different event sources emitting the same events of the same type.

The delete button announces its ability to act as a source for click events by implementing two methods for registering and removing click event listeners and some means to store the list of registered event listeners (here a vector data type):

```
public class deleteButton extends java.awt.button {};
   private Vector ClickEventListeners = new Vector();

   public void addClickEventListener(ClickEventListener l) {
       ClickEventListeners.addElement(l);
       }

   public void removeClickEventListener(ClickEventListener l) {
       ClickEventListeners.removeElement(l);
       }
```

All methods for registering or removing event listeners begin with the strings `add` or `remove`, followed by the name of the listener interface. Furthermore, the source component has to implement functionality for notifying all registered event listeners, whenever an event of the specific type happens:

```
private void eventHappened{
   ClickEvent e = new ClickEvent();
   for (int i=0; i<ClickEventListeners.size();i++)  {
       ClickEventListeners(i).handleClickEvent(e);
       }
   }
```

Note that similar to the event listener component, the event source component can only have one port for sending events of a certain type, because the JAVABEANS component model does not support port names.

The parameters (or properties) of the delete button component are read and changed using getter and setter methods of the component:

```
public int getXpos();
public void setXpos(int x);
```

The getter and setter methods always begin with the strings `get` or `set`, followed by the name of the parameter and permit simple tailoring of component instances.

A JAVABEANS component instance consists of a primary object which serves to identify the component instance and which is instantiated first. Additionally, an arbitrary number of auxiliary objects (like data structures etc.) can be instantiated by the primary object. The number of objects attributed to a component instance can change over its lifetime whose length is solely determined by the primary object.

A JAVABEANS component is thus defined by a single, primary JAVA class file (class files are JAVA binaries), auxiliary class files and other resources (like bitmaps). All elements are packaged together in a JAR file (JAVA ARchive) that is the compression and transfer standard of the JAVA programming language. The JAR file contains a manifest file that describes the contents of the file. The primary class file of a component can be identified with the help of the information in the manifest file. A JAR file can contain several JAVABEANS component. Once a component contained in a JAR file is identified and loaded into the JAVA virtual machine, its primary class file can be inspected with the help of the reflection facilities of JAVA. The reflection facilities permit querying a class (which is reflectively represented as an object) concerning its methods and interfaces, thus identifying its ports and parameters:

- methods beginning with `add` and `remove` identify event source ports of a certain port type which is determined by the string following the `add` and `remove` (compare e.g. the `addClickEventListener` method).

- classes implementing a ...`listener` interface identify event listener ports of a certain port type which is determined by the string preceding `listener` (compare e.g. the `clickReceiver` class).
- methods beginning with `get` and `set` identify parameters. The name of the parameter is given by the string following the `get` or `set`.

Thus it is possible to find out everything one needs to know about the ability of a JAVABEANS component to interact and to be parameterized. JAVABEANS can be instantiated via a constructor method (like all JAVA objects), parameterized via the set methods of the different parameters, and finally connected to other instances by calling the add and remove methods of the event source component instances.

It is important to note that JAVABEANS component instances are not to be seen as units of activity, as they are merely traversed by one or several threads of control. A thread can "jump" from component instance to component instance whenever an event is fired, because in this case the event handling methods of the listener is called by the event source.

JAVABEANS are designed for composition in assembly tools and support a dual usage model: they have a "design time" mode and a "run time" mode. The mode can be employed to determine the behavior and the GUI representation of a component instance. More details on JAVABEANS and in particular the complete specification of the component model can be found in (JavaSoft 1997).

3.4.1.2 COM

By defining a binary component standard, Microsoft's component object model COM (Microsoft 1999) is by nature programming language independent and thus supports interoperability of heterogeneous components. COM is insofar remarkable that it (together with its predecessor VBX – Visual Basic Controls, see Microsoft 1996) is the first component technology to constitute the basis of a real software component subindustry as envisioned by Doug McIlroy:

> "*The market for third-party components based on COM has been estimated at US$670 million dollars in 1998, with a projected 65 percent compound annual growth rate, growing to approximately US$3 billion dollars by 2001.*" (Microsoft 1999)

The central concept of COM is the *interface*. An interface defines a collection of functions that a component with that interface is supposed to implement. Interfaces are assigned unique immutable identifiers that form the basis for compatibility (type) checks. All COM components provide the interface `IUnknown` which permits its environment (e.g. other components or development environments) the inspection of the provided services (as represented by the implemented interface) of the up to then "unknown" component. This mechanism is similar to the use of the JAVA reflection facilities in the JAVABEANS component model.

Since COM is not used in the implementation of the concepts developed in this work, the reader is referred to (Microsoft 1999) for a more detailed description.

3.4.2 Distributed Component Interaction

The standards and technologies developed in this area solve the principal problem of permitting a component on one machine to use services provided by a component on another machine. In an object-oriented environment "using services" usually means that a client object calls methods of a provider (server) object. Following the presentation of the JAVA and Microsoft approaches in the last subsection, this subsection discusses how these approaches deal with distributed component interaction. Furthermore, CORBA is described as a third, language and platform independent approach.

3.4.2.1 JAVA RMI

JAVA RMI is designed to permit a JAVA program executing on one virtual machine the invocation of a method of an object on another virtual machine (i.e. possibly another physical machine) as if it were on the same machine. The basic architecture of RMI is depicted in figure 3.10:

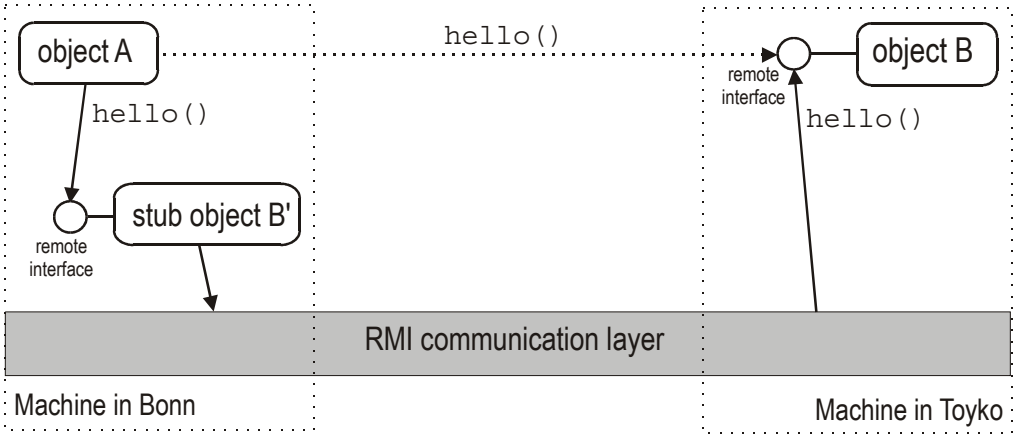


Figure 3.10: Basic architecture of JAVA RMI (version 1.2)

Instead of calling object B directly, object A calls a stub (or proxy) object B' in the namespace of its own (virtual) machine. The stub object takes the parameters of the method call (not shown in figure 3.10) and transmits them via the RMI communication layer to the other machine, where the appropriate method of object B is called. The result of this method call is transmitted back to the stub object B' which then returns the method call to object A with the result returned from object B.

There are two ways, in which object A may obtain a reference to stub object B'. First, it can explicitly employ a bootstrap naming service (the *RMI registry*) and request a reference to a remote object registered under a particular name with the service (object B could, for instance, be registered under the name "TokyoObject"). The naming service is accessed via the RMI communication layer which also instantiates the appropriate stub object and passes its reference to object A. Second, if object A is already in possession of the remote reference to another object C on the machine in Tokyo (actually a reference to the stub object C'), it can receive a remote reference to object B (to its stub object B') as a result of a remote method call to object C.

Objects A and B have to fulfill certain requirements in order to be able to participate in the remote method call. First, object B has to implement a remote interface, i.e. an interface which extends the RMI "marker" interface `java.rmi.Remote` and specifies these methods which are supposed to be accessible from a remote location (`hello()` in the example). Furthermore, object B should extend the class `java.rmi.UnicastRemoteServer` that provides functionality to make the object remotely accessible. Second, object A has to be prepared to handle exceptions that might occur due to the remote access to object B. It either has to "catch" exceptions of type `java.rmi.RemoteException` that are possibly thrown by the methods called via the remote interface, or its own methods have to be able to "throw" these exceptions, i.e. pass them through. Furthermore, accesses to a remote object always have to refer to the remote interface of object B, not its class. Last but not least, the class file for object B' has to be generated (at design time of class B) with the help of a special compiler (`rmic.exe`) provided with the JAVA development kit.

For a more detailed description and complete code examples for the JAVA RMI technology, the reader is referred to (JavaSoft 1998).

3.4.2.2 DCOM

The basic architecture for remote interaction supported by DCOM (Distributed Component Object Model) uses *proxy objects* on the client side (which play the role of *stub objects* in JAVA RMI) and *stub objects* on the server side (which are equivalent to the *skeleton objects* in older versions of JAVA RMI). A call from the client is made to the proxy of the remote server and DCOM transparently manages the marshalling and unmarshalling of the parameters. When object references are passed, DCOM maps them to appropriate proxy objects that are local with respect to the server object.

In contrast to JAVA, the format in which parameter data is stored can vary from machine to machine (recall that compiled COM components are platform specific). Consequently, DCOM needs a platform independent representation format for parameter passing (which is given by NDR – network data representation).

Since DCOM is not used in the example implementation, the reader is again referred to (Microsoft 1999) for more details.

3.4.2.3 CORBA

CORBA, the Common Object Request Broker Architecture, defined by the independent, non-profit organization OMG (Object Management Group, OMG 1995), also addresses the problem of interacting components on different physical machines. In contrast to JAVA RMI, CORBA is not programming language specific, but defines an architecture supporting the interaction of heterogeneous components. The central element of the architecture is the object request broker (ORB) that provides the core functionalities for remote method invocation. Figure 3.11 depicts the ORB in relation to other elements of the architecture:

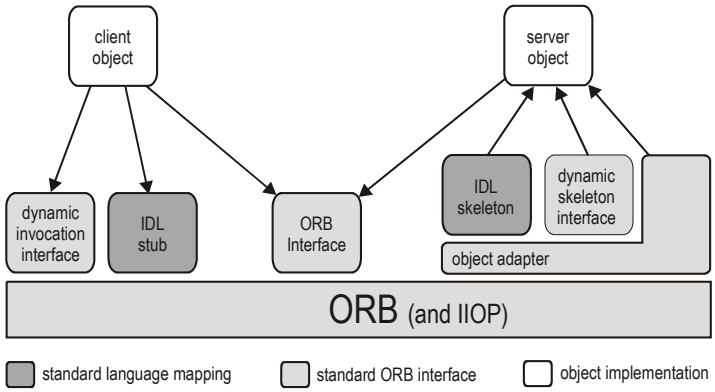


Figure 3.11: CORBA architecture

The basic design patterns for enabling remote object interaction are similar to the ones employed in Java RMI (compare figure 3.10). A client object wanting to call a method of a remote server object can do so transparently via a stub object that in turn communicates with the ORB. On the server side a skeleton object handles the actual method call on the server object. The object adapter shown on the right of figure 3.11 permits the implementation of a server object to use certain ORB services like dynamic object activation.

As an alternative to the remote method call via a stub object, a client can also employ the ORB's dynamic invocation interface that permits direct access to the underlying request mechanism. This supports a more flexible handling of method call targets during runtime, as the client object implementation does not have to have any compile time knowledge about the server object's interfaces.

The ORB itself offers a number of interfaces through which object implementations may access certain services and helper functions like converting remote object references to strings etc. ORBs implemented by different vendors may be integrated using the standardized Internet Inter Orb Protocol (IIOP). This protocol specifies how different ORBs exchange method calls. Further information about CORBA can be found in (OMG 1995).

3.4.3 Interaction of Heterogeneous Components

The discussion of two of the component technologies presented above – COM and CORBA – already hinted at the support of the interaction of components implemented in different programming languages. COM achieves this by defining a *binary standard*, i.e. its specifications solely concern the already compiled component implementation. Thus it is independent of any source programming language. In CORBA, programming language independence is achieved by defining an independent *Interface Definition Language* (IDL) that is mapped to different programming languages (currently C++, JAVA, Smalltalk, Cobol and Ada).

Even though the JAVA based component technologies are – obviously – programming language specific, bridging technologies permit the use of e.g. COM components in JAVABEANS frameworks and vice versa. The MICROSOFT JAVA virtual machine incorporates functionality for making JAVA objects accessible from COM environments.

While the interaction of heterogeneous components is of no great importance for the purpose of this work, it should nevertheless be noted that it might well be a central success factor in many real world projects.

3.4.4 Managing Component Structures

The preceding subsection presented a number of component standards and technologies permitting a programmer the development of reusable components that can interact across language and machine (or rather: process) boundaries. These technologies support the development of complex distributed object-oriented applications. However, the structure of these systems, i.e. the way the components are instantiated and connected, is usually hidden in the source code.

For component-based tailorability, it is necessary to explicitly represent the compositional structure and – in order to support runtime tailorability and independent, dynamic extensions – maintain the structure after system compilation. The following gives an overview of a number of approaches – in chronological order – which aim at making the compositional structure of a system explicit and regard composition as a distinct part of the design and maintenance process of software.

3.4.4.1 Module Interconnection Languages

(DeRemer and Kron 1976) have coined the term "Module Interconnection Languages" (MIL) for languages that explicitly describe the structure of a software system. A module has to be distinguished from a component, because there is no difference between *module plan* and *module instance*. A module is either part of a software system or it is not. It cannot be instantiated several times in the same system like a component. Consequently, a MIL does not have to deal with this distinction and simply provides a way to describe the architecture of a software system as a hierarchical interconnection of (unique) modules. In (DeRemer and Kron 1976, p. 85) the authors present an example of such an architecture description in MIL75 code (which is their implementation of a MIL) for part of compiler system:

```
system Input
author 'Sharon Sickel'
data   'July, 1974'
provides Input_parser
consists of
 root module
       originates        Input_parser
       uses derived      Parser, Post_processor
       uses nonderived   Language_extensions
 subsystem Scan
       must provide Scanner
 subsystem Parse
       must provide Parser
```

```
        has access to Scan
    subsystem Post
        must provide Post_processor
```

Abstracting from language details, this code example depicts the basic idea of textually describing the hierarchical structure of a `system` as an aggregation of interconnected `subsystems`. (DeRemer and Kron 1976) suggest a development environment that takes MIL code together with module code as input and – according to the instruction in the MIL code – produces an executable.

For the purposes of the work presented here, this early approach has two disadvantages: first, the fact that the language is module-based restricts its applicability to the more flexible concepts of components. In particular, it is not possible, to share a component plan (and subsequent changes) among several instances. A later part of this work argues for the importance of this type of sharing. Second, the approach focuses only on the development phase. It produces a monolithic executable that does not permit structural manipulations after deployment (i.e. instantiation of the system). (Prieto-Diaz and Neighbors 1986) give an overview of a number of other MILs.

3.4.4.2 Specification of Concurrent Algorithms

Another research area that has to deal with specifying component structures is the field of parallel algorithms (see e.g. Gibbons and Rytter 1988). Many computational problems can be broken down in sub-problems that can be solved in parallel (concurrently) and later be combined in order to yield the result of the original problem. Quite often, these sub-problems are similar so that they can be solved by a number of concurrently active instances of the same computational machine. Consequently, writing a concurrent algorithm involves the specification of a network of such "component" instances.

(Cremers and Hibbard 1985) give an executable specification for concurrent algorithms. The specification is based on the computational model of data space theory that is employed and described in more detail in chapter 5. Here, the specification of the structure of concurrent algorithms is of interest. The authors give the example of the wavefront matrix multiplication algorithm which involves n^2 processing elements performing the necessary computation by having the two input matrices traverse the square network in a perpendicular fashion while performing locally synchronized computations. The following code example gives the reader an impression of the structural aspects of the specification (Cremers and Hibbard 1985, p. 351):

```
SPACE Sij
 SYNCELLS a_left | a_right | b_up | b_down HOLD REAL
 CELLS c HOLD REAL
 OMEGA
       LET c = REF c + REF a_left * REF b_up
       LET a_right = REF a_left          'omitted for Sin'
       LET b_down = REF b_up             'omitted for Sni'
 EQUIVALENT Sij.a_right, Si(j+1).a_left  (j < n)
 EQUIVALENT Sij.b_down, S(i+1)j.b_up     (i < n)
```

Each processing element S_{ij} of the array communicates with its neighbors via shared memory cells (`SYNCELLS`). In the specification, these cells are connected to each other employing the `EQUIVALENT` statement. This statement in effect specifies that two `SYNCELLS` of different component instances are implemented by the same memory cell. Two thus connected component instances can interact by depositing values in (and reading them from) the shared memory cell. The specification is hierarchical in that the resulting main space M is again a data space which can serve as subspace of another, higher-level space.

In contrast to the strict separation of structural and computational aspects in MIL75, the notation by Cremers and Hibbard also permits the specification of the computational behavior of the subspace that is defined by the function `OMEGA` in the code example above. While the concise mathematical notation of the structural specification supports the

distinction between component plan and instance ($S_{2,5}$ can be regarded as an "instance" of S_{ij}), the thus defined structures are static during runtime.

3.4.4.3 Architecture Description Languages

The approaches discussed so far aim at supporting the programmer by making the structural properties of a software system more explicit and by providing development tools that permit development on this higher level of abstraction. More recent work in the field of *Software Architecture* (for an overview see e.g. Garlan and Perry 1995) attempts to gain benefits beyond mere development support, e.g. architecture validation, proof of freedom from deadlocks, and formal description of high level architectural styles or patterns. Most work in this area is based on the two abstractions *component* and *connector*. While components encapsulate computational aspects of a system, connectors define the protocols that components employ to interact with each other. While all *Architecture Description Languages* (ADL) specify compositions of components and connectors, they vary in the level of detail concerning the specification of the interaction behavior of components and connectors. For instance, the ADL UNICON (see Shaw et al. 1995) defines its interactional semantics using a fine-grained model of roles and players. By implementing an interface, a component signals its ability to act as a player (or as a number of different players). A player can take a certain role in an interaction. The roles are defined in the connector specification. An example composition in UNICON is given in the following (Shaw et al. 1995, p. 320):

```
IMPLEMENTATION IS
...
USES caps INTERFACE upcase
USES shifter INTERFACE cshift
USES req-data INTERFACE const-data
...
USES P PROTOCOL Unix-pipe
USES Q PROTOCOL Unix-pipe
...
CONNECT caps.output TO P.source
CONNECT shifter.input TO P.sink
...
```

`Caps` and `shifter` are components implementing certain interfaces. `P` and `Q` are Unix-pipes. `P` is used to connect the `caps` and the `shifter` components by identifying the player `caps.output` with the connection role `source` of the connector `P`. The real behavior is supplied by concrete implementations of atomic components (in C++). The legitimacy of a composition is determined according to the properties of components and connectors as reflected in the player / role model. While supporting detailed descriptions of compositions of components and connectors, the UNICON approach still appears more geared at producing executable systems than at supporting reasoning about system structures.

In contrast to UNICON, the WRIGHT ADL (see Allen 1997) goes beyond static structural descriptions. On the structural level it is very similar to UNICON, because it explicitly permits the specification of compositions of components and connectors. In order to model interactional behavior, WRIGHT additionally integrates a variant of CSP (Communicating Sequential Process, see Hoare 1985) – a calculus for modeling the behavior of communicating processes. Without going into the details of CSP, the following code example gives the reader an impression of the specification of a connector and the behavior expected of the different component roles interacting via this connector (Allen 1997, p. 56):

```
Connector Procedure-call
 Role Caller = call→return→Caller ∏ §
 Role Definer = call→return→Caller □ §
 Glue = Caller.call→Definer.call→Glue
        □ Definer.return→Caller.return→Glue
        □ §
```

The basic units of activity in CSP are events. In the example, `call` and `return` are events. The behavior specification has the form of traces of activity for each role in the interaction. (Allen 1997) has extended CSP with the notion of initiation events and observing events with the initiation events being underlined (actually "overlined" in the original publication). The **`Glue`** defines how the behavior of the roles interacts via the connector. Formal behavioral specifications like this permit the use of analysis tools, e.g. for deadlock checks. While the behavior specification can be regarded as a kind of "abstract" executable, the WRIGHT ADL is to be regarded more as analysis and reasoning support than as a development tool.

3.4.4.4 Distributed Configuration Management

When writing distributed software systems, the developer is forced to make the structure of the system explicit after compilation because a system running on several different physical machines can hardly consist of only one executable. Consequently, a number of approaches for developing and managing distributed component structures have come out of the field of *Distributed Systems*.

Earlier approaches like CONIC (see Magee et al. 1989) focus on describing the static structure of a distributed system. A configuration management system evaluates the description and instantiates (starts) and connects the single executables accordingly. The use of the term "executable" instead of component already implies one of the differences to the approach pursued in this work. In Distributed Configuration Management approaches – even though the term *component* is used there as well – the entities that are composed are more heavyweight than components employed in software development. They usually incorporate their own process, with the entire computational overhead implied. Nevertheless, the languages for structural description are quite similar to MILs or certain ADLs (without behavior specification) with the additional advantage that they are designed for describing system which are subject to dynamic change – in contrast to earlier approaches which utilize the structural explicitness only in the development phase.

A configuration management system can include facilities for interactive configuration management. (Fosså and Sloman 1996) describe such facilities of a specific configuration management system (the Regis system, see Magee et al. 1995). The REGIS system is based on the DARWIN configuration language that is used to persistently store a composition. Dynamic changes are applied directly to the running system. From this, a new DARWIN description can be generated and stored if the system is to be shut down and later restarted. The following code example describes a simple filter configuration in DARWIN:

```
component band-pass-filter {
      require input <sound>;
      provide output <sound>;
      inst
            A: low-pass-filter;
            B: high-pass-filter;
      bind
            A.output -- output;
            B.output -- A.input;
            input -- B.input;
      }
```

The `require` and `provide` keywords denote the input and output ports of the filter, the `inst` statement is followed by list of components to be instantiated and the `bind` statement specifies the connections between the instances.

Furthermore, dynamic changes can also be directly specified in the DARWIN language. Regard the example of a master component that is responsible for instantiating subordinate processes to deal with a dynamically changing "customer" base (e.g. client processes which log on to the master component in irregular intervals). DARWIN permits the definition of a specific `create` port of the master component that receives requests for dynamically instantiating and connecting subordinate processes according to DARWIN specifications.

While directly specifying dynamic change in the composition language is not needed for component-based tailorability, these languages are – in purpose and design – close enough to the objectives of this research to be straightforwardly applicable. The next chapter presents an experiment that employs a derivative of DARWIN (CAT) for the description of compositions.

3.5 Tailorable Groupware

Tailorability plays an important role for groupware systems, because the group dimension adds a number of requirement-dependencies on factors which are less or even not at all relevant for single user application design. Regard, for instance, country specific laws or even company specific agreements governing the access to sensitive (e.g. personnel related) documents. If a company-wide groupware system cannot be appropriately adapted to restrict access to these documents according to the applying laws and agreements, the success of the system's introduction is in serious doubt. Other examples are notification (or awareness) services in groupware. They are supposed to make users aware of activities that are relevant for the coordination of cooperative work (e.g. a users is notified, whenever a certain document – perhaps containing company rules – is edited). Since – depending on his or her role in the organization and the nature of the assigned tasks – the user's need to be aware of certain events can vary substantially, a non-tailorable awareness service could cause a flood of unwanted notifications or a scarcity of needed ones (or even both at the same time).

Additionally, the introduction and subsequent tailoring of groupware systems has a high potential for conflict (see e.g. Wulf 1995). The handling of these conflicts is – apart from CSCW-"induced" technical issues like distribution and concurrency – one of the challenges in the design and implementation of groupware tailorability. Regard, for instance, the example of tailoring a workflow system that has been in use for some time. The changes could upset established division of labor within one department, e.g. by assigning certain tasks differently. As soon as some members of the organization perceive themselves as losers in this development, the adaptation becomes a source of conflict and confrontation within the organization.

However, tailorability can also alleviate or even avoid conflicts in the context of groupware projects. If, for instance, a rigid system imposes a certain style of cooperation (e.g. highly structured workflows) across several – perhaps rather differentiated – groups within one organization, some groups may be forced to work in an unsuitable fashion. Conflicts are the obvious consequence. A tailorable system permitting the adaptation of the cooperation style for certain groups, i.e. for a certain scope of validity see (see Wulf et al. 1999), can alleviate such conflicts. Alternatively, Herrmann (1995) suggested in the context of workflow management systems to offer negotiation mechanisms within the groupware system itself which permit users affected by adaptations the rejection, acceptance, modification or discussion of these changes.

Because of the relevance of tailorability for the design of CSCW systems, there are a number of research prototypes that focus on this property. The following subsections present five research approaches (and one commercial system) and discusses the lessons learnt.

3.5.1 OVAL

The OVAL system by (Malone et al. 1995) represents one of the earlier attempts to design tailorable groupware. OVAL is based on a small, fixed set of generic groupware "components" – *Objects*, *Views*, *Agents* and *Links* – which can be composed to emulate different types of groupware applications. The four components provide the following functionalities:

- Semistructured *Objects* represent things in the world such as people, tasks, messages, or meetings. They permit the construction of domain models. An object consists of a collection of fields and field values upon which a set of actions can be performed. The object model supports inheritance along a hierarchy of more and more specialized types of objects. The objects are called semistructured, because the user is not forced to fill in all fields and because the fields can be of any type (e.g. generic text fields).
- *Views* permit the summarization of collections of objects and provide access to individual objects for the purpose of inspection and editing. For instance, a calendar view can be used to summarize objects with dates in one of their fields.
- *Agents* are rule-based (compare section "Software Technical Foundations of Tailorability") and by specifying the rule-base of an agent, a user can delegate certain tasks like mail filtering to the agent. The rules consist of a description of the objects they apply to and the tasks that are supposed to be executed on these objects.
- *Links* are used to specify relationships among objects. They can be employed, for instance, to indicate the connection between a email-message and its reply. A user can follow links in a hypertext-like fashion. Links can also be tested by rules and used in creating displays that summarize relationships.

(Malone et al. 1995) describe an experiment that demonstrates that the primary functionality of a number of groupware systems can be re-constructed within OVAL. The experiment supports their claim that the set of four generic components captures a lot of the functionality present in groupware applications. However, the experiment also showed that in each of the four cases a certain amount of system level programming had to be performed in order to emulate the target application more closely. Furthermore, the applications emulated were never put to real use, which raises doubts concerning the "real world" applicability of the approach.

3.5.2 SHARE

The SHARE system (Greenberg 1991) is an application sharing system that permits a group of users the joint use of a single user application. While the single user application is running on one machine, the output is distributed to the workstations of the users. Individual users can take control of the application according to a *floor control policy* that (Greenberg 1991) identifies as a necessarily tailorable aspect of an application sharing system. One needs different floor control policies, because "*groups will differ in how its members interact with each other.*"

In order to support a number of different floor control policies, the SHARE system's architecture implements a (domain specific) protocol for handling floor control policies. A policy is embodied by a so-called *Turntaker* which is a process running on each user machine and which communicates via the protocol with the *Chair Manager*. Users can change the policy by selecting a new *Turntaker* from a library. A programmer can extend this library without having to change the source code of the SHARE kernel that is independent from the implementation of the *Turntakers*. The programming of a new *Turntaker* is supposed to be quite straightforward, because the policy can be easily described with the help of protocol primitives.

While the tailorability of Share is simply a selection of predefined alternatives (the *Turntakers*), the system's architecture nevertheless acknowledges the need for accommodating unanticipated alternatives by supporting extensibility with the help of a clearly defined protocol. In contrast to OVAL, SHARE is not based on an implicit "genericity" assumption.

3.5.3 PROSPERO

PROSPERO (Dourish 1996) is an approach that aims more at groupware developers than end users. The system is implemented as a highly adaptable toolkit based on the reflective programming language CLOS (see Kiczales et al. 1991). Contrary to the "genericity" approach of OVAL and similar to SHARE, PROSPERO is based on the understanding that especially in cooperative work not all eventualities of the circumstances of use can be anticipated and embodied in one fixed toolkit or set of components. Therefore, it employs the approach of *open implementation* (see Kiczales 1996) in order to permit developers the inspection and manipulation of the implementation of toolkit functionality in the face of unanticipated requirements in a specific groupware project.

The toolkit focuses on two areas of functionality that are elementary for CSCW applications: *distributed data management* and *consistency and conflict control*. The former is concerned with the way the jointly used data is distributed over the network and the latter deals with the occurrence of conflicting inconsistencies causes by the divergence and later convergence of different streams of (user) activity manipulating the same data. The open implementation approach permits changing, for instance, the strategies used for data distribution and conflict resolution.

The approach focuses on the development phase of the software lifecycle. Consequently, PROSPERO does not have to deal with issues like runtime tailorability or interfaces for tailoring. However, it is discussed here because it is the only approach mentioned in the literature that explicitly uses *reflective techniques* (compare section “Software Technical Foundations of Tailorability”) to provide extensive adaptability of system properties.

3.5.4 DWCPL

DCWPL by Cortes (2000) represents an approach that clearly separates computational from coordination issues by capturing the latter in a special purpose language (DCWPL = *Describing Collaborative Work Programming Language*). The language offers constructs for session management, conflict resolution, awareness support and other issues relevant for groupware.

A complete groupware application consists of a number of computational modules (e.g. in C++) that at some points rely on coordination decisions, which are specified separately in a DCWPL program. This program is not compiled but interpreted in a special environment, thus permitting runtime changes to e.g. floor control policies in a synchronous groupware application (compare with SHARE in 3.5.2).

The design of the language draws a fixed line concerning which parts of a groupware application are tailorable and which are not. In contrary to extensible or open implementation approaches like SHARE or PROSPERO, unanticipated requirements are difficult to address. However, the DCWPL language is quite powerful and expressive and thus probably is able to cover a lot of requirements. This expressiveness can also be a disadvantage, as it makes the language rather complex and thus places it probably beyond the capabilities of even some system administrators.

3.5.5 Structuring Guidelines

While the work presented above concerns specific applications, toolkits or CSCW languages, (ter Hofte 1998) has approached the challenge of designing tailorable groupware from a more fundamental perspective. He has analyzed a large number of existing groupware systems (research and commercial) and – based on a generic model of CSCW and groupware – has distilled a set of *structuring guidelines* which are supposed to help a developer in decomposing an envisioned groupware system into functional components “*in such a way that extensibility and composability can be achieved*” (p. 114). While, for instance, OVAL is based on a fixed set of components and in SHARE the dimension of extensibility is fixed, the structuring guidelines as a generic development aid result in different decompositions for

different applications in so-called *grouplet services* (ter Hofte's terminology for groupware components). He states five guidelines:

1. *Separate different media in different grouplet services*. This guideline is based on the observation that the combination (or composition) of different media (e.g. audio, video, bitmaps, or text) in a collaborative session heavily depends on the task at hand, the preferences of the group, time pressure etc.
2. *Keep together different forms of coupling and aggregation in a medium*. A group of users tends to cooperate more closely during some periods of cooperative work and more loosely during others. If the degree of coupling changes, the users should not be forced to transfer the data to other media, or – on the technical level – to another component.
3. *Encapsulate a single-user application in a grouplet service*. This guideline stems from rather pragmatic considerations, namely that a lot of functionality that is of use in a groupware system (e.g. editing of text data) is already present in single user applications. By encapsulating existing single user applications in grouplet services, first the groupware development and introduction is jumpstarted due to the users' familiarity with single user applications and secondly, the migration to more closely coupled future services is already taken into account.
4. *Separate conference management services from other groupware services*. Since most synchronous groupware applications are based on the concept of a conference (*video* conference, *data* conference etc.), the separation of these aspects enhances reusability of existing components.
5. *Separate coordination services from other groupware services*. An example for the viability of this guideline is the exchangeable floor control policy in the SHARE system described above. The appropriate floor control policy depends on many group-related factors and can and should be separated from the other aspects of the groupware system.

While these guidelines reflect a lot of what is know today about how groupware systems should be structured and how they evolve, these guidelines have so far only been applied to the one example system (a shared text editing system) developed in (ter Hofte 1998). It remains to demonstrate the applicability of the approach to a wider variety of groupware systems.

3.5.6 A Commercial Example: LOTUS NOTES

Since the research community claims that tailorability is such an important property of groupware systems, it is interesting to see how this need for tailorability is reflected in an existing commercial groupware system that has been on the market for some time in a number of consecutive releases. Lotus Notes is an example for such a system (see Dierker and Sander 1997) and – not surprisingly – offers a cornucopia of tailoring (or extension) techniques:

- The end user can "program" a personal agent (e.g. an email filtering agent) by simply selecting its behavior from a list of *simple actions*. This tailoring option is similar to way agents are tailored in OVAL (see above).
- If sequences of actions have to be specified, a more experiences user can employ a *macro language*.
- *LotusScript* is a full-fledged, Basic-like (but object-based) programming language that permits the specification of more complex behavior and computations.
- The *Lotus HiTest API* provides a high level application-programming interface (API) which aggregates the more fine-grained functionality of the other two APIs described in the following.

- The *C-API* provides an interface to almost all NOTES functionality on a rather fine level of granularity. The use of this interface is recommended only for professional developers.
- The *C++-API* is similar to the C-API. However, it builds more straightforwardly on the object-oriented concepts supported by LOTUS NOTES.

In addition to these tailoring techniques the system also offers supporting services that, for instance, help the developer when distributing the application to remote locations of an organization. There exists a whole market of NOTES add-on products which – employing the mechanisms described above – support specific domains like banks or insurance companies.

Furthermore, it is interesting to observe that NOTES also meets the requirement of providing multiple tailoring interface for tailors of different skill levels (compare subsection 3.2.2) by providing different tailoring techniques requiring only a gradual increase in skill.

3.6 Conclusions

The first objective of this chapter was to give the reader an overview of the multifaceted nature of tailorability and – for the technical perspective taken here – to review the requirements that stem from the other, not purely technical facets of the problem.

Related work in the area of HCI suggests the usefulness of multiple user interfaces for tailoring with only modest increases in necessary skill when switching from an easier interface to a more powerful interface. Consequently, a tailorable system's architecture *should support to view and manipulate a system's implementation on multiple levels of detail and complexity*. Furthermore, it was argued that it is inherent in tailorability that some implementation concepts are exposed to the tailoring user. Thus *the basic architectural concepts should be simple and should lend themselves to a clear presentation at the user interface level*. Obviously, subjective cognitive notions like "simple" and "clear" usually escape a formal scientific treatment and can only be approximated in empirical evaluations (which are not within the scope of this work). Regarding tailoring as a collaborative activity exhibits the requirement to permit the *identification and isolation of system adaptations as exchangeable and (perhaps even runtime) shareable entities*. The process perspective is probably farthest from the software technical perspective taken here. However, once the programmer has developed an understanding of which aspects of the system are supposed to be flexible and which can be rigid, *a tailoring architecture should permit the straightforward implementation of the envisioned flexibility*. While this work develops technical foundations for tailorability, it explicitly takes these four requirements into account. In this fashion, the interdisciplinary nature of tailoring is acknowledged.

The second objective of this chapter was to present the technologies and concepts which are fundamental for or related to component-based tailorability.

Regarding tailorable systems as meta- or even reflective system suggests *the use of programming languages with reflective facilities and meta-object protocols* when implementing these systems. From this perspective, the programming concepts offered by the JAVA programming language – which was used in the prototypical implementation of the concepts developed in this work – were briefly surveyed and positioned. As another fundamental concept for flexible systems, the notion of design patterns was introduced. An example from the current literature was given which demonstrated *how design patterns can be used as a software-technical means to support tailorability* in a specific case. The implementation of the prototype described in chapter 8 uses a combination of design patterns to provide tailorability on a more generic level. Finally, a number of other approaches to tailorability were discussed, which have in common that they incorporate *explicit models of environment aspects* like access control policies or organizational structures and processes.

These "system-as-a-model" approaches were contrasted with the "system-as-a-tool" view that prevails in this work.

Software components were introduced and their traditional role as *support for cost-efficient development of high-quality software* was discussed. It is important to contrast this perspective with the novel approach taken in this work that employs software components to gain benefits (namely tailorability) *after development and deployment*. The presentation of a number of current component-related technologies (*component models*, *remote and heterogeneous interaction protocol standards*) demonstrated the *essential relationship between component technologies and standardization efforts*. The JAVABEANS component model and the JAVA RMI remote interaction protocol were described in detail due to their use in this dissertation's implementation. Finally, a number of approaches are reviewed which deal with *management of software structures*. Related work from the area of *distributed configuration management* – in particular the DARWIN configuration language and the REGIS system – appears to be closest in intention to the objectives pursued here, because the nature of distributed systems inherently requires explicitly maintaining the compositional structure after development and deployment.

The third objective was to give an overview of the tailorability implemented in a number of current research and commercial (groupware) systems. These examples support two points:

First, when designing tailorability for a certain class of applications (here: groupware systems) the central question is always*: which aspects of the system are supposed to be tailorable, which can remain rigid and at which points should the system be extensible*? In all research prototypes this questions was addressed (in one fashion or another) by separating these aspects that were expected to evolve separately and by combining those that were supposed to evolve together. This point emerges most prominently in the recent work by (ter Hofte 1998) who explicitly focuses on *decomposition as the fundamental prerequisite for tailorability and extensibility*.

Secondly, the example systems demonstrate how a variety of the techniques presented in the preceding sections can be employed to design tailorable groupware. Especially the commercial example LOTUS NOTES shows that *there is no one best tailoring technique* but that a real world system should incorporate a number of those for different tailorable system properties and tailoring users.

While the first point supports and further motivates the basic concept of component-based tailorability, the second point suggests – for every tailoring technique – to explore the limits of its applicability.

Chapter 4

Exploratory Case Study

4.1 Introduction

This chapter describes the initial exploratory application of the component-based tailorability approach to a real-world tailoring problem in the context of the POLITeam groupware project. It serves as an introductory example showing how component-based tailorability can be realized. Furthermore, the study constitutes a proof of concept for the approach. It also raises and motivates the specific questions addressed in this dissertation by demonstrating the problems with the state-of-the-art technologies employed in the study.

4.2 Setting and Technologies

In the introduction, the expected advantages of component-based tailorability were described. In order to verify these expectations, to prove the general feasibility of the approach and to identify possible open issues, a case study was conducted. This study covered all aspects of component-based tailorability from the first implementation of a prototype to its introduction in a field of application. Furthermore, during the experiment a number of representative state-of-the-art technologies and concepts were employed as basis for the four elements of component-based tailorability (as discussed in chapter 2): JAVABEANS as component model, a DARWIN derivative as the composition language, and an extensively modified BEANBOX IDE (Integrated Development Environment) as user interface for tailoring and for the realization of the connection between representation and application (for a more extensive overview of the modifications of the BeanBox and the implementation of the example component set see Won 1998). The purpose of using these technologies was to evaluate their appropriateness for the component-based tailorability approach and to identify possible deficiencies of the current state-of-the-art. The remainder of this section describes the POLITeam project, the search tool tailoring problem, and the three technologies employed.

4.2.1 The POLITeam Project and the Search Tool Tailoring Problem

The POLITeam project was concerned with providing electronic support for the cooperative work of the distributed German government in Bonn and Berlin (for an overview see Klöckner et al. 1995, Cremers et al. 1998). The two fields of application relevant for this case study were a federal ministry and the representative office of the State of Mecklemburg Vorpommern in Bonn. In the course of the project the commercial groupware platform LINKWORKS (DEC 1997) was tailored to the specific requirements of the respective fields of applications and introduced as pilot installations. The POLITeam system supported asynchronous document-based cooperation and provided functionality for circulation folders and shared workspaces in a virtual desktop environment (for more details see the literature cited above). The project employed a cyclical and participatory approach with several consecutive phases of system change and evaluation in user workshops and work place visits.

In the first phase of the project a number of requirements emerged which could not be met by tailoring LINKWORKS because the relevant system properties were not easily accessible using the tailoring mechanisms offered by the system. A number of these requirements concerned the LINKWORKS search tool employed in the POLITeam system to search for documents (e.g. MICROSOFT WORD documents) in the shared system database. Figure 4.1 shows a screenshot of the original LINKWORKS search tool:

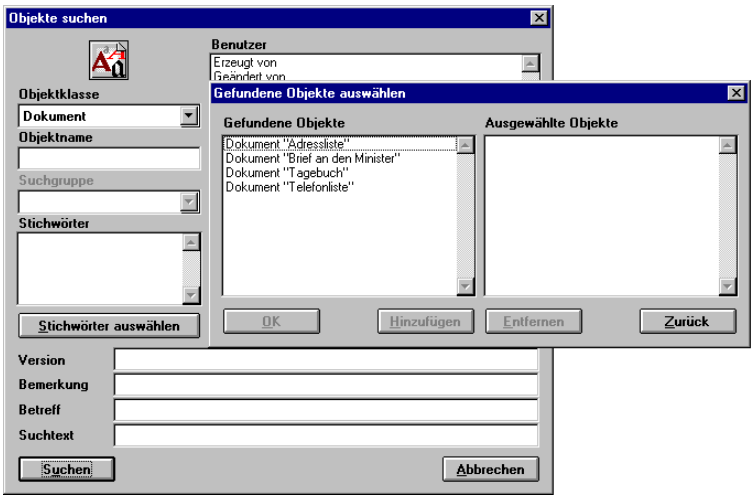


Figure 4.1: The original LINKWORKS search tool (German version)

The window in the background (search window) permits the specification of a search by declaring a number of document attributes like its type (e.g. word document or spreadsheet), its name, keywords etc. The search results are displayed in the list on the left side of the front window (result window). The user can select a number of found documents and create links to these documents on his own virtual desktop by adding them to the list on the right and pressing the OK button.

While the search tool functionality is by no means complex, its implementation nevertheless proved to be rather problematic for the users (compare Kahler 1996). In several workshops and interviews they articulated rather diverse opinions about a number of requirements:

First, the search window was much too complex. Most users only used a small subset of attributes for specifying a search and were confused and irritated by the large number of GUI elements present in the search window. However, there was no common subset of required attributes, because some users required document name and type, while others preferred to search for documents created within a certain time interval.

Second, the end users were concerned about the power of the search tool to find documents on the virtual desktops of other users. The virtual desktop metaphor had led them to believe that a private desktop was indeed private. However, the LINKWORKS search tool does not respect the virtual location of a document, only the explicitly set access rights which include the right to find a document. Thus it was possible for a user to find (and create a link to) a document on another user's presumably private desktop, if the find right for that document was not explicitly denied. Because of the insufficient usability of the LINKWORKS access control system (Stiemerling 1996), the users required a search tool with a search space limited to the desktop of the searching user and certain shared workspaces. However, in special cases, trusted users were supported in searching for documents on the virtual desktops of absent colleagues (compare Stiemerling and Wulf 1998).

Third, the result window as shown in figure 4.1 presents the search results as a simple list. Several users wanted to have a mechanism permitting the grouping of search results according to a variety of criteria. One user in the state representative body wanted to have the results grouped according to certain time intervals. These requirements reflected the practice in the German Bundesrat (in which representative of all German states vote on legislation) to discuss a new law in two sittings separated by a few weeks. Concerning the result window, the user – who was responsible for preparing a state's vote on certain topics – wanted the found documents to be presented in two lists: one list was supposed to show the recent documents (creation date less than one month in the past) concerning the second sitting; the other was supposed to show the older documents of the first sitting. The user assumed this grouping strategy would facilitate the use of the search tool for his specific tasks in preparing the state's vote. Other users – in the Federal Ministry – preferred a different grouping of results according to a document's location in the virtual desktop environment in two groups: "my own desktop" and "everywhere else".

Finally, the users criticized the inflexible handling of the documents selected in the result window. Some users wanted the search tool to create a copy of the document on their own virtual desktop; others found the already implemented creation of a link sufficient. A few wanted both options. The problem with the creation of a link was the fact that changes made to a linked document automatically concern everybody else who is sharing it. In some cases this was exactly what was intended, but in others it caused confusion and resulted in lost data.

In addition to the diversity of requirements indicated by these four cases, it was also observed that requirements appeared to change over time. Sometimes this change was rather short-cycled and task dependent. For instance, the state representative body worked on a strict weekly schedule with different tasks to be executed on different days of the week. In one case this was reflected in the grouping requirement. On days when the vote in the Bundestag was prepared, grouping according to time-intervals was considered helpful. On

other days other grouping strategies or even only one list of results appeared more appropriate.

Taking a more a long-term perspective, it was observed that experienced search tool users (e.g. the system administrator in the state representative body) did not have as many difficulties with the complex search window as other, less experienced users. The administrator actually preferred a search tool with many attribute options that permitted him to better refine his searches. Consequently, it was concluded that the search tool requirements of a user or group of users might change over time with increasing qualification.

These four areas of diverse and possibly dynamic requirements the end users articulated for the search tool demonstrate why the search tool was an attractive real world test case for the component-based tailorability approach.

However, the search tool also had its limitations as a test case because it represented only a small local part of an otherwise distributed groupware application. While this fact reduced the value of the case study concerning the issues of scalability and applicability to whole groupware applications, it nevertheless made it possible to go through all aspects of the approach from software-technical issues of implementation to the more user-oriented issues of tailoring interface and the process of tailoring itself. Furthermore, the additional requirements on the software-technical side when applying the approach to whole groupware applications are quite straightforward. They are discussed at the end of this chapter.

In the course of the experiment the functionality of the search tool was decomposed into a set of components that were then implemented according to the JAVABEANS component model that was already described in chapter 3. The next subsection describes the component set and an example composition of a search tool.

4.2.2 The Search Tool Component Set

The set of components was designed to support a range of possible compositions covering the diverse and dynamic requirements described in the last section. Table 4.1 gives an overview of a representative selection of the resulting JAVABEANS components (implemented by Won 1998).

As described in chapter 3, JAVABEANS interact via events. The components shown in table 4.1 essentially rely on three types of events: *click events* are employed to transmit user commands from the GUI, *attribute events* transmit changed search attributes to the search engine, and *result events* are used to exchange lists of found documents. Since the JAVABEANS model does not support port names, the two source ports for result events of the *location* and *date switch* component had to be implemented using two different subclasses of the result event listener interface (the problems with this rather "dirty" workaround will be discussed later on in this chapter). Figure 4.2 shows a simple example search tool composition.

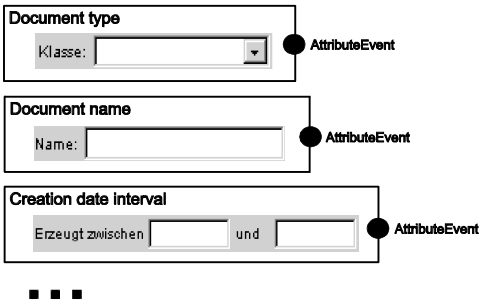
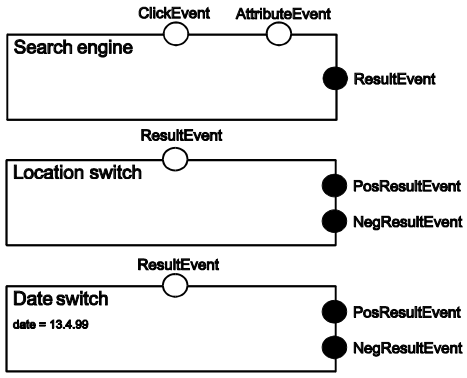
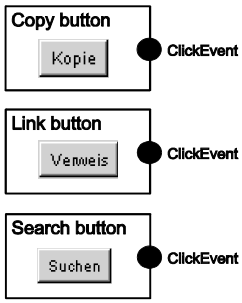
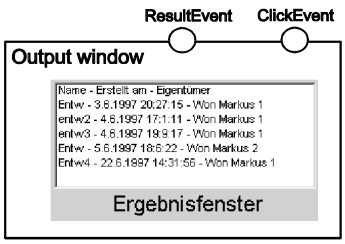
	**Attribute components** These components have a GUI representation that permits users the specification of document attributes like name, type or creation date interval. All attribute components have an `AttributeEvent` source port which is usually connected to a search engine (next row), Whenever the contents of an attribute component change, it emits an event which is received by the search engine and stored there in anticipation of the start of a new search. There are several other attribute components not shown here which work along the same lines.
	**Invisible components** These components have in common that they do not have a GUI representation. The *search engine* connects the search tool to the LINKWORKS document database. It has a port for listening to `AttributeEvents`. The `ClickEvent` listener port is usually connected to the *search button* (next row) and initiates a search. The result of the search is transmitted with the help of a `ResultEvent`. The two switch components receive a `ResultEvent` and group the result according to either the document's location (own desktop or elsewhere) or the document's creation date that is given by a parameter. The switches then emit two result events (subtypes of `ResultEvent`).
	**Button components** These components are visible at the GUI and permit the user to execute certain actions. The *link* and *copy button* components can be connected to an output window. If an entry in the output windows is marked, either a copy or a link will be created on the user's desktop. The *search button* component can be connected to the search engine and pressing it triggers a search according to the attributes specified earlier. The actual copy, link and search functionality is not encapsulated within these components but in the output window (next row) or search engine, respectively.
	**Output component** The output window displays the list of found documents that it receives on its `ResultEvent` listener port. It permits the selection of a document (by clicking on its list entry with the mouse). Via the `ClickEvent` port a copy of or a link to the selected document can be created on the user's virtual desktop, depending on which buttons are connected.

Table 4.1: Components of the search tool component set (GUI representation in German)

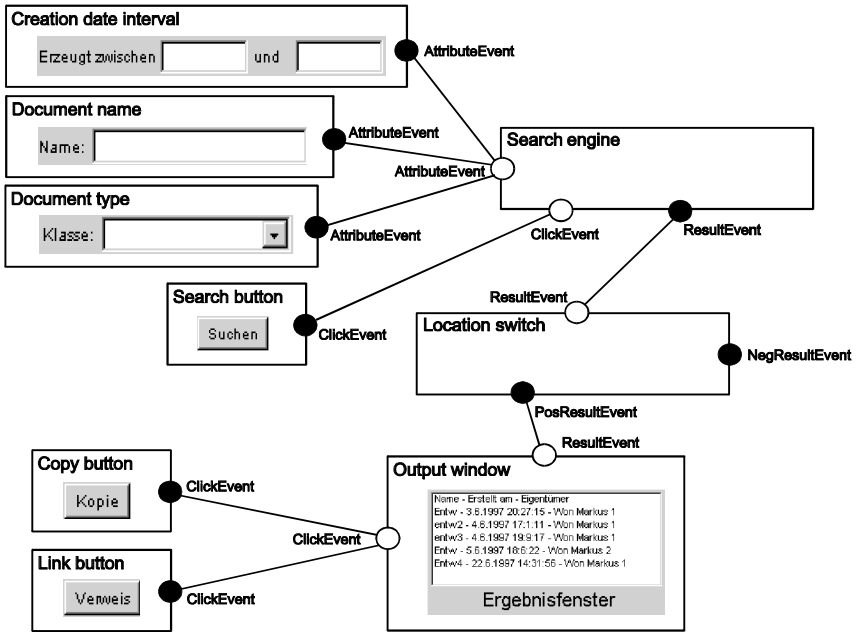


Figure 4.2: Example composition of a simple search tool for searching only on the user's own virtual desktop and for optionally creating copies of or links to found documents.

The search tool composition depicted in figure 4.2 permits a user the specification of only three search attributes: *document name*, *document type* and a *creation date interval*. The *attribute* components are connected to the *search engine* component that is informed whenever the user changes one of the attributes. From the attributes received, the search engine compiles a query that is sent to the LINKWORKS database, when the *search button* is pressed. The results that are returned to the search engine from the database are then forwarded to the *location switch* component which divides the result list into one list containing all documents found on the user's own desktop and a second list containing all other documents. The first list is sent to the *output window* component, the second list is lost, because the `NegResultEvent` source port is not connected to anything. Thus the list appearing in the *output window* only contains documents from the user's own desktop. The user can then select single documents in that list and create either a copy of or a link to them by pressing either the *copy* or the *link button*.

It is quite straitforward to compose a number of different search tools meeting the requirements stated in the last section. For instance, in order to increase the power of the search tool of figure 4.2 for the trusted users who are allowed to find documents on their colleagues desktops, one only has to add another *output window* (with assorted instances of the *link* and *copy button*) to the search tool and connect it to the `NegResultEvent` port of *location switch* component. The requirements of more experienced users such as the system administrator who prefers a more extensive selection of attributes can also be easily met by adding additional attribute components (not shown in table 4.1. In the original component set there were seven other attribute components). Other grouping strategies (e.g. according to a specific creation date) can be supported by exchanging the *location switch* component with another *switch* component.

4.2.3 Representation of Compositions

In order to persistently maintain a representation as depicted in figure 4.2 in the system, a derivative of the DARWIN configuration language (see Magee et al. 1995) was used in the exploratory case study. Originally, DARWIN was used to describe in executable form the (possibly dynamic) structure of large distributed systems, with each component incorporating at least one thread of control or even a whole process. For the purpose of the

case study, the language was simplified by removing all language constructs concerning aspects of distribution and dynamic component instantiation. The resulting DARWIN derivative is called CAT (Component Architecture for Tailorability); its syntax is specified in appendix A and is used throughout this dissertation. The CAT composition language permits the specification of static, hierarchical compositions of components with typed, named and polarized ports. The fact that CAT supports named ports was not relevant for the experiment, because of the deficiency of JAVABEANS concerning this point.

The operational semantics of CAT is quite intuitive (in chapter 6 it will be more formally defined). The following shows a part of the CAT representation of the search tool composition of figure 4.2:

```
i_component searchEngine {
 required init ClickEvent;
 required attribs AttributeEvent;
 provided out ResultEvent;
 config_parameter resultMask int;
};
i_component outputWindow {
 required buttons ClickEvent;
 required in ResultEvent;
};
i_component locationSwitch {
 required in ResultEvent;
 provided pos posResultEvent;
 provided neg negResultEvent;
};
i_component searchButton {
 provided out ClickEvent;
};
... // not all declarations of implemented components are shown

a_component exampleSearchTool {
 subcomponent searchB searchButton;
 subcomponent searchE searcheEngine{resultMask := 31;};
 subcomponent locationS locationSwitch;
 subcomponent resultList outputWindow;
 ... // not all subcomponent declarations are shown

 bind searchB.out → searchE.in;
 bind searchE.out → locationS.in;
 bind locationS.pos → resultList.in;
 ... // not all binds are shown
}
```

In CAT the atomic components of the initial JAVABEANS component set are declared as implemented components (`i_component`) in order to indicate that they are available in binary form and not composed from other components. The `required` and `provided` keywords indicate the polarity of a port. The `config_parameter` keyword indicates tailorable parameters. Complex components are declared as "abstract" components (`a_component`) in order to indicate that they do not have a direct binary implementation but are compositions of other components (possibly even of other complex components). A complex component is specified by listing components that have to be instantiated in order to form the complex component (the `subcomponent` statement) and possibly the parameterization of the subcomponents. Furthermore, employing the `bind` statement, a CAT program declares which ports are to be connected within a complex component. The parameterization of components is not shown in the code example above, because it is quite straitforward (see appendix A).

4.2.4 A Simple Runtime and Tailoring Environment

The last subsections described the implementation of a set of atomic search tool components and the DARWIN derivative CAT to represent compositions of these components. This subsection is about the two elements still missing for component-based tailorability: the functionality for manipulating the internal representation and the connection of the representation to the actual application. Both elements are provided by a simple runtime and tailoring environment which is based on the extensively modified JavaSoft BEANBOX. The BEANBOX is the example IDE provided by JavaSoft together with the JAVABEANS component model. Its original purpose is to demonstrate to IDE developers how JAVABEANS components can be loaded, analyzed and assembled into applications. The BEANBOX was used as the basis for the experiment, because its source code is freely available and it provides a lot of functionality also needed in a runtime and tailoring environment. However, the user interface and the mechanisms for component assembly were extensively modified reflecting the specific needs of component-based tailorability. In contrast to the original BEANBOX, the new user interface shows the connections between components and their ports and permits direct manipulation of connections via the mouse. During component assembly the original BEANBOX generates and compiles glue-code in form of adaptor objects that mediate the connection of components. There is no explicit system-internal representation of the composition. The modified BEANBOX maintains the composition using the CAT composition language and relies on direct connections between components (using the add and remove methods of event source to directly connect them to event listeners) instead of code-generation and compilation. (Won 1998) describes the alterations made to the original BEANBOX in more detail.

The modified BEANBOX interface provides the user with two distinct modes: the *tailoring mode* (similar to the "design" mode supported by the JAVABEANS component model) and the *use mode*. Figure 4.3 shows the use mode (upper screenshot) and tailoring mode (lower screenshot) for a simple search tool composition:

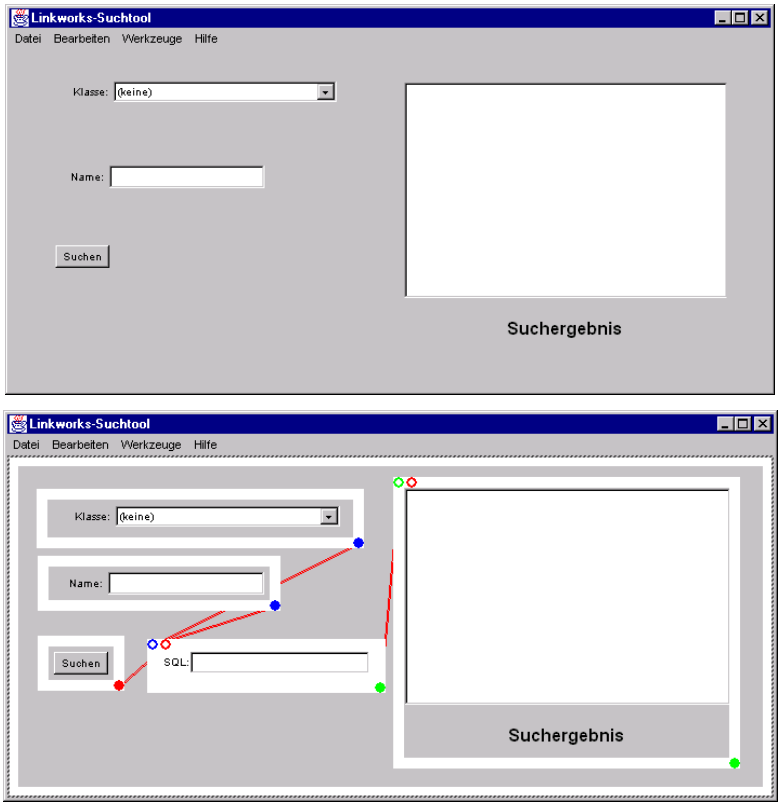


Figure 4.3: Use and tailoring mode of the modified BEANBOX interface (showing a simple search tool composition)

The visual representation of the composition is similar to the graphical representation used in this dissertation. Component ports are represented by small circles – with filled circled indicating event sources and empty circles indicating event listeners. The color of the circles denotes the event type (blue `AttributeEvent`, green `ResultEvent`, and red `ClickEvent`). Two component ports are connected by red lines.

In tailoring mode, the user can connect and disconnect component ports, instantiate new components (from a toolbox-like window not shown in figure 4.3) and remove existing instances. Furthermore, component instances can be grouped together to form a complex component instance, which permits viewing and manipulating a composition on different levels of abstraction and complexity. A whole search tool is regarded as a complex component. Several alternative search tool compositions can be selected from a (extensible) menu, allowing a user to rapidly tailor the tool to different requirements arising from different tasks.

The modified BEANBOX does not provide any functionality for automating phases of the tailoring processes. In terms of the phase model of (Kühme et al. 1992), the user is responsible for initiating a tailoring activity by switching from use to tailoring mode (via a menu item), develop a (mental) plan for changing the composition of the tool according to the new requirements, decide to use this plan and finally execute it by directly manipulating the composition via the mouse. Consequently, the manipulation functionality implemented in the case study makes the system *user tailorable* as opposed to *adaptive*.

The second element of component-based tailorability addressed in this subsection – the connection of the representation of the composition and the actual search tool – is realized in the modified BEANBOX by reading a CAT-file describing the current search tool at start-up time and building up a structure of proxy objects, each of which manages one specific instance of a component (these proxy objects are called "wrappers" in the terminology of (Won 1998), because they implement the wrapper design pattern described by (Gamma et al. 1995)). When the user switches to tailoring mode, the proxy objects provide the tailoring operations by appearing as frames around component instances (also see lower screenshot in figure 4.3). The frames contain the little circles indicating ports and also serve as handle for moving components around the screen. During shutdown or whenever the user issues an explicit "save" command, the proxy structure is evaluated and the old CAT file is overwritten. Figure 4.4 depicts the basic architecture of the runtime and tailoring environment:

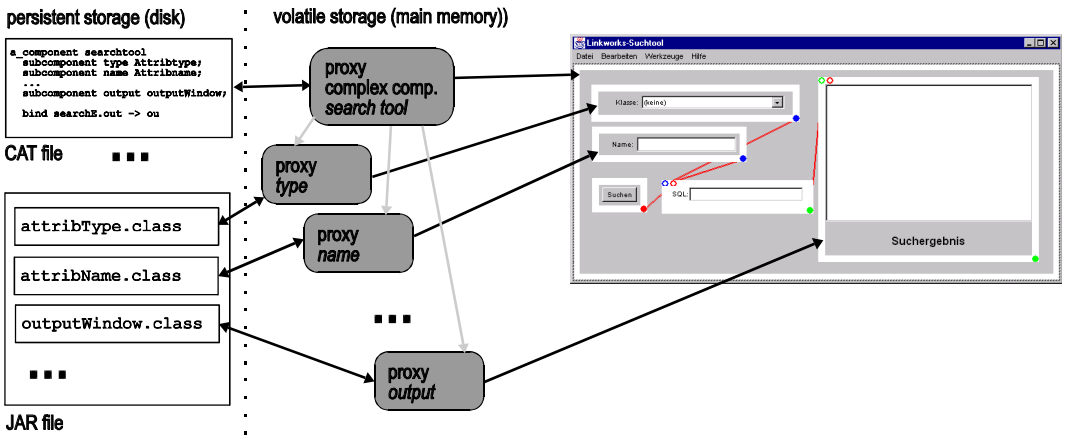


Figure 4.4: Architecture of the simple runtime and tailoring environment

There are two types of proxy objects: simple proxies manage atomic component instances; complex proxies manage compositions. Consequently, the latter can have children (either atomic or again complex components). The *has_as_child* relation is represented in figure 4.4 by the gray lines.

The way the connection between the representation (CAT file) and the instantiated application (search tool) is realized in the modified BeanBox is similar to the way the configuration language DARWIN is used in the REGIS project for interactive configuration management of distributed system. A DARWIN file is also evaluated during start-up and modifications to the running system are persistently stored by analyzing the structure of the system and translating it back to a DARWIN file (see Fosså and Sloman 1996).

4.2.5 Installation in the Offices of the State Representative Body

The new search tool was integrated into the POLITeam system (accessible via an icon on the virtual desktop) and presented to a group of four users (the system administrator, two clerks, and a department head) from the state representative body in a half-day workshop. The workshop's purpose was to verify that the set of components indeed covered the whole range of requirements, to evaluate the user interface for tailoring by letting the users solve small tailoring problems along the lines of "Add a link button to your search tool!“ and to familiarize the users with the new tool. Consequently, the results of the workshop mainly concerned usability-oriented issues not relating to the technical side of view that is of interest here.

After the usability issues had been resolved, the new search tool was installed in the offices of the state representative body on the workstations of three of the users (the department head's workstation was not equipped due to technical reasons). In order for the users to be able to share search tool compositions, all CAT files were saved in a shared directory indiscriminately accessible for everyone. The contents of this directory were selectable as menu items in the main window of the runtime and tailoring environment. Figure 4.5 depicts the architecture of the installation:

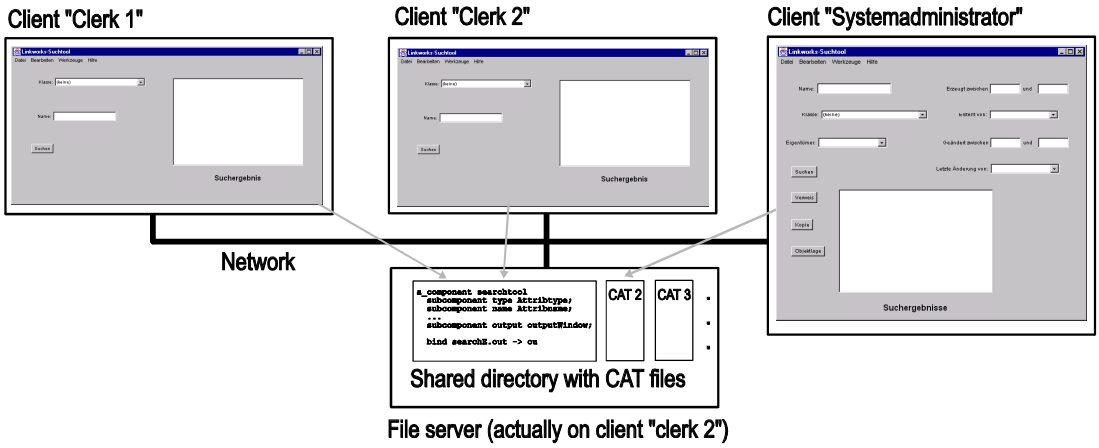


Figure 4.5: Installation of the search tool

In figure 4.5 the two clerks share the CAT file for a simple search tool, while the system administrator has selected a more powerful but complex composition. Changes made to the first CAT file shared by the two clerks affect both (after they restart their search tools).

The search tool installation remained in place one and a half month until the end of the POLITeam project. (Engelskirchen 2000) reports in detail on the workplace visits and interviews conducted during this period.

4.3 Results and Discussion

The case study proves that component-based tailorability is feasible and that the approach can be employed to make at least parts of groupware systems tailorable. The expectations formulated in the introduction were basically met:

First, the hierarchical composition permitted tailoring of the search tool on different levels of abstraction and complexity. The exchange of whole search tools (complex components) qualifies as tailoring on a high level, while exchanging only a switch component in order to support another grouping policy can be regarded as more low level tailoring. Furthermore, an additional level could be introduced by defining the search specification and the output part as two complex components, as shown in figure 4.6:

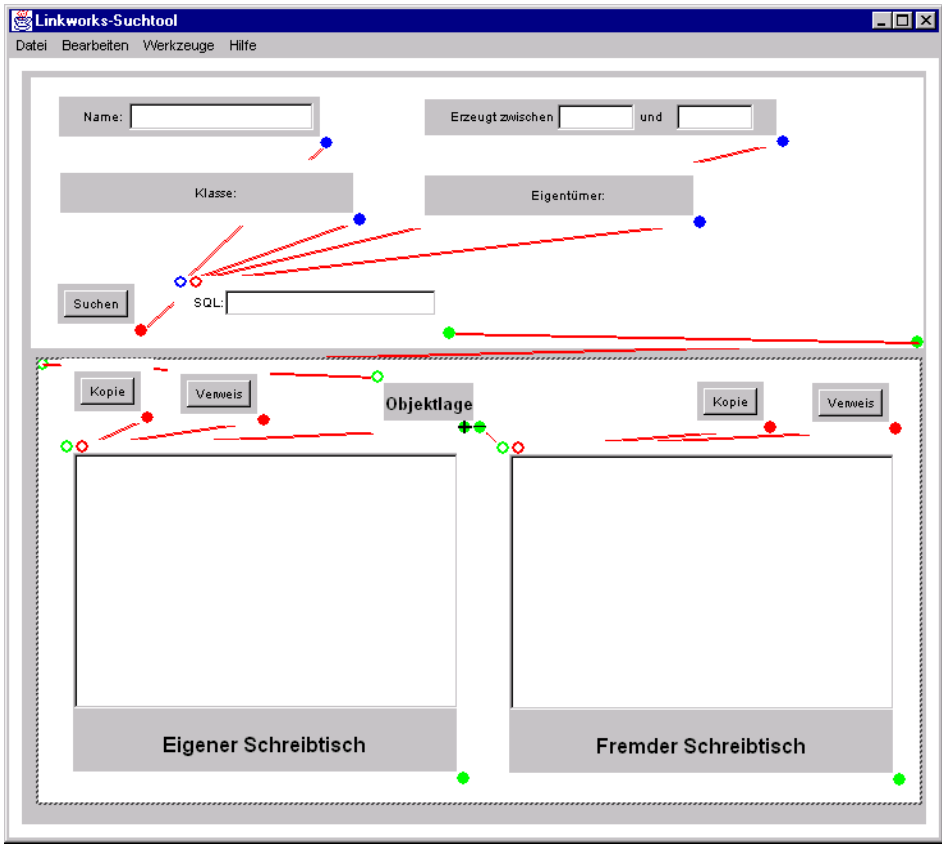


Figure 4.6: Hierarchical composition of a search tool

The white background behind the specification part and the output part indicates the fact that these are complex components. The two complex components are connected via a `ResultEvent` link. Note that the ports of the complex components are defined by connecting them to ports of the same polarity on the lower level. For instance, the specification part `ResultEvent` source port is connected to the respective port of the search engine.

Second, as shown in figure 4.5, complex components could rather straightforwardly be exchanged and shared between users (both clerks in figure 4.5 have specified the first CAT file as search tool composition). It was also possible to exchange only part of a search tool – for instance, the complex search specification component in figure 4.6.

The third point, the separation of tailoring functionality (as modified BEANBOX) and application functionality (as set of search tool components) was only achieved in principle. Theoretically, the runtime and tailoring environment could be (re-) used with any other set of components. Practically, the implementation took some shortcuts, e.g. by hard-coding the mapping of port colors to port types. However, these shortcuts could have been avoided quite straightforwardly with some more programming effort.

Fourth, the expectation of extensibility was met. During the workshop the users demanded additional functionality, e.g. a counter component for found documents that can be deployed in the place of an output window. The counter component was quickly implemented and simply added to the set of existing components. No other components had to be changed in order to accommodate these new requirements. However, in the next subsection a special

case (which did not originate from the workshop) will be discussed in which the extensibility of the system was restricted because of the event interaction style prescribed by the component model.

Apart from demonstrating feasibility, the purpose of the case study was to identify and explore open issues and questions raised by the approach. More specifically, the objective was to identify problems with the state-of-the-art solutions chosen for the four elements of component-based tailorability (component model, composition language, connection between representation and application, and the user interface for tailoring). Only the implementation of the second element – the CAT composition language – completely fulfilled its requirements. CAT is expressive enough to describe hierarchical compositions of components with typed, polarized and named ports. The problems with the component model and the component management provided by the modified BEANBOX are discussed in the following. Problems with the user interface for tailoring are out of the scope of this dissertation.

4.3.1 The JAVABEANS Component Model

The component model determines how (and whether at all) a specific decomposition of an application's functionality – i.e. a set of software components – can be implemented. Ideally, it should not force the programmer in any way to separate concerns that belong together or to compose concerns that should be separated. If it is not possible to implement a chosen component set, the resulting compromises are bound to negatively impact upon the degree and quality of the system's tailorability. Furthermore, if the implementation is possible, but awkward and inefficient due to restrictions caused by the component model, programmers will refrain from adopting the approach.

Concerning the implementation of a component's internal functionality, the JAVA programming language proved to be fully satisfactory. It contains a rich library of standard APIs that provide a lot of functionality relevant for groupware systems (e.g. for security, communication, or visualization). Furthermore, since a component instance does not have to incorporate a thread of control or even a whole process, the model supports rather fine-grained decompositions. A single component instance can contain several threads at one time and none at all at another time.

The implementation of the components' interactions was more problematic. Building on events as the only dynamically re-wireable interaction style of the component model, caused both problems mentioned above: a sometimes awkward and inefficient implementation of components and subtle, component-model dependent changes to the decomposition impacting upon the quality of the resulting tailorability of the system. Additional problems were caused by the fact that JAVABEANS does not support the concept of port names.

4.3.1.1 Problems Due to Event-Only Interaction

As an example for awkward and inefficient implementation, regard the way search attribute values are communicated from the attribute components to the search engine, as depicted on the left of figure 4.7 (the right side shows an alternative implementation which is discussed later):

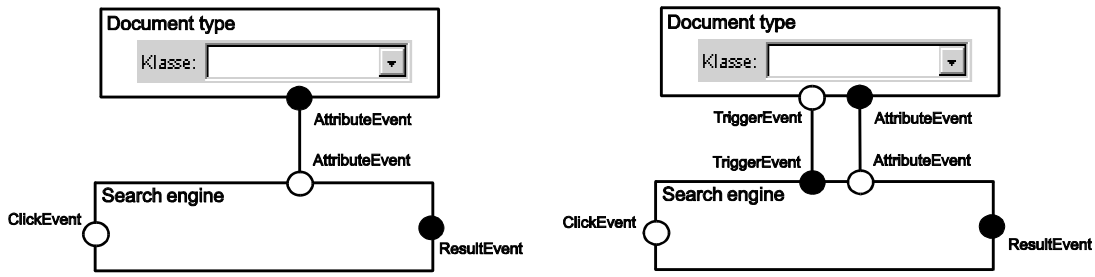


Figure 4.7: Two alternative implementations of the interaction between the attribute component *document type* and the *search engine*. Left: whenever the attribute changes, an `AttributeEvent` carrying the attribute value is sent to the search engine. Right: the search engine queries the attribute component to get the attribute value when it is needed (when a search is executed).

The search engine needs the attribute value only when a search is about to be executed. However, an `AttributeEvent` is sent to the search engine every time the value changes. Thus it can happen that the attribute components interact more frequently than necessary with the search engine component. In the search tool example, the additional cost associated with these unnecessary interactions does not have a serious impact. However, if a lot of data is transferred or if the interacting components are distributed, the costs can be considerable in terms of longer response times or network bandwidth used.

A workaround for this problem would be to design the search engine in a way that it queries the attribute components before it executes a search. In an event-based component-model like JAVABEANS, this could be done by sending an event to all attribute components, which triggers them to each send an `AttributeEvent` to the search engine, as depicted on the right of figure 4.7. However, this is not necessarily more efficient in terms of the number of events sent around the system, because each search now requires a constant number of event interactions that is twice the number of attribute components. This number can obviously be considerably higher than in the first solution, if, for instance, a search is specified only by the name attribute component (ideally, in the first solution only one event would be sent to the search engine in this case). Furthermore, the composition and implementation becomes more complex, because each attribute component now has to have two event ports. Thus both possible solutions have disadvantages that are caused by the fact that the JAVABEANS component model only offers events as a basis for (dynamically re-wireable) component interaction. (Note that these disadvantages concern the use of JAVABEANS for component-based tailorability, where one has to rely on events as the only compositional mechanism. When using JAVABEANS for development, the problem could easily be solved programmatically, i.e. by writing additional code that directly accesses the relevant data structures of the attribute components).

An example for an inappropriate decomposition of an application – caused by the event-only interaction style – is the placement of the link and copy functionality within the *output window* component. The actual core function of this component is to display a result list and permit the user the selection of an entry in that list. Result handling (i.e. creating a copy or link) is a separate concern. Still, the result handling functionality was implemented within the output window and not elsewhere (e.g. as part of the *link* and *copy button* component), because of the event interaction style. Placing the functionality within the link or copy button would have caused a problem similar to the awkward interaction between *attribute* components and *search engine*. The programmer (Won 1998) decided to move the functionality to the output window, which permits effective component interaction (only one event is sent per button click) and implementation. However, as a consequence the concerns of result display and result handling are not separated anymore. If, in the future, the users require a *move* button (which removes the found document from its current location and places it on a user's virtual desktop) the output window will have to be changed. This can be a major problem, if its source code is not available. If the result handling functionality were separated from the display functionality, it would be possible to implement the new button

together with the result handling functionality without having access to the source code of the *output window*. Thus the event interaction style of the component model impacted negatively upon the future extensibility of the system.

4.3.1.2 Problems Due to Missing Port Names

As an example for problems caused by the fact that the JAVABEANS component model does not support port names, regard the *location switch* component. The basic function of the *location switch* component is to receive results from the search engine and divide the results into two groups according to the location of document: either on the user's own desktop or elsewhere. Then it outputs the two groups of documents as `ResultEvent`s at two distinct `ResultEvent` source ports. However, because the component model does not support names, a component can only have a single `ResultEvent` source port. Consequently, a switch component cannot be implemented straightforwardly using the JAVABEANS component model. In the search tool example, the programmer developed a workaround using two different subtypes of the `ResultEventListener` interface: `PosResultEventListener` and `NegResultEventListener`, as depicted on the left of figure 4.8:

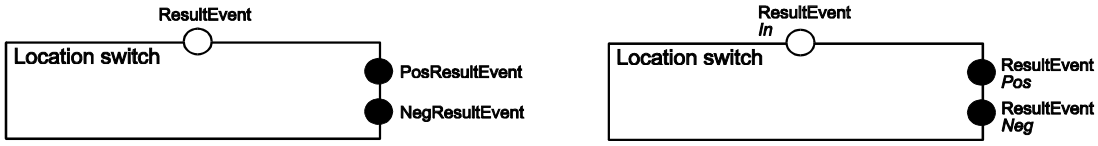


Figure 4.8: *Location switch* component. Left: implemented in JAVABEANS with typed ports only. Right: possible implementation, if typed and named ports are available

Consequently, it was possible to maintain two distinct lists of event listeners for each interface type. However, this solution has the severe drawback that for each named port of a component one has to specify an additional subtype interface, each resulting in an additional class file to be delivered with the component. Furthermore, the solution is conceptually unsound, because the specialized event listener interface does not denote a specialized event listener or event. The event listener connected to e.g. the `PosResultEvent` source port can again be a regular `ResultEvent` listener. Here the specialization is solely used to introduce some notion of port naming. This uncalled-for mixing of the concepts of names and types also impacts upon the runtime and tailoring environment that has to "know" (hard-coded) that even though the source port requires a `PosResultEvent` listener port, it can be connected to a – more general – `ResultEvent` listener port.

The discussion of port names so far has only considered the problems of not having names for event source ports. Similarly, a JAVABEANS component can implement a specific event listener interface only once, which makes it difficult to distinguish events from different sources. Either the distinction has to be made programmatically in the event handling method (which is obviously not desirable in the context of component-based tailorability) or one can employ adaptor objects. Adaptor objects implement event listeners of the appropriate type and are registered with the event source. They call the appropriate handling method of the actual, event receiving primary object of a component instance. By using a number of adapter objects, it is possible to handle events of different sources in different event handling methods, thus the receiving component does not have to be programmatically altered during composition. However, the adapter objects have to be generated and compiled during composition, which is something to be avoided here. An alternative suggested in the JAVABEANS specification (JavaSoft 1997, p. 35-37) is a generic adapter object, which is criticized by Szyperski as "*slow and unnatural*" (Szyperski 1998, p. 231). Furthermore, in both cases the question arises, how the runtime and tailoring environment identifies these methods of the receiving components that are supposed to be called from the adapter object, as no naming patterns are specified for them.

4.3.2 Component Management

The architecture of the runtime and tailoring environment as depicted in figure 4.4 realizes the connection between the representation of a composition and the structure of component instances by evaluating the CAT file at start-up time and building up the component instance structure. Runtime changes are directly applied to the component instances, while the representation gets updated only via a save command or when shutting down the tailored tool.

This mechanism is problematic, if complex components are shared, i.e. if they are instantiated more than once. Sharing can either apply to whole CAT files as depicted in figure 4.5, or within a CAT file. Conceptually, these two cases are the same, if one regards the three clients in figure 4.5 as one distributed application specified as a complex component instantiating three complete search tools as subcomponents. In the example, two of these search tool instances share the same complex component (CAT file 1) as specification. Within a single search tool specification, structurally equivalent parts (e.g. two output windows with a link and copy button each like in the lower part of figure 4.6) could be specified as one complex component that is instantiated twice or more, as depicted in figure 4.9:

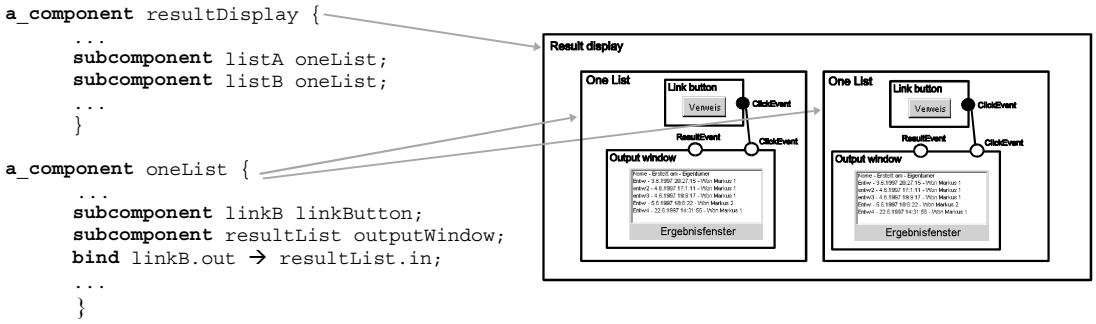


Figure 4.9: Sharing of a complex component's specification (left side as CAT file) by two instances (right side)

While the runtime and tailoring environment would correctly evaluate and instantiate a specification like this (left side of figure 4.9), it does not recognize structural equivalencies in the component instance structure (right side) and thus they would be lost when updating the CAT file again.

However, whole CAT files can be shared among users as depicted in figure 4.5. If an instance of a shared complex component specified by a CAT file is tailored (e.g. by adding a new subcomponent to the search tool of client 2 in figure 4.5), the change becomes effective immediately for that instance only. The other instances are affected after the instance structure of the changed instance has been saved and they are shut down and then re-instantiated according to the updated CAT-file. This raises two problems:

First, since tailoring operations become only partially effective during runtime, the whole instantiated application (namely the other search tool instances) might not be consistent with its persistent representation. Full consistency can only be achieved by shutting down and re-instantiating all affected search tools. The delays caused by these actions make the process of tailoring an application and validating the results quite complex and time-consuming. However, there are scenarios in which delayed effects can be quite useful, e.g. when two users share a CAT file that is being updated by one user, while the other user is currently executing a search.

Second, it is difficult to determine the scope of a tailoring operation. For instance, a user tailoring client 2 does not know who else shares the CAT file he is about to update. During the period the search tool was installed in the offices of the state representative body, the users were afraid of negatively affecting the work of others in this way. However, the

sharing of CAT files was also regarded as an important feature of the installation, e.g. for distributing a specific change to all users.

Both points demonstrate that the design of the connection between shared component representations and their instances in a runtime and tailoring environment impacts upon the quality of the system's tailorability.

4.3.3 Additional Requirements for Distributed Component-Based Tailorability

The search tool is only a local part of the LINKWORKS groupware platform. Even the *search engine* component connects to the document database via the LINKWORKS client on the local machine. While tailoring operations applied to one search tool instance have the potential to affect search tool instances on other machines (in the delayed fashion described in the last subsection), they achieve this through means outside the runtime and tailoring environment (the shared file system) with the complex consequences described above. Thus two issues have to be considered, when extending the applicability of the approach to whole groupware systems:

First, component instances as parts of the tailorable application itself have to be able to interact over machine boundaries as well as locally, because groupware systems are almost by nature distributed over a network including several remotely located physical processors. Furthermore, the representation of the system's composition has to include distribution aspects.

Second, tailoring operations as part of the runtime and tailoring environment have to be capable of affecting every part of the distributed groupware system. At the same time, the scope of these operations and the time when they take effect has to be under the control of the runtime and tailoring environment.

4.4 Summary and Resulting Questions

In this chapter the application of the component-based tailorability approach to a groupware tailoring problem was described and a number of problems with the state-of-the-art techniques employed were discussed:

First, while the JAVABEANS component model proved to be sufficient for the implementation of the search tool example, its characteristics nevertheless caused a number of awkward and unnatural workarounds. Especially JAVA events as the only dynamically re-wireable interaction style provided by the model impacted upon the efficiency of the application and caused the programmer difficulties in implementing the desired decomposition, which subsequently restricted the extensibility of the resulting system. This raises the question:

- *Which interaction style or set of styles should a component model support in order to permit the implementation of arbitrary decompositions?*

The second fundamental problem arising is the design of the connection between the representation of shared complex components and the application. While the runtime and tailoring environment itself did not support the sharing of representations of structurally equivalent parts of the composition, whole search tool compositions could be shared among users via the file system. This feature proved to be a mixed blessing: it was useful for propagating intended changes to multiple parts of the system; it was problematic because of the possibility of unintentional changes. Furthermore, because of the particularities of the connection approach employed, the changes caused by a tailoring operation became effective at a later time depending on factors not under the control of the system, thus possibly causing inconsistencies. This raises the question:

- *How does one design the connection between representation and application in a way that supports sharing of complex components, lets changes take effect in all instances of such a component immediately, and permits the control of a change's scope?*

The first two questions are addressed in chapter 5 and 6 employing mathematical models of interacting decomposable systems and hierarchical compositional representations. These models abstract from the specific programming language JAVA and the context of groupware systems.

The third problem is JAVA specific and concerns the fact that the JAVABEANS component model does neither support named ports nor explicit remote interaction. While the latter critique did not originate from the case study but from subsequent considerations concerning the application of the approach to whole distributed groupware systems, the former forced the programmer of the search tool example to develop a rather unsound workaround. Alternative solutions suggested in the JAVABEANS specification and the literature have similar disadvantages, thus raising the question:

- *How can the JAVABEANS component model be adapted to support the concept of port names and remote interaction?*

This problem is addressed in chapter 7, resulting in the FLEXIBEANS component model. This model also incorporates results from chapter 5, concerning the interaction styles to be supported.

Fourth, the runtime and tailoring environment employed in the search tool case was only suitable for local parts of groupware systems. Neither the compositional representation nor the architecture of the environment could support whole systems, which raises the question:

- *How does one design a runtime and tailoring environment that can serve as platform for component-based distributed groupware systems?*

Building on the mathematical model of sharing developed in chapter 6 and the FLEXIBEANS component model of chapter 7, this question is addressed in chapter 8 by presenting the EVOLVE system, which supports multi-user client-server applications distributed over a TCP/IP network like e.g. the Internet.

Finally, one has to substantiate and qualify the claim of wide applicability of the component-based tailorability approach. As stated in the introduction, one expectation was that the conceptual application independence of software components permits the reuse of concepts and even implemented functionality in a variety of cases. Chapter 9 presents a number of case studies employing the EVOLVE system as a basis for tailorable groupware and other systems.

Chapter 5

Component Interaction

5.1 Introduction

This chapter addresses the question "*Which interaction style or set of styles should a component model support in order to permit the implementation of arbitrary decompositions?*" First, the chapter demonstrates how component-based software systems executed in a multithreaded environment can be modeled in the framework of data space theory. The theory is then employed to model a number of component interaction primitives such as JAVA events, method calls, shared variables and others. Furthermore, the chapter discusses what it means to "simulate" the interaction protocol induced by a given decomposition with a limited set of interaction primitives. Subsequently this notion of simulation is formalized, providing distinct quality levels. Finally – as the main result of the chapter – a theorem is given which relates and qualifies the expressive power of two interaction primitives (JAVA events and shared variables) and demonstrates that a combination of both primitives permits simulation of arbitrary interactions at the highest level of simulation quality. The consequences of this result for decomposing an application's functionality are discussed.

5.2 Modeling Component-Based Software Systems

The goal of this chapter is to develop a framework for reasoning about component interaction primitives and to apply the framework to the question concerning the interaction primitives to be supported by a component model for component-based tailorability. As foundation for this endeavor one needs a programming language independent model of component-based software systems that can express a wide variety of different interaction styles. The next two subsections describe data space theory, how it can be used to model component-based software systems, and how a part of the search tool example appears in the model.

5.2.1 Data Space Theory and Software Components

Data space theory by (Cremers and Hibbard 1978) provides a model of an abstract state machine. The model has been employed to describe layered virtual machines, i.e. implementations of one virtual machine in another. (Cremers and Hibbard 1978) discuss, for example, the mapping of a functional language data space to that of an ALGOL-like language, and the implementation of the latter in a FORTRAN-like data space that, in turn, is mapped into an Assembler-like data space. The discussion in this chapter, however, builds on later work (Cremers and Hibbard 1986), which concerns the decomposition of data spaces in independent but communicating subspaces.

A data space essentially is a (possibly infinite) transition system, combined with an information structure defined as a set of functions mapping the current state of the system to values in application-dependent ranges. The set of functions can, informally, be interpreted as a set of attributes or variables, the functions giving the current value of the attribute or variable for a certain state of the system. The set of state functions has to be complete and orthogonal, in the sense that – modulo constants – the attribute values determine the system state and that the attributes are independent of each other.

DEFINITION 5.1: (X,F,p) *is a* data space, *iff.*

(X,p) *is a transition system, with X being a (possibly infinite) set of states and* $p \subseteq X \times X$. *F is an information structure on X, i.e. a (possibly infinite) set of functions totally defined over X with arbitrary ranges, and*

(1) $\forall_{x,y \in X}(\forall_{f \in F}\ f(x) = f(y)) \Rightarrow x = y$ 'Completeness

(2) $\forall_{e \in F \to X}\exists_{x \in X}\forall_{f \in F} f(x) = f(e(f))$ 'Orthogonality

(Notation: $F \to X$ denotes the set of total functions mapping each function f of F onto a state in X. An element e of $F \to X$ can thus be interpreted as a particular choice of argument for each function f. Therefore, condition 2 implies that for every choice e of arguments, a state x can be found which lets all functions of F assume the chosen values. Thus, the functions of F are independent w.r.t. each other)

A feature which makes data spaces attractive for modeling decomposable software is that one can focus on certain partitions of a data space, determined by a subset F' of the information structure F. Regard the following equivalence relation:

$$x = y \;||\; F' :\Leftrightarrow \forall_{f \in F'} f(x) = f(y)$$

Employing this relation, one can construct a new data space $([X]_{F'},[F'],[p]_{F'})$ using the equivalency classes as new states:

$[x]_{F'} := \{y \mid y = x \;||\; F' \wedge x \in X\}$ 'equivalence class of x, $x \in X$

$[X]_{F'} := \{[x]_{F'} \mid x \in X\}$ 'set of equivalence classes

$[F'] := F'$ *defined over* $[X]_{F'}$ 'functions over $[X]_{F'}$

$[p]_{F'} := \{([x]_{F'},[y]_{F'}) \mid (x,y) \in p)\}$ 'transition relation w.r.t. F'

Note that the domain of the functions in $[F']$ is not X but $[X]_{F'}$, with $f([x]_{F'}) = f(x)$. One can show that $([X]_{F'},[F'],[p]_{F'})$ is indeed a data space:

PROPOSITION 5.1: *Let $D = (X,F,p)$ be a data space and $F' \subseteq F$. Then $D' = ([X]_{F'},[F'],[p]_{F'})$ is a data space.*

PROOF: *In order to prove proposition* 5.1, *one has to show that* $[F']$ *is complete and orthogonal w.r.t.* $[X]_{F'}$:

(1) *Completeness*

Given $[x]_{F'}$ and $[y]_{F'}$, $\forall_{f \in [F']} f([x]_{F'}) = f(x) = f(y) = f([y]_{F'}) \Rightarrow [x]_{F'} = [y]_{F'}$.

(2) *Orthogonality*

Given $e:[F'] \rightarrow [X]_{F'}$, *assume* $\forall_{[x]_{F'} \in [X]_{F'}} \exists_{i \in [F']} i([x]_{F'}) \neq i(e(i))$

Take any $g:F \rightarrow X$ *with*

$g(f) \in e(i)$ *if* $\exists_{i \in [F']} \forall_{[x]_{F'} \in [X]_{F'}} i([x]_{F'}) = f(x)$, *else with*

$g(f) \in X$.

By orthogonality of D, $\exists_{z \in X} \forall_{f \in F} f(z) = f(g(f))$.

By definition of D' and g, $i([z]_{F'})) = f(z) = f(g(f)) = i(e(i))$.

This is a contradiction to the assumption. ■

(Cremers and Hibbard 1986) demonstrate how data space theory can be employed to describe the decomposition of a data space into several independent but communicating subspaces. The decomposition is described by a set of subsets of the information structure of the data space with additional constraints concerning the independence of the components:

DEFINITION 5.2: *A* splitting *of* (X,F,p) *is a set* $\{F_1 , \ldots , F_k\}$ *of subsets of F with the following property*:

Whenever $(x,y) \in p$ *there exists* (*at least one*) *i such that, given v and* $w \in X$:

(1) $y = x \,||\, F - F_i$ *'no change outside* F_i

(2) $v = x \,||\, F_i \wedge w = y \,||\, F_i \wedge v = w \,||\, F - F_i \Rightarrow (v,w) \in p$ *'no dependencies outside* F_i

In this context, the pair (x,y) *is called* i-move.

Condition one implies that an *i*-move only changes these aspects of the whole system's state that are modeled by the functions in F_i. Condition two implies that an *i*-move can be executed independently of all aspects of the system's state that are modeled by the functions outside F_i.

The following briefly demonstrates how the elements of data space theory relate to the concepts of hierarchically component-based software systems as defined in chapter 2:

- Each data space induced by a F_i of a splitting S is *independent* in the sense that its behavior only depends and impacts on the contents of its own information structure (see definition 5.2).
- Data spaces can be *composed hierarchically*, because proposition 5.1 implies that all $([X]_{F_i},[F_i],[p]_{F_i})$ induced by the splitting are again data spaces (called subspaces of D).
- Data spaces *interact* (via overlapping information structures).

Figure 5.1 shows how data space theory reflects the concepts of components, ports, and composition:

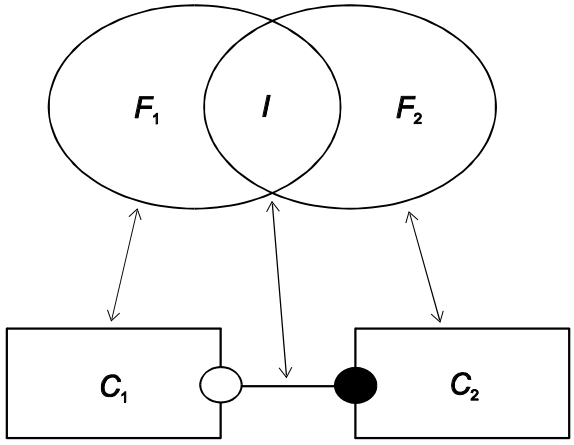


Figure 5.1: Relationship between data spaces and software components

The upper part of figure 5.1 shows the (overlapping) sets F_1 and F_2 of a splitting $\{F_1;F_2\}$ of (X,F,p). Set I contains the overlap of the two information structures. Thus the subspaces induced by F_1 and F_2 respectively represent components C_1 and C_2 in the model (lower part of figure 5.1), the subspace I of each subspace represents the port of each component, and (X,F,p) represents the composition of the two components.

5.2.2 Example

Regard the search tool component set described in chapter 4 (table 4.1). It contains a *document type* component for specifying the type of document to search for. This component interacts via an `AttributeEvent` with the *search engine* component that is triggered by a `ClickEvent` from the *search button* component and provides the search results at a `ResultEvent` source port. In this subsection, two components (the *document type* component and the *search engine* component) and their interaction are modeled as data spaces. In chapter 4 their interaction was used to demonstrate the problems with the JAVA event interaction style.

The *document type* component is modeled as data space $D_1 = (X_1,F_1,p_1)$. Its state is characterized by three elements:

- the state of the drop down box containing the list of document types, i.e. which type is currently selected. The state of the drop down box is changed by the users via the GUI (in the model the user action appears as a spontaneous state transition). For simplicity, it is assumed here that the drop down box can only have the two states $\{t_1;t_2\}$. This means that its list only contains two entries.
- the current state of the attribute event interaction: $\{\#;t_1;t_2\}$, with # meaning "ready to interact", t_1 meaning "currently sending event carrying document type t_1", and t_2 meaning "currently sending event carrying document type t_2".
- a flag which indicates whether the user's choice has already been sent to the search engine: {0;1}, with 0 meaning "not sent" and 1 meaning "sent".

Thus one gets $F_1 = \{$ *drop_down_box ; attribute_event ; flag* $\}$ with

$$\begin{aligned} \textit{drop\_down\_box}&: X_1 \rightarrow \{t_1;t_2\} \\ \textit{attribute\_event}&: X_1 \rightarrow \{\#;t_1;t_2\} \\ \textit{flag}&: X_1 \rightarrow \{0;1\} \end{aligned}$$

and $X_1 = \{t_1;t_2\}\times\{\#;t_1;t_2\}\times\{0;1\}$. Defining X_1 as the Cartesian product of the ranges of F_1's functions trivially fulfills the conditions of completeness and orthogonality of the data space definition, if the functions are defined to project their part of the state. The state transitions defining p_1 are shown as black arrows at the top of figure 5.2. State transitions caused by the user are gray arrows and state transitions caused by other component instances are dotted

arrows. The start state $(t_1,\#,1\}$ is underlined. Assuming the user selects t_2 in the drop down list, the component instance is brought into state $\{t_2,\#,0\}$ indicating that t_2 is selected, but not yet sent (communicated to the search engine). Then the event interaction is started by sending an event carrying type t_2 as indicated by state $\{t_2,t_2,1\}$. The flag now indicates that the user's choice has been sent. After the event has been processed by the receiver (the *search engine* component instance), the state of the event interaction is brought back to "ready to interact" by the receiving component instance. If the user switches back to t_1, the process is equivalent.

The *search engine* component is modeled as data space $D_2 = (X_2,F_2,p_2)$. Its state is characterized by five elements:

- the state of the click event interaction: $\{\#;s\}$, with # meaning "ready to interact" and *s* meaning "searching".
- the state of the attribute event interaction: $\{\#;t_1;t_2\}$.
- the state of the result event interaction: $\{\#;r_1;...;r_n\}$, with # meaning "ready to interact" and r_1 to r_n representing the space of possible results.
- a flag indicating whether the results have already been sent: {0;1}.
- a store for the current document type: $\{t_1;t_2\}$.

F_2 and X_2 are constructed in the same fashion as above. The state transitions defining p_2 are depicted in figure 5.2:

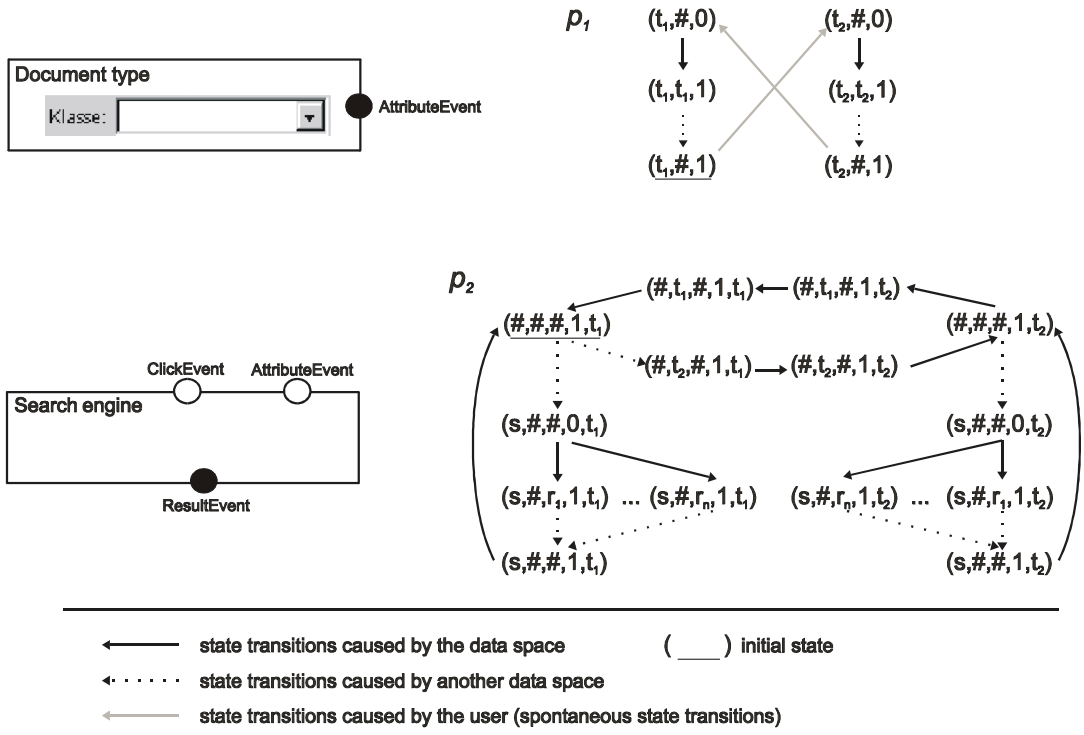


Figure 5.2: Modeling the *document type* and *search engine* components as data spaces

The start state is $(\#,\#,\#,1,t_1)$. If an `AttributeEvent` carrying type t_2 is sent to the *search engine* (from the *document type* component), it brings the search engine to state $(\#,t_2,\#,1,t_1)$. The store is adjusted accordingly (state $(\#,t_2,\#,1,t_2)$) and the `AttributeEvent` interaction is returned to its initial state $(\#,\#,\#,1,t_2)$.

If the *search engine* receives a `ClickEvent` (from the *search button* component), it is brought to state $(s,\#,\#,0,t_2)$ also indicating that no search results have been sent out yet. The subsequent interaction with the LINKWORKS document database is not modeled here; it is

simply assumed that the search engine gets the search result r_i with $1 \leq i \leq n$ from somewhere and then sends out a `ResultEvent`: $(s,\#,r_i,1,t_2)$. The result event receiver (a *switch* or an *output window* component – which are both not modeled here) then returns the `ResultEvent` interaction going to state $(s,\#,\#,1,t_2)$. The search engine then returns the `ClickEvent` interaction going to state $(\#,\#,\#,1,t_2)$.

So far, only the modeling of single components has been discussed. By indicating state transitions caused by other components (the dotted arrows in figure 5.2) it was already implied that parts of a component instance's state have to be shared with other component instances in order for interaction to take place. Consequently, the composition of the two components above is modeled by a data space $D = (X,F,p)$ with $F = \{flag_1;\ click\_event;\ result\_event;\ store\ attribute\_event;\ drop\_down\_box;\ flag_2\}$, X the Cartesian product of F's functions' ranges (in the order in which the elements of F are listed above), and a splitting $S = \{\{drop\_down\_box;\ attribute\_event;\ flag_2\};\ \{click\_event;\ attribute\_event;\ result\_event;\ store;\ flag_1\}\}$. The two sets of S intersect in $\{attribute\_event\}$. The state transitions defining p are derived from p_1 and p_2 by:

$$((a_1,b_1,c_1,d_1,e_1,f_1,g_1),(a_2,b_2,c_2,d_2,e_2,f_2,g_2)) \in p :\Leftrightarrow ((e_1,f_1,g_1),(e_2,f_2,g_2)) \in p_1 \vee ((a_1,b_1,c_1,d_1,e_1),(a_2,b_2,c_2,d_2,e_2)) \in p_2$$

Consequently, the subspace of D induced by $\{drop\_down\_box;\ attribute\_event;\ flag_1\}$ is equal to D_1 up to isomorphism and the subspace induced by $\{click\_event;\ attribute\_event;\ result\_event;\ store;\ flag_2\}$ is equal to D_2 up to isomorphism.

Because of the definition of p, each state transition in p can be attributed to one of the subspaces D_1 or D_2. Both subspaces interact via the shared function *attribute_event*. D_1 can initiate the event interaction by executing state transition $x \rightarrow x'$ with $attribute\_event(x) = \#$ and $attribute\_event(x') = drop\_down\_box(x)$. After processing the event with a number of internal state transitions (e.g. $x' \rightarrow x''$ with $store(x'') = attribute\_event(x')$), D_2 returns the event interaction, i.e. it goes to state x''' with $attribute\_event(x''') = \#$.

The other components and their interactions can be modeled in a similar fashion.

5.3 Components and Multithreaded Execution

The example demonstrates how a component-based software system can be modeled as a data space with several interacting subspaces representing single component instances. A state transition (x,y) can always be attributed to a specific component instance i (as an i-move, see definition 5.2) and can be executed, whenever the data space is in state x, resp. when the subspace i is in state $[x]_{F_i}$. Modeling component instances in this fashion captures the essential component property of encapsulation (see definition 5.2: "*no change outside* [...] *no dependencies outside* [the component ...]"). However, it assumes that a component instance can always perform a state transition. Thus a component is implicitly considered an active entity.

For many applications this view is fully sufficient, because component instances often are essentially that: active entities. In the field of distributed systems, for example, components are usually treated synonymously to processes (see e.g. Magee et al. 1995), because they have to run on different physical machines. In the field of concurrent processing, components are – by definition – active entities that can execute concurrently (see e.g. Gibbons and Rytter 1988). Or – approaching the hardware-level of a machine – algorithms implemented directly by physical circuitry on a chip can rely on the electrical properties of the wafer for "activation".

However, for the investigation of the problems raised by the search tool example, a more differentiated model is required. There are component models (for instance, the JAVABEANS component model) that do not regard component instances and processes (or rather: threads)

as equal. Taking this perspective, component instances can be thought of essentially as stateful entities with associated code. Only if the associated code is *traversed* by a thread of control, the component instance is able to change its state and to interact with other component instances. The code can be traversed by multiple threads at the same time. If no thread is executing the associated code of a component instance, the instance is said to be inactive.

In the context of this discussion, a *thread of control* is an operating system concept; it is basically a virtual processor that executes code. The operating system usually maps several threads to one or more physical processor. The difference between threads and processes is that multiple threads can be active in the same address space, while processes have their own address space. A more detailed discussion of processes and threads in the context of the JAVA programming language can be found in (Gosling et al. 1996).

Taking the object-oriented language JAVA as an example, in the component model JAVABEANS a component instance is a single object or a set of objects. The state of a component instance is the state of the object(s). The associated code is given by the set of methods associated with the object(s). When executing a JAVABEANS-based program, methods of the component instance objects are called – thereby activating the component instance (i.e. a thread of control enters the realm of a component instance). A method itself can call other methods of other objects constituting component instances. In this fashion, one or more threads traverse the network of component instances and thus execute the component-based software system. Since, for instance, the search tool application is executed by a single thread of control (which enters the application when the user interacts with it via buttons or combo boxes like the document type component in figure 5.2), it is vitally important that the interaction between component instances not only results in a possible exchange of information but also in a possible transfer of the control flow (i.e. the thread of control). This transfer of control flow is thus an essential property of component interaction. Consequently, the model developed here has to take control flow transfer into account.

Separating the concepts of component and thread of control requires an explicit representation of threads and their "movements" between component instances in the model. For the purpose of the discussion in this chapter, it is not necessary to model threads in every detail. Since the focus lies on component interaction, it is, for instance, not required to model the creation and destruction of threads and the exact mechanisms of thread synchronization. These aspects are obviously important, but here one can abstract from them.

Given a data space $D = (X,F,p)$ and an associated splitting $S = \{F_1 , \ldots , F_k\}$, one assumes an initial (and constant) number of threads which traverse the component instance structure. Let $s \in X$ be the initial state of the data space D. The threads and their "host"-component instances (subspaces) in state s are given by the function

$$t : C \rightarrow \{0,1,...\} ,$$

where $C := \{1,..,k\}$ is the index set of the splitting S. Thus $t(i)$ denotes the number of threads initially active in subspace (component instance) i.

The function t only defines the state of affairs at the beginning of an execution. Each executed state transition can potentially change the way the threads are assigned to subspaces. The way the control flows are passed between component instances is usually determined by the code associated with each instance. Here, the *control flow passing behavior* of the subspaces is given by the function

$$a : C \times P \rightarrow C .$$

If state transition $(x,y) \in P$ is executed by subspace i (i.e. it is an i-move), a single control flow is passed from subspace i to subspace $j = a(i,(x,y))$. If $i = j$, obviously no control flow is passed. Function a can model, for instance, how in the first step of a method call, the control flow is transferred from client to server and how in a later step it is returned (together with the results) to the client (the next will section discuss this in more detail).

Given the functions t and a and the initial state s for a data space D and a splitting S, one can define the notion of an *executed data space* D_{sat}, in which only these components can execute a state transition that are currently traversed by a control flow:

DEFINITION 5.3: *Given a data space $D = (X,F,p)$, a splitting $S = \{F_1 , \dots , F_k\}$, functions t and a, and an initial state s, an* executed data space *D_{sat} is a data space (X',F',p') with the following properties:*

$$X' := X \times \{0,1,...\}^k$$

$$F' := F \cup \{t_1,..t_k\}$$

$$p' := \{((x,n_1,...,n_k),(x',n'_1,...,n'_k)) \mid (x,x') \in p \wedge \exists_{i,j \in C} : (\forall_{l \notin \{i;j\}} : n_l = n'_l) \wedge a((x,x'),i) = j \wedge n_i > 0 \wedge ((i \neq j \wedge n_i - n'_i = n'_j - n_j = 1) \vee (i = j \wedge n_i = n'_i \wedge n_j = n'_j))\}$$

Furthermore, there exists a state s' in X' that corresponds to the initial state of the original data space and the initial distribution of control flows, that is:

$$s' = (s,t(1),...,t(k)) .$$

The functions $\{t_1,..t_k\}$ that are added to F are given by $t_i : X' \rightarrow \{0;1;\dots\}$, with

$$\forall_{x \in X} : t_i((x,n_1,...n_k) := n_i .$$

Definition 5.3 essentially represents a restriction on the computations data space D can perform. A state transition (x,x') of D can only be executed in D_{sat}, if the executing subspace i is currently traversed by a control flow ($n_i > 0$). Furthermore, if the control flow passing behavior as given by $a((x,x'),i)$ calls for a transfer of a control flow to component j, the current state of D_{sat} is modified accordingly.

Summing up, this section provides a simple model of the execution of a component-based software system in a potentially multithreaded environment. The possibly limited availability of processing power in the form of threads of control restricts the way a system can be executed. In the following, interaction between subspaces is investigated in the context of an executed data space in order to capture this essential property of component interaction.

5.4 Component Interaction

Since the question addressed in this chapter focuses on interaction, the formal model needs a construct that captures relevant aspects of interaction and permits reasoning about different types of interactions. In this section the notions of *interaction spaces* and *interaction traces* are developed, providing a static and a dynamic view on interaction. First interaction spaces are derived from the concepts of data space, splitting, and executed data space. Then interaction traces are defined based on interaction spaces.

5.4.1 Interaction Spaces

Regard an executed data space D_{sat} with $D = (X,F,p)$ and $S = \{F_1;F_2\}$. The subspaces $([X]_{F_1},[F_1],[p]_{F_1})$ and $([X]_{F_2},[F_2],[p]_{F_2})$ interact, if $F_1 \cap F_2 \neq \emptyset$. According to proposition 5.1, $([X]_I,[I],[p]_I)$ with $I = F_1 \cap F_2$ is again a data space. Data space $([X]_I,[I],[p]_I)$ represents the interaction between the two subspaces of D. Regard the following definition:

DEFINITION 5.4: *Given an executed data space D_{sat} with $D = (X,F,p)$, $S = \{F_1;F_2\}$ and $I = F_1 \cap F_2$, its* interaction space *IS is a 5-tuple* $([X]'_I,[p]_I,[s]_I,[a]_I,l)$,

with $[X]'_I := \{ x \mid x \in [X]_I \wedge \exists_{y \in [X]_I} ((x,y) \in [p]_I \vee (y,x) \in [p]_I)\}$

$[a]_I : [p]_I \times C \rightarrow C$, *with* $[a]_I(([(x,y)]_I,i)) := a((x,y),i))$,

$l : [p]_I \rightarrow \{\{1\};\{2\};\{1;2\}\}$, *with* $\{i\} \in l([(x,y)]_I) :\Leftrightarrow (x,y)$ *is i-move, and*

$[(x,y)]_I := \{(v,w) \mid v \in [x]_I \wedge w \in [y]_I\}$.

Note that since $[X]'_I$ only includes states that are beginning or end of a transition, $([X]'_I,[I],[p]_I)$ is not necessarily a data space anymore. The notion of an interaction space given here only allows for transitions that deterministically pass the control flow; or formally: given (x,y), $(v,w) \in (t,u) \in [p]_I$ and $i \in C$:

$$a((x,y),i) = a((v,w),i) .$$

This means that each transition (t,u) within the interaction space, i.e. $(t,u) \in [p]_I$ either always or never passes the control flow to the respective other component. A consequence of this restriction is the unambiguous definition of $[a]_I$ in definition 5.4. Applying this restriction in definition 5.4 is a rather severe decision, which is defendable, because the usefulness of interaction primitives that are non-deterministic w.r.t. control flow passing is questionable (e.g. a method call always passes the control flow to the other component and back again, regardless of the state of each component). However, the reader should keep the existence of this restriction in mind, when modeling new interaction mechanisms.

Table 5.1 shows a number of different example interaction spaces modeling common interaction primitives. The first column shows a graphical representation of the respective interaction space, which is derived in the following way: a directed graph representation with nodes X and edges p is used. A row of three dots indicates other states of the same subset of X. The initial state is #, the labeling is indicated by the shading of an edge (*black* represents component 1, *gray* represents component 2). If $l(x,y) = \{1;2\}$, the representation shows two edges of different colors. If a transition causes the control flow to be passed ($a((x,y),i) \neq i$), a little circle is added to the respective edge.

The different interaction spaces given in table 5.1 show how the model supports the representation of a variety of interaction styles. The next subsection addresses the second requirement: modeling the combination of interaction spaces yielding more complex interaction styles.

5.4.2 Direct Interaction of More Than Two Components

Definition 5.4 covers only component interaction involving two components, resp. subspaces. However, data spaces theory also permits direct interaction of three or more subspaces (e.g. $F_1 \cap F_2 \cap F_3 \neq \emptyset$). These cases are not discussed here, because (Cremers and Hibbard 1986) demonstrate, that every data space with a direct interaction of n subspaces can be replaced by a (possible refined and thus more complex) data space in which every interaction only involves two subspaces. Regard the following theorem (Cremers and Hibbard 1986, p. 394):

THEOREM OF CREMERS AND HIBBARD: *Let $D = (X,F,p)$ be a data space and $\{F_1,...,F_k\}$ a splitting of F. There exists a micro space $D' = (X',F',p')$ of D and a splitting $\{F'_1,...,F'_k\}$ of F' such that*

(*a*) *$F'_i \cap F'_j \cap F'_l$ is empty if i, j, l are distinct,*

(*b*) *the subspace associated with F'_i is a micro space of the subspace associated with F_i.*

In this context, a micro space D' of a data space D is a data space that associates each state in D with a group of states in D'. Thus D' refines D and is able to perform the same

computations. The formal definition can be found in (Cremers and Hibbard 1986, p. 393-394). Condition (a) indicates that no three subspaces share an interaction space.

While the theorem of Cremers and Hibbard guarantees that each subspace of D' performs the same computations as the associated subspace in D', the complexity is potentially higher, i.e. the subspaces can need more state transitions for computation and interaction. However, this is irrelevant for the discussion in this chapter. The question of complexity becomes more relevant when discussing a concrete implementation of component model. The presentation of the FLEXIBEANS model in chapter 7 touches upon this subject.

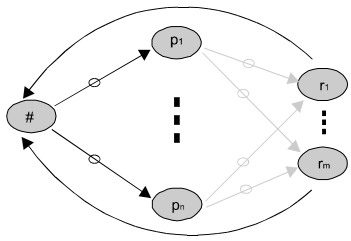
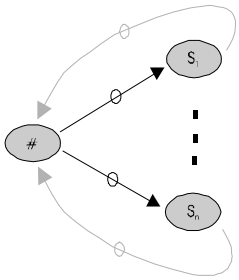
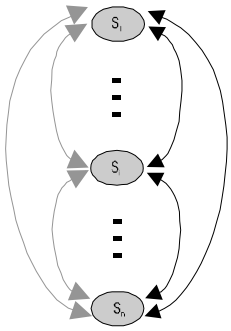
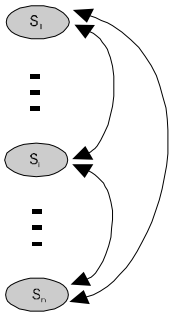
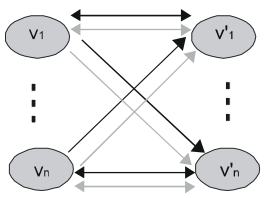
Method call-like IS 	$IS_{method\text{-}call}(P,R) :=$ (X,p,s,a,l) $X := \{\#\} \cup P \cup R$ with $P \neq \emptyset$, $R \neq \emptyset$ $p := \{init\} \times P \cup P \times R \cup R \times \{init\}$ $s := \{init\}$ $a(((x,y),1)) := 2$ *if* $x = \{init\} \wedge y = P$ $:= 1$ *if* $x = P \wedge y = \{init\}$ $a(((x,y),2)) := 1$ *if* $x = P \wedge y = R$ $a(((x,y),.)) := 1$, *else.* $l((x,y)) := 1$ *if* $(x = \{init\} \wedge y = P) \vee (x = R \wedge y = \{init\})$ $:= 2$, *else.*	A common (call by value) **method call** of a client (black) transfers the control flow together with the parameters to the server (gray) that processes the request and returns the result and the control flow to the client. The client then consumes the result (i.e. assigns it to a variable) and returns the interaction into its initial state #.
Java event-like IS 	$IS_{java\text{-}event}(S) :=$ (X,p,s,a,l) $X := \{\#\} \cup S$ *with* $S \neq \emptyset$ $p := \{\#\} \times S \cup S \times \{\#\}$ $s := \{\#\}$ $a(((x,y),i)) := 3\text{-}i$ $l((x,y)) := \{1\} \Leftrightarrow x = \{\#\} \wedge y \in S$ $:= \{2\} \Leftrightarrow y = \{\#\} \wedge x \in S$	**JAVA events** (see JavaSoft 1997) are implemented as void method calls, transfering some stateful entity (the event object) and the control flow. Starting from the initial state #, the event source (black) takes the interaction space in a state representing the state of the event object. After processing the event, the other event listener (gray) returns the control flow and brings the interaction space back to #.
Shared variable-like IS 	$IS_{shared\text{-}variable}(S) :=$ (X,p,s,a,l) $X := S$ *with* $\lvert S \rvert > 1$ $p := S \times S$ $s \in S$ $a(((x,y),i)) := i$ $l((x,y)) := \{1;2\}$	**Blackboards** (or **shared variables**) are another common component interaction paradigm (see e.g. Nierstrasz and Tsichritzis 1995). In this interaction style, all participating components have access to a shared state space that they can read and manipulate at will. However, writing or reading the shared variable does not induce a transfer of control flow as in the event or method-call like interaction.
Observable-like IS 	$IS_{observable}(S) :=$ (X,p,s,a,l) *like Shared variable, except for*: $l((x,y)) := \{1\}$	**Observables** are a special form of blackboards. Only one component can read and write, while other components can only read, i.e. observe the changes made by the first component (contrast: the observable design pattern in Gamma et al. 1995). Again, the flow of control is not transferred.
Synchro cell-like IS 	$IS_{synchro\text{-}cell}(V) :=$ (X,p,s,a,l) $V' := \{v' \mid v \in V\}$, *with* $\lvert V \rvert > 1$ $X := V \cup V'$ $p := V \times V' \cup \{(v,v') \mid v \in V\}$ $s \in V$ $a(((x,y),i)) := i$ (possibly not specified) $l((x,y)) := \{1;2\}$	**Synchro cells** (Cremers and Hibbard 1985) are similar to shared variables except that they contain an additional flag that indicates whether the variable should be read or set. This mechanism permits synchronized access to a shared variable, e.g. in order to enforce data flow.

Table 5.1: Common interaction primitives modeled as interaction spaces.

5.4.3 Combination and Inversion of Interaction Spaces

Two components might use several (perhaps even different) interaction primitives for interaction in order to address more complex synchronization and communication requirements. As observed earlier, the JAVABEANS component model only offers events as (runtime re-"wireable") interaction primitives. If two JAVABEANS need to interact in a method call-like style, two events have to be used: one for "firing" the parameters to the server and the other for returning the results to the client. In this case, two event interaction primitives are combined in order to yield a more complex interaction style. Keeping this example in mind, regard the following definition:

DEFINITION 5.5: *Given two interaction spaces* $IS_1 := (X_1,p_1,s_1,a_1,l_1)$ *and* $IS_2 := (X_2,p_2,s_2,a_2,l_2)$*, their* combination $IS_1 \otimes IS_2$ *is defined by* (X,p,s,a,l) *with*:

$X := X_1 \times X_2$

$p := \{((x,y),(v,w)) \mid [(x=v) \wedge ((y,w) \in p_2)] \vee [(y=w) \wedge ((x,v) \in p_1)]\}$

$s := (s_1,s_2)$

$$a((((x,y),(v,w)),i)) := a(((x,v),i)) \quad \text{if } y = w,$$
$$:= a(((y,w),i)) \quad \text{if } x = v,$$
$$l(((x,y),(v,w))) := l((x,v)) \quad \text{if } y = w,$$
$$:= l((y,w)) \quad \text{if } x = v.$$

PROPOSITION 5.2: *The operator* $\otimes$ *is associative and commutative up to isomorphism:*

$$IS_1 \otimes IS_2 =_{iso} IS_2 \otimes IS_1 \text{ and } (IS_1 \otimes IS_2) \otimes IS_3 =_{iso} IS_1 \otimes (IS_2 \otimes IS_3)$$

PROOF: *Employing the isomorphism* $m_c((x,y)) := (y,x)$ *for commutativity and* $m_a(((x,y),z)) := (x,(y,z))$ *for associativity, in both cases the resulting interaction spaces are structurally equivalent w.r.t. p, s, a, and l.* ■

In order to model the above example of simulating a method call with two combined events, one needs the following "role-swapping" operator that – informally – swaps the roles of component F1 and F_2 in the interaction, or – formally – is given by:

DEFINITION 5.6: *Given an interaction space* $IS = (X,p,s,a,l)$*, its* inverse $\downarrow IS$ *is defined by* (X,p,s,a',l') *with*:

$$l'((x,y)) := \{1\} \quad \text{if } l((x,y)) = \{2\},$$
$$:= \{2\} \quad \text{if } l((x,y)) = \{1\},$$
$$:= \{1;2\} \quad \text{else.}$$

$a'(((x,y),3 - i)) := 3 - a(((x,y),i))$ with $i \in \{1;2\}$.

The combination of the two event primitives in the method call simulation example is modeled by the following expression:

$IS_{simulated\text{-}method\text{-}call}(P,R) := IS_{java\text{-}event}(P) \otimes \downarrow IS_{java\text{-}event}(R)$.

At the bottom of figure 5.3, the combined interaction space $IS_{simulated\text{-}method\text{-}call}(P,R)$ is shown in graphical notation:

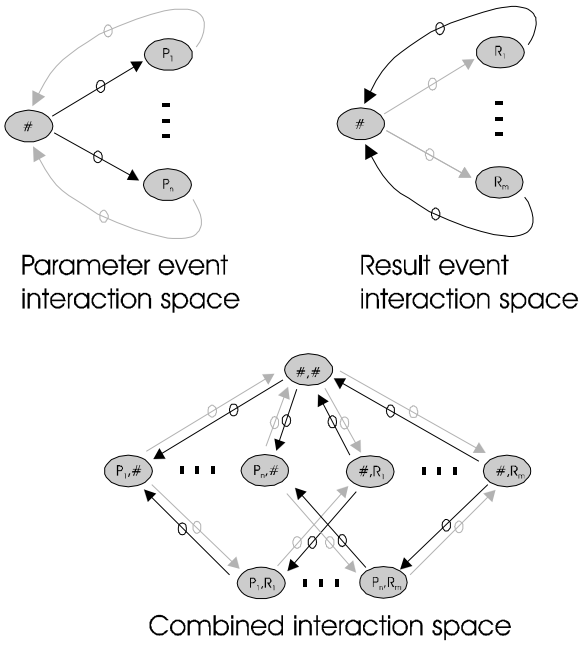


Figure 5.3: Simulation of a method call using the combination of two events

Within $IS_{simulated\text{-}method\text{-}call}(P,R)$ a method call is simulated in the way described above. Actually, there are three possible ways of first communicating the parameters to the server and then the result to the client, given by the following three traces through the interaction graph:

1. $(\#,\#) \rightarrow (p_i,\#) \rightarrow (p_i,r_j) \rightarrow (p_i,\#) \rightarrow (\#,\#)$
2. $(\#,\#) \rightarrow (p_i,\#) \rightarrow (\#,\#) \rightarrow (\#,r_j) \rightarrow (\#,\#)$
3. $(\#,\#) \rightarrow (p_i,\#) \rightarrow (p_i,r_j) \rightarrow (\#,r_j) \rightarrow (\#,\#)$

Note that in a single-threaded execution system ($t(1) = 1$, $t(2) = 0$) only the first alternative is executable. The second and third alternative require both components to initially be in possession of at least one control flow ($t(1) = 1$, $t(2) = 1$), in order for component 2 to be able to execute transition $(\#,\#) \rightarrow (\#,r_j)$ in alternative one and $(p_i,r_j) \rightarrow (\#,r_j)$ in alternative two.

5.4.4 Interaction Traces

Before formalizing the – up to this point rather informal – discussion of "simulating" one interaction space within another, this subsection introduces the notion of interaction traces that capture the history of one specific interaction between two subspaces.

DEFINITION 5.7: *Given an interaction space* $IS = ([X]'_I,[p]_I,,[s]_I,[a]_I,l)$ *in an executed data space* $D_{sat} = (X',F',p')$ *with* $D = (X,F,p)$ *and* $S = \{F_1,F_2\}$. *The set of all executable* interaction traces *through the interaction space is given by*:

$$T_{IS} := \left\{((x_1,i_1,x_2),(x_2,i_2,x_3),\ldots,(x_{m-1},i_{m-1},x_m)) \;\middle|\; m > 1 \wedge x_m \in [X]_I \wedge x_1 = [s]_I \wedge \forall_{1<j\le m} : x_{j-1} \in [X]_I \wedge i_{j-1} \in C \wedge (x_j,i_{j-1}) \in Exe_{x_{j-1}}\right\} \cup \{\varepsilon\}$$

$$\text{with, given } z \in [X]'_I,\ Exe_z := \{(w,i) \mid w \in [X]'_I \wedge \exists_{a\in z,\, b\in w}((a,\ldots,n_i,\ldots),(b,\ldots)) \in p' \wedge n_i > 0 \wedge (a,b) \text{ is } i\text{-move in } D\}$$

(Remember: C is the index set of the splitting S.) An element t of T_{IS} is a sequence of labeled state transitions always starting from the initial state $[s]_I$ of the executed data space. In the following some notation for working with interaction traces is introduced:

DEFINITION 5.8: *Given*

$t = ((x_1,i_1,x_2),...,(x_{m-1},i_{m-1},x_m))$ *and* $t' = ((y_1,j_1,y_2),...,(y_{n-1},j_{n-1},y_n))$,

both with $1 < n < m$, $y_k \in [X]'_l$, $x_k \in [X]'_l$, $i_k \in C$, *and* $j_k \in C$,

(*Note that t and t' are not necessarily elements of* T_{IS}. *This generalization is necessary to include sequences that do not begin with the initial state* $[s]_l$.)

one defines:

- $len(t) := m\text{-}1$, *i.e. number of state transitions on the trace*.
- $len(\varepsilon) := 0$.
- $pos_k(t) := (x_k,i_k,x_{k+1})$, *i.e. state transition at position k*.
- $end(t) := x_m$, *i.e. the state after interaction sequence t.*
- $first_k(t) := ((x_1,i_1,x_2),...,(x_{k-1},i_{k-1},x_k))$ *for* $k > 1$, *i.e. the first k transitions of t.*
- $first_1(t) := \varepsilon$.
- $first(t) := first_2(t)$
- $t + t' := ((x_1,i_1,x_2),...,(x_{m-1},i_{m-1},x_m),(y_1,j_1,y_2),...,(y_{n-1},j_{n-1},y_n))$, *i.e. concat. of t and t'.*
- $t' < t :\Leftrightarrow \forall_{1 \le k < n} (x_k,i_k,x_{k+1}) = (y_k,i_k,y_{k+1})$, *i.e. t' is beginning of t*.
- $comp((x_k,i_k,x_{k+1})) := i_k$
- $cf((x,i,y))$ $:= -1$ *if* $a(((x,y),i)) \neq i \wedge i = 1$
  $:= 1$ *if* $a(((x,y),i)) \neq i \wedge i = 2$
  $:= 0$ *else.*

given $t' < t$:

- $t - t' := ((x_n,i_n,x_{n+1}),...,(x_{m-1},i_{m-1},x_m))$, *i.e. t without its beginning t'*.

An interaction trace captures the complete history of an interaction between two components. However, often this history is built up by repeatedly traversing cycles in the interaction path. Regard, for instance, the method call-like interaction space: after consuming the result values, the space is returned into its initial state and can be used again for further interaction. During the interaction, the control flow is passed back and forth between the two components. Thus when returning to the initial state, the balance of control flow passing is equal again. For the considerations in the following sections, it is useful to define the notion of an interaction cycle and certain properties of the cycle concerning control flow passing:

DEFINITION 5.9: *Given interaction space* $IS = (X,p,s,a,l)$,
$a = ((x_1,i_1,x_2),...,(x_{m-1},i_{m-1},x_m)) = t\text{-}b$ *with* $t \in T_{IS}$ *and* $b < t$,
a is an interaction cycle $:\Leftrightarrow$

$(x_1 = x_m) \wedge \forall_{1 \le h < j < m} x_h \neq x_j$.

Further notation concerning control flow passing properties:

- $im(IS) := max\{|\Sigma_{k \in \{1;..;len(a)\}} cf(pos_k(a))| \mid a \text{ is interaction cycle in } T_{IS}\}$.
- $maxim_1(IS) := max\{\Sigma_{k \in \{1;..;len(a)\}} cf(pos_k(b)) \mid b < a \wedge a \text{ is interaction cycle in } T_{IS}\}$.
- $maxim_2(IS) := max\{-\Sigma_{k \in \{1;..;len(a)\}} cf(pos_k(b)) \mid b < a \wedge a \text{ is interaction cycle in } T_{IS}\}$.

The expression $im(IS)$ denotes the maximal imbalance of control flow passing *after* traversing an interaction cycle of *IS*. The expression $maxim_i(IS)$ denotes the maximal imbalance towards component *i during* the traversal of an interaction cycle of *IS* (i.e. component *3-i* has received $maxim_i(IS)$ more control flows from component *i* than it has

transferred to it). Looking at the examples of table 5.1, one can easily observe: $im(IS_{method\text{-}call}) = 0$ and $maxim_1(IS_{method\text{-}call}) = 1$, while $im(IS_{shared\text{-}variable}) = 0$ and $maxim_1(IS_{shared\text{-}variable}) = 0$.

5.5 Simulation of Interaction

The example of simulating a method call with the help of two events built on an intuitive notion of simulation. This section is devoted to a more thorough treatment of this concept.

Why does the interaction space $IS_{simulated\text{-}method\text{-}call}(P,R)$ shown at the bottom of figure 5.3 permit the "simulation" of a method call-like interaction space shown in the first row of table 5.1? Proper simulation has to accomodate three essential features of component interaction:

- *Communication*: by traversing the interaction graph shown in figure 5.3, both subspaces make deliberate choices – e.g. the choice of the parameter state made by the first subspace with the transition $((\#,\#),(p_i,\#))$ – that are visible to the respective other subspace. Thus a specific interaction trace encodes as certain amount of information. A simulation of the interaction trace in another interaction space should therefore encode at least the same amount of information. Traversing $IS_{simulated\text{-}method\text{-}call}(P,R)$ in the fashion discussed in subsection 5.4.3 clearly communicates the choice of parameters from client to server and the choice of result from server to client. Thus the communication during the method-call interaction example is simulated properly.
- *Synchronization*: the order in which state transitions in an interaction space can be executed is used by subspaces to synchronize their activities. During a method call, first the parameters are passed, then the server performs its computations (invisible in the interaction space), then it returns the results that are finally consumed by the client. A proper simulation should maintain this order. Regarding the first execution sequence below figure 5.3, one observes an additional state transition – $((p_i,\#),(\#,\#))$, attributed to the (gray) server subspace – that returns the interaction space into its initial state by returning the initial parameter event call. This additional transition – while putting additional burden on the subspaces involved in the interaction – does not relax the original synchronization order of parameter passing, result passing, and result consuming. Thus, while being more complex, the synchronization is simulated properly.
- *Control flow passing*: the third feature of an interaction is the passing of the flow of control. During the "original" method call, the client passes the control flow to the server together with the parameters and thus permits the (formerly inactive) server to compute the results. Afterwards, the server passes the control flow back to the client together with the results. During the simulated method call the server also receives the control flow together with the parameters and returns it with the results. While there are two more control flow passes until the interaction space returns to its initial state (the events have to be returned, also returning the respective control flows), the final balance remains the same as in the original interaction space. Thus, because the control flow is within the correct subspace at the correct time and the balance remains the same, control flow passing is simulated properly.

Thus in order to be able to simulate one interaction space within another, it has to be possible to unambiguously map an interaction trace of the original interaction space onto an interaction trace of the simulating interaction space, while communicating the same information between the subspaces, maintaining the synchronization order and the control flow passing balance. This notion of simulation is captured in:

DEFINITION 5.10: *Given two executed data spaces D_{sat} and D'_{sat} with $S = \{F_1,F_2\}$, $S' = \{F'_1,F'_2\}$, and their interaction spaces IS and IS', one says IS' permits simulation of IS (or $IS' \equiv IS$) :⇔*

$$\exists_{m:T_{IS} \to T_{IS'}} : \forall_{t,u \in T_{IS}} : (t \neq u) \Rightarrow (m(t) \neq m(u)) \wedge \quad (1)$$

$$(t < u) \Rightarrow (m(t) < m(u)) \wedge \quad (2)$$

$$\forall_{(x,i),\,(y,j) \in Exe_{end(t)}} : \textit{communication} \wedge \textit{synchronization} \wedge \textit{balance} \quad (3)$$

($Exe_{end(t)}$ taken from definition 5.7) with:

$$\textit{communication} := (x \neq y) \Rightarrow \exists_{h \text{ with } 1 \le h \le min(\, len(a)-len(m(t)) \,;\, len(b)-len(m(t)) \,)} : \{ first_h(a\text{-}m(t)) = first_h(b\text{-}m(t)) \wedge first(c) \neq first(d) \Rightarrow comp(first(c)) = comp(first(d)) = i \} .$$

$$\textit{balance} := \Sigma_{k \in \{1;..;len(a)\}} cf(pos_k(a)) = \Sigma_{k \in \{1;..;len(t+(last(t),i,x))\}} cf(pos_k(t+(end(t),i,x))) .$$

$$\textit{synchronization} := \exists_{k \in \{1;..;len(a\text{-}m(t))\}} : comp(pos_k(a\text{-}m(t))) = i .$$

with:

$a := m(t+(end(t),i,x))$,

$b := m(t+(end(t),j,y))$,

$c := a\text{-}(m(t) + first_h(a\text{-}m(t)))$ *for* $h > 0$, and

$d := b\text{-}(m(t) + first_h(b\text{-}m(t)))$ *for* $h > 0$.

Condition (1) of the definition implies that the mapping is unambiguous, i.e. that no two traces of *IS* are mapped onto one trace of *IS'*. Condition (2) guarantees that the mapping respects the continuation relation, i.e. that if t is a beginning of u, than $m(t)$ is also a beginning of $m(u)$. Condition (3) concerns the set $Exe_{end(t)}$ of executable state transitions continuing interaction trace t (this set is given in definition 5.7). Regard figure 5.4:

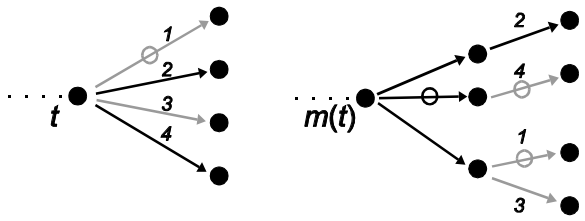


Figure 5.4: Four possible continuations of interaction trace *t* (left) are mapped to four distinct continuations of *m*(*t*) (right). Here, the mapping is expressed by the numbering of the trace segments

Trace t of the original interaction space *IS* has four possible continuations, 1 and 3 are executed by component two (gray) and 2 and 4 executed by component one (black). These four continuations are mapped onto four distinct continuations of $m(t)$ in the simulating interaction space (the length of these continuations can but does not have to be greater than one). Condition (3) now poses three subconditions on this mapping.

Communication

Conditions (1) and (2) already ensure that at least the amount of information encoded in the continuations of t (in the example of figure 5.4: 2 bit) can also be encoded in the continuations of $m(t)$. However, it is not guaranteed that the information is communicated in the correct direction. Of the four continuations of t in figure 5.4, two are executed by the gray component and two are executed by the black component. Thus by continuing trace t, actually only 1 bit can be communicated in each direction. The purpose of the first

subcondition *communication* is to ensure that the mapping *m* respects the direction of information flow, i.e. that the choice, which (simulation) trace to chose resides with the correct component. In figure 5.4, regard continuations 1 and 3 executed in *IS* by the gray component. These continuations are simulated in *IS′* by two continuations that share the same beginning and then later develop two distinct branches. The subcondition *communication* demands that whenever a choice is made, which trace to take, this choice is made by the correct component, i.e. the component that executes the single state transitions making up the original continuation in *IS*. In figure 3 the choice regarding 1 or 3 is – in the original IS – made by the gray component. In the simulation on the left, it is also made by the gray component, even though this occurs after a transition executed by the black component. The choice between 2 and 4 is correctly made by the black component during the first transition of the continuation.

Synchronization

The second subcondition of (3) – *synchronization* – deals with the correct turn-taking during interaction. This condition demands that at least one of the state transitions in the continuation of a trace is executed by the correct subspace, i.e. that subspace that executed the original continuation. However, the quality of this kind of synchronization is rather weak. Regarding continuations 1 and 3 in figure 5.4, it is obvious that while the choice between 1 and 3 is made by the correct (the gray) component, the initial decision to permit the gray component to act at all is made solely by the black component with the first state transition. In the original interaction, the decision to act could be made by both components (in a non-deterministic, "first-come-first-serve" fashion). In some cases this kind of simulation might be sufficient. In other cases a stricter condition is needed:

$$synchronization' := comp(first(a\text{-}m(t))) = i\ .$$

(Note that *i* is already bound in the first part of definition 5.10.) This alternative to the synchronization subcondition requires the *first* state transition in a continuation to be executed by the correct component. Consequently, if in the original interaction space a component can decide to act, it can do so in the simulating interaction space, as well. The remaining problem with *synchronization′* is the fact that in a simulated continuation, the originally executing component might rely on actions by the other component. Regard the simulation of continuation 4 (originally executed by the black component) where the gray component has to execute the last transition. An interaction simulated in this way couples the components closer together than necessary. The highest level of simulation quality is achieved with the following alternative subcondition:

$$synchronization'' := \forall_{\,k \in \{1;..;len(a\text{-}m(t))\}} : comp(pos_k(a\text{-}m(t))) = i\ .$$

Here, all transitions of a continuation have to be executed by the correct subspace. Continuation 2 is an example for such a continuation.

The three alternative definitions of the synchronization subcondition imply three different definitions of simulation denoted by ≡" , ≡', and ≡, with

$$(IS' \equiv'' IS) \;\Rightarrow\; (IS' \equiv' IS) \;\Rightarrow\; (IS' \equiv IS), \quad \text{for given } IS' \text{ and } IS.$$

Balance

The final subcondition – *balance* – requires that each simulated continuation affects the control flow balance (i.e. the function *t*(...) of the executed data space) in the same way as the original continuation does. Regarding the example in figure 5.4, one can see that the original continuation 1 entails the passing of one control flow from gray to black. The simulated continuation 1 in the last transition also passes one control flow from gray to black, thus meeting the subcondition.

As another example, the original continuation 4 does not pass the control flow, while its simulation passes the control flow twice. However, since it is passed back and forth, the balance after the simulation is not disturbed and thus the condition *balance* is met as well.

Summing up, the three different levels of simulation defined here require the simulating interaction space to respect the quantity and direction of the information flow and maintain the correct control flow balance. The three levels of simulation quality stem from three increasingly strict synchronization conditions. In the next subsection, these levels are used as basis for the comparative investigation of the expressive power of the JAVA event and shared variable interaction primitive with respect to simulation.

5.5.1 Comparing the Expressive Power of Different Interaction Primitives

The preceding definitions have laid the groundwork for the following theorem[3] that qualifies the expressive power of the JAVA event primitive and contrasts it with the shared variable primitive.

THEOREM 5.1: *Given two executed data spaces D_{sat} and D'_{sat} with $S = \{F_1,F_2\}$, $S' = \{F'_1,F'_2\}$, and their interaction spaces $IS = (X,p,s,a,l)$, $IS' = (X',p',s',a',l')$, $im(IS) = 0$, and $|X|$ finite*:

(1) $IS' = IS_{shared\text{-}variable}(X) \otimes$
$(IS_{java\text{-}event}(\{1\}) \otimes ... \otimes IS_{java\text{-}event}(\{maxim_1(IS)\})) \otimes$
$\downarrow(IS_{java\text{-}event}(\{1\}) \otimes ... \otimes IS_{java\text{-}event}(\{maxim_2(IS)\}))$

$\Rightarrow$

$IS' \equiv'' IS$ ‘shared variables and events together permit $\equiv''$ simulation

(2) $IS' = IS_{java\text{-}event}(X) \otimes \downarrow IS_{java\text{-}event}(X) \otimes$
$(IS_{java\text{-}event}(\{1\}) \otimes ... \otimes IS_{java\text{-}event}(\{maxim_1(IS)\text{-}1\})) \otimes$
$\downarrow(IS_{java\text{-}event}(\{1\}) \otimes ... \otimes IS_{java\text{-}event}(\{maxim_2(IS)\text{-}1\}))$

$\Rightarrow$

$IS' \equiv' IS$ ‘events alone permit $\equiv'$ simulation

PROOF (1):

The technique is proof by induction over the length of the interaction trace. The induction start for length zero is obvious: $m(\varepsilon) := \varepsilon$, and the properties of definition 5.10 are trivially met for this part of m. Given $t \in T_{IS}$, with $len(t) = n > 0$, let U be the set of traces with $u \in U \Rightarrow len(u) = n+1 \wedge t < u$. The induction hypothesis (the properties of definition 5.10) is assumed to hold for all traces with length less or equal than n. Because the state transitions in an interaction space are irreflexive and $|X|$ is finite, one gets $w = |U| \leq |X| - 1$. Assuming the elements of U to be numbered as u_1 to u_w, one has to construct w traces $m(u_1)$ ... $m(u_w)$ that are distinct continuations of the $m(t)$ given by the induction hypothesis (condition 1 and 2 of definition 5.10) and fulfill the conditions of *communication*, *synchronization* and *balance*. With $x = end(t)$, $y_i = end(u_i)$, $x' = end(m(t))$, $y'_i = end(m(u_i))$, figure 5.5 depicts the construction problem:

[3] Theorem 5.1. could be extended to the more general case of infinite interaction spaces. However, this would require the simulating interaction space to contain a combination of a potentially infinite number of JAVA events in order to meet the condition of balanced control flow passing. For the practical questions addressed here, this level of generality is not necessary. Furthermore, the initial assumption of infinite interaction spaces is only of theoretical interest. In practice, one can assume the state space of every interaction primitive to be – perhaps very large – but always finite.

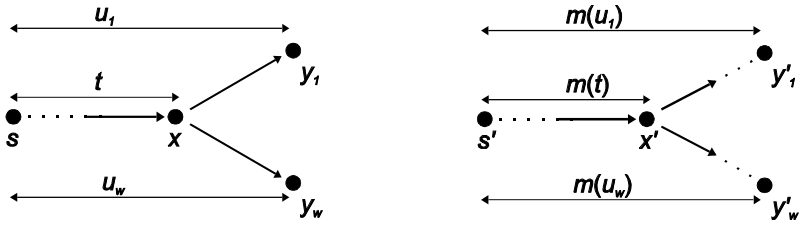


Figure 5.5: For each continuation u_i of t (left side) one has to construct a distinct continuation $m(u_i)$ of $m(t)$ (right side), meeting the respective conditions of definition 5.10.

Assume w.l.o.g. the shared variable of IS' to be in state z. Take elements x_1 to x_w of X-$\{z\}$. Each $m(u_i)$ is constructed by letting the component denoted by $comp(pos_{len(u_i)}(u))$, i.e. the component that executes state transition (x,y_i), change the state of the shared variable to x_i. In case the control flow is passed during (x,y_i), let the same component either return an event that was sent by the other component, or – if such an event is not available, because the other component has not yet sent one – send one event. The condition $im(IS) = 0$ and the number of event interaction primitives available ($maxim_1(IS)$ from component one to component two and $maxim_2(IS)$ for the other direction) guarantee that there is always an event that can be sent or returned in order to pass the control flow.

Consequently, $m(t)$ is continued to $m(u_i)$ by either a state transition changing the shared variable or by a first state transitions changing the shared variable and a second transition firing or returning an event. This construction meets the conditions of definition 5.10, because it is a unique continuation of $m(t)$, different from all other $m(u_j)$ because of the w states of the shared variable used. Furthermore, the condition of *communication* is trivially fulfilled, because only the component executing state transition (x,y_i) is involved in the state transition(s) from x' to y'_i. Thus also the condition of *synchronization''* is met. The *balance* condition is met, as well, because in case the control flow is passed, either an event is returned or sent by the component executing state transition (x,y_i). Both options transfer the control flow to the other component.

PROOF (2):

This part of the theorem concerns the quality of simulation with only event interaction primitives available. The proof is similar to proof (1). The construction of the continuation is the same, except for the fact that the shared variable is replaced by two events which can communicate up to |X| different event states each. Instead of a component changing the state of the shared variable to x_i, it fires an event with state x_i. The event is returned immediately and the construction continues in the same fashion as for proof (1). Consequently, the continuation consists of either two or three state transitions, the second of which is always executed by the component not executing state transition (x,y_i).

Again the construction yields a unique continuation of $m(t)$. The condition of *communication* is met, because the component executing state transition (x,y_i) also decides the state of the first event fired. The condition of *synchronization'* is also met because the first state transition of the continuation is executed by the component executing state transition (x,y_i) in the original interaction space. However, note that the condition of *synchronization''* is not met, because the second state transition is always executed by the component not executing state transition (x,y_i) in IS. The condition of *balance* is met, because the event fired first is always returned immediately. The passing of control flow is then solely determined by possibly firing (or returning) an additional event. ■

In addition to the theorem, it is obvious that with events alone, *synchronization"* cannot always be achieved, because it is not possible to simulate an interaction cycle executed by only one component. In an event-based interaction no such cycles are possible, because the other subspace always has to give back the control flow in order to avoid violating the *im*(*IS*) = 0 condition of the theorem.

Furthermore, it is obvious that shared variables alone cannot simulate an interaction space that at some time passes the control flow, because shared variables cannot pass the control flow.

5.5.2 Consequences for Implementing Component Interaction

The theorem – together with the last two observations – permits the interpretation that JAVA events alone are more powerful than shared variables, but less than JAVA events and shared variables together. The former distinction is rather obvious: shared variables do not transfer control flow. How does the latter influence the implementation of component interaction in a chosen decomposition?

When implementing component interaction only with JAVA events, one component might be forced to take action (e.g. give back the control flow, as in the method call-simulation example in figure 5.3) when it could actually remain passive in the intended interaction. This causes a stronger synchronization than necessary. Components have to wait for other components to perform state transitions that are just simulation "chores". This was also the problem with the implementation of the search tool example described in chapter 4. Solely event-based component interaction proved cumbersome and inefficient. Regard, for instance the interaction between the *document type* and the *search engine* component. "Elegant" ways to implement their interaction would be either a *method-call* from the *search engine* or a *shared variable* changed by the *document type* component. However, these interaction protocols are not available at the abstraction level of the JAVABEANS component model and the composition language. Therefore, in the actual implementation, a *shared variable*-like interaction protocol is simulated by simply sending events to the search engine whenever a state transition of the observable is to be simulated. In the alternative implementation depicted in figure 4.7, two events simulate a *method-call* like interaction protocol. In both cases, the quality of the simulation is only ≡' in the more formal terminology of this chapter.

Assuming that the component model offers shared variables as interaction primitive in addition to JAVA events, one could implement the interaction between the *document type* and the *search engine* component in the fashion depicted in figure 5.6:

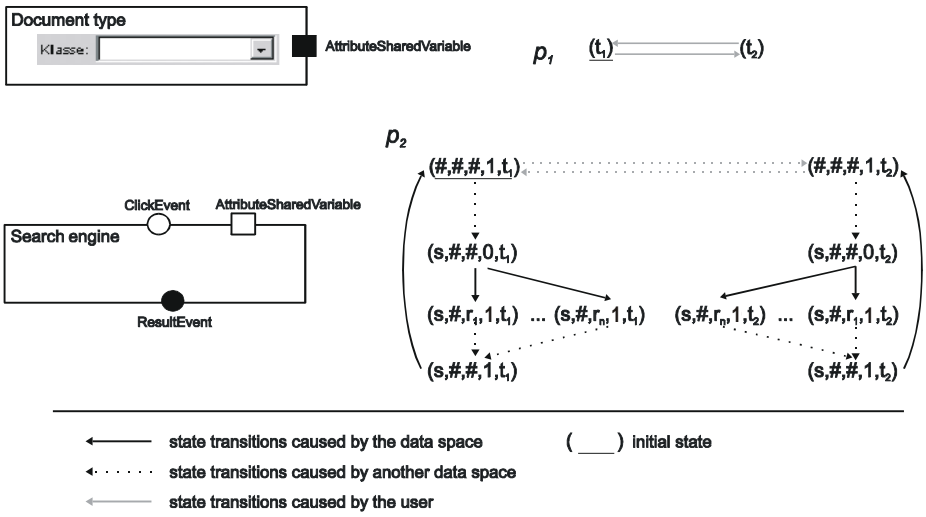


Figure 5.6: *Document type* and *search engine* component interaction implemented with a shared variable

Instead of communication the state of the type attribute via an event, the *document type* component now provides the state of its selection list as a shared variable that can be accessed by the *search engine* component when needed – and only then.

In comparison to figure 5.2, it is obvious that the new implementation is simpler. The *search engine* component is only involved in the interaction (i.e. its behavior depends upon or is synchronized with the *document type* component), when it actually needs the state of the drop down list – not before. Thus if the user changes the state several times before he initiates a search, the two components do not unnecessarily communicate or have to wait for each other – as it is the case when sending an event each time a change to the drop down list occurs. This demonstrates how the problems encountered in the JAVABEANS-based search tool example could be solved in a component model offering both JAVA events *and* shared variables as interaction primitives.

Consequently – as main result of this chapter – the theorem suggests a combination of shared variable-like and JAVA event-like interaction primitives as an answer to the initial question "*Which interaction style or set of styles should a component model support in order to permit the implementation of arbitrary decompositions?*" This result is the basis of chapter 7 which develops the FLEXIBEANS component model.

5.6 Summary and Discussion

The purpose of this chapter was to formally address the problems raised by the application of the JAVABEANS component model for component based tailorability. The applicability and relevance of the theory developed here obviously depends on the question, whether the model captures the essential aspects of software components and their interaction. The aspects of component interaction captured in the notion of interaction spaces are:

- direction and quantity of information flow
- synchronization of component activities
- control flow passing

The formalization of *interaction simulation* is based on these aspects as well. The relevance of the theory is supported by the fact that it was possible to model a number of rather diverse interaction styles as interaction spaces (see table 5.1). Furthermore, the theory can explain the component interaction problems encountered in the case study of chapter 4. The problems appear in the formal model as insufficient interaction simulation quality.

Apart from these modeling and explanatory purposes, the quality of a theory is also measured by its predictive power. As main result of this chapter, the theory predicts that a component model supporting both *JAVA events* and *shared variables* should make it easier for a component developer to implement a prescribed decomposition and the resulting specific component interactions. Chapter 9 describes the implementation of a number of groupware systems with the FLEXIBEANS component model (which supports both interaction primitives). The evaluation of these implementations shows that the necessary component interactions could be implemented more efficiently and naturally, thus validating (or rather not falsifying...) the theory developed in this chapter.

Related work on formal foundations for component software often employs process calculi like CCS (Milner 1980), CSP (Hoare 1985), or the π-calculus (Milner et al. 1992) as basic model.

(Radestock and Eisenbach 1994), for instance, show how the π-calculus can be used to model the operational semantics of the configuration language DARWIN (Magee et al. 1995), i.e. the initial message passing of a number of distributed processes (regarded as components) with the goal of binding services provided by one process to other processes requiring these services. The model helped to uncover problems with circular message

passing. However, in contrast to the work presented here, it was not concerned with the comparison of different interaction primitives.

The WRIGHT language (see Allen 1997), employs CSP to model the connections between components. Components as well as connectors between components are modeled as CSP processes. WRIGHT connectors are similar to the notion of interaction primitives developed here. However, the language is primarily designed and used for descriptive purposes and automatic consistency checking. It is not used for comparative analysis.

A general problem of using process calculi to model software components and their interactions is the fact that components (at least in the understanding of this dissertation) are not necessarily active entities in their own right. A software system might consist of hundreds of component instances all residing within one process and all being traversed by only one thread of control. Thus modeling components (and even more so interactions) as active processes appears unnatural. (Lumpe et al. 1997) report on the implementation of a π-calculus-based composition language modeling components as processes. The implementation in JAVA employs a 1:1 mapping of calculus processes to JAVA threads, which leads to unwieldy and rather inefficient applications with more than a hundred threads of control even for small systems. This is an indication for the mismatch of modeling components as processes (Lumpe et al. see the problem more in their implementation than in the underlying formal model).

(Vion-Dury et al. 1997) reconcile the notion of activity (and passivity) with the process calculus CCS. Their extension of CCS introduces a new composition operator which restricts the way processes can evolve on either side of the operator: a process on the left can only perform output operations while a process on the right can only perform input operations. When the joint interaction activity is performed, the sides of the operator are reversed, thus modeling the passing of the control flow during value-passing. However, their calculus is tailored to the OLAN configuration language (Bellissard et al. 1995) and it seems that it would be rather complicated to retrace the notions presented here in their notation.

Nevertheless, it is an interesting question whether and how the results presented here could be achieved within the theoretical framework of a process calculus.

Another possible formal base for an investigation of this kind is given by ASMs (Abstract State Machines, see Gurevich 1997), formerly known as *evolving algebras*. The intention of AMSs is similar to that of data spaces, namely to support the specification of computational systems on different levels of abstraction and detail. (Börger and Schulte 1998), for instance, have formally modeled the programming language JAVA, including aspects like multi-threading and object interaction (via method calls). Thus it appears to be possible to model components and component interaction within the ASM framework. However, since ASMs and data spaces have the same expressiveness, not much would be gained by retracing the notions presented here as ASMs. Nevertheless, an interesting point for future work would be to apply the (semi-) automatic verifiability methods for ASMs developed by (Spielmann 2000) to the problem of dynamic runtime changes of compositions (compare chapter 10). One could specify a computational system whose composition is changed over time, while performing computations (i.e. a complex system of interleaving use and tailoring transitions). The work by (Spielmann 2000) then permits the formal specification and verification of correctness properties of this system. Consequently, the runtime and tailoring environment could check (assuming the use transitions are known for a certain time interval in the future), if a certain tailoring operation is safe with respect to a certain correctness property.

Chapter 6

Managing Component Structures

6.1 Introduction

This chapter addresses the question "*How does one design the connection between representation and application in a way that supports sharing of complex components, lets changes take effect in all instances of such a component immediately, and permits control of a change's scope?*" A formal model is presented that defines the basic constituents of a runtime and tailoring environment with these properties and supports the development of these environments independent of the programming language. The model covers the representation of component-based systems, the instantiation operations, the instantiated system, the tailoring operations and finally the scope control operations.

6.2 Overview

In existing systems the connection between the representation of a component-based system and the running application is realized in the fashion depicted in figure 6.1:

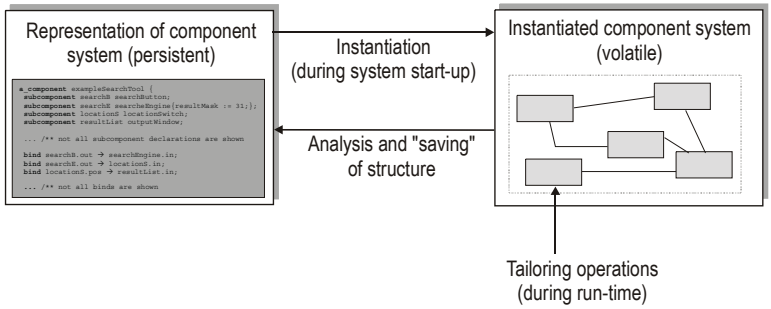


Figure 6.1: Traditional runtime management of component based systems

The component system is represented in persistent form. This representation (left side of figure 6.1) is evaluated during system start-up resulting in the instantiated, running application (right side of figure 6.1). This method is used, for instance, in the distributed systems OLAN (Bellissard et al. 1996) and REGIS (Magee et al. 1995). It was also employed in the exploratory case study described in chapter 4. REGIS and the search tool environment also permit interactive compositional changes directly to the instantiated system (tailoring operations, see figure 6.1). In order to make these changes persistent, the structure of the system can be analyzed and stored as DARWIN or CAT file, respectively. As described in chapter 4, the primary problem of this type of architecture is that it does not support runtime sharing of component structures. This chapter develops a formal model of an alternative architecture for a runtime and tailoring environment that supports sharing. Figure 6.2 depicts the basic elements of this architecture:

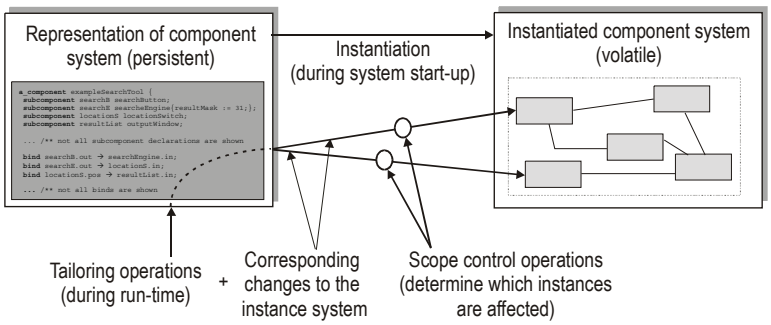


Figure 6.2: Component management with runtime sharing and scope control

The instantiation process during system start-up remains the same. The difference is that tailoring operations are applied directly to the persistent representation (or a copy thereof, residing in the machine's main memory). The runtime and tailoring system is responsible for applying corresponding changes to the instantiated, running system (left side of figure 6.2). *Corresponding* means that the running system is changed in such a way that it exactly resembles the system that would result from the instantiation of the changed persistent representation. When the specification of a complex component is shared, i.e. instantiated several times, a change to the persistent representation of that complex component correspondingly results in changes to *all* instances.

Since it can be necessary that instances of the same complex component evolve separately, the runtime and tailoring environment has to provide an operation to control the scope of tailoring operations. In the new architecture this is achieved by creating a copy of the persistent specification of the complex component's representation and declare these instance to which tailoring operations are supposed to apply as instances of the copied representation. Subsequently, the tailoring operations are applied only to the copy of the

representation, thus effectively delimiting the scope of the change to a subset of instances of the original complex component.

This chapter formally defines the central elements of the architecture – independent of programming language. The model is based on set theory and first order predicate logic[4] and supports the developer of a component-based runtime and tailoring in the implementation of the different operations. It covers the following elements:

- *The representation of a component-based system*. This part essentially models the information contained in a CAT file: atomic components, complex components and the representation of a whole component-based system.
- *The instantiation operation and the instantiated system*. The formal model of the instantiation operation consists of two operators – α and β – that are applied in turn to the representation of the component-based system and yield a model of the instantiated system.
- *The set of tailoring operations*. Each tailoring operation is modeled as a change to the (persistent) representation, together with the corresponding changes to the instantiated system.
- *The scope control operation*. Given a complex component and the corresponding set of instances, this operation is modeled as a change to the persistent representation.

The following four sections develop these elements of the formal model. Chapter 8 presents an implementation of the model in the object-oriented programming language JAVA – the EVOLVE system.

6.3 Representation

Based on set theory, this section develops formal representations of atomic components, complex components, and whole component-based systems that relate to CAT files and CAT language constructs.

In the following, the non-empty and finite sets $N_{components}$, $N_{instances}$, N_{ports}, $N_{porttypes}$ and $N_{parameters}$ serve as identifiers (names). For all practical purposes, names can be regarded as strings. The definitions in this section are simplified by expressing a number of obvious but necessary conditions (e.g. uniqueness of names) in textual form. The complete formal definitions in first order predicate logic can be found in Appendix B.

6.3.1 Atomic Components

Atomic components are the basic building blocks of a composition. Since they are implemented in a programming language and deployed in binary form, they cannot be further decomposed and can only be tailored by parameterization. They offer mechanisms for introspection that permit the environment to learn about the component's tailorable parameters and the composition interface, i.e. the set of ports with types, polarities and names. In CAT this information is represented by the `i_component` construct – e.g. as in the search tool example in chapter 4:

[4] The set theoretic and predicate logic style representation was chosen, because it best serves the needs of programmers, who want to implement the model. A partially graph-style representation was considered as well. However, the necessary degree of precision and the resulting complexity was not well supported by graphical methods. The level of detail (see e.g. definition 6.3) is difficult to accommodate in a diagram.

```
i_component searchEngine {
 required init ClickEvent;
 required attribs AttributeEvent;
 provided out ResultEvent;
 config_parameter resultMask int;
};
```

The name of the implemented component is at the same time a reference to its implementation (in the part of the search tool example shown above it is the `searchEngine.class` file). Since this chapter – like the preceding one – abstracts from programming language, the implementation of an atomic component is modeled as a data space (*X,F,p*). In accordance with the discussion in the last chapter, the ports of a component are modeled as disjunct subsets of *F*. Parameters are modeled as single distinct elements of *F*, which are – if a value is given – set to that value, i.e. the initial state of the component is parameterized. The following definition provides all information about a component which is contained in an `i_component` CAT specification and which is needed for instantiation and definition:

DEFINITION 6.1: *An* atomic component *is a tuple* (*n,D,P,A*) *with*

- $n \in N_{components}$,
- *D is a data space* (*X,F,p*),
- $P \subseteq N_{ports} \times N_{porttypes} \times \{req.\ ;\ prov.\ ;\ sym.\} \times \wp(F)$, *and*
- $A \subseteq N_{parameters} \times F$.

($\wp(F)$ *denotes the set of all possible subsets of F, including* $\emptyset$ *and F itself*)

subject to the following conditions:

1. *port names are unique and the ports' information structures do not overlap.*
2. *parameter names are unique and denote a part of the information structure of a component which does not overlap with ports or other parameters.*
3. *a port's information structure is not empty.*

As in the CAT language, only component interactions with one role (symmetric, represented by `sym.`) or two distinct roles (asymmetric, represented by `req.` and `prov.`) are supported. The three conditions concern the uniqueness of port and parameter names within the atomic component and the distinctness of their mapping to the information structure of the data space. Further conditions apply to a set of atomic components as given by definition 6.2:

DEFINITION 6.2: *A* component set *is a non-empty set C of atomic components subject to the following conditions:*

1. *component names are unique*
2. *a port type is either symmetric or asymmetric*
3. *two ports of the same type have the same information structures up to isomorphism*

The conditions enforce uniqueness of component names and consistent usage of port types and polarities throughout a set of atomic components.

6.3.2 Complex Components

A complex component consists of a number of subcomponent instances that can either be specified by other (possibly parameterized) complex or atomic components. A complex component also defines ports and parameters, which are bound to ports and parameters of subcomponents. A subcomponent's ports can also be bound to other subcomponent's ports. In CAT a complex component is represented by the `a_component` construct – e.g. as in the search tool example in chapter 4:

```
a_component exampleSearchTool {
 subcomponent searchB searchButton;
 subcomponent searchE searcheEngine{resultMask := 31;};
 subcomponent locationS locationSwitch;
 subcomponent resultList outputWindow;

 bind searchB.out → searchEngine.in;
 bind searchE.out → locationS.in;
 bind locationS.pos → resultList.in;
```

The three `bind` constructs all connect subcomponents' ports. Not shown in this CAT code example are possible definitions of ports and parameters of the complex component and their bindings to subcomponents' ports and parameters (refer to Appendix A for the full CAT specification). Formally, a complex component is defined as follows:

DEFINITION 6.3: *Given a component set C, a* complex component *is a tuple* $(n, P, A, S, B_1, B_2, B_3)$ *with*:

- $n \in N_{components}$,
- $P \subseteq N_{ports} \times N_{porttypes} \times \{req.\ ;\ prov.\ ;\ sym.\}$,
- $A \subseteq N_{parameters} \times R$,
- $S \subseteq N_{components} \times N_{instances} \times \wp(N_{parameter} \times V)$,
- $B_1 \subseteq N_{instance} \times N_{ports} \times N_{instance} \times N_{ports}$,
- $B_2 \subseteq N_{ports} \times N_{instances} \times N_{ports}$,
- $B_3 \subseteq N_{parameters} \times N_{instances} \times N_{parameters}$,

with R the set of all ranges of functions in the F's of the data spaces of C, with V the union of all elements of R, and subject to the following conditions:

1. *component instances' names are unique*
2. *port names are unique*
3. *parameter names are unique*
4. *referential integrity of subcomponent binds w.r.t. instance names*
5. *referential integrity of port binds w.r.t. instance and port names*
6. *referential integrity of parameter binds w.r.t. instance and parameter names*
7. *no conflicts between bound and given parameters of subcomponents*
8. *a subcomponent port is bound exclusively either to a port of another subcomponent or to a port of the complex component*
9. *subcomponents' parameters are bound at most once*
10. *subcomponents' ports are bound to at most one port of the complex component*

The conditions concern the uniqueness of port, parameter and subcomponent instance names, the internal referential integrity and the correct internal binding of parameters. There are three types of binds: the binds defined in B_1 describe port bindings between subcomponent instances, the binds in B_2 describe port bindings between subcomponent instances and the complex component, and the binds in B_3 describe parameter bindings between subcomponent instances and the complex component. Subcomponent instances can be parameterized.

6.3.3 Representation of Component-Based Systems

In a CAT file, a component-based system is represented by a number of atomic components (`i_component`), a number of complex components (`a_component`) and a single system component (`s_component`) – a special complex component that is characterized by the fact that it is not used as a subcomponent in any of the other complex components. The system

component is also called *root component*. The formal representation of a component-based system is given by:

DEFINITION 6.4: *A* representation of a component-based system *is a tuple* (*r, H, C*) *with:*

- $r \in N_{component}$
- *H a non-empty set of complex components*
- *C a component framework*

subject to the following conditions:

1. *complex and atomic component names are unique*
2. *referential integrity of subcomponent binds' left side*
3. *referential integrity of subcomponent binds' right side*
4. *referential integrity of port binds*
5. *referential integrity of parameter binds*
6. *static type and polarity compatibility of subcomponent binds*
7. *static type and polarity compatibility of port binds*
8. *static type compatibility of parameter binds*
9. *static type compatibility of subcomponent parameterization*
10. *no cycles and no self references*
11. *the root component of the system exists and is not a subcomponent of another component*
12. *referential integrity w.r.t. all subcomponents*

The conditions concern the uniqueness of component names, type compatibility of all binds, referential integrity, non-cyclical definition of complex components, and the above-mentioned special characteristics of the root component. Type and polarity compatibility in this context means that only ports with the same type and opposite polarity are connected in the representation (if the polarity is *symmetric*, the polarity compatibility condition obviously does not apply).

6.4 Instantiation

The definitions given in the last section constitute a formal model of the representation of component-based systems. By formally defining the process and the result of deriving instantiated software systems from these representations, this section contributes the second element required for modeling the connection between representation and application in a component-based tailorable system.

6.4.1 The Process of Instantiation

The process of deriving an instantiated software system from a representation is quite straightforward. One begins with the root component and in a first pass through the representation recursively creates and parameterizes the subcomponent instances specified by the complex components. The first pass results in a set of instantiated atomic components and the instance hierarchy (i.e. the aggregation hierarchy given by the structure of the complex components together with the hierarchical bind information). During the second pass, the instance hierarchy is traversed and bindings between complex components are realized by connecting the corresponding atomic component instances, i.e. the bindings

between complex component instances are passed down the instance hierarchy. This two pass-method of evaluating the representation of a component-based system is formalized in definition 6.5:

DEFINITION 6.5: *An* instance system $\mathfrak{I}$ *of the representation of a component-based system* (r,H,C) *is a tuple* (I,B_1,B_2,A) *with*

- $(I,B,B_2,A) := \alpha(r,"root",\varnothing)$ and
- $B_1 := \beta(B,B_2)$.

The first pass is modeled by the application of the recursive instantiation operator α (given in definition 6.6) upon the root component of the system. The result is a set of component instances I (complex and atomic) and the bind information of the instantiation hierarchy in form of a set B of binds between ports of subcomponent instances, a set B_2 of binds between ports of subcomponent instances and parent components instances, and additionally a set A containing parameterization information.

The second pass is modeled by the application of the iterative bind operator β (given in definition 6.7) upon the sets B and B_2, resulting in a set B_1 that contains the bindings of the ports of all components.

The instantiation operator α is given by:

DEFINITION 6.6: $\alpha(c,name,\tilde{A})$ $= (I,B,B_2,A)$ *is the* instantiation operator *with:*

- $c \in H \cup C$ *'the component to be instantiated*
- $name \in (\text{root}|N_{instances})\{.N_{instances}\}^*$ *'names are period-delimited sequences of instance names, possibly beginning with root*
- $\tilde{A} \in \wp(N_{parameters} \times V)$ *'the parameterization of the component*

with V like in definition 6.3 and the definition subject to the condition:

1. *all instance parameters exist and their parameterization is type compatible*

$\alpha(c,name,A^*) = (I,B,B_2,A)$ *is defined for the two cases* $c \in H$ *and* $c \in C$ *(for some remarks on notation see footnote[5]):*

first case $c \in H$ *(c is a complex component) with* $c = (n,P,A,S,B_1,B_2,B_3)$*:*

$$I := \bigcup_{s \in S_c} I_{\alpha(n_s,name+"."+i_s,P_s)} \cup \{(n_c,name)\}$$

$$B := \bigcup_{s \in S_c} B_{\alpha(n_s,name+"."+i_s,A_s)} \cup \bigcup_{(i_1,p_1,i_2,p_2) \in B_{1_c}} \{\{(name+"."+i_1,p_1);(name+"."+i_2,p_2)\}\}$$

$$B_2 := \bigcup_{s \in S_c} B_{2_{\alpha(n_s,name+"."+i_s,A_s)}} \cup \bigcup_{(p_1,i_1,p_2) \in B_{2_c}} \{(name,p_1,name+"."+i_1,p_2)\}$$

---

[5] In order to simplify expressions accessing single elements of tuples, the following notation is introduced: Given a tupel $t = (a,b,c)$, a_t denotes the first element of the tuple, b_t the second etc. Consequently, S_c denotes the set S of $c = (n,P,A,S,B_1,B_2,B_3)$, $I_{\alpha(...)}$ the set I of $(I,B,B_2,A) = \alpha(...)$, etc. Furthermore, if an expression only accesses a subset of tuple element, the irrelevant elements can be substituted by a dot, as in S_c *denotes the set S of* $c = (.,.,.,S,.,.,.)$.

The names of subcomponents are constructed by taking the name of the parent component and extending it with a dot and the name of the subcomponent instance: $name+"."+i_s$.

$$A := \bigcup_{s \in S_c} A_{\alpha(n_s, name+"."+i_s, A_s \cup \{(p,v) \mid (p',v) \in A^{\sim} \wedge (p', i_s, p) \in B_{3_c}\})} \cup \bigcup_{a \in A_c} \{(name, p_a, v_a)\}$$

second case $c \in C$ (*c is an atomic component*) *with* $c = (n,D,P,A)$*:*

$$I := \{(n_c, name)\}$$

$$B := \varnothing$$

$$B_2 := \varnothing$$

$$A := \bigcup_{a \in A_c} \{(name, p_a, v_a)\}$$

The bind operator β is given by:

DEFINITION 6.7: β *is the* bind operator *and is defined by* $\beta(B,B_2) := D$ *with*

$$D_0 := B$$

$$D_i := D_{i-1} \cup \Big\{ \{(i_1,p_1);(i_2,p_2)\} \mid \exists_{\{(i'_1,p'_1);(i'_2,p'_2)\} \in D_{i-1}} \big((i'_1,p'_1,i_1,p_1) \in B_2 \vee (i'_1,p'_1) = (i_1,p_1)\big) \wedge \big((i'_2,p'_2,i_2,p_2) \in B_2 \vee (i'_2,p'_2) = (i_2,p_2)\big) \Big\}$$

$$D := D_n \text{ with } n = min(\{n \mid D_n = D_{n+1}\})$$

The instance system tuple (I,B_1,B_2,A) resulting from the application of the α and β operators models the structure of the instantiated software system. The following notations are needed in the next section:

$I_c := \{i \mid i = (c,.) \in I\}$ 'set of all instances of component c

$[(i,p)] := \{(i',p') \mid \exists_{b \in B_1}\ b = \{(i,p);(i',p')\}\}$ 'all ports connected to port (i,p).

6.5 Tailoring Operations

Based on the models of representation and instance system developed in the last two sections, this section presents a set of tailoring operations permitting runtime tailoring of shared complex components. As stated at the beginning of this chapter, the underlying idea is to apply tailoring operations directly to the representation (r,H,C) and immediately update the instance system (I,B_1,B_2,A) accordingly. Thus each change to a shared complex component of the representation is mirrored by a number of changes to all instances of that component, thus maintaining consistency between representation and instance system. In contrast to the approach employed in the search tool example, changes involving a shared complex component can thus become effective immediately in all affected instances. While this section deals with making this immediate sharing possible, section 6.6 presents a methods to control the scope of the change (i.e. the set of affected instances) in cases when the effect of a change is not supposed to affect certain instances.

The following subsections each specify tailoring operations in terms of elements of the representation (r,H,C). For each operation, the corresponding changes to the instance system (I,B_1,B_2,A) are given. The treatment is designed to support the developer of a component-based runtime and tailoring environment by formally and completely describing the semantics of the tailoring operations in a fashion independent of programming language.

6.5.1 Adding and Removing Subcomponent Instances

Adding and removing subcomponent instances is always performed in the context of a complex component $c = (n,P,A,S,B_1,B_2,B_3) \in H$. Assume that the subcomponent instance s =

(n,i,A) is to be added to c (also assume that none of the conditions in definitions 6.1 – 6.4 are violated):

Changes to (r,H,C)	Corresponding changes to (I,B_1,B_2,A)
$H_{new} := H - \{c\} \cup \{c_{new}\}$ $c_{new} := (n_c, P_c, A_c, S_{new}, B_{1_c}, B_{2_c}, B_{3_c})$ $S_{new} := S_c \cup \{s\}$	$I_{new} := I \cup \bigcup_{(.,j)\in I_c} I_{\alpha(n_s, j+"."+i_s, A_s)}$

While in the representation (n,i,A) is simply added to the set S_c of component c, in the instantiated system the new subcomponent has to be instantiated for each instance of c. Consistency is maintained, because instantiating (r,H_{new},C) obviously yields (I_{new},B_1,B_2,A).

Now assume that a subcomponent instance $s = (n,i,A) \in S$ is to be removed from c. Also assume that all bindings concerning this subcomponent have been removed beforehand (using the operations defined in subsection 6.5.2).

Changes to (r,H,C)	Corresponding changes to (I,B_1,B_2,A)
$H_{new} := H - \{c\} \cup \{c_{new}\}$ $c_{new} := (n_c, P_c, A_c, S_{new}, B_{1_c}, B_{2_c}, B_{3_c})$ $S_{new} := S - \{s\}$	$I_{new} := I - \bigcup_{(.,j)\in I_c} I_{\alpha(n_s, j+"."+i_s, A_s)}$ $A_{new} := A - \bigcup_{(.,j)\in I - I_{new}} \{(i,p,v) \mid (i,p,v) \in P \wedge i = j\}$

The corresponding changes to the instance system are quite extensive, involving all instances of component c and all parameters concerning the subcomponent instances of these instances.

6.5.2 Binding and Unbinding Ports of Subcomponent Instances

Like adding and removing subcomponent instances, binding and unbinding subcomponents is done in the context of a complex component $c = (n,P,A,S,B_1,B_2,B_3) \in H$. Here, a bind ($i_1,p_1,i_2,p_2$) is performed, again meeting the conditions of definitions 6.1 – 6.4:

Changes to (r,H,C)	Corresponding changes to (I,B_1,B_2,A)
$H_{new} := H - \{c\} \cup \{c_{new}\}$ $c_{new} := (n_c, P_c, A_c, S_c, B_{1_{c\text{new}}}, B_{2_c}, B_{3_c})$ $B_{1_{c_{new}}} := B_{1_c} \cup \{(i_1,p_1,i_2,p_2)\}$	$B_{1_{new}} := B_1 \cup \beta(\bigcup_{(.,j)\in I_c} \{(j+"."+i_1, p_1, j+"."+i_2, p_2\}, B_2)$

When connecting two subcomponent instances of c, one has to connect them in all instances of c as well. Furthermore, as the new connection can be on a high level of the component aggregation hierarchy, the β operator has to be applied in order to connect all lower instances and finally the atomic implementation instances.

One proceeds in a similar fashion, when disconnecting two subcomponent instances of c. Now, the bind $(i_1,p_1,i_2,p_2) \in B_{1c}$ is removed:

Changes to (r,H,C)	Corresponding changes to (I,B_1,B_2,A)
$H_{new} := H - \{c\} \cup \{c_{new}\}$ $c_{new} := (n_c, P_c, A_c, S_c, B_{1_{c\text{new}}}, B_{2_c}, B_{3_c})$ $B_{1_{c_{new}}} := B_{1_c} - \{(i_1,p_1,i_2,p_2)\}$	$B_{1_{new}} := B_1 - \beta(\bigcup_{(.,j)\in I_c} \{(j+"."+i_1, p_1, j+"."+i_2, p_2\}, B_2)$

Note, that because a port of a subcomponent instance can exclusively either be connected to a port of the parent component or to a port of another subcomponent instance (condition 8 of definition 6.3), the definition of the changes in the instance system maintains consistency. Because of this condition the connection between two atomic component instances can be

realized by a bind only on one level of the aggregation hierarchy, i.e. there are no redundant binds.

6.5.3 Binding and Unbinding Ports of Subcomponent Instances and the Parent Component

While the binds in the last subsection concerned the connections of subcomponents with each other, this section regards the connections between ports of the complex component and ports of subcomponent instances. Here, the bind (p,i_1,p_1) is to be added to the B_{2_c} of component $c = (n,P,A,S,B_1,B_2,B_3)$:

Changes to (r,H,C)	Corresponding changes to (I,B_1,B_2,A)
$H_{new} := H - \{c\} \cup \{c_{new}\}$ $c_{new} := (n_c, P_c, A_c, S_c, B_{1_c}, B_{2\,c_{new}}, B_{3_c})$	$B_{1_{new}} := B_1 \cup \bigcup_{(.,j)\in I_c} \left\{ \{(i'_1, p'_1);(i'_2, p'_2)\} \middle\| \begin{matrix}(i'_1, p'_1) \in [(j,p)] \wedge \\ (i'_2, p'_2) \in [(i_1, p_1)]\end{matrix} \right\}$
$B_{2_{c_{new}}} := B_{2_c} \cup \{(p,i_1,p_1)\}$	$B_{2_{new}} := B_2 \cup \bigcup_{(.,j)\in I_c} \{(j,p,i_1,p_1)\}$

The corresponding changes in the instance model can be quite extensive, as each port connected to port p of instances of c now has to be connected to every port connected to p_1 of each subcomponent instance i_1 of each instance of c.

Removing a bind $(p,i_1,p_1) \in B_{2_c}$ has similar consequences:

Changes to (r,H,C)	Corresponding changes to (I,B_1,B_2,A)
$H_{new} := H - \{c\} \cup \{c_{new}\}$ $c_{new} := (n_c, P_c, A_c, S_c, B_{1_c}, B_{2\,c_{new}}, B_{3_c})$	$B_{1_{new}} := B_1 - \bigcup_{(.,j)\in I_c} \left\{ \{(i'_1, p'_1);(i'_2, p'_2)\} \middle\| \begin{matrix}(i'_1, p'_1) \in [(j,p)] \wedge \\ (i'_2, p'_2) \in [(i_1, p_1)]\end{matrix} \right\}$
$B_{2_{c_{new}}} := B_{2_c} - \{(p,i_1,p_1)\}$	$B_{2_{new}} := B_2 - \bigcup_{(.,j)\in I_c} \{(j,p,i_1,p_1)\}$

Consistency is again maintained, because condition 10 of definition 6.3 demands that each subcomponent instance's port is bound to one port of the complex component at most, thus ensuring that if (p,i_1,p_1) is removed, there is no other way for the connections induced by this bind to be specified elsewhere.

6.5.4 Binding and Unbinding Parameters

Parameters of a parent component can be bound to parameters of subcomponent instances. Changing the value of such a parameter consequently might cause changes in a number of components instances on lower levels of the aggregation hierarchy. The parameters $A_{(c,p)}$ (transitively) bound to the parameter $(p,f) \in A$ of a complex component $c = (n,P,A,S,B_1,B_2,B_3) \in H$ are iteratively determined by:

$$A_0 := \{(c,p)\}$$

$$A_{i+1} := \left\{(c',p') \mid \exists_{(c,p)\in A_i} \exists_{(c',.,j)\in S_c} (p,j,p') \in B_{3c} \wedge \left((p',.,.) \in P_{c'} \vee (p',.,.,.) \in P_{c'}\right)\right\}$$

$$A_{(c,p)} := A_n \text{ with } n = min(\{n \mid A_n = A_{n+1}\})$$

Assume parameter p of a component $c = (n,P,A,S,B_1,B_2,B_3) \in H$ is to be connected to parameter p_1 of subcomponent instance i_1. , i.e. bind (p,i_1,p_1) is to be realized. One has to distinguish two cases. First assume that subcomponent instance i_1 denotes an atomic component:

Changes to (r,H,C)	Corresponding changes to (I,B_1,B_2,A)
$H_{new} := H - \{c\} \cup \{c_{new}\}$ $c_{new} := (n_c, P_c, A_c, S_c, B_{1_c}, B_{2_c}, B_{3_{cnew}})$ $B_{3_{cnew}} := B_{3_c} \cup \{(p, i_1, p_1)\}$	$A_{new} := A \cup \bigcup_{(.,j)\in I_c} \{(j+"."+i_1, p_1, v) \mid (j,p,v) \in A\}$

Parameter p_1 of subcomponent instance i_1 of all instances of c is set according to the value of parameter p of the respective instance of c.

The second case is that subcomponent instance i_1 denotes a complex component named n'. The corresponding changes are given by:

Changes to (r,H,C)	Corresponding changes to (I,B_1,B_2,A)
as above	$A_{new} := A \cup \bigcup_{(.,j)\in I_c} \left\{ (j+i', p', v) \middle\vert \begin{matrix} \exists_{(j,p,v)\in A} \exists_{(n'',p')\in A_{(n',p)}} \\ (n'', j+i') \in I \end{matrix} \right\}$

In the instance system all parameters that are bound to parameter p_1 of the subcomponent instance in all instances of c are parameterized according to the value of parameter p of the respective instance of c. In the specification of the corresponding change to the instance system, n'' denotes a component that possesses instances that have a parameter bound to p.

Removing a parameter bind $(p,i_1,p_1) \in B_{3_c}$ of complex component $c = (n,P,A,S,B_1,B_2,B_3)$ is performed analogously:

Changes to (r,H,C)	Corresponding changes to (I,B_1,B_2,A)
$H_{new} := H - \{c\} \cup \{c_{new}\}$ $c_{new} := (n_c, P_c, A_c, S_c, B_{1_c}, B_{2_c}, B_{3_{cnew}})$ $B_{3_{cnew}} := B_{3_c} - \{(p, i_1, p_1)\}$	case 1: i_1 atomic $A_{new} := A - \bigcup_{(.,j)\in I_c} \{(j+"."+i_1, p_1, v) \mid (j,p,v) \in A\}$ case 2: i_1 denotes complex component n' $A_{new} := A - \bigcup_{(.,j)\in I_c} \left\{ (j+i', p', v) \middle\vert \begin{matrix} \exists_{(j,p,v)\in A} \exists_{(n'',p')\in A_{(n',p)}} \\ (n'', j+i') \in I \end{matrix} \right\}$

Again the condition that a parameter of a subcomponent instance is either set locally or bound to the parameter of its complex component guarantees consistency.

6.5.5 Changing the Parameterization of Subcomponent Instances

While the last subsection concerned subcomponent instance parameters that are bound to parameters of complex components, this subsection deals with changing the direct parameterization of subcomponent instances. Assume the parameterization of a subcomponent instance $s = (n',i,A)$ of a component $c = (n,P,A,S,B_1,B_2,B_3)$ is to be changed by adding parameter/value pair (p,v) to A_s (again one has to distinguish the two cases of i denoting a complex or atomic component).

Changes to (r,H,C)	Corresponding changes to (I,B_1,B_2,A)
$H_{new} := H - \{c\} \cup \{c_{new}\}$ $c_{new} := (n_c, P_c, A_c, S_{new}, B_{1_c}, B_{2_c}, B_{3_c})$ $S_{new} := S_c \cup \{(n', i, A_{s_{new}})\} - \{s\}$ $A_{s_{new}} := A_s \cup \{(p,v)\}$	case 1: i atomic $A_{new} := A \cup \bigcup_{(.,j)\in I_c} \{(j+"."+i_1, p, v)\}$ case 2: i denotes complex component n' $A_{new} := A \cup \bigcup_{(.,j)\in I_c} \left\{ (j+i', p', v) \middle\vert \exists_{(n'',p')\in A_{(n',p)}} (n'', j+i') \in I \right\}$

Removing a parameter/value pair $(p,v) \in A_s$ is performed analogously:

Changes to (r,H,C)	Corresponding changes to (I,B_1,B_2,A)
$H_{new} := H - \{c\} \cup \{c_{new}\}$	case 1: i atomic
$c_{new} := (n_c, P_c, A_c, S_{new}, B1_c, B2_c, B3_c)$	$A_{new} := A - \bigcup_{(.,j)\in I_c} \{(j+"."+i_1, p, v)\}$
$S_{new} := S_c \cup \{(n',i,A_{s_{new}})\} - \{s\}$	case 2: i denotes complex component n'
$A_{s_{new}} := A_s - \{(p,v)\}$	$A_{new} := A - \bigcup_{(.,j)\in I_c} \left\{(j+i', p', v) \middle\vert \exists_{(n'',p')\in A_{(n',p)}} (n'', j+i', .,.) \in I\right\}$

Again the condition that a parameter of a subcomponent instance is either set locally or bound to the parameter of its complex component guarantees consistency. If one intends to change the value v of a parameter p to v', one first has to remove (p,v) in the fashion specified above and then add (p,v').

6.5.6 Adding and Removing Ports

Adding or removing a port of a complex component does not affect the instantiated system, assuming that – in the case of removing – the port is not connected to anything. Assume a port (p,t,pol) is to be added to a complex component $c = (n,P,A,S,B_1,B_2,B_3)$:

Changes to (r,H,C)	Corresponding changes to (I,B_1,B_2,A)
$H_{new} := H - \{c\} \cup \{c_{new}\}$	none
$c_{new} := (n_c, P_{c_{new}}, A_c, S_c, B1_c, B2_c, B3_c)$	
$P_{c_{new}} := P_c \cup \{(p,t,pol)\}$	

As prerequisite for removing a port $(p,t,pol) \in P_c$, all binds of that port to subcomponent instances within c and all binds to the port as part of a subcomponent instance of other complex components have to be removed with the help of the operations defined in the subsections above. The port removal operation is specified by:

Changes to (r,H,C)	Corresponding changes to (I,B_1,B_2,A)
$H_{new} := H - \{c\} \cup \{c_{new}\}$	none
$c_{new} := (n_c, P_{c_{new}}, A_c, S_c, B1_c, B2_c, B3_c)$	
$P_{c_{new}} := P_c - \{(p,t,pol)\}$	

While adding a port does not impact upon the instantiated system, it makes future bind operations possible, which have an impact as specified.

6.5.7 Adding and Removing Parameters

Adding and removing parameters is done analogously to the operations defined for ports and does not have an impact upon the instantiated system, as well. Assume a parameter (p,F) is to be added to a complex component $c = (n,P,A,S,B_1,B_2,B_3)$:

Changes to (r,H,C)	Corresponding changes to (I,B_1,B_2,A)
$H_{new} := H - \{c\} \cup \{c_{new}\}$	none
$c_{new} := (n_c, P_c, A_{c_{new}}, S_c, B1_c, B2_c, B3_c)$	
$A_{c_{new}} := A_c \cup \{(p,F)\}$	

As prerequisite for removing a parameter $(p,F) \in A_c$, all binds of that parameter to subcomponent instances within c and all binds to the parameter as part of a subcomponent

instance of other complex components have to be removed with the help of the operations defined above. The parameter removal operation is specified by:

Changes to (r,H,C)	Corresponding changes to (I,B_1,B_2,A)
$H_{new} := H - \{c\} \cup \{c_{new}\}$	none
$c_{new} := (n_c, P_c, A_{c_{new}}, S_c, B1_c, B2_c, B3_c)$	
$A_{c_{new}} := A_c - \{(p,F)\}$	

While adding a parameter does not impact upon the running system, it makes future parameterization and bind operations possible, which have an impact.

6.5.8 Adding and Removing Complex Components

Prerequisite for removing a complex component c is that it is not instantiated in the system, i.e. $I_c = \emptyset$, and that is not employed as subcomponent instance of another un-instantiated complex component. Thus adding and removing complex component does not impact upon the instantiated system. Assume a complex component $c = (n,P,A,S,B_1,B_2,B_3)$ is to be added:

Changes to (r,H,C)	Corresponding changes to (I,B_1,B_2,A)
$H_{new} := H \cup \{c\}$	none

Assume a complex component $c = (n,P,A,S,B_1,B_2,B_3) \in H$ is to be removed:

Changes to (r,H,C)	Corresponding changes to (I,B_1,B_2,A)
$H_{new} := H - \{c\}$	none

While adding complex components does not impact upon the system, it serves as basis for specifying new component structures (e.g. by adding and binding subcomponent instances) which can then be instantiated in the running system by adding them as subcomponent instances to another – already instantiated – complex component, e.g. the root component.

6.5.9 Changing the Set of Atomic Components

Similar to the operation of removing a complex component, the prerequisite for removing an atomic component c is that it is not instantiated in the system, i.e. $I_c = \emptyset$, and that is not specified as subcomponent instance of an un-instantiated complex component. Thus adding and removing atomic component does not impact upon the instantiated system. Assume an atomic component $c = (n,D,P,A)$ is to be added:

Changes to (r,H,C)	Corresponding changes to (I,B_1,B_2,A)
$C_{new} := C \cup \{c\}$	none

The newly added atomic component can now be used as subcomponent instance in the specification of an instantiated complex component and thereby be instantiated in the system.

Assume that an atomic component $c = (n,D,P,A) \in C$ is to be removed:

Changes to (r,H,C)	Corresponding changes to (I,B_1,B_2,A)
$C_{new} := C - \{c\}$	none

6.5.10 Aggregated Tailoring Operations

The set of tailoring operations presented so far is complete in the sense that (disregarding naming issues) any valid representation (r,H,C) of a component-based system can be transformed into any other valid representation (r',H',C'). Thus the operations suffice to realize the full expressiveness of component-based tailorability.

However, in many cases it might be useful to provide operations like the exchange of one component with another, or the removal of a component that is still bound to other components. These operations can be expressed as sequential executions of the elementary tailoring operations presented in this section.

6.6 Scope Control

The tailoring operations specified in the last section permit changes to become effective immediately in all instances of shared complex components. This addresses the first part of the chapter's central question. However, as illustrated by the search tool example, there are situations in which it is not desirable to let a change affect all instances of a shared complex component. This section presents an additional operation which permits limiting the scope of a change to a subset of instances of the shared complex component while maintaining consistency between the representation (r,H,C) and the instance system (I,B_1,B_2,A), thus addressing the second part of this chapter's question.

Assume a change of a shared complex component $c = (n,P,A,S,B_1,B_2,B_3)$ is about to be executed, consisting of a sequence of tailoring operations. However, the change is supposed to affect only a subset $I'_c \subset I_c$ (e.g. because the instances in I_c - I'_c are used by others, who are not supposed to be affected by the changes).

The idea behind the approach to scope control presented here is to create a copy c' of the shared complex component c and change the representation (r,H,C) in a way that the instances in I'_c are instances of c' and that the instances in $(I_c - I'_c)$ are instances of c. The pointers in the instance structure have to be changed accordingly. The intended tailoring operations are now executed on complex component c' and consequently only affect the instances in $I_{c'}$. Thus the scope is limited to these instances formerly in I'_c. The effect of the scope control operation is illustrated in figure 6.3 with an example:

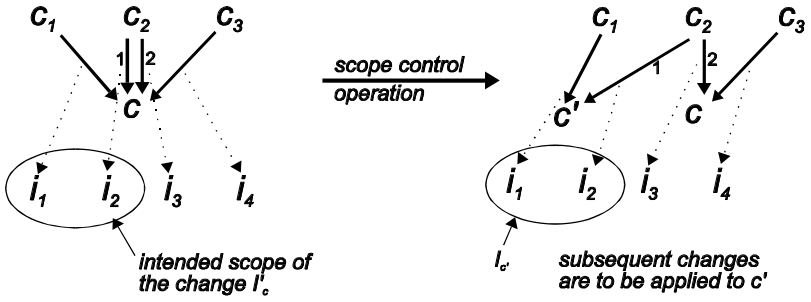


Figure 6.3: Applying the scope control operation

Complex component c is instantiated as subcomponent in components c_1, c_2 and c_3 – even twice in c_2. A number of tailoring operations are to be executed on c. However, they are supposed to affect only instances i_1 and i_2. As described above, the scope control operation consists of creating a copy c' of c and changing component c_1 and c_2 in order to instantiate c' as subcomponent instead of c. The instance system is changed accordingly, i.e. without actually changing i_1 and i_2 they are registered as instances of c'. This maintains consistency, because c' is an exact copy of c.

In the following the scope control operation is specified in terms of the model. Let $\tilde{n} \in N_{components}$ be an unused component name for denoting component c'. First, one has to determine which complex components specify the instantiation of component c (named n) as a subcomponent instance (the c_1 to c_3 in the example of figure 6.3). Furthermore, one needs the instance name i of c as subcomponent in the context of these complex components:

$$F_c := \{(n',i) \mid (n,j+"."+i) \in I'_c \wedge (n',j) \in I\}$$

F_c thus is a set of tuples (n',i) that represent all instances i of components n' that use component c (named n) as subcomponent. Assume that each complex component f referred to in F_c is not shared, i.e. $|I_f| = 1$. Otherwise the approach described in this section does not work, because it does not take into account sharing on higher levels of the component hierarchy (in the implementation described in chapter 8, scope control operations are only applied to the highest level of the hierarchy, i.e. subcomponents of the obviously unshared root component of the whole system).

Each complex component referred to in F_c (by name n') is responsible for creating one or more of the instances in I'_c as subcomponent instances of itself. These subcomponent instances have to identified and "re-directed" to the new component name $\tilde{n}$.

$$\Delta S_{old}(n') := \{(n,i,A') | (n',i) \in F_c \wedge (n',.,.,S',.,.,.) \in H \wedge (n,i,A') \in S'\}$$

This set identifies all relevant subcomponent instances of a complex component with name n'.

$$\Delta S_{new}(n') := \{(\tilde{n},i,A') | (n,i,A') \in \Delta S_{old}(n')\}$$

This set contains the subcomponent instances pointing to the new name $\tilde{n}$. Finally, the set of complex components is changed by replacing the new versions of the complex components referred to in F'_c and adding the new complex component c' which is named $\tilde{n}$, but otherwise identical to c.

$$\Delta H_{old} := \{(n',P',A',S',B'_1,B'_2,B'_3) \mid (n',P',A',S',B'_1,B'_2,B'_3) \in H \wedge (n',.) \in F_c\}$$

$$\Delta H_{new} := \{(n',P',A',S_{new},B'_1,B'_2,B'_3) \mid (n',P',A',S',B'_1,B'_2,B'_3) \in \Delta H_{old} \wedge S_{new} = S' - \Delta S_{old}(n') \cup \Delta S_{new}(n')\}$$

$$H_{new} := (H - \Delta H_{old}) \cup \Delta H_{new} \cup \{(\tilde{n},P,A,S,B_1,B_2,B_3)\}$$

This concludes the changes to (r,H,C). The corresponding changes to the instance system (I,B_1,B_2,A) only concern the component names referred to in the old set I'_c.

$$I_{c_{new}} := \{(\tilde{n},i) | (n,i) \in I'_c\}$$

$$I_{new} := (I - I'_c) \cup I_{c_{new}}$$

Now the intended tailoring operations can be applied to c' taking an effect only on the elements in $I_{c'}$.

6.7 Summary and Discussion

The question addressed in this chapter was how to design the connection between the representation of a component-based system and the instantiated system in a way that makes sharing of complex components possible, while:

- permitting changes to become effective immediately in all instances of the shared complex component
- permitting the restriction of a change's scope to a given subset of these instances.

These requirements were not met by the (state-of-the-art) solution employed in the search tool example.

Section 6.3 presented a formal model of the representation of a component-based system with the possibility of sharing complex component. The model reflects the constructs of the CAT composition language. Section 6.4 specified the process of instantiating such a representation during system start-up and models the resulting instantiated system. Section 6.5 presented a solution for the first requirements (changes taking effect in all instances immediately) by specifying a number of tailoring operations applied to the representation of

component-based system and the corresponding changes to the instantiated system. Section 6.6 presents a solution for the second requirement (restricting the scope of a change to a subset of instances), by specifying an operation that changes the representation in a way that restricts the effect of future tailoring operations to a given subset of instances.

The solutions developed in this chapter are presented in a programming-language independent form, based on set theory and first order predicate logic. They are supposed to guide the development of component-based runtime and tailoring environments in an arbitrary programming language (assuming that this language supports reflection and dynamic loading of components, as argued in chapter 3). Chapter 8 presents the implementation of such an environment in the object-oriented programming language JAVA, providing both runtime tailoring operations on shared complex components and scope control. During the development of this environment, the formal models and specifications developed in this chapter proved quite useful. The basic set-theoretic approach of the model made it easy to identify the basic data structures needed for the runtime tailoring and scope control operations and to design an object model correctly implementing them.

Chapter 7

The FLEXIBEANS Component Model

7.1 Introduction

This chapter provides an overview of the FLEXIBEANS component model. The FLEXIBEANS component model is an adaptation of the JAVABEANS model, with additional support for shared variable interaction primitives (implemented as shared objects), port names, and remote interaction. Implementation in the FLEXIBEANS component model is a prerequisite for a component to be deployed in the EVOLVE environment presented in the next chapter. The requirement to support shared variables stems from the theoretical comparison of different interaction primitives presented in chapter 5, while the second and third requirement are direct results of the question raised by the case study described in chapter 3 ("*How can the JAVABEANS component model be adapted to support the concept of port names and remote interaction?*"). A simple example of an application implemented as a set of FLEXIBEANS is given.

7.2 Interaction Via Shared Objects

The analysis of the case study presented in chapter 4 demonstrated the deficiencies of the JAVABEANS component model for the purpose of the component-based tailorability approach developed in this dissertation. One problem was the limitation to JAVA events as the only dynamically re-wireable interaction primitive. JAVA events only permit a push like interaction style between components. This either leads to an awkward and inefficient implementation of interactions (in cases when a pull like interaction would be more appropriate) or to unwanted changes in the functional decomposition. The latter effect has a negative impact upon the future extensibility of a component set, because concerns that are expected to evolve separately are placed in the same component.

The main objective of the theoretical investigation of chapter 5 was to identify a set of interaction primitives that permits the efficient and natural implementation of arbitrary decompositions. The main criterion for evaluating how well an intended interaction can be implemented with a certain set of interaction primitives was how closely the implemented synchronization of activities of both components resembles the intended synchronization. In a high quality implementation, the initiative for pursuing an interaction always resides with the correct (intended) component and each component only needs to actively participate in the interaction when it is its intended turn. Unfaithful synchronization was identified in the case study as the major culprit for the problems arising when using only JAVA events. Chapter 5 models a number of different interaction primitives (method calls, JAVA events, shared variables, observables, synchro-cells) and subsequently shows that – in the context of the formal model – with a combination of JAVA event and shared variable interaction primitives one can always achieve a high quality implementation of arbitrary decompositions. This is not possible with events or shared variables alone.

Based on this theoretical result, the FLEXIBEANS component model supports shared variable interaction in addition to JAVA events. The concept of shared variables is realized with the help of objects that are shared between two or more components. Apart from describing this object-oriented realization, the following subsections present the method signature patterns for indicating shared object interaction ports, discuss concurrency issues, and propose possible usage scenarios for the primitive.

7.2.1 Shared Objects

In an object-oriented programming language like JAVA, the shared variable interaction primitive can be realized as a shared object that encapsulates the state of the shared variable. The state can be read and changed via method calls specified in the object's interface. This interaction primitive is symmetric, i.e. both components can indiscriminately call all methods specified in the interface. Figure 7.1 depicts the static structure of this interaction primitive:

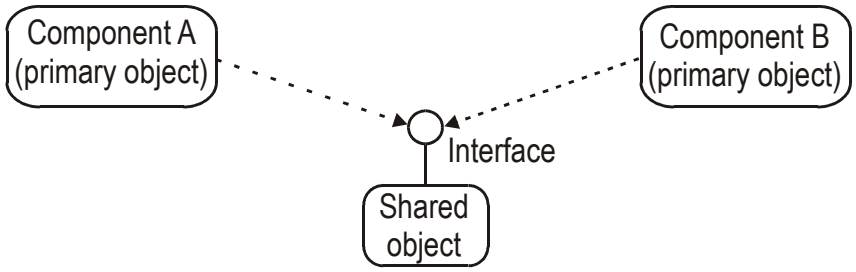


Figure 7.1: Basic object-oriented realization of a shared variable interaction primitive (the representation of the interface follows UML's "lollipop" notation)

Assuming both components (instances) are supposed to share a text string, the respective shared object could be realized implementing the following JAVA interface (the significance

of the `RemoteException` and the extension of the `Remote` interface is explained in the description of JAVA RMI in chapter 3):

```
public interface Text extends java.rmi.Remote {
    public String getText ( ) throws java.rmi.RemoteException;
    public void setText (String text) throws
                    java.rmi.RemoteException;
      }
```

The shared object itself can be every object that implements the interface `Text`. The component instances can obviously only change the shared object's state via the methods declared in the interface. Beyond simply setting or getting the state of the shared variable like in the `Text` interface example, more sophisticated update and read operations are conceivable. Regard the following interface that specifies the methods to access a shared list:

```
public interface List extends java.rmi.Remote {
     public void addEntry (Entry e) throws
                 java.rmi.RemoteException;
     public void changeEntry (Entry old, Entry changed) throws
                 java.rmi.RemoteException;
     public java.util.Vector getList ( ) throws
                 java.rmi.RemoteException;
     public void removeEntry (Entry e) throws
                 java.rmi.RemoteException;
      }
```

Even though the shared variable interaction primitive is symmetric by nature, the FLEXIBEANS component model assigns the responsibility to instantiate the shared object to one of the components (the *provider* of the shared object). In the graphical representation, the port providing the shared object is represented as a filled-in rectangle as depicted in figure 7.2:

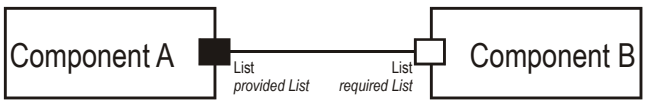


Figure 7.2: Graphical representation of the shared object connection.

The port requiring a shared object of a specific type is represented by an empty rectangle. The next subsection covers the method signatures that the primary class of a component has to adhere to in order to indicate provided and required shared object ports of a certain type.

7.2.2 Signature Patterns

A component providing a shared object of a certain type indicates this port by implementing a method with the following signature (the representation of the signatures follows the conventions in the JAVABEANS specification):

```
public <Interface> getShared<Interface>_<Port name>();
```

`Interface` is the name of the interface of the shared object. The concepts for naming ports are introduced in a later section. Component A of figure 7.2 provides a port named `providedList` of type `List` and indicates this port by implementing the following method:

```
public List getSharedList_providedList();
```

One can obtain a list of the provided shared object ports of a component's primary class file by inspecting all method signatures beginning with `getShared`. By calling such a method, the environment receives a reference to an object implementing the respective interface. This reference can be passed on to a component instance requiring a shared object of the specified type.

A component requiring a shared object of a certain type indicates this port by implementing a pair of methods with the following signatures:

```
public setShared<Interface>_<Port name>(<Interface> o);
public forgetShared<Interface>_<Port name>();
```

The object reference obtained by calling the `getShared...()` method of the shared object provider is passed to the component requiring a shared object by calling the `setShared...()` method. Again regarding the example in figure 7.2, the method signatures appear like:

```
public setSharedList_requiredList(List o);
public forgetSharedList_requiredList();
```

Sometimes it can be useful to employ one of the primary objects of the components to implement the interface in order to restrict the proliferation of objects in the running system. The realization in this case is depicted in figure 7.3:

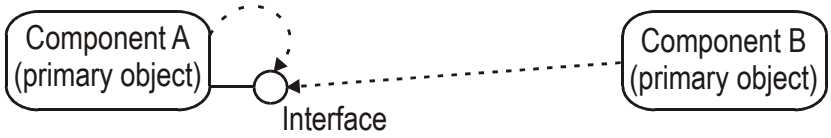


Figure 7.3: The primary object of component instance A implements the interface itself and thus serves as shared object

Here the `getShared...()` method of the primary class of component A simply returns a self-reference. Obviously, the primary class now has to implement the methods of the interface itself. The implementation of the shared object provided by component A is transparent to component B.

7.2.3 Concurrency Issues

Since JAVA supports multithreading, one has to consider the "classical" problems of concurrent access to shared resources when implementing shared object interaction. Regard a simple integer counter that is to be implemented as a shared object with an interface providing a `read` and a `set` method (the example is adapted from Silberschatz and Galvin 1998). The example is depicted in figure 7.4 (using a notation similar to the *interaction diagrams* provided by UML):

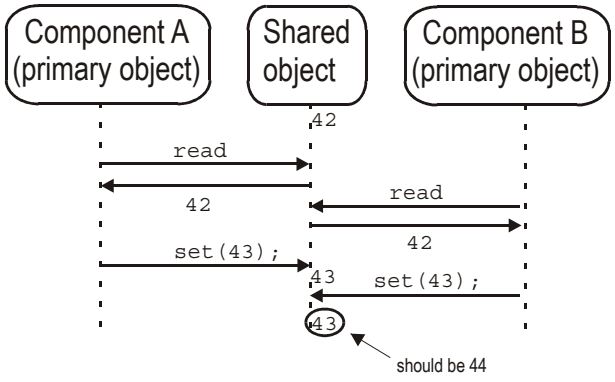


Figure 7.4: Concurrent access to a shared object can result in inconsistencies

Components A and B both intend to increase the counter (current state: 42) by one. This should result in a final counter state of 44. However, each one reads the same state of the counter, computes the new state (43) and calls the `set` method. Consequently, the set method is called twice with the same value, resulting in a final counter state of 43 that is – obviously – not the intended result 44.

There are a number of general solutions to this problem given in the literature (compare e.g. Silberschatz and Galvin 1998). One could, for instance, provide an atomic `increase()` method in the shared object's interface. In this case atomicity means that the method body of `increase()` can only be traversed by one thread at a time which can be achieved by employing the `synchronized` keyword provided by JAVA. Another conceivable solution would be an atomic `test_and_set(state old, state new)` method which checks whether the shared object is still in the assumed state `old` and only then changes the state to `new`. If another component has managed to change the state such that the current state ≠ `old` then the method throws an exception and the executing component can act accordingly (e.g. check the state again and re-compute).

As concurrency problems of this type have already been exhaustively dealt with, the reader is advised to consult the literature (Silberschatz and Galvin 1998) when designing a shared object that is subject to concurrent access (note that many applications will only use shared objects to let data "flow" in one direction between two components, thus not raising concurrency problems).

7.2.4 Using Shared Objects

The extension of the basic set of interaction primitives of the FLEXIBEANS component model with shared objects was initially motivated by the problems encountered in the exploratory case study described in chapter 4. The subsequent theoretical treatment in chapter 5 suggests JAVA events and shared objects together as an appropriate set of interaction primitives permitting the natural and efficient implementation of arbitrary decompositions.

One concrete problem encountered in the case study was the inefficient implementation of the interaction between the document type component and the search engine (see left side of figure 7.5):

Figure 7.5: Interaction between the document type component and the search engine with events (left) and with shared objects (right).

The problem with the implementation using JAVA events (left of figure 7.5) is that the search engine is informed of every change in the attribute (document type) component, even though the search engine only requires the information when a search is about to be executed. With a shared object (right of figure 7.5) this interaction can be implemented more naturally and efficiently. The search engine only reads the search object when it needs the information, thus avoiding (maybe several) unnecessary event transmission.

Another concrete problem was the uncalled-for composition of the concerns of result visualization and result handling in the visualizer component. By providing the selected item in the visualizer list as a shared object the result handling functionality could be – more appropriately – placed in another component (here: e.g. the copy button component). The advantage of the thus achieved separation of result handling and visualization concerns is that additional – not anticipated – result handling functionality can be added to the system without having to change the visualizer component which might only be available in binary code. The example at the end of this chapter uses this technique to manipulate marked entries in a shared to do list.

Attempting to (informally) generalize the appropriate usage of the shared object interaction primitive, one could say that shared objects are best used, whenever data has to be shared

symmetrically between components or when a data flow has to be initiated by the recipient of the data (pull-like interaction). It is important to note, that shared object interaction cannot transfer the flow of control between component instances (If the shared object is implemented by one of the participating components, this obviously does not hold).

7.3 Interaction Via JAVA Events

In addition to shared objects, the FLEXIBEANS component model also permits component interaction via JAVA events. The implementation of these events essentially follows the JAVABEANS specification (JavaSoft 1997). Chapter 3 gives an overview of how event interaction is implemented there. However, there is one difference in the specification of the event listener component, which is due to the fact that FLEXIBEANS are supposed to support port names and remote interaction.

Originally, a JAVABEAN indicates the fact that it can receive events of a certain type by implementing the appropriate event listener interface, e.g. the `ActionListener` interface for receiving certain GUI events (see Figure 7.6-a). All events of that type – irrespective of their source – are handled by the event handling methods specified in that interface. Consequently, JAVABEANS cannot directly distinguish different sources sending out events of the same type (this has to done programmatically within the component or a special adapter object).

In contrast to JAVABEANS, FLEXIBEANS indicate an event listener port for a certain type and name by providing a method with the following signature:

```
public <Interface> getEventPort<Interface>_<Port name>();
```

The method returns a reference to an object that implements `Interface`. If the component is supposed to implement more than one port of the same type, it needs to provide special port objects (i.e. instantiate them during the instantiation of the component) that serve as intermediate event listeners in turn calling the appropriate methods of the primary component object. This approach is depicted in figure 7.6-b.

The FLEXIBEANS approach is similar to the adapter design pattern suggested for this purpose in the JAVABEANS specification. However, by setting up the structures shown in figure 7.6-b during the instantiation of the primary object of component B and by providing the `getEventPort...()` methods, one can quickly change the event wiring without having to create and compile (adapter) glue code or use a slow generic adapter (compare the discussion in chapter 4).

Additionally, figure 7.6-c depicts how one can save one port object by having the primary object of the component itself play the role of event listener for one of the named port (here port T_2). All ports that have a unique type (i.e. there are not other ports of that component with the same type) should anyhow be implemented like port T_2 in order to avoid the proliferation of objects in the system.

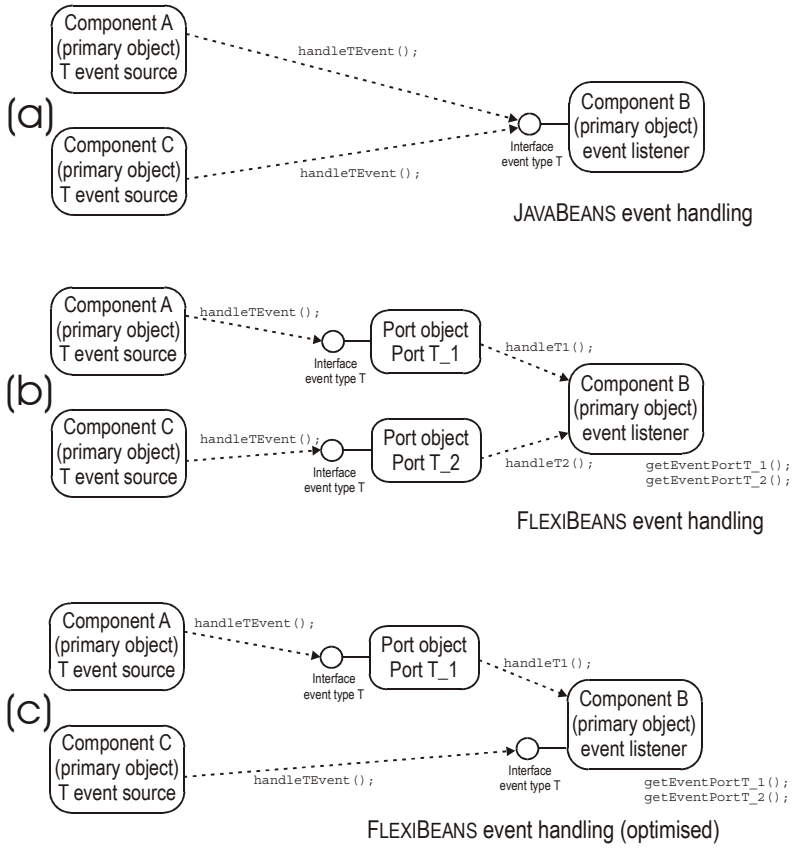


Figure 7.6: (a) JAVABEANS handle events of the same type on one port. (b) FLEXIBEANS can handle these events on different ports. (c) The primary object of the component can also serve as one port object in order to reduce the number of objects in the running system.

The method signatures for identifying the event source ports of a component are similar to the one specified for JAVABEANS:

```
public void add<Interface>_<Port name>(<Interface>
listener);

public void remove<Interface>_<Port name>(<Interface>
listener);
```

The `Interface` is the JAVA event listener interface. The add and remove methods are used by the environment to connect and disconnect event listeners and event sources.

7.4 Port Names

Apart from the problems with the JAVA event interaction primitive, the exploratory case study also demonstrated the need for port names in order to enhance the expressiveness of the compositional level. Therefore, FLEXIBEANS support port names. The method signatures given in the preceding sections for indicating shared object and event interaction ports are always appended by

```
_<Port name>
```

in order to indicate the name of the port. Consequently, it is possible to distinguish between different ports of the same type and polarity (see e.g. the two T event listener ports in figure 7.6-b).

7.5 Remote Interaction

While the search tool example in the exploratory field study was only a small, local part of a groupware system, these systems are usually distributed across a network (e.g. the Internet or intra-organizational networks). Therefore, component instances have to be able to interact across machine boundaries. Component A might reside on a computer in Bonn, while component B resides on a computer in Tokyo. Assume that component A provides a shared object containing some data to be shared with component B. Furthermore, component A sends events whenever the data changes in order to inform interested parties of that change. Figure 7.7 depicts this example scenario:

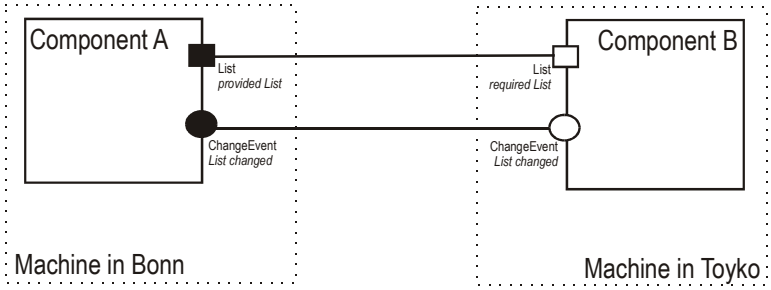


Figure 7.7: In order to support distributed groupware applications, the FLEXIBEANS interaction primitives need to support interaction across machine boundaries

In order to maintain the benefits of component-based tailorability demonstrated in the local setting during the case study, remote interaction should meet the following requirements.

- In order to permit quick changes to the application, it should be possible to connect two remote components **without having to dynamically generate and compile code** for the remote interaction.
- Equipping a component with the capability for remote interaction **should not impact upon the implementation of the component's intended functionality** (i.e. possibly changing the decomposition).
- Implementing remote capability should only be a **negligible additional burden for the component developer**.
- In case the two component instances are placed on the same machine, the **interaction should work locally as well**. Furthermore, multi-cast like interaction (i.e. many event listeners or many component instances sharing the same object) should be possible with remote and local component instances at the same time.

The FLEXIBEANS component model employs JAVA Remote Method Invocation technology (RMI) to make the event and shared object interaction primitives remote capable. JAVA RMI is described in detail in chapter 3. The integration of FLEXIBEANS and RMI is designed to meet the requirements stated above. The following subsection shows how the FLEXIBEANS interaction primitives are made remote-capable.

7.5.1 Remote JAVA Events

In order for the event listener and the event source to reside in different virtual machines, the method call from source to listener, which forces the latter to react to the event has to be executed remotely. Consequently, the event listener interface has to extend the `java.rmi.Remote` interface and its methods have to be specified to throw `java.rmi.RemoteException`.

Furthermore, on the side of the event listener, the object actually acting as event listener (either a port object or the primary object of the component instance itself – compare figure 7.6-b and -c) should extend the class `java.rmi.server.UnicastRemoteObject` in order to make its event handling method(s) remotely accessible. Note that using port objects makes

it possible for the primary class of the object to extend an arbitrary class. Thus there are no restrictions as to which components can be made remote capable as event listeners.

On the side of the event source, the method calling the event handling method of the event listener has to be prepared to handle exceptions of type `java.rmi.RemoteException`. Consequently, the method call either has to reside inside a `try/catch` block sensitive to this type of exception, or the calling method has to be able to throw exceptions of that type (or a supertype) itself.

Finally, the event object used as parameter of the remote method call has to implement the interface `java.io.Serializable` in order for it to be passed by value and not by reference (otherwise it would have to be a remotely accessible object itself). An object implementing this interface permits its complete state to be converted to a byte stream (this process is called *serialization*) and reconstructed into an object again. Thus a serializable object can be copied to another virtual machine, which is precisely what happens with the event object parameter during a call to the remote event handling method.

7.5.2 Remote Shared Objects

The basic pattern of interaction via shared objects is depicted in figure 7.1. The shared object itself always resides in the virtual machine of the component instance that is responsible for its instantiation. The interface for accessing the shared object also extends `java.rmi.Remote` and its methods throw `java.rmi.RemoteException`. Consequently, methods accessing the shared object have to be prepared to handle exceptions of that type (in the same fashion as exceptions caused by remotely calling event handling methods – see above). The shared object itself should extend the class `java.rmi.server.UnicastRemoteObject`.

7.6 Example: Shared To-Do Lists

While the preceding sections presented the object-oriented implementation of the FLEXIBEANS component model, this section gives an example how a distributed groupware application can be built from a set of FLEXIBEANS components.

Shared to-do lists (see e.g. Kreifelts et al. 1993) support coordination of work activities in small groups (2-10 persons). They contain entries that describe a task to be done, its title, begin, deadline, and a flag indicating the status of completion (in progress, completed). Depending on the structure of the group and its work habits, there can be distinct group members who add tasks, perform tasks, check for completion, and monitor or distribute work. In groups with a flat hierarchy, everybody might be allowed to add, mark as done, delete, and monitor tasks. In more hierarchical organizations, only the manager might be allowed to add tasks, while subordinates are only supposed to indicate completion. As groups evolve, members fluctuate, and group tasks change, the configuration of the application has to change, new lists have to be introduced for subgroups and old lists become obsolete.

The shared to-do list application presented here is highly simplified for presentation purposes. However, applications with the same basic functionality are used in IT support departments, call centers, and generally for coordination in small groups that are confronted with a lot of short-term, well-defined tasks.

The shared to-do list component set consists of four visible and one invisible component. Table 7.1 gives an overview and informally describes the semantics of each component. The components rely on three different interaction types: a remote shared object `RemoteList` that contains a list of tasks, another remote shared object `RemoteEntry` that contains a single list entry, and a remote event `RemoteActionEvent` that is used to notify other components of changes in a specific to-do list.

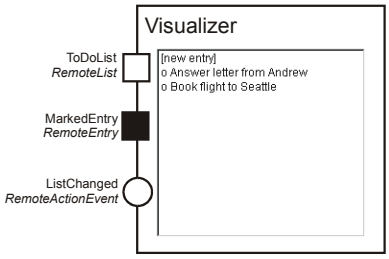
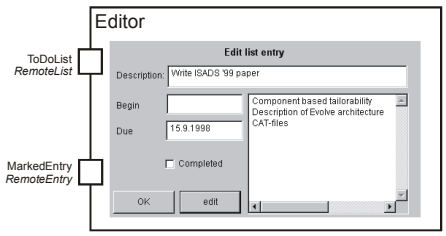
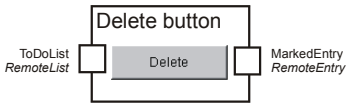
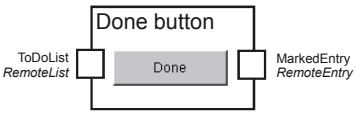
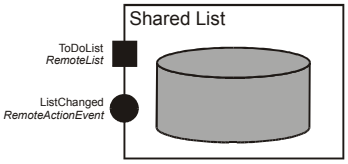
	**The *visualizer* component** This component is visible for the user and displays the current contents of a shared list. It has three ports: the first port connects to a shared list component, the second port receives events that indicate a change in a shared list, and the third port shares the currently marked entry in the list with every interested (i.e. connected) component.
	**The *editor* component** This component is actually a complex component that is composed of several visual subcomponents. For simplicity it is regarded as atomic here. The two ports connect to a shared list and the marked entry of a *visualizer* component. The user can add new entries to the list (if the content of marked entry is [new entry]) or edit other selected entries in the list. The large text box on the right can be used to describe the task.
	**The *delete button* component** This component is connected to a shared list and a marked entry and – if pressed – deletes the marked entry in the list
	**The *done button* component** This component is connected like the delete button. When pressed, it sets the flag of the marked entry to “completed”.
	**The *shared list* component** This component is the only invisible component. It usually resides on a server and maintains a list of tasks. The list is shared via the ToDoList port and other components are notified of changes via the ListChanged event port.

Table 7.1: Components of the shared to-do list framework

The set of components described in table 7.1 can now be used to compose a distributed shared to-do list application. Figure 7.8 shows an example application built with the component set:

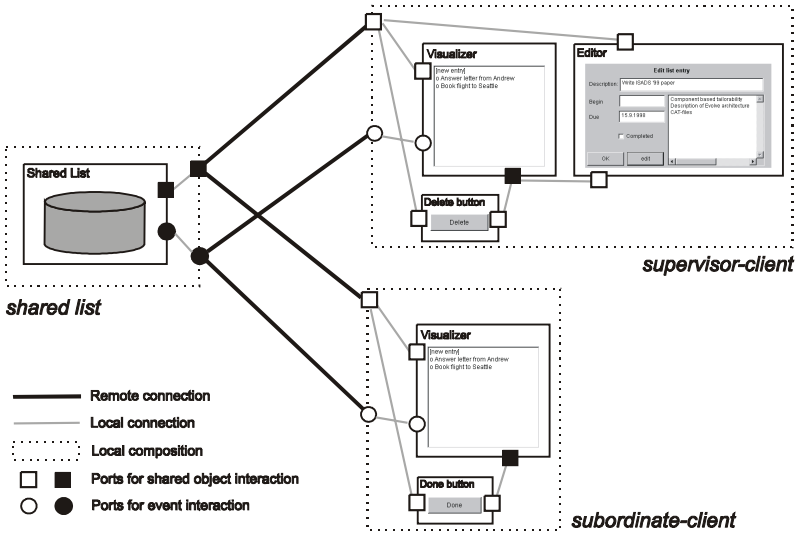


Figure 7.8: Example application providing a shared to-do list between a supervisor and a subordinate.

The application depicted in figure 7.8 is distributed over three locations. The *shared list* component instance resides on a server and is connected to an instance of the *supervisor client* on one machine and to an instance of the *subordinate client* on another machine. The different requirements of supervisor and subordinate are met by different compositions of components taken from the set of table 7.1.

One can imagine several ways in which the requirements posed upon the whole application might change. For instance, if a second subordinate joins the group, another instance of the subordinate client would have to be introduced on a fourth machine. Or it could be decided that the subordinate deletes finished tasks directly. In this case the done button component in the subordinate client would have to be substituted by a delete button component.

7.7 Summary and Discussion

This chapter described the FLEXIBEANS component model as an adaptation and extension of the JAVABEANS component model. Its main constituent is the object-oriented realization of the two interaction primitives *shared variable* and *JAVA event*, the latter already being supported in the JAVABEANS model. A central feature is the integration of JAVA RMI technology in order to provide remote interaction capability to both interaction primitives. Furthermore, the introduction of port names extends the expressiveness of the model on the compositional level.

A component designed to comply with the FLEXIBEANS component model has the following characteristics:

- All its **ports** and **parameters** can be detected by analyzing the signature patterns of the methods of the primary class of the component. For each port, the signature patterns indicate the type of interaction primitive (*shared object* or *event*), the type of the data involved (i.e. the type of the event or shared object), the polarity of the port (*provided* or *required*), and the name of the port. For each parameter, the patterns indicate its type and its name.
- Employing the dynamic class loading facility of JAVA, a component that did not exist at system start-up time can be **loaded and instantiated** in the running system.
- A component instance port **can be connected and disconnected during runtime** to a (compatible) port of another (or the same) component instance by calling standardized methods. There is no need to generate and compile glue code, even for remote interactions.
- A parameter can be **read and set during runtime**, again by calling standardized methods.

These characteristics provided by the FLEXIBEANS component model permit the EVOLVE environment described in the next chapter to manipulate the component structure of a running distributed groupware application. The integration with JAVA RMI supports an arbitrary distribution of interacting components over a network. The supported set of interaction primitives – in theory – supports the natural and efficient implementation of the decomposition of an application's functionality as a set of components. The latter claim is substantiated in chapter 9 by demonstrating how a number of groupware systems were implemented as sets of FLEXIBEANS.

Chapter 8

The EVOLVE Platform

8.1 Introduction

This chapter presents an object-oriented implementation of the concepts developed before. The EVOLVE runtime and tailoring platform is designed to provide component-based tailorability for distributed groupware applications on the Internet. After an overview of the central elements of the implementation, a detailed account of the system's internal representation of distributed component structures is given. The following subsection discusses the object-oriented architecture of the platform. The architecture provides runtime sharing of adaptations, scope control and multiple levels of complexity. The tailoring operations supported by the architecture can be accessed via a tailoring API. This API can be employed to connect different types of tailoring interfaces to the platform. As an example, the last subsection presents a 3D interface that permits runtime adaptations and navigation through a component scene visualizing the whole distributed groupware system.

8.2 Overview

The reason for the implementation of the EVOLVE runtime and tailoring environment described in this chapter is to demonstrate the technical feasibility of the architectural concepts developed in chapter 6 and to serve as basis for further evaluation concerning the applicability of the FLEXIBEANS component model. Since the motivation for this dissertation originated from the field of CSCW, the EVOLVE runtime and tailoring environment is designed specifically to support groupware systems. However, the concrete requirements of these systems on the platform level are very similar to that of many multi-user systems that are distributed over a network of physical machines, namely:

- support for client-server applications
- dynamic client logon / logoff
- client can be an applet in a web-page
- works on top of the TCP / IP network protocol

Consequently, the resulting platform can also be employed to support other software systems, as demonstrated in chapter 9. Figure 8.1 depicts the constituent elements of the EVOLVE platform:

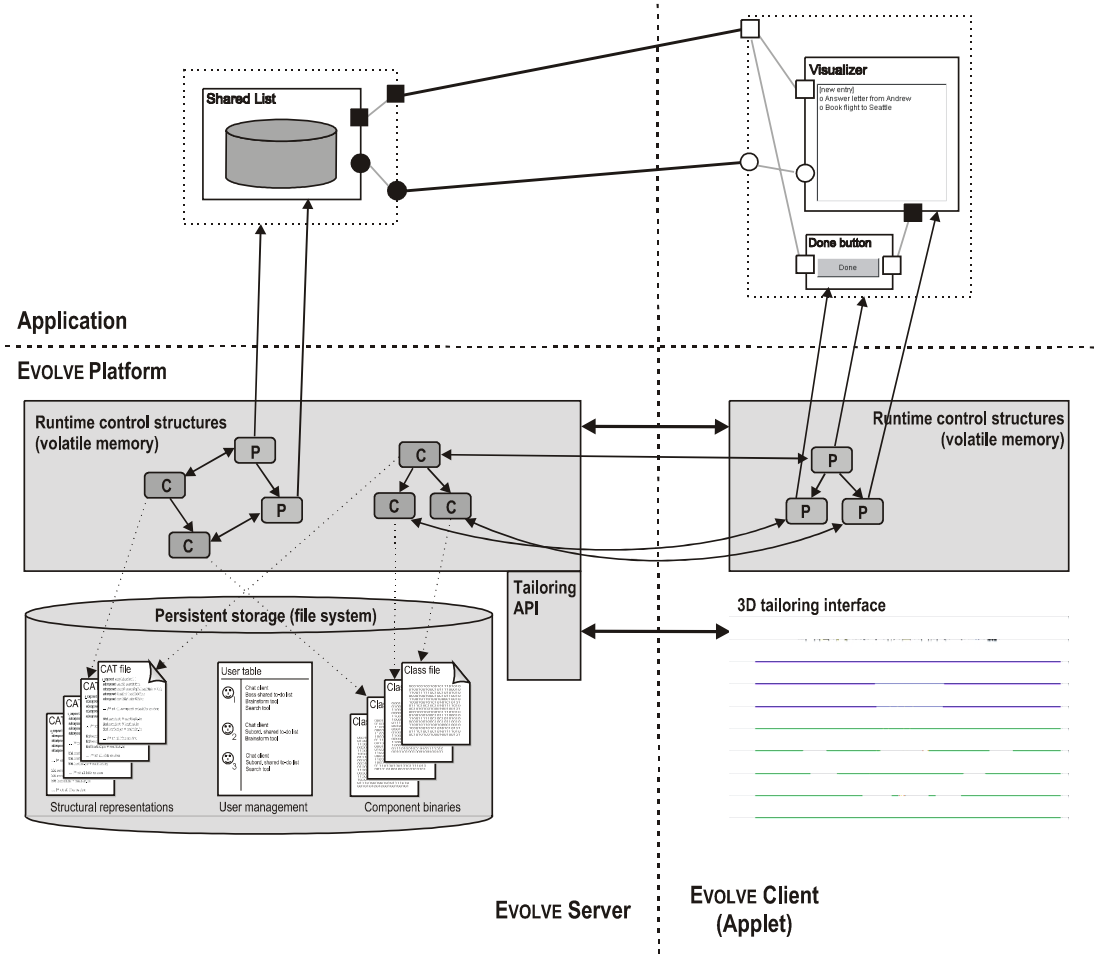


Figure 8.1: Elements of the EVOLVE platform's architecture

A central feature of the EVOLVE environment as depicted in figure 8.1 is the clear separation between the *platform* (bottom) and the *supported groupware application* (top). This separation enables the *same* platform (and tailoring interface) to be used for many *different* groupware applications. While the following always employs the simple shared to-do list

example of chapter 7 in order to explain the technical concepts of the architecture, chapter 9 presents some more demanding applications of the EVOLVE environment.

It is assumed that a groupware system initially consists of a set of FLEXIBEANS (which are essentially JAVA binaries, see lower left of figure 8.1) and a set of client and server compositions as CAT files (*structural representations*). A *user table* defines which user may use which client compositions. Together with information on how to connect client and server component structures, these data represent the whole distributed groupware system. Section 8.3 describes this representation scheme in more detail.

When the EVOLVE *platform server* (left side of figure 8.1) is started up, the CAT files for both client and server structures are evaluated and represented in the main memory of the server as component object trees (the object structures in figure 8.1 which are denoted with "*C*"). The leaves of these trees refer to component binaries, the inner nodes to complex components. The server component structures are then instantiated according to these volatile representations. In the shared to-do list example, only a single binary component (the *shared list*) has to be instantiated. Instantiation is managed with the help of proxy object structures (denoted with "*P*"). The "*P*" object structures are exact copies of the "*C*" structures. However, the "*P*" objects represent *instances* of components, thus for every "*C*" object there can potentially be several proxy objects (the "*P*" and "*C*" structures are the object-oriented implementation of the *component system* and the *instance system* which are formally defined in chapter 6).

When a user starts an EVOLVE *platform client* (right side of figure 8.1), then – via the user table – the platform determines the client compositions the user may access and instantiates them on the client machine. This is done in the same fashion as with the server components, with the sole exception that the proxy object structures ("*P*") reside on the client machine. In the example application, the subordinate client composition is instantiated and connected to the *shared list* component instance on the server.

During tailoring, the "*C*" structures are manipulated. All changes are directly broadcasted to all instances, i.e. "*P*" structures, which apply them to the running groupware system. Section 8.4 describes the object structures and their interactions during the different phases (start-up, login etc.).

Tailoring operations are defined as a set of public methods provided by the objects of the "*C*" structures and offered – remotely – via the *Tailoring API* (lower middle of figure 8.1). Section 8.5 gives an overview of the defined operations. They are a superset of the complete set of operations formally defined in chapter 6.

The Tailoring API is used to connect arbitrary tailoring functionality (lower right of figure 8.1) to the platform. While section 8.6 describes an example of a user interface for tailoring, one can also imagine an intelligent agent in its place that automatically adapts the groupware system to changing requirements.

More details of the implementation and a discussion of the object-oriented design patterns can be found in (Hinken 1999).

8.3 The Representation of Distributed Software Systems

In contrast to the simple search tool example presented in chapter 4, the EVOLVE platform is supposed to support whole distributed groupware systems, whose overall instance structure depends on which users are currently logged in (dynamic client login/logoff). The distribution and the structural dynamics are reflected in the representation scheme used to persistently describe the system. Instead of employing a single compositional specification (i.e. a single CAT file), the structure of the system is described by five different types of files:

- *Server CAT files* describe the composition of system parts which reside on the server. Such a file describes the shared list component in the example in figure 8.1. Server CAT files can obviously only refer to invisible components, i.e. components which do not have a representation on the user interface
- *Client CAT files* describe the compositional structure of the client part of the system. Such a file describes the subordinate client in the example. Client CAT files can contain visible and invisible components. The visible component instances are all mapped onto an operation system window.
- *Remote bind files* define how the ports of a Client CAT file are to be connected to the ports of a Server CAT file.
- *DCAT files* tie the whole distributed application together by specifying a system component which has as its subcomponents both client and server components. It refers to a specific remote bind file that determines the connection between clients and servers.
- *The user table* finally relates DCAT files to users and server component instances. In the example, a user might have access to two different shared to-do lists that are accessed via two instances of the same client CAT file. The user table then contains two entries for the same client CAT file, while connecting the two instances to different *shared list* component instances on the server.

These five types of files permit the specification of the structure of the groupware system. The server and client CAT files finally refer to the implementation of the components as JAVA binaries (.class files). The relationship of these five types of files among each other and with the JAVA binaries is given in figure 8.2:

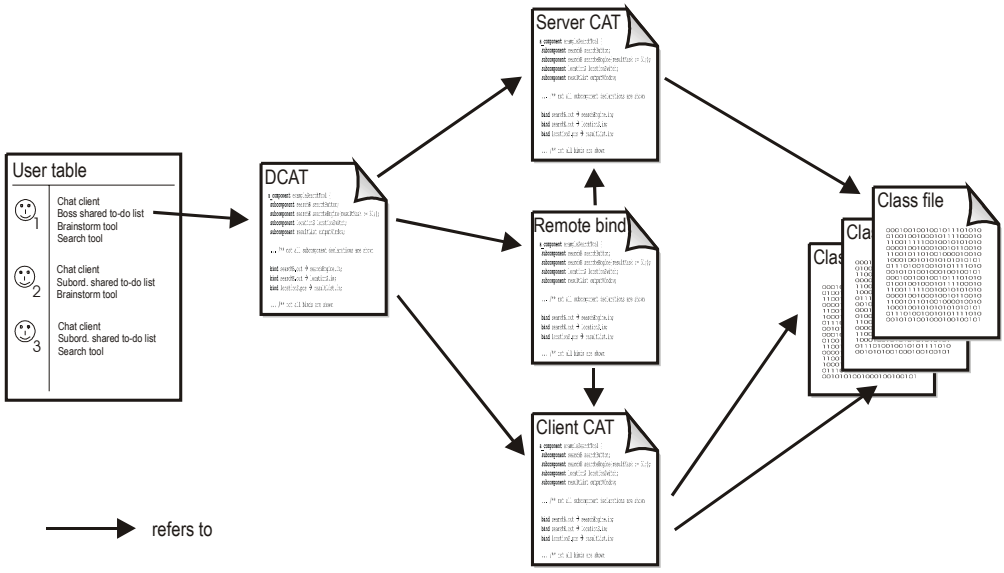


Figure 8.2: The relationship between the different parts of the specification

This multi-part representation scheme was chosen in order to permit the decentralized (i.e. parallel) evaluation and instantiation of a system in a network infrastructure.

Based on the simple shared to-do list application depicted in figure 7.8, the following gives the reader more details on the five types of files employed for the structural representation. A formal definition of the general syntax of CAT files is given in Appendix A.

8.3.1 Client CATs

A client CAT file describes the structure of the client part of a distributed system. It refers to a number of component binaries (`i_component`), defines at least one complex component

(a_component) and can have an arbitrary depth of nesting. The following code example describes the subordinate shared to-do list client taken from the example in figure 7.8:

```
i_component Lists.Visualizer {
 required ToDoList RemoteList sharedObject;
 required ListChanged RemoteActionListener event;
 provided MarkedEntry RemoteEntry sharedObject;
 config_parameter X-Pos int;
 config_parameter Y-Pos int;
 config_parameter Width int := 200;
 config_parameter Height int := 400;
};

i_component Lists.Doner {
 required ToDoList RemoteList sharedObject;
 required MarkedEntry RemoteEntry sharedObject;

 config_parameter X-Pos int;
 config_parameter Y-Pos int;
 config_parameter Width int := 80;
 config_parameter Height int := 30;
};

a_component ListClient {
 required ListChanged RemoteActionListener event;
 required ToDoList RemoteList sharedObject;

 subcomponent doner1 Lists.Doner {
     X-Pos := 80;
     Y-Pos := 450;
 };

 subcomponent visual1 Lists.Visualizer{
     X-Pos := 25;
     Y-Pos := 25;
 };

 bind visual1.MarkedEntry - doner1.MarkedEntry;
 bind ListChanged - visual1.ListChanged;
 bind ToDoList - visual1.ToDoList;
 bind ToDoList - doner1.ToDoList;
};
```

The complex component ListClient defines two ports, instantiates and parameterizes two subcomponents and binds the subcomponents to each other and to the two ports defined for the parent component itself. Note that the parameterization of subcomponents can either be generic (the height and width of the two visible components resides in the unique i_component definition) or instance specific (the x and y-position resides in the subcomponent specification which could be different for every instance).

The name of the implemented (atomic) components always refers to the JAVA package and the class name of the binary (e.g. Lists.Visualizer). In case of multiple levels of nesting, this a_component is interpreted as the root component that has the same name as the CAT file (here: ListClient).

8.3.2 Server CATs

Server CAT files essentially follow the same specification as client CAT files, with the sole exception that they do not refer to visible components. The following code example shows the code for the (single) *shared list* component in figure 7.8:

```
i_component Lists.SharedList {
 provided ToDoList    RemoteList sharedObject;
 provided ListChanged RemoteActionListener event;
};

a_component ListServer {
 provided ToDoList    RemoteList sharedObject;
 provided ListChanged RemoteActionListener event;

 subcomponent list1 Lists.SharedList;

 bind ToDoList    - list1.ToDoList;
 bind ListChanged - list1.ListChanged;
};
```

Even though there is only a single component, for reasons of consistency it is represented as a complex component with one subcomponent.

8.3.3 Remote Binds

Remote bind files specify the way two CAT files (usually a server and a client CAT file) can be connected. The necessity for an explicit specification stems from the fact that a CAT file can define several ports of the same type. This enables differently "wired" connections that lead to different semantics. Or it could be necessary to leave certain ports unconnected. The following code example describes how the shared to-do list server CAT and the subordinate client CAT are supposed to be connected.

```
rbind ToDoList    - ToDoList ;
rbind ListChanged - ListChanged ;
```

The `rbind` command specifies the connection between two ports by name.

8.3.4 DCATs

In the shared to-do list example the DCAT file is used to instantiate and connect the client and the server component structures, employing the remote bind file `ListClientRemoteBind` to specify the semantics of the high level `bind` command:

```
s_component ListClientSimple {
 subcomponent simple_ToDo_client ListClient LOCAL ;
 subcomponent simple_ToDo_server ListServer SERVER;

 bind simple_ToDo_client simple_ToDo_server ListClientRemoteBind;
};
```

The `LOCAL` and `SERVER` keywords define the distribution of the different CAT files. The form of DCAT file shown in the code example above usually suffices to describe the structure of simple client-server architectures. However, the DCAT concept is flexible enough to describe other application structures. Figure 8.3 shows some commonly used possibilities:

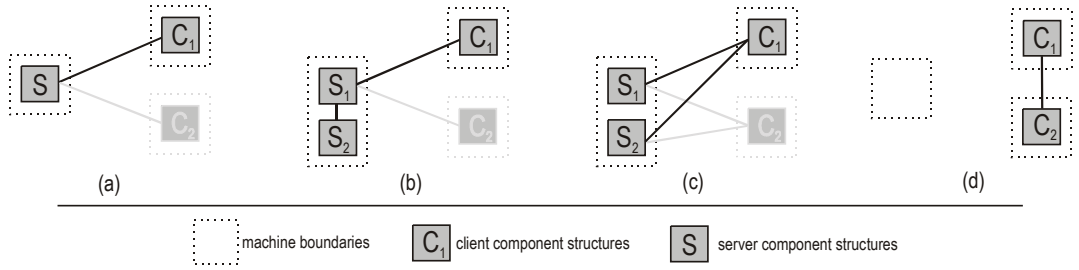


Figure 8.3: Possible DCAT application structures

Case (a) depicts the simple client-server architecture. A DCAT then describes how to connect a client to a server. The same DCAT file can be reused to connect many clients to the server (e.g. additionally the gray client C_2). Case (b) also involves the connection of two formerly independent server components. This structure can be used, for instance, to connect the shared to-do list to a persistency mechanism. Case (c) connects the client to two different server component instances. In case (d), component instance C_1 is connected to another client component which already has to be up and running, when client C_1 is instantiated. This structure obviously does not apply, if the EVOLVE clients are running in a secure applet environment, which can only directly communicate with its host server (see Lindholm and Yellin 1996).

8.3.5 User Management

The user table specifies which client parts of the distributed groupware system are accessible to the user. The table simply relates login names to DCAT file names, together with the name under which the application is supposed to appear in the user's client application selection menu. Whenever a user logs into the system, the list of all DCAT files accessible for that user is sent to the EVOLVE client which presents this list in form of menu. The user can then choose which part(s) of the groupware system he wants to use:

User	DCAT file	Menu name
Ralph	BossListClient.dcat	MyToDoList
Oliver	SimpleListClient.dcat	ToDo

The user management table is the final element of the representation scheme employed in the EVOLVE platform.

8.4 The Object-Oriented Architecture

The last section gave an account of the representation scheme for whole groupware systems in order to support persistent storage in the EVOLVE server's file system. This section presents the object-oriented structures that realize the connection between the representation of the system and its instantiated form. The patterns given here are not JAVA specific and should be reusable in any object-oriented programming language. They are depicted as UML 1.0 diagrams (see Fowler and Scott 1997). Throughout this section, the implementation concepts are related to the elements of the formal model developed in chapter 6.

8.4.1 Component Structures

In order to directly apply tailoring operations to the representation of the component structures as defined by single CAT files, these structures have to be present in the EVOLVE server's volatile main memory (the subsection "Persistency" in the next section describes how these object-oriented structures and their file-based versions are kept consistent). The class diagram depicted in figure 8.4 shows the basic structure for representing hierarchically structured compositions as object networks.

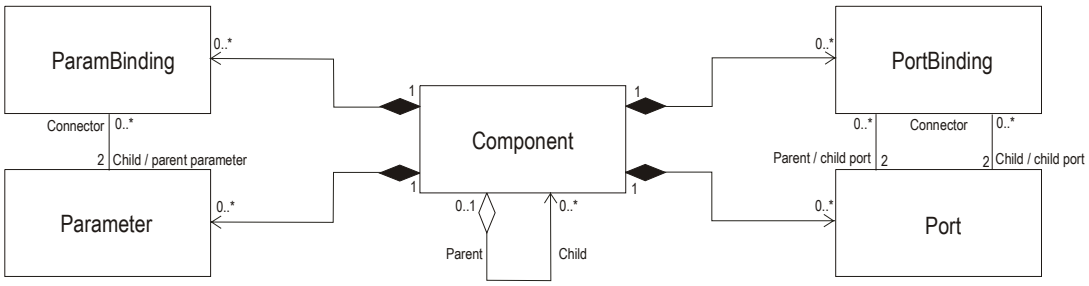


Figure 8.4: UML class diagram for representing hierarchically structured compositions

Note how this pattern closely resembles the formal definition of a complex component given in chapter 6 (definition 6.3). The tuple (n, P, A, S, B_1, B_2, B_3) gives the elements of the diagram. `Component` n has two (possible empty) sets P and A of ports and parameters. The set S of subcomponents is represented by the parent-child relation between components. Finally there is one set B_3 of parameter bindings between a component and its subcomponents and two sets B_1 and B_2 of port bindings between subcomponents and between subcomponents and their parent component.

The structures defined in such a way represent only local parts of the otherwise distributed groupware system. Consequently, one needs further structures to represent the more high-level distribution and user oriented structures kept in the DCAT, remote bind and user table files. Figure 8.5 depicts the UML class diagram for representing these elements:

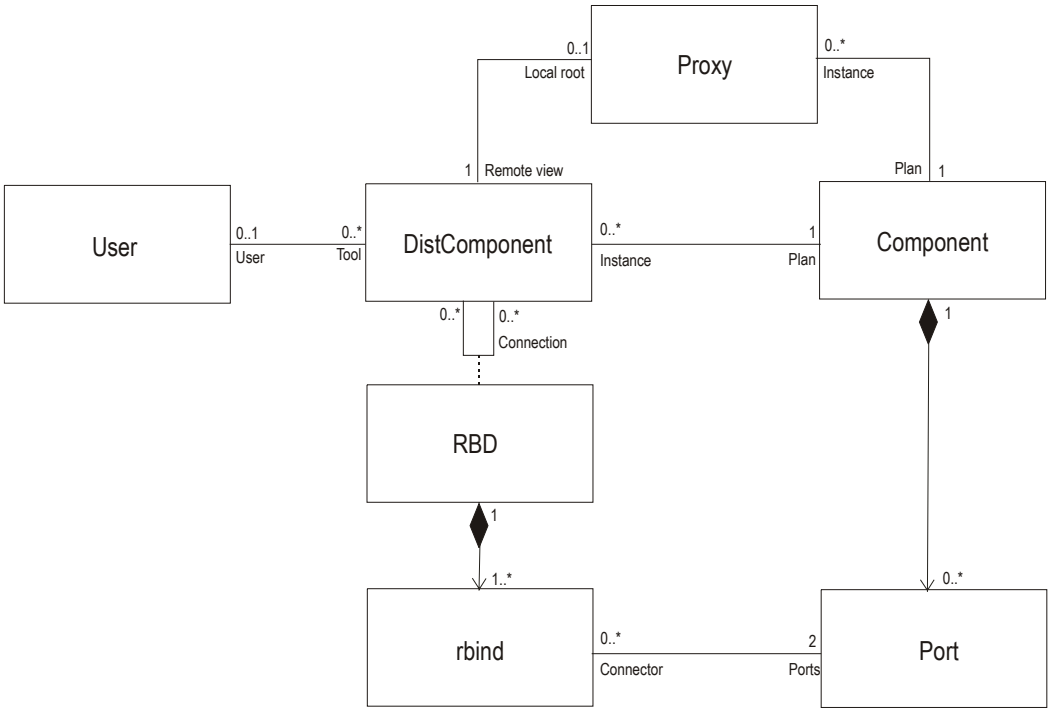


Figure 8.5: UML class diagram for the representation of the distributed system structure

The class `DistComponent` represents a "remote view" on local component structures as defined by a `Component` object. A `DistComponent` either belongs to a specific `User`, or it is regarded as a server component. In the first case, if the respective user is logged in, the `DistComponent` refers to the local `Proxy` structure.

A `DistComponent` can be connected to other `DistComponents`. This connection is described by a `RBD` object that consists of a number `rbinds` (remote binds). The rbinds specify the connection between two `Component Ports`.

8.4.2 Proxy Structures

After instantiation (compare definition 6.6) of the component structures, the resulting instance structures are controlled by proxy structures. Each component structure can be instantiated several times, resulting in an equivalent proxy structure for each instance structure. Figure 8.6 shows a UML class diagram representing the basic proxy structures:

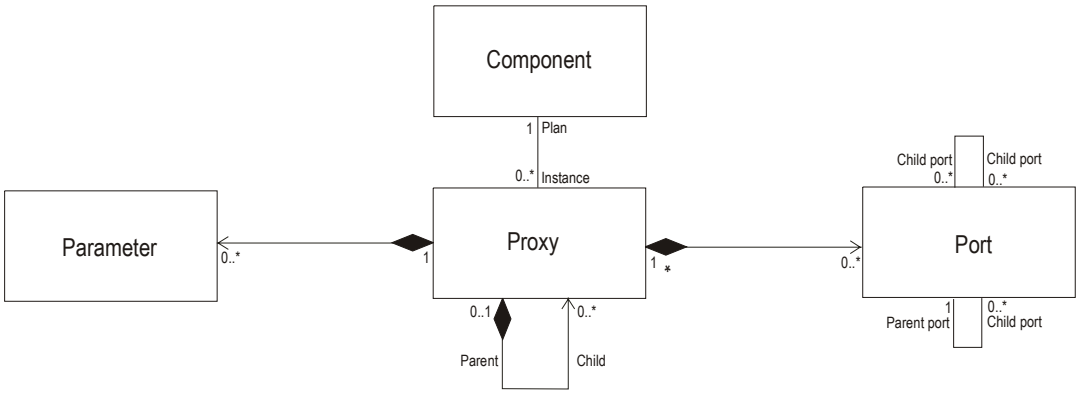


Figure 8.6: UML class diagram for proxy structures

Again this pattern closely resembles the formal definition of an instance system of chapter 6 (definition 6.5). The instance system tuple (I,B_1,B_2,A) gives the elements of the diagram. Each component instance in I is controlled by a proxy. Its ports can be connected to siblings (set B_1) or to parent component instances (set B_2). The latter "vertical" binding information is maintained explicitly to support the computation of the effect of a "horizontal" bind operation on a higher level in the component hierarchy. Finally, set A contains the set of component parameters and its current values.

8.4.3 The System Start-Up Phase

During system start-up the client and server CAT files described in the last section are evaluated and the respective object structures are built up in the main memory of the EVOLVE server (see figure 8.1). The user table is evaluated as well and all server components that it refers to are instantiated, together with their corresponding `Proxy` structures (see figure 8.6). For each local client or server root component instance a `DistComponent` is instantiated (see figure 8.5). Now the EVOLVE system is ready for user login.

8.4.4 The User Login Phase

If a user logs into the system, the client first finds the EVOLVE server via the JAVA RMI naming service (the RMI registry) and connects itself to it. From the server it receives references to all `DistComponent` structures on the server that belong to the current user. The corresponding `Proxy` structures are built up in the client's main memory according to the `Component` structures on the server. Finally, the Proxy structures instantiate the components and connect them – if visible – to the client's GUI. Coming back to the simple example used throughout this section, figure 8.6 depicts the shared to-do list client interface as perceived by the user:

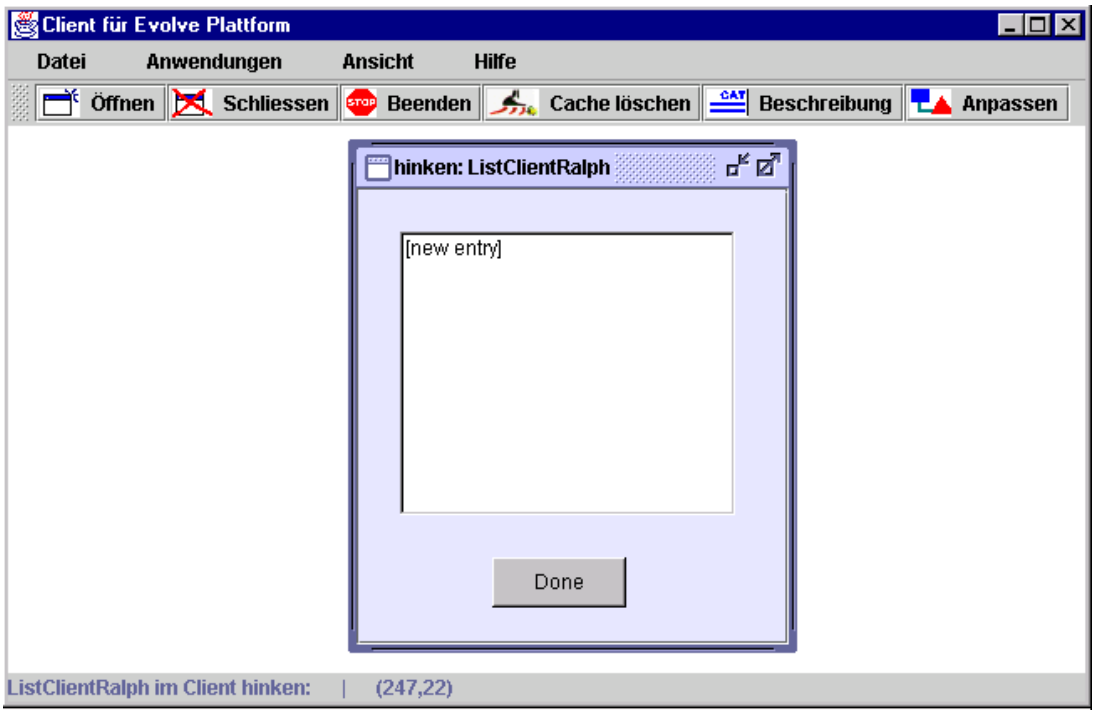


Figure 8.6: Simple shared to-do list client interface after login

Each `DistComponent` on the client is presented to the user in a window within the window of the whole EVOLVE client. Thus the user can switch easily between different EVOLVE applications. Figure 8.7 depicts a more complex client of the shared to-do list example:

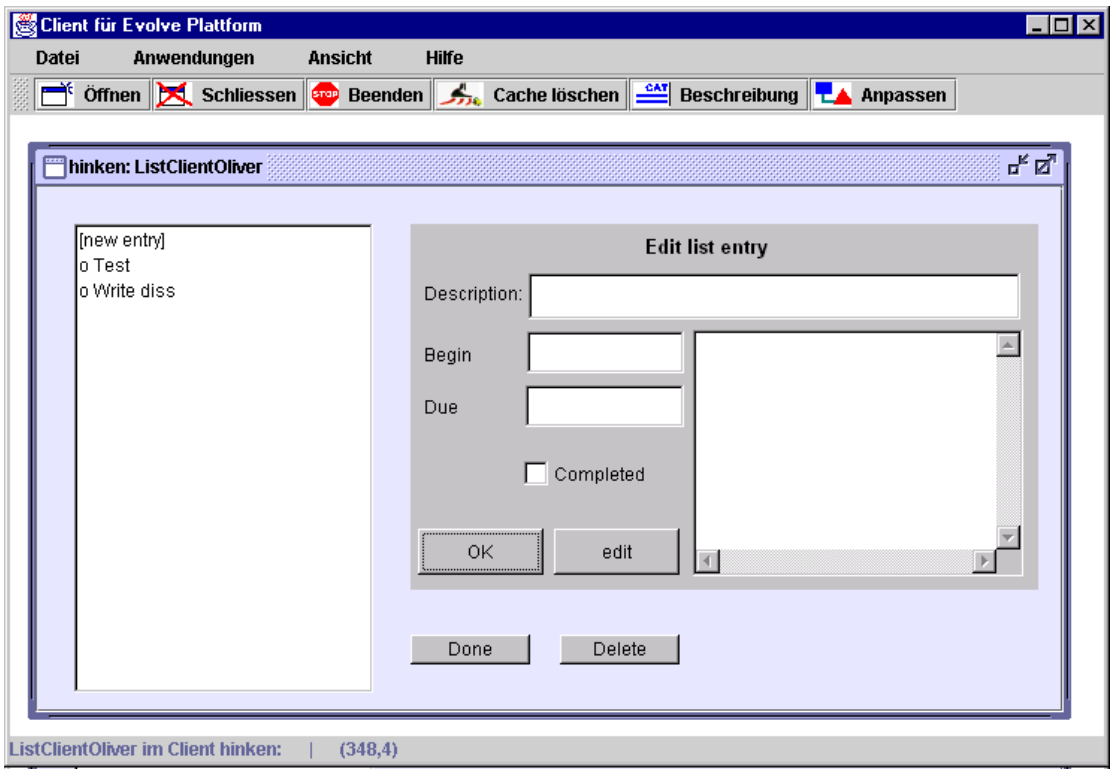


Figure 8.7: Complex shared to-do list client

The client in figure 8.7 also includes a complex *editor* component for adding entries to the list and changing them. Furthermore, it additionally contains a *delete button* component. The client it figure 8.6 might be used by a subordinate, while the client in figure 8.7 could be owned by a supervisor in a hierarchically organized group. Both could be connected to the

same shared to-do list on the server (even though the two clients in figure 8.6 and figure 8.7 are not, as can be seen by comparing the content of the two lists).

Now the groupware application is up and running. If one desires to tailor it, one would have to change the CAT files in the server's file system and restart the whole EVOLVE platform. The *Tailoring API* presented in the next section provides a more elegant and powerful way of tailoring the system during runtime.

8.5 The Tailoring API

The purpose of the *Tailoring API* (Application Programming Interface) is to provide access to the tailoring functionality of the EVOLVE platform. Arbitrary tailoring mechanisms – e.g. user interfaces, intelligent agents or even the component-based groupware application itself – can inspect and change the full compositional structures of EVOLVE applications, including the component instances' parameterization. The Tailoring API consists of a set of JAVA interfaces that are implemented by the object structures described in the last section.

An important aspect of the Tailoring API is the fact that it can be accessed remotely from every point in the network. All interface are remote capable, based on JAVA RMI technology (see chapter 3). From the RMI naming service (the registry), a tailoring mechanism can request the remote reference to the EVOLVE `server`, which serves as starting point for every tailoring operation. The server provides operations to get references of all distributed (client and server) root components as specified in the DCAT files of the application – either for a single user:

```
Server.getDistComponent(Username, ID, Typename);
```

or for the whole application:

```
Server.getAllDistComps( );
```

Once one is in possession of the (remote) reference to a `DistComponent`, one can, for example, inspect the connections to other components:

```
DistComponent.getLinkedDistComps( );
```

Generally, all elements of the application's structural representation in main memory as specified in the last section are accessible – in particular the server-side object structures for the description of the component structures. Most tailoring operations described in the following are applied directly (but possibly remotely) to these component structures. From these the changes are then distributed to the proxy objects on the clients (and also to the server proxy objects). The following subsection gives an overview of how the tailoring operations specified formally in chapter 6 are implemented. The second subsection describes the way, changes are propagated to the proxy structures and finally to the component instances. Subsections three and four finally focus on the operations for scope control and persistency. The complete documentation of the EVOLVE Tailoring API can be found in (Hinken and Stiemerling 1999).

8.5.1 Tailoring Operations

Chapter 6 gave formal definitions for the necessary set of tailoring operations. Most of these operations are applied in the context of a complex component $(n,P,A,S,B_1,B_2,B_3) \in H$. In the object-oriented implementation these operations are consequently provided as methods of the `component` class. The following table gives an overview of the methods implementing the operations (the parameters of the methods are simplified here for better presentation):

Tailoring operations	**Methods** (all on class `component`)
Adding and removing subcomponents	**`addSubComponent`**`(type, name);` **`removeSubComponent`**`(name);`
Binding and unbinding ports of subcomponent instances	**`addPortBinding`**`(portname1, subcomponentname1, portname2, subcomponentname2);` **`deletePortBinding`**`(portname1, subcomponentname1, portname2, subcomponentname2);`
Binding and unbinding ports of subcomponent instances and the parent component	**`addPortBinding`**`(portname1, NULL, portname2, subcomponentname2);` **`deletePortBinding`**`(portname1, NULL, portname2, subcomponentname2);`
Binding and unbinding parameters	**`addParamBinding`**`(parametername1, subcomponentname, parametername2);` **`deleteParamBinding`**`(parametername1, subcomponentname, parametername2);`
Changing the parameterization of subcomponent instances	**`setParam`**`(subcomponentname, parametername, value);`
Adding and removing ports	**`addPort`**`(name, type, polarity, protocol);` **`deletePort`**`(name);`
Adding and removing parameters	**`addParam`**`(name, type);` **`deleteParam`**`(name);`

Table 8.1: Tailoring operations implemented as methods of class `component`

A completely new root component (resulting in a new CAT file on the EVOLVE server) can be created by:

```
Server.getNewDistComponent(Username, ID, Typename);
```

The new DistComponent can be populated with new components using the methods:

```
Server.getNewComponent(typename);
```

Removal of components works analogously. While the operations given in this subsection are complete in the sense of chapter 6, the EVOLVE Tailoring API contains many more methods for facilitating the usage of the interface. For instance, one can directly rename ports, parameters and subcomponents without having to go the hard route of deleting the port and creating a new one with the new name. Other methods concern the inspection of component structures: queries for root components, conversion of interaction types to strings (for easy display on a user interface) etc.

8.5.2 The Tailoring Phase

The methods implementing the tailoring operations specified in chapter 6 are applied directly to the objects specifying the component structures. However, one component structure can be instantiated several times, on several different physical machines. Therefore, tailoring operations applied to the components have to be propagated to all their instances (compare figure 6.2 in chapter 6). The interaction diagram in figure 8.8 depicts the interaction patterns when applying a tailoring operation (here: the setting of a parameter) to a component that has two instances.

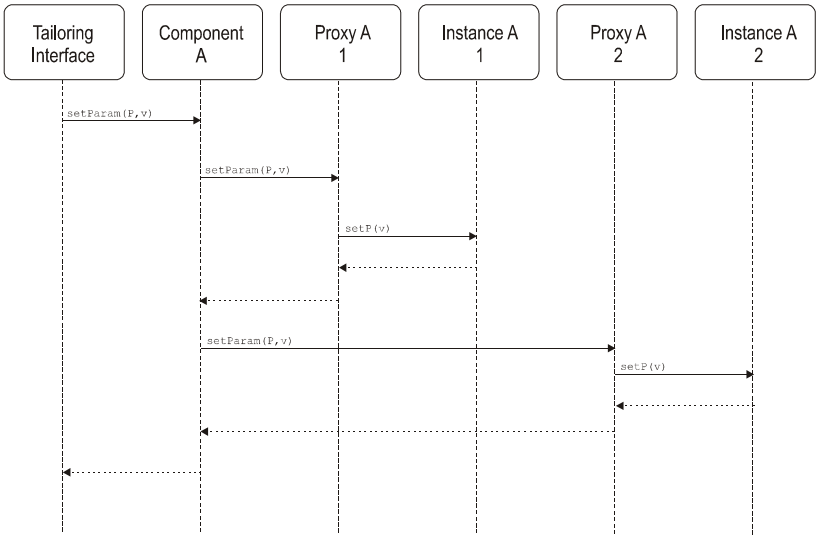


Figure 8.8: Interaction diagram showing how changes (here: changing a parameter value) are propagated from the component object structure via the proxies to multiple instances

The component in figure 8.8 is obviously atomic. In the case of a change to a complex component, the proxies compute the corresponding changes to the instances on a lower level by simply propagating the changes down the proxy hierarchy according to the specification given in chapter 6 (*corresponding changes*). A description of the algorithmic aspects can be found in (Hinken 1999).

In the current implementation of the EVOLVE platform, the changes are propagated to multiple proxies sequentially. Obviously this induces a certain network-caused delay. Future optimizations can be achieved through parallelisation of propagation actions.

8.5.3 Scope Control

Controlling the scope of a tailoring operation (e.g. limiting the effect of the parameter setting in figure 8.8 to instance 2 of component A) is an important requirement on the implementation of the EVOLVE platform. Since the platform only permits sharing of specifications on the root level (e.g. whole CAT files), the scope control operation only has to regard sharing as defined by the user table (which users share a client CAT file?) and the DCAT files (which server component instances share a server CAT file?).

The scope control operation is implemented in the `DistComponent` class and has to be executed before applying any tailoring operations to a complex component. It works like defined in chapter 6, i.e. it copies the complex component's definition and re-directs the proxy references of these instances to be affected by the tailoring activities to this new component:

```
DistComponent.changeReferences(NewTypename, Scope);
```

The `NewTypename` specifies the name of the copy of the old complex component and the `Scope` is the list of users and/or server component instance names that are supposed to be affected by the tailoring activity. After executing the `changeReference` method call, the tailoring methods can be applied to the new complex component and will only take effect in the instances specified in `Scope`.

8.5.4 Persistency

So far, tailoring operations have only been performed and have taken effect in volatile main memory. In order to make these changes to the compositional structure of a groupware application persistent – i.e. recoverable after system re-start – the platform provides

operations for saving the whole structure (or only parts of it) from volatile memory onto the file system of the EVOLVE server. This process is essentially the reverse process of the system start-up described earlier. It should be executed whenever a series (or a transaction) of tailoring operations has been concluded and the system is in a desirable state. If the whole system crashes before the persistency operation has been performed, the last consistent state is recovered.

Since tailoring operations are always applied in the context of a complex component or result in the creation of one, the persistency operation can be applied on the level of complex components:

```
Server.saveComponent(Component, Directory);
```

So far, the EVOLVE platform does not provide any security mechanisms for concurrent tailoring transactions, e.g. by many different interfaces accessing the Tailoring API. In this case, perhaps some kind of locking mechanism might be useful to maintain consistency. Then, only one user can change a specific complex component at a time.

8.6 The 3D Tailoring User Interface

The focus of this dissertation is placed on software technical questions raised by component-based tailorability. However, in order to demonstrate the viability of the basic architecture of the EVOLVE platform (as depicted in figure 8.1), this section describes a 3D user interface for tailoring. It can be "plugged into" the Tailoring API described in the last section. The interface was implemented by (Hallenberger 2000).

The initial experiences with the 2D tailoring interface in the POLITeam project support and reflect findings reported in the literature concerning the viability of visual programming. A number of visual programming approaches faced severe problems concerning the limited expanse of screen space when representing a program (Nardi 1993). In almost every non-trivial case it appears to be impossible to display all relevant information in a certain phase of programming. In particular, the search tool case study (chapter 4) demonstrated that the presentation of these aspects of a software system visible to end users (e.g. the arrangement of buttons and other interface widgets on the windows) and the logical structure of a program is hard to jointly depict in a visual programming interface. If one chooses a joint representation (as in the search tool example), invisible components (only relevant for the logical structure of the software) tend to leave blank spots in the regular user interface. However, separating the visible and the logical structure of a software system in the tailoring interface's representation makes it hard – especially for end users – to comprehend the connection between the usual appearance "on-stage" and the "back-stage" arrangements. The objective should be to make the move from the known to the unknown as smooth as possible. In addition to the transition from the visible to the invisible, this objective also concerns the move from a coarse understanding of program structures to the "finer details" of the system, and – in the case of distributed applications – from the local components to these on the server or on other clients (compare the shared to-do list example described in chapter 7).

These problems quite naturally point to 3D as a possible solution. In particular the problem with the blank spots in the regular user interface caused by jointly presenting visual and logical structures in the tailoring interface can be tackled with the help of an additional dimension. During tailoring, invisible components can be placed "behind" the plane of visible GUI components. The following subsections demonstrate how the problems described above can be approached with the help of three dimensions. While the main focus of this section lies on representation, the last three subsections also describe the way users can navigate through the representation and how they can interactively manipulate the represented groupware application. Figure 8.9 gives the reader an overall impression of the look of the prototype user interface.

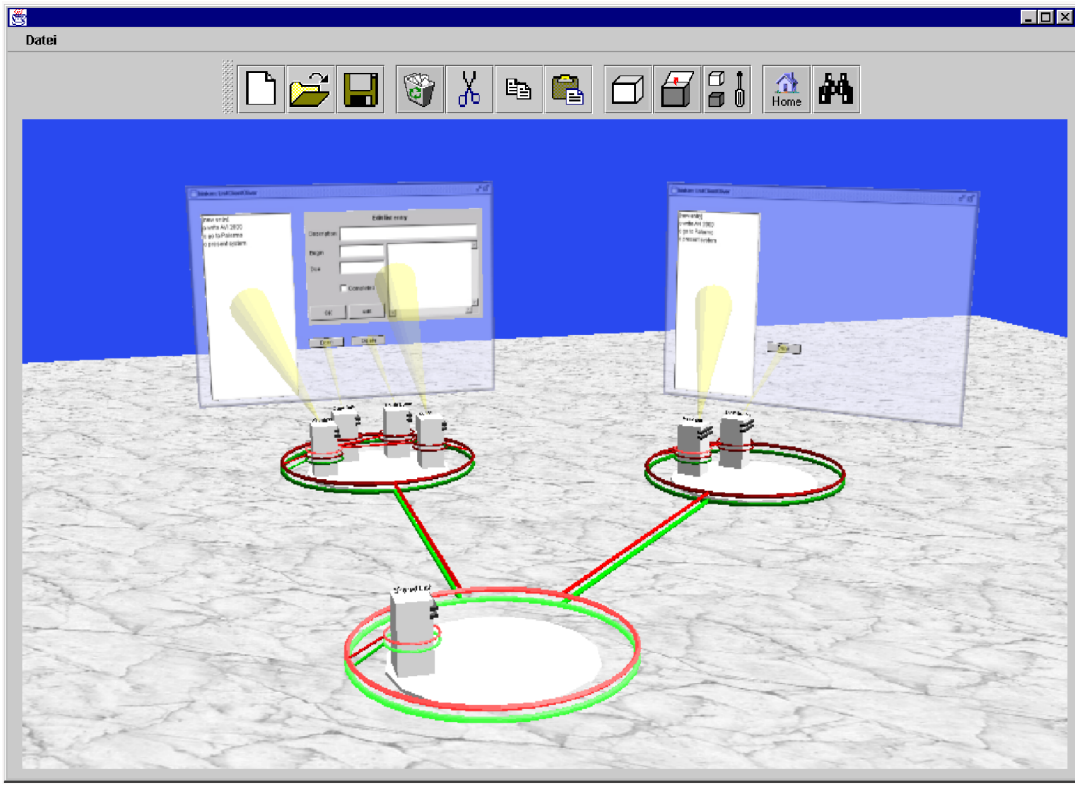


Figure 8.9: 3D interface showing a high level view on the example groupware application of figure 1.3 (the *shared to-do list* with a supervisor and a subordinate client)

The interface can be started as a stand-alone application on every computer in the network. It is implemented using the JAVA 3D API (JavaSoft 1998). The interface consists of a tool-bar offering component-based operations like aggregating (grouping) and parameterizing components. The representation consists of a server in the foreground and two clients (with visible components) in the background. The application represented in figure 8.9 is the shared to-do list application described in chapter 7. The 3D tailoring interface can be used while other users are working with the application.

8.6.1 Component Representation

The compositionally relevant aspects of a component are obviously its ports, their types, polarities, and interaction protocols. Figure 8.10 shows the 3D representation of the *visualizer* and the *done button* component:

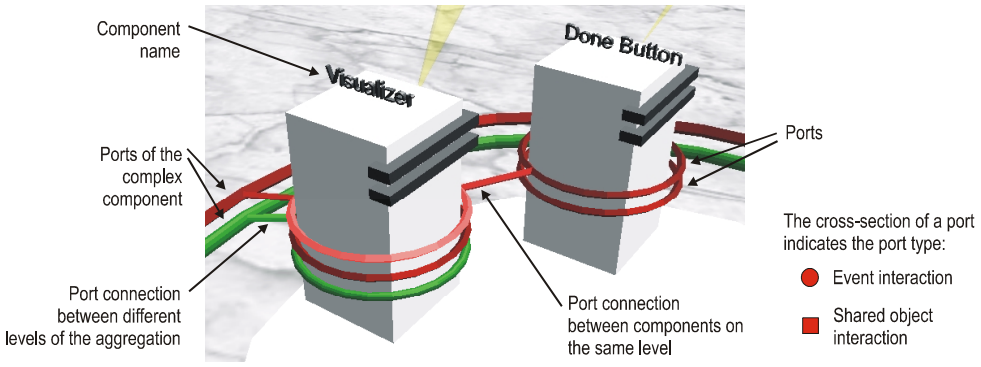


Figure 8.10: Component representation (the long black boxes resemble the UML notation for component representation)

While the component itself is represented as a simple box, its ports are represented as rings in order to enable connections from all directions on the plane (The interface is supposed to present the user with the notion of a "floor". Consequently, most connections between ports are likely to be horizontally.

The cross-section of the ring determines the interaction protocol: a circle indicates event-based interaction, while a square depicts shared object interaction. The color of the ring indicates the (data) type of the interaction. The user does not need to know the exact nature of the type because the color is enough to establish its connectability.

The polarity of the port (i.e. whether it is event source or sink, resp. shared object provider or user) is determined by the intensity of the color: lighter shading indicates a provided port, while darker shading stands for a required port.

Component connections are represented by thin "tube"-like objects touching both connected ports.

8.6.2 Visible Components

As demonstrated by the search tool example, the combined representation of the visible and the logical structure of a software system constitutes a major challenge. In the 3D interface, the logical structure is given by the component connection as described in the preceding subsection. The visual appearance of the GUI components is projected onto a semi-transparent plane that hovers in front of the logical composition. The plane implements a "cinema" metaphor. Figure 8.11 depicts this for the case of the supervisor's interface:

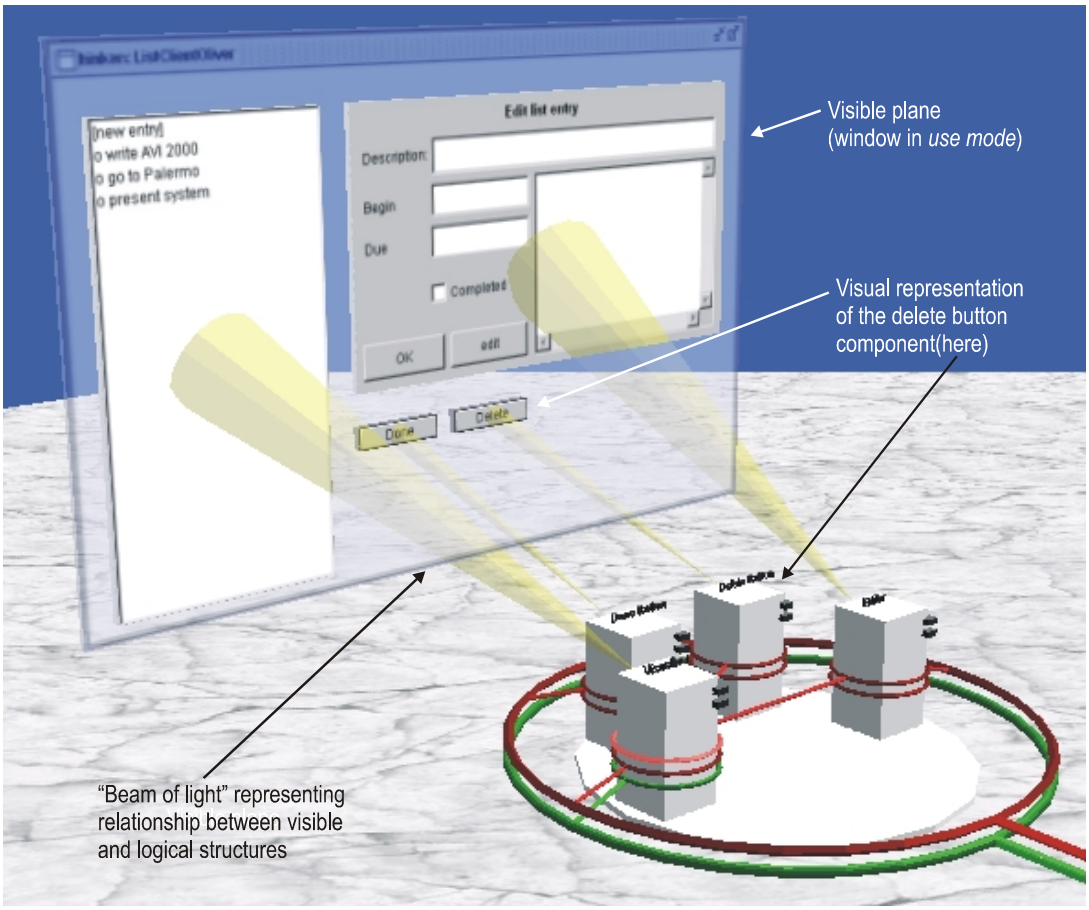


Figure 8.11: Representing visible components with a "cinema", screen-like metaphor

The tailor can move the projections around the visible plane representing the window in normal use mode.

When the user switches from use mode to *tailoring mode*, i.e. if he enters the 3D virtual world, he is positioned directly in front of the visible plane. Thus he is mainly confronted with things he already knows, i.e. the appearance of the regular user interface. However, through the semi-transparency, he can also peek "under the hood" of the system and regard its logical structure. If he wants to change the logical structure, he simply moves through (or around) the semi-transparent plane. By following the "beams of light" indicating the projection, he reaches the component instance representation embedded into the logical structure.

8.6.3 Distribution

Since the EVOLVE platform is designed to support groupware systems, its application model is oriented at client/server architectures. Consequently, in the 3D interface, clients form a half circle around a server in the middle. Clients and servers are indicated by horizontal disks. Note that while the server always resides on a specific physical machine, the clients are synonymous with user logins, i.e. they can be instantiated on varying physical machines, depending on where the end user decides to log into the system (perhaps even in a Web-Browsers). Figure 8.12 shows a high level view on the distributed structure of a shared to-do list configuration with one subordinate and one supervisor client (as in figure 8.9):

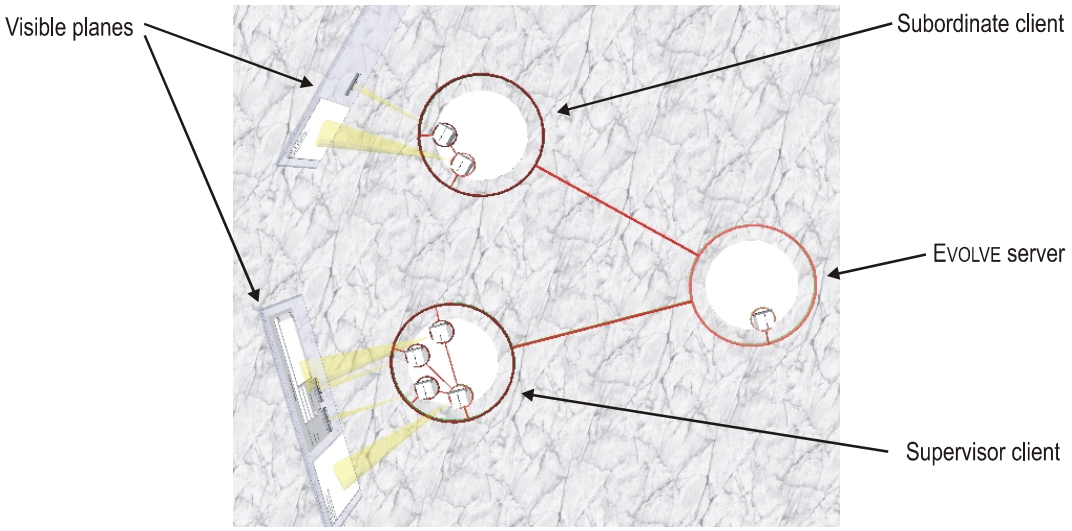


Figure 8.12: Representation of distribution (a bird's eye view on the *shared to-do list application* of figure 8.9)

Each client has a plane for visualizing the appearance and structure of visible client components. Only invisible components can have instances on the server.

8.6.4 Hierarchy

The concept of nested component structures is supposed to support viewing and manipulating of the tailorable application on different levels of abstraction and complexity. This feature permits end users to approach the application from a rather coarse structural understanding and gradually move to lower, finer levels of granularity and understanding. Figure 8.13 depicts how component aggregation hierarchies are represented in the 3D interface:

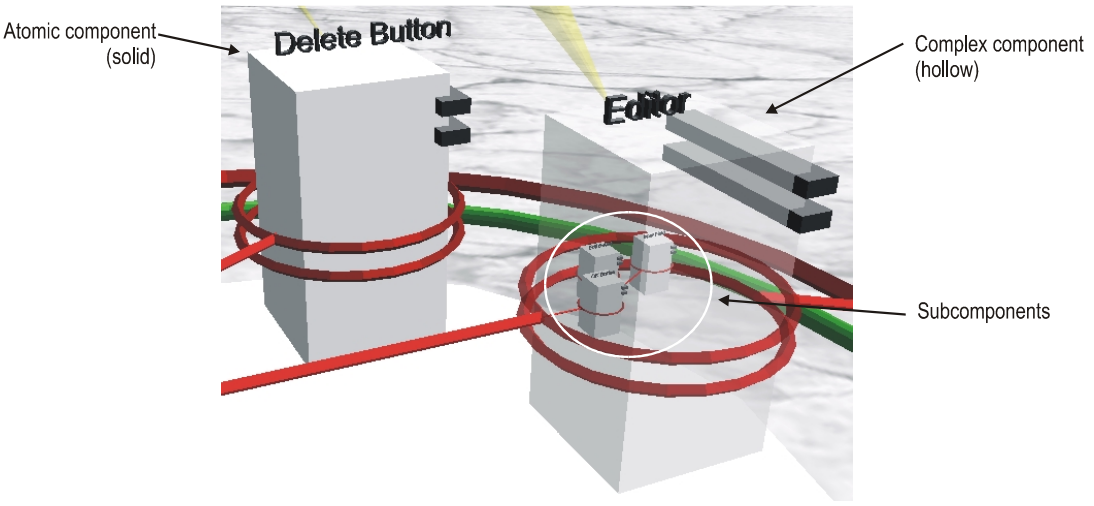


Figure 8.13: Representation of nested component structures (the *Editor* is a complex component and contains three subcomponents, while the *Delete Button* is atomic)

The representation of components is the same on all levels and the user can smoothly move between levels by navigating closer to the encapsulating high-level component. The encapsulation boundary becomes more and more transparent as the user closes in on it. Finally the barrier disappears, while the rings representing the ports of the high level component remain. Thus the user can manipulate the connection between ports of this component with ports of its subcomponent.

8.6.5 Navigation

User controlled navigation is the main prerequisite for the appropriate smooth transition between local and remote components, visual and logical structures and different levels of nesting as described in the preceding subsections.

The 3D interface supports navigation with the help of different input devices. The user can indicate vectors of movements with the mouse. A three-button mouse permits the selection of whether the indicated vector determines the speed of a horizontal, vertical or turning movement. In addition to the mouse, the user can also employ the keyboard for navigation.

8.6.6 Changing Compositional Structures

The interface offers all tailoring operation necessary for transforming every possible composition (with respect to the set of available atomic components) into every other possible composition. The user can select and instantiate atomic components from a list in a menu, connect component instances by double clicking on the two ports and create new levels of component nesting by selecting a number of components on the same level and clicking the aggregation button. Components and connections can be also removed or reconnected again.

In addition to these compositional operations, the user can select a component for parameterization. In this case a window appears which permits changing the different settings of a component. Furthermore, the x and y position parameters can be changed by moving the visual representation of a component's GUI around the semi-transparent plane.

8.6.7 Sharing of Adaptations

It is important to note, that the 3D interface represents component instances. However, the tailoring operations are directly applied (as specified in chapter 6) to a complex component's plan, i.e. its structural description as CAT file. Thus if the tailoring interface is used to adapt

one of multiple instances of a component plan, these changes are propagated during runtime to all other instances of that component plan.

Sometimes this feature of the EVOLVE platform is desired, e.g. when a system administrator wants to tailor the groupware system for many users at the same time. However, at other times, one might want to restrict the scope of the change to a certain subset of users (or perhaps even only one). In this case, one can apply the scope control operation (defined in chapter 6). Thus changes to this copy are only propagated to instances within the scope.

Chapter 9

Applications and Discussion

9.1 Introduction

This chapter demonstrates the applicability of the component-based tailorability approach to groupware design and beyond. Furthermore, it discusses how the theoretical results of chapters 5 and 6 are reflected in practice.

At the time of writing the component-based tailorability approach has been applied in the design of five prototypical groupware systems:

- The **search tool** of the POLITeam groupware system ("pre"-EVOLVE, see chapter 4);
- The **shared to-do list** (employed as technical example in chapter 7);
- **MUD**, a synchronous, text-based communication tool (in this chapter);
- **ADOS X**, a asynchronous system for document exchange (in this chapter);
- **EMS**, an electronic meeting system (currently being implemented, see Bohning 2000).

For the purposes of this chapter, two of these systems – MUD and ADOS X – are presented in sections 9.2 and 9.3. The MUD system offers *synchronous text-based cooperation*. ADOS X is designed to support *asynchronous document exchange* between cooperating companies. In both cases, the focus of the presentation lies on the way in which specific tailorability requirements are reflected in the decomposition of the system's functionality into component sets. In addition to these two systems, subsection 9.4 envisions a (currently not implemented) non-groupware application in the financial sector. The purpose of this last case study – which is based on real world requirements documented in the literature – is to support the claim that the concepts developed here are not necessarily groupware specific.

The discussion in section 9.5 builds on the technical experiences gained during the implementation of the MUD and ADOS X component sets. It focuses on the question how the theoretical results and core concepts of this work are reflected in programming practice. In addition to these qualitative considerations, section 9.5 summarizes the results of a series of performance tests of the EVOLVE system.

9.2 MUD

The name MUD stands for MULTI USER DUNGEON. Despite this name, the system is not a game, but denotes a component set for constructing and evolving text-based, virtual meeting rooms. Users can log into these rooms, see who else is present at the moment and communicate based on text messages. (Churchill and Bly 1999) describe the use of a similar application to support remote cooperation in a research center. The primary attractiveness of a MUD lies in the fact that it is a communication tool somewhere "between" email and the telephone ("*As I see it, it fits between where you have the telephone and electronic mail*." Churchill and Bly 1999, p. 102). It does not require constant attention like the telephone, but is still more synchronous than email, because messages are shown instantly to all collaborators in a room. (Churchill and Bly 1999) also report on the use of the MUD application to keep in contact with the research center when working at home or abroad.

The implementation of the MUD as a set of FLEXIBEANS is designed to demonstrate how a groupware application can be decomposed into FLEXIBEANS to benefit from the tailorability provided by the EVOLVE platform. In particular, it is shown how diverse requirements stemming from different uses of the tool can be met by differently tailored versions.

9.2.1 The MUD Component Set

The initial set of FLEXIBEANS comprises of eight components that are depicted in table 9.1. The table also summarizes their functionality. The following describes the motivation for the implemented functionality and its chosen decomposition.

The core functionality of a MUD application is given by the first three components in table 9.1: the *MUD server*, the *main text window* and the *text input* component. The need for the decomposition in server and client components is obvious. The separation in *text window* and *input* component is motivated by the consideration that there might be users, who (1) only need to "listen" to conversations in the chat room or (2) want to change the relative position of the input and output (text window) components on the screen.

Additionally, the set contains two components that show who is online and permit users the choice of a self-description (usually the name plus a short message like, for instance, "*back in five minutes*"). The self-description is also used to personalize the messages inputted in the text input component in the form "`name > ...text`". The decomposition in two components is justified by the fact that some people might only be interested in seeing who is online.

These five components suffice to compose a rather basic MUD. However, (Churchill and Bly 1999) describe usage situations, in which additional functionality appears to be useful. For example, if the system is used on a workstation, the MUD application is likely to run in the background, while the user is concerned with other tasks and other applications. In order to support these situations, the FLEXIBEANS component set offers two *warner* components that can be configured to react to certain key phrases in the conversation. If one wants to be alerted, for instance, if somebody talks about lunch, one can add a component to the application, which emits a ringing sound, if the word "*lunch*" is mentioned in the MUD. Alternatively, the other warner component displays a little window on top of all other applications with the text "*somebody said 'lunch'!*"

Finally, it has to be assumed that not all users enter a conversation at the same time. Therefore, the FLEXIBEANS set offers a component that permits a latecomer, the repetition of the last x sentences in the MUD. The next section demonstrates how these components can be composed to yield MUD applications for different requirements.

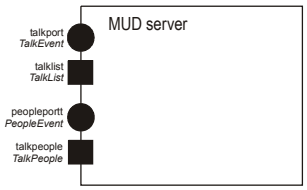
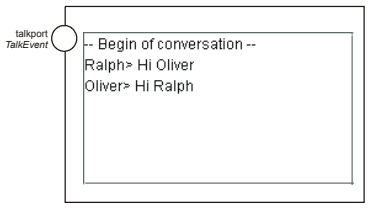
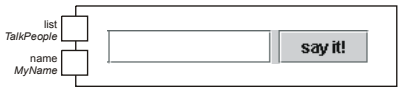
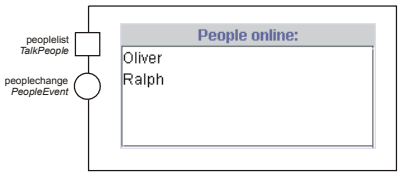
	**MUD server component** This server component is responsible for storing the list of active participants and for distributing the text messages instantly to each party. It also sends out events informing clients whenever the current list of participants has changed. Furthermore, it permits the downloading of specific sequences of the conservation history in order to help latecomers to enter the discussion.
	**Main text window** This component can receive and display the text messages distributed by the server component. Whenever the contents exceed the visible area, a scrollbar appears which permits moving the window to the part of the conversation that is currently of interest. However, new text is always added at the end.
	**Text input** This component permits the user to input and send a text string to the server component that distributes it to all other participants. If a name component (see below) is attached to it, then the name of the user is added to text string, e.g. "Joe> ....".
	**User Name** The user can type his or her name that is entered into the list of active participants maintained by the server component. The user can change the name during a session (e.g. in order to add things like "back in 5 minutes" or "out for lunch"). The change becomes instantly visible in the people online component (see below)
	**People online** This component visualizes the list of active participants maintained by the server component. Whenever a change in this list occurs, an event is received and the visualization is updated accordingly.
	**Repeater component** This component is intended to support latecomers to the MUD room. Pressing the button repeats the last x lines of text stored by the server component. The repeater component connects both to the server component and the main text window.
	**The sound warner** This component can be parameterized with a string. Whenever this string appears in a conversation (it can also be the name of another participant), the EVOLVE client emits a ringing sound. This component is intended for users who cannot constantly monitor the conversation in the room, but need to be aware whenever certain key words (like "lunch") are uttered.
	**The window warner** This component provides essentially the same functionality as the sound warner. However, instead of the ringing sound, a small window appears on top of all other applications running concurrently.

Table 9.1: The components of the MUD component set

9.2.2 Different MUD Compositions

The components shown in table 9.1 can be composed yielding different MUD applications. Figure 9.1 depicts four example compositions for different users and purposes:

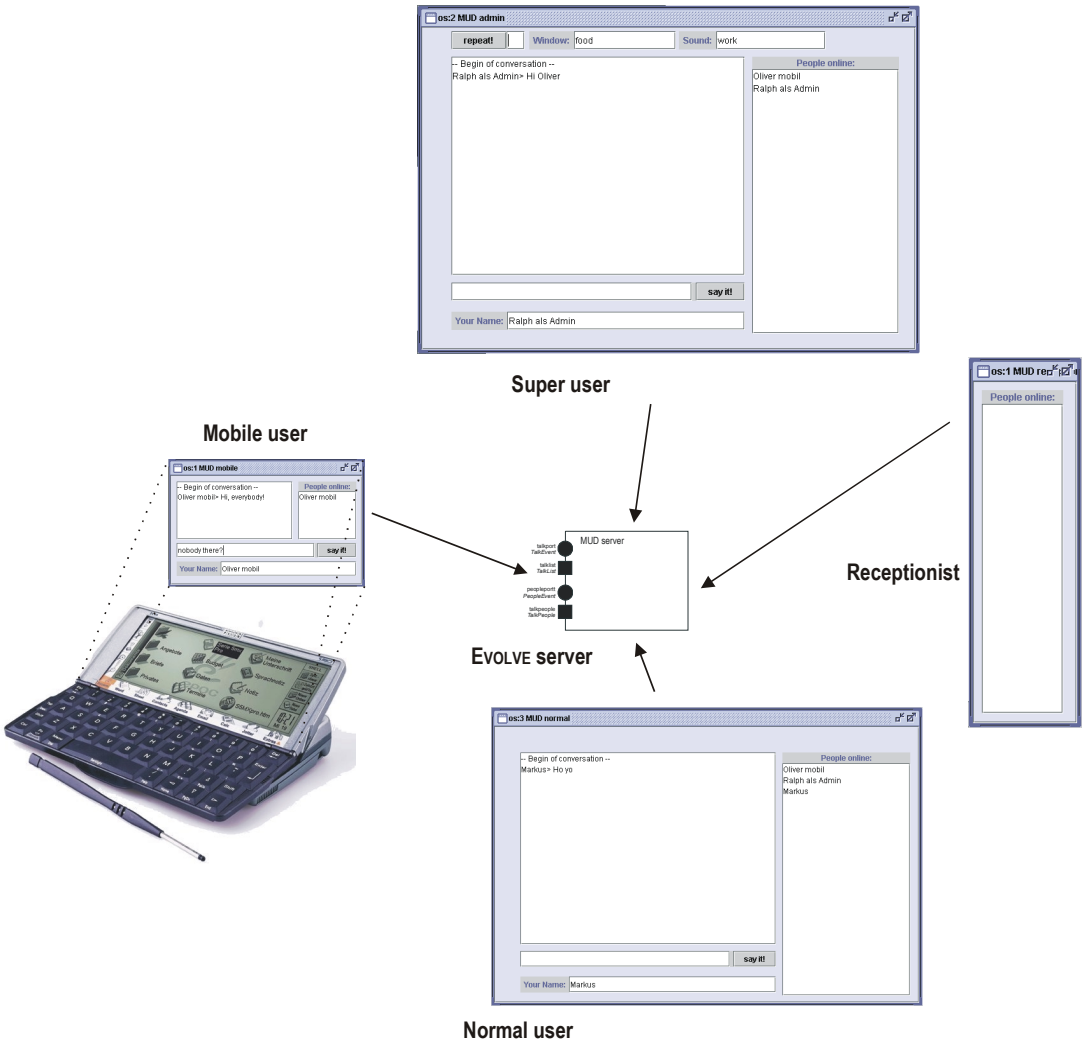


Figure 9.1: Different MUD compositions

The *super user* application (upper part of figure 9.1) contains all possible functionality. It is designed for use on a workstation running several other applications concurrently. The user can be alerted, if certain key phases are mentioned in the MUD. The composition for the *normal user* (lower part) is simpler, because it only contains the core components for participating in the MUD. The *receptionist* application (right side) contains only the people online component and is used by a receptionist or telephone operator to have an indication of who is currently reachable. Finally, a *mobile user* is supported by an application that contains the same components as for the normal user. However, they are resized (via the x/y-parameters) in order to accommodate the restricted display space provided by a hand-held computer (here a PSION SERIES 5). Alternatively, the mobile user could place the application in two different windows (according to the design of the EVOLVE client described in chapter 8), one for the list of people online and the other for the chat functionality (text window and input component) in order to save screen space.

Furthermore, there are a number of subtle ways in which MUD compositions can differ to meet diverse requirements. For instance, the *name* component could be disconnected from the *MUD server* component in order for the user not to appear in the list of the *people online* component. Or a *warner* component could be connected to the repeater instead of the server in order to be able to search in the conversion history for a certain text string.

9.2.3 Further Evolution of the System and Summary

The compositional mechanisms are not restricted to the client of the MUD systems. One could imagine a suite of server components that integrate the MUD system with other telecommunication infrastructures. For instance, it should be possible to add a component to the MUD server that acts as a SMS (Short Messaging System) gateway, transmitting the whole conversation (or certain key phases, similar to the warner components) to a mobile phone. This could be useful in a helpdesk scenario in which a system administrator is often away from his workstation. Requests for help could be sent directly to the mobile phone.

Summarizing, the MUD example demonstrates how the concept of component-based tailorability can be applied to the design of a primarily synchronous groupware system. The initial set of components can be employed to compose (and re-compose) different types of MUD applications. Additionally, the system can be further evolved by adding new components that provide functionality not initially anticipated. This section focused on the way the MUD functionality is decomposed and how the resulting component set can be employed to compose different applications to meet different or dynamic requirements. The discussion in section 9.5 again refers to the MUD application. However, it is concerned with the question how the theoretical results of chapter 5 and 6 are reflected in programming practice – also taking up the ADOS X case study described in the following section.

9.3 ADOS X

The second case study describes the application of component-based tailorability in the ADOS X system of the ORGTECH project. The first subsection gives an overview over the relevant aspects of the ORGTECH project, while the second subsection focuses on the concrete range of tailorability necessary in the context of document management in virtual enterprises. Subsection three presents the developed component set and discusses how differently tailored compositions of these components cover the range of requirements elaborated in subsection two. For a more detailed account of ADOS X, see (Stevens 2000).

9.3.1 The ORGTECH Project

The ORGTECH project is concerned with the development and introduction of groupware systems for cooperating enterprises (see e.g. Stiemerling et al. 1998). In particular, it looks at supporting cooperation in the engineering sector. Three concrete scenarios under investigation are partnerships of small engineering companies to provide joint services, tight integration of small engineering firms as subcontractors of large customers and finally tele-work. While the project also covers synchronous tele-cooperation (e.g. video-conferencing), only the asynchronous aspects of the second, subcontracting scenario of interest here.

In order to fulfill their contracts, the external engineering firms need access to the huge construction drawing archive of the large customer (a steel mill in the Ruhr area, whose machinery is defined by 300.000+ construction drawings). The drawings are steel mill-internally available in an electronic format via the so-called *ADOS* systems. One objective of the ORGTECH project was to make the archive accessible to the external engineers in order to improve their cooperation with the steel mill. The resulting system is called *ADOS X* (the 'X' stands for eXternal access). It is subject to highly dynamic and diversified requirements which are summarized in the following subsection and which constitute the need for tailorability in the ADOS X system.

9.3.2 The Required Range of Tailorability

The requirements for software systems spanning several organizations are naturally subject to more intensive controversy than these of a single organization (see e.g. Rittenbruch et al. 1999). Especially the customer-supplier relationship in the steel mill scenario is a touchy

one. The external engineers work for different departments of the steel mill and thus cooperate with different people. Some engineers are well known in one department and less so in others. Personal relationships are important for receiving contracts. Once a contract has been issued, access to the drawing archive has to be granted. In theory, there is a central construction department that is responsible for the central archive. Consequently, all access should run through this department. In practice, the external engineers can take the short cut via their local "contact" in the other department where the contract originated. This person (or the responsible engineer in the construction department) receives a fax with a request for documentation, accesses ADOS (the steel mill-internal system for accessing and printing construction drawings), searches for the relevant documents and orders paper blueprints or electronic copies. These are then either picked up by hand or sent per email. All in all, the whole process takes up a lot of time of the steel mill's and engineering companies' employees. Furthermore, access to relevant drawings is severely delayed due to absences and overwork (sometimes the delays amount to up to a week).

The ADOS X system was designed to reduce the workload of everybody involved and to increase the speed of document delivery by having a completely integrated electronic process from externally searching and ordering a drawing to receiving, locking it against further changes and finally updating changed drawings. Thus the objective was to make as much functionality of the existing ADOS system as possible (and justifiable) available to the external engineers. However, the question, which functionality was to be made available and under which precise practicalities, was a matter of intense controversy. Even the different employees and managers within the steel mill did not agree in some cases.

Generally, the central most important determinant of ADOS X requirements from the steel mill's side is *trust*, i.e. trust in the external engineers. With some "old hand" engineers it appears practical to grant access the archive simply subject to protocol functionality that can be used to hold the engineering firm accountable for any questionable document access. Others should be permitted access to documents only after an explicit positive intervention by a steel mill employee (e.g. pressing an "ok" button). For others, access to certain parts of the drawing archive should be completely restricted. However, the opinions concerning these different levels of access (also compare Stiemerling and Wulf 1998) vary considerably, because of the many factors involved.

One area of general agreement became obvious during the requirements elicitation (or rather: envisioning) process (several interviews and workshop reported in Stevens 2000): the ADOS X system is regarded as a kind of agent associated with individual steel mill employees. It should be possible to tailor these individual agents to handle requests for drawings by the external engineers in an appropriate fashion. There was no general agreement concerning the question of who was supposed to perform the tailoring operations. Furthermore, it should be possible to implement company wide policies in order to take into account security directives from top-management or the central archive.

Another source of dynamics is introduced by company policies concerning general access to the Internet. At the time of the ADOS X development, only email was permitted as protocol for electronically interacting with the world outside the steel mill. Some steel mill employees speculated that this rather strict policy might change in the future, driven by the growing importance of the Internet. However, for the time being, ADOS X has to rely on email as sole means for communication between engineers and steel mill.

Furthermore, at development time, the meta-information of the construction drawings was stored in an Oracle database. It is expected that in the future this information will be stored as integrated part of the steel mill's SAP R/3 database.

Summarizing, the necessary range of tailorability is determined by a number of factors. First, the relationship between the external engineers and the steel mill appears to be under continuous negotiation. Consequently, the extent and the particularities of the access are subject to change and diversity. In particular, the persons directly involved are quite weary of spontaneous top-management decisions and policy changes without any apparent

relationship to the realities and practicalities of their everyday work. Secondly, technical factors are an influence on the requirements posed upon ADOS X. Anticipated changes in communication protocols and data storage mechanisms had to be accommodated.

Based on these dynamic and diverse requirements, a FLEXIBEANS component set was implemented by (Stevens 2000). The component set is described in the next subsection. Furthermore, it is demonstrated how the various flexibility demands stated in this section can be met by different compositions.

9.3.3 The ADOS X Component Set

The complete ADOS X component suite consists of more than 50 different client and server components (compare Stevens 2000). For the purpose of this chapter it shall suffice to discuss a simplified subset of these components that are important for supporting the range of flexibility presented above.

First, it is interesting to see how different levels of trust in external engineers can be reflected in differently composed access strategies. The means for implementing different strategies are essentially the five components depicted in table 9.2:

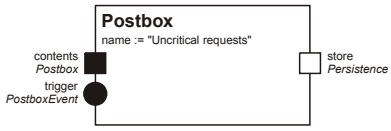
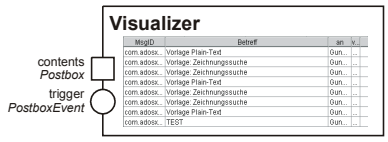
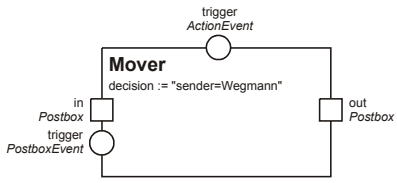
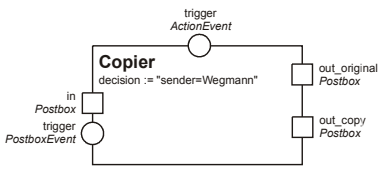
	**Receiver component** This component resides on a steel mill server and can be parameterized to access a certain company mail account that is used to send request from the external part of ADOS X to the internal part. The received requests are immediately written to a postbox component (see below).
	**Postbox component** As the name indicates, this server component persistently stores requests. In order to provide persistent storage, it connects to an EVOLVE external database.
	**Visualizer Component** This component presents the contents of a postbox at the user interface. Requests are selectable and the choice can be accessed by other components (e.g. the mover described below)
	**Mover component** The mover component is usually connected to two container components. According to its parameterization, it selects certain messages from one container and moves them to the other. The parameterization options are quite extensive and permit the specification of logical expressions for request selection (e.g. it can be parameterized to select all request by a certain external engineer concerning a certain contract). The mover can either be triggered automatically (by a timer) or by a button component. Consequently, it supports manual or automatic transport of messages from one container to the other.
	**Copier component** The copier provides functionality similar to the mover components. However, each request is copied and moved to an additional postbox.

Table 9.2: Component set for modeling different access policies

These five components form the core of a "component language" that permits the formulation of different access policies. The following demonstrates how the different levels of trust attributed to the external engineers are reflected in the composition of the ADOS X application at the side of the steel mill (the external client is not discussed here).

If the trust in an external engineer is high, the system can be composed in a fashion which automatically services certain requests. Figure 9.2 depicts such a composition:

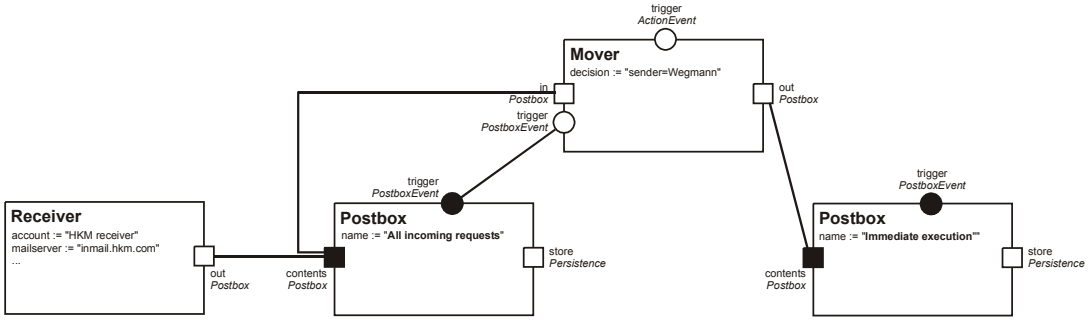


Figure 9.2: Requests of a trusted engineer are serviced automatically

The requests from the external engineer (the external client is not shown in figure 9.2. However, a screenshot is depicted in figure 9.7) arrive via email in the postbox "HKM receiver" on the host `inmail.hkm.com`, where it is read by the *receiver* component (on the very left of figure 9.2). The receiver stores a request in the postbox "all incoming requests". The state change in this postbox alerts (triggers) the *mover* component that – if the sender of the request is the engineer "Wegmann" (specified as parameter in the *mover* component instance) – moves the request to the postbox "immediate execution" (right side). Not shown in figure 9.2 is the part of the application that executes the requests and sends the results back to the external engineer. This example demonstrates, how the application can be composed, if the level of trust is high.

If an external engineer is not yet trusted, it is possible to interpose a manually triggered mover component between the input and the execution container. Consequently, an employee of the steel mill always has to explicitly agree to a request. This composition is depicted in figure 9.3:

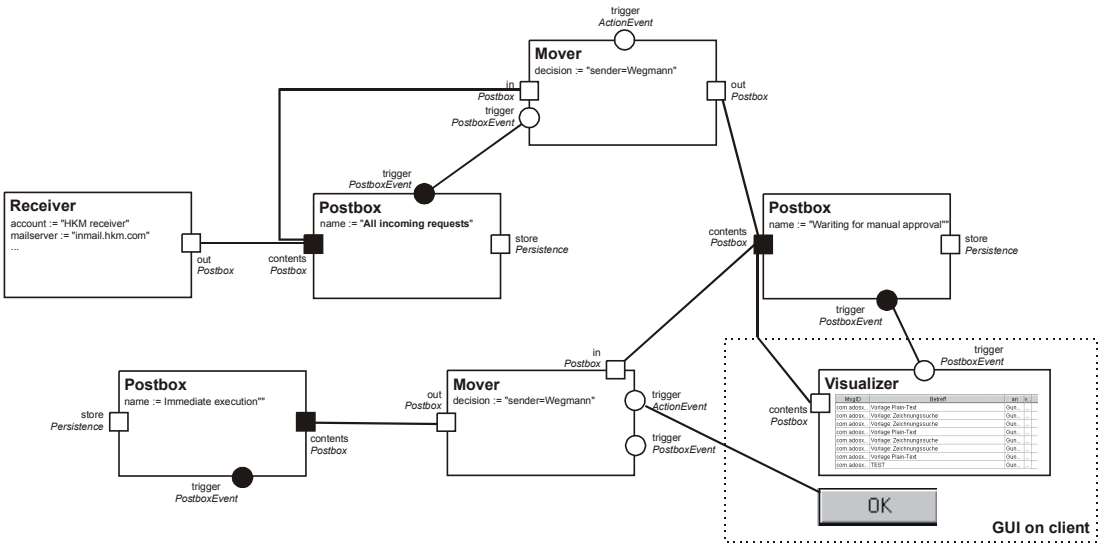


Figure 9.3: Execution of requests from another engineer has to be manually triggered

Here the initial handling of the request is similar to the one depicted in figure 9.2. However, the request is not moved directly to the postbox "immediate execution", but first to the postbox "waiting for manual approval". A client in the steel mill visualizes all requests in this postbox. If a steel mill employee decides that the request is valid and justified, he

approves with the "ok" button and a second mover component transfers the request to the postbox for immediate execution. Again, the further handling of the request is not shown here.

Another access policy permits automatically serviced requests. However, each request is protocolled in a postbox that can be checked by a steel mill employee. Figure 9.4 depicts this type of composition:

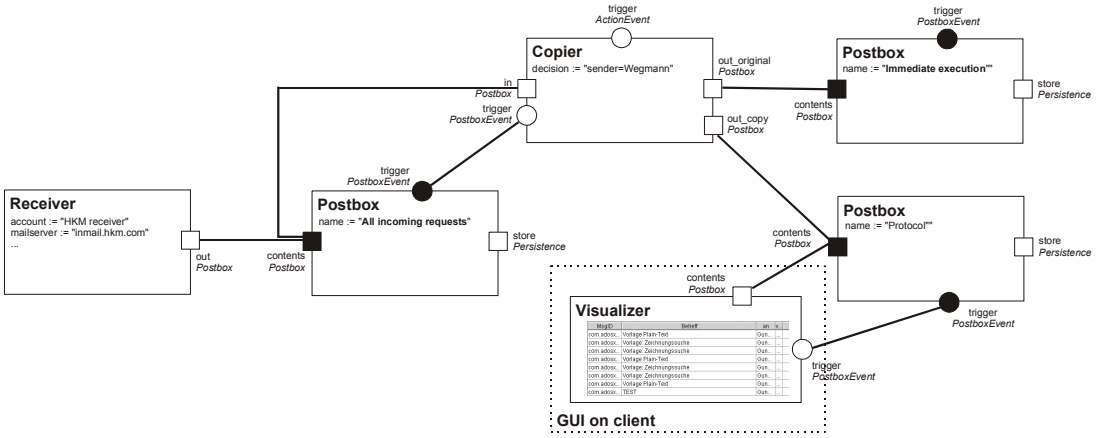


Figure 9.4: Protocolled execution of requests

Here the request is moved directly to the postbox "immediate execution". However, in the process a copy is stored in the postbox "protocol" that is visualized on a client in the steel mill.

These three example compositions demonstrate how the different levels of trust in the external engineers can be reflected in the compositional access control system of the ADOS X system. This is the main flexibility requirement that came out of the elicitation process. Furthermore, the decomposition of the ADOS X system clearly separates the access to the ADOS database from the other concerns of the system. This separation is expected to make a later connection of the system to a SAP R/3 database possible.

The component set for the visible parts of the system is designed to support the different requirements posed by external and internal users. Functionality can be quickly added and removed from the menu structure of the client. Table 9.3 shows a few of the interface components that control the document exchange process:

Create search request Suche item *MenuItems* Create order Bestellung item *MenuItems*	**Request Components** These components are used to create requests of different types on the client for the external engineers. They can be connected to a menu component (see below).
Menu Anfragen items *MenuItems*	**Menu Component** Components providing specific functionality can be connected to this component. The functionality is then accessible via a menu bar at the top of the client interface.

Table 9.3: Some interface components

Summarizing, the ADOS X component set demonstrates how different and sometimes dynamic requirements elicited in the development process can be met by different compositions. The simplified selection of components presented here gives an overview of the main concepts behind the ADOS X component set.

9.4 A Non-Groupware Application in the Financial Sector

The implementation of the EVOLVE platform is not particularly groupware specific. It generally supports client-server (even peer to peer) multi-user applications, distributed over a TCP/IP network. Consequently, the system should also be applicable to domains other than groupware. This section supports this claim by presenting a scenario in which component-based tailorability is applied to a non-groupware system in the banking sector. While the application has not been implemented, the scenario nevertheless demonstrates quite straightforwardly how the concepts developed in this work could be applied to a non-groupware application.

(Koch and Murer 1999) describe a banking system of the Credit Suisse in Zürich, Switzerland. The bank offers a wide range of financial products that are marketed in branch offices around the country. The branch office employees have access to the banking system via window-based GUIs on their desktop computers. They can, for instance, compute loans, open accounts, buy and sell stocks – in short everything a bank earns its money with. The system relies on data stored in large server-side databases (actually maintained on mainframe computers for "historic" reasons). Between the GUI and these servers lie other system components that define the products, i.e. the business logic of the bank. These components define, for instance, which conditions a loan is subject to or how many percent interest money in a certain type of account earns.

Since the bank is active in a highly dynamic market place, it is forced to continuously adapt its marketing strategy and product-line according to the changing demands of their customers and the activities of the competition. (Koch and Murer 1999, p. 194-195) describe how these dynamics impact upon the requirements for the banking system. The GUI components are subject to the most frequent changes, as distribution channels change and new customer groups are targeted. While a typical banking product has a lifetime of 10 years, its GUI components usually have to be changed after only 2 years. Considering the large number of differentiated products a modern bank has to offer, it is obvious that the banking application as a whole is subject to continuous change.

For the purpose of this section, it is interesting to contemplate how the introduction and fine-tuning of a new financial product could be supported by the EVOLVE platform. The next subsection describes a hypothetical example scenario and the subsection after that discusses how the EVOLVE platform supports it.

9.4.1 Scenario: A Building Renovation Loan Product

Assume that a new law has been passed by a country's legislative branch that intends to support the renovation of old buildings with the objective of creating small apartments for students in university towns. Subject to a catalog of criteria, landlords get certain tax-reductions when they participate. The bank decides to profit from this new law and creates a product that has the form of a special loan for the necessary construction work. The payback schedule and other conditions are adjusted to the specifics of the law. Once the economists of the bank have designed the product, the IT department is commissioned to implement the necessary IT support, i.e. the GUI and the assorted business logic components. A branch in a typical university town has been selected as pilot partner for the introduction of the new product. The IT department is supposed to install and fine-tune the first version of the product at that branch. During the pilot project it becomes obvious that, while the business logic components appear to work just fine, the arrangement of the various GUI components (e.g. input-fields for customer data) does not resemble the normal flow of the sales talk of the branch employees. They either have to jump around the screen or change their preferred sales pitch. Both of these alternatives are not acceptable for the end users. Furthermore, when the new product is presented to other branch offices, the bank becomes aware of several local laws in certain towns, which have to be taken into account in the banking system, thus changing the business logic. During the subsequent lifetime of the product, it is

possible that the competition comes up with an improved version of the product concept, forcing the bank to adapt its product as well.

9.4.2 Supporting the Banking Scenario with the EVOLVE Platform: A Future Scenario

The scenario exhibits a clear need for a tailorable system design owing to the dynamics and differentiation of the banking application domain. The EVOLVE system offers a number of different ways to support this scenario. During the development phase of the IT support for the new financial product, the EVOLVE system is used within the IT department to assemble the new application from pre-existing components (e.g. GUI components for inputting applicant's names and personal details) and newly implemented ones (e.g. components encapsulating the particularities of the new legislation). The application can then be alpha tested (without end user participation) and improved locally. Product components (containing the business logic) are placed either centrally on the EVOLVE server at the bank's head office or decentrally on the EVOLVE clients, depending on communication load and security considerations. Let the application be specified as a DCAT file (see chapter 8) named `new_loan.dcat` with assorted server and client CAT files. When the IT department is satisfied with the application and deems it ready for deployment in the pilot branch office, it simply adds the DCAT file to the `user management table` (see chapter 8) for the pilot end users. From that moment on, these users can access the new application. It appears in their EVOLVE client interface as a new item in the applications menu. Figure 9.5 shows the basic architecture of the application for two branch offices:

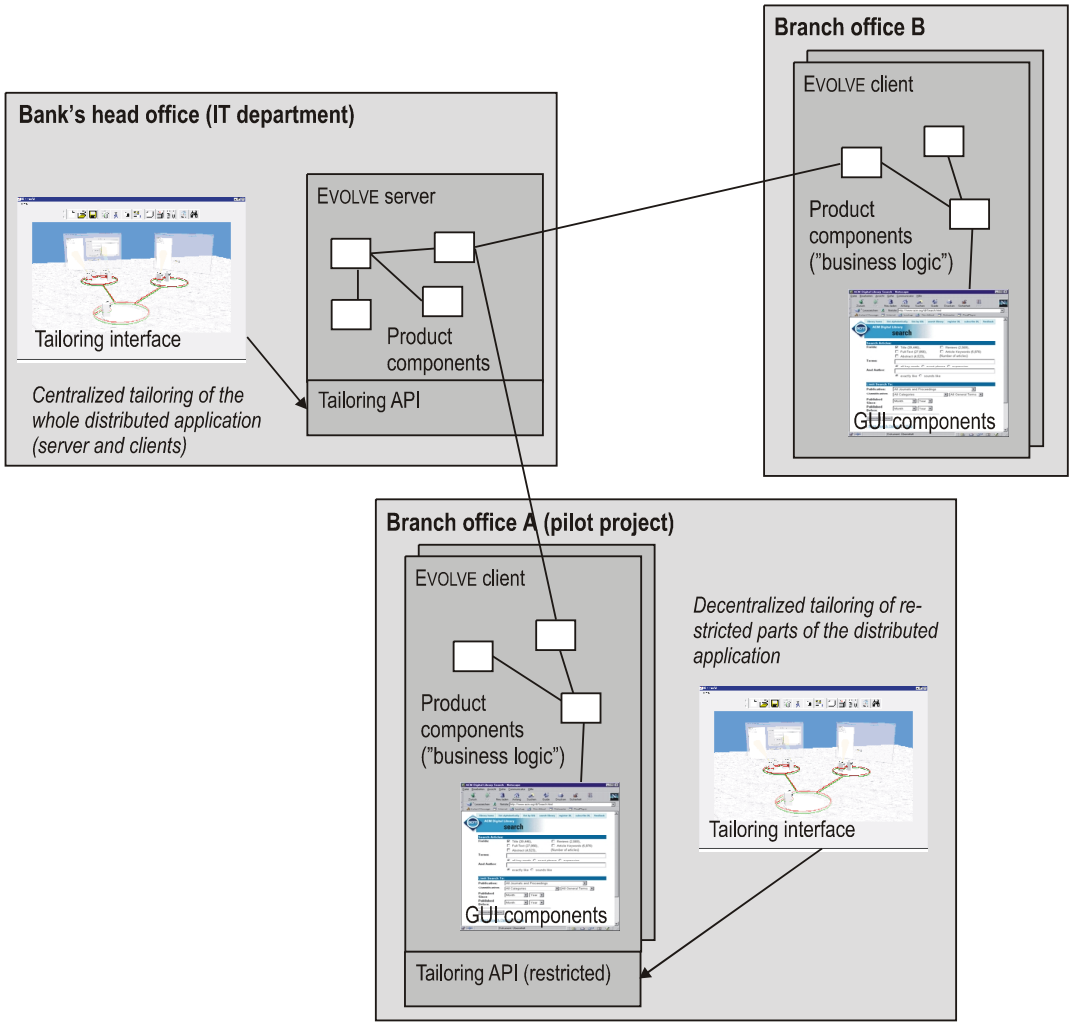


Figure 9.5: Supporting the banking scenario with the EVOLVE platform

However, the scenario described in the last subsection suggests that the end users are not satisfied with the arrangement of the GUI components, because they are in conflict with the structure of their usual sales talk. If the bank pursues a policy that is based on a high degree of centralized control (something IT departments are usually very fond of, especially in banks), the tailoring activities have to be performed by the IT department. Owing to the runtime tailoring capabilities of the EVOLVE platform, the complaining end user can simply call a member of the IT support staff who then – with immediate effect – changes the interface according to the instructions of the user. The staff member employs the 3D tailoring interface to navigate to the particular client. Before he begins to tailor, he restricts the scope of the changes to the single end user's application instance. Now he can move the GUI components around the screen (see the description of the 3D tailoring interface in chapter 8), until the end user is happy with the layout. For all this, the end-user does not even have to shut down his application and he sees the effects of the tailoring immediately.

If the bank has a less centralized IT policy, one could imagine the end user himself to start up a tailoring interface (see figure 9.5) and change the GUI according to his preferences. Currently, the EVOLVE platform does not offer facilities for specifying and enforcing tailoring rights. In the banking scenario it appears to be sensible to restrict the end users to simple tailoring operations (changing the x/y parameters) on GUI components. This is a point for future work.

After fine-tuning the application with the participation of the users in the pilot branch, the application can be deployed in all the bank's branches throughout the country – again simply by making the respective DCAT file available on the EVOLVE clients of all users.

While the tailoring operations performed so far only concern the GUI, i.e. visible components, the scenario also suggests possible changes of the invisible components defining the business logic, i.e. the product characteristics. These changes can be caused by differentiated local legislation or the necessity to react quickly to competitors' actions. Assume there is an non-GUI client component which checks the possible range of interest rates for the construction loans. A local competitor of the bank suddenly decreases its interest rates for that type of loan below the limit set and controlled by the component. In order to keep business running in that city, the bank decides to offer low interest loans as well. The necessary change to the application (perhaps a re-parameterization of the respective component) can be scope-controlled and thus applied to only the one branch in question. The runtime tailoring capabilities of the EVOLVE tailoring platform ensure, that the change is implemented immediately, without shutting down the system and waiting for the restart on the next day.

While the scenario's implementation is admittedly hypothetical, it nevertheless demonstrates that the concepts developed in this dissertation are applicable in a sensible way to domains other than groupware systems. Furthermore, the discussion also showed that in industrial applications – especially critical ones like in banking – a fine-grained model of tailoring rights can be useful. Its implementation should reflect how self-contained the different decentralized branches of an enterprise are. In a fully centralized scenario, only the IT department may change the application, while in a more decentralized organization, end users (or local experts) may perform certain tailoring activities.

9.5 Discussion

The three applications described in the first part of this chapter demonstrate how the component-based tailorability approach can be applied to groupware design and beyond. This section discusses how the implementations also serve the empirical validation of the theoretical results and concepts described in chapter 5 and 6. In particular, the theoretical claim concerning the implementation of efficient and natural component interaction is investigated. Furthermore, the question is raised, whether component management based on

static structures (as declared in CAT-files) is suitable for the description of multi-user applications. Another critical success factor is the EVOLVE platform's impact upon the performance of a software system. A significant degradation in start-up and runtime of the system would constitute a serious obstacle to the application of the approach. These three points,

- component interaction,
- component management, and
- performance

are discussed in the following subsections. The discussion is based on the experiences gained during the implementation of the MUD and ADOS X component sets.

9.5.1 Component Interaction

The main problem with the use of the JAVABEANS component model is that only JAVA events are supported as interaction primitive. As demonstrated in chapter 4, this leads to inefficient interactions and unwanted shifts in the chosen decomposition of an application's functionality. The theoretical treatment of this problem in chapter 5 identified overly tight synchronization and wrongly assigned initiative for interaction as the culprits of these effects. The theorem given in the same chapter demonstrates that JAVA events and shared variables together can solve this problem. The applications presented above are both based on the new FLEXIBEANS component model (presented in chapter 7) that takes the theoretical results into account.

Consequently, the developers of the two example applications (MUD and ADOS X) were expected not to encounter the problems that arose during the POLITeam search tool implementation. This proved to be true. The decompositions of the MUD system and ADOS X were implemented without shifts in decomposition (i.e. the concern which needed to be separate could always be implemented in different components without hindrance or restrictions imposed by the available interaction primitives). In no case, the interactions exhibit the overly tight synchronization or wrong placement of initiative that characterizes the POLITeam search tool implementation. Thus the theoretical claims are supported by the empirical evidence gathered through the example applications.

However, the implementation of the ADOS X system suggests an improvement concerning the specification of the way shared object connections are handled in the FLEXIBEANS component model. In the version of the model described in chapter 7, the shared object is provided by only *one* component instance. A reference to this *single* shared object can be given to a port of a component instance requiring a shared object. This port can only be connected to one shared object at a time (with the `setShared...()` and `forgetShared...()` method pattern). Consequently, the shared state space demanded by the theoretical results of chapter 5 is always implemented as a single shared object. During the development of the ADOS X system it became apparent that in some cases it is useful to construct the shared state space by aggregating *several* shared objects provided by different components. Regard, for instance, the menu structure depicted in figure 9.6:

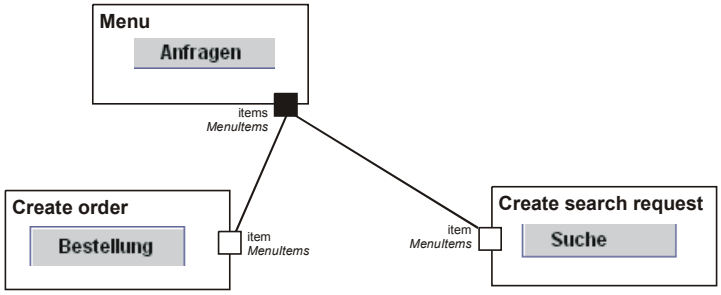


Figure 9.6: Implementation of a simple menu structure with FLEXIBEANS in ADOS X

At this point the developer of the ADOS X system wanted to provide the specifics of a menu item (e.g. the name of the item) as a shared object which can be added to a menu. Since a menu usually contains more than one entry, it would have been useful to construct the required shared state space by aggregating several menu items. Owing to the FLEXIBEANS specification this was not possible and the designer had to implement a workaround (the menu provides a list of entries as a single shared object in which the menu items can register themselves). He considered the workaround as unnatural and unintuitive. However, this problem does not concern the theoretical results (which do not make any suggestion as to how the shared state space is to be implemented) and can – consequently – be rectified by a change in the FLEXIBEANS specification. The `setShared...()` and `forgetShared...()` methods are replaced by `addShared...()` and `removeShared...()` methods which permit the construction of a shared state space as a list of shared objects. The integration of these method patterns is a point for future work (discussed in the next chapter).

Summarizing, the experiences made during the implementation of the two applications support the theoretical results developed in chapter 5. The only problem encountered concerns the FLEXIBEANS specification and can be solved straightforwardly.

9.5.2 Component Management

Component-based tailorability implies that the component structure remains static during the regular use of a software system. An activity that changes the structure is considered to be tailoring. For the implementation of applications that are more complex than the single window POLITeam search tool, the assumption of static component structures proves to be too simplistic. Multi-user applications (as supported by the EVOLVE system) obviously change their component structures during regular use, as well. If, for instance, a user logs into the system, the client components of the application are instantiated on the user's workstation. The server components are usually already running. If the user logs off again, the client structures are shut down again without affecting the server-side structures. The design of the EVOLVE platform accommodates these dynamically changing component structures by dividing the representation of these structure in server and client CAT files, as described in chapter 8.

During the implementation of the ADOS X system it became apparent that dynamic component structures are also required within a client. Examples for this are MDIs (Multi Document Interfaces) that permit the display of a variable number of document windows within one application. Figure 9.7 shows an example of MDI in the user interface of the ADOS X client:

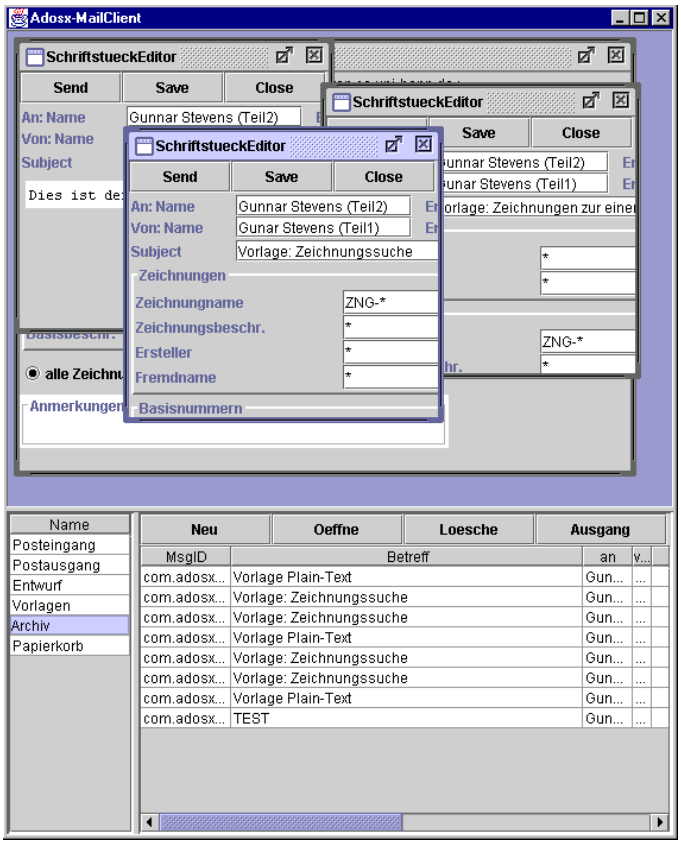


Figure 9.7: Multiple document windows (upper part of the screen shot) within the ADOS X client.

Several requests for construction drawings can be displayed as document windows at the same time. It would have been useful to describe the document window component structure as a CAT file, as well. However, the design of the EVOLVE platform does not permit dynamically changing component structures within a client. The developer had to implement a workaround which implements the whole MDI part of the user interface as one component instance *within which* document windows can be dynamically opened and closed. Consequently, the structure of the document windows cannot be tailored within EVOLVE but has to be re-programmed whenever the need for change arises – with all the disadvantages this implies (system shutdown, additional programming efforts etc.).

In order to provide the whole application – including MDIs – with component-based tailorability, one would have to permit the instantiation of CAT files by other application components. There could be, for instance, an "*open additional document window*" button component that – when pressed – instantiates a new window according to a structure definition in a CAT file. This is currently not supported by the EVOLVE system. However, the introduction of such a reflective facility (i.e. the application can reason and act upon its own compositional structure. Compare chapter 3) is problematic, because then the FLEXIBEANS components would obviously have to be "EVOLVE-aware". This means that they would need to access EVOLVE specific APIs (e.g. for reading and instantiating a CAT). Consequently, they cannot be deployed independently of the EVOLVE system anymore.

However, there are additional arguments for making FLEXIBEANS components more EVOLVE-aware. For instance, the MUD application contains a name component that is needed to have access to the name of the current user. Whenever the application is started up, this component either has to be pre-parameterized with the user name (requiring a new client CAT file for each user) or the user has to type in the name before using the tool. If the name component could access the user name already present in the EVOLVE client (because of the login procedure), the use of the tool would be facilitated. Other information stored within the EVOLVE system (e.g. a list of all user) could also be useful for application

components. Consequently, future work concerning the EVOLVE platform should address the question of access to EVOLVE information and the development of a model of compositional reflection for EVOLVE applications.

9.5.3 Performance

The core concepts of component-based tailorability developed in chapter 5 and 6 are not concerned with the performance of applications subjected to the approach. Nevertheless, it is interesting to see how the implemented approach impacts upon the time required for start-up and running an application. (Hinken 1999) has performed a number of performance test with the EVOLVE platform. The main results are summarized in the following.

First, it has to be noted that the direct interaction between application components is in no way mediated by the EVOLVE system. Consequently, during regular use, FLEXIBEANS applications deployed on the EVOLVE platform perform as well as monolithic applications (using JAVA RMI for remote interaction). The differences arise in the time needed for starting up an application, because the CAT file has to be read and evaluated. The tests by (Hinken 1999) demonstrate that a client application consisting of 64 (visible) components takes about 2 seconds longer to start up than a comparable monolithic application. Even disregarding the fact that the EVOLVE system is still a research prototype, this result is satisfactory.

Furthermore, it is of interest to measure the time it takes to adapt a component plan that is shared by several clients over the network. The tests show that removing a button (the "done" button) from a to-do list client shared by 16 users (EVOLVE clients) takes about 0.7 seconds. Again this result is more than satisfactory for a research prototype that is not yet optimized for execution speed. A more extensive presentation and discussion of these results can be found in (Hinken 1999).

9.6 Summary

The chapter demonstrates the applicability of the component-based tailorability approach by presenting two groupware systems that have been implemented as FLEXIBEANS component sets. A not yet implemented scenario involving an information system in the financial sector indicates the approach's applicability beyond groupware design.

The experiences made during the implementation of the example applications support the validity and relevance of the theoretical results developed in chapter 5. Possible improvements concern the FLEXIBEANS specification and subsequently the implementation of the EVOLVE platform:

- Support for *aggregated shared objects*: required shared objects ports should maintain lists of objects instead of permitting only the connection to a single shared object.
- Support for *compositional reflection*: application components should be able to reason about and act upon the application's compositional structures, e.g. in order to dynamically instantiate multiple document windows.
- Support for applications to *access EVOLVE data*: application components should be able to access information about users and the current usage situation (e.g. which user is logged in).

The performance tests demonstrate that the execution speed of applications running on the EVOLVE platform is not significantly degraded. The EVOLVE platform has no impact upon the regular execution of the system. The additionally required start-up time remains within acceptable bounds.

Chapter 10

Summary and Future Work

The work presented here demonstrates how the concept of a software component can be employed to provide a software system with the property of tailorability. The component-based tailorability approach supports developers in the design of software systems that can be adapted to diversified and dynamic individual, group or organizational requirements. Such software systems are expected to be more humane, effective and marketable than non-tailorable, monolithic systems.

First, an exploratory case study in the context of a groupware development project identified two primary technical problems in this endeavor:

- finding a *component model* which permits efficient implementation of a system's decomposition into components. The primary determinant of such a component model is the set of supported *interaction primitives*.
- *managing the compositional structure* of a software system after development and deployment. Desirable properties of an approach to this problem are *sharing* of compositional plans in plan instances and being able to *control the scope of a change* to a plan with respect to its instances.

In order to address the first problem, a theory of components and interaction was developed – based on the framework provided by data space theory by Cremers and Hibbard. The theory permits modeling and comparing different interaction primitives with respect to the problems identified in the exploratory experiment. A theorem was given which states that a component model providing both events and shared variables as interaction primitives permits efficient implementation of arbitrary system decompositions. Based on these considerations and the experiences gained in the case study, the FLEXIBEANS component model was developed. The empirical evidence provided by the implementation of three applications as sets of FLEXIBEANS support the theoretical results.

The second problem was addressed by developing a model for component management with the desired properties of sharing and scope control. The model is defined in a set-theoretic fashion independent of programming language. It is designed to support the development of runtime and tailoring environments. Developers can refer to the model when implementing the data structures and algorithms for representing, instantiating and dynamically tailoring compositional structures. The model was employed to support the development of the

EVOLVE system which provides component-based tailorability for FLEXIBEANS-based, distributed multi-user applications. The system includes a 3D user interface for tailoring.

The combination of FLEXIBEANS, the EVOLVE system and the three applications demonstrated the practical feasibility of the component-based tailorability approach for distributed multi-user applications in the area of groupware and beyond. The experiences made in the applications suggest a number of improvements which concern the specification of the FLEXIBEANS model and the EVOLVE system, namely support for *aggregated shared objects* and *compositional reflection*. They do not relate to the theoretical foundations developed as the main contribution of this work.

Thus the first element of future work is concerned with developing these improvements. While aggregated shared objects can be integrated into the FLEXIBEANS specification rather straightforwardly, a model for supporting compositional reflection in the EVOLVE system appears to be a challenge. Several tradeoffs concerning the degree of independence between component model and runtime and tailoring environment have to be taken into account.

Apart from these concrete improvements, the work presented here is intended to serve as a platform for the future investigation of questions not addressed so far.

First, if business critical applications are to be deployed on the EVOLVE system, a model for *tailoring rights* becomes imperative. The design of such a tailoring control system has to be undertaken with great care, in order to avoid the negation of the flexibility gained by the component-based tailorability approach. The model itself has to provide a great degree of flexibility and should be designed to accommodate different types of organizations and tailoring cultures (see chapter 3). Furthermore, in the implementation, the model should be cleanly integrated in the security model of the JAVA programming language.

Secondly, the novel runtime tailoring capabilities of the EVOLVE system raise the question of *safe tailoring*. Changing the composition of an executing application might lead to undesired or inconsistent system states. In the applications described in chapter 9, this proved generally uncritical, because concurrent usage was not high. However, if one takes applications that execute elaborate server-side computations (perhaps even arranged in transactions), removing a component instance during runtime might cause serious damage. Consequently, the development of a notion of consistency (general or application specific) and its support in the EVOLVE system is an important technical area of future work.

Other opportunities for future work are more interdisciplinary. For instance, from an empirical point of view, it would be interesting to investigate the question how well the component-based tailorability approach integrates with existing approach of component-based development. Can existing decompositions be used as basis for component-based tailorability and how much additional effort has to be put into the development process in order to come up with a component set suitable for the EVOLVE system? Chapter 3 has already mentioned the gap in the state of the art concerning design methodologies for eliciting diverse and dynamic requirements and transforming these into an appropriate system decomposition.

Another area for possible future work concerns the design of tailoring mechanisms building on component-based tailorability. For instance, the 3D interface, which was briefly described in chapter 8, ought to be systematically evaluated and tested with different types of end users. It could be rewarding to investigate how – especially non-interface – components can be represented in 3D and how they are perceived by the users. Furthermore, the interface offers a number of opportunities to technically support cooperative tailoring. Several remotely located tailors could “meet” in the virtual world of the EVOLVE system and adapt applications cooperatively.

In addition to the purely user-oriented approach pursued in the design of the 3D interface, the investigation of at least partially adaptive techniques for manipulating component-based applications is of interest. Especially for the initial configuration of the system, an assistance system might be helpful. Such a system could – according to certain yet to be determined

organizational parameters – automatically configure a first version of an application. Later on, this application could be fine-tuned by manual tailoring with the help of the 3D interface.

Coming to the end of this work, the author expresses his hopes that the considerations presented here constitute a small step towards technologies which are not overwhelming and frightening but supportive and perhaps even enjoyable. In a world where everyday life is more and more influenced by these technologies, it is important that they are understood and adapted to serve human and humane purposes.

Appendix A

The Syntax of CAT

A.1 Introduction

This appendix describes the textual syntax of CAT, using a simple context free grammar (see Hopcroft and Ullman 1979). Certain definitions (simple types, identifiers, and literals) will draw on the official JAVA Language Specification (see Gosling et al. 1996), because of this dissertation's affinity to that programming language.

A CAT-program describes the architecture of an application as a composition of components. At the lowest level there are implemented (or atomic) components. These components actually have a programmatic definition somewhere outside the CAT-program (e.g. in JavaBeans JAR-files, see JavaSoft 1997). Their properties are defined first. Then they can be aggregated over several levels of complex components. The final (mandatory) component is the system component, which contains all other components. It should be noted, that the implemented, complex and system component constructs describe types of components, which can be reused in different parts of the application (except for the unique system component). At system start-up, the system component construct serves as a unique starting point for the recursive instantiation of the application. In the search tool implementation described in chapter 4, the system component is part of the single CAT file describing the application. In the EVOLVE implementation described in chapter 8, the system component is defined in the DCAT file and refers to a number of simple CAT files without system components, which describe local parts of the application.

A first time reader is advised to begin by taking a look at the example at the end of the section, because many aspects of the language are supposed to be "intuitive".

A.2 CAT Body

The program body is described by the following grammatical productions. It contains a list of implemented component types, a list of abstract component types, and a system component:

```
CAT →                          Implemented_components_list Abstract_components_list
System_component
                               Implemented_components_list System_component

Implemented_components_list →  Implemented_components_list Implemented_component
                               Implemented_component

Abstract_components_list →     Abstract_components_list Abstract_component
                               Abstract_component
```

A.3 Atomic Components

Implemented component types are described by the following productions. The component body consists of a list of provided ports, a list of required ports, and a list of parameter definitions. The provided ports represent services that the component can offer if it is composed with other components. The required ports represent services that the component needs if it is composed with other components. The notion of ports and services is rather abstract. If CAT is used in conjunction with an industrial component model, it has to be mapped to the composition mechanism of the respective component model (e.g. the event-interaction style of JAVABEANS as described in chapter 3).

```
Implemented_component →   i_component comp_type_name { i_component_body }
i_component_body →        ports parameters

ports →                   required_list provided_list

required_list →           required_list required_port
                          required_port
```

Every port has a name and a type. The type represents the composition mechanism that applies to this port (e.g. *Push-Event-Source* in a hypothetical JavaBeans Button). In the CAT implementation described in chapter 8, the type is actually a combination of the data type and the behavioral type of the interaction (e.g. `RemoteEntry sharedObject`). The name is local to the component. In CAT the port types can be used to check the compositional correctness of the program (i.e. whether all port connections are legal).

```
required_port →           required port_name type_name;

provided_list →           provided_list provided_port
                          provided_port

provided_port →           provided port_name type_name;
```

The parameters of an implemented component are supposed to map to the properties of the respective component implementation (e.g. as a JAVABEAN). Therefore, they support the basic JAVA types.

```
parameters →              parameter_list parameter
                          parameter

parameter →               config_parameter parameter_name javatype;
                          config_parameter parameter_name javatype :=
                          javaliteral;
```

A.4 Types, Literals, and Names in CAT

As mentioned above, due to its current purpose, CAT makes use of the definition of simple types and literals of JAVA as described in (Gosling et al. 1996). However, in order to include

the string type (which is – in JAVA – not specified as a primitive type, but as a class) a construct was used that names all used types including strings. However, for the definition of literals, the reader is referred to page 19 in (Gosling et al. 1996). All non-terminals ending with *_name* are IdentifierChars (see p. 17 in Gosling et al. 1996), but not CAT-keywords (i_component, a_component, s_component, subcomponent, required, provided, config_parameter, bind, boolean, integer, byte, short, int, long, char, float, double, string).

`javatype` →	**boolean** **integer** **byte** **short** **int** **long** **char** **float** **double** **string**
`javaliteral` →	`Literal (p. 19 in Gosling et al. 1996)`
`*_name` →	`IdentifierChar (p. 17 in Gosling et al. 1996) but not CAT-keyword`

A.5 Complex Components

Complex components (in the CAT specification they are called abstract components for "historic reasons") are aggregations of implemented components or other – already specified – complex components. Thus, recursive specification of complex components is not allowed. In addition to the components ports and parameters, the component body contains the specification of subcomponents and bindings. The bindings can be between subcomponents only or between a subcomponent and the complex component. In former case, only required and provided ports of the same type can be bound. In the latter case required ports can only be bound to required ports, and provided ports only to provided ports. Again the type restriction applies. All ports of subcomponents and all newly defined ports of the complex component have to be bound.

Parameters of subcomponents can either be statically defined within the subcomponent parameter settings body or bound to parameters of the complex component. As complex components have no implementation, all parameters of the complex component have to be bound to parameters of subcomponents. Type restriction applies.

`Abstract_component` →	**a_component** `comp_type_name` { `i_component_body a_component_body` }
`a_component_body` →	`ports subcomponents bindings`
`subcomponents` →	`subcomponents subcomponent`
`subcomponent` →	**subcomponent** `local_name comp_type` { `parameter_settings` };
	subcomponent `local_name comp_type`;
`parameter_settings` →	`parameter_settings parameter_setting` `parameter_setting`
`paramerter_setting` →	`parameter_name := javaliteral;`
`bindings` →	`component_bindings parameter_bindings`
`component_bindings` →	`component_bindings component_binding`
`component_binding` →	**bind** `local_name . port_name -> local_name.port_name ;` **bind** `port_name -> local_name . port_name ;`

	bind local_name **.** port_name **->** port_name **;**
parameter_bindings →	parameter_bindings parameter_binding
parameter_binding →	**bind** parameter_name **->** local_name **.** parameter_name **;**

A.6 System Component

The system component is similar to a complex component, the difference being that it has no ports of its own. It represents a closed system. The system component can have, however, configuration parameters, which – again – have to be bound to parameters of subcomponents.

System_component →	**s_component** comp_type_name **{** s_component_body **}**
s_component_body →	paramters subcomponents bindings

A.7 Remarks on the Textual Syntax

The grammar presented here is designed for easy understanding and not for use in a compiler.

A.8 Example

The example presented here is drawn from (Magee et al. 1995), extended with the notion of configuration parameters. Regard the following CAT-program:

```
i_component low-pass-filter {
      required in sound_signal;
      provided out sound_signal;
      config_parameter border_frequency integer;
      }
i_component high-pass-filter {
      required in sound_signal;
      provided out sound_signal;
      config_parameter border_frequency integer;
      }
a_component band-pass-filter {
      required in sound_signal;
      provided out sound_signal;
      config_parameter upper_border_frequency integer;
      config_parameter lower_border_frequency integer;
      subcomponent filter1 low-pass-filter;
      subcomponent filter2 high-pass-filter;
      bind filter1.in -> in;
      bind filter1.out -> filter2.in;
      bind filter2.out -> out;
      bind upper_border_frequency -> filter1.border_frequency;
      bind lower_border_frequency -> filter2.border_frequency;
      }
s_component filter-system {
      config_parameter frequency integer;
      subcomponent filter3 band-pass-filter {
            upper_border_frequency := 1000;
            };
      bind band-pass-filter.lower_border_frequency ->
      frequency;
      }
```

This example contains the important elements of the CAT-language. At first, two implemented components are specified. Both have an in- and out-port each. The ports are of the type *sound-signal*, which is not interpreted in CAT, just statically checked within a bind-construct. The components each have one configuration parameter, which allows customizing the filters to a specific lower, resp. upper frequency-border.

The complex component defines a band pass filter, which consists of a pair of serially connected low pass and a high pass filters. The complex component defines two configuration parameters that are bound to the respective frequencies parameters of the subcomponents. The ports of the complex component are bound to the in-port of the first filter, resp. the out port of the second filter. The out-port of the first filter is connected to the in-port of the second filter.

The system component finally specifies the whole system as consisting of one band pass filter component fixed to a upper frequency of 1000, while the lower frequency is bound to a configuration parameter of the system.

Appendix B

Formal Conditions

B.1 Introduction

This appendix contains the complete conditions for the formal model of component management presented in chapter 6. The notation draws from set theory and first order predicate logic. For easier readability, the following convention is introduced: whenever a tuple is used in a logical expression, only those tuple-components that are actually referred to in the expression are named. The other tuple-components are simply written as dots (“.”). So, if only the name of a complex component is needed, one does not have to write (n, P, A, S, B_1, B_2, B_3), but can simply write (n,.,.,.,.,.,.).

B.2 Atomic Component

DEFINITION 6.1: *An* atomic component *is a tuple* (n,D,P,A) *with*

- $n \in N_{components}$,
- *D is a data space* (X,F,p),
- $P \subseteq N_{ports} \times N_{porttypes} \times \{req.\ ;\ prov.\ ;\ sym.\} \times \wp(F)$, *and*
- $A \subseteq N_{parameters} \times F$.

subject to the following conditions:

1. *Given* $(x,t,p,G) \in P$ *and* $(y,u,q,H) \in P$, *then* $x=y \Rightarrow (x,t,p,G) = (y,u,q,H)$ *and* $x \neq y \Rightarrow G \cap H = \emptyset$

   (port names are unique and the ports' information structures do not overlap)

2. *Given* (x,g) *and* $(y,h) \in A$, *then* $x=y \Rightarrow (x,g) = (y,h)$ *and* $x \neq y \Rightarrow g \neq h$

   (parameter names are unique and denote a unique part of the information structure of a component)

3. *Given* $(x,t,p,G) \in P$, *then* $G \neq \emptyset$

   (a port's information structure is not emtpy)

B.3 Component Framework

DEFINITION 6.2: *A* component framework *is a non-empty set C of atomic components with – given* $(n,D,P,A) \in C$, $(m,E,Q,B) \in C$, $(x,t,p,G) \in P$ *and* $(y,u,q,H) \in Q$:

1. $n=m \Rightarrow (n,D,P,A)= (m,E,Q,B)$

   (component names are unique)

2. $t=u \Rightarrow \{p;q\} \notin \{\{req.;sym.\};\{prov.;sym.\}\}$

   (a port type is either symmetric or asymmetric)

3. $t=u \Rightarrow \exists_{bijection\ m:G\to H}\ \forall_{f\in G}\ range(m(f)) = range(f)$.

   (two ports of the same type have the same information structures up to isomorphism)

B.4 Complex Components

DEFINITION 6.3: *Given a component framework C, a* complex component *is a tuple* $(n, P, A, S, B_1, B_2, B_3)$ *with*:

- $n \in N_{components}$,
- $P \subseteq N_{ports} \times N_{porttypes} \times \{req.\ ;\ prov.\ ;\ sym.\}$,
- $A \subseteq N_{parameters} \times R$,
- $S \subseteq N_{components} \times N_{instances} \times \wp(N_{parameter} \times V)$,
- $B_1 \subseteq N_{instance} \times N_{ports} \times N_{instances} \times N_{ports}$,
- $B_2 \subseteq N_{ports} \times N_{instances} \times N_{ports}$,
- $B_3 \subseteq N_{parameters} \times N_{instances} \times N_{parameters}$,

with R the set of all ranges of functions in the F's of the data spaces of C, with V the union of all elements of R, and subject to the following conditions:

1. $|S| = |\{n \mid (.,n,.) \in S\}|$

   (component instance names are unique)

2. $|P| = |\{n \mid (n,.,.) \in P\}|$

   (port names are unique)

3. $|A| = |\{n \mid (n,.) \in A\}|$

   (parameter names are unique)

4. $(n_1,.,n_2,.) \in B_1 \Rightarrow n_1, n_2 \in \{n \mid (.,n,.) \in S\}$

   (referential integrity of subcomponent binds w.r.t. instance names)

5. $(p_1,n_1,.) \in B_2 \Rightarrow n_1 \in \{n \mid (.,n,.) \in S\} \wedge p_1 \in \{n : (n,.,.) \in P\}$

   (referential integrity of port binds w.r.t. instance and port names)

6. $(p_1,n_1,.) \in B_3 \Rightarrow n_1 \in \{n \mid (.,n,.) \in S\} \wedge p_1 \in \{n : (n,.,.) \in P\}$

   (referential integrity of parameter binds w.r.t. instance and parameter names)

7. $\forall_{(.,i,P_i) \in S} \forall_{(.,i,p) \in B_3} \; p \notin \{p' \mid (p',.) \in P_i\}$

   (there are no conflicts between bound and given parameters of subcomponents)

8. $\forall_{(.,i,.) \in S} \forall_{(a,i,p),\,(b,i,p') \in B_3} \; p = p' \Rightarrow (a,i,p) = (b,i,p')$

   (subcomponents' parameters are bound once at most)

B.5 Component System

DEFINITION 6.4: *A* component system *is a tuple* (r, H, C) *with:*

- $r \in N_{component}$
- *H a non-empty set of complex components*
- *C a component framework*

subject to the following conditions:

1. $|H|+|C| = |\{n \mid (n,.,.,.,.,.,.) \in H\} \cup \{n \mid (n,.,.,.) \in C\}|$

   (complex and atomic components' names are unique)

2. $\forall_{(.,.,.,S,B_1,.,.) \in H} \; \forall_{(n_i,n_p,.,.) \in B_1 \wedge (n_c,n_i,.) \in S} \; (\exists_{(n',P',.,.,.,.,.) \in H} \; n' = n_c \wedge (n_p,.,.) \in P') \vee (\exists_{(n',.,P',.) \in C} \; n' = n_c \wedge (n_p,.,.,.) \in P')$

   (referential integrity of subcomponent binds' left side)

3. $\forall_{(.,.,.,S,B_1,.,.) \in H} \; \forall_{(.,.,n_i,n_p) \in B_1 \wedge (n_c,n_i,.) \in S} \; (\exists_{(n',\; P',.,.,.,.,.) \in H} \; n' = n_c \wedge (n_p,.,.) \in P') \vee (\exists_{(n',.,P',.) \in C} \; n' = n_c \wedge (n_p,.,.,.) \in P')$

   (referential integrity of subcomponent binds' right side)

4. $\forall_{(.,.,.,S,.,B_2,.) \in H} \; \forall_{(.,n_i,n_p) \in B_2 \;\wedge\; (n_c,n_i,.) \in S} \; (\exists_{(n',\; P',.,.,.,.,.) \in H} \; n' = n_c \wedge (n_p,.,.) \in P') \vee (\exists_{(n',.,P',.) \in C} \; n' = n_c \wedge (n_p,.,.,.) \in P')$

   (referential integrity of port binds)

5. $\forall_{(.,.,.,S,.,.,.,B_3)\in H}\ \forall_{(.,n_i,n_a)\in B_3 \wedge (n_c,n_i,.)\in S}\ (\exists_{(n',.,A,.,.,.,.)\in H}\ n'=n_c \wedge (n_a,.)\in A) \vee (\exists_{(n',.,.,.,A)\in C}\ n'=n_c \wedge (n_a,.)\in A)$

(referential integrity of parameter binds)

6. $\forall_{(.,.,.,.,S,B_1,.,.,.)\in H}\ \forall_{(n_i,n_p,n'_i,n'_p)\in B_1}\ \forall_{(n_c,n_i,.),(n'_c,n'_i,.)\in S}\ \forall_{[(n_c,P,.,.,.,.,.,.,.)\in H \vee (n_c,P,.,.,.)\in C]}\ \forall_{[(n'_c,P',.,.,.,.,.,.,.)\in H \vee (n'_c,P',.,.,.)\in C]}\ \forall_{[(n_p,p_t,p_p)\in P \vee (n_p,p_t,p_p,.,.)\in P]}\ \forall_{[(n'_p,p'_t,p'_p)\in P' \vee (n'_p,p'_t,p'_p,.,.)\in P']}\ p_t = p'_t \wedge \{p_p;p'_p\} \notin \{\{req.;sym.\};\{prov.;sym.\}\}$

(static type and polarity compatibility of subcomponent binds)

7. $\forall_{(.,P,.,.,S,.,.,B_2,.,.)\in H}\ \forall_{(n_p,n'_i,n'_p)\in B_2}\ \forall_{(n'_c,n'_i,.)\in S}\ \forall_{[(n'_c,P',.,.,.,.,.,.,.)\in H \vee (n'_c,P',.,.,.)\in C]}\ \forall_{[(n_p,p_t,p_p)\in P]}\ \forall_{[(n'_p,p'_t,p'_p)\in P' \vee (n'_p,p'_t,p'_p,.,.)\in P']}\ p_t=p'_t \wedge \{p_p;p'_p\} \notin \{\{req.;sym.\};\{prov.;sym.\};\{prov.;req.\}\}$

(static type and polarity compatibility of port binds)

8. $\forall_{(.,.,A,S,.,.,.,B_3)\in H}\ \forall_{(n_p,n'_i,n'_p)\in B_3}\ \forall_{(n'_c,n'_i,.)\in S}\ \forall_{[(n'_c,.,A',.,.,.,.,.,.)\in H \vee (n'_c,.,A',.,.)\in C]}\ \forall_{(n_p,.,R)\in A}\ ((n'_p,R')\in A' \Rightarrow R'=R) \wedge ((n'_p,f)\in A' \Rightarrow range(f)=R)$

(static type compatibility of parameter binds)

9. $\forall_{(.,.,.,.,S,.,.,.,.)\in H}\ \forall_{(n_c,.,P)\in S}\ \forall_{(n_p,v)\in P}\ \exists_{[(n'_c,.,A',.,.,.,.,.,.)\in H \vee (n'_c,.,.,A',.,.)\in C]}\ [\exists_{(n_p,R)\in A} v\in R \vee \exists_{(n_p,f)\in A} v\in range(f)]$

(static type compatibility of subcomponent parameterization)

10. *Given* $T \subseteq H \times H$ *with* $(x,y) \in T \Leftrightarrow \exists_{n\in N_{components}}\ x = (.,.,.,S,.,.,.) \wedge y = (n,.,.,.,.,.,.,.) \wedge (n,.,.,.) \in S$. *Let T* be the transitive closure of T:*

$\forall_{(x,y)\in T^*}\ (y,x) \notin T^*$

(no cycles and no self references)

11. *Given T defined as above:*

$\exists_{h\in H}\ h = (r,.,.,.,.,.,.,.) \wedge \forall_{(x,y)\in T}\ y \neq h$

(root component of the system exists and is not a subcomponent of another component)

12. $\forall_{(.,.,.,S,.,.,.)\in H} \forall_{(n,.,.)\in S}\ (n,.,.,.,.,.,.,.)\in H \vee (n,.,.,.)\in C$

(referential integrity w.r.t. all subcomponents)

References

1. Allen, R. J., “A Formal Approach to Software Architecture”, Ph.D. thesis, *School of Computer Science* Pittsburg: Carnegie Mellon University, 1997.
2. Bellissard, L., Atallah, S. B., Boyer, F., and Riveill, M., “Distributed Application Configuration,” in: Proceedings of 16th International Conference on Distributed Computing Systems, Hong-Kong, IEEE Computer Society, pp. 579-585, 1996.
3. Bellissard, L., Atallah, S. B., Kerbrat, A., and Riveill, M., “Component-based Programming and Application Management with Olan,” in: Proceedings of Object-Based Parallel and Distributed Computation France-Japan Workshop OBPDC '95, J. P. Briot, J. M. Geib, and A. Yonezawa, Eds., Springer Verlag, pp. 290-308, 1995.
4. Bentley, R. and Dourish, P., “Medium versus Mechanism: Supporting Collaboration through Customisation,” in: Proceedings of ECSCW 95, H. Marmolin, Y. Sundblad, and K. Schmidt, Eds., Stockholm, Sweden, Kluwer, pp. 133-148, 1995.
5. Bohning, C., “A tailorable electronic meeting system”, Master thesis, *Department of Computer Science III* Bonn: University of Bonn, 2000.
6. Börger, E. and Schulte, W., “A Programmer Friendly Modular Definition of the Semantics of Java,” in *Formal Syntax and Semantics of Java(tm)*, J. Alves-Foss, Ed.: LNCS (Springer-Verlag), 1998, pp. (to appear).
7. Carter, K. and Henderson, A., “Tailoring Culture,” in: Proceedings of 13th IRIS, R. Hellmann, M. Ruohonen, and P. Sørgaard, Eds., Abo Akademi University, pp. 103-116, 1990.
8. Churchill, E. and Bly, S., “Virtual Environments at Work: ongoing use of MUDs in the Workplace,” in: Proceedings of WACC '99, D. Georgakopoulos, W. Prinz, and A. L. Wolf, Eds., San Francisco, California, ACM Press, pp. 99-108, 1999.
9. Collins, *Collins Cobuilt Dictionary*. Birmingham: Collins Publishers, 1987.
10. Cortes, M., “A Coordination Language For Building Collaborative Applications,” *Journal of Computer Supported Cooperative Work*, Special Issue on Tailorability and Cooperative Systems, pp. 5-31, 2000.

11. Cremers, A. B. and Hibbard, T. N., “Formal Modeling of Virtual Machines,” *IEEE Transactions on Software Engineering*, vol. 4, 5, pp. 426-436, 1978.

12. Cremers, A. B. and Hibbard, T. N., “Executable Specification of Concurrent Algorithms in Terms of Applicative Data Space Notation,” in *VLSI and Modern Signal Processing*, S. Y. Kyng, H. J. Whitehouse, and T. Kailath, Eds. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1985.

13. Cremers, A. B. and Hibbard, T. N., “A Programming Notation for Locally Synchronized Algorithms,” in: Proceedings of International Workshop on Parallel Computing and VLSI, P. Bertolazzi and F. Luccio, Eds., Amalfi, Italy, North-Holland, pp. 341-376, 1985.

14. Cremers, A. B. and Hibbard, T. N., “Subspaces: Factorization and Communication,” in: Proceedings of Computational And Combinatorial Methods in Systems Theory, C. I. Byrnes and A. Lindquist, Eds., Stockholm, Sweden, Elsevier Science Publishers B.V. (North-Holland), pp. 383-396, 1986.

15. Cremers, A. B., Kahler, H., Pfeifer, A., Stiemerling, O., and Wulf, V., “PoliTeam - kokonstruktive und evolutionäre Entwicklung einer Groupware,” *Informatik Spektrum*, vol. 21, 4, pp. 194-202, 1998.

16. Cypher, A., *Watch What I Do - Programming by Demonstration*. Cambridge, Massachusetts: The MIT Press, 1993.

17. DEC, “LinkWorks,” 3.0 ed, 1997.

18. DeRemer, F. and Kron, H. H., “Programming-in-the-Large Versus Programming-in-the-Small,” *IEEE Transactions on Software Engineering*, vol. 2, 2, pp. 80-86, 1976.

19. Dierker, M. and Sander, M., *Lotus Notes 4.5 and Domino*: Addison-Wesley, Bonn, 1997.

20. Dourish, P., “Open implementation and flexibility in CSCW toolkits”, Ph.D. Thesis, London: University College, 1996.

21. Ecklund, E. F., Delcambre, L. M. L., and Freiling, M. J., “Change Cases: Use Cases that Identify Future Requirements,” in: Proceedings of OOPSLA '96, CA, USA, ACM Press, pp. 342-358, 1996.

22. Ellis, C. A., Gibbs, S. J., and Rein, G. L., “Groupware - some Issues and Experiences,” *Communications of the ACM*, vol. 34, 1, pp. 38-58, 1991.

23. Engelskirchen, T., “Explorierbarkeit von Groupware”, Master thesis, *Department of Computer Science III* Bonn: University of Bonn, 2000.

24. Ferber, J., “Computational Reflection in Class-based Object Oriented Languages,” in: Proceedings of OOPSLA '89, ACM Press, 1989.

25. Fosså, H. and Sloman, M., “Implementing Interactive Configuration Management for Distributed Systems,” in: Proceedings of International Conference on Configurable Distributed Systems (ICCDS '96), Annapolis, Maryland, 1996.

26. Fowler, M. and Scott, K., *UML Distilled*. Reading, Massachusetts: Addison Wesley, 1997.

27. Fox, D., Burgard, W., Thrun, S., and Cremers, A. B., “Position Estimation for Mobile Robots in Dynamic Environments,” in: Proceedings of 15th Conference on Artificial Intelligence, Madison, Wisconsin, 1998.

28. Friedrich, J., “Adaptivität und Adaptierbarkeit informationstechnischer Systeme in der Arbeitswelt - zur Sozialverträglichkeit zweier Paradigmen,” in: Proceedings of GI - 20. Jahrestagung, A. Reuter, Ed., Springer-Verlag, 1990.

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

30. Gantt, M. and Nardi, B. A., "Gardeners and Gurus: Patterns of Cooperation Among CAD Users," in: Proceedings of CHI '92, ACM Press, pp. 107-117, 1992.

31. Garlan, D. and Perry, D. E., "Introduction to the Special Issue on Software Architecture," *IEEE Transactions on Software Engineering*, vol. 21, 4, pp. 269-274, 1995.

32. Gibbons, A. and Rytter, W., *Efficient Parallel Algorithms*. Cambridge: Cambridge University Press, 1988.

33. Gosling, J., Joy, B., and Steele, G., *The Java Language Specification*. Reading, Mass.: Addision-Wesley, 1996.

34. Greenbaum, J. and Kyng, M., *Design at Work*. Hillsdale, New Jersey: Lawrence Erlbaum Associates, Publishers, 1991.

35. Greenberg, S., "Personalizable groupware: accommodating individual roles and group differences," in: Proceedings of ECSCW '91, Amsterdam, The Netherlands, Kluwer, pp. 17-31, 1991.

36. Grønbæk, K., Kyng, M., and Mogensen, P., "Cooperative Experimental System Development - Cooperative Techniques Beyond Initial Design and Analysis," in: Proceedings of Third Decennial Conference: Computers in Context: Joining Forces in Design, Aarhus, Denmark, Aahrus University, pp. 20-29, 1995.

37. Grudin, J., "Groupware and social dynamics: eight challenges for developers," *Communications of the ACM*, vol. 37, 1, pp. 93-105, 1994.

38. Gurevich, Y., "Draft of the ASM Guide," University of Michigan, Michigan, Technical Report Nr: CSE-TR-336-97, May, 1997.

39. Hallenberger, M., "Programmierung einer interaktiven 3D-Schnittstelle am Beispiel einer Anpassungsschnittstelle für komponentenbasierte Anpassbarkeit", Master thesis, *Department of Computer Science III* Bonn: University of Bonn, 2000.

40. Harel, D., "On Visual Formalisms," *Communications of the ACM*, vol. 31, 5, pp. 514-530, 1988.

41. Henderson, A. and Kyng, M., "There's No Place Like Home: Continuing Design in Use," in *Design At Work*, J. Greenbaum and M. Kyng, Eds. Hillsdale, New Jersey: Lawrence Erlbaum Associates, Publishers, 1991, pp. 219-240.

42. Herrmann, T., "Workflow Management Systems: Ensuring Organizational Flexibility by Possibilities of Adaptation and Negotiation," in: Proceedings of COOCS '95, Milipitas, California, ACM-Press, pp. 83-94, 1995.

43. Hinken, R., "Verteilte komponentenbasierte Anpassbarkeit von Groupware - Eine Laufzeit- und Anpassungsplattform", Master thesis, *Department of Computer Science III* Bonn: University of Bonn, 1999.

44. Hinken, R. and Stiemerling, O., "The Evolve Tailoring API Specification," 1.0 ed, 1999, (available at: http://www.informatik.uni-bonn.de/~os/Documentation/index.html).

45. Hoare, C. A. R., *Communicating Sequential Processes*. Englewood Cliffs, New Jersey: Prentice Hall International, 1985.

46. Hopcroft, J. E. and Ullman, J. D., *Introduction to Automata Theory, Languages, and Computation*. Reading, Mass.: Addison-Wesley, 1979.

47. Hughes, J., King, V., Rodden, T., and Anderson, H., “Moving Out from the Control Room: Ethnography in System Design,” in: Proceedings of CSCW '94, R. Furuta and C. Neuwirth, Eds., Chapel Hill, NC, USA, ACM-Press, pp. 429-440, 1994.

48. IBM, “Visual Age for Java,” 1.0 ed, 1998.

49. Jacobsen, I., Christerson, M., Jonsson, P., and Övergaard, G., *Object-Oriented Software Engineering. A Use Case Driven Approach*: ACM Press, 1992.

50. JavaSoft, “JavaBeans 1.0 API Specification,” 1.00-A ed. Mountain View, California: SUN Microsystems, 1997.

51. JavaSoft, “Java 3D API,” beta 1 ed, 1998, (available at: http://java.sun.com /products /java-media /3D/index.html).

52. JavaSoft, “Java RMI,” , 1998.

53. Kahler, H., “Developing Groupware with Evolution and Participation, a Case Study,” in: Proceedings of PDC '96, J. Blomberg, F. Kensing, and E. Dykstra-Erickson, Eds., Cambridge, Massachusetts, USA, ACM Press, pp. 173-182, 1996.

54. Kahler, H., Stiemerling, O., Wulf, V., and Höpfner, J.-G., “Gemeinsame Anpassung von Einzelplatzanwendungen,” in: Proceedings of Software Ergonomie '99, Walldorf, 1999.

55. Kiczales, G., “Beyond the Black Box: Open Implementation,” *IEEE Software*, vol. 13, 1996.

56. Kiczales, G., Rivières, J. d., and Bobrow, D. G., *The Art of The Metaobject Protocol*. Cambridge, Massachusetts: The MIT Press, 1991.

57. Klöckner, K., Mambrey, P., Sohlenkamp, M., Prinz, W., Fuchs, L., Kolvenbach, S., Pankoke-Babatz, U., and Syri, A., “POLITeam --- Bridging the Gap between Bonn and Berlin for and with the Users,” in: Proceedings of ECSCW '95, H. Marmolin, Y. Sundblad, and K. Schmidt, Eds., Stockholm, Sweden, Kluwer, pp. 17-32, 1995.

58. Koch, T. and Murer, S., “Service Architecture Integrates Mainframes in a CORBA Environment,” in: Proceedings of EDOC '99, Mannheim, Germany, IEEE Computer Society Press, pp. 194-203, 1999.

59. Kreifelts, T., Hinrichs, E., and Woetzel, G., “Sharing To-Do Lists with a Distributed Task Manager,” in: Proceedings of ECSCW '93, G. d. Michelis, C. Simone, and K. Schmidt, Eds., Milan, Italy, Kluwer Academic Publishers, pp. 31-46, 1993.

60. Kühme, T., Dieterich, H., Malinowski, U., and Schneider-Hufschmidt, M., “Approaches to Adaptivity in User Interface Technology: Survey and Taxonomy,” in: Proceedings of IFIP '92, pp. 225-252, 1992.

61. Lampson, B. W., “Protection,” *ACM Operation System Review*, vol. 8, pp. 18-24, 1974.

62. Lindholm, T. and Yellin, F., *The Java Virtual Machine Specification*. Reading, Massachusetts: Addison-Wesley, 1996.

63. Lumpe, M., Schneider, J.-G., Nierstrasz, O., and Achermann, F., “Towards a formal composition language,” in: Proceedings of ESEC '97 Workshop on Foundations of Component-Based Systems, G. T. Leavens and M. Sitamaran, Eds., Zürich, Switzerland, pp. 178-187, 1997.

64. Mackay, W. E., “Patterns of sharing customizable software,” in: Proceedings of CSCW '90, Los Angeles, CA, ACM Press, pp. 209-221, 1990.

65. Mackay, W. E., “Triggers and Barriers to Customizing Software,” in: Proceedings of CHI '91, ACM Press, pp. 153-160, 1991.

66. MacLean, A., Carter, K., Lövstrand, L., and Moran, T., "User-tailorable systems: Pressing the issue with Buttons," in: Proceedings of CHI '90, ACM Press, New York, pp. 175-182, 1990.

67. Maes, P., "Computational Reflection", Ph.D. Thesis, *Department of Computer Science* Bruessels: University of Bruessels, 1987.

68. Magee, J., Dulay, N., Eisenbach, S., and Kramer, J., "Specifying Distributed Software Architectures," in: Proceedings of 5th European Software Engineering Conference, Barcelona, LNCS 989 (Springer-Verlag), pp. 137-153, 1995.

69. Magee, J., Kramer, J., and Sloman, M., "Constructing Distributed Systems in Conic," *IEEE Transactions on Software Engineering*, vol. 15, 6, pp. 663-675, 1989.

70. Malone, T. W., Lai, K.-Y., and Fry, C., "Experiments with Oval: A Radically Tailorable Tool for Cooperative Work," *ACM Transactions on Information Systems*, vol. 13, 2, pp. 177-205, 1995.

71. McIlroy, D. M., "Mass-produces software components," in: Proceedings of North Atlantic Treaty Organization (NATO) Conference on Software Engineering, J. M. Buxton, P. Naur, and B. Randell, Eds., Garmisch-Partenkirchen, Germany, 1968.

72. Mens, K., Steyaert, P., and Lucas, C., "Reuse Contracts: Managing Evolution in Adaptable Systems," *Special Issues in Object-Oriented Programming: Workshop Reader of the 10th European Conference on Object-Oriented Programming ECOOP '96, Linz*, pp. 37 42, 1996.

73. Microsoft, "Visual Basic," 4.0 ed, 1996.

74. Microsoft, "Exchange," 1.0 ed, 1998.

75. Microsoft, "COM," 1.0 ed, 1999, (available at: http://www.microsoft.com /com/about.asp).

76. Milner, R., *A Calculus of Communicating Systems*, vol. 92 (LNCS). Berlin, Heidelberg, New York: Springer-Verlag, 1980.

77. Milner, R., Parrow, J., and Walker, D., "A calculus of mobile processes, Parts I and II," *Journal of Information and Computing*, vol. 100, pp. 1-77, 1992.

78. Mørch, A., "Method and Tools for Tailoring of Object-oriented Applications: An Evolving Artifacts Approach", PhD-Thesis, *Department of Computer Science* Oslo: University of Oslo, 1997.

79. Myers, B. A., "Taxonomies of Visual Programming and Program Visualization," *Journal of Visual Languages and Computing*, vol. 1, pp. 97-123, 1990.

80. Nardi, B. A., *A Small Matter of Programming - Perspectives on End User Programming*. Cambridge, Massachusetts: The MIT Press, 1993.

81. Nierstrasz, O., "A Survey of Object-Oriented Concepts," in *Object-Oriented Concepts, Databases and Applications*, W. Kim and F. Lochovsky, Eds.: ACM Press and Addison-Wesley, 1989, pp. 3-21.

82. Nierstrasz, O. and Tsichritzis, D., *Object-Oriented Software Composition*. Upper Saddle River, New Jersey: Prentice Hall, 1995.

83. Oberquelle, H., "Situationsbedingte und benutzerorientierte Anpaßbarkeit von Groupware," in *Menschengerechte Groupware - Software-ergonomische Gestaltung und partizipative Umsetzung*, A. Hartmann, T. Herrmann, M. Rohde, and V. Wulf, Eds. Stuttgart: Teubner, 1994, pp. 31-50.

84. OMG, "The Common Object Request Broker: Architecture and Specification," 2.0 ed: Object Management Group, 1995.

85. Paepcke, A., *Object-Oriented Programming - The CLOS Perspective*, vol. 1. Cambridge, Massachusetts: The MIT Press, 1993.

86. Parnas, D. L., "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM*, vol. 15, 12, pp. 1053-1058, 1972.

87. Pree, W., *Komponentenbasierte Softwareentwicklung mit Frameworks*: dpunkt Verlag, 1997.

88. Prieto-Diaz, R. and Neighbors, J., "Module Interconnection Languages," *Journal of Systems and Software*, vol. 6, 4, pp. 307-334, 1986.

89. Radestock, M. and Eisenbach, S., "What do you get from a pi-calculus semantics?," in: Proceedings of Parallel Architectures and Languages Europe, Springer, pp. 635-647, 1994.

90. Rittenbruch, M., Kahler, H., and B.Cremers, A., "Unterstützung von Kooperation in einer Virtuellen Organisation," in: Proceedings of Wirtschaftsinformatik '99, A. W. Scheer, Ed., Saarbrücken, 1999.

91. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorenson, W., *Object-Oriented Modeling and Design*. Englewood Cliffs, New Jersey: Prentice Hall, 1991.

92. Shaw, M., DeLine, R., Klein, D. V., Ross, T. L., Young, D. M., and Zelesnik, G., "Abstractions for Software Architecture and Tools to Support Them," *IEEE Transactions on Software Engineering*, vol. 21, 4, pp. 314-335, 1995.

93. Shen, H. and Dewan, P., "Access Control for Collaborative Environments," in: Proceedings of ACM Conference on Computer Supported Cooperative Work (CSCW '92), Toronto, Canada, ACM Press, pp. 51-58, 1992.

94. Shneiderman, B. and Maes, P., "Direct Manipulation vs Interface Agents," *Interactions*, November + December, pp. 42-61, 1997.

95. Silberschatz, A. and Galvin, P., *Operating System Concepts*: John Wiley & Sons, 1998.

96. Simone, C. and Schmidt, K., "Taking the distributed nature of cooperative work seriously," in: Proceedings of 6th Euromicro Workshop on Parallel and Distributed Processing, Madrid, Spain, IEEE Press, pp. 295-301, 1998.

97. Smith, B. C., "Reflection", Ph. D. Thesis, Cambridge, Masschusetts: MIT, 1982.

98. Spielmann, M., "Model Checking Abstract State Machines and Beyond," in: Proceedings of Abstract State Machines - ASM 2000, Y. Gurevich, P. W. Kutter, M. Odersky, and L. Thiele, Eds., Monte Verita, Switzerland, ETH Zürich, pp. 357-375, 2000.

99. Stallmann, R. M., "EMACS: The Extensible, Customizable Self-Documenting Display Editor," in: Proceedings of ACM SIGPLAN SIGOA Symposium on Text Manipulation, Portland, Oregon, 1981.

100. Stevens, G., "A tailorable inter-company groupware system", Master thesis, *Department of Computer Science III* Bonn: University of Bonn, 2000.

101. Stiemerling, O., "Anpaßbarkeit von Groupware - ein regelbasierter Ansatz", Diploma thesis, *Department of Computer Science III* Bonn: University of Bonn, 1996.

102. Stiemerling, O., Kahler, H., and Wulf, V., "How to Make Software Softer - Designing Tailorable Applications," in: Proceedings of DIS '97, G. v. d. Veer, A. Henderson, and S. Coles, Eds., Amsterdam, ACM Press, pp. 365-376, 1997.

103. Stiemerling, O., Rohde, M., and Wulf, V., "Integrated Organization and Technology Development - The Case of the OrgTech Project," in: Proceedings of Concurrent

Engineering '98, S. Fukuda and P. L. Chawadhry, Eds., Tokyo, Japan, ISPE, pp. 181-187, 1998.

104. Stiemerling, O. and Wulf, V., “"Beyond Yes or No" - Extending Access Control in Groupware with Negotation and Awareness,” in: Proceedings of COOP '98, F. Darses and P. Zaraté, Eds., Cannes, France, INRIA, pp. 111-120, 1998.

105. Syri, A., “Tailoring Cooperation Support through Mediators,” in: Proceedings of ECSCW '97, J. A. Hughes, W. Prinz, T. Rodden, and K. Schmidt, Eds., Lancaster, Kluwer, pp. 157-172, 1997.

106. Szyperski, C., *Component Software - Beyond Object-Oriented Programming*. Reading, Massachusetts: Addison-Wesley, 1998.

107. Teege, G., “Users as Composers: Parts and Features as a Basis for Tailorability in CSCW Systems,” *International Journal of Computer Supported Cooperative Work*, Special Issue on Tailorability and Cooperative Systems, pp. 101-122, 2000.

108. Tekinerdogan, B. and Aksit, M., “Adaptability in Object-Oriented Software Development: Workshop Report,” in *Special Issues in Object-Oriented Programming - Workshop Reader of the 10th European Conference on Object-Oriented Programming ECOOP '96*, M. Mühlhäuser, Ed. Linz: dpunkt Verlag, 1996, pp. 7-11.

109. ter Hofte, H., “Working Apart Together - Foundations for Component Groupware”, Ph.D. Thesis, Enschede, the Netherlands: Telematica Instituut, 1998, (available at: http://www.trc.nl/publicaties/1998/wat/wath.pdf).

110. Terveen, L. G. and Murray, L. T., “Helping Users Programm Their Personal Agents,” in: Proceedings of CHI '96, M. J. Tauber, Ed., Vancouver, BC Canada, ACM, pp. 355-361, 1996.

111. Trigg, R. H., “Participatory Design meets the MOP: Accountability in the design of tailorable computer systems,” in: Proceedings of 15th IRIS, G. Bjerknes, T. Bratteig, and K. Kautz, Eds., Oslo, 1992.

112. Vion-Dury, J.-Y., Bellisard, L., and Marangozov, V., “A Component Calculus for Modeling the Olan Configuration Language,” INRIA, Research Report, Technical Report Nr: 3231, August, 1997.

113. Wegner, P., “Why Interaction Is More Powerful Than Algorithms,” *Communications of the ACM*, vol. 40, 5, pp. 80-91, 1997.

114. WFMC, “The Workflow Reference Model,” Workflow Management Coalition, Technical Report Nr: TC00-1003, 19 Januar 1995, 1995.

115. Won, M., “Komponentebasierte Anpaßbarkeit am Beispiel eines Suchtools für Groupware”, Master thesis, *Department of Computer Science III* Bonn: University of Bonn, 1998.

116. Wulf, V., “Konfliktmanagement bei Groupware”, Ph.D. thesis, *Fachbereich Informatik* Dortmund: University of Dortmund, 1995.

117. Wulf, V., Stiemerling, O., and Pfeifer, A., “Tailoring Groupware for Different Scopes of Validity,” *BIT - Behaviour and Information Technology*, 1999.