

A Component-Based Groupware Development Methodology

Cléver Ricardo Guareis de Farias, Luís Ferreira Pires, Marten van Sinderen
Centre for Telematics and Information Technology, University of Twente
P.O. Box 217, 7500 AE, Enschede, The Netherlands
{farias, pires, sinderen}@cs.utwente.nl

Abstract

Software development in general and groupware applications in particular can greatly benefit from the reusability and interoperability aspects associated with software components. Component-based software development enables the construction of software artefacts by assembling prefabricated, configurable and independently evolving building blocks, called software components. This paper presents a methodology for the development of groupware applications using a set of composable software components. This methodology consists of splitting the software development process according to four abstraction levels, viz., enterprise, system, component and object, and three different views, viz., structural, behavioural and interactional. The use of different abstraction levels and views allows a better control of the development process. We illustrate this methodology using a chat application as a case study.

1. Introduction

The development of Computer Supported Cooperative Work (CSCW) systems is a difficult and challenging task since it involves both social and technological issues. The process of developing a groupware application can be roughly split into three steps [5], viz., the design of the system functionality, the decomposition of the application into objects, and the use of tools and deployment environments for implementing and supporting these objects.

Implementation issues, such as the choice of objects to implement an application, the decomposition of objects into concurrent threads, the distribution of objects into different address spaces and hosts, and the choice between centralised, replicated or hybrid architectures, has long been identified as core issues that must be tackled during the development process [4]. However, not enough attention is given to reusability issues to a level

greater than the reuse of object class definitions in general.

Reusability is a key issue in software engineering. Its benefits include the reduction of costs and time-to-market of software products. In the CSCW research reusability issues are mainly addressed by cooperative toolkits. These toolkits, such as GroupKit [20], Rendezvous [9] and Prospero [6], aim at reducing the complexity of cooperative systems development, by providing reuse of solutions for common problems, mostly in terms of cooperative widgets and environment support.

Nevertheless, the reusability provided by the toolkits is restricted by two factors, viz., the infrastructure provided by the toolkit and the implementation language chosen. Given a particular cooperative object, its reuse may be restricted by the use of others objects or the toolkit support itself. Furthermore, the implementation language in which this object has been implemented plays an important role. For example, GroupKit is implemented in Tcl, while Prospero is implemented in Common List Object System (CLOS). If two cooperative objects are implemented in different languages, their interoperation is hard to achieve unless a middleware platform based on international standards is used.

Component-based software development has emerged to increase the reusability and interoperability of pieces of software. Component-based development aims at constructing software artefacts by assembling prefabricated, configurable and independently evolving building blocks, the so-called components. Components are binary, self-contained and reusable building blocks providing a unique service that can be used either individually or in composition with the service provided by other components [22].

Traditional object-oriented software development aims at providing reusability of object type definitions (object classes) at design and implementation levels. In contrast, component-based development aims at providing reusability of components at deployment level. In this way, components represent pieces of functionality that are

ready to be installed and executed in multiple environments.

Methodologies for groupware development are normally classified as pragmatic or theory/model based. In pragmatic methodologies, the system is rapidly prototyped and iteratively improved by means of the experience gained while using it. In theory/model based methodologies one first captures some knowledge of the application domain, and based on this knowledge the system is developed by focusing on the most relevant issues in early design phases. Application domain knowledge also helps structuring the system in a coherent way. Our research aims at combining these two approaches in order to profit from their individual benefits.

This paper presents a methodology for the development of groupware applications that combines a model into a pragmatic development process. Our approach consists of combining a component-based development process [8] based on the Unified Modelling Language (UML) [2, 17] with a conceptual cooperative model [7] to design and structure groupware applications in terms of a set of composable components. UML is a process independent modelling language with growing acceptance in both academic and industrial settings. UML basically consists of a collection of diagrams used to model a system under different and often complementary perspectives. To exemplify parts of our methodology we use a simple chat application as a case study.

This paper is further structured as follows: section 2 provides an overview of our component-based groupware development methodology; sections 3 to 5 detail our process; section 6 illustrates the development of a chat application as a case study; section 7 discusses some related software development processes and some drawbacks of using UML; finally, section 8 presents some conclusions and outlines some future work.

2. Methodology overview

Our methodology identifies four abstraction levels for the development of a groupware application, viz., enterprise, system, component and object.

The enterprise (or business) level aims at capturing the vocabulary and other domain information of the system being developed. This level has similar goals as the enterprise viewpoint of the RM-ODP [11] and provides the most abstract description of the system being produced.

The system level aims at identifying the boundary of the system being developed. This level aims at obtaining a clear separation between the system and its environment by capturing and defining the system requirements.

The component level aims at representing the system in terms of a set of composable software components and interfaces.

The object level aims at representing a component in terms of a set of related objects. This level corresponds to traditional object-oriented software development.

Figure 1 depicts the layering structure our methodology.

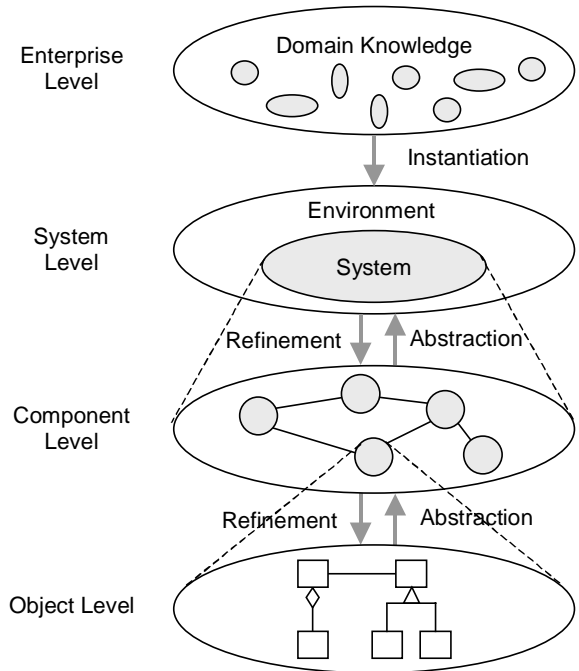


Figure 1. Abstraction levels of the methodology.

The four abstraction levels are related to each other in different ways. For example, the system level corresponds to one possible instantiation of the domain concepts present at the enterprise level. Different systems can be generated based on the same set of concepts. The component level corresponds to a refinement of the system level, in which the system is refined into a set of software components. The object level corresponds to a refinement of the component level, in which each component can also be refined into a set of objects.

We can also abstract from a set of objects to form a component and abstract from a set of components to form the system. However, it is not always possible to abstract from the system to obtain the complete description of the enterprise level because the concepts present at the system level may correspond only to a subset of the enterprise concepts.

Besides structuring into abstraction levels, we also consider different views at each one of these levels. Each view offers a different perspective of the system being developed. These perspectives are interrelated so that the

information contained in one view can partially overlap the information contained in the others.

We identify three basic views, viz., structural, behavioural and interactional. The structural view provides information about the structure of active or conceptual entities. The behavioural view provides information about the behaviour of each active entity in isolation, while the interactional view provides information about the behaviour of the different active entities as they interact with each other. Both the behavioural and the interactional views can be seen as dual views on the same aspect, viz., behaviour.

Figure 2 illustrates how the different views spans across the abstraction levels. Because the enterprise level is primarily a conceptual level, there is no clear division between the views, which is reflected by considering a unique representation among the different views at this level.

	Structural View	Behavioural View	Interactional View	...
Enterprise Level				
System Level				
Component Level				
Object Level				

Figure 2. Views versus abstraction levels.

3. Enterprise level

The enterprise level captures the vocabulary and other domain knowledge information of the system being developed. The information is used both to communicate with the users of the system and to serve as the basis for delimiting the system with respect to its environment.

An interesting characteristic of the enterprise level is its relative independence from the target application. In other words, because the information present at this level is mainly domain specific, it is common to several applications in this domain. For example, suppose we are developing a shared whiteboard. Once we have identified the concepts that are likely to be found in most shared whiteboards, we can create different systems based on these concepts, each one possibly considering a instantiation of different subsets of these concepts.

The concepts that should be captured at this level are the same as the concepts present in a conceptual cooperative model [7]. This cooperative model is based on four key concepts, viz., activity, actor, information and service, and on a set of relationships between them. Figure 3 represents the conceptual cooperative model in a

UML class diagram. A class diagram describes the types of objects and the different kinds of static relationships that connect them, while an object diagram describes an instance of a class diagram.

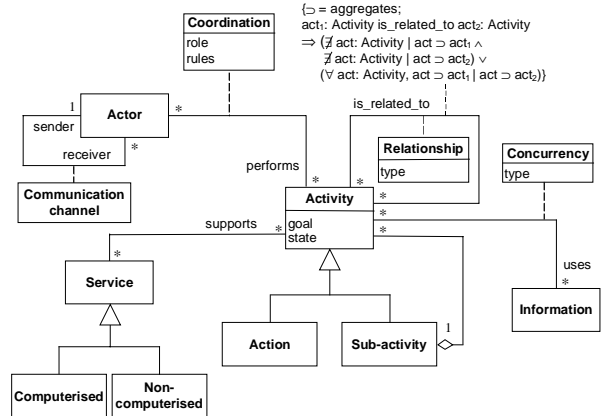


Figure 3. Cooperative model.

An activity represents a cooperative procedure; it can be decomposed into sub-activities and actions. The differences between a sub-activity and an action are twofold: a sub-activity can be further decomposed while an action is atomic; and a sub-activity is considered long-lived, while an action abstracts from duration by only considering the moment when the action is completed.

Activities that belong to the same aggregation level can be related to each other. Activities belong to the same aggregation level if these activities are top-level activities, i.e., they are not part of other activities, or these activities are part of the same activity. Examples of relationship between activities include the disabling of an activity by another and the sequential or the synchronised execution of two activities.

An actor represents an entity responsible for performing an activity. An actor can be either a human being or an autonomous agent, and is either individual or collective. A role is used to describe the responsibility taken by an actor in an association, while coordination rules, such as policies and floor control mechanisms, regulate the relationship between different actors performing the same activity.

Actors must communicate with each other to properly perform an activity. Communication occurs through a communication channel shared by the actors of an activity. The communication channel may represent an audio/video channel, an electronic mail tool or even live communication, in case the actors are all located in the same room.

Information represents any kind of electronic data either consumed or produced by the activity, such as messages, documents or database records. Frequently, the same information is shared by multiple activities. Si-

multaneous access to a piece of shared information is handled by two alternative mechanisms: locking of information or concurrency control to monitor the access to the information. A service represents any kind of computerised or non-computerised service that supports the execution of an activity.

Different techniques can be used to capture the information present at the enterprise level, such as a glossary of terms [14] and object diagrams.

The use of a glossary aims at maintaining a standard documentation of the terms encountered in the domain of the system. The use of such kind of documentation is common in software engineering and often appears with different names, such as data or model dictionaries.

An entry in the glossary should contain the name of the term, its type, such as actor, activity, service, etc., and some brief description.

In order to precisely describe some of the activities, preconditions and postconditions should be used whenever possible. A precondition is a constraint that must be true before the execution of the activity, while a postcondition is a constraint that must be true after the completion of the activity. To formally describe preconditions and postconditions we suggest the use of the Object Constraint Language (OCL) [17, 23]. OCL is an expression language defined as part of UML to describe constraints on object-oriented models.

UML object diagrams can be used to represent possible instantiations of the concepts identified in the glossary of terms and their relationships. Often there is no direct mapping between the identified concepts and their possible implementations.

4. System level

The system level defines the boundary between the system and its environment by capturing the system requirements. External services that support the system are identified at this level as well. At the system level the differences between the three views become apparent so that at this level these views get a more prominent role in the development process.

The structural view of a cooperative application at the system level is captured mainly through UML use case and package diagrams. Use case diagrams aim at capturing the system requirements, while package diagrams aim at capturing the static relationship between the system and external support services or systems.

The static relationship between an external service that supports the activities and the system itself may be represented by the presence of an actor indicating an external entity associated with a use case in a use case diagram. Alternatively, we can use a package diagram to

represent dependencies between these external services and the system itself.

Although a use case diagram is useful to identify the possible use cases of the system being developed, this type of diagram usually says little about the order in which the use cases should be executed.

One possible solution to explicitly represent the execution order of use cases is the adoption of constraints {precedes} or dependencies «precedes» between use cases. This solution can be suitable for simple use case diagrams. Nevertheless, for complex use case diagrams the adoption of this solution can be cumbersome because it burdens the understanding of the diagram. In this way, we suggest the use of (non-standard) use case sequence and collaboration diagrams [10] to capture the behavioural view of an application at the system level. Standard sequence and collaboration diagrams represent sequences of messages exchanged between a set of objects. Use case sequence and collaboration diagrams are not explicitly present in the UML notation guide, but they are allowed according to the UML metamodel [17].

According to UML, use cases are not allowed to communicate with each other. Further, they are always initiated by a signal from its associated actor. This makes it impossible to model situations in which a use case is initiated during the execution of another use case.

To overcome these restrictions we use invoke messages that represent the invocation of use case constructors. These constructors map to the signals from the actors to the use cases, either directly or indirectly. Invoke messages are the only messages that can be exchanged between use cases.

The interactional view of an application at the system level explicitly captures the possible interactions between the system and its environment, either actors or support systems and services, by using (non-standard) package sequence and collaboration diagrams [10]. These diagrams are also not explicitly present in the UML notation guide, but similarly to use case sequence and collaboration diagrams package sequence and collaboration diagrams are also allowed according to the UML metamodel.

5. System internal structure

This section presents the inner levels of our development process, i.e., the component and object levels.

5.1. Component level

The component level represents the system being developed in terms of a set of composable components and

their interfaces. A component provides access to its services via one or more interfaces. These services usually can be customised by adjusting some properties of the component.

When building a cooperative system from components in principle we do not need to know how these components are internally represented as objects. Actually, a component does not have to be necessarily implemented using an object-oriented technology, although this technology is generally recognised as the most convenient way to implement a component.

Components can be off-the-shelf, adapted from similar components and constructed from scratch. So far, most of the effort spent on building component-based applications concentrates on building new components. Nevertheless, the more mature and widespread this technology becomes the more likely it is that this effort will move towards adapting similar components and reusing existing ones.

Components can be developed at different levels or with different granularities, such as small, medium and large. The composition of components to form a larger component or application presents many problems, such as how to cope with incompatible interfaces and how to provide a unified interface for a composed component. Much research has been done on how to compose software in general and components in particular [1, 15]. Because component composition is a research topic in its own, we exempt ourselves from discussing it further.

The structural view of a cooperative application at the component level can be represented using package diagrams. The use of package diagrams aims at capturing the static relationship and dependencies between the internal components of the system and between these components and external systems. A deployment diagram can also be used to capture the physical distribution of the components in processing nodes.

A component can be graphically represented using either a package with a «component» stereotype or the UML component notation. However, UML uses a broader definition for a component encompassing software modules, such as executables, libraries, tables, files and documents. Thus, this notation should be used with caution. Since we have a specific connotation to the term component we suggest the use of a more specific notation.

The structural view also comprises the representation of the interfaces of the components. A component interface is a collection of operations that specify the service provided by the component. This interface can be represented as an interface class to show its operations; an interface class is an object class without attributes and exhibiting the «interface» stereotype.

The behavioural view of an application at the component level can be represented using activity diagrams for each component, while the interactional view of an application at the component level is captured mainly through package sequence and collaboration diagrams. The use of package diagrams aims at capturing the possible interactions between the internal components of the system and between these components and external systems.

5.2. Object level

The object level corresponds to the internal structure of the components. A component is structured using a set of related objects, which are implemented in a programming language.

The structural view of an application at the object level can be represented using use case, class and object diagrams. The behavioural view can be represented using statechart and activity diagrams, while the interactional view can be represented using sequence and collaboration diagrams.

The development process of a component at the object level corresponds to traditional object-oriented software development processes and therefore it does not require further discussion.

6. Case study

This session presents a case study where the development of a chat application using our methodology is illustrated.

6.1. Problem definition

The chat application used as a case study allows a group of participants engaged in a common chat session to exchange messages asynchronously amidst the session.

Before starting using the chat capabilities of the application the participant must first establish a connection either registering (first time users) or simply connecting (registered users) to the system.

After establishing a connection the participant may create a chat session or join an existing one. Just after the participant has joined the session she is notified about the number of messages that were exchanged in that session. The participant may choose to retrieve a number of messages equal to or less than the total number of messages exchanged within the session.

After joining a session the participant may invite new participants. If the invited participant is currently connected to the system she is immediately notified about the

invitation; otherwise the participant will be notified the next time she connects to the system. In this way, a participant may receive an invitation request at any time. The participant has to accept the invitation in order to join the session.

During the session the participant may exchange messages until he or she leaves the session. Changes in the session, such as the exchange of a message, the joining or leaving of participants into/out of the session or the addition of new participants to the session are reported to all participants currently engaged in that session.

6.2. Enterprise level modelling

The enterprise level modelling of the chat application starts with the identification of the main cooperative concepts and their description in the glossary.

Figure 4 shows some entries of the chat application glossary of terms. The entries for an actor, two activities and information are depicted. For simplification purposes we consider neither different kinds of actors nor floor control policies. The entry for the activity *Join* has a precondition described informally. This precondition constrains this activity to only those participants that are registered to the session they want to join.

Name	Level	Type	Description
Participant	Enterprise	Actor	Person who creates chat sessions, joins and leaves these sessions, invites new participants and exchanges messages within the session.
Register	Enterprise	Activity	Activity performed by all participants at the first time they establish a connection to the application.
Join	Enterprise	Activity	Activity performed by a participant in which this participant joins a session in order to collaborate. Pre: The participant has to be registered to the session to be able to join it.
Message	Enterprise	Information	Textual information that is exchanged among the participants of a chat session.

Figure 4. Glossary of terms.

The glossary should be maintained and updated as the development of the system continues. Consequently, the abstraction level at which the term was defined should be mentioned as well in the glossary since this term may be assigned different types as the development of the application evolves.

6.3. System level modelling

The system level modelling started with the capture of the structural view of the chat application. To capture this view a direct mapping was performed from the enterprise concepts of actor and activities to the use case diagram concepts of actor and use case, respectively. Each activity can be mapped to a separate use case or two

or more related activities can be combined in a same use case.

Figure 5 presents a simplified version of the chat application use case diagram. In this diagram several activities identified at the enterprise level were combined in a same use case at this stage of design. For example, the activities *Register* and *Connect* were mapped to the use case *Access Chat System*; the activities *Send Message* and *Receive Message* were mapped to the use case *Exchange Message*; the activities *Invite Participant* and *Answer Invitation* were mapped to the use case *Manage Invitation*, and so on.

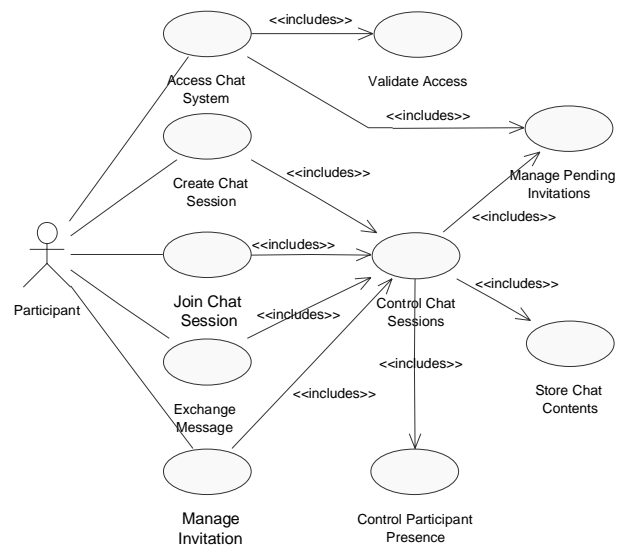


Figure 5. Use case diagram at the system level.

Each use case was described informally using text. For each use case we provided the following information: name, associated actors, purpose, overview, preconditions and postconditions (optional), associated use cases and typical courses of events (actor actions + system responses). This description scheme is based on the approach proposed in [14].

To capture the behavioural aspects (behavioural + interactional views) we provided some usage scenarios. Each scenario describes different situations in which the application is used and usually involves the execution of several use cases. The complexity of a scenario can vary, but we suggest a mix of simple scenarios with more complex ones. The provision of usage scenarios is an activity performed together with the application users.

In this case study, the behavioural view was captured using a use case sequence diagram for each scenario provided. This diagram captured the order in which the use cases can be executed. For example, the use case *Join Chat Session* can be executed only after the use case *Access Chat System*, while the use case *Exchange Message*

can be executed only after the use case *Join Chat Session*, and so on.

The interactional view was captured in two steps. Initially, for each use case we captured the interactions between the system and its environment using separate package sequence diagrams. Later, we captured the interactions present in each usage scenario.

Figure 6 depicts a package sequence diagram of the chat application. This diagram captures the interactions present in a simple scenario involving one participant. According to this scenario the participant establishes a connection registering herself, creates a chat session and then disconnects from the application.

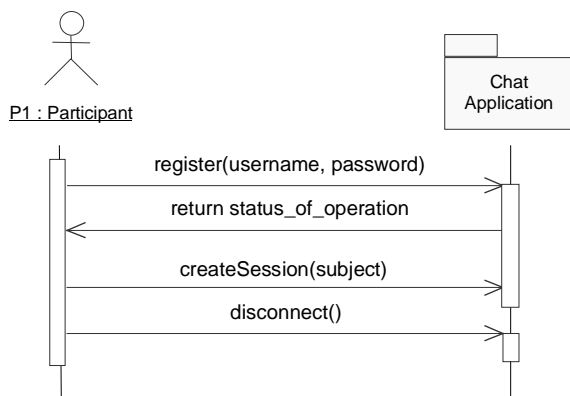


Figure 6. Package sequence diagram at system level.

6.4. Component level modelling

The first step to model an application at the component level is to identify the components and their static relationships (structural view).

In our methodology, we prescribe that one should try to assign the use cases identified at the system level to components, such that these components correctly support the use cases. However, there is no rule of thumb on how to assign use cases to components. A good practice is to keep similar functionalities in a same component and distinct functionalities in separate components. Although similarity and distinction are subjective terms, sometimes it suffices to rely on the individual judgement and experience of the application designer. In case a use case is likely to be supported by two or more components, it is possible that this use case is too complex and that it should be refined in multiple simpler use cases. In this case we can either return to the system level to carry out the necessary changes or create a new use case diagram in another abstraction level.

For the chat application we modelled the component level in two distinct phases. In the first phase we considered the chat application as a composition of two major

components, a client component and a server component. This distribution reflects our option for a centralised architecture for this particular case study.

The assignment of use case to these components was straightforward at this stage. Basically the left-hand side of the use cases presented in Figure 5 were assigned to the client, while the right-hand side of those use cases were assigned to the server. Additionally, an extra use case was created to control the participant access at the server side; client and server actors were added as well. Subsequently, modifications were made in the textual description of the use cases to cope with the changes.

We then modelled the behaviour of these components, particularly the interactional view, using package sequence diagrams. For each use case and for each scenario defined previously we created a package sequence diagram to capture the interactions between the environment and the client component and between the client component and the server component.

In the second phase we refined the client and server components into a composition of smaller components. The structural view at this phase was captured using package diagrams.

The assignment of use cases to components at this phase was also straightforward. At the client component side the assignment proceeded as follows: the use case *Access Chat System* was assigned to the component *Connection Management Client*; the use cases related to a chat session (*Create Chat Session*, *Join Chat Session*, etc.) were assigned to the component *Session Management Client*; the use case *Exchange Message* was assigned to the component *Message Exchange*; and, finally, the use case *Manage Invitation* was assigned to the component *Invitation Management Client*.

At the server component side the assignment proceeded as follows: the use cases *Control Participant Access* and *Validate Access* were assigned to the component *Connection Management Server*; the use case *Manage Pending Invitations* was assigned to the component *Invitation Management Server*; the use case *Store Chat Contents* was assigned to the component *Control Chat Contents*; the use cases *Control Participant Presence* and *Control Chat Sessions* were assigned to the component *Session Management Server*.

Figure 7 depicts the package diagram for the server components. Normally different alternative sets of components may all correctly support the same application, i.e., the different sets of components produce all equivalent results.

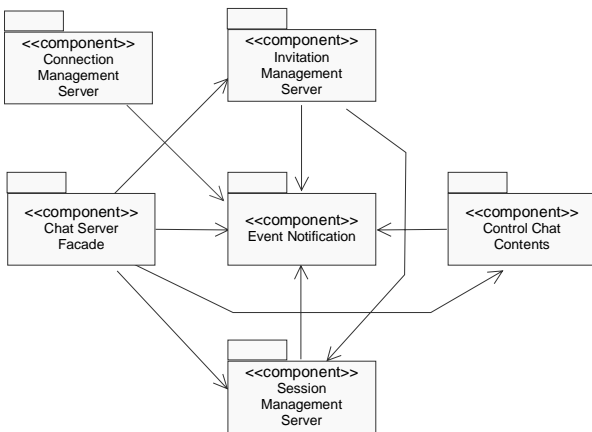


Figure 7. Package diagram for the server component.

The component Chat Server Facade was introduced to serve as a facade between the server components (except the component *Connection Management Server*) and the client components. The facade component provides a simple and unified interface for the functionality provided by a number of (smaller) components. A similar solution was adopted at the client side of the application.

In order to minimise the dependencies between the components both at the client side and at the server side we introduced an Event Notification component. This component is compliant with the CORBA Event Service specification [16]. It was used to decouple as much as possible one component from another, contributing for the individual reuse of the identified components.

The behavioural view modelling of the components was carried out using activity diagrams, while the interactional view modelling was carried out using package sequence diagram. At this stage we only modelled the interactions present at the usage scenarios previously defined.

Figure 8 illustrates a package sequence diagram for the components of the chat application. This diagram shows the interactions at the server side of the chat appli-

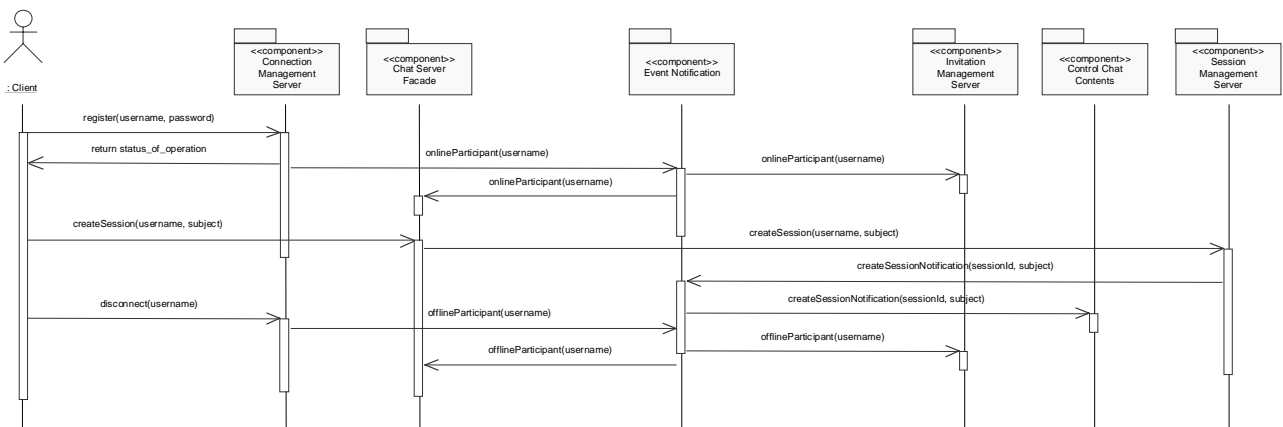


Figure 8. Package sequence diagram at component level.

cation for the same scenario as the one depicted in Figure 6 (system level).

6.5. Other considerations

The design of the components at the object level obeyed an approach similar to traditional software development processes.

We make no assumptions regarding the component model that will be used to implement the components. One can use CORBA, DCOM or any Java component model. For this particular application, the components were implemented using Java IDL [21].

Java IDL is a simple Object Request Broker (ORB) provided with the Java Platform. It can be used to define, implement, and access CORBA objects from the Java programming language. The Java IDL ORB supports only transient CORBA objects. This ORB also provides a transient name server, which is compliant with the CORBA Naming Service specification [16].

Because Java IDL does not provide an event notification service, we had to implement the component *Event Notification* ourselves. However, being based on a standard service this component can be reused in other applications. Similarly, the components that make use of the services provided by this component can be reused in other platforms without changes.

7. Discussion

This section compares our methodology with similar approaches based on UML. This session also indicates some drawbacks of the use of UML.

7.1. Related work

The Unified Process [12] is a software development

process based on UML. Actually, the Unified process is not really a development process but more like a process framework, since it describes best practices in software development but still has to be specialised to be suitable for different projects. The Unified Process identifies two dimensions: time (cycles, phases and iterations) and workflows. Each workflow captures a set of activities and artefacts; models are the most important kind of artefacts. Possible models include domain, use case, analysis, design, process, deployment, implementation and test. The Unified Process also defines some views, such as use case, design, process, deployment and implementation.

A pattern of four deliverables is used to describe software products in [10]. According to this pattern a software artefact can be described at several levels of abstraction and from different views. The pattern defines four main levels of abstraction, viz., system, architectural, class and procedural. Other levels, such as domain, document and testing, are also possible but less frequent. The defined views are use case, logical, component and deployment. At each level and at each view a software artefact can be described using static relationships, dynamic interactions, responsibilities and state machines.

The Catalysis approach [3] is yet another development process based on UML. Similarly to the Unified Process, the Catalysis approach is much like a process template, which can be tailored accordingly to the situation. Catalysis is based on three modelling concepts (type, collaboration and refinement) and frameworks. A type specifies the external behaviour of an object; a collaboration specifies the behaviour of a group of objects, while a refinement relates different levels of behaviour description. Frameworks describe recurring patterns of these three concepts. Catalysis also splits the development process in three levels: the domain/business, the component or system specification and the component implementation. The component specification describes the externally visible behaviour, while the component implementation describes the internal structure and behaviour.

Our development process is not so generic and complete as Unified Process and Catalysis; however, the relative simplicity of our methodology may constitute its major benefit. The processes presented here structure, in a more or less extent, the development process of a software system in different levels and according to different views. Still, we do believe that the levels and views adopted in our process are the most reasonable and pragmatic choices for a component-based groupware development process. Our development process considers the use of components explicitly, while both the Unified Process and the pattern deliverable process use a broader definition of a component.

7.2. Drawbacks of UML

UML is suitable to model most of the development process of a software component, but one can still identify some drawbacks. First and foremost, UML does not support the explicit specification of quality of service (QoS) requirements. To describe simple and isolated requirements, we can attach some constraints or textual descriptions to use cases or interfaces, but if QoS requirements are pervasive throughout the whole system these ad hoc constraints and descriptions are not enough. Recognising the importance of QoS specification, OMG launched a request for proposals for a UML profile that defines standard paradigms of use for modelling QoS and other aspects of real-time systems [18].

The specification of complex behaviours using statechart and activity diagrams can also be problematic. These types of diagram provide roughly three kinds of constructs to describe the relationship between states or activities: enabling, interleaving (parallelism) and synchronisation. Guards can also be used in combination with the enabling construct, allowing one to represent a kind of deterministic choice. However, non-deterministic choices and disabling cannot be directly represented using UML models. The extension of statechart and activity diagrams with these two concepts could facilitate behavioural specification.

Most of UML commercially available supporting tools, such as Rational Rose, Together J and Select Software, do not support use case and package sequence and collaboration diagrams because these diagrams are not described in the UML notation guide, although they are allowed by the UML metamodel. This shortcoming exposes the limitations of UML for supporting component-based software development. However, a major change in UML is expected to occur in 2001 with the release of the UML 2.0 specification [13]. This release aims at, amongst others, providing better support to component-based development, including CORBA, Enterprise Java Beans and DCOM.

8. Conclusion

This paper presented a component-based methodology for the development of groupware applications. According to this process, the development of an application is organised using four different abstraction levels. At each level different views are used to capture structural, behavioural and interactional aspects of the application under development. We illustrated this methodology using a chat application as a case study.

The three different views presented in this paper seem to be the most relevant ones for application design. Still we could have introduced other views, such as a test view. In this case, at each abstraction level the test view would capture the information required to test the system as a whole, and components and objects individually.

Unlike most software development processes, which normally prescribe the development of a set of objects followed by their grouping into components, our methodology aims at identifying a set of components, possibly reusing existing ones, and refining them into objects afterwards.

UML is suitable to model most of the development process of a software component, but UML still does not support the explicit specification of QoS. Besides, the support for component modelling should be improved.

The proposed development process is general enough to be applied in several different areas rather than groupware. However, our research is focused on the development of several groupware applications, such as a voting application and a multimedia conferencing tool.

We will also investigate the use of other techniques to be applied in combination with UML. In particular, we are interested in the use of the architecture modelling language proposed in [19].

Acknowledgements

This work has been carried out in the scope of the AMIDST project (<http://amidst.ctit.utwente.nl>). Cléver Ricardo Guareis de Farias is supported by CNPq (Brazil).

References

1. Bergmans, L.: Constructing reusable components with multiple concerns. *International Symposium on Software Architectures and Component Technology (SACT)*. Enschede, The Netherlands. To be published in M. Aksit (ed.), Kluwer, 2000.
2. Booch, G., Rumbaugh, J. and Jacobson, I.: *The Unified Modelling Language user guide*. Addison Wesley, USA, 1998.
3. D'Souza, D. F. and Wills, A. C.: *Objects, Components and Frameworks with UML: the Catalysis Approach*. Addison Wesley, USA, 1999.
4. Dewan, P.: A technical overview of CSCW. *Tutorial presented at the European Conference on Computer Supported Cooperative Work (ECSCW'99)*, Copenhagen (Denmark), 1999.
5. Dewan, P.: Architectures for Collaborative Applications. In Beaudouin-Fafon, M. (Editor), *Computer Supported Cooperative Work (Trends in Software, 7)*, John Wiley & Sons, USA, pp. 169-193, 1999.
6. Dourish, P.: Using Metalevel Techniques in a Flexible Toolkit for CSCW Applications. *ACM Transactions on Computer-Human Interaction*, 5(2), pp. 109-155, 1998.
7. Guareis de Farias, C. R., Ferreira Pires, L., and van Sinderen, M.: A conceptual model for the development of CSCW systems. *Fifth International Conference on the Design of Cooperative Systems (COOP 2000)*, Sophia Antipolis, pp. 189-204, 2000.
8. Guareis de Farias, C. R., van Sinderen, M., and Ferreira Pires, L.: A systematic approach for component-based software development. In *Proceedings of the Seventh European Concurrent Engineering Conference (ECEC 2000)*, pp. 127-131, 2000.
9. Hill, R.D., Brinck, T., Rohall, S.L., Patterson J.F. and Wilner, W.: The rendezvous architecture and language for constructing multiuser applications. *ACM Transactions on Computer-Human Interaction*, 1(2), pp. 81-125, 1994.
10. Hruby, P.: Structuring Design Deliverables with UML. In *Proceedings of UML'98 International Workshop*, pp. 251-260, 1998.
11. ISO/IEC: *Open Distributed Processing – Reference Model: Part 3: Architecture*, International Standard, 1995.
12. Jacobson, I., Booch, G. and Rumbaugh, J. *The unified software development process*. Addison Wesley, USA, 1999.
13. Kobryn, C.: UML 2001: a standardization odyssey. *Communications of the ACM*, 42(10), 29-37, 1999.
14. Larman, C.: *Applying Uml and Patterns: An Introduction to Object-Oriented Analysis and Design*. Prentice Hall, USA, 1997.
15. Lewandowski, S. M.: Frameworks for component-based client/server computing. *ACM Computing Surveys*, 30(1), pp. 3-27, 1998.
16. Object Management Group: *Corba Services: Common Object Services Specification*, 1998.
17. Object Management Group: *Unified Modeling Language 1.3 specification*, 1999. Available at <http://www.omg.org>.
18. Object Management Group: *UML profile for modeling quality of service and fault tolerance characteristics and mechanisms*. Draft RFC, version 2, 1999.
19. Quartel, D. *Action relations: basic design concepts for behaviour modelling and refinement*. PhD thesis, University of Twente, Enschede, the Netherlands, 1998.
20. Roseman, M. and Greenberg, S.: Building real time groupware with GroupKit, A groupware toolkit. *ACM Transactions on Computer Human Interaction*, 3(1), pp. 66-106, 1996.
21. Sun Microsystems. *Java IDL*. Available at <http://java.sun.com/products/jdk/1.2/docs/guide/idl/index.html>
22. Szyperski, C.: *Component software: beyond object-oriented programming*, Addison-Wesley, USA, 1998.
23. Warmer, J. and Kleppe, A.: *The object constraint language: precise modeling with UML*. Addison-Wesley, USA, 1999.