OBJECT-ORIENTED FRAMEWORKS AND COMPONENTS TO SUPPORT CUSTOMIZATION AND TAILORING IN GROUPWARE

THÈSE N. (à definir)

PRÉSENTÉE AU DÉPARTEMENT DE SYSTÈME DE COMMUNICATION
(MULTIMEDIA COMMUNICATIONS DEPARTEMENT – EURÉCOM)

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES TECHNIQUE

PAR

Jakob Hummes

Diplom-Informatiker, Universität Karlsruhe, Allemagne
de nationalité allemande

composition du jury:

Prof. J. Labetoulle, président du jury
Prof. B. Merialdo, directeur de thèse
J. Davy, corapporteur
Prof. R. Molva, corapporteur
Prof. M. Mühlhäuser, corapporteur
Prof. C. Petitpierre, corapporteur

Lausanne, EPFL
1999

Contents

Chapter 1

Introduction

Computer-supported cooperative work (CSCW) addresses the activity of groups of people. The computer and network environment supports the communication and coordination between members of the group to help them to collaborate and thus to fulfill their tasks cooperatively even if they are geographically dispersed.

Research on computer-supported collaborative work has been interdisciplinary from its beginning. CSCW has emerged as an identifiable research field from a range of disciplines including computer science, artificial intelligence, human-computer interaction, psychology, sociology, organizational theory, and anthropology [Gre88].

As a computer scientist, my objective in this thesis is to focus on questions regarding not only the initial development, but also the evolution and customization of CSCW applications. This thesis delivers solutions, which enable different user-groups to adapt a CSCW system in different states of its life-cycle.

I specifically address reusability in the design and implementation of software components for CSCW systems, customization of these components by developers and system users, and tailoring of the software of running CSCW applications by end-users to their needs on demand.

1.1 Groupware

Groupware is a synonym for “CSCW application”. I will use both terms interchangeably throughout this document.

A groupware system is an enabling system, which helps the human users of this system to interact cooperatively. If the users are spatially dispersed, the groupware system supports the communication and cooperation between the participants of a collaborating group. This implies that the groupware system consists of at least one local groupware application per user and that these local applications communicate with each other either directly or indirectly.

With the advent of general networking, the Internet, and affordable personal computers (PCs) and workstations, the infrastructure for distributed groupware is given in most large and medium companies, institutions, and universities. Even a large group of home users may now have Internet access.

Already a wide range of groupware products and prototypes exist: video conferencing systems, application sharing tools, systems for remote education, group decision support systems, group calendars, and workflow systems are examples for CSCW applications.

1.2 Changing work practices

Companies are interested in benefiting from efficiency gains by changing the structure of both internal and external work processes. During the last decades, the organization of work has been radically changing. Traditional hierarchical organized organization are restructuring to build work teams. Work is generally assigned more dynamically to address current needs quickly. Groupware plays an enabling role to implement those business engineering processes [KB95].

The work structure inside companies is also changing due to new business goals, out-sourcing tasks that have been previously processed in-house, acquisitions and merger with other companies. The work structure changes also, because more motivated workers deliver better results, and empowering employees so that they can make decisions within their teams motivate them. To accomplish their tasks within a team, the group members must exchange information with the right persons quickly and coordinate their goals.

Teams are often created around one task. Persons, which are part of one team can also be part of another team, if they are assigned more than one task. To investigate and resolve particular issues, ad hoc teams are composed which spawn formal organizational boundaries [Boc92].

1.3 Requirements to groupware

To be successful, groupware must address the different organizational cultures with their different work practices. At the same time, groupware must be adaptable to new work conditions to support the new work practices after a reorganization. Furthermore, groupware systems must be open for sudden changes and exceptions.

Since groupware affects the way people are collaborating and thus the work conditions, the design and deployment of a groupware system should be coordinated with the actual users. In Scandinavian countries, the labor unions have proposed in agreement with the companies to let the users participate in the design of those systems, which affect the work place. The research discipline "participatory design" has strong influence on the CSCW community [SS97, Mør97c, SKW97].

To accomplish a constant work with the users, system analysts and designers become part of the entire development cycle of a groupware product [Boc92]. Furthermore, since the activity of the group keeps changing, the development life cycle does not end until the product is replaced.

Since the development of complex software from scratch is expensive, companies evaluate existing groupware applications. Unfortunately, existing groupware systems impose also a specific work behavior. This leads to the question, if the work should be organized around the tool, or a new tool must be created to support the work characteristics of the company.

The second case is generally considered as the only reasonable choice, which leads to the requirement that groupware should be adaptable. In this perspective, recent advances in software engineering can be exploited to introduce innovative and effective solution in groupware design.

If groupware is designed and developed from building blocks, these software components can be assembled to build the groupware product, which fits the company's need. When components for groupware become usable in different products, the development costs per component decrease. More-over, the quality of components increase, since they are used and thus tested in different applications.

To be reusable, these software components must conform to a standardized component model in which they are assembled. The components must be customizable to work in different configurations. In such an environment, the developer of the final groupware product for a company is more an integrator of groupware components than a specialized programmer. The integrator must have a clear understanding of the organizational structure of the group activity and works thus as a domain specialist. Tools must support the integrator to concentrate on combining these high-level business objects [ES98] instead of programming low-level code.

Since the work structure in organizations does not remain static, neither the configuration of the installed groupware does. A groupware integrator can change the groupware to reflect these changes. However, also some technically experienced user, so-called power-users, may posses the ability to incorporate small changes in the system, if such modifications are well supported by tools.

Furthermore, some changes need to take effect immediately to support for example special needs of an ad hoc group or to react on exceptions in a workflow. To anticipate such changes, groupware should incorporate tailoring support, which let the user customize the groupware system without leaving the application. By enabling the actual user of the groupware system to tailor it to its needs this "support for customization is support for innovation" [BD95].

Companies and institutions have already invested in various hardware platforms and operating systems. To ease development and deployment of groupware, the groupware components should offer the same interfaces for all deployment platforms. Furthermore, a distributed groupware system may be deployed over several platforms, thus demanding for interoperability across platforms.

1.4 Objectives of this thesis

The objective of this thesis is to design reusable, customizable, and tailorable software components and frameworks for groupware systems. This thesis targets the requirements to offer users from different user categories – component developers, groupware integrators, groupware administrators, power-users, and end-users – the means to customize groupware applications at design-time and at run-time. The level of customization is thereby dependent on the skills of the user category; hence, this thesis provides different customization and tailoring functionalities to support these different levels and skills.

From a technical and conceptual point of view, the demand for reusable and easy adaptable groupware requires that the core groupware functionality is separated from its potential customizations and extensions. This thesis addresses this issue by combining the software engineering concepts of frameworks and components. I present in this document my approach to design groupware frameworks, which capture the core functionality of groupware tasks, and groupware components, which deliver pluggable extensions for these frameworks. Both, groupware frameworks and components are developed to be reusable also in other settings and all implemented frameworks and components can be combined differently to produce various groupware systems.

I have chosen the object-oriented language Java to implement my approaches. I have chosen Java mainly out of three reasons:

1. Java offers with Java Beans a standard component model. Several commercially available integrated development environments offer visual composition editors for Java Beans. Visual editors abstract from the implementation and this thesis propose to use them to offer customization towards the end-users.

2. Java compiles to platform-independent byte-code. A Java program can be deployed on any platform, for which a Java Virtual Machine exists. Platform independence is vital for groupware systems.

3. By using only the standard and widely used Java and Java Beans technologies for their realization, my concepts become applicable and usable by other groupware developers.

The design of my contributions in this thesis are independent from Java. Some, however, are optimized for the Java Beans component model. Some implementations greatly benefit from Java language features, such as introspection, reflection, object serialization, and dynamic code loading.

This thesis applies the contributed approaches in examples for remote education. I have chosen tele-teaching as domain for the proof of concept, since educational settings demand customizable and tailorable groupware to support different course styles. While lectures, exams, and laboratory courses in universities are similar, they also differ in regard to the teacher's style, the class size, the students' prerequisites, the associated courseware, and the available infrastructure.

1.5 Organization of this document

Chapter 2 reviews the state of the art in the domain of software engineering to draw the technical requirements on reusable groupware. This chapter also presents related work in the CSCW domain, which addresses the building of customizable groupware. Finally, it introduces the approach taken in this thesis.

The following chapters present my approaches to deliver design and implementation for customizable and tailorable groupware. Chapter 3 introduces group communication components, which deliver groupware component programmers and groupware integrators the means to distribute events between applications. Chapter 4 shows how component-based groupware applications can be visually customized on different levels by composition techniques. Chapter 5 presents my central approach, which splits the design of a groupware system into an invariant part, which is designed as a framework, and into variant parts, which are implemented as pluggable components for the framework. Chapter 6 delivers the end-user with a solution to customize groupware components at design-time within a standard visual builder tool for Java Beans. Chapter 7 shows my design and implementation, which uses code distribution, to tailor a groupware system by distributing and inserting at run-time new components. Chapter 8 introduces my related approach, which uses the distribution of objects, to extend a running groupware system. Finally, chapter 9 presents my generic design to offer the end-user the ability to change the behavior of a groupware framework at run-time.

Chapter 10 concludes this thesis.

Chapter 2

Relevant Concepts for this thesis

This chapter introduces first the relevant concepts from the area of software engineering. These concepts lay the basis of the design and implementation of my own approaches.

This chapter also presents related work from CSCW researchers. The section about related work focuses on customization and tailoring in general, groupware programming toolkits that support customizations, and groupware that uses component models.

The last sections from this chapter formulate the objective of this thesis and the approach that has been followed in this work.

2.1 Relevant concepts in software engineering

Software engineering tries to optimize the main phases in the software life-cycle: analysis and design, implementation, and maintenance of software. Optimizations include reliability, development time and cost criteria.

2.1.1 Reusability

Software reusability is the key to stay competitive in the Information Technology (IT) market. The idea of reusing code is not new, but with the advent of object-oriented languages reusing code through inheritance was highly touted. Yourdon [You92] sees reusability techniques as one of the major contributions to software productivity and quality in the 1990s, but also warns to focus only on code reuse. He postulates that analysis and design reuse have an even higher impact than code reuse. Yourdon proposes an organizational culture to encourage actual reuse practices. In his famous "No Silver Bullet" article [Bro87], Brooks identifies causes of failing software projects. He sees the biggest challenge in the "inherent properties of this irreducible essence of modern software systems: complexity, conformity, changeability, and invisibility." Another study [Gib94] reveals complexity as the major cause for huge software disasters. In an answer, Cox breaks the problem down to a reuse problem and proposes the (re-)use of software components, since "encapsulated complexity is no longer complexity at all. It's gone, buried forever in somebody else's problem" [Cox95]. This continues Cox' mission to deliver chip-level and card-level components for object-oriented technologies [Cox90].

Although object-oriented development is nowadays wide in use, it has never delivered the proposed level of reuse of code [Kie98]. Object-oriented libraries tend to include many interdependent objects, which makes reuse of small portions difficult if not impossible. Using object-oriented class libraries often imposes that the application developer calls several library objects in the right sequence. Customization of library objects is accomplished by inheritance to add domain-specific methods or to override existing methods with specialized code. However, using inheritance can endanger the correct behavior of a system of objects; composition techniques are safer for the price of managing inter-object relations [Rya97]. Component based programming techniques facilitate composition of components by providing standard means for interconnecting components within a component model.

Reusability also provides a mechanism for prototyping [You92]. Often software firms discuss the design of a new product by showing a non-functional prototype, which implements only the user-interface. Reusing existing components instead, prototypes can be developed that are already functional, but need further customization. Those "real" prototypes can be used to detect functional specification problems in the early stages of the development cycle. Under this observation, it is not surprising that the Software Productivity Consortium (SPC) has adopted Boehms prototype driven spiral software development cycle [Boe88] as the basis for evolutionary development of software systems [Sof96]. The SPC highlights that it has adopted the spiral model especially because it recognizes important concepts such as engineering for reuse and incremental development.

2.1.2 Customization and tailoring

Customization and reusability are highly dependent. Artifacts on all levels of the software development cycle (analysis, design, implementation, testing, maintenance) can only be reused, if they can be customized to the specific needs of the new or evolved software product.

System modifications and extensions which were once strictly in the domain of the programmer are now being shifted into the domain of the end-user. Off-the-shelf applications such as word processors and spreadsheets offer already customization and tailoring facilities. To tailor those applications, Sumner and Stolze propose a participatory evolutionary development, which brings together end-user computing and participatory design [SS97].

Customization, which is done by the end users during run-time is called tailoring. Tailoring is also a form of application evolution, and thus relates very closely to software reuse [Mør97b].

Mørch distinguishes three levels of tailoring [Mør97a]. These levels are classified by the design distance which is experienced by the end-user during tailoring. Generally speaking, with an increasing level the tailoring possibilities for a user increase, but also become more complex.

2.1.3 Frameworks: Towards extensible applications

A framework is a skeleton of cooperating classes that forms a reusable implementation. An application framework defines the overall architecture of the applications that are created by adapting the framework. Framework-based applications are adapted by extending the framework at explicit plug-points also known as "hot spots" [Pre94].

Frameworks are currently successfully employed for general purpose software units, such as graphical user interfaces, system infrastructure, and middleware integration frameworks; also application domain specific frameworks are emerging [FS97].

Frameworks are distinguished into white-box and black-box frameworks [RJ98]. Object-oriented white-box frameworks use inheritance to offer the developer extension facilities. To insert extensions into white-box frameworks the developer must understand the class hierarchy and derive new classes which have to be relinked with the framework. Black-box frameworks use object composition and delegation instead. Black-box frameworks anticipate extensions by defining interfaces and providing hooks to insert new objects.

Applications that can be extended at run-time need hooks like black-box frameworks. Unfortunately, designing frameworks – and especially black-box frameworks – is substantially harder than designing an application. However, the hot spots for a framework can be designed and implemented stepwise by a sequence of generalization transformations [Sch97]. The evolution of a black-box framework by several generalization steps is so typical that a pattern language is proposed to describe these steps [RJ98].

Since applications using a framework must conform to the framework's design and model of collaboration, the framework encourages developers to follow specific design patterns [Joh97]. In the other direction, developers can use design patterns to generalize an object-oriented application into a framework [Sch95].

2.1.4 Component Technology

In the field of software engineering, component based software development is seen as a major factor to facilitate reuse. Components can be purchased from third party vendors, customized and assembled within a component model. Examples for major component models are Microsoft's Component Object Model (COM) [Rog97] and SUN's component model Java Beans [Ham97] for Java. The component technology is predicted to acquire a significantly increasing importance [Kie98]. Furthermore distributed component platforms are emerging, which allow interaction between components across system boundaries [KA98].

A component is an independent "unit of software that encapsulates its design and implementation and offers interfaces to the outside, by which it may be composed with other components to form a larger whole" [DW98]. Frameworks provide a reusable context for components [Joh97]. Components become most powerful within black-box frameworks, where they can be used to extend these frameworks at defined plug-points.

The recursive compositions of components to form larger components structure component-based applications into layers. Component models allow thus to reason on all levels of compositions of such an application [SC98]. Visual composition tools for a component model add a high level of abstraction to those compositions, allowing even end-users to make changes [Wei97]. Component models and visual builder tools are thus enabling technologies for customization.

2.1.5 Design Patterns

Design patterns help one to reason about recurring design problems. Object-oriented design patterns describe "communicating objects and classes that are customized to solve a general design problem in a particular context" [GHJV94]. Patterns abstract from the used programming language and provide a basis for reusable design building blocks: "Design patterns are the micro-architectural elements of frameworks" [Joh97].

Design patterns are surprisingly useful to detect the hot-spots in an application design and to transform it into a domain-specific framework design [Sch95]. Actually, the idea of hot spots was first introduced as a meta-pattern for framework design [Pre94]. In the domain of CSCW and user-interface design some patterns are well-known, such as the Model-View-Controller and Presentation-Abstraction-Control patterns [BMR$^+$96]. Syri [Syr97] describes the use of the Mediator pattern to design tailorable cooperation support in CSCW systems.

2.2 Related work in CSCW

This section introduces related work in the domain of CSCW research regarding the key aspects for this thesis: reuse, customization, and tailoring.

2.2.1 Demand for adaptability

Adaptability as a basic requirement for CSCW applications was identified already in the early stages of CSCW research. Office information systems, today known as workflow systems, incorporated in the 1980's features to change the order of activities based on conditions. To offer the end-user this flexibility, very-high-level languages were researched [EN88]. In essence, workflow systems today use the same principle, although most workflow systems are programmed visually.

Office automation and workflow systems are contrasted by other CSCW research that focuses more on the enabling technology for cooperation. Adaptability within this broader spectrum mostly stems from the need to build prototypes early in the development phase to let the end-users experience with the system and include their feedback in the further design [SKW97]. The requirement of an early participation of the end-users within the design and development process of groupware has been also demanded by large labor unions in Scandinavian countries [SS97, Mør97c]. To enable rapid prototyping, reusability is a key concern. To test a prototype in different environments, the system must be easy adaptable.

Reusability and flexibility in CSCW systems are addressed by new languages, which can be used by the end-users such as in workflow systems [EN88], but also on a lower level by groupware toolkits for the system developer, such as GroupKit [RG96a] and Prospero [Dou96]. These toolkit approaches, however, do not support directly adaptations by end-users; instead, they offer programming constructs in form of libraries.

The high-level languages approach, on the other hand, often lacks the expressiveness or performance of general purpose languages: those languages can be embedded within the CSCW system, but are not suitable for developers to program groupware [Mør97c]. Having separated toolkits and languages for the developer and the end-user implies that in those CSCW systems adaptation by the users is limited and – more crucial – the level of adaptation is given by the system developers at design-time.

In most CSCW systems, local application software and group software are closely coupled [Gre88]. So, software components for cooperative work must easily integrate with other applications.

Dourish [Dou95] summarizes the following concepts for customization of groupware: flexibility, by providing generic, reusable objects and behaviors; parameterizability, by offering a range of alternative behaviors that users can select; integrability, by linking with other applications in the environment; tailorability, by allowing users to make changes to the system itself. An adaptive and evolving system should support the tailoring and the sharing of adaptions. Such a system should keep core functionality separately from the tailorings and adaptions to support their potential transportation. Dourish defines coadaption as the mutual evolution of systems and work practices, which can be gained through tailoring for example. The idea behind coadaption and customization concepts is that the system design and development does not end with its delivery, which leads to the term of "evolving systems".

2.2.2 Toolkits and libraries for CSCW

On the implementation level, various toolkits and libraries are proposed to support the creation of groupware systems. Here, I introduce representative groupware toolkits and resume their key concepts.

Rendezvous

Rendezvous was one of the first toolkits devoted to groupware development. The project has meanwhile ceased to exist. Nevertheless, the Rendezvous architecture is interesting on the conceptual level.

The Rendezvous architecture [HBP$^+$93] introduces the concept of separating the information common to all users from the user's view and to integrated this concept to its groupware development language. The name "Abstraction-Link-View" (ALV) for this concept is derived from the object roles; this principle influenced most of the toolkits developed later, e.g. GroupKit [RG96a] and Egret [Joh96a].

An abstraction is the centralized entity in Rendezvous that collects information common to all users in a single place. Each user has her or his own view (or views) to the abstraction, i.e. to the common information. A link connects the abstraction with a view. A link is an object, which contains constrains between the variables in the abstraction and its respective view to maintain consistency. It ensures also structural consistency by assuring that each object in a view has a corresponding object in the abstraction.

GroupKit

GroupKit is a toolkit to build real time groupware [RG96a, RGJ96]. GroupKit offers a run-time environment and an API; groupware applications are written in the scripting language Tcl [Ous94] and use the GroupKit API to distribute group related information. The run-time consists of the user-written distributed applications, which are bound via a registrar client to a well-known registrar server that stores all conference related information. Applications built on top of GroupKit are called conferences, because their group mechanisms act synchronously. A session consists of an ongoing conference; states between conferences are not saved [Joh96b].

Tcl and Tk allow in conjunction with GroupKit rapid development of groupware prototypes. However, for complex projects, Tcl is not well suited, since it lacks support for often needed programming constructs (such as arrays and linked lists) [Sta94].

Prospero

Prospero [Dou96] is an approach to deliver a flexible toolkit for CSCW systems. Prospero use the open implementation approach [Kic96] to change the cooperative behavior within Prospero frameworks. Prospero implements possibilities to "reach in" the implementation and to apply changes by using reflection capabilities of a meta object protocol defined in the Common Lisp Object System. The invented reflective model is object-oriented and allows customizations through inheritance and overloading [Dou95].

For this thesis, Prospero's approach is from interest, since it uses special programming language features (reflection) to support customization in groupware systems. However, the offered customization features are intended to be used by groupware developers; end-users do not have the necessary skills to use them.

2.2.3 Frameworks and platforms for CSCW

Egret

Egret has a multi-client, multi-server, and multi-agent architecture, which offers a framework for CSCW applications [Joh96a]. Egret supports particularly the development of (hyper-)text based collaborative applications; the clients and agents are extensions to the XEmacs editor.

Egret's design philosophy supports applications, which "require the collection, manipulation and propagation of information about *state of collaboration*" [Joh96a]. This includes information about the current state, the history, and the process of the collaboration. Agents in Egret are viewed as first-class users; they are used for example to trigger events from the outside or to collect and display periodically information.

Multiple projects [WJ94] have shown that Egret is well suited to implement shared editors, learning environments based on hyper-text, or textual based MUDs.

Clock

Clock is a declarative language to support the development of interactive applications. Since it also supports distributed multi-media applications, it fits well the necessities of a groupware development language. The Clock-Work environment supports the developer with a visual environment to build Clock applications in an object-oriented way [GMU96, Gra95]. The visual programming environment can be offered as graphical user-interface so that the user is able to customize an application.

The visual environment of ClockWork is from special interest for this thesis. Clock is however not available for the general public. It is still a research prototype.

2.2.4 CSCW applications that use components

Oval

Oval is a “radically tailorable tool for cooperative work” [MLF95]. It applies to asynchronous groupware based upon message exchanging, e.g. email systems. It is said to be radically tailorable, since very large changes by the end-user are possible without requiring real programming. The name “Oval” stands for objects, views, agents, and links: its building blocks.

The objects are semistructured representations of things in the real world. They are semistructured in the sense that the filled-in information does not necessarily comply to specific types. The views are customizable by the end-user; they summarize collections of objects. Rule-based agents are used to perform active tasks without requiring the direct attention of the users. They are triggered by events, such as arriving messages or time-outs. Links represent relationships between objects. These hypertext links can be used for manual navigation between objects, but also for further processing by agents.

Oval's component model is very simple and supports only the four described component types. However, these components are optimized for tailoring; which makes this approach interesting for this thesis.

TeamWave

TeamWave [RG97] defines and uses an own Component Model for CSCW applications. TeamWave is originated from TeamRooms [RG96b]. TeamWave is based on GroupKit. TeamWave allows the users to insert their own components within the running groupware system to extend its functionality.

TeamWave relies on the distribution of Tcl code to insert dynamically new components within all distributed applications in a conference. The components are realized as Tcl applications, which are inserted and executed by slave interpreters in the local groupware applications.

2.3 Approach of this thesis

Before this section will introduce my approach, I define some terms, which will be used throughout this thesis.

2.3.1 Definitions

This section defines some terms that will be consistently used throughout this thesis.

User group

During the life-cycle of a groupware product different groups of users may develop, adapt and use the application or its components. The following groups are identified:

- *end-users*: End-users work with the groupware application to solve one or more tasks. The skills of end-users include an understanding of the business process, at least that portion of the business process in which they are directly involved. End-users are not assumed to have knowledge in programming. However, they may adapt the groupware system behavior, if the system supports end user tailoring.

- *power-users*: Power-users also work regularly with the groupware system. Their skills differ from end-users in thus far that they oversee the whole business process and are capable to modify the groupware system by using different tools. Power-users often act as translators [Mac90]: users that tailor an application for other end-users.

- *groupware administrators*: Groupware administrators are the system administrators for a groupware system. The role of administering a CSCW application is often played by power-users. In this thesis they are distinguished only to denote some more skills. An administrator does not need to use the administered groupware regularly, but is able to set-up new applications and to edit configuration files. A groupware administrator has also the skills to compose new applications from existing large-grained components and to customize such components.

- *groupware integrators*: Groupware integrators design and compose new groupware applications. Integrators are often consultants, who do not work within the organization, which installs eventually the groupware. Integrators must be able to analyze the needs of the organization and need to know the market of groupware products and components. Integrators have domain-specific knowledge about the business processes. While they need not to be experienced programmers, they can assemble components within integrated development environments and write small portions of code.

- *groupware component developers*: Component developers for groupware primarily program basic components that may be assembled to form a new CSCW application. Besides programming skills, they need design practice to shape the components for maximal reuse.

This thesis supports the reuse and adaptation process for each user category. The main focus is on the support for reuse for developers and integrators, and the support for customization and tailoring for power-users.

Adaptation definitions

Groupware can be adapted to changing or similar, yet different, business processes by all user categories, depending on the required skills. The reuse of components by developers is based on object-oriented technologies, such as inheritance and composition. The following definitions characterize adaption processes that can also be performed by end-users and power-users:

- *tailoring*: Tailoring denotes the adaptation process during run-time.
- *customization*: Customization denotes the adaptation process during design-time, if supported by an integrated development environment. Customization includes setting of parameters for configuration, but also the visual composition of components.

The distinction between tailoring and customization is used consistently within this thesis. In literature, these terms are often found in different contexts. Some authors denotes with tailoring all adaptation processes that are made after the installation of an application (see for example [Mac90, Syr97]). Mørch's taxonomy of tailoring on the other hand defines customization as the simplest level of tailoring [Mør97a]; tailoring in his model takes place always at run-time.

2.3.2 Component model: Java Beans

This thesis uses Java as the implementation language. The component model for Java is Java Beans.

The specification for JavaBeans outlines that "a Java Bean is a reusable software component that can be manipulated visually in a builder tool" [Ham97]. Beans are self-descriptive Java classes that follow design patterns that let builder tools or applications introspect a bean. Properties reflect the accessible state of a bean. The Java Beans component model uses an event mechanism to interconnect the beans. A bean sends an event to all beans that have registered their interest in that event. The standard distinguishes two extraordinary states in the life-cycle of a bean: A bean can be manipulated in an integrated development environment at design-time or behave like an ordinary object during run-time.

Properties and events can be manipulated within visual builder tools. The Java Beans standard offers additional associated classes for each bean, which contain meta-information about the bean including special customizers and property editors to support a more intuitive interaction with the developer.

The component-based approach together with visual integrated development environments (IDEs) directly support my goal to be able to customize an existing application at design-time and to be able to build new similar applications by reusing the components. Beans with associated customizers allow even non-programmers to customize applications in an intuitive way. The easy grasp is achieved by the use of graphical and form-based editors within the IDEs.

2.3.3 Approach

Groupware tends to be more complex than stand-alone applications. Furthermore, to be successful, groupware must be well integrated within the organizational structure of the group activities and should be adaptable to the users' cooperation modes.

The complexity of groupware demands a heavy investment in development. From the argument that groupware must be adapted to the organizational culture of its users follows that a groupware product can hardly be designed monolithically to fit a large variety of customer's needs. To allow the development of groupware at acceptable costs, reusing and customizing existing designs and code becomes a necessity. Reusable and customizable components for groupware can be used to facilitate building new products rapidly, which can also evolve through end-user tailoring after it has been deployed; this thesis shows an illustrated argumentation for this strategy.

The goal of this thesis is to show how to apply software engineering results about reusability, maintainability, and support for evolution to groupware development. This work uses intentionally only wide-spread accepted technology, such as a general purpose object-oriented language (Java) and its component model. This thesis does not intend to invent a new component model, which is optimized for CSCW applications. Instead, existing technology features are adapted to provide groupware components that can be embedded within real-world CSCW applications.

General groupware components need to be adapted to domain-specific solutions by the application developer. The such created basic groupware applications can be customized to special and evolving cooperative work situations by power-users, or even by end-users.

The task of using existing components to create a new groupware application is a programming job. For this task, the components should be manageable with state-of-the-art programming tools. To be available for a huge group of developers, the components should be based on an accepted general purpose language, which allows platform independent development. The programmer should be supported by off-the-shelf tools to profit from already existing tools (also a form of reuse). This approach also minimizes the learning curve of using the offered groupware components, since each developer can choose her or his preferred tools.

An application can be tailored by using tailoring functionality, which is offered by the application itself, by using scripting languages to write macros for a specific application, and by using tools outside the application. This thesis focuses on the latter, using visual editors to customize and tailor the application; although specialized components may still offer their own tailorability interfaces.

This thesis encapsulates groupware functionality within components that can be assembled and manipulated by different user categories to create and adapt CSCW software. By relying on a standard component model, which is well supported by visual builder tools, not only the system developer is supported, but also the system integrator and power-user. This thesis shows also that components can be designed and implemented in a way that they are even usable by end-users, who do not know how to program; those components are used by drag and drop operations.

2.4 Conclusion

From literature research, I have deduced that groupware systems need to be adaptable to fit the requirements of modern work environments. As the structure of group activities is often changing, groupware that aims to support these activities must be anticipate potential changes.

The goal of this thesis in to research reusability, customization and tailoring for groupware applications. This thesis takes the approach of exploiting software engineering concepts to reach this goal. Component-based development offers the possibility to modularize the support cooperative activities in components. My approach proposes to build new groupware systems by composition of – and thus reusing – groupware components and their customization to domain-specific requirements.

This thesis offers solutions, which enable users from different categories to customize and tailor in all states of the software life-cycle groupware systems and their building blocks: groupware frameworks and groupware components. Solutions do not only address customization and reconfigurations at design-time, but include also the insertion of new functionality and the modification of behavior at run-time.

Chapter 3

Infrastructure framework for groupware developers and integrators: Group communication components

This chapter introduces group communication components, which hide the complexity of distributed systems from the developer. My approach introduces the notion of a group: senders publish events to a chosen group name, receivers subscribe to the same group name to receive these events.

My approach provides CSCW application developers and groupware integrators with easy-to-use components, which hide the complexity of distributed systems. The components, a GroupSender and a GroupReceiver, extend the Java Beans event model [Ham97] to the distributed case. The group communication beans support the manipulation with visual builder tools, thus easing their usage. The beans are the access points to a framework, which reliably distributes Java events from one sender to multiple receivers.

All implementations of the examples in the following chapters use these group communication beans.

3.1 Contribution

- Requirement analysis for group communication components.
- Design and implementation of group communication beans that offer the developer easy-to-use access point to the framework, which distributes Java events.
- Design and implementation of the underlying distributed framework so that the actual used distribution mechanism can be replaced without affecting the implementation of the access points, i.e. the group communication beans.

3.2 Infrastructure components: Group sender and receiver

Since the JavaBeans component model defines only the interaction between beans in the same virtual machine, I have designed group communication beans, which act as access points to distribute an event to a group and to subscribe to a group in order to receive those events.

The design of the group communication beans follow two basic goals: First, the group communication beans must integrate well with the Java Beans model. Second, they must be independent from the underlying distributed system to be able to migrate seamlessly to other distributed middleware implementations.

So, the group communication beans have to fulfill the following requirements:

- explicit access points to send and receive events;
- distribution of arbitrary Java events from senders to receivers;
- independence of the interface of the access points from the actually used underlying distributed system;
- visual customizability within integrated development environments for Java Beans;

- adaption to new event types by inheritance and standard extension, i.e. only the handle routines for the new event types must be added.

Most distributed middleware tries to hide the differences between local and remote object communication. The syntax of sending a message to a remote object is embedded in the normal syntax of the programming language. CORBA [Sie96], the advancement of the remote procedure call (RPC) paradigm [Nel81] towards object-oriented technologies, prominently represents this approach. This approach is sometimes criticized, since it hides not only the complexity from the developer, but it also hides potential different failure semantics [TR88, WWWK94].

In contrast to remote method invocation architectures (e.g. Java RMI, CORBA), the group communication beans are not tied to a specific object, but accept Java events to distribute them. So, they make the distributedness explicit, but do not complicate the programming of the application objects. In fact, to support the collaboration of spatially dispersed people, messages must be exchanged between their local applications; location transparency is often counterproductive for the communication between CSCW applications.

The group communication beans also support directly a one-to-many relationship. An event is passed from a GroupSender to all GroupReceivers for the same group. Thus the notion of a group corresponds to the publisher/subscriber pattern [BMR$^+$96] of the Java Beans event model.

Figure 3.1 shows an overview about the use of the group communication beans. The beans are visible at design-time within integrated development environments (IDE), which support the visual composition of Java Beans. Within an IDE, the group communication beans expose the Java event model visually to the developer for remote event communication. Two beans are necessary: The GroupSender forwards an event to all GroupReceivers, which are configured with the same group name. The design for group communication follows the publisher/subscriber pattern [BMR$^+$96], where the group name corresponds to the subscription. The group name is a property of the beans and so can easily be set within an IDE at design-time or can be exposed as an option in the user-interface to allow changes at run-time.

3.3 Design

The beans for group communication are designed on a higher level than the distributed system actually used for the implementation. The same design

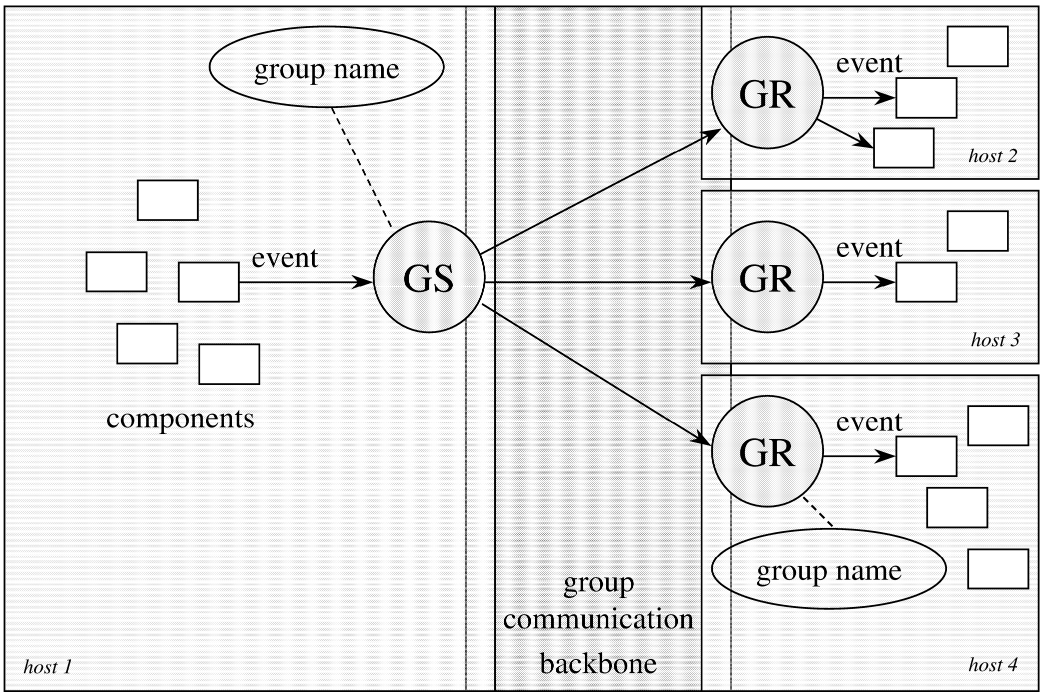


Figure 3.1: The infrastructure for group communication is provided by a GroupSender (GS) and a GroupReceiver (GR) bean.

can be used for transport mechanisms provided by the Java Remote Method Invocation (RMI) API, CORBA object request brokers [MZ95], or reliable multicast implementations as iBus [Maf97]. The current implementation uses the agent-enhanced ORB for Java of Voyager [Obj98]. The design of the beans guarantees that only the interface to the underlying communication system must be developed in order to exchange the distributed system.[1] The higher level beans, the exposed GroupSender and GroupReceiver interfaces, are not affected thus that the communication system can be exchanged without affecting the implementation and configuration of groupware systems that use these group communication beans. No recompilation of the higher levels is needed, since a configuration file determines which actual groupware framework is dynamically loaded and instantiated at run-time, when the groupware applications are initialized.

3.3.1 Design towards the user

Figure 3.2 shows the UML diagram, how the group communication beans represent themselves towards the application programmer. The programmer has only to deal with the classes `GroupSender` and `GroupReceiver` in the independent groupware communication layer. Their implementation is hidden from the application programmer.

To use the groupware communication framework with custom events, the application programmer creates a new sender and a new receiver bean, which are inherited from `GroupSender` and `GroupReceiver` respectively. The only additions to the new classes are handle and fire methods for the custom event (`handleMyEvent` and `fireMyEvent` in the diagram). In the `handleMyEvent` method, the event is passed to the `sendEvent` method of the superclass; the `fireMyEvent` method is invoked automatically by the `GroupReceiver`, whenever a new event from that type is received for the subscribed groups.

The creation of a new pair of group communication beans for new event types is supported by inheriting all methods from the offered beans `GroupSender` and `GroupReceiver`. Most integrated development environments offer wiz-

[1]This statement is proven by the fact that the original implementation used the first version of Voyager [Obj97]; the migration to the current implementation, which uses the second version of Voyager [Obj98], took place without changes on the already developed applications or the need of recompilation. Note that the Voyager updates came along with complex API changes. An earlier, yet incomplete version, took Java RMI to distribute events.

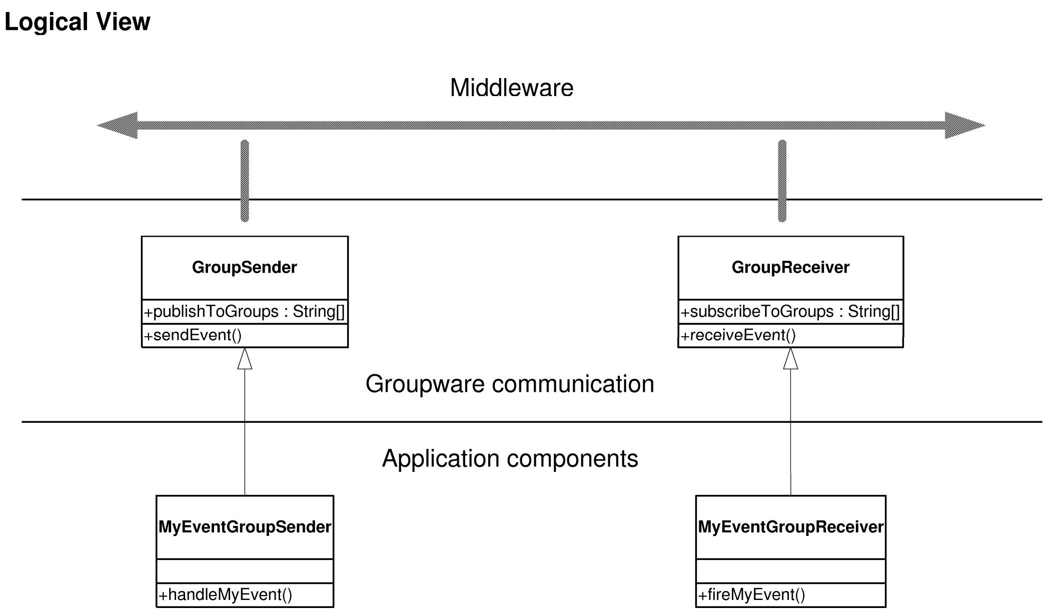


Figure 3.2: The design of the group communication framework as it is seen by an application programmer.

ards to add methods for a new event type. An experienced developer can create a new pair within minutes.

The offered group communication beans can be extended in the same way to handle more than one event. In this case the newly created group communication beans must contain handle and fire methods for all supported event types.

Whenever the developer places a group communication bean in the application, a new instance of this bean is created later in the running groupware application. This is needed to support the individual settings for the group name property. However, the underlying group communication framework is only instantiated once; it is the same instance, which serves the GroupSender and GroupReceiver beans.

3.3.2 Internal design of the group communication framework

The `GroupSender` and `GroupReceiver` classes are Java Beans conform and expose the group communication framework towards the application developer. To be independent from the actually used middleware, which transports the data over the network, they delegate their tasks to classes that implement the corresponding interfaces. The group communication beans do never see the actual classes that are middleware dependent. To accomplish this separation, the internal design uses the Abstract Factory design pattern [GHJV94]. Figure 3.3 depicts the design in UML notation.

At initialization time, the group communication beans instruct the abstract `BackboneFactory` to decide, which implementation to use. The `BackboneFactory` reads the name of the concrete factory from a configuration file, and instantiates the corresponding class. The group communication beans then let the factory create the middleware dependent objects, which implement the interfaces. To highlight the flexible design, figure 3.3 shows not only the actual used classes with the Voyager backbone, but also the classes that symbol the usage of an other middleware `OtherFactory, OtherGS, OtherGR`.

To avoid huge consumption of resources (e.g. memory, threads) the group-communication backbone is realized as Singleton [GHJV94]. At initialization time exactly one instance of the framework is instantiated in each local groupware application, which incorporates the group communication beans. This instance serves both, all GroupSender and all GroupReceiver beans, in the groupware application.

3.4 Applicability and needed skills

The group communication beans are intended to being used by groupware application programmers.

The creation of GroupSenders and GroupReceivers for new event types requires knowledge about Java programming. It involves the creation of a two new classes, which is simplified by using inheritance and adding only the methods, which are needed to handle the new event type. This customization of the group communication beans follows a cookbook: the GroupSender for the new event type must offer a method, which accepts this event and pass it to its superclass. The GroupReceiver must provide `add` and `remove` methods for this event type so that local beans can add themselves as listeners. Most IDEs support the generation of these methods.

To use existing GroupSenders and GroupReceivers for a given event type within a visual builder tool, the developer uses only composition techniques. The developer draws event connections to and from these group communi-

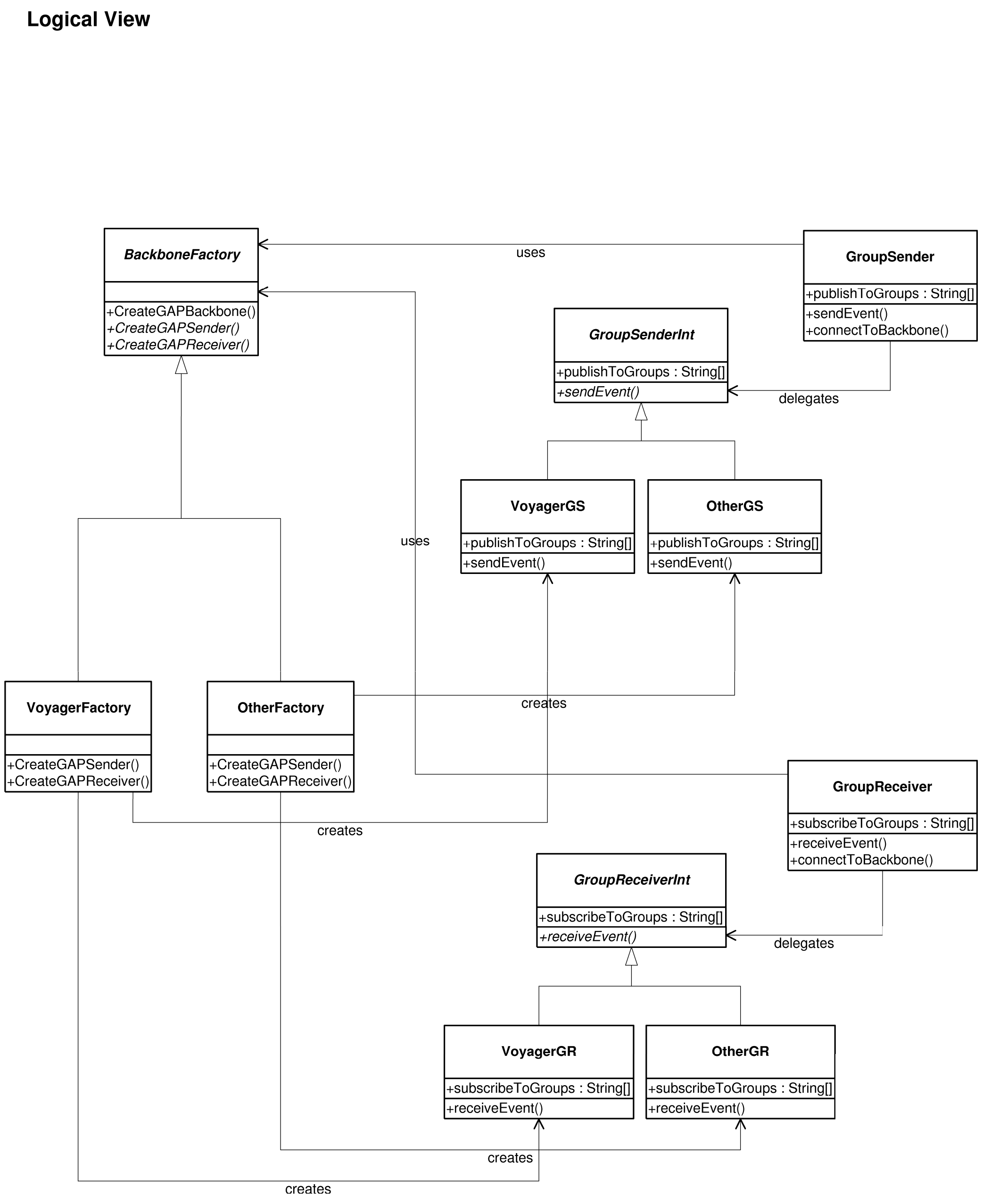


Figure 3.3: Application of the Abstract Factory pattern within the design of the internal classes of the group communication framework.

cation beans. Those composition techniques can be applied by groupware developer and groupware integrators, but also by power-users.

If the group communication beans are designed for more than one event type, the user has to draw connections for all events. If a local component needs always to register for a large event set, drawing all connections can become cumbersome. In this special case, the design phase can be supported more efficiently by implementing methods in the group communication beans, which automatically call the registration methods for all methods for the component. The design and implementation of such methods is analog to the mutual registration methods as introduced in chapter 5.2.

The group communication beans expose the group name as a bean property. Events that are sent by GroupSenders are only received by GroupReceivers for the same event type, if the group name property is set to the same value. Visual builder tools support setting the group name with a property editor. While setting the group name is an easy task, end-users normally should not assign other values, because changing this property has consequences on the interaction between all deployed local groupware applications. This property is typically used by groupware integrators and administrators.

Other beans can also manipulate the group name property at run-time. This is useful to create dynamically sessions (e.g. a chat session) or to create personal receivers, which are often initialized with the login name of the user.

3.5 Examples

Unlike the following chapters, this chapter does not provide a complete example on how to use the group communication beans. Instead, I will give here an overview, where these beans are deployed in the examples of the following chapters.

Chapter 4 introduces tutoring components as examples for component composition techniques. The group communication beans are visible at prominent places, where “help” request are distributed.

In the example of chapter 5, the groupware beans are an integral part of the presented awareness service framework. The group communication beans distribute the events internally within this framework, but are no longer visible without decomposing the framework.

Chapter 7, which introduces the extension of frameworks at run-time, uses the group communication beans to deliver information about the component, which is to be plugged into the framework. This can include the code for the component itself.

In chapter 8, the group communication components distribute whole, potential large, objects, which are encapsulated within an event. The example of a tele-teaching application uses the group communication beans to distribute and receive exams.

3.6 Conclusion

The introduced group communication beans offer uniform access points to send and receive events in a distributed system. The GroupSender bean publishes events to a named group. GroupReceivers that have subscribed to the same group receive these published events.

The group communication beans are optimized for their use in visual composition editors for Java Beans. The user can visually connect events to and from the group communication beans with local components. The group name is exposed as property and can be accessed at design-time with a property editor; it can also be changed dynamically at run-time to support the creation of ad hoc groups.

The design of the group communication components follows the Publish-Subscriber design pattern. The publisher does not know how many subscribers for a given group exist. This design has the advantage that the implementation of the group communication framework can use true multicast to address scalability issues for larger groups. It also fits well the notion of an interest group for groupware applications.

Groupware applications, which need to know all connected users before instantiating a cooperation, can implement a two-way protocol with the group communication beans. First a discover message is sent to the group, which needs to be acknowledged by all subscribed members. The cooperation is then only instantiated if the criteria for the group is met.

In the case that a groupware component wants only to display the current members of a group (e.g. a chat component), the group communication beans should support join and leave messages.

The group communication components assume that the delivery of events is reliable; i.e. that an event, which is published is received by all subscribed applications. The underlying group communication framework should be implemented to support reliability.

The group communication beans are not well suited to support transactions. While a two-phase commit protocol can be implemented with the help of the offered beans, it seems to take unnecessary effort to use them for this reason; especially, since transactions are normally used in a client-server setup, i.e. the group would exist only of exactly one sender and one receiver.

Chapter 4

Reusing components: Decomposition and composition

This chapter shows how component based programming fosters reuse techniques by providing the means of decomposing existing components and recomposition to new components. Through recursive application of composition techniques, component-based applications become customizable on all levels.

This chapter also introduces groupware components, which I define as components that interact with a custom event-based protocol over system boundaries. To be meaningful, each participating groupware application needs locally at least one instance of groupware components. With my approach, groupware components can be customized within visual builder tools in the same way as local components.

This chapter also introduces the use of visual builder tools for Java Beans, which simplify component decomposition and composition.

4.1 Contribution

- Principles of decomposition and composition of components to foster reusability and customizability on all levels of a groupware application.
- Introduction of distributed groupware components, which interact using a cooperation protocol, and application of the composition principles on them.

- Presentation of the composition techniques in an examples of a distributed tutoring system, which uses groupware components.

4.2 Reuse components on all levels

A component is an independent "unit of software that encapsulates its design and implementation and offers interfaces to the outside, by which it may be composed with other components to form a larger whole" [DW98]. By using composition techniques recursively, reuse and customization become available on all levels of a component based application.

With the help of visual builder tools, users can visually compose components to larger components, but also decompose a given component into their parts. One reason why I concentrate on Java Beans as component model is that a variety of integrated development environments (IDE) exist, which support visual composition techniques.

This chapter focuses on the design and implementation of groupware components: components that deliver a solution to a common groupware problem. These groupware components are composed of smaller, specialized components that are normally hidden from the user. However sometimes a similar cooperative task cannot be achieved by using a given groupware component. Instead of implementing this similar task from scratch and the library, the groupware component is decomposed, the necessary changes are made, and the result is composed into a variant of the existing groupware component. This customization technique leaves the original component interface intact, thus enabling the seamless integration within the original groupware application. Figure 4.1 denotes the general customization process by using composition techniques.

After a short review of the Java Beans component model, I will introduce the notion of distributed groupware components.

4.2.1 Java Beans

The specification for JavaBeans outlines that "a Java Bean is a reusable software component that can be manipulated visually in a builder tool" [Ham97]. Beans are self-descriptive Java classes that follow so called design patterns that let builder tools introspect a bean and let a bean be self-descriptive. The

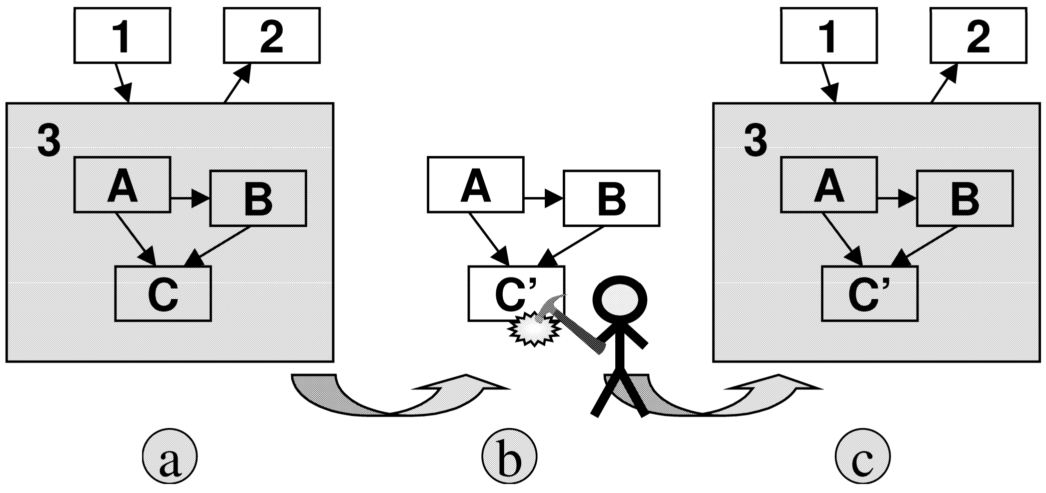


Figure 4.1: A subcomponent (3) of a larger component (a) is decomposed (b) by a user to exchange the internal component (C) with another (C') and eventually recomposed (c) with the same interfaces as the original.

standard distinguishes two extraordinary states in the life-cycle of a bean: A bean can be manipulated in a builder tool at design-time or behave like an ordinary object during run-time.

Properties reflect the accessible state of a bean. Beans can be customized during design-time by altering bean properties. The JavaBeans component model uses an event mechanism to facilitate component compositions. The event-mechanism allows the coupling of state transitions of a bean with an action in another bean. The Beans specification describes how the coupling has to be implemented, thus it can be automated through an integrated development environment (IDE).

Components and especially Java Beans exist in a range of granularity. They encompass the range from simple GUI controls (such as the Java AWT buttons and panels) up to applications (such as spreadsheets or Web browsers). Smaller components can be assembled to form larger components. In the same way larger components can be decomposed into its contained components.

4.2.2 Groupware components

Groupware components solve a particular cooperative problem. To assemble a groupware system, local groupware applications are assembled by using existing groupware frameworks and components.

Groupware systems are symmetric, if all local groupware applications are composed by the same components and support the same interactions with the user; groupware systems are asymmetric, if users play different roles and have different rights. Especially in educational settings users have different roles (e.g. tutor and student). The users interact thus with different local applications, which support their roles.

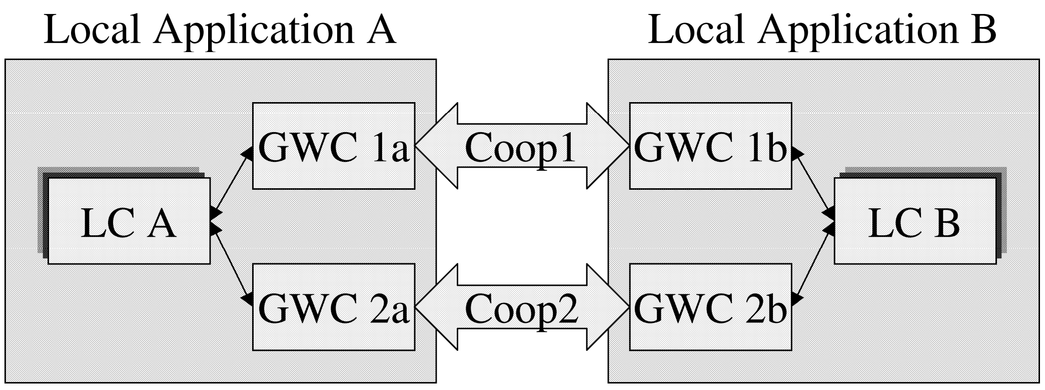


Figure 4.2: A groupware system consists of local groupware applications, which cooperate by groupware components (GWC). In an asymmetric cooperation, the groupware components are different, depending on the role, which is supported by the local application. The groupware components cooperate by their own protocols (Coop1, Coop2). The groupware components interact also with local components (LC) within the local applications.

Figure 4.2 gives a high-level overview about cooperating local applications within a groupware system. The cooperation is actually done by embedded groupware components, which support the different roles of the local applications. Each local application embeds also local components, which interact with the groupware components.

If an groupware system as depicted in figure 4.2 becomes to be changed, two cases can be distinguished. If the cooperation among the local applications remains unchanged, only the local components (e.g. the graphical user interface) needs to be customized. Such a change affects only the local application. If the cooperation itself is changed, all local applications which contains the cooperating groupware components must be updated. This can be done by decomposing each local application and exchanging the groupware component with another groupware component, e.g. GWC 1a with a new groupware component GWC 3a in the local application A and the change of GWC 1b with GWC 3b in the local application B.

From this overview, some requirements for groupware components can be drawn.

- For each different role, a groupware component with specific behavior should exist. This can be accomplished by providing groupware components with selectable roles at design-time (i.e. the instance of the component stores the role) or by providing a family of groupware components for each supported cooperation (i.e. the components are different, but interact with the same cooperation protocol).

- A groupware component shall encapsulate the cooperation protocol and hide it from the user. Even if the protocol is provided by different components, they should be assembled within the groupware component. Thus a user cannot break the cooperation protocol unless the groupware component itself is decomposed and changed.

- For cooperation with local components, a groupware component exposes its interface; in the Java Beans component model that means that each groupware bean exposes methods, which accept events from other components, and fires events, when it wants to signal an observable state change to the outside. The interface should expose the control over the cooperation of the groupware components, but should not provide access to the cooperation protocol itself.

- Groupware components that support similar tasks should implement the same interface to be easily exchangeable. This requirement for groupware components can be easily achieved, if they are assembled by the same developers or producers. Components from different third parties, however, do normally not implement the same interfaces. In such cases – and if exchanging groupware components is anticipated – it is better to build a common adapter, which implements the interface and is taken as the groupware component, than to change the cooperation with the local components each time, when a groupware component is exchanged.

4.3 Applicability and needed skills

An existing component can be decomposed in its contained components. This process can be applied recursively until the component is atomic (or no source code exist).

Although visual builder tools for component based programming support the decomposition and composition of components, this stays a programming job. The use of these techniques thus requires knowledge of the component model, programming skills, and familiarity with the used visual programming environment.

Groupware components cooperate by their cooperation protocol. In an asymmetric cooperation, for each supported role a specialized groupware component exists. The groupware components offer control over their cooperation, but hide the internal cooperation protocol from the user. Thus different cooperation protocols can be implemented by groupware components that offer the same interface. For example, a textual chat component and a component that support audio and video conferencing can be used to support communication between spatially dispersed end users. Both groupware components implement the same interface to set-up and control the communication (e.g. start, stop cooperation or passing a token).

One of the key benefits of assembling CSCW system with groupware components is that they can be easily exchanged to adapt the system to new technological developments (such as the availability of real-time audio) and to adapt the system to changed work situations. However, customization by decomposing components and their recomposition requires strong skills and is especially suitable for radical and deep changes.

Groupware components themselves can be composed of other groupware components. The cooperation protocol of the larger component thus corresponds to the protocols of the internal groupware components plus their coordination.

The skills, which are needed by the user who customizes a groupware system by decomposition techniques, depend highly on the customization task and the supported interfaces by the existing components. On the composition level, it is a relatively easy task to exchange two groupware components that implement the same interface. However, this customization technique requires an understanding of the component-based groupware system, although the exchange can completely be done within a visual builder tool without writing lines of code. In my definition, a power-user possesses the needed skills.

If a groupware component should be replaced by another groupware component that provides a similar cooperation, but a different interface, the user must first develop an adapter. This task is already programming oriented. Those skills are possessed by groupware system integrators and developers.

4.4 Example: Component based tutoring system

To support remote laboratory courses in a university, groupware components for tutoring framework have been developed during this thesis.[1] The CSCW system, which is composed of groupware and local components, offers a solution to find a suitable tutor among the group of tutors. Two groupware components implement the cooperation protocol to contact a tutor; one is placed in the student application, the other in the tutor application. The tutoring components allow students to contact a tutor, when they want assistance. The tutor gives peer-to-peer advice by using cooperation components. The basic tutoring components are extensible by different sets of cooperation components that implement different cooperation modes and use different communication components.

4.4.1 Motivation: The "get help" problem

The "get help" problem manifests itself in tutoring situations, as in laboratory courses or in hot-line situations. In the traditional teaching environment of a classroom, students are assigned exercises during a laboratory course. The students are solving their tasks on computers, while one or more tutors are in the classroom to instruct, supervise and help the students, when they need assistance.

Figure 4.3 shows the general setting in a tutoring situation. One of the students wants assistance and informs a tutor. In classroom situations, the tutor becomes aware about the student's wish, when he sees that the student has raised his hand. The tutor then moves to the student, offers his help and they solve the problem cooperatively.

[1] Arnd Kohrs implemented the first version of these components [Koh97].

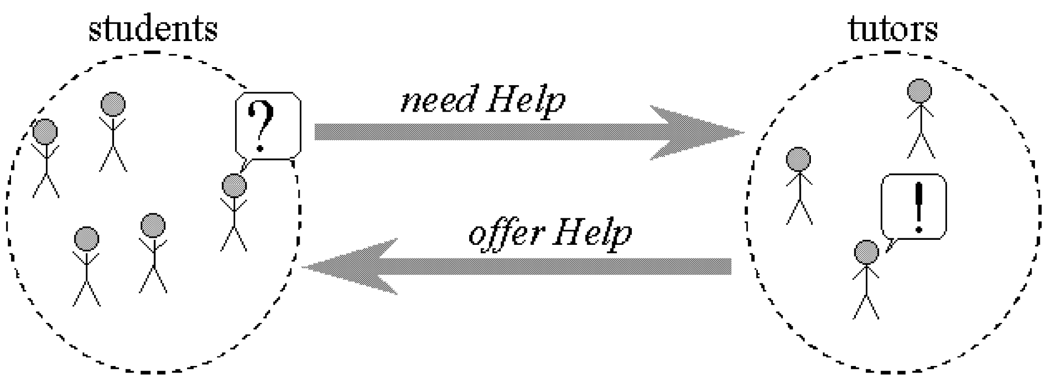


Figure 4.3: General scenario of the "get help" problem.

When this situation is to become computer-supported, some questions become obvious.

- What is known about the student's problem?

  In the classroom scenario a tutor sees only that a student has raised his hand. In a remote laboratory course, where tutors and students are spatially separated, an analogy for the hand raising must be found. However, the student may also describe his problem briefly, before he asks for help or the tutoring system extracts automatically information from the application, which the student currently uses.

- Which tutor is informed about the help request?

  In the classroom scenario all available tutors are aware about the student's request. They also see, when a tutor moves to this student thus that they can concentrate on the other students. A computer-supported lab-course could inform all tutors about a request and inform the others, when one tutor has decided to help this student. Especially if more information about the request is available, the tutoring system could react more sophisticatedly. The system could assign a tutor, who is the best suited to solve that particular problem or try to balance the work between several tutors.

- How can the tutor and the student communicate?

  In the classroom scenario, the communication is evident: The tutor moves to the student; they communicate face-to-face with all possible interactions. In the remote tutoring scenario, this is not possible and the interaction must also be computer supported. The interaction can include symmetric communication forms, as audio, video, and text-based chat. It is also possible to just provide the student with hints in form of a file containing answers for frequently asked questions (FAQ). An additional form of interaction would share the application and use tele-pointers.

These questions and the variety of the possible answers give an example that a monolithic application that wants to support all degrees of freedom is not applicable. Instead it is favorable to customize the tutoring system so that it supports the wanted functionality. This section offers an approach, how the degrees of freedom in the "get help" problem can be preserved by using components, which can be assembled and customized easily to be incorporated in such a tutoring framework.

4.4.2 Design of the tutoring components

The tutoring components are responsible to manage "get help" requests by the students and to deliver them to the appropriate tutors. Different strategies on how to find the appropriate tutors are possible. Eventually, the request are shown to the tutors, who then decide on how to react. Different means are possible to provide the student with help.

The main goal is to isolate the common functionality and to implement it in high-level groupware components. These tutoring components delegate the actual tutor-student cooperation to specialized groupware components, when the tutor has accepted a "get help" request. Different components for different tutor-student cooperation facilities can exist. If new cooperation components has been developed and it becomes feasible to include them (e.g. audio/video components that require hardware, such as microphone and speakers), the tutoring applications are decomposed and the cooperation components are exchanged.

The tutoring components are thus designed to be comprehensible and manageable by groupware integrators, administrators, and power-users, e.g. by tutors. So, the components must conform to the following points:

1. Comprehensible units that are powerful enough to support the given use cases. However, they should not overwhelm the tutor with to many options.

2. Adaptable to the context of different tele-teaching applications. It

might be necessary to pass arbitrary information with a help request or its answer to support different cooperation forms.

3. Delegation of the actual used tutor-student cooperation to specialized groupware components. To use a different pair of cooperation components, the applications, which contain the tutoring components and their interaction with the cooperation components, are decomposed and the cooperation components are replaced.

4. Support for customization. Tutors, groupware system administrators or integrators use visual builder tools to customize the tutoring system. This requires that the tutoring applications are decomposable and the visual representation of the contained components and their interactions are comprehensible. This includes especially that the tutoring applications themselves do not contain too much components on the same level. The level after the first decomposition should contain only high-level components and their events (high-level in the sense: components, which have a similar abstraction and complexity).

The tutoring groupware components interact with the tutoring protocol. The tutoring protocol is defined by the events, which are passed between the distributed tutoring components.

The tutoring component, which resides in the student application, is named `StudentBean`, its counterpart in the tutor application `TutorBean`.

The following events are sufficient to implement a distributed tutoring system, which also offers the needed degrees of freedom to support various customizations, such as support events which can trigger different cooperation components. For this reason the events may carry additional arbitrary information.

NeedTutor: The `NeedTutor` event is sent from the `StudentBean` to the `TutorBean` (or to the trader, if configured). It carries information to identify the student (e.g. the name), a call-back address to set up the cooperation, and an arbitrary object that describes the student's problem. The description can be supplied by the student directly or derived by the state of the student's tele-teaching application. It is freely configurable.

OfferTutoring: The `OfferTutoring` event is the answer to a `NeedTutor` event and is sent from a `TutorBean` to the requesting `StudentBean`. It contains information about the identity of the tutor (e.g. the name) and the help method. The help method is used to lance the facility of cooperation support, which can include a textual chat, audio/video conference, or any other method. It is freely configurable.

Other events: A `StudentBean` can fire a `Cancel` event to cancel a priorly issued `NeedTutor` event. The `Finish` event can be used by both, the `StudentBean` and the `TutorBean`, to finish a cooperation, after the problem has been solved.

4.4.3 Tutoring application examples

This section shows examples, how IBM's Visual Age for Java, which is a visual builder tool for Java Beans, represents the compositions of the tutoring applications, which use the introduced tutoring and cooperation beans.

The tutor application could be assembled as shown in figure 4.4, the student application visually assembled as in figure 4.5.

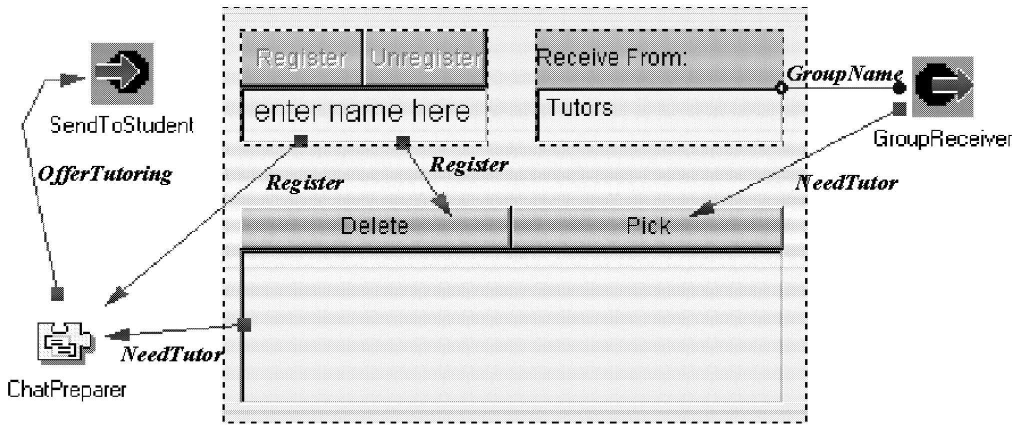


Figure 4.4: Decomposed tutor application with chat component.

Both screenshots contain visible, i.e. graphical user-interface, beans and invisible beans that are only visible in the builder tool at design-time. The beans named `GroupSender` and `SendToStudent`, are customized `GroupSender` beans; the `GroupSender` of the screenshot handles `NeedTutor` events, `SendToStudent` handles `OfferTutoring` events. `PersonalReceiver` and `GroupReceiver` are customized `GroupReceiver` beans; `PersonalReceiver` emits the incoming `OfferTutoring` events, the `GroupReceiver` emits `NeedTutor` events.

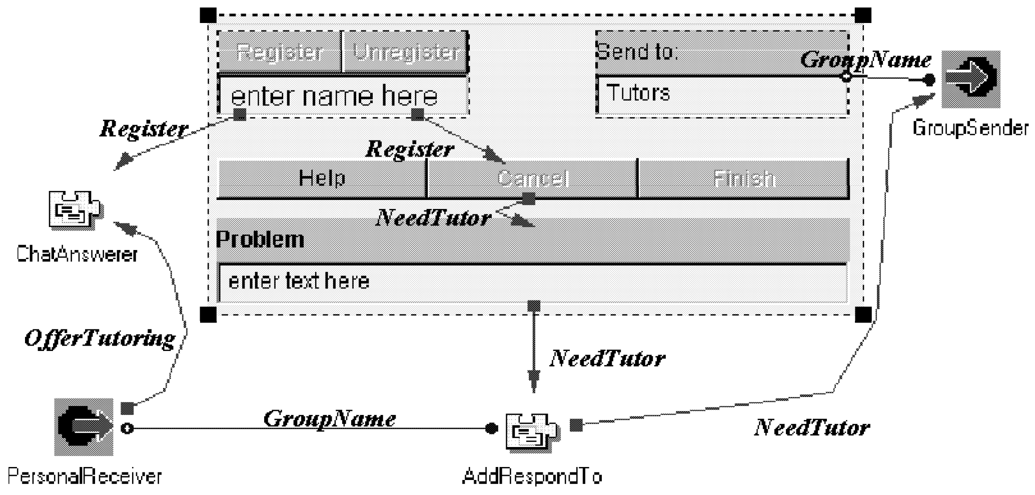


Figure 4.5: Decomposed student application with chat component.

In these examples the `NeedTutor` event is sent to a group, which can be reconfigured at run-time, and defaults to the name "Tutors". The bean for registration is reused in both components.

When a tutor picks a request for help, the `NeedTutor` event is forwarded to a `ChatPreparer` bean, which issues a `OfferTutoring` event back to the requesting student. The `ChatPreparer` in the tutor application and the `ChatAnswerer` in the student application manage the in this example configured cooperation facility: a (here not shown) chat bean for a text-based conference. To change the cooperation mechanism only these beans must be exchanged. During the thesis additional cooperation beans were implemented, such as a frequently-asked-questions (FAQ) components and beans that set-up and support real-time audio and video conferencing.

4.5 Conclusion

Visual builder tools support well the component based reuse technique of decomposition of a larger component and the recomposition of those smaller components into a similar larger component with a similar, yet different behavior. This technique enables developers to focus on the right level of the implementation, since not decomposed components hide all their complexity.

My approach introduces concepts to design and implement the stated requirements for groupware components. The approach benefits from the existence of third-party visual builder tools for Java Beans, since it does not change the Java Beans component model. Thus, users can use these tools to customize groupware components.

Still, this customization activity requires a good knowledge with the used builder tool and also of the decomposed component and its contained components with their interactions. When the experience with customizing large components, such as the tutoring components, matures, often a redesign into object-oriented frameworks happens. The next chapter discusses groupware frameworks and shows a strategy to evolve the here presented tutoring components into a more general tutoring framework.

Chapter 5

Customization by extending groupware frameworks with components

Chapter 4 has introduced composition techniques for component-based groupware applications to foster reuse and to offer customizability on all levels.

This chapter promotes the design of groupware frameworks which anticipates change. The principle is to design the invariant parts as groupware framework and to offer plug-points to extend the framework with specialized components. This chapter introduces the theoretical foundation for this design paradigm.

In this chapter, I apply this paradigm to show a Java Beans compliant design of framework plug-points for groupware frameworks. As example, I present a redesign of the tutoring application, which has been introduced in the previous chapter 4.4. This redesign encapsulates the core functionality to contact a tutor within the framework and offers plug-points to insert specialized groupware components, which handle the peer-to-peer cooperation between the accepting tutor and the requesting student.

5.1 Contribution

- From literature adapted general design principle for customization: Invariant framework, which is extended by domain-specific components.

- Introduction of a Java Beans compliant design pattern for plug-points, which define the extension points of a black-box framework for specialized components.

- Design of groupware frameworks, which offer plug-points for groupware components, which interact with their proprietary cooperation protocol.

- Application of the introduced design principles to redesign the tutoring example of chapter 4.4 as tutoring framework.

5.2 Design for customization: Framework + components

In this section, I will give a short introduction in frameworks as found in the literature to define the technical terms, which are used throughout this thesis. Then, this section introduces my design pattern for the extension points of Java Beans based frameworks and finally apply this framework approach to distributed groupware frameworks, which are extended by groupware components.

The general idea is to design the static parts of groupware applications as frameworks and to anticipate change by designing the changeable parts as components, which can be plugged into the groupware framework.

5.2.1 Overview about framework design

Components, which encapsulate collaborating smaller components to offer common behavior, are easy-to-handle black boxes that communicate with other components in the application only over defined interfaces. This is wanted until a developer needs to change a certain behavior within the component. Therefore, the component must first be decomposed into its subcomponents and then re-composed to form the new component. A better design anticipates the potential changes and offers plug-points [DW98], also called “hot spots” [Pre94]. Plug-points offer an interface of the framework to the outside, where other components can be plugged in to provide a customized behavior. By providing plug-points a component can be seen as an application framework.

Frameworks are commonly distinguished into white-box frameworks and black-box frameworks [FS97] dependent on how they are extended. White-box frameworks rely on inheritance, black-box frameworks on polymorphic composition.

Inheritance as extension mechanism results in strong coupling between the framework and its extending classes; it is static and cannot be easily changed at run-time. Inheritance, on the other side, offers the developer to override all visible methods of the super-class, thus extending the class even if these methods were not anticipated to be changed [RJ98].

Black-box frameworks, on the other hand, use polymorphic composition as extension technique. To apply composition, the framework developer must anticipate the potential changes and design special plug-points. Compositions can be changed at run-time. Through composition the framework and its extending classes are decoupled; any class, which fits the interface can extend the framework.

5.2.2 Framework design for Java Beans

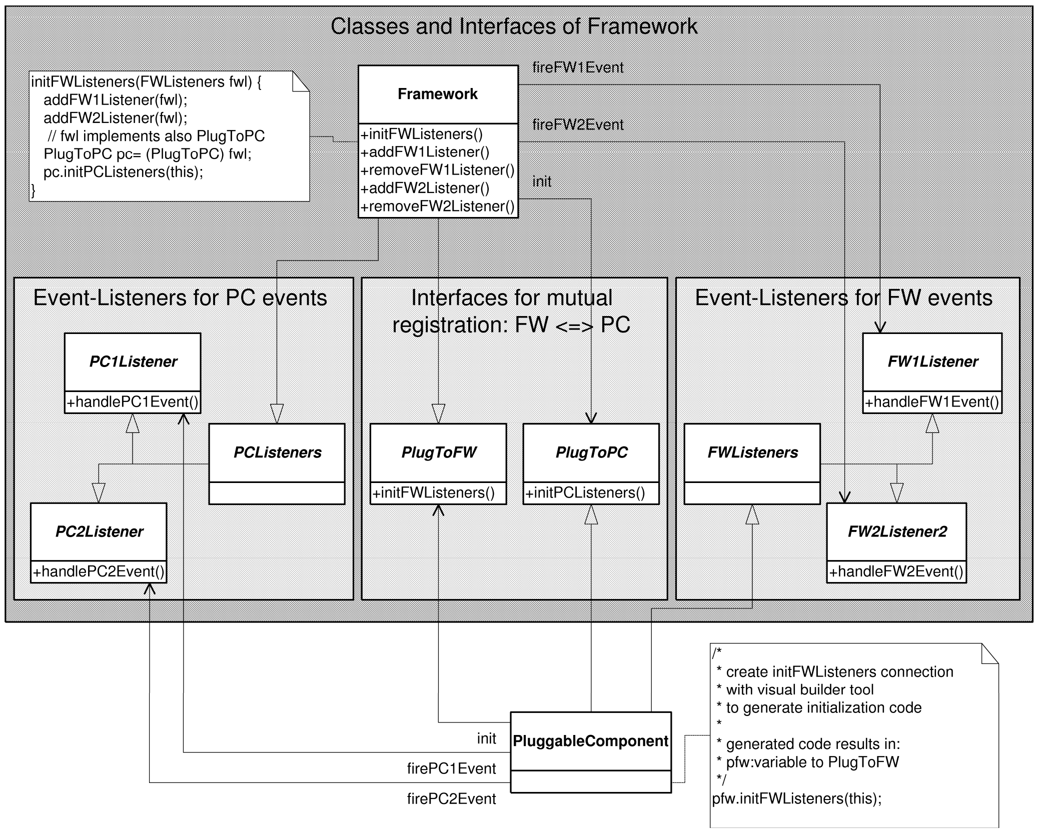
Component models, such as Java Beans, rely mostly on composition techniques. Components interact via an event model. In Java Beans, a bean receives an event only, if it has been registered as listener for this event type by the emitting bean.

Frameworks, which use beans as extensions, are thus designed as black-box frameworks. The plug-points are realized as interfaces, which define the event sets for the protocol between the framework and its extending components; i.e. the interface contains the necessary methods to add and remove the listeners for the event types. Since the Java Beans event model is designed for multicast events, this approach automatically allows the insertion of multiple components for each plug-point.

Visual builder tools that support the Java Beans specification, such as IBM's Visual Age for Java, generate at design-time the method calls to add a bean as a listener for a specific event, when the event is drawn visually. Thus builder tools for Java Beans support visually the extension of frameworks with components.

If the framework and its extending component communicate with many events, drawing the connection within a visual builder tool becomes awkward. Furthermore, if the correct functioning depends on the correct wiring of all events, the developer has to check manually that all event connections are drawn.

To prevent from potential failures that results from forgetting event connections, but also to offer the developer an easy way to extend a framework, I propose the design of plug-points as depicted in figure 5.1.



```
initFWListeners(FWListeners fwl) {
  addFW1Listener(fwl);
  addFW2Listener(fwl);
   // fwl implements also PlugToPC
  PlugToPC pc= (PlugToPC) fwl;
  pc.initPCListeners(this);
}
```

```
/*
 * create initFWListeners connection
 * with visual builder tool
 * to generate initialization code
 *
 * generated code results in:
 * pfw:variable to PlugToFW
 */
pfw.initFWListeners(this);
```

Figure 5.1: Design of a plug-point in a framework.

Figure 5.1 shows the static structure of my approach to define rigid plug-points in a framework. They are rigid, because the pluggable component must implement all event listeners for potential events from the framework and the framework must implement all defined event listeners for events from the component.[1] Additionally, the framework and the extending component must each implement a interface for mutual registration.

[1]Chapter 9 introduces a design, which relaxes this rigid approach and allows the insertion of components, whose event set is discovered at run-time.

The key benefit from this design is that the user, who extends the framework, needs to call only one method of the framework, which connects then automatically the event sources with their listeners; in a visual builder tool the user draws exactly one connection.

In figure 5.1, the gray boxes contain the interfaces for the plug-point. The listener interfaces for all events, which are sent from the **PluggableComponent** to the **Framework** are summarized by the **PCListeners** interface, which inherits the definition of the event handler methods from the listener interfaces for these events (left gray box). The **Framework** must implement these interfaces to be able to receive the events from the component. Similarly, the right gray box contains the listener interfaces, which must be implemented by the **PluggableComponent** to be able to receive events from the framework. Essentially, the listener interfaces define the communication protocol between the framework and its extending component according to the Java Beans event model.

In figure 5.1, the middle gray box contains two interfaces, which define the methods to mutually register the component with the framework. The **Framework** implements the interface **PlugToFW**. **PluggableComponent** calls the method *initFWListeners* of this interface with *this* as parameter. Its implementation in the framework adds this instance of **PluggableComponent** as listener for all framework events. Subsequently, the *initPCListeners* method of the added component is called. Note that the user needs only to draw one connection in the visual builder tool to call the *initFWListeners* method of the framework to mutually register the instance of **PluggableComponent** with the actual instance of **Framework**. The by the tool generated code is executed at initialization time.

5.2.3 Groupware framework

A groupware framework is in my definition an application framework, which provides a groupware service. A groupware framework is distributed. An instance within a local application interact with other distributed instances using a proprietary groupware framework cooperation protocol, which is defined by the distributed events that they exchange; an instance also interact with local components with standard Java Beans events.

Groupware frameworks and groupware components are similar, but differ in their design. While groupware components are ready-to-use components, a groupware framework offers plug-points to be extended. In general, an extension is needed to adapt a groupware framework with domain specific components to be usable in a particular CSCW system.

Like groupware components, groupware framework instances use a cooperation protocol to interact over system boundaries. In the case of asymmetric groupware systems, the protocol can be provided by different implementations of a groupware framework or by providing properties to select the role of the framework instances. On an more abstract level, e.g. the analysis and design level, its is sufficient to note the role that a groupware framework plays; figures 5.2 and 5.3 use this approach.

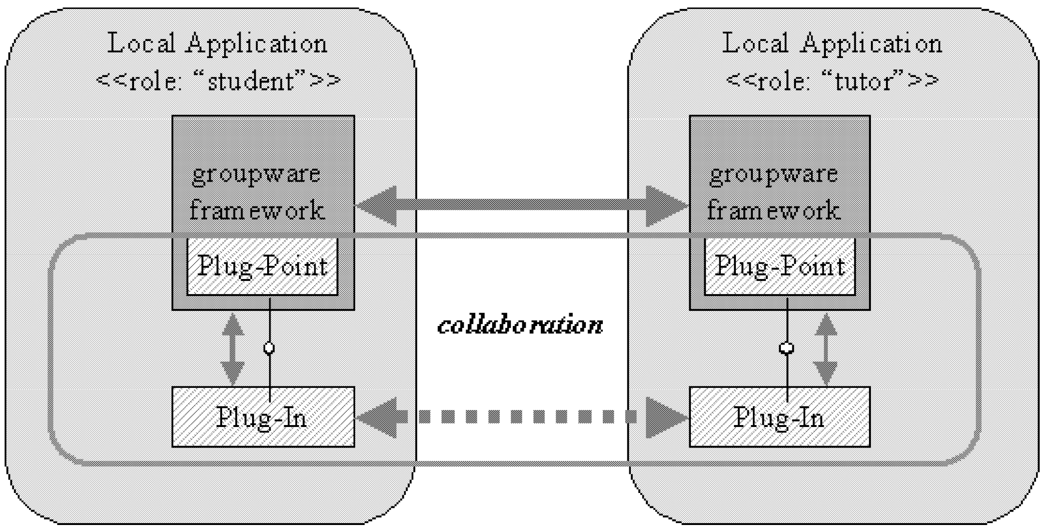


Figure 5.2: Groupware framework extended with groupware components.

The plug-points, which are used to customize a groupware framework, anticipate domain-specific extensions. The design for groupware framework plug-points uses the prior introduced design (see figure 5.1). Plug-points can be defined for local components and for groupware components. Plug-points that assume groupware components as extensions need to be extended by suitable components in all distributed instances in order to support the cooperation protocol of its extending components.

Figure 5.2 gives a general overview about an extension of a groupware framework with groupware components. The figure shows the cooperation protocol between the instances of a groupware framework as well as the cooperation protocol of its extending groupware components. Further, the framework and the components interact over the by the plug-points defined interfaces. This figure assumes an asymmetric setting, with two roles, a student role and a tutor role.

5.3 Applicability and needed skills

Frameworks often evolve from prior object-oriented or component-based design [RJ98]. Groupware components, such as discussed in chapter 4, which are often adapted, are good candidates to evolve into groupware frameworks by factoring out the common parts and designing plug-points for the specific parts.

The design of extensible frameworks is a demanding task for designers of groupware components. However, the use of an existing groupware framework and its extension by domain-specific components is a task, which requires only moderate programming skills and some experience with a visual builder tool. Groupware integrators, administrators, and power-users have the skills to adapt groupware frameworks by composing them with domain-specific components.

5.4 Example: Redesign of Tutoring framework

To illustrate the introduced principles, this section redesigns the groupware components for tutoring from chapter 4.4 into a tutoring framework.

The basic service of the tutoring framework is to deliver contact requests from one person to a group of other persons (e.g. a request from a student to one or more available tutors). The framework offers means to control the contact request, i.e. it allows to cancel a request, to show the request only to a sub-group or a specific person and to react on a request. When a tutor answers a request a specific collaboration tool for peer-to-peer advice is triggered, which handles the collaboration. So, the design of the tutoring framework includes a plug-point for the groupware component, which manages this collaboration.

Figure 5.3 depicts the redesign for the tutoring framework. Two different instances of the tutoring framework exist: One for the requester (e.g. the student), one for the tutoring provider (e.g. the tutor). Both instances of

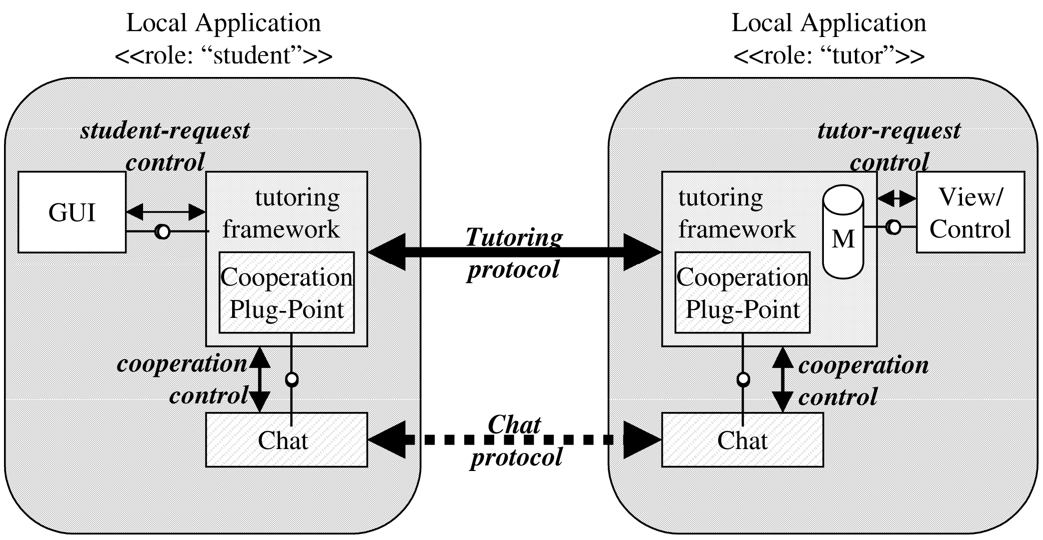


Figure 5.3: Tutoring framework with plug-points to local components (GUI, Model/View) and groupware components (chat).

the framework offer a plug-point for groupware components that provide the actual cooperation between a student and a tutor, when the tutor wants to answer a request. In the figure, a chat component is used as an example.

The protocols between the tutoring framework and the local and groupware components fulfill the requirements as they have been analyzed in chapter 4.4.2.

The interface of the student access point of the framework offers the possibility to request assistance by a tutor and to control own requests. In analogy to the analysis in chapter 4.4.2, the framework accepts the following events from local components (*student-request control*): `NeedTutor`, `Cancel`, and `Finish`.

The plug-point for local components of the tutor access point of the tutoring framework is based on the Model/View/Control design pattern [BMR+96]. The model stores the requests by the students and manages them. The plug-point supports the extension by one or more external view components, which are updated whenever the contents of the model change. The plugged component informs also the framework (and thus the model) about changes, i.e. when a tutor selects a request to answer. In analogy to the analysis in chapter 4.4.2 the framework accepts the following events: `OfferTutoring` and

`Finish`. In contrast to the design of the tutoring component, the framework internally manages all requests by the students within the model.

The tutoring framework offers for both roles (student and tutor) a plug-point, which must be extended by components that handle the actual co-operation between the requester and the tutoring provider. Essentially this interface controls the begin and the end of the cooperation; i.e. the framework can send events to the extending groupware components to initiate a cooperation (`StartCoop`) and to stop a cooperation (`StopCoop`). However, also the cooperation components might signal the end of the cooperation; they use the `Finish` event.

In this asymmetric case of a tutoring framework, both the student and the tutor side must be extended by fitting groupware components. Fitting means that the extending components support the same cooperation protocol and that the components implement the plug-point interfaces of the framework. The cooperation components can also be asymmetric but this is not mandatory. As an example, the chat component in the example figure is the same at both sides and supports a symmetric cooperation.

The tutoring framework does not only distribute events, but is also checks on integrity. This means that an event to cancel a call tutor request is only accepted, if the request is still open. The integrity check includes also that the model reflects at any time the current state of the requests; regardless if one or more tutors guide the requesters. How the framework achieves this integrity is not defined by this high-level design, but a simple implementation would use a central database to hold all requests and to which all tutoring framework instances are connected.

5.5 Conclusion

This chapter proposes to split the design of a cooperative service into two parts. The invariant part is offered by a general groupware framework, which is extended by the part, which implements the domain-specific behavior. Plug-points define the interface between the framework and its extending domain-specific components.

Both groupware frameworks and its extending groupware components may use their own cooperation protocol to communicate among their remote instances. This cooperation protocol is not defined by the plug-point, which only determines the local communication protocol between the framework and the components. The user is responsible to extend a groupware framework with fitting groupware components, i.e. with components that interact with a matching cooperation protocol.

The proposed Java Beans optimized design for plug-points ensures that users can easily extend a groupware framework with visual builder tools during design-time.

Chapter 6

Special customization support

This chapter shows how the customization process can be further supported by specialized components, if the component model foresees such possibilities. Java Beans offers the ability to link so-called Customizers with a bean, which act as wizards to help the customization process.

Visual builder tools execute the Customizers, when a user selects a bean for customization. Customizers offer a graphical user-interface for a bean to guide the user through the creation of a specific instance of a bean. With the help of Customizers, the visual creation of complex instances (e.g. a bean that contain many other beans in a specific instantiation) becomes feasible. In the extreme, end-users can create large nested components together with their customized settings by using only visual "drag and drop" operations together with the support by Customizers.

Customizers can be used to initialize even complex groupware frameworks and components within integrated development environments without requiring programming.

The use of Customizers at design-time to create new bean instances integrates very smoothly with the distribution of objects at run-time. Chapter 8 will introduce this approach.

6.1 Contribution

- Use of Customizers to lower the needed skills for customization of particular components; thus allowing even end-users to customize those components.

- Use of the combination of using a visual builder tool to "drag and drop" beans into a container and customization of these beans by Customizers; thus creation of many new instances, which are encapsulated within a component.
- Applicability of Customizers to provide different initialization states of groupware frameworks and components.
- Presentation of the questionnaire example, where new questions are added by the end-user and their contents are set through Customizers.

6.2 Use of customizers

This section introduces first the concepts of Java Bean Customizers to show then how they actively support customization.

6.2.1 Java Beans Customizers

The Java Beans component model is specifically designed to support the visual manipulation of beans within a builder tool [Ham97]. Chapter 4 has already introduced visual composition techniques to customize a bean. To use those composition techniques, the user needs to be familiar with the visual builder tool and its visual programming paradigm.

The naming conventions of Java Beans allow a builder tool to introspect a bean and thus expose the bean's properties, event sets, and public methods towards the user. Additionally the Java Beans specification [Ham97] supports so called `BeanInfo` classes. Each bean can have an associated `BeanInfo` class, which – if present – must be used instead of introspection by a builder tool to present the bean information. A `BeanInfo` class can link a special `Customizer` for the bean.

To change the properties of a bean, a builder tool analyzes the bean and shows a property editor for the accessible properties. Java already includes property editors for basic types (such as `String`, `int`, etc.). The Java Beans component model does not constrain the type of a property so that any class can be used within a bean as property. User defined classes can accompany a property editor to let the user set the property value within a graphical user-interface.

Property editors support well the independent setting of properties, but they are not tied to a specific bean. Instead the builder tool selects the appropriate property editor solely on the type information of the property. Customizers are wizards that are associated with a particular bean to offer a graphical interface towards the user at design-time in a builder tool. A Customizer optimizes the ease of setting the properties of a particular bean. Since it has special knowledge of the bean, the Customizer can call each visible method of the bean or set instance variables which are not exposed as properties.

Property editors and Customizers are hybrid in the sense that they are Java classes, which are instantiated and run by the builder tool. So, the border between design-time and run-time is fuzzy, since at least these classes are executed at design-time.[1]

6.2.2 Customization support

The user experiences a design distance during customization. The distance is low for setting properties of domain-centered presentation objects (i.e. selecting a new color, but also setting the group name in the communication beans (cfg. chapter 3)); the distance is high, if programming lines of code is involved.

The design distance, which is experienced during customization components at design-time corresponds to the distance during tailoring as defined by Møørch [Møør94, Møør97a, Møør95]. Møørch distinguishes three levels [Møør97a]. The first level allows customization by changing the appearance of presentation objects and to change their attributes. This corresponds to the use of property editors at design-time. The second level allows the integration of new components or commands by composition of existing functionality within the application. Møørch gives small user-written scripts or macros as example. The third level allows the extension of an application by adding newly developed code of a multi-purpose programming language. Generally speaking, with an increasing level the customization possibilities for a user increase, but also become more complex. To overcome the design distance, Møørch [Møør95] proposes to use so called “application units”.

[1]Customizers can also be integrated within an application and then be executed as “normal” Java code at run-time [Rod99].

Application units consist of three parts: a presentation-object, which is the user-interface, a rationale that provides meta-information about the intended use, and the actual implementation.

Although visual builder tools like IBM's Visual Age for Java offer an even higher abstraction than scripting languages and are thus usable by end-users [Wei97], the user needs to learn to use the tool and the visual programming paradigm. A bean, which is accompanied with a Customizer, however can be easily adapted by using only the user-interface of the Customizer.

Java Beans Customizers allow to shift the design distance from the second level to the first level. Customizers can contain a rationale to provide meta-information, but their biggest advantage is to use them as wizards, which automatically change the customized object, when the user has committed her or his choices.

A Customizer works on the implementation. With the help of Customizers, a user can compose a complex bean without the need to learn (visual) programming. The customizer guides the user through the options.

Since builder tools execute Customizers, a Customizer can present itself differently towards users from different user categories. A Customizer may get the information about the user category either by prompting the user or by reading a configuration file at initialization time. Customizers themselves are also beans; by factoring out the test into an own component in which user category a user belongs, this component can be reused for all Customizers, which need different behavior corresponding to the user category.

Customizers and visual composition techniques complement each other very well. Beans that act as containers can be populated by dropping matching beans on their surface within visual builder tools. The user only needs to select the wanted component within the component library and to drop it into the container, where the user opens the Customizer to specialize its content and behavior. Thus, even end-users can assemble and manage large component structures without having programming knowledge.

The composition of large components by dropping beans within a container is a common application of a black-box framework. The container specifies the interfaces, which a component must implement, but may specify also some additional interfaces. When the container with its components is instantiated, it searches the components for the implementation of those additional interfaces and adds their event sets to its listener. The Questionnaire bean in the example section discovers so automatically evaluatable questions and processes them accordingly on return to the professor.

6.3 Applicability and needed skills

Customizers are most useful for two cases: Support of the setting correct values for dependent properties, and support of the customization by users, who are not familiar with the visual builder tool.

Although the user needs to know some basic functionality of the builder tool (such as finding and selecting the bean to customize), knowledge about visual programming is not required. Customization support for users that are not necessarily familiar with the use of a builder tool includes especially support for the following user categories: end-users, power-users, and groupware administrators.

Visual builder tools support with visual composition techniques and Customizers for beans the means to allow end-users the composition of large new beans even if they do not have programming skills.

Customizers, which present themselves accordingly to the user category, do not only support the users depending on their actual skills, but also imposes customization rights to prevent accidental changes. Restricting and granting customization rights for different user categories is seen as a requirement for adaptive CSCW systems [SC98]. So, Customizers allow to provide exactly those customization means which fit the skills of the user.

Within the context of groupware, Customizers are especially useful for the following tasks:

- Setup modifications of the groupware system by the groupware administrator.

- Customization of the local components, which interact with a groupware framework or a groupware component.

- Ad-hoc composition of active compound documents (see example below) and to prepare them for distribution (see chapters 7 and 8).

6.4 Example: Creating a new questionnaire with active components

This example introduces the creation of an exam by an end-user, typically a professor. The exam is an instantiated questionnaire bean, which is populated with specific question bean instances.

The professor uses a visual builder tool for Java Beans to drag and drop question beans from the component library on the questionnaire container. The professor then opens the associated Customizer for each question – a simple double-click within IBM's builder tool Visual Age for Java, which is used in all examples.

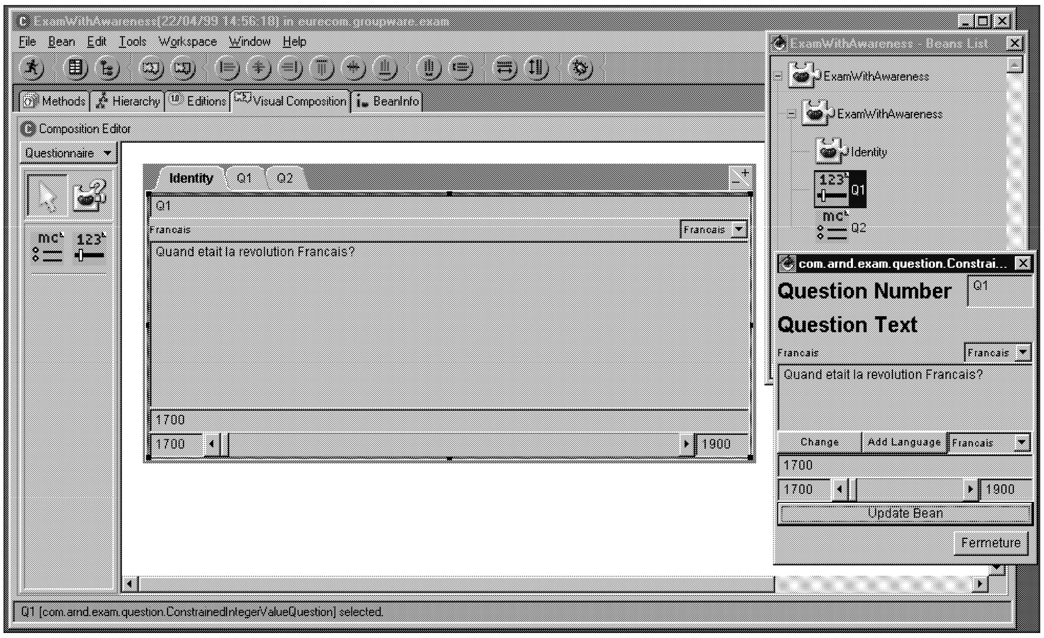
Since Eurécom is an international institute with two official languages (French and English), the question beans are designed to hold more than one language for each question. All users can select at run-time their preferred languages.

Figures 6.1 and 6.2 show two screenshots during customization. Both figures show the questionnaire container within the visual composition editor of Visual Age for Java. Visual Age allow to display beans on a palette (left side); for the Questionnaires example two question beans have been added to the palette: A bean for multiple choice questions (the icon marked with "mc") and a bean for an integer value question, which includes a scrollbar for setting the value (the icon marked with "123"). For easier navigation within the questionnaire, the professor has opened the Visual Age's Beans List (the window under the actual Customizer window in both examples); the currently selected bean is highlighted.

The questionnaire, as depicted in the figures, actually contains two question beans. The professor has prior to these screenshots dragged and dropped the beans from the palette on the questionnaire bean.

Figure 6.1 shows the Customizer for the first question, which is an integer value question. The Customizer (lower right) let the professor choose the language and has a text area to enter the question. The range for the solution for this question is constrained by the numbers, which are entered left and right from the scrollbar. The scrollbar within the Customizer can be used to supply a default value.

Figure 6.1 shows the Customizer for the second question, which is a multiple choice question. As the other question type, the multiple choice question

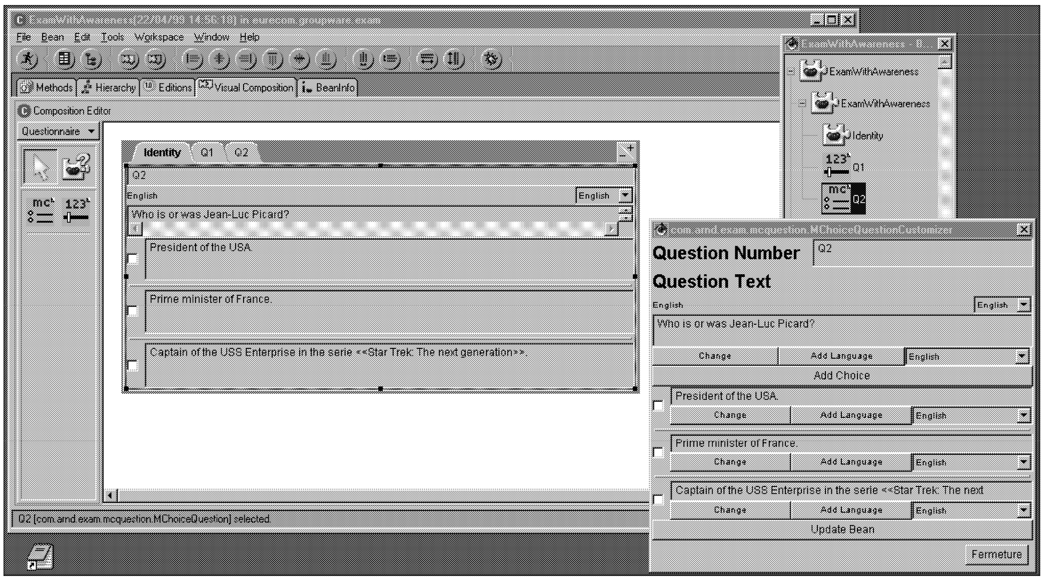

ExamWithAwareness[22/04/99 14:56:18] in eurecom.groupware.exam
File Bean Edit Tools Workspace Window Help
Methods Hierarchy Editions Visual Composition BeanInfo
Composition Editor
Questionnaire
Identity Q1 Q2
ExamWithAwareness - Beans List
ExamWithAwareness
ExamWithAwareness
Identity
Q1
Q2
com.arnd.exam.question.Constrai...
Change
Add Language
Update Bean
Fermeture

Q1
Francais
Quand etait la revolution Francais?
1700
1700 1900

Question Number Q1
Question Text
Francais
Quand etait la revolution Francais?
1700
1700 1900

Q1 [com.arnd.exam.question.ConstrainedIntegerValueQuestion] selected.

Figure 6.1: Integer value question Customizer.


ExamWithAwareness[22/04/99 14:56:18] in eurecom.groupware.exam
File Bean Edit Tools Workspace Window Help
Methods Hierarchy Editions Visual Composition BeanInfo
Composition Editor
Questionnaire
Identity Q1 Q2
ExamWithAwareness - B...
ExamWithAwareness
ExamWithAwareness
Identity
Q1
Q2
com.arnd.exam.mcquestion.MChoiceQuestionCustomizer
Change
Add Language
Add Choice
Update Bean
Fermeture

Q2
English
Who is or was Jean-Luc Picard?
President of the USA.
Prime minister of France.
Captain of the USS Enterprise in the serie <<Star Trek: The next generation>>.

Question Number Q2
Question Text
English
Who is or was Jean-Luc Picard?
President of the USA. English
Prime minister of France. English
Captain of the USS Enterprise in the serie <<Star Trek: The next English

Q2 [com.arnd.exam.mcquestion.MChoiceQuestion] selected.

Figure 6.2: Multiple choice question Customizer.

Customizer supports multiple languages and has a text area for the question itself. The Customizer allows to add an arbitrary number of potential answers, which are displayed with a checkbox. Actually each potential answer itself is a bean; the Customizer thus adds new beans to the multiple choice bean, although the user does not notice these internals.

The changes made through a Customizer are committed, when the user closes the Customizer. The builder tool reflects the changes after commitment or when the user asks for an update of the bean.

Both shown Customizers in the examples intentionally offer user interfaces, which are close to the user interfaces of the customized beans. Thus, the design distance between the customized component and the applied methods (i.e. the Customizer) are lowered. This nearly what-you-see-is-what-you-get (WYSIWYG) fashion follows prominent user interface design examples (such as word processors, or spreadsheets).

6.5 Conclusion

Visual builder tools already offer the means to reason on a higher level (closer to the application domain) on components and their composition than the actual source code. However, such general tools do not have an a priori knowledge about the specifics of a component. The Java Beans component model offer the possibility to further describe a bean in an associated `BeanInfo` class, which is evaluated by the builder tool. A wizard, called Customizer, offers the user a graphical interface to make changes to a bean. Thus the Customizer guides the user through the possible options and prevents from accidental incoherent changes. Furthermore, the user needs not to learn to use the builder tool in detail. Customizers lower thus the design distance, which is experienced by the user.

The presented example uses Customizers to allow end-users the creation of questionnaires. This example becomes part of the example in chapter 8, which distributes questionnaires during a tele-exam from the professor to all students.

Chapter 7

Customization becomes tailoring I: Code distribution

This chapter introduces how the design of extensible groupware frameworks anticipates changes that can be introduced at run-time. The customized code, which is not known at initialization time of the groupware application is later distributed and inserted at plug-points of the framework.

7.1 Contribution

- Introduction of my two-step approach of tailoring: Customization and composition of new components within a visual builder tool at design-time; insertion of the customized components at run-time.
- A general Extensibility design pattern for plug-points, which allows the dynamic insertion of components in groupware frameworks at run-time.
- Relation of the Extensibility pattern with the general plug-point design for Java Beans, which is described in chapter 5.2.2.
- Using code distribution for pluggable components to extend all cooperating framework instances at the same time.
- Discussion of applicability of the Extensibility pattern and of the risks of dynamic code loading.
- Presentation of some examples to highlight possibilities.

7.2 Motivation

Inserting new functionality into a running application is an act of tailoring. Tailoring is recognized in the CSCW literature as the key requirement for a system to adapt to different cooperative contexts [MLF95, TB94]. For Bentley and Dourish [BD95], "support for customization is support for innovation".

This chapter focuses on one important subset of tailoring: the ability to insert new functionality into an application and thus to change the behavior of the system. New functionality can be discovered by an extensible application at initialization time. It is harder to design applications that can be extended at run-time. Even harder is the design of extensibility at run-time in distributed interactive applications, such as synchronous groupware. This chapter presents a general design pattern to solve the latter problem.

To design CSCW applications and groupware frameworks that are tailorable by extension, their hot spots must be discovered in the design phase and then implemented as plug-points. To ease the implementation this chapter introduces a design pattern which can be used to insert those plug-points into a groupware framework. The pattern focuses on the ability to insert new code at run-time that conforms to an interface. By applying this pattern, one thus designs an extensible black-box framework for a specific CSCW problem.

7.3 The two-step approach

The tailoring support for extending a running program is split into two different support-systems, the customization and the insertion support. Figure 7.1 illustrates this approach. In the first step, an end-user uses an off-the-shelf visual builder tool to customize a component at design-time. This component is then, in a second step, inserted into the running distributed CSCW application. Decoupling the customization tool for the components from the actual CSCW application has the following advantages for the developer:

- The product can be earlier delivered, because the tailoring functionality is not built into the product.
- The developer can save resources, because a proprietary tailoring tool needs not to be developed.

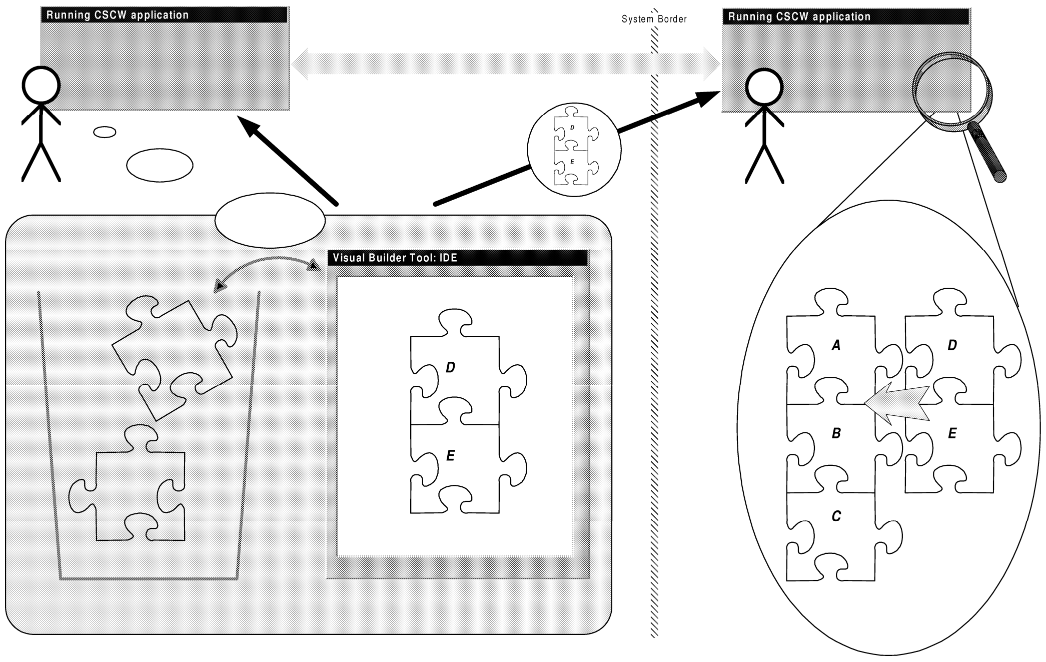


Figure 7.1: My approach uses two steps to tailor a running CSCW application: A user can customize components within an IDE and then import them into the distributed application by sending them to all local instances of this CSCW application.

- General off-the-shelf IDEs are continuously improved by third party vendors.

The end-users profit from the decoupling as well. They can use their favorite builder tool for that component model and do not need to accustom to a new tool for every application.

7.4 Enabling Technologies for Extending CSCW Applications

This section introduces a design pattern, which is used to insert plug-points in the design of applications. Since CSCW applications are inherently distributed, the pattern is accompanied with components that allow the distribution of arbitrary events to a group. By using the event mechanism and encapsulating code within an event, a group communication receiver in the pattern allows the simultaneous extension of synchronous CSCW applications at run-time.

7.4.1 Design Pattern for Extensibility

In a component model, applications are developed by interconnecting and customizing components. The components themselves are composed of other, smaller components. The design pattern for extensibility, which is introduced here, can be encapsulated into one component.

The Extensibility pattern is intended to be used to provide a default behavior, which can be changed at run-time. To change the behavior a new class can be inserted at a plug-point, which can either add new functionality or replace an existing class. The application sees only the specified behavior of a **Proxy** class.

The structural representation of a pattern is given by the relationship between the used classes. Figure 7.2 shows the structure of the Extensibility pattern in the UML notation. This pattern consists of a **Proxy**, which extends the interface of a **Subject** that may be inserted at run-time.[1] Inside

[1] In this pattern, the **Proxy** extends the **Subject** interface to be conform with the Proxy pattern. However, for the framework developer it is only important that the definition of the **Proxy** remains stable, while the developer of the pluggable components (the **Real**

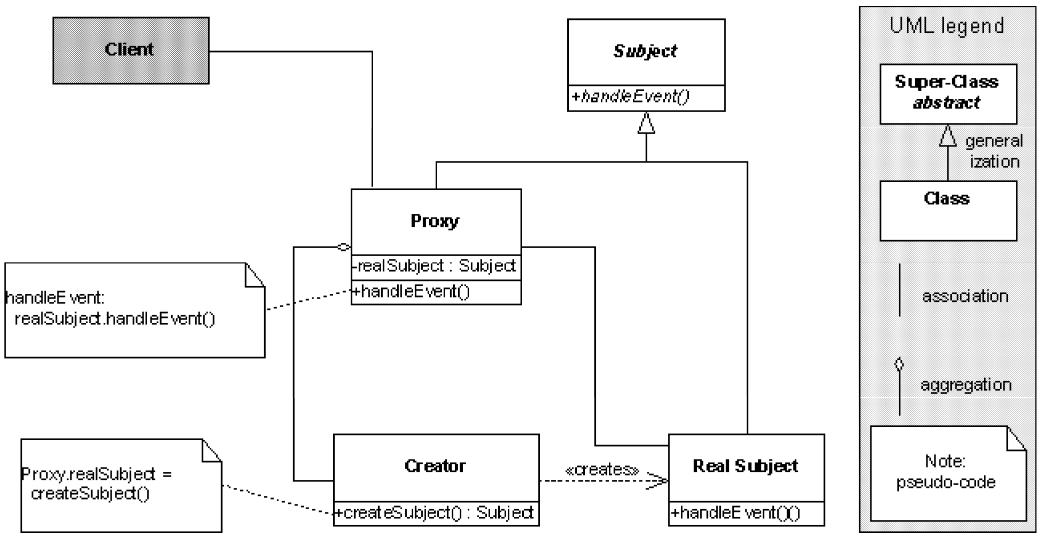


Figure 7.2: Structure of the Extensibility pattern.

the **Proxy** exists a **Creator**, which is responsible to create a new object of an arbitrary class **Real Subject** conforming with the interface **Subject**. Actually this pattern is a combination of the Proxy and the Factory Method patterns from [GHJV94].

Figure 7.3 shows the interaction between the objects. At initialization time, the **Creator** object passes a reference to a default **Real Subject** to the **Proxy**. Any event that the **Proxy** receives is delegated to the default **Real Subject**. When the **Creator** receives an event (how that happens will be discussed soon) to create a new **Real Subject** it instantiates the respective class and sets the reference in the **Proxy** to the newly created object. The **Proxy** now forwards all subsequent events to this object, unless the **Creator** changes the reference to a **Real Subject** again.

A slight variation of the pattern allows to add instances of new classes instead of replacing the old objects. This can be easily accomplished by letting the **Proxy** store a set of all **Real Subjects**. All incoming events are then forwarded to all instantiated **Real Subjects**. This variation is useful if new functionality is added, which is independent in the application logic from the already existing objects.

Subjects) relies on a stable definition of **Subject**. The **Proxy** class does not need to extend the same interface and may act as Adapter or Bridge. Proxy, Adapter, and Bridge are described in [GHJV94].

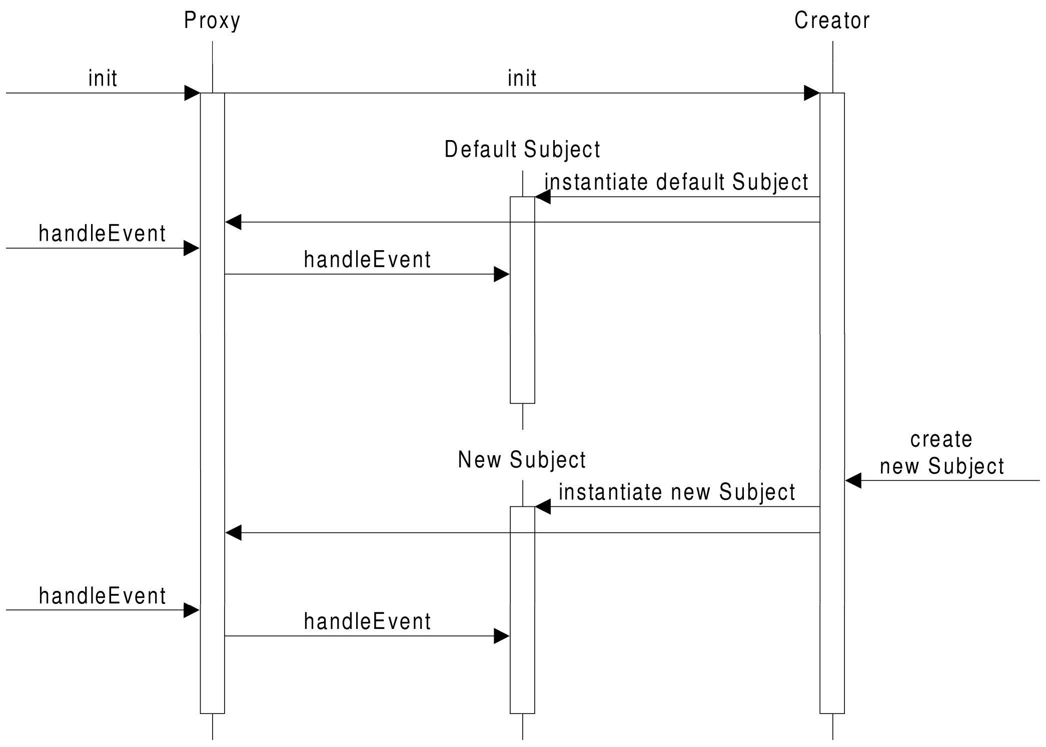


Figure 7.3: Interaction diagram for the Extensibility pattern.

7.4.2 Distributing and Inserting Components

Readers who ask themselves how the **Creator** in the Extensibility pattern is triggered, will get their answers here. The **Creator** encapsulates a *GroupReceiver* that subscribes to a group on which events may arrive that carry the classes to be instantiated.

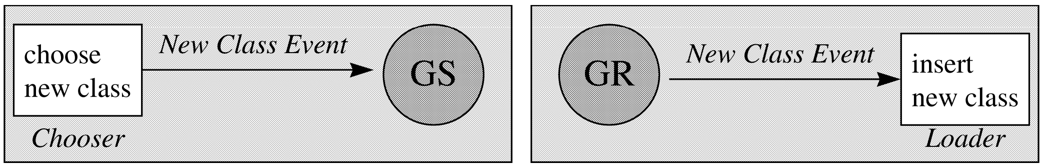


Figure 7.4: Distribution of a *NewClass* event.

The distribution of a new component is handled by this design as the distribution of a *NewClass* event by the beans for group communication (figure 7.4). The bean, which acts as a *Chooser*, selects the class, which should be inserted in the distributed application. Often a *Chooser* is embedded in the user-interface to let the user decide, which class should be inserted. Eventually, the *Chooser* fires a *NewClass* event. The event is simply passed by the *Chooser* to a bean that is derived from a *GroupSender*, which publishes it to the configured group. The event is then received by all beans that extend a *GroupReceiver* for this event type and are subscribed on that group. The *GroupReceiver* passes the event to a *Loader*, which instantiates the class. The resulting object can then be used by the **Creator** to replace or add a new **Real Subject**.

The combination of the Extensibility pattern with the group communication beans can be used to extend well specified hot spots in distributed applications; the specification is the interface **Subject**. If a plug-point defines a lot of methods, each component has to implement these methods, before it could be used to extend the plug-point. Sometimes, however, it is not feasible to be constrained by an interface. In the case of truly independent components, such as applications, it would be needed to write an adapter [GHJV94] to insert them. On the other side, even such components may use some of the available information by the loading component. Instead of using the static information provided by the interface, a variation of the Extensibility pattern uses the reflection mechanisms of Java and Java Beans to connect to the available plug-points.

Figure 7.5 illustrates this concept. A *NewClass* event arrives at the *loader* (a). Upon arriving, the *loader* loads the class and instantiates it (b).

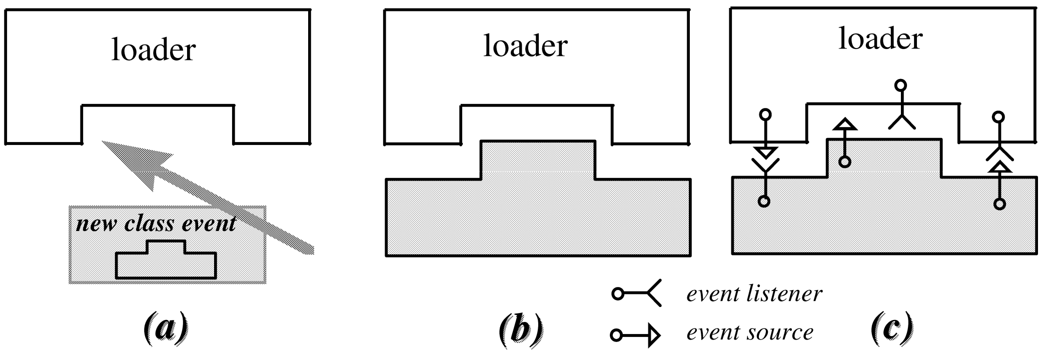


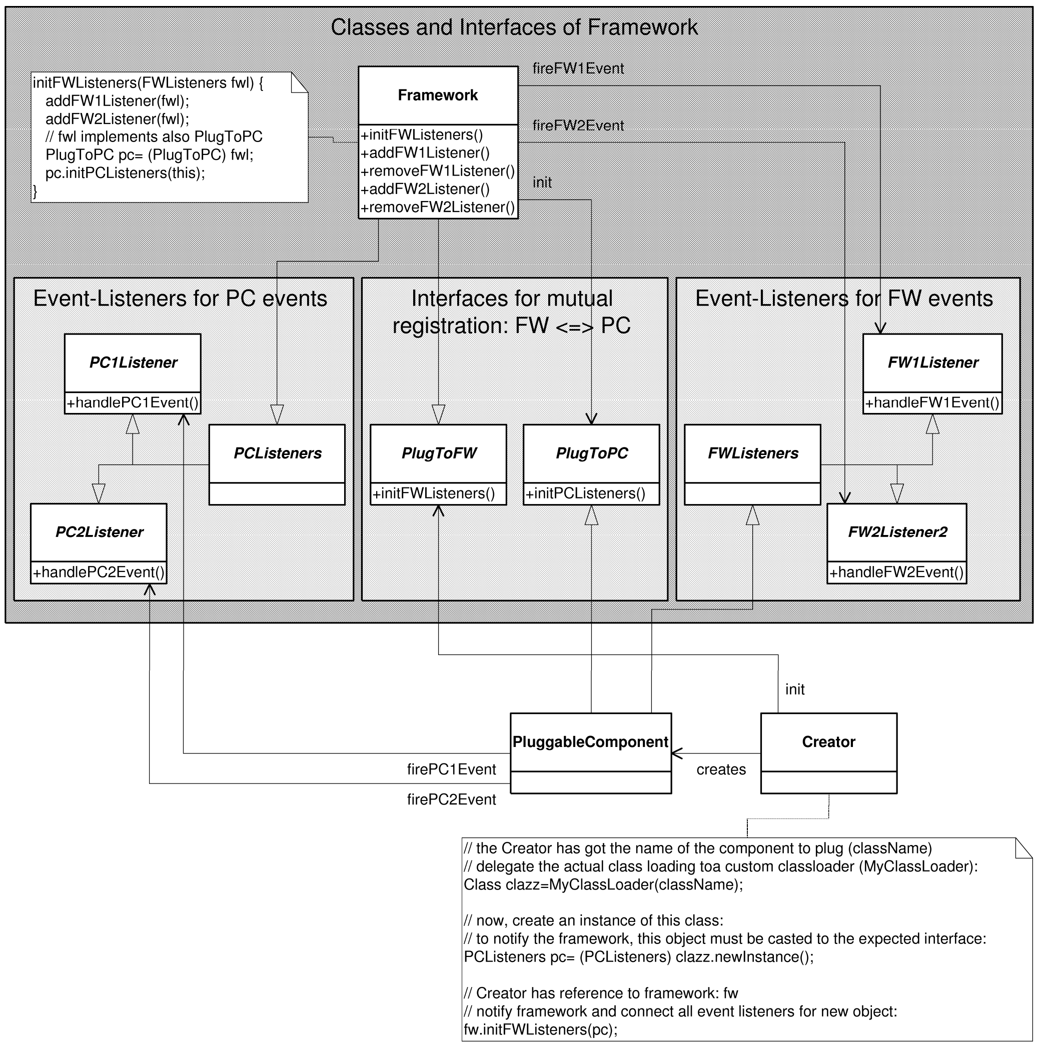
Figure 7.5: A loader receives a new class (a) and instantiates it (b); then the loader and the new object can register mutually (c).

Since the loader does not know at this time the features of the arrived bean, it uses introspection to discover the events, which can be fired by the new bean. For the events it is interested in, the loading bean adds its interest by calling the discovered registration methods (c). Now, the loading bean can receive events from the new bean. If the loading bean provides itself events and has discovered by introspection that the new bean implements the appropriate method to connect itself, it invokes that method. Then the loaded bean uses the same mechanism to subscribe itself to the events it is interested in.

The *NewClass* event may additionally carry the name of a start method. If the new class is not a bean, no events are connected, but the start method will still be called. Thus it is possible to pass arbitrary Java programs and start them remotely. The newly loaded code can interact with the loading application by means of two mechanisms: by mutual registration for the provided events that are discovered during initialization and by the presented group communication beans. The latter are also used to communicate with other remote applications.

7.4.3 Framework design for dynamic extension by components

Frameworks, which requires that a newly loaded component support a given event set, use a variation of the rigid design for plug-points as described in chapter 5.2.2. The communication protocol between the framework and the extending component is captured by the listener interfaces for the event types. Additionally, the interfaces for mutual registration are used to automatically register all events between the component and the framework. Figure 7.6 shows this design. Note, that this design differs from the original design (figure 5.1) only by introducing a new **Creator** component.



```
initFWListeners(FWListeners fwl) {
  addFW1Listener(fwl);
  addFW2Listener(fwl);
  // fwl implements also PlugToPC
  PlugToPC pc= (PlugToPC) fwl;
  pc.initPCListeners(this);
}
```

```
// the Creator has got the name of the component to plug (className)
// delegate the actual class loading toa custom classloader (MyClassLoader):
Class clazz=MyClassLoader(className);

// now, create an instance of this class:
// to notify the framework, this object must be casted to the expected interface:
PCListeners pc= (PCListeners) clazz.newInstance();

// Creator has reference to framework: fw
// notify framework and connect all event listeners for new object:
fw.initFWListeners(pc);
```

Figure 7.6: Design of a plug-point in a framework to be extended at run-time.

The **Creator** component in figure 7.6 implements all the class loading of the new component, its instantiation and then notifies the framework to connect all event listeners. **Creator** is designed and implemented as described for the Extensibility pattern. **PluggableComponent** in this design can be any component, which implements the required interfaces. The actual class code of **PluggableComponent** is retrieved by **Creator** at run-time and thus not available at design-time.

The **Creator** component can be itself part of the framework or be a component which is added at design-time by the developer. Having this flexibility, a framework which implements my plug-point design of chapter 5.2.2 can be taken without modification to being extensible at run-time by applying the design as depicted in figure 7.6.

The design shown in figure 7.6 applies the Extensibility pattern. The event listener interfaces correspond to **Subject** of figure 7.2. It adds to the Extensibility pattern the interfaces for mutual registration and the listener interfaces for events sent to the framework.

7.4.4 Applicability

Extensibility of CSCW applications can be introduced on various levels of granularity, varying from the one extreme, where only new applications can be started, to the other extreme, where every component may be extended. The place and number of plug-points in the design determine the extensibility of the application framework. But, the number of plug-points does not only worsen the performance of the application, but it increases also the necessary effort of maintenance.

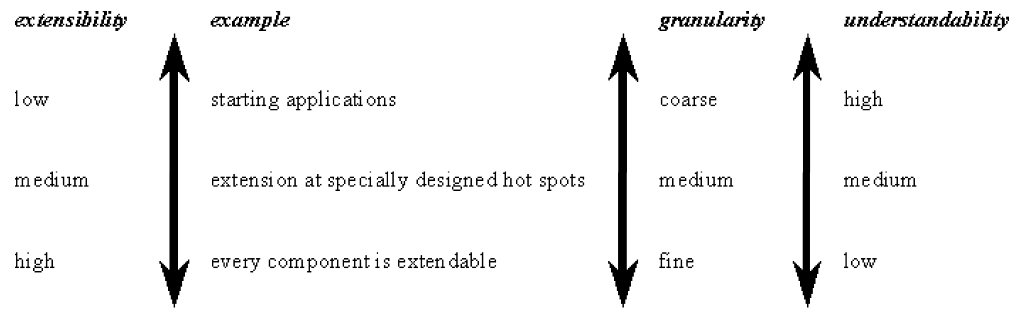


Figure 7.7: Trade-off between application extensibility, component granularity, and understandability for the end-user.

Figure 7.7 shows how the level of extensibility relates with the granularity of components that can be inserted and the understandability and maintainability for the end-user. The MBone tools [Eri94] may serve as an example for very small but successful extensibility: the user can click in the session directory (sdr) on a session, which starts the needed tools to join the audio and video session. The tools are stand-alone applications, which are started in a different process. Medium extensibility is granted by domain specific frameworks with some plug-points; TeamWave [RG97] is a groupware application, which uses a custom made component model on top of GroupKit [RG96a] to offer extensibility and tailoring support. The highest level of extensibility would be the usage of the Extensibility pattern for every component in a system.

If extensibility is only provided by means of starting applications in new processes, the original and the new application must use a protocol to exchange data, which is normally different from local interaction. Therefore, inserting components into a running application has the advantage that they can be integrated seamlessly; the new components become part of the application. The components can interact locally and use same the interaction protocol of the component model.

Experiments with the Extensibility pattern and component based CSCW applications suggest that most extensions of groupware applications happen at anticipated places. If the application uses design patterns, some plug-points can be found during the design phase [Sch97]. However, it remains an art rather than pure engineering to design extensible applications. The next section will give some examples, how extensibility can be designed and implemented in CSCW applications.

The skills, which are needed to create new components within a visual builder tool at design time, are the same as have been discussed in the chapters 4.3, 5.3, and 6.3.

Extending a groupware application by inserting new components at run-time requires support to access these components at run-time. Basically the end-user selects the components from a component library.

In the examples, I have used a special application for loading components or I have built these loading capabilities directly into the user-interface of the groupware application. However, the examples lack a design and implementation of a loader, which checks prior to the insertion, if the selected components fit the plug-point.

7.5 Examples

This section gives some examples, how the Extensibility pattern is used to design extensible CSCW applications. The first example presents a minimal CSCW component, which is used to distribute and start other cooperative components. We use a chat component as example to demonstrate the application of the Extensibility pattern. The insertion of a voting component during a chat session highlights the use of the Extensibility pattern to support unforeseen cooperation modes. Finally, this section summarizes some experiences of using the Extensibility pattern in tele-teaching components.

7.5.1 Design of a minimally extensible CSCW application

An example for a minimally extensible CSCW application is a loader that offers the functionality to distribute and insert coarse grained components, which are actually CSCW applications themselves. When the user selects a new component for insertion, the code is distributed to all participants of the group and started within their instances of the loader.

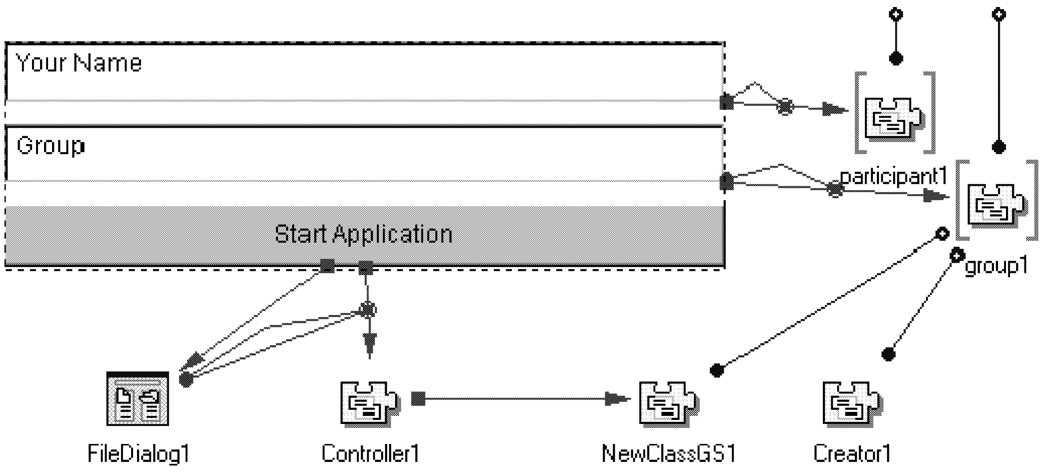


Figure 7.8: The loader application in a visual IDE.

Figure 7.8 shows the composition of the loader within a visual IDE[2]. The user interface consists of two beans to enter the participant and the group

[2]This and all subsequent examples are built with IBM's Visual Age for Java. A puzzle piece denotes a non-visual bean, a puzzle piece in brackets a variable, an arrow a connection between an event and a method, and a dotted line a connection between two properties.

name and a button to insert a new component. When the user presses the button, a file dialog pops up, which lets the user select a component. After choosing the component an event is passed to a non-visual *Controller* bean, which generates a *NewClass* event and passes it to a *GroupSender*, which is configured with the group name. All loaders of the group members will eventually receive the *NewClass* event and start the associated component by the *Creator*. The *Creator* for the loader is configured to add every received component and to use the reflection capabilities to register for available events. The loaded component can query the properties of the loader via reflection – in this case it finds the participant and group name.

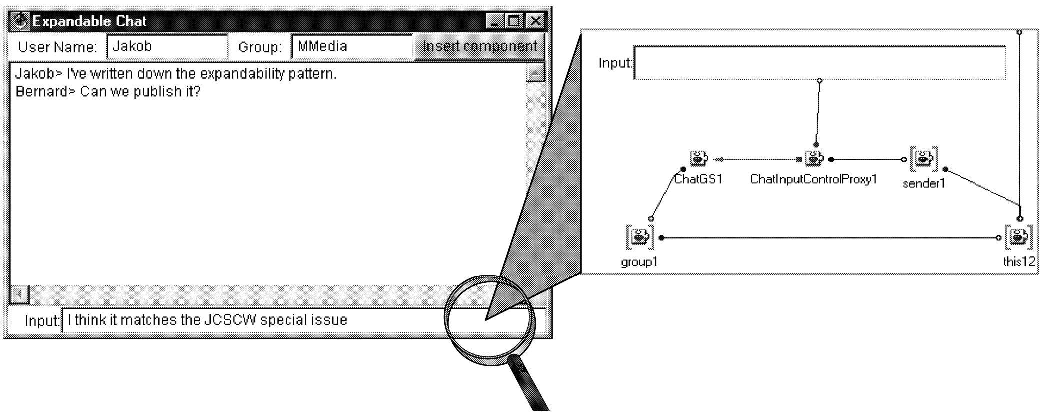
In the presented form, the loader supports the insertion of symmetric CSCW applications, i.e. applications that are executed at each participant. For example, the loader can be used to insert the components of the next examples: a chat and a voting component. An also developed alternative is a loader component for asymmetric groupware, which supports the local insertion of a server component, and distributes clients for this component to all other participants.

7.5.2 Design for functional extensions

A well-known example for a synchronous CSCW application is a chat. A chat allows the exchange of textual messages between all members of a group. This example will focus on the design of an extensible chat and present a component that can be inserted at run-time to support a simple floor control policy.

Figure 7.9 shows a running chat application, and the component composition at design-time for the input part of the chat. A new message is distributed by a *Chat* event to all participants; the output part of the chat component eventually receives the event and shows it to the user. The reaction on user input is performed within the bean *ChatInputControlProxy*, which has access to the input field and some environment properties. Whenever the proxy generates a new *Chat* event, it is distributed by a *GroupSender* for this event (*ChatGS*) to all participants. In this example, the user-interface additionally offers a button to insert a new component into the running application.

Figure 7.10 shows the internals of the proxy, which allows the replacement of the default strategy. The *BeanCreator* can receive new beans that implement the interface *ChatInputControlI*; it takes as default the component



User Name: Jakob

Group: MMedia

Jakob> I've written down the expandability pattern.
Bernard> Can we publish it?

Input: I think it matches the JCSCW special issue

Figure 7.9: The design of an extensible input component (right) for a chat tool (left). The *this* variable gives access to the methods and variables of the defining bean (here: the input component).

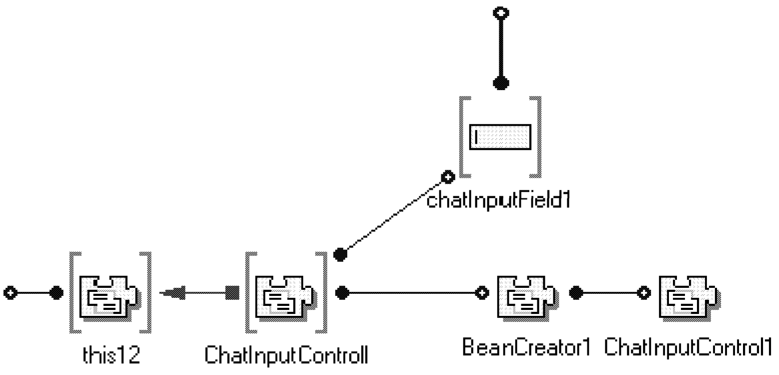


Figure 7.10: The design of the *ChatInputControlProxy*.

ChatInputControl. The input field and the current instance of the input control are associated with a variable of the type *ChatInputControlI*. Depending on the actual input control bean, a *Chat* event is fired to the proxy, which forwards it to the *GroupSender*.

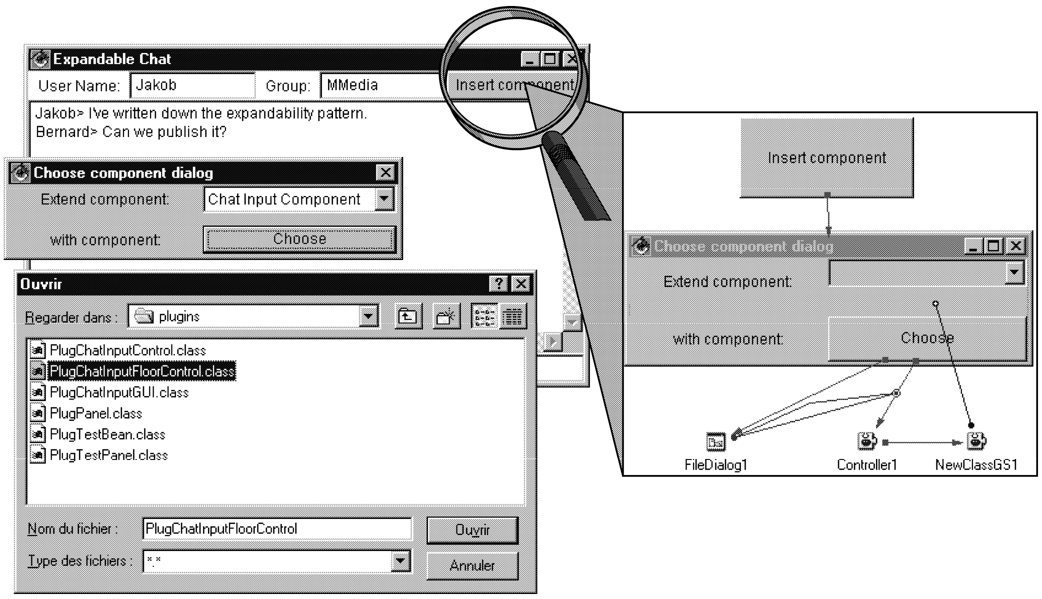


Figure 7.11: User-interface for insertion of a new component and its implementation.

This example implements a very simple mechanism to plug a new component into the running system (see figure 7.11). When the user clicks on the "Insert component" button of the chat application, a dialog box pops up and the user selects the plug-point to extend. Then the user chooses from a list of available components. The actual design and implementation for the selection uses the same components as the simple loader, which was previously described. As will be discussed later, a more sophisticated mechanism should be used in real-world applications.

To add a floor control mechanism, the default implementation of *ChatInputControl* (figure 7.12, left) can be replaced by *ChatInputFloorControl* (figure 7.12, right) during run-time. The new component displays an additional simple user-interface to request the token for input; the input field of the chat bean is only enabled, if the user has the token. It also uses *GroupSenders* to request and release[3] a token. The newly inserted compo-

[3]This implementation implicitly releases the token after a user has sent a message. The server for the floor control is not shown here.

nent interacts seamlessly with the existing components, since it implements the same interface *ChatInputControlI*.

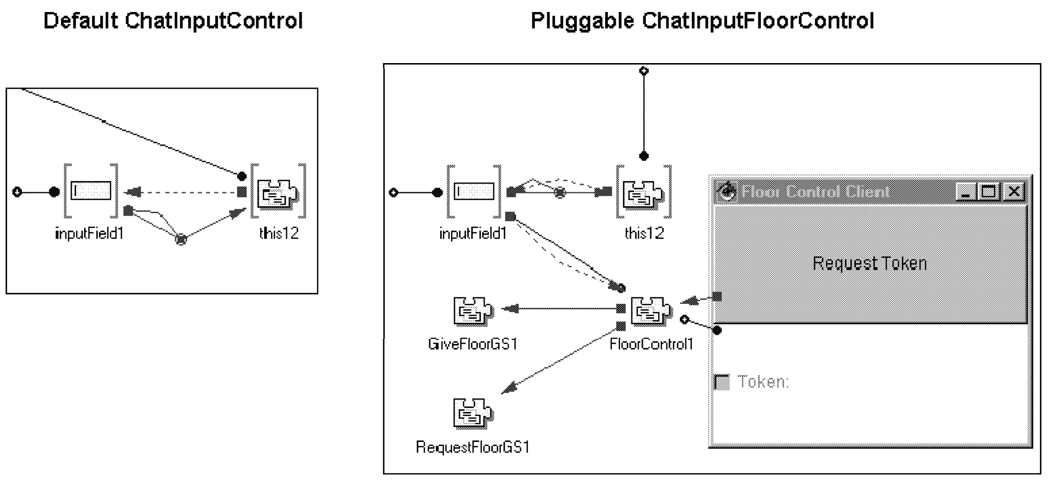


Figure 7.12: The *ChatInputControlProxy* is configured by default with the bean *ChatInputControl* (left); it can be replaced by *ChatInputFloorControl* (right) to support a token based floor control policy.

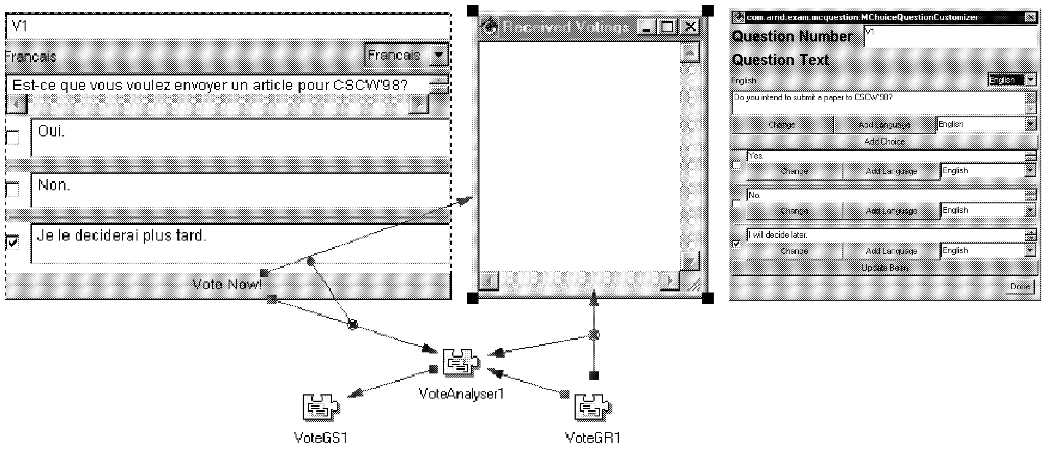
The design of the chat components follows a simplified Model-View-Controller pattern [BMR+96]. To insert components, which provides new behavior, the controller of the chat input component is designed to be exchangeable; the design uses the presented Extensibility pattern. The other chat components are designed in a similar way. Another plug-point is designed in the chat output component; a possible extension would be to add a component to write a log file of the discussion.

The chat example has shown the applicability of the Extensibility pattern to change the component's behavior at specially defined plug-points. It is thus classified in the medium level of granularity.

7.5.3 Design of a second application for insertion

The loader can be used to start more than one cooperative tool for all group members. For example, the chat tool is inserted for a discussion in a meeting with remote participants. After a while, a decision must be made about the discussed topics. The chair decides to create a list of the topics, and each participant has to vote for one item on the list. So, the chair uses an IDE to customize a voting component to be inserted and distributed using the loader.

The voting component is shown to each participant; after a participant has submitted his vote, a separate frame shows all arriving votes from the others.


V1

Francais

Est-ce que vous voulez envoyer un article pour CSCW'98?

Oui.

Non.

Je le deciderai plus tard.

com.arnd.exam.mcquestion.MChoiceQuestionCustomizer

Question Number V1

Question Text

English

Do you intend to submit a paper to CSCW'98?

Yes.

No.

I will decide later.

Figure 7.13: A voting component (left) and the associated customizer (right).

The design-time customization of a vote component is a very easy task: A question component is dropped on the vote panel within a visual IDE (figure 7.13). The vote panel has an associated customizer to add new questions, to manipulate them, and to provide different language features. The customizer offers the user a graphical interface to hide all details of programming. The end-user only performs drag-and-drop operations and fills in text fields. The customizer constructs a new voting component with this information, which can then be inserted by the loader to be distributed to all participants.

This example has shown the applicability of the Extensibility pattern for coarse grained components to support the insertion of new cooperation forms. It also has validated the approach to use off-the-shelf visual builder tools to let the end user build a new component by design-time customizing existing beans, which are then distributed and inserted into the running CSCW system.

7.5.4 Other examples

To prove the applicability, some of the earlier developed remote education frameworks and components have been redesigned to offer extensibility. As an example, the Extensibility pattern is placed in remote tutoring components (see chapter 4.4) to allow the insertion of arbitrary components supporting cooperation among the students and tutors. The tutoring components allow students to contact a tutor, if they want assistance in a remote laboratory course. The tutor gives peer-to-peer advice by using cooperation beans. In the original implementation the components for cooperation could be changed only at design-time; the new implementation can use several cooperation forms by inserting them at run-time. The tutor can now also distribute questionnaires (see chapter 6.4, but also chapter 8) to all students at the end of a laboratory course to monitor their learning progress. The tutor has prepared the questionnaire during the laboratory course based on the issues that have been discussed with the students. The creation of such a questionnaire is highly supported by customizers within a visual IDE. The presented customizer for the vote panel is actually a reused component for multiple choice questions from this tele-exam framework.

7.6 Discussion

The examples have shown the applicability of the Extensibility pattern within component based CSCW applications. By using the pattern one actually designs domain specific application frameworks. These application frameworks can be extended at run-time by inserting new components. The new components can be created by the end-user outside the application within visual IDEs.

The examples have used the Extensibility pattern to insert coarse and medium grained components. The placement of the plug-points with the pattern in the examples is based on the anticipation of possible extensions. This leads to the question whether a rule can be given where the plug-points should be located.

The main problem is that a conflict exists between the level of extensibility and the level of understandability (cfg. section 7.4.4). If each component was made extensible, the design and implementation would become unnecessary complex. Even, if the performance affected by the added complexity could be improved (for example with the Flyweight pattern [GHJV94]), the maintainability criteria still limits the amount of plug-points. On the other hand, too few plug-points limits the extensibility of the application. So, a compromise must be found depending on the domain of the application.

Experience shows that the design of cooperation offers a good starting-point to insert plug-points in CSCW applications. Components that are triggered by user actions and perform operations depending on these actions are candidates to be extended. If the design of an application that uses design patterns, the location of potential plug-points can be derived from the design [Sch97]. In the case of CSCW applications, patterns used for cooperation must be examined. In the often used Model-View-Controller pattern, a potential plug-point for extension in each application is located in the Controller, while the View would be a candidate for being extended only locally. Cooperations that can use different strategies, can change their strategies by placing the Extensibility pattern within the Strategy pattern [GHJV94]. A good example for adding a new strategy component would be a new algorithm for video encoding and decoding in conference systems. The Mediator pattern [GHJV94] can be used to design tailorable CSCW systems by attaching cooperation enablers [Syr97] to cooperative artifacts. By placing the Extensibility pattern within the Mediator new enablers could be introduced in the running system. This section has presented a non exhaustive list of potential locations, where plug-points could be useful, it is still up to the groupware designer to decide, where plug-points will eventually be placed. Her or his analysis will be oriented on the domain of the application.

New components can be inserted on the demand of other components or on the demand of the end-user. In the latter case, the end-user must be supported by a user-interface to select the appropriate hot-spot and component to obtain a certain behavior by his extension. I have used a simple file chooser in my examples. A more sophisticated approach would present the user the potential plug-points and a list of available components that are available to extend each one. By introspecting the selected component, such a list can be created automatically. Additionally, the user should also get a description of the intent, effects and possible side-effects for each component.

The presented implementation to insert components at run-time uses code distribution. To inform remote applications to insert a new component, the group communication beans are applied to distribute arbitrary events. The needed information about the new components is encapsulated in an event. So, the implementation is coherent with the Java Beans event model. Thus it is supported by visual builder tools for Java Beans.

The distribution of code has the advantage that components have access to the local system properties. Thus user-interface components can also be distributed. Another advantage lays in increased performance compared with remote object communication if the inserted component is often used. The biggest advantage in a cooperative environment is that the component which should be inserted in the running application needs not be installed at the remote machines before the application is started.

The operation of loading and instantiating classes via the network opens severe security risks. Since Java is a network language, these risks are well-known and methods for protection exist. Java code can be signed. A signature authenticates the creator of the code. If code is manipulated after signing, this can be detected. Although signed code allows one to only accept code by trustworthy sources, the problem of who to trust remains. In a cooperative environment, this question is hard to answer. Even if all persons that are allowed to distribute new code are trustworthy, failures in the distributed code can cause damage [Zha97]. The problem can be partly solved by giving explicit rights for customizing code [SC98]. Another barrier can be inserted by granting new classes only the rights they need to function. If, however, a class claims to need full rights and is created by a person of full trust, the problem remains. This problem can not be generally solved.

7.7 Conclusion

This chapter focuses on the insertion of new components into running synchronous CSCW applications to tailor their behavior. My approach is to split the act of tailoring into the steps of the design-time customization of new components within visual IDEs and their insertion into the running application. This decoupling leads to a shorter development cycle of applications. Furthermore, the end-user needs only to accustom to one IDE to tailor different applications. When IDEs will be delivered as components, my approach can be taken to extend CSCW applications with those pluggable builder tools.

My approach uses a design pattern, which is focused on modeling plug-points for remote insertion in a general way. Extensions are implemented as Java beans and distributed through remote events. They are then automatically inserted at the provisioned hot spots. Once inserted, the new components are seamlessly integrated within the running application. Independent coarse grained components that function also without information about their environment can be inserted without conforming to a predefined interface. Nevertheless they can query their environment via reflection to register for events or to read and write properties. Thus arbitrary applications can be distributed and started remotely.

This chapter discusses the tension between extensibility and understandability in the design. Increasing the extensibility increases also the complexity of the design and thus decreases the understandability. This leads to the conclusion that a design is not reasonable where all components are extensible or exchangeable during run-time. It remains still a task for the application designers to identify the plug-points from their expertise. Once potential insertion points are recognized, the developer can uniformly design the plug-points using the introduced Extensibility pattern.

One problem, which should be addressed by further work, is how the extensibility can be presented to the end-user. The presentation should include the hot spots of an application and their possible extensions. Such a presentation must find means for an intuitive graphical user-interface to insert components at the right places.

Chapter 8

Customization becomes tailoring II: Object distribution

This chapter introduces another mechanism to insert customized components within a distributed framework. Instead of distributing the code of the components (i.e. the classes), this approach distributes objects (i.e. instantiated classes). This approach is particular useful to instantiate a group of objects remotely at run-time, if the static class structure is already known at design-time.

8.1 Contribution

- Adaptation of the two-step approach of tailoring: Creation of customized component instances in a visual builder tool; insertion of these instances at run-time.
- Design of groupware frameworks, which support the remote insertion of component instances at run-time through object distribution.
- Discussion of differences between this approach and the approach of dynamically loading the component code.
- Use of object distribution and insertion at run-time for initialization of remote groupware applications.
- Presentation of the synergy effects by using Customizers at design-time together with object distribution at run-time on the example of the

Questionnaire framework to create, distribute, recollect, and evaluate active exams.

8.2 Frameworks that support object distribution

Chapter 7 describes how groupware is extended at run-time by dynamically loading new code, i.e. by loading new classes and their subsequent initialization. In contrast, inserting objects, i.e. instantiated classes, at run-time requires that the class code is already accessible by the Java Virtual Machine.

The distribution of objects at run-time has several advantages over the alternative, which is the remote method invocation to set up the state in the distributed objects. Since this thesis concentrates on customization and tailoring, I have chosen to incorporate object distribution to set-up remote groupware applications, because it facilitates drastically the ease of use. The distribution of objects instead of using remote method invocations eliminates potential synchronization problems, if otherwise several calls would been needed. It is also more natural for the user to distribute a whole entity than to set multiple properties. From the implementation point of view, Java supports very well object distribution by offering object serialization.

Serialization is used to store persistently an object or to transmit the object over system boundaries. The Java Beans specification [Ham97] requires that a bean is serializable to store its settings, after being manipulated within a builder tool.[1] Builder tools may use serialization to persistently store customized bean instances for later reuse.

My approach benefits from Java's serialization possibilities two times: First, the builder tools generate serialized beans, when the user customizes these beans. So, this approach integrates very well with the customizer approach (see chapter 6). Second, the serialized objects are transmitted to the remote CSCW applications, where they are deserialized and inserted in the running local application. From the viewpoint of the local applications they behave like normal objects after the insertion.

While object distribution together with customization features within

[1]Although the Java Beans specification allows that a builder tool creates initialization code to set up the properties of a bean, it highlights that serialization should be preferred and must be supported.

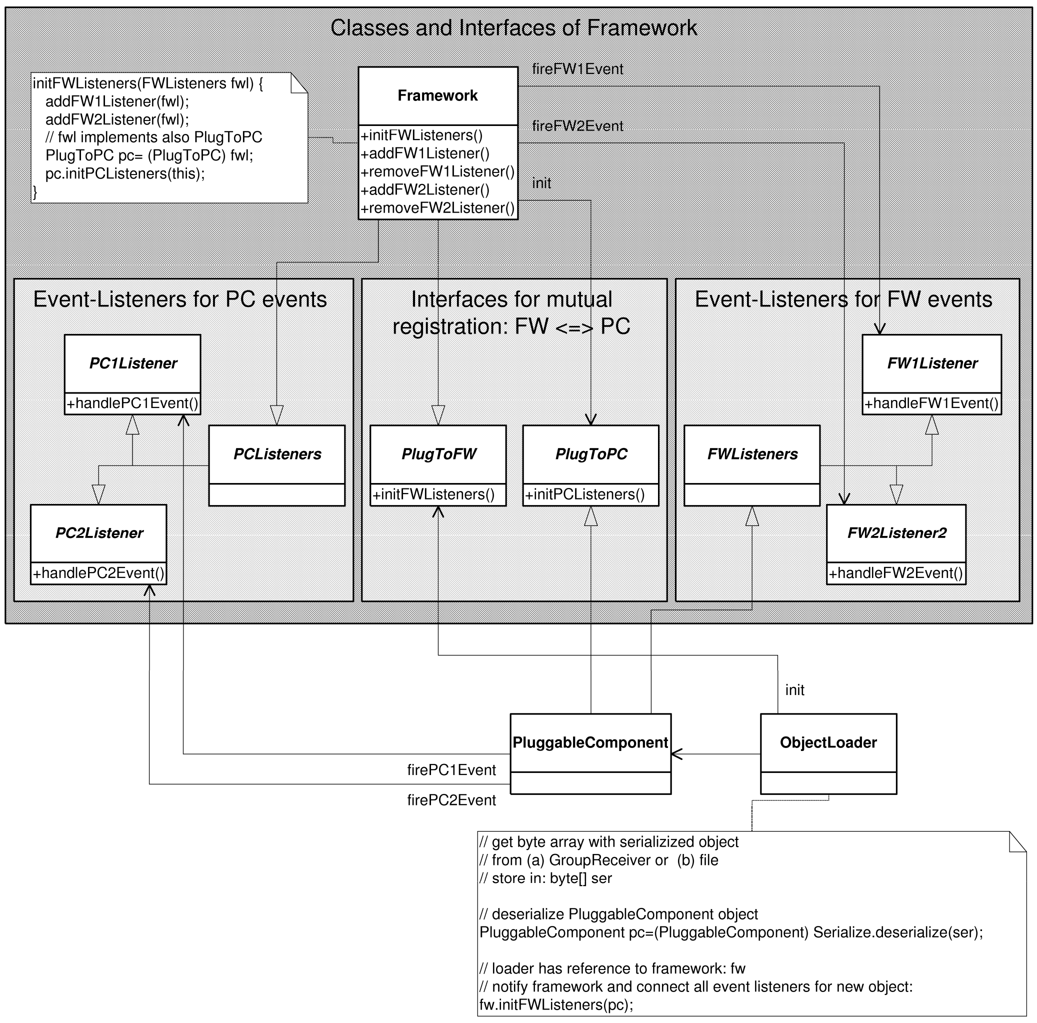
builder tools directly support my goal to tailor groupware frameworks at run-time, some of my frameworks use serialization to move objects from one user to another at run-time. The Tele-Exam example (see below) does not only distribute the customized questionnaires to the students, but also collects after a given time all questionnaires from the students. By answering the questions, the students have changed the state of the question beans. These changes are captured in the serialized questionnaires and the tutor application allows for persistent storage and automatic evaluation.

CSCW systems can benefit from the possibility of dynamically loading of objects in other areas, too. Serializing and loading objects facilitates the implementation to save and resume a session, to remotely initiate a cooperation, and to administer centrally a CSCW system.

My approach uses frameworks as the part of each local groupware application, which is initialized at start-up time and does not need a different configuration each time they are used. Such a framework knows about the the class structure of those objects which are dynamically inserted; however that local application, which initiates the cooperation delivers in this approach all bean instances, which are needed to support the particularities of this specific collaboration. This is often the case in asymmetric groupware, which has one "super-user" (in tele-teaching: the professor) and many normal users (students); but also in symmetric CSCW applications, the actual cooperation is normally initialized by one user and this user decides – often implicit – how the components need to be initialized to support the desired cooperation.

My approach is particularly useful, if at design-time it cannot be determined how many objects are needed, these objects however are held in a tree structure and the framework needs only to know how to traverse the tree and to access the objects. This design pattern is known as Composite [GHJV94]. The Questionnaire framework is an example, which uses the Composite pattern to store all questions in one container, which can be then distributed.

Figure 8.1 shows the framework design, which includes the **Object-Loader** component to dynamically insert an instance of **PluggableComponent** in the framework. **ObjectLoader** is responsible to get the serialized object, to deserialize it, and to notify the framework about its availability. **ObjectLoader** get the serialized object by a **GroupReceiver** (not shown), if the initialization happens remotely. In the case, the new instance is inserted on behalf of the local user it is read from a file or downloaded from



```
initFWListeners(FWListeners fwl) {
   addFW1Listener(fwl);
   addFW2Listener(fwl);
   // fwl implements also PlugToPC
   PlugToPC pc= (PlugToPC) fwl;
   pc.initPCListeners(this);
}
```

```
// get byte array with serializized object
// from (a) GroupReceiver or  (b) file
// store in: byte[] ser

// deserialize PluggableComponent object
PluggableComponent pc=(PluggableComponent) Serialize.deserialize(ser);

// loader has reference to framework: fw
// notify framework and connect all event listeners for new object:
fw.initFWListeners(pc);
```

Figure 8.1: Design of a plug-point in a framework to dynamically insert objects.

the network. The **ObjectLoader** component can be itself part of the framework or be a component which is added at design-time by the developer. Again, implementations that follow the original framework design (see chapter 5.2.2) do not need to be changed in order to support the dynamically remote initialization.

8.3 Applicability and needed skills

This approach supports tailoring by the same two-step approach as described in chapter 7.3: In the first step, an end-user uses an off-the-shelf visual builder tool to customize a component at design-time. The customized bean is then distributed and inserted in the local applications of the already running CSCW system.

For the developer, a framework, which distributes serialized objects, is easier to develop than to dynamically distribute and insert new code as proposed in chapter 7. Dynamically loading serialized objects is also more secure than inserting new code, since the class code for the serialized objects is already known at design-time. On the other side, the knowledge about the class code at design-time also excludes that the CSCW system is functionally extended at run-time.

Combining the framework approach to tailor a CSCW system by the distribution of prior customized components is most promising, if during the design phase the structure of the system is known, but its actual instantiation changes from time to time. The framework consists of the constant parts; therefore it needs to be deployed only once. The components that are often customized are factored out during design-time; their instantiations are then distributed and inserted after their customization.

Within groupware, such a remote instantiation usually happens at the begin of a new collaborative session, which can however also happen long after the start of the groupware application. Furthermore my approach is well suited to support late-comers.

Since this approach supports only the distribution of component instances, the customization within visual builder tools is restricted to setting properties by the means of the offered property editors and the customization with the help of Customizers. Such a customization is relatively easy and well supported by most builder tools. Users, who customize the components, need to be familiar with the offered means to customize a bean. Power-users and groupware administrators have these skills. Groupware integrators may offer libraries of pre-customized beans, which are the selected by the users. By using Customizers for the dynamically distributed beans, also end-users profit by this approach (see chapter 6).

This approach can be combined with the approach of dynamically extending a CSCW system with new components as proposed in chapter 7. The thus newly inserted components defer their instantiation to a later time, when the user submits the customized beans.

Another variation is to implement Customizers so that they can also be executed at run-time. The end-user opens then the Customizer at run-time to change the bean instance, which is subsequently serialized and distributed. This variation has the advantage of offering a seamless integration within the running application on the expense of losing the possibilities of visual builder tools, which include adding beans by drag and drop from a component library and using property editors. This variation requires that every customizable bean is supplied with an executable Customizer, which is accessible from the user interface. I have decided that these disadvantages overweight the advantage of seamless integration and focus here therefore on the two-step approach.

8.4 Example: Distributing a new questionnaire with active components

Exams are held in all educational environments to control the progress of the learners. Exams can differ not only by their contents, but also by their purpose and how the questions are being asked. This example focuses on written exams, which are distributed at the same time to the learners and must be returned before a certain amount of time has been elapsed.

Figure 8.2 shows a typical life-cycle for questionnaires in an exam; the figure holds for traditional paper-based exams as well as for tele-exams. A professor designs the questions for the exam. The resulting questionnaire is then copied and distributed to all students at the beginning of the exam. The students fill out the questionnaires. Should they have questions they may ask a tutor for clarification. The exam must be completed in a given time; when the period has elapsed all filled questionnaires are collected. The professor evaluates each exam to grade the students afterwards. Finally all questionnaires are stored persistently.

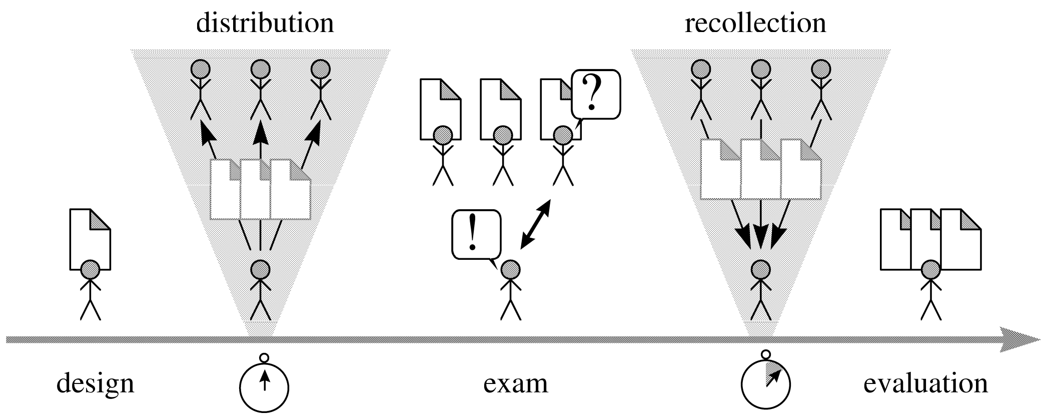


Figure 8.2: Life cycle of a questionnaire.

On-line exams are useful to support spatially dispersed people in remote education scenarios. On-line exams differ from traditional exams also in the way that the exams can use the computer to offer capabilities, which are not available in traditional paper-based exams. Using questionnaires that contain active code add more features as would be possible in paper-based exams. The questionnaires offer the students a better and more intuitive user-interface and let them choose properties, such as the language for presenting the questions. Animations can be delivered as well to help understanding; an example would be a rotateable 3D view on a molecule for a chemistry course.

In order to support exams by computers, not only their distribution must be solved, but the professor should be given an environment to easily design the artifacts and to evaluate and store the results after collecting the answers. Our questionnaire framework supports exams during the whole life-cycle, from the design of a questionnaire, through the actually held exam, to the possibility of automatically evaluating the answers.

8.4.1 Design phase

For a tele-exam, the assembling of question-beans to questionnaires needs to be as simple as possible. Special customizers for all offered question-beans ensure that a composition at design-time can be done with drag-and-drop.

The professor needs only to drag and drop questions within an IDE in a questionnaire container. The questions can be customized visually with the help of Customizers (see also the example in chapter 6.4). The connections

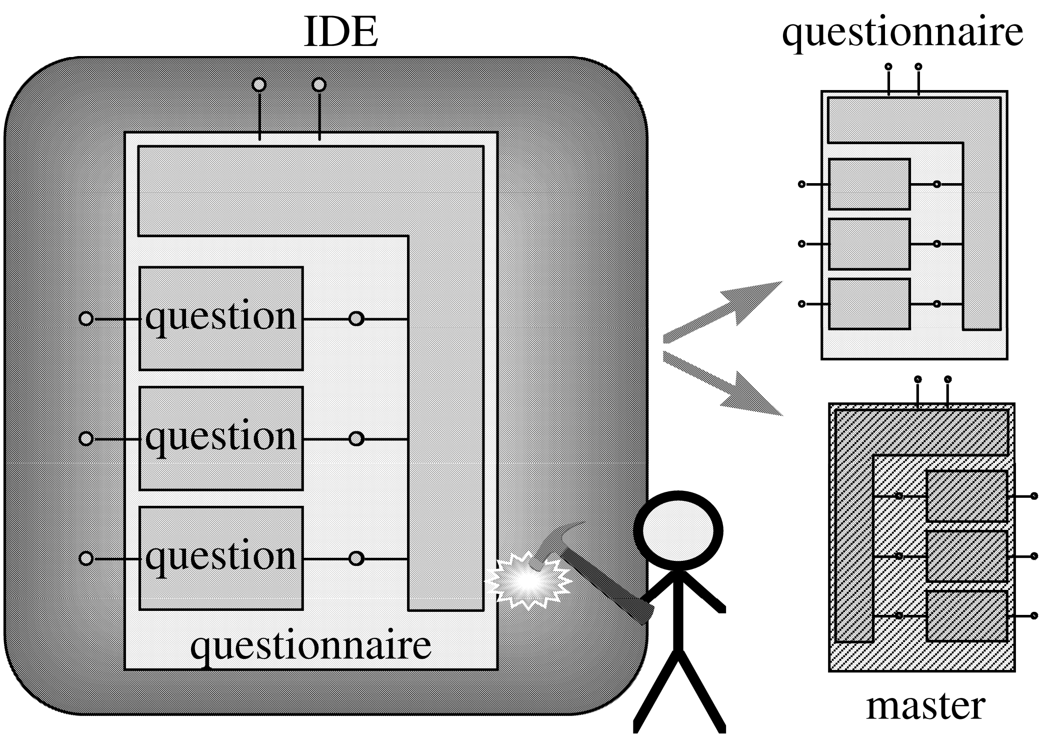


Figure 8.3: Composition of a questionnaire.

for the interactions between the questions and the questionnaire are done automatically. The resulting questionnaire and questions implement all needed interfaces to be plugged into the student application and for the final evaluation by a master copy of the questionnaire together with the right answers (see figure 8.3).

8.4.2 Distribution phase

A questionnaire with the contained questions is for the system an ordinary Java object. Also a blank questionnaire contains state (e.g. text, animations, timer). So the distribution takes advantage of the capability of Java to transmit objects. The questionnaire and the contained questions implement the needed interfaces to be plug-able in the applications for the students and the teacher. Other beans could have been used also by the teacher, as long as they conform to the interfaces.

The introduced beans for group communication are used to transport events between distributed parts of the system. The questionnaire is sent to all students at the beginning of the exam. Figure 8.4 illustrates how a questionnaire is distributed. The responsible component in the tutor application (here: a questionnaire manager, see also figure 8.6) signals a GroupSender bean to send the questionnaire to the group, e.g. all students. The GroupSender serializes the questionnaire and puts it, encapsulated in an event, onto the underlying communication system.

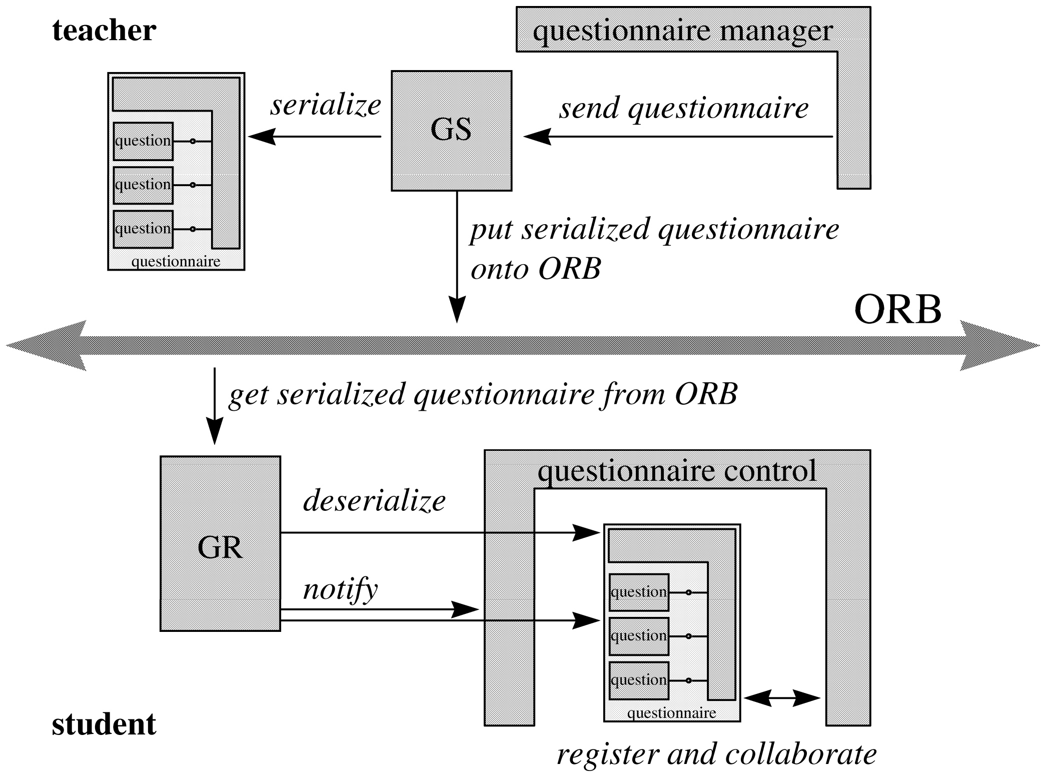


Figure 8.4: Distribution of questionnaires.

All GroupReceivers that are configured to listen on this event, receive the serialized questionnaire, de-serialize it, and notify the responsible components. In a student application, the questionnaire control registers its interest in some offered events from the questionnaire, and the questionnaire registers itself for events from the control. After the registration phase, the components are plugged, and they may start collaborating.

8.4.3 Exam phase

During the exam, students answer the questions by filling out the questionnaire. In my current design, the questionnaire itself is responsible to offer an appropriate user-interface.

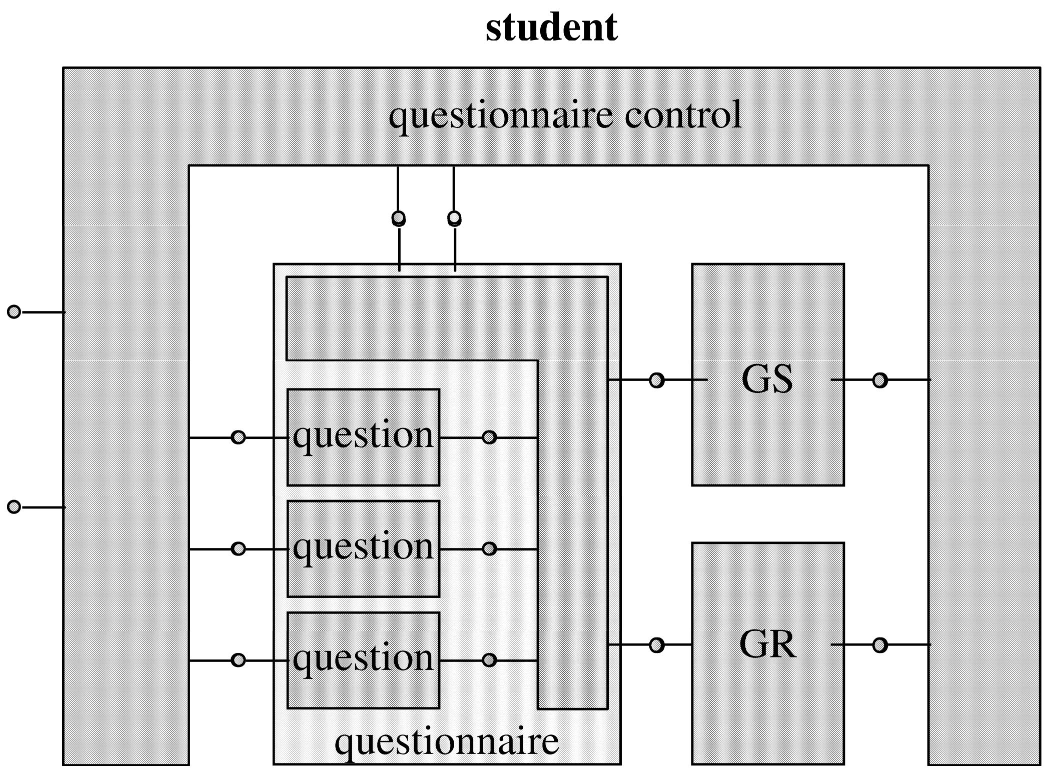


Figure 8.5: Questionnaire in a student application.

Figure 8.5 shows the collaboration between a questionnaire and the control. The student uses the questionnaire control to manipulate properties of the questionnaire and its questions. Examples are the preferred language for the questions or to lock the questionnaire against unwanted accidental overwriting at browsing the questions.

The questionnaire control bean has also the possibility to communicate with other beans. Since also carefully designed exams are sometimes ambiguous, the student needs the possibility to contact a tutor. Other beans, such as beans for tutoring (see chapter 4.4), can collaborate with the control to retrieve more information, e.g. the question which causes the problem for the student.

A special bean, which can be inserted into the questionnaire, manages a count-down timer. When the time has elapsed, the timer informs the control that the exam is over and triggers the GroupSender to send the questionnaire back to the configured address, e.g. the professor's application. The functionality for the collection of the exams is the same as described for the distributing case. The questions have changed their states due to the given answers of the students; the serialized questionnaire which is sent back to the professor reflects the new states, i.e. it contains the answers.

8.4.4 Evaluation phase

The professor uses a questionnaire manager within his application to evaluate the returned questionnaires, which are plugged in as described in the distribution phase. The manager holds all questionnaires and offers for automatically evaluatable questions an evaluator bean, which is connected with the master copy of the questionnaire as obtained from the design phase (see figure 8.6), which contains the correct answers.

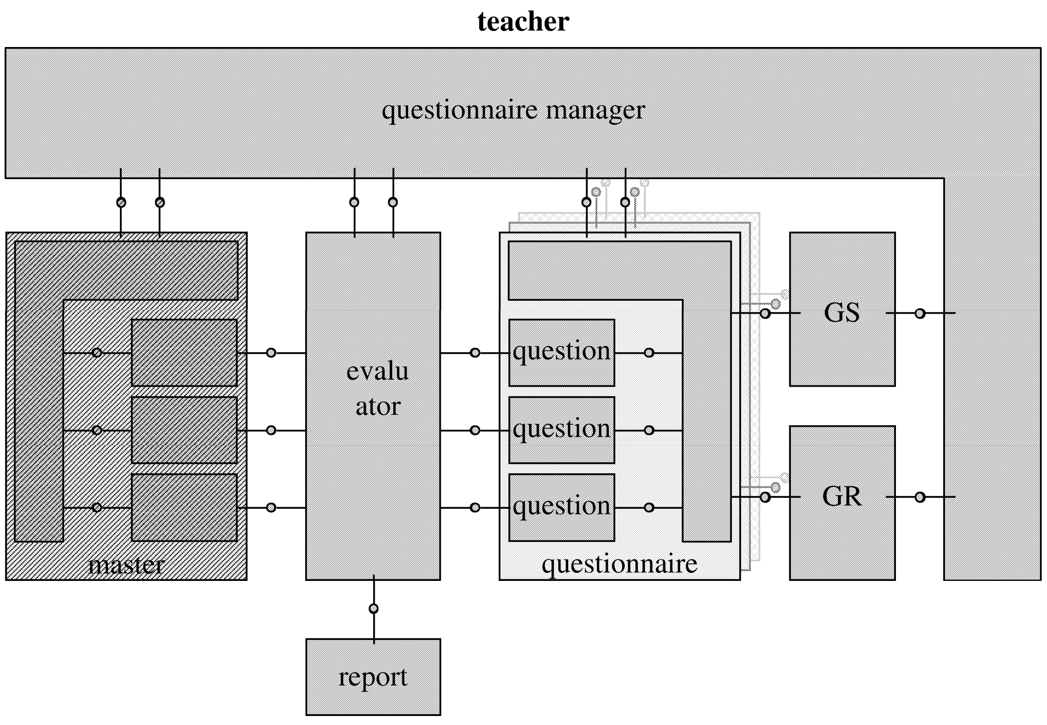


Figure 8.6: Questionnaires in a tutor application.

The evaluation for each questionnaire is passed to a report generator bean. The report can be edited manually by the teacher to include corrections of not automatically evaluated answers, comments and the final grade of the exam.

8.4.5 Composed applications

This section gives an overview of how student and tutor applications can be built upon the beans presented in the previous sections. To demonstrate the usefulness of the component approach, I outline the combination with other cooperative beans.

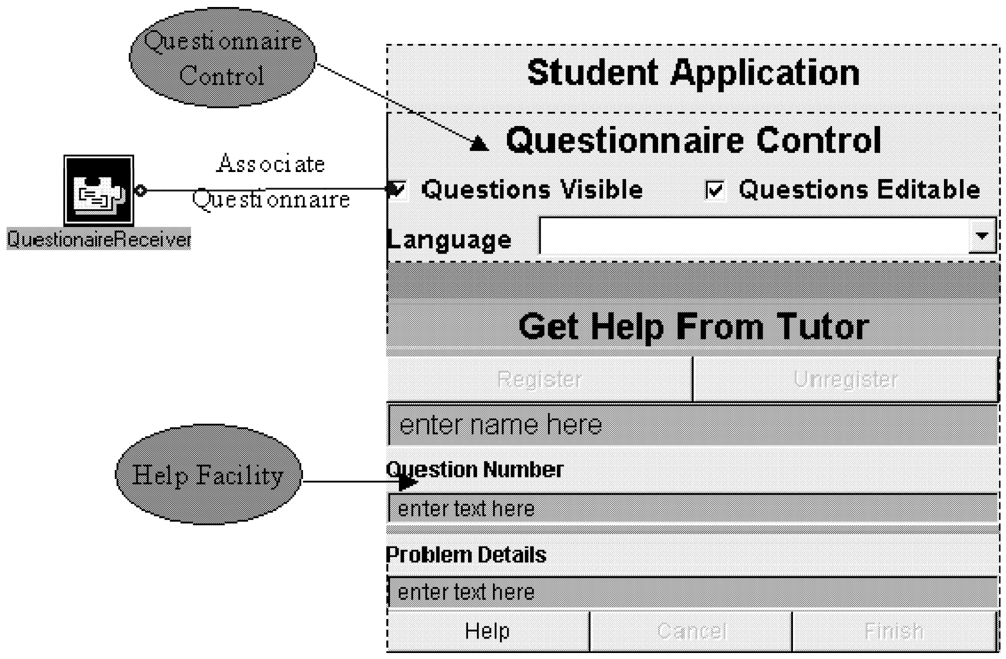


Figure 8.7: Student application in IDE.

A student uses the questionnaire control during the exam to receive the questionnaire and set global properties. The questionnaire control bean is combined with the tutoring beans (see chapter 4.4) to allow the student to ask the teacher questions during the exam. Figure 8.7 shows the questionnaire control, which allows one to hide the questions, toggle a lock for editing and select the preferred language. The settings can be made before the

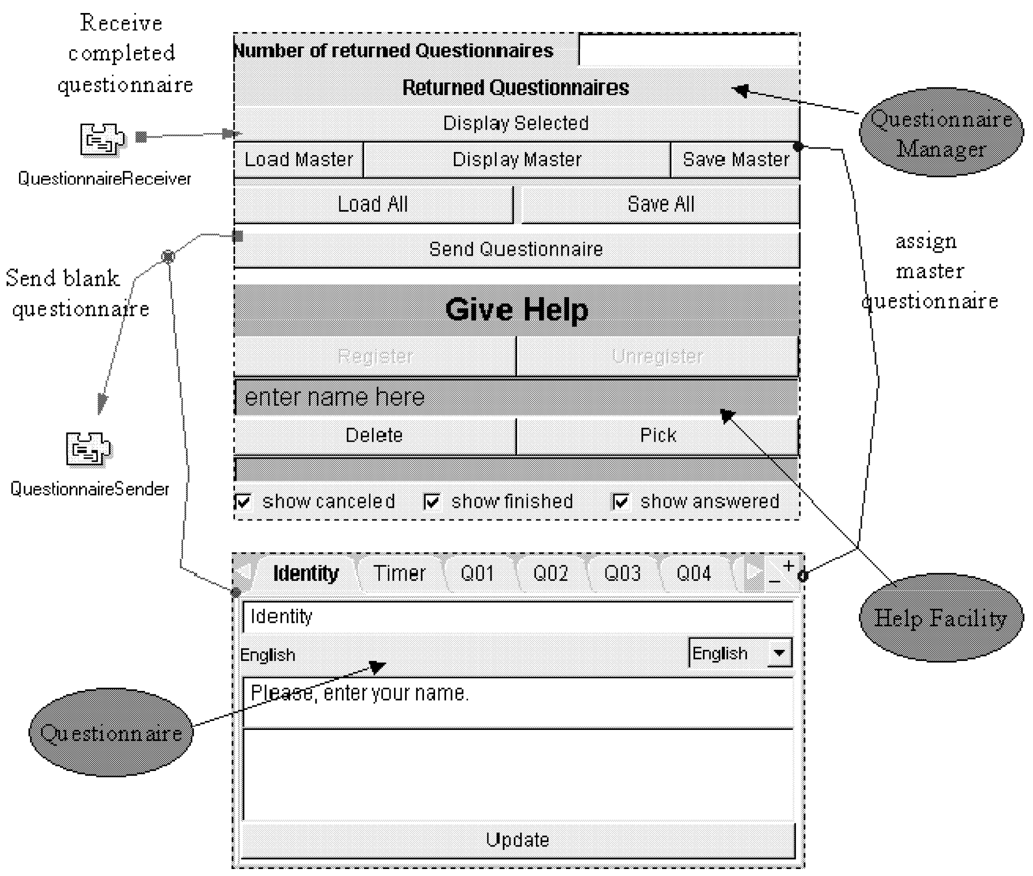


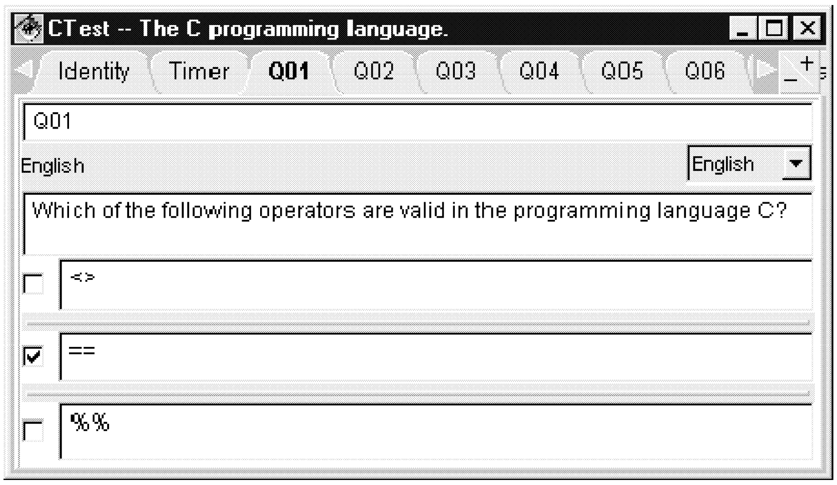
Figure 8.8: Tutor application in IDE.

questionnaire is received and during the exam. The figure shows also the bean for getting help by a tutor.

Figure 8.8 shows the beans for the questionnaire manager combined with the give help facility used by the professor. The shown questionnaire manager has a button to send a blank questionnaire to all students. After the answered questionnaires are received, the manager can show them and they can be compared against the master questionnaire. For automatically evaluatable questions an evaluator supports the correction. Questionnaires can be saved persistently and retrieved later.

The help facility shows help requests by the students and can be answered individually. When the tutor chooses a help request a configured communication tool is used; currently beans for chatting and sending back a list of answers of frequently asked questions (FAQ) are implemented. Additionally this composition supports the generation of an HTML FAQ from a chat session during the exam, which can be viewed by the students with Web browsers. To guarantee that the viewed FAQ shows always the up-to-date contents, this implementation incorporates additionally the prior developed beans for active annotations of Web pages [HKM97].

The shown example uses an exam about the C language, how it is held at Eurecom to test the knowledge of newly arriving students. Since Eurecom is an international university in France, the developed questionnaires support the languages English, French and German. The design of the question interfaces however allows one to offer as many languages as wanted.



Q01

English: English

Which of the following operators are valid in the programming language C?

[ ] <>

[✓ ==]

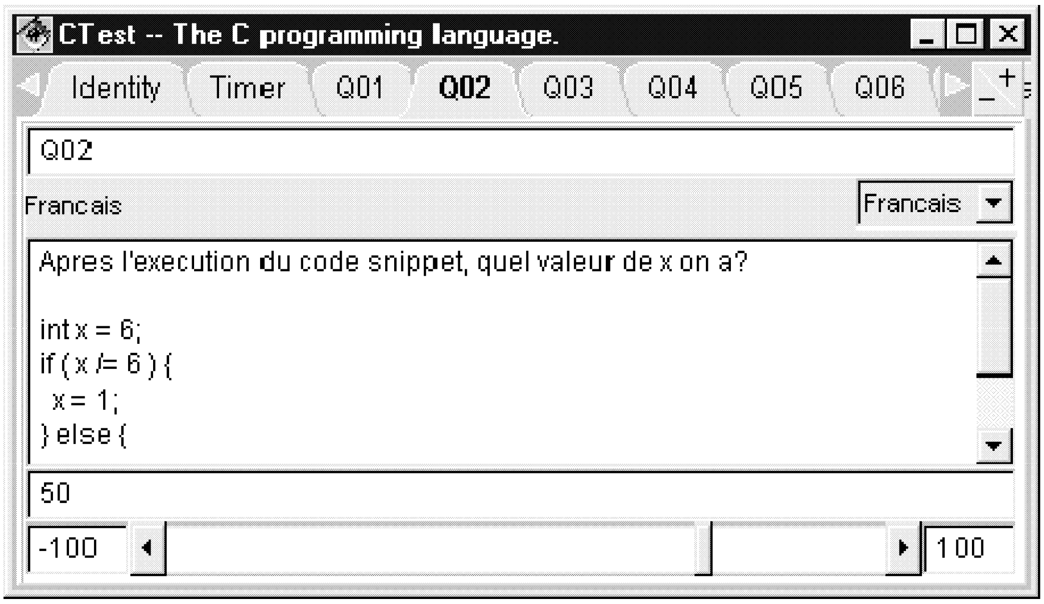
[ ] %%

Figure 8.9: Multiple Choice question.

Questions, which can be evaluated automatically, implement the interface `Evaluatable`. Currently, the component library offers two such question types: an integer value question and a multiple choice question. Both types are accompanied also with a user-interface, which supports intuitive input. The multiple choice question type is shown in figure 8.9.

A scrollbar supports the input for the integer value question type, as shown in figure 8.10. Of course, also other representations could be chosen during design-time. Note that the language can not only be chosen for all questions by using the control, but also individually for each question.

Other beans can be included in a questionnaire as long as they are serializable. They are recognized as question type, if they implement the `Question`



Q02

Apres l'execution du code snippet, quel valeur de x on a?

```
int x = 6;
if ( x /= 6 ) {
  x = 1;
} else {
```

50

Figure 8.10: Integer value question.

interface. Specialized beans for user identification and a timer are normally included in an exam to get the student name and to constrain the time of the exam.

8.5 Conclusion

This approach uses also the introduced two-step approach for tailoring. A groupware system is tailored by inserting customized component instances on demand of a user without halting the system. These objects are distributed to all local groupware applications, which insert them at the defined plug-points.

In contrast to the approach of chapter 7, this approach can only insert objects for already known classes. However, inserting stateful objects is very important to ensure that all local groupware applications have the same state for a specific cooperation. This approach addresses well the needs for remote instantiations.

Distributing stateful objects instead of code has the advantages that the frameworks and their plug-points are easier to implement and that the code of the objects is already known thus reducing potential security risks.

As shown by the example, this approach integrates well with the use of Customizers (see chapter 6). By using Customizers, components with many encapsulated instances can be easily configured. The component with its contained configured objects is then distributed as one object and remotely inserted.

Chapter 9

Tailoring support within frameworks

As introduced in chapter 5, groupware frameworks offer plug-points to plug domain-specific components. The application designer or a power-user can extend the framework at design-time by adding such components to the framework; the framework can also be extended dynamically at run-time as shown in chapters 7 and 8.

This chapter focuses on tailoring the behavior of a groupware application by offering the end-user the means to select the components, which process specific events. This tailorability goes beyond the abilities of tailoring only the appearance of the application (e.g. setting the color preferences for the application); tailoring of the presentation objects is normally directly supported by these components themselves and is not further discussed here.

This chapter introduces a generic design of a Connector component, which allows groupware frameworks to offer a default user interface. End-users can adapt with this user-interface the control flow in their local instances of the groupware application. I also discuss how this Connector components can be implemented so that they are independent from extensions of the framework by allowing to pass new event types, which have not been known at the time when the groupware framework has been designed and implemented.

9.1 Contribution

- Introduction of my design of a Connector component for groupware frameworks, which offer the end-user tailoring facilities to change the component composition at run-time.
- Discussion of implementation issues: Using introspection and reflection to discover automatically tailoring opportunities depending on the current state within the CSCW system.
- Presentation of an example, which uses filter components in an awareness service framework to dynamically adjust the event processing to the user's needs on demand.

9.2 Design of tailoring support within frameworks

Black-box frameworks are extended by composition. The binding between the framework and its extending components is deferred until run-time.

My approach benefits from this late binding by offering the user the means to control this binding dynamically at run-time. Thereby, this approach is independent from the extension mechanism: whether the framework was extended at design-time (see chapter 5), which results in binding the components in the initialization phase, or later by one of the approaches given in the chapters 7 and 8.

9.2.1 Tailoring support

Tailoring support beyond the customization of presentation objects often involves end-user programming (e.g. scripting) [Mør97a]. Some groupware systems, which rely on message exchanging in textual form (e.g. email and news readers), already allow to set visually rules to filter, sort, and process incoming messages (e.g. Information Lens [MGL+88] and Netscape's Messenger). Oval [MLF95] is another example of a radically tailorable tool, which uses agents to let the end-user define the rules how semi-structured textual messages are processed.

In the Java Beans component model, events define the interaction between components [Ham97]. I present in this chapter an extension of these ideas by allowing the end-user to define rules for the processing of events in a component-based groupware application. This approach is thus applicable on all levels and well integrated in the multiple purpose language Java and its component model Java Beans.

My approach to support run-time tailoring beyond the appearance level for component-based groupware systems let the end-user apply visually rules to the event-flow in the application. The end-user decides at any time, which component shall receive which events. So, the end-user can manipulate how events are processed, which can result in a radical dynamic reconfiguration.

9.2.2 Overview

My approach places a Connector component within the framework, which offers the end-user the means to dynamically decide, which event is forwarded to which beans. On the implementation side, this means that the Connector manages the event flow by adding or removing beans as listeners for a specific event type.

In the case that the framework developer already knows all components (e.g. because the components reside within the framework), the implementation of this approach is straight-forward. The developer just needs to add a user-interface, which offers the means to enable or disable a component. This chapter will not further discuss this case.

Instead, I present here an approach, which discovers at run-time the beans, which are added to the framework and which are potential listeners. Furthermore, this approach allows to control the event processing of event types, which have not been known at the time, when the framework was designed and developed.

This approach let the user select the event flow in the local groupware application. In a tele-tutoring example, the tutor may want to tailor the application to use an additional notification component, such as a bell, which is triggered, if a student requests help. Or the tutor wants to get alarmed by the groupware system, if a student is late with answering a prior distributed questionnaire (see also the example below).

9.2.3 Design and implementation

This general solution uses a Connector component, which is placed in the groupware framework.[1] The Connector actively manages a mapping table between event types and listener components. The Connector is specialized in a specific event type, but accepts also subclasses of this event type.[2] Every time a new component is added to the groupware framework, the component is automatically analyzed and the event types of its event set are forwarded to the Connector, which updates its mapping table. Similarly, the Connector updates its table with the accepted event types of the listener methods of each newly added listener component.

A component is typically added to the groupware framework at design-time by the groupware integrator or a power-user. These users are supported by a visual builder tool; adding a component to the framework with a Connector is similar to the activity of adding a component to a framework, which offers a plug-point as described in chapter 5. However, a component can also be added later with the introduced approach of chapter 7. Since the Connector analyzes the component in that moment, when the component actually registers with the framework, from the viewpoint of the Connector, it makes no difference, whether this registration method is called at initialization time or at the time, when a new component is inserted later at run-time by the end-user.

The user-interface displays the potential events and the matching listener components. The end-user manipulates with the graphical user-interface the mapping functions. The user-interface should show only those listeners, which match a given event to prevent forbidden combinations. The design of the Connector follows here the Model-View-Controller design pattern [BMR$^+$96]: the mapping table is the Model, the user-interface contains the View and the Controller.

[1]Of course, instead of the direct integration of the Connector component within the framework, the framework can just use my plug-point approach from chapter 5.2.2 to define the Connector interfaces. Thus the framework can be extended with different Connectors, which might implement different strategies. In fact, the AS Framework example uses this approach. Here, however, I try to keep things as simple as possible.

[2]This restriction results from my implementation approach and is not from principal nature. Since the implementation uses introspection to discover potential event types of senders and also to discover matching listeners, this restriction allows to take only events and listeners in account, which are meaningful for the framework. This restriction forces the developer to use inheritance for new event types, but on the other side, the superclass can be abstract and empty – thus being just a marker.

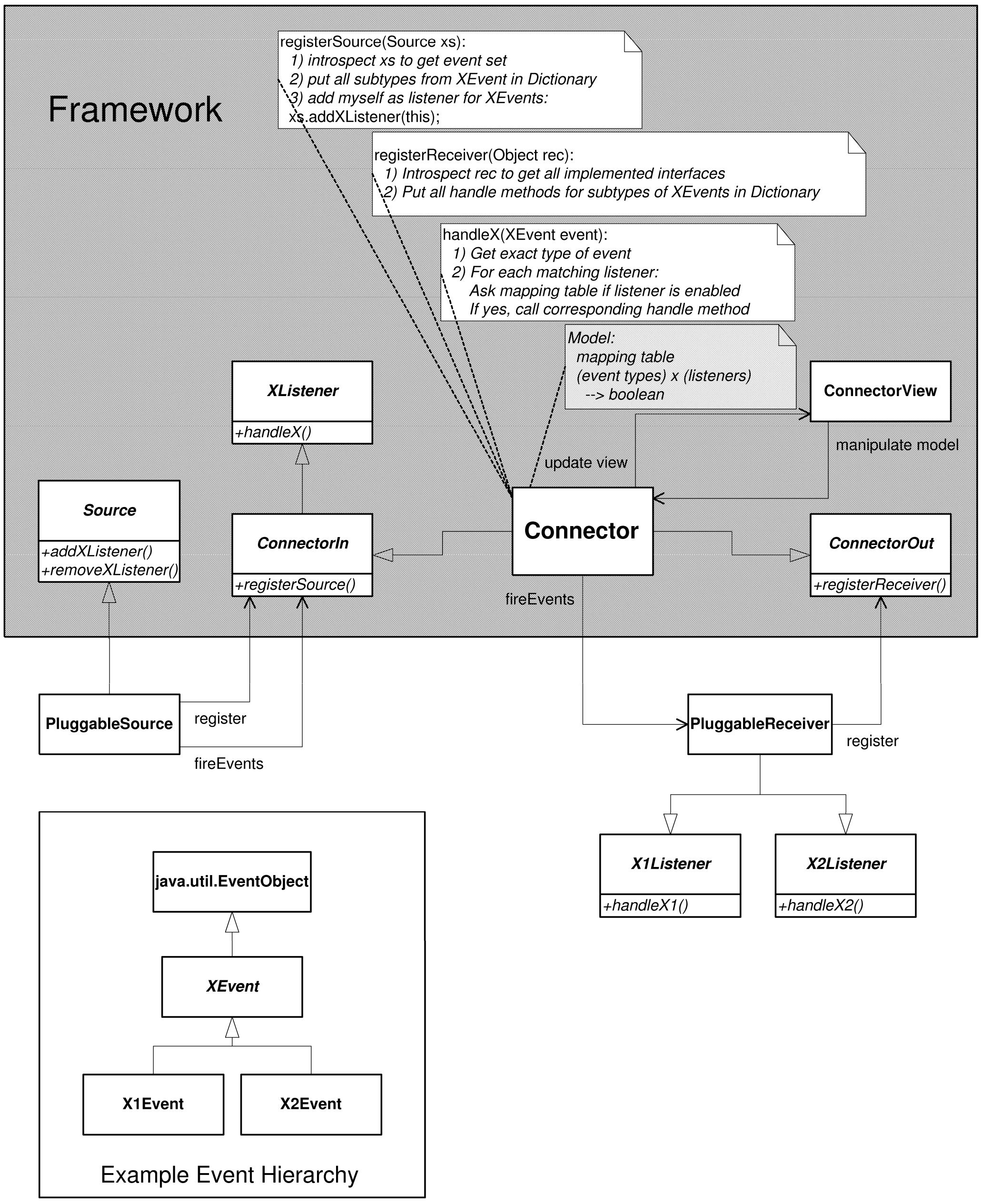

Framework
registerSource(Source xs):
1) introspect xs to get event set
2) put all subtypes from XEvent in Dictionary
3) add myself as listener for XEvents:
xs.addXListener(this);
registerReceiver(Object rec):
1) Introspect rec to get all implemented interfaces
2) Put all handle methods for subtypes of XEvents in Dictionary
handleX(XEvent event):
1) Get exact type of event
2) For each matching listener:
Ask mapping table if listener is enabled
If yes, call corresponding handle method
Model:
mapping table
(event types) x (listeners)
--> boolean
XListener
+handleX()
ConnectorView
manipulate model
update view
Source
+addXListener()
+removeXListener()
ConnectorIn
+registerSource()
Connector
ConnectorOut
+registerReceiver()
fireEvents
PluggableSource
register
fireEvents
PluggableReceiver
register
X1Listener
+handleX1()
X2Listener
+handleX2()
java.util.EventObject
XEvent
X1Event
X2Event
Example Event Hierarchy

Figure 9.1: Design of tailorable framework with a Connector.

Figure 9.1 shows the design of a framework, which incorporates a Connector component. The design allows to distribute arbitrary events, which are subtypes of a given event type (in the example: **XEvent**), from sources to receivers. The end-user, however, can manipulate which events will actually be delivered to which components.

The design splits the interface of the **Connector** component into two interfaces: one for the source (**ConnectorIn**) and one for the receiver (**ConnectorOut**). This design anticipates its use in groupware frameworks, where sources and receivers are usually placed on different hosts. In such groupware frameworks, proxies will forward the needed information to the Connector component.

The **Connector** manages a mapping table of all supported event types (including newly through introspection discovered types) to all registered listeners. The mapping table holds all potential matching pairs. The **Connector** also maintains within the model, if such a pair is enabled or not. The end-user can manipulated this model with a view component (**ConnectorView**).

To support this functionality also for event types, which the designer of the framework has not known, the implementation relies on introspection capabilities of the Java Beans component model. Since the components, which can be plugged into the framework, may also fire or receive event types, which should not be distributed by the framework (e.g. AWT events), I propose the use of a special abstract event class as marker from which the event types are inherited (in the example: **XEvent**), which are accepted by the framework. The introspection process can thus automatically discard all event types, which are not subclasses from this abstract event class. An alternative design would mandate to supply all event types as parameters in the registration process.

Whenever a new source registers itself, the Connector processes this registration as follows:

1. The Connector introspects the source component to get its event set, i.e. all event types, which the source can potentially fire. The introspection process includes also all superclasses.

2. Each discovered event type is stored in a Dictionary, if it is a subtype of the event type, for which the framework takes responsibility. The Dictionary holds only one entry for each type (it is often implemented as a HashTable).

3. The model is updated.
4. The Connector adds itself as listener for the abstract event type from the source component.

When the source component fires an event, it calls the handle routine of the abstract class of the **ConnectorIn** interface. The source component casts the actual event to the abstract type prior to this call.

Whenever a new receiver registers itself, the Connector processes this registration as follows:

1. The Connector introspects the listener component to get all implemented interfaces.
2. Each interface, which is an event listener interfaces for a subclass of the supported event type, contains the corresponding definition for the handle method. This handle method is entered in a Dictionary.
3. The model is updated.

When the Connector receives an event in its handle method for the abstract event type (in figure 9.1: **handleX**), the Connector performs the following processing:

1. The Connector get the actual type of the event by using the reflection capabilities of Java.
2. The Connector looks up the mapping table to get a list of all listeners, which accept this event type.
3. For each matching listener, the event type asks the model, if the event should be forwarded to the listener or if it should be discarded.

The model holds all pairs of matching event types and listeners. When queried, the model returns *true* or *false* for a given pair, indicating whether this listener is enabled. Note, that the model can also delegate the decision to another component by implementing the Strategy pattern [GHJV94].

The end-user can manipulate the model by using a view component. Usually this view component shows a graphical representation of the active mapping, e.g. it will show which components are enabled for which components.

Whenever the model changes due to the registration of new sources or receivers, the view is updated accordingly to reflect these changes.

9.2.4 Alternatives in distributed environments

In a groupware framework, sources and receivers are distributed. The Connector can be placed in the instance of the framework for the sources, and in the instance for the receivers. The framework developer chooses the place according to the users, who should be supported by this tailoring facility.

The Connector is included in the receiver's framework, if the end-users need to tailor only the behavior of their local groupware application. The events, which are distributed by the sources are all received by the Connector and the end-users may choose those components, which finally process these events. For this configuration, the interface of the Connector for the sources (**ConnectorIn**) is implemented by a proxy, which analyses the registered sources and send their event sets to the Connectors. Whenever the sources fire an event, this event is simply forwarded to the Connectors.

The Connector is included in the source's framework, if the end-user needs to tailor which event types should be distributed within the framework. This allows blocking event types, which carry sensible information, and can be used to ensure privacy needs. However, this configuration works only, if the correct function of the framework and its extending components is independent from the distribution of all event types; i.e. if the semantic of one event does not depend on a previously distributed event. For this configuration, a Connector is placed in the source part of the framework, which offers the end-user a user-interface for tailoring; another Connector is embedded within the receiver part, which does not offer an interface to the outside: this Connector simply forwards all incoming events to the connected components.

A groupware framework can include a Connector for the source part and a Connector for the receiver part to allow both tailoring forms. Actually, the during this thesis implemented awareness service framework uses a Connector in its source part to filter sensitive events and a Connector in the receiver part to forward the events to the receiver components. This framework is described in the example below.

9.3 Applicability and needed skills

The design and implementation of my approach for tailorable frameworks is very demanding for the framework developer.

It becomes, however, much less complex, if the developer decides to allow only the management of event types, which are known at the time, when the framework is developed, since then the implementation does not need to rely on introspection. On the other hand, it restricts the extensibility of the framework. This design decision must be carefully made.

The goal of my approach is to give the end-user the possibilities to tailor the behavior of the framework. At the same time, this approach allows that the framework is extended at run-time.

There is a danger that the end-user has difficulties to understand the functioning of the framework, if too many event types and listeners are managed by the Connector. To prevent this problem, I propose to create an abstract event class for semantically different events, and only use the same abstract class, if the events are used in the same context. It is perfectly legitimate to use more than one Connector component to separate unrelated events and listeners.

The mapping function, which decides whether a specific event should be delivered to a given component, can be arbitrary complex. Instead of basing its decision solely on the event type, this function can also test the values of contained variables. This allows to trigger some components only if a certain semantic is present.

In a groupware framework, the design typically includes a Connector at its receiving part. The end-users may thus tailor their local groupware application, but there is little risk that they may break the cooperation of other users by disabling a local components (unless this is a groupware component). Placing a Connector in the sending part requires more caution, since the suppression of an event type will affect all receivers. This is only useful, if such events do not initiate global state changes, which are needed for the processing of further events.

9.4 Example: Awareness service framework

This example introduces design and implementation issues of the Awareness Service Framework.[3] The Awareness Service Framework uses the Connector component to filter awareness events at sources and receivers. This groupware framework can handle arbitrary types of awareness events.

[3]The Awareness Service Framework was implemented by Verena Fastenbauer [Fas99].

9.4.1 Motivation

Awareness is a key requirement for cooperation [GGC96]. Within groupware systems, spatially dispersed cooperating users must be notified about actions of others. Apart from a notification mechanism, an awareness service should offer a filtering mechanism. Since in a personalized setting, different users want to be notified in different ways, each user may tailor the filter to personal needs. For efficient cooperation the notification settings must be tailorable at any time.

This example introduces a tailorable groupware framework, which offers a general awareness service. This distributed framework uses the notion of awareness events, which are passed by producing components to consuming components. The end-user tailors the appearance and behavior of the framework with the help of filter components. Filters provide the means to select at run-time the components, which compute and present the awareness data. This section focuses on the design and implementation issues of the extensible awareness framework, which supports tailoring functionality at run-time. I present also an example, which shows how an end-user uses the filter to tailor the framework's behavior at run-time.

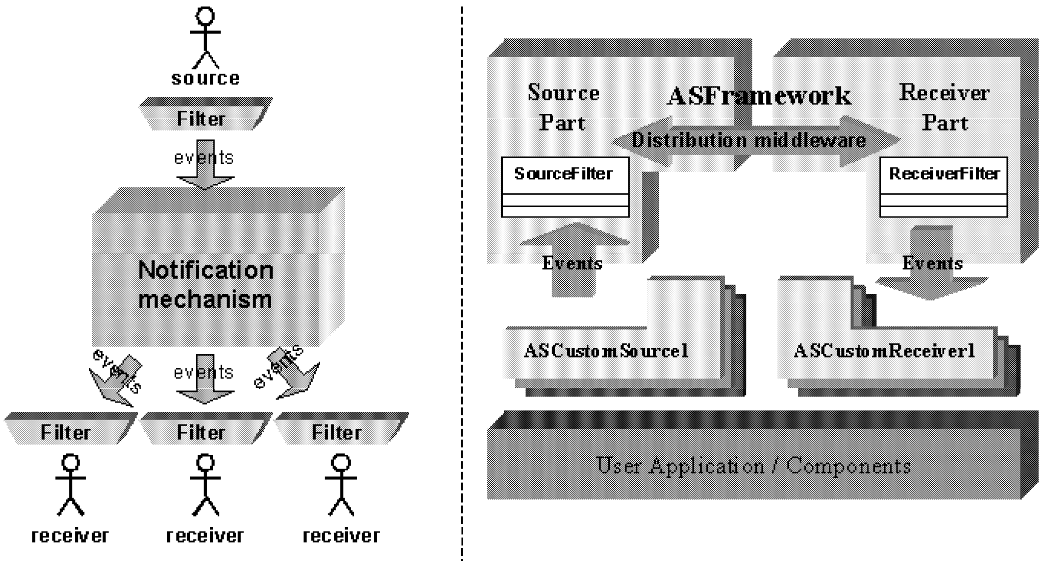
In the following, I use the short notion **AS Framework** for the complete name **Awareness Service Framework**.

Awareness in groupware systems means that the group members gain the needed information about the others to perform their work. I use the notion of awareness events, which are produced by components at the source and consumed by other components at the receiver. The main intent of the AS Framework is to offer the enabling infrastructure to support awareness among a distributed group of persons.

The AS Framework uses the notion "filter" instead of Connector component, since the anticipated use of the tailoring facilities is more likely to suppress the forwarding of events to its processing components than to connect additional components. Filtering is used on both ends of the AS Framework: suppressing the publishing of events at the sources to ensure privacy needs, and suppressing the processing of the events at the receiver side to prevent information overload.

9.4.2 Design issues

A high-level use case illustrates the requirements of the AS Framework (figure 9.2(a)). Two different actors are distinguished: Sources and Receivers. Components at the sources emit awareness events. Filters can be applied by the user to ensure privacy. The events are then distributed to all receivers. Users set filters to receive only those events, they are interested in. The users specify the components which finally process the events. So, different presentations are achieved by end-user tailoring. The AS Framework itself is independent of the actual transmitted events. It does not have to be changed for different scenarios.



(a) Illustration of a high-level use case

(b) Structure of the customized AS Framework

Figure 9.2: Overview of the AS Framework

Since the AS Framework offers a general awareness service, future event types cannot be anticipated. Each specific problem domain will have its own awareness events. Components which support those events can be plugged into the framework without changing the AS Framework itself. The component developer uses the offered abstract event type to create new awareness events through inheritance.

Figure 9.2(b) depicts the event flow. After the registration of the customized source components to the AS Framework, awareness events are accepted by the AS Framework. Using a distribution middleware, which is accessed in the current implementation with the already introduced group communication components, the framework passes the events to every receiver part of the AS Framework. At the receiver part, the events are passed to the customized receiver components with regard to the filter's setting.

Design-time customization covers the creation of the problem specific event and listener types. In addition, the source and receiver components of those events are created. These components are plugged into the AS Framework and integrated into the user application.

9.4.3 Implementation issues

Java Beans offers introspection to analyze components at run-time. In addition, the reflection API of Java enables very late binding of method calls by looking up interfaces and methods at run-time [DW98]. This allows to keep the framework invariant from the plugged domain specific components.

The awareness events may be filtered at the source and at the receiver side. The filtering mechanism at the source side guarantees the assurance of privacy needs. The filtering mechanism at the receiver side covers the ability to select the events and to change the processing of the events. In the following, this example concentrates on the filtering mechanism at the receiver side.

The implementation of the filter component follows my approach for design and implementation of the Connector component.

The standardized event model in conjunction with the introspection mechanism of Java Beans allows to offer the filtering mechanism for events without knowing their specific type at design-time of the framework. The filter dynamically manages the connections between the interested listeners and the AS Framework. In addition, the filter ensures that the end-user is not overwhelmed with all events that may occur. Only those awareness events which may be emitted by the sources are considered. Using the filtering mechanism, the user decides, how the events are processed during run-time.

The filter stores the by introspection discovered event types and the interested receivers in a dictionary. One entry in the dictionary consists of a key, which is the event type, and a vector containing the references to all interested receivers. The main task of the filtering component is to manage the entries of the dictionary. Each entry reflects one filter setting. Every change to the filter settings causes an update of the receiver vector. Furthermore, the filter vetoes the attempt of incorrect settings.

Apart from the management of the filters settings, the filter is responsible to process the awareness events properly. Whenever an awareness event arrives, the filter queries its dictionary and forwards the event to all interested receivers. Although the receiver components differ, they work with the same filtering mechanism.

In order to ensure that the filter considers only those events which can be emitted by a source component, the filtering mechanism must be supplied with all types of potential awareness events. Therefore, the event types of the source components are extracted during their registration process. This data are transmitted to the receivers. Referring to late-comers, a trade-off has been made in the current implementation. To ensure the tailorability of the filter, the sources are queried about their supported event types. However, the current implementation does not supply late comers with those events that have already been distributed.

9.4.4 Extension of the AS Framework with domain-specific components

Figure 9.3 shows, how a developer extends the AS Framework at design time. This example assumes that the receiver components are already developed and the event types are defined.

The developer drops in a visual composition editor for Java Beans the proxy for the framework (*ASFrameworkProxy*), a filter (*ASReceiverFilter*), two receiver components (*ASLogFile* and *ASDialogAnnouncer*) and the viewer for the log file. The model of the log file is attached to the corresponding view (1) and plugged into the AS Framework (2). In order to offer the facility to be informed immediately of certain events, additionally a dialog announcer is attached to the AS Framework (3). The receiver filter is plugged into the the AS Framework (4) to provide selection at run-time. These activities suffice to add two new receiver components among their filter to the AS Framework at design-time.

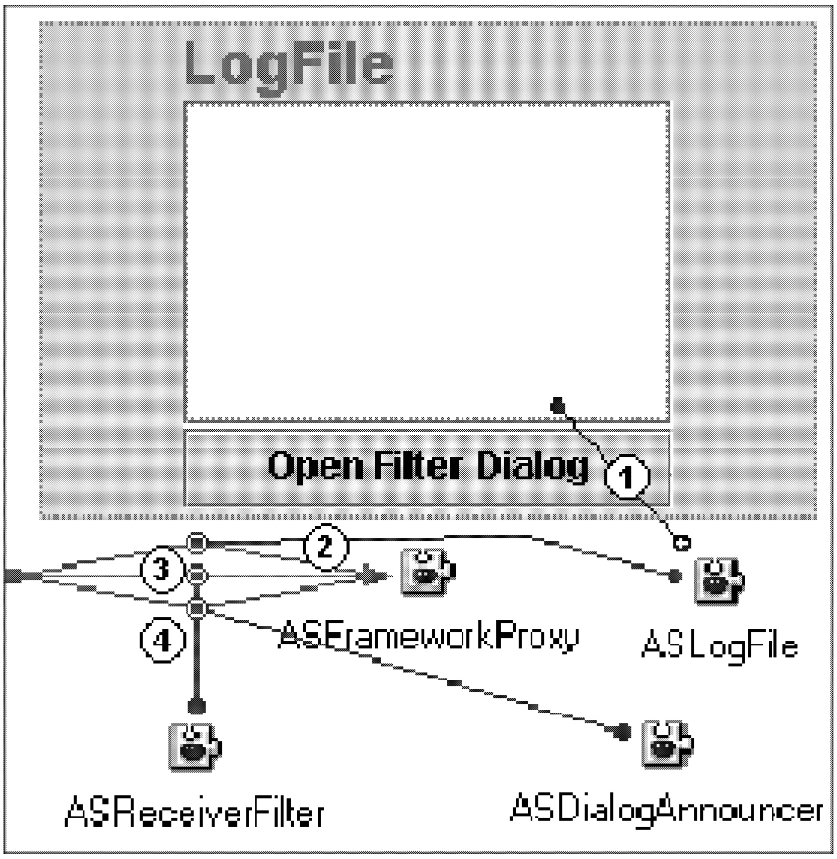


Figure 9.3: Extension of the AS Framework with a log file at the receiver side.

9.4.5 Tailoring example for a laboratory course

The following example combines the tele-exam example (see chapters 6.4 and 8.4) with the AS Framework to support a situation where a professor supervises students doing exercises with the questionnaire tool, and the professor needs some feedback on how the students are performing to provide assistance to the slowest. Such situations typically arise in laboratory courses, where a pre-test is given to the students to evaluate their knowledge about the topic of this session.

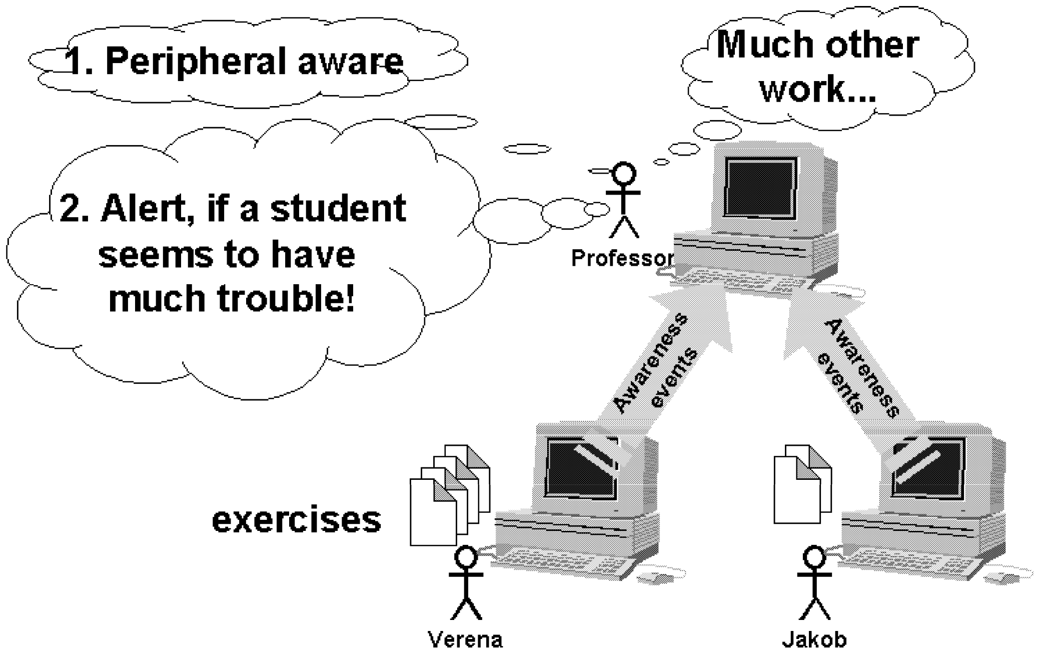
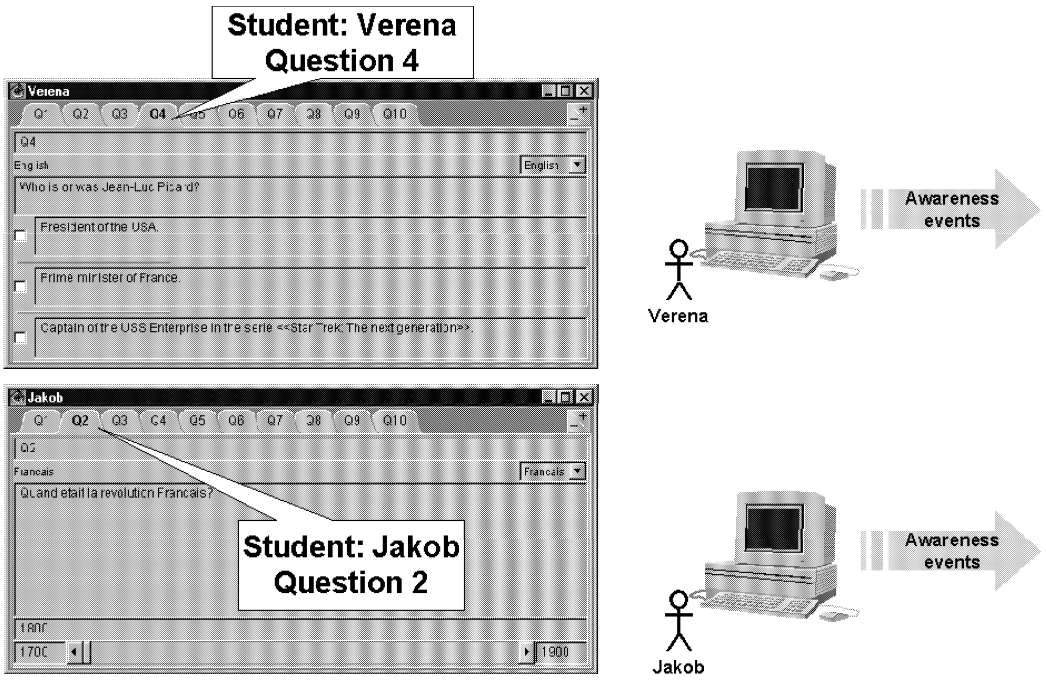


Figure 9.4: Setup of example: Use of the AS Framework to monitor student progress.

Figure 9.4 introduces my example: Two students are answering a questionnaire, which has been previously distributed by the professor. Student Verena is currently working on the fourth question, while student Jakob is still struggling with the second question. The professor has opened a monitoring tool, which shows the progress of the students, but the professor is only peripheral aware about the students, since he is doing at the same time other work.

The example will show how the professor tailors this setup so that not only the monitoring component receives awareness events, whenever a student has completed a question, but alarms the professor in the case that a student surpasses a time limit, which is associated prior with each question.



Q4

Who is or was Jean-Luc Picard?

President of the USA.

Prime minister of France.

Captain of the USS Enterprise in the serie <<Star Trek: The next generation>>.

Q2

Quand etait la revolution Francais?

1800

1700 1900

Figure 9.5: Current state: Verena works on question 4, Jakob on question 2. The questionnaire component emits awareness events, when a student switches the question or did not proceed within a given time

For this example, I have combined the Questionnaire component with awareness source components: Each time, when a student switches to the next question, an awareness event from the type **QuestionPerformed** is sent. Additionally, a timer component is associated with each question, which sends a **TimeElapsed** event, if the student has not been proceeded within a given time. Figure 9.5 shows the questionnaires for the two students.

Figure 9.6 shows the current setup of the professor's groupware application. Only events from the type **QuestionPerformed** are currently processed; they are passed to the monitoring tool.

This setup suits the professor's goal to glance from time to time on the performances of the students, but not yet the goal to be alarmed, when a student is late.

Figure 9.7 shows how the professor uses the default graphical user-interface

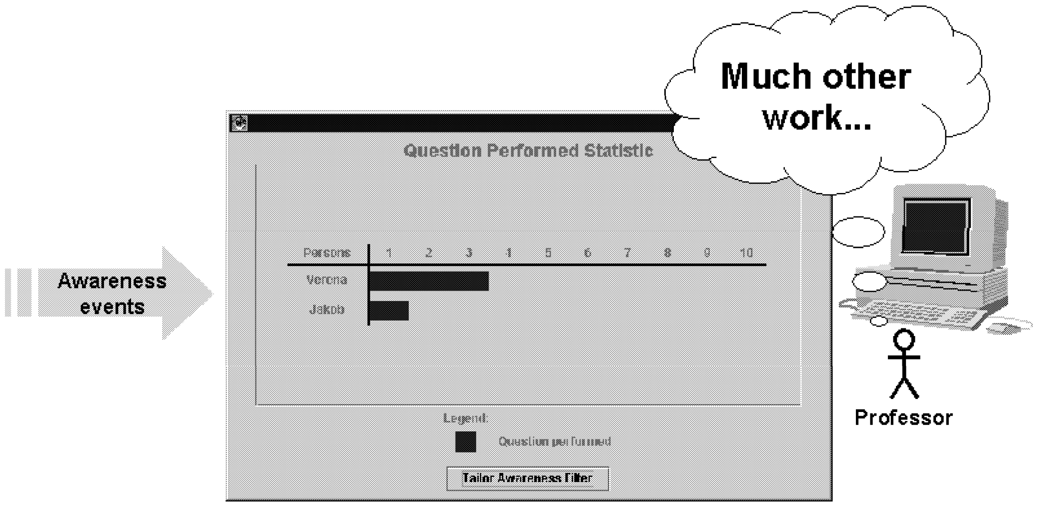


Figure 9.6: In the current setup, the professor receives only awareness events from the type **QuestionPerformed**, which are passed to the monitoring tool.

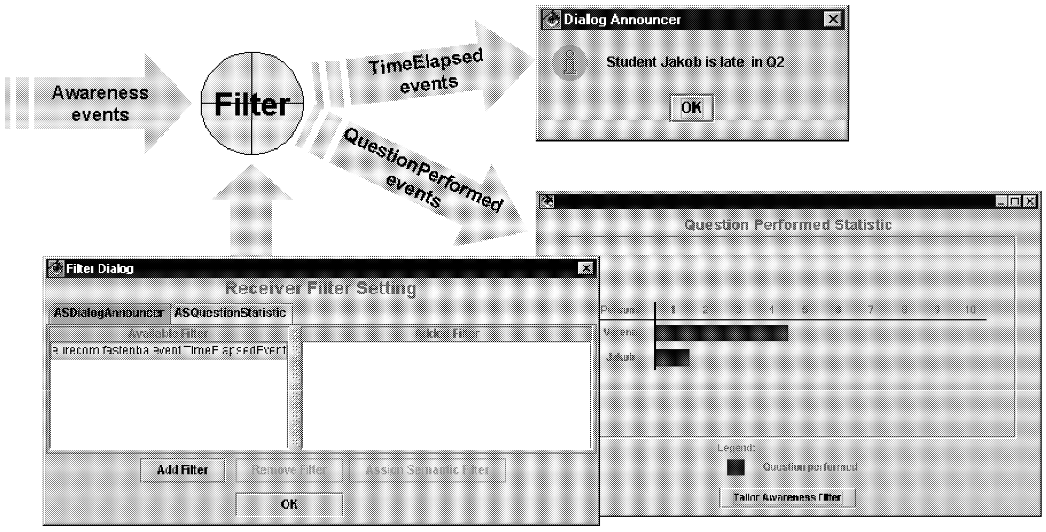


Figure 9.7: In the current setup, the professor receives only awareness events from the type **QuestionPerformed**, which are passed to the monitoring tool.

of the filter component to tailor the event processing. The professor instructs the filter to forward **TimeElapsed** events to the component **ASDialogAnnouncer**. This component pops up, when an event arrives, and thus alerts immediately the professor.

In this example, student Verena has currently completed question 4 resulting in the processing of the **QuestionPerformed** event by the monitoring tool. Student Jakob, however, has exceeded the time limit for question 2 resulting in a **TimeElapsed** event, which is caught by the **ASDialogAnnouncer** component to warn the professor.

9.5 Conclusion

Tailoring support can be included in the design and implementation of a framework. My approach presented in this chapter is to allow the end-users to tailor the event flow in their local groupware applications.

This approach uses a Connector component, which is placed with the framework and which controls the event flow. The Connector component offers the end-user a user-interface to visually map event types to components. Whenever an event from the given type arrives, the from the user specified component is triggered, i.e. the event is forwarded to this component.

The presented design defines one event type for which the Connector is specialized. The Connector handles all event types, which are subtypes of this event type. However, this design does not require that the Connector knows all potential sub-types. Instead it relies on introspection to extract the event type during run-time. So, a framework with an embedded Connector component does not to be changed, when in future applications events from other sub-types must be handled. This design maximizes the reusability of such a framework.

Chapter 10

Conclusions

10.1 Conclusions

Organization of work is not static, but changes frequently. Groupware, which supports the cooperative work activities among people, must be extensible to reflect a potential reorganization of work processes. Already the design of groupware must anticipate future changes and facilitate adaptations. Since work practices differ among companies and institutions, groupware systems must be customizable. The design and implementation of a groupware system should offer different user categories the means to modify the system and its parts.

Customization and tailoring of groupware is identified as a major research topic in CSCW literature. This thesis underlines the connection between customization and reuse practices, which are mainly researched in the software engineering domain. So, this thesis applies software engineering concepts – object-oriented design, framework design, component-based programming in connection with visual builder tools, and design patterns – to provide concepts to build customizable and tailorable groupware systems.

This thesis intentionally uses a component model, which is supported by various integrated development environments (IDE). I have chosen Java Beans, which is the component model for Java, since Java compiles to platform independent code and the components can thus be deployed on multiple platforms. Furthermore, by using a widely used multi-purpose programming language for their implementations, the concepts of this thesis become applicable for other groupware developers.

My approach is to let the user choose her or his favorite IDE to customize component-based applications on all levels. Customization is anticipated by all components and frameworks, which are introduced by this thesis. By offering customization means for these building blocks only through Java Beans compliant mechanisms, the user in independent from the actual used tool. Furthermore, this approach invests in the future, since the development tools are constantly improved from the tool vendors or open source communities. On the other side, this approach prevents from a tighter integration of customization within a specific IDE by calling by calling its API. This strategy also prohibits the creation of an own component model for groupware components. In my opinion, the advantages of using off-the-shelf IDEs compensate potential advantages by defining a new groupware model.

This thesis introduces the concept of groupware components and frameworks. These building blocks are highly reusable, and can be composed to create new groupware systems. To customize component-based groupware applications, the user can decompose the applications on all levels. The introduced groupware frameworks offer plug-points, which are extended by components to provide domain-specific adaptations. The black-box design of the framework provides that they can be easily extended within visual builder tools, but support also the dynamic extension at run-time. The thesis provides design patterns for groupware frameworks, which support the distribution of components to remote applications, where they are automatically inserted within the running application.

This thesis proposes a two-step approach to tailor a running groupware system. First the user customizes components visually in an IDE. Second the user instruct the local application to insert these components; the system then distributes these components to all local groupware applications, where they are loaded and integrated.

This thesis discusses the advantages of the two-step approach for tailoring over embedding tailoring functionality in each groupware framework and component. Although the two-step approach is general and can be used in many settings, specialized tailoring functionality can and should be placed in some groupware frameworks. The approach of tailoring the event flow at run-time is one example for embedded tailoring functionality.

10.2 Summary of contributions

This thesis provides solutions to design and implement reusable frameworks and components for groupware, which support different user categories with the means to customize and tailor groupware systems.

The contribution of this thesis is in the area of computer supported cooperative work and software engineering, and consists of the following main parts:

1. *Motivation to use software engineering concepts in groupware*: the thesis motivates to use software engineering design concepts, such as frameworks, components, and design patterns to design groupware systems, which can be customized at design-time and tailored at run-time to the company's and user's needs.

2. *General design for groupware components and frameworks*: this thesis presents a general design for distributed groupware components and frameworks. Both, groupware components and frameworks, can define their own interaction protocol based on the distribution of events. The event distribution is uniformly handled by specialized group communication components. The introduced design concepts focus on reusability through customizing and extension. The thesis also presents customization wizards, which are used to offer the end-user a convenient user-interface to customize components. By relying only on the standard Java Beans component model, users can customize the components and their compositions with any available visual builder tool for Java Beans.

3. *Two-step approach of tailoring:* the thesis invents a new approach of tailoring groupware applications by dynamic extensions. The two-step approach allows the users to customize in a first step components with their favorite tools for the Java Beans component model. In the second step, the user inserts the customized components into the local groupware application, which distributes them to all remote applications, where they are integrated within the running groupware applications.

4. *Code and object distribution*: the two-step approach relies on the possibility to distribute components over the network and to insert them dynamically within the running local applications. This thesis presents and discusses two different approaches. One approach uses the distribution of code to insert newly developed components into the remote

applications. The other approach distributes stateful objects to instantiate a cooperation.

5. *Design for tailoring support*: the thesis introduces a framework design, which allows the end-user to manipulate the event-flow in the groupware application, and thus to tailor its behavior. The design anticipates future extensions of the framework. This thesis also discusses the implementation issues which arise when the framework must serve new event types. By using the reflective power of Java, this thesis presents a solution, which enables that the framework stays invariant, but can also be extended both at design-time and at run-time.

6. *Discussion of applicability*: the thesis discusses each concept to show, where the concepts are applicable and which skills a user needs to use a given concept. Since one goal of this thesis is to provide all user categories during the life-cycle of a groupware system with means to customize the system, the theses relates the needed skills for a specific customization concepts to the expertise of a user category.

7. *Proof of concept implementations:* this thesis comprises examples, which present actual implementations, which use the prior introduced concepts. The examples serve to highlight the concepts; some other implementations are omitted from this document.

10.3 Future Research

In this section, I give a short overview about further research topics, which I have identified during my thesis work. These topics, all related with this work, include not only a more formal evaluation of this work, but also recent trends in researching patterns and distributed programming.

10.3.1 Evaluation of experiences

The presented thesis concentrated on the concepts of designing customizable and tailorable groupware and on implementation issues, which became apparent during the development of the prototypes. Experiences with actual customization of the presented components have been limited. Customization experiences were made mainly by the developers of prototypes including myself.

Students, which have stayed with Eurécom for an internship, and last year students have used the introduced groupware frameworks and components to create new ones. The projects included the creation of components for tutoring [Koh97], for an awareness service [Fas99], audio and video beans [VHJ98], and beans for controlling Eurécoms Mediaspace [FK97, FPO97]. All students have appreciated working with component technology and their highly motivated work has resulted in some publications on international conferences and workshops [HKM98b, HKM98a, FHM99]. All students have reused prior developed components and their constructive critics have been incorporated in newer versions of these components to provide higher reusability.

I used prior developed groupware components to build simple groupware systems, which were used within two distributed laboratory courses between Eurécom and the University of Linz in the years 1997 and 1998. Building these prototypes, I have noted that the developed groupware components are in fact reusable and customizable.

The next step would be to conduct a formal evaluation of the approaches of this thesis in a real context. Given that all necessary resources are available, the concepts introduced by this thesis should be evaluated as follows. Groupware components are developed by a team of developers, while another team of domain-specific experts, which are familiar with the use of visual builder tools assembles them to support different scenarios. The different forms of tailoring are tested by end-users, who are given specific problems to solve with tailoring. It is crucial in this set-up that the members of the groups do not overlap to deduce valid results.

10.3.2 Patterns

This thesis introduced general design approaches for groupware components and frameworks, and design patterns for runtime extensible frameworks. These designs have evolved from generalizations of specific problems. Interesting would be to analyze the design of existing groupware systems and to capture similarities in cooperative design patterns.

An interesting research topic would be to analyze how people customize their work settings. Is it possible to generate patterns from their actions, which can be translated into design patterns for customizable groupware?

10.3.3 Directions in distributed programming

Recently, two new approaches for component-based distributed programming have emerged. The Open Management Group is standardizing its effort of an interoperable component model on top of CORBA. Sun has introduced with Jini a new model for distributed programming. Both efforts are aimed for different purposes, but further research on the topics of this thesis could benefit from either of them. Both efforts are not yet matured, and it remains to see, whether they will be widely accepted in the future.

The CORBA component model [OMG99] aims to standardize a component model for distributed programming, which is independent from the actual used programming language. If this proposal becomes a standard and widely accepted, this component model may facilitate the deployment of the concepts presented in this thesis. By relying on CORBA [MZ95] as distribution infrastructure, such a standard would especially facilitate the integration of groupware application with already existing software. A widely accepted language independent component standard may boost the component development efforts by third parties and thus create a larger market. Also, such a standard would have a positive effect on reuse practices.

Jini [Sun99], on the other side, introduces a new concept for distributed programming. Jini is based on Java and is designed to automatically discover services available in a federation on the network. Jini greatly relies on Java's ability to move code from service providers to service clients. Jini's programming model include leasing, transactions, and remote events. A Jini service can be written as a Java Bean. Jini's strength is the discovery mechanism together with code distribution, which enables to find easily services in a Jini federation and to use these services dynamically in the local application. Groupware systems can benefit from Jini by defining common groupware services, which are inserted in a Jini federation. A local groupware application would look up the existing services and use them when needed. Jini seems to be a good platform to be used in conjunction with my two-step approach for tailoring. Jini is especially interesting for further research on groupware applications for mobile users, who need to connect to groupware services in different environments from their mobile devices, such as laptops or Palm organizers.

Bibliography

[BD95] Richard Bentley and Paul Dourish, *Medium versus mechanism: Supporting collaboration through customization*, Proceedings of the fourth European Conference on Computer–Supported Co-operative Work (Stockholm, Sweden) (H. Marmolin, Y. Sundblad, and K. Schmidt, eds.), Kluwer Academic Publishers, September 1995, pp. 133–148.

[BMR$^+$96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal, *Pattern–oriented software architecture – a system of patterns*, John Wiley & Sons, Inc., 1996.

[Boc92] Geoffrey Bock, *Introdcution – groupware: The next generation for information processing?*, Groupware: Software for Computer Supported Cooperative Work (David Marca and Geoffrey Bock, eds.), IEEE Computer Society Press, 1992, pp. 1–6.

[Boe88] Barry Boehm, *A Spiral Model of Software Development*, IEEE Computer (1988), 61–72.

[Bro87] Frederick P. Brooks, *No silver bullet – essence and accidents of software engineering*, IEEE Computer (1987), http://www.virtualschool.edu/cox/Publications.html.

[Cox90] Brad J. Cox, *Planning the sofware industrial revolution*, IEEE Software magazine (1990), http://www.virtualschool.edu/cox/Publications.html.

[Cox95] Brad Cox, *No silver bullet revisted*, American Programmer Journal (1995), http://www.virtualschool.edu/cox/Publications.html.

[Dou95] Paul Dourish, *Developing a Reflective Model of Collaborative Systems*, ACM Transactions on Computer–Human Interaction **2** (1995), no. 1, 40–63.

[Dou96] Paul Dourish, *Open Implementation and Flexibility in CSCW Toolkits*, Ph.D. thesis, University College London, June 1996.

[DW98] Desmond F. D'Souza and Alan Cameron Wills, *Objects, components and frameworks with uml: The catalysis approach*, Object Technology Series, Addison-Wesley, October 1998.

[EN88] Clarence A. Ellis and Gary J. Nutt, *Office information systems and computer science*, Computer–Supported Cooperative Work – A Book of Readings (Irene Greif, ed.), Morgan Kaufmann Publishers, 1988, pp. 199–247.

[Eri94] Hans Eriksson, *MBONE: The multicast backbone*, Communications of the ACM **37** (1994), no. 8, 54–60.

[ES98] Peter Eeles and Oliver Sims, *Building business objects*, John Wiley and Sons, 1998.

[Fas99] Verena Fastenbauer, *Components for a generic awareness service*, Master's thesis, Eurécom and Johannes Kepler University of Linz, Austria, February 1999.

[FHM99] Verena Fastenbauer, Jakob Hummes, and Bernard Merialdo, *Design and implementation of a tailorable awareness framework*, Position Paper at the Workshop on Implementing Tailorability in Groupware, Int. Joint Conf. WACC'99, San Francisco, CA, February 1999, available at http://www11.in.tum.de/workshops/wacc99-ws-impltailor/.

[FK97] Wolfgang Fueg and Gregory Kuhlmey, *Srms java beans*, Tech. report, Eurécom, 1997, Student project.

[FPO97] Vincent Fayet and David Puig Oses, *Srms java implementation*, Tech. report, Eurécom, 1997, Student project.

[FS97] Mohamed E. Fayad and Douglas C. Schmidt, *Object–oriented application frameworks*, Communications of the ACM **40** (1997), no. 10, 32–38.

[GGC96] Saul Greenberg, Carl Gutwin, and Andy Cockburn, *Using distortion-oriented displays to support workspace awareness*, Research report 96/581/01, Department of Computer Science, University of Calgary, Calgary, Canada, November 1996.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design patterns – elements of reusable object–oriented software*, Addison–Wesley, 1994.

[Gib94] W. Wayt Gibbs, *Software's chronic crisis*, Scientific American (1994), 72–81.

[GMU96] Nicholas T.C. Graham, Catherine A. Morton, and Tore Urnes, *ClockWorks: Visual Programming of Component–Based Software Architecture*, Journal of Visual Languages and Computing (1996), http://www.cs.yorku.ca/People/graham/pubs.html.

[Gra95] Nicholas T.C. Graham, *The Clock Language: Reference Manual*, Tech. report, York University, 1995, http://www.cs.yorku.ca/people/graham/pubs.

[Gre88] Irene Greif, *Overview*, Computer–Supported Cooperative Work – A Book of Readings (Irene Greif, ed.), Morgan Kaufmann Publishers, 1988, pp. 5–12.

[Ham97] Graham Hamilton, *Java Beans 1.01 API specification*, Sun Mircosystems, July 1997, http://java.sun.com/beans.

[HBP+93] Ralph D. Hill, Tom Brinck, John F. Patterson, Steven L. Rohall, and Wayne T. Wilner, *The Redezvous Language and Architecture*, Communications of the ACM **36** (1993), no. 1, 63–67.

[HKM97] Jakob Hummes, Alain Karsenty, and Bernard Merialdo, *Active Annotations of Web Pages*, Voting, Rating, Annotation – Web4Groups and other projects: approaches and first experiences (Wien, München) (Roland Alton-Scheidl, Rupert Schmutzer, Peter Paul Sint, and Gernot Tscherteu, eds.), Schriftenreihe der Österreichischen Computer Gesellschaft, vol. 104, Oldenbourg Verlag, Wien, München, 1997.

[HKM98a] Jakob Hummes, Arnd Kohrs, and Bernard Merialdo, *Questionnaires: a framework using mobile code for component-based tele-exams*, Proceedings of IEEE 7th Intl. Workshops on Enabling Technologies: Infrastructure for Collaborating Enterprises (WET ICE) (Stanford, CA, USA), June 1998.

[HKM98b] Jakob Hummes, Arnd Kohrs, and Bernard Merialdo, *Software components for cooperation: A solution for the "get help" problem*, COOP'98: Third International Conference on the Design of Cooperative Systems (Cannes, France), May 1998.

[Joh96a] Philip Johnson, *Egret: A Framework for Advanced CSCW Applications*, ACM Software Engeneering Notes **21** (1996), no. 2, file://ftp.ics.hawaii.edu/pub/tr/ics-tr-95-23.ps.Z.

[Joh96b] Philip Johnson, *State as an Organizing Principle for CSCW Architectures*, Tech. Report ICS-TR-96-05, Collaborative Software Development Laboratory, University of Hawaii, March 1996, file://ftp.ics.hawaii.edu/pub/tr/ics-tr-96-05.ps.Z.

[Joh97] Ralph E. Johnson, *Frameworks = (components + patterns)*, Communications of the ACM **40** (1997), no. 10, 39–42.

[KA98] Daniel Krieger and Richard M. Adler, *The emergence of distributed component platforms*, IEEE Computer (1998), 43–53.

[KB95] Setrag Khoshafian and Marek Buckiewcz, *Groupware, workflow, and workgroup computing*, Wiley & Sons, 1995.

[Kic96] Gregor Kiczales, *Beyond the black box: Open implementation*, IEEE Software (1996), 8–11.

[Kie98] Don Kiely, *Are components the future of software?*, IEEE Computer (1998), 10–11.

[KM97] M. Kyng and L. Mathiassen (eds.), *Computers and design in context*, The MIT Press, Cambridge, MA, 1997.

[Koh97] Arnd Kohrs, *Development of a generic component based help request facilty for CSCW*, Master's thesis, Eurécom and University of Karlsruhe, Germany, December 1997.

[Mac90] Wendy E. Mackay, *Patterns of sharing customizable software*, CSCW'90: proceedings of the Conference on Computer-Supported Cooperative Work (Los Angeles, CA), ACM, October 1990, pp. 209–221.

[Maf97] Silvano Maffeis, *iBus – The Java Intranet Software Bus*, Overview paper, Olsen and Associates, 8008 Zurich, Switzerland, February 1997, http://www.olsen.ch/export/proj/ibus/.

[MGL$^+$88] Thomas W. Malone, Kenneth R. Grant, Kum-Yew Lai, Ramana Rao, and David Rosenblitt, *Semistructured Messages are Suprisingly Useful for Computer–Supported Coordination*, Computer–Supported Cooperative Work – A Book of Readings (Irene Greif, ed.), Morgan Kaufmann Publishers, 1988, pp. 311–331.

[MLF95] Thomas W. Malone, Kum-Yew Lai, and Christopher Fry, *Experiments with Oval: A Radically Tailorable Tool for Cooperative Work*, ACM Transactions on Information Systems **13** (1995), no. 2, 175–205.

[Mør94] Anders I. Mørch, *Designing for radical tailorability: coupling artifact and rationale*, Knowledge-Based Systems **7** (1994), no. 4, 253–264.

[Mør95] Anders Mørch, *Application units: Basic building blocks of tailorable applications*, Proceeding Fifth International East-West Conference on Human-Computer Interaction, Lecture Notes in Computer Science, vol. 1015, Springer, 1995, pp. 45–62.

[Mør97a] Anders Mørch, *Three levels of end-user tailoring: Customization, integration, and extension*, In Kyng and Mathiassen [KM97], pp. 51–76.

[Mør97b] Anders I. Mørch, *Evolving a generic application into a domain-oriented design environment*, Scandinavian Journal of Information Systems **8** (1997), no. 2, http://iris.informatik.gu.se/sjis/.

[Mør97c] Anders I. Mørch, *Method and tools for tailoring of object-oriented applications: An evolving artifacts approach*, Ph.D. thesis, University of Oslo, Norway, April 1997.

[MZ95] Thomas J. Mowbray and Ron Zahavi, *Essential CORBA – Systems Integration Using Distributed Objects*, John Wiley and Sons, Inc., 1995.

[Nel81] B. J. Nelson, *Remote Procedure Call*, Ph.D. thesis, Carnegie-Mellon University, 1981.

[Obj97] ObjectSpace Inc., Dallas, Texas, *Objectspace voyager – the agent orb for java – core technology user guide*, July 1997, http://www.objectspace.com/developers/voyager/white/index.html.

[Obj98] ObjectSpace Inc., Dallas, Texas, *Objectspace voyager. version 2.0.0 user guide*, 1998, http://www.objectspace.com/developers/voyager/white/index.html.

[OMG99] OMG, *Omg tc document orbos/99-02-05: Corba components*, March 1999, Joint submission. http://www.omg.org/.

[Ous94] John K. Ousterhout, *Tcl and the tk toolkit*, Addison-Wesley, 1994.

[Pre94] Wolfgang Pree, *Meta patterns—a means for capturing the essentials of reusable object–oriented design*, Proceedings of ECOOP'94 (Bologna, Italy), September 1994.

[RG96a] Mark Roseman and Saul Greenberg, *Building Real Time Groupware with GroupKit, A Groupware Toolkit*, ACM Transactions on Computer Human Interaction **3** (1996), no. 1, 66–106, http://www.cpsc.ucalgary.ca/projects/grouplab/papers/papers.html.

[RG96b] Mark Roseman and Saul Greenberg, *TeamRooms: Groupware for Shared Electronic Spaces*, ACM SIGCHI'96 Conference on Human Factors in Computing System, Companion Proceedings, 1996, http://www.cpsc.ucalgary.ca/projects/grouplab/papers/papers.html, pp. 275–276.

[RG97] Mark Roseman and Saul Greenberg, *Simplifying component development in an integrated groupware environment*, Proceedings of ACM UIST'97 Symposium on User Interface Software and Technology (Banff, Alberta), ACM Press, October 1997, pp. 65–72.

[RGJ96] Mark Roseman, Saul Greenberg, and Shannon Jaeger, *GroupKit User's Manual*, University of Calgary, Canada, 1996, delivered with sources version 3.1.

[RJ98] Don Roberts and Ralph Johnson, *Evolving frameworks – a pattern language for developing object-oriented frameworks*, Pattern Languages of Program Design 3 (Martin Robert, Dirk Riehle, and Frank Buschmann, eds.), Addison-Wesley, 1998.

[Rod99] Lawrence Rodrigues, *On javabeans customization*, Java Developer's Journal **4** (1999), no. 5, 24–28.

[Rog97] Dale Rogerson, *Inside com (microsoft's component object model)*, Microsoft Press, 1997.

[Rya97] Ivan Ryant, *Why inheritance means extra trouble*, Communications of the ACM (1997), 118–119.

[SC98] Oliver Stiemerling and Armin B. Cremers, *Tailorable component architectures for cscw-systems*, Proceedings of the 6th Euromicro Workshop on Parallel and Distributed Programming (Madrid, Spain), IEEE Press, January 1998, http://www.cs.uni-bonn.de/ os/, pp. 302–308.

[Sch95] Hans Albrecht Schmid, *Creating the architecture of a manufactoring framework by design patterns*, Proceedings of OOPSLA'95 (NY), ACM, 1995.

[Sch97] Hans Albrecht Schmid, *Systematic framework design by generalization*, Communications of the ACM **40** (1997), no. 10, 48–51.

[Sie96] Jon Siegel, *CORBA – Fundmentals and Programming*, John Wiley and Sons, Inc., 1996.

[SKW97] Oliver Stiemerling, Helge Kahler, and Volker Wulff, *How to make software softer – designing tailorable applications*, Proceedings of DIS'97 Designing Interactive Systems (Amsterdam, The Netherlands), ACM Press, August 1997, http://www.cs.uni-bonn.de/ os/, pp. 365–376.

[Sof96] The Software Productivity Consortium, 2214 Rock Hill Road, Herndon, VA 10170-4227, *Evolutionary spiral process model guidebook*, 1996, SPC-91076-MC. http://www.software.org/pub/Products/esppd.html.

[SS97] Tamara Sumner and Markus Stolze, *Evolution, not revolution: Participatory design in the toolbelt era*, In Kyng and Mathiassen [KM97].

[Sta94] Richard Stallman, *Why you should not use tcl*, posting in newsgroup comp.lang.tcl on Fri, 23 Sep 94 19:14:52 -0400, September 1994.

[Sun99] Sun Microsystems Inc., *Jini architecture specification, revision 1.0*, 1999, http://www.sun.com/jini.

[Syr97] Anja Syri, *Tailoring cooperation support through mediators*, Proceedings of the fifth European Conference on Computer–Supported Cooperative Work (Lancaster, UK) (John A. Hughes, Wolfgang Prinz, Tom Rodden, and Kjeld Schmidt, eds.), Kluwer Academic Publishers, September 1997, pp. 157–172.

[TB94] Randall H. Trigg and Susanne Bødker, *From implementation to design: Tailoring and the emergence of systematization in cscw*, Proceedings of the Conference on Computer Supported Cooperative Work (Chapel Hill, NC, USA) (Riachard Furuta and Christine Neuwirth, eds.), ACM Press, October 1994, pp. 45–54.

[TR88] Andrew S. Tanenbaum and Robbert van Renesse, *A Critique of the Remote Procedure Call Paradigm*, Research into Networks and Distributed Applications, Wien (R. Speth, ed.), North–Holland, 1988, pp. 775–783.

[VHJ98] Bruno Van Haetsdaele and Arnaud Jacquet, *Audio/video conferencing components for java*, Tech. report, Eurécom, 1998, Student project.

[Wei97] R. Weinreich, *A component framework for direct-manipulation editors*, Proceedings of TOOLS-25 (Melbourne, Australia), IEEE, November 1997.

[WJ94] Dadong Wan and Philip M. Johnson, *Computer Supported Collaborative Learning Using CLARE: the Approach and Experimental Findings*, ACM Conference on Computer Supported Collaborative Work (North Carolina), Chapel Hill, October 1994, file://ftp.ics.hawaii.edu/pub/tr/ics-tr-93-21.ps.Z, pp. 187–198.

[WWWK94] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall, *A note on distributed computing*, Tech. Report TR-94-29, Sun Labs, November 1994.

[You92] Edward Yourdon, *Decline & fall of the american programmer*, Yourdon Press, 1992.

[Zha97] X. Nick Zhang, *Secure code distribution*, IEEE Computer (1997), 76–79.