A SYSTEMATIC APPROACH FOR COMPONENT-BASED SOFTWARE DEVELOPMENT

Cléver Ricardo Guareis de Farias, Marten van Sinderen and Luís Ferreira Pires
Centre for Telematics and Information Technology (CTIT)
PO Box 217, 7500 AE, Enschede, the Netherlands
{farias, sinderen, pires}@cs.utwente.nl

KEYWORDS

Component-based development, component software, UML, reusability.

ABSTRACT

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 an approach for the development of component-based software artefacts. This approach 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.

INTRODUCTION

Reusability, whose benefits include both the reduction of costs and time-to-market of software products, is a key issue in software engineering. 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 (Szyperski, 1998).

Traditional object-oriented software development aims at providing reusability of object type definitions (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.

This paper presents an approach to the development of component-based software based on the Unified Modelling Language (UML) (Booch et al., 1998; OMG, 1999a). Our software development process identifies four abstraction levels for the development of a software artefact, viz., enterprise, system, component and object.

This paper is further structured as follows. The next section presents an overview of our development process. The following sections discuss the enterprise level, the system level and the internal system levels of the development process, i.e., the component and object levels, respectively. Finally, we draw some conclusions and outlines some future work.

ABSTRACTION LEVELS AND VIEWS

Our software development process identifies four abstraction levels for the development of a software artefact, 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 (ISO/IEC, 1995) 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. Components used in this level come from three alternative sources: off-the-shelf components, adaptation of available components and construction of the new components from scratch.

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 of the software development process.
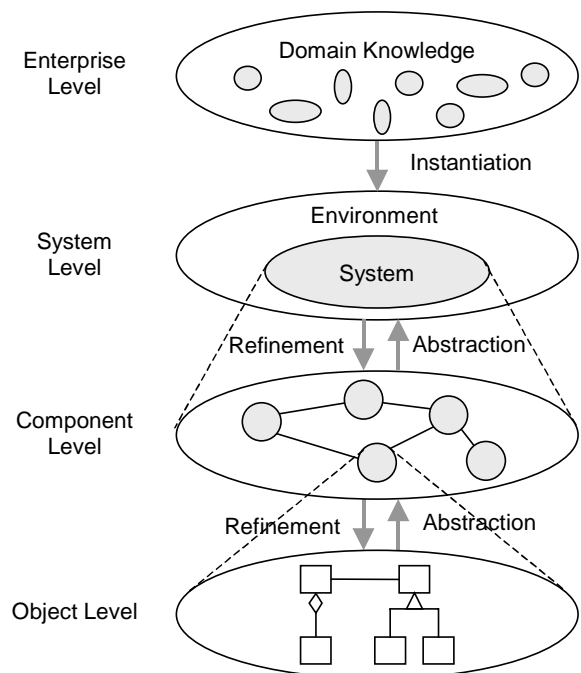

Figure 1. Abstraction levels in the development process.

The four abstraction levels are related to each other in different ways. For example, the system level corresponds to a 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 and 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.

In this work, we identify three different 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.

Figure 2. The use of views in the development process.

In the sequel we discuss in further detail each one of the abstraction levels and illustrate some of steps of the development process using some excerpts from a voting application example.

ENTERPRISE LEVEL

The enterprise level captures the vocabulary and other domain knowledge information of the system being developed. This level is similar to the enterprise viewpoint of RM-ODP (ISO/IEC, 1995) in the sense that it defines the purpose, scope, actors, activities, rules, policies, support services and so forth, of the system being developed.

The information captured at the enterprise level is used both to communicate with the users of the system being devel-

oped 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.

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

The use of a glossary (Larman, 1997) aims at mantaining 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 dictionary or model dictionary.

An entry in the glossary should contain the name of the term, its type, such as actor, activity, rule or policy, and some brief description. The glossary should be mantained 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 system evolves.

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.

Figure 3 shows entries of the voting application glossary.

Name	Level	Type	Description
Controller	Enterprise	Actor	Person who creates new sessions and authorises new participants.
Vote	Enterprise	Activity	Activity performed by all participants of a voting session in which they choose a winning proposal by voting for it.

Figure 3. Glossary of terms.

Concept diagrams can also be used at the enterprise level to capture domain-related concepts. Often there is no direct mapping between the identified concepts and their possible implementations. A concept diagram consists of a UML class diagram in which classes represent concepts and associations between these classes represent relationships between the concepts (see (Guareis de Farias et al., 2000) for usage examples of concept diagrams). UML object diagrams can complement the use of class diagrams by representing a possible instantiation of the concepts captured.

SYSTEM LEVEL

The system level clearly 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 enterprise concepts of actor and activities are directly mapped to the use case diagram concepts of actor and use case, respectively. The static relationship between an external service that support the activities and the system itself may be represented by the presence of an actor representing 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.

Figure 4 depicts the use case diagram of the voting application. In this figure, an ellipse represents a use case, a "stick man" represents an actor and «includes» represents a relationship between use cases.
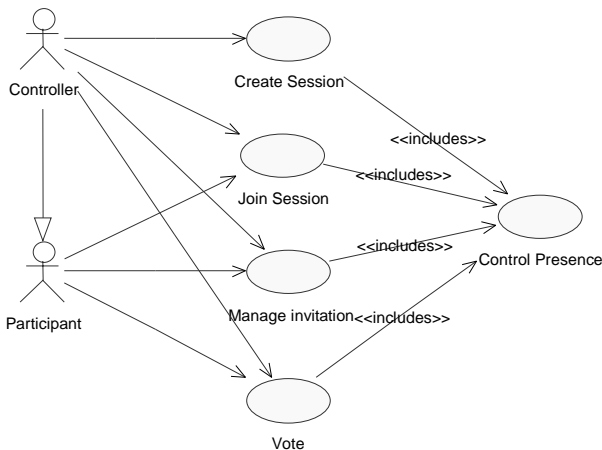

Figure 4. Use case diagram.

Although a use case diagram is useful to identify the possible use cases of the system being developed, a use case diagram usually says little about the order in which the use cases should be executed. In this way, we suggest the use of (non-standard) use case sequence and collaboration diagrams to capture the behavioural view of an application at the system level (Hruby, 1998). 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 (OMG, 1999a).

According to UML, use cases are not allowed to communicate with each other. Furthermore, use cases 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.

Figure 5 shows a use case sequence diagram. According to this diagram a participant executes the use cases *Join Session* and *Vote*. The execution of the former use case triggers the execution of the use case *Control Presence*, which is not directly executed by the actor.

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 (Hruby, 1998). 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 (OMG, 1999a).
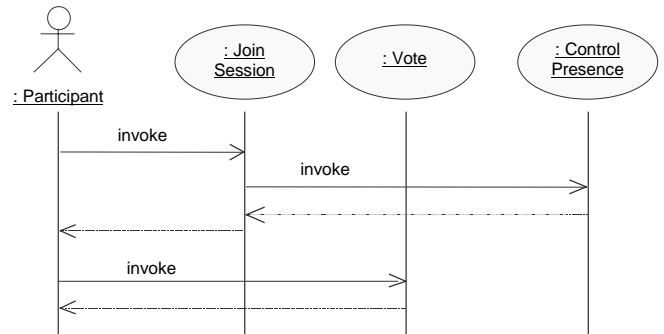

Figure 5. Use case sequence diagram.

SYSTEM INTERNAL STRUCTURE

This section presents the inner levels of our development process, i.e., the component level and the object level, and discusses some issues pertaining to component composition.

Component level

The component level represents the system being developed in terms of a set of composable components and their interfaces. A component is a binary, self-contained and reusable building block that provide a unique service that can be used either individually or in composition with the service provided by other components (Szyperski, 1998). 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.

In principle when building an application from components 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 concentrate 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 components and reusing existing ones.

Components can be developed at different levels or with different granularities, such as small, medium and large. On one hand, small components are usually easier to develop, being much alike object classes, and they are normally general purpose. However, when small components are used the amount of work required to compose them can be considerably large. On the other hand, medium to large components are more difficult to develop and they are usually specific purpose. Nevertheless, the composition of medium and large components is normally less time-consuming than the case of small components. Frequently, larger components are built using smaller ones.

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 (Keller and Schauer, 1998; Lewandowski, 1998; Bergmans, 2000). Because component composition is a research topic in its own, we exempt ourselves from discussion it further.

The structural view of an 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 application 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.

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.

We can formalise the operations on an interface using the Object Constraint Language (OCL) (OMG, 1999a; Warmer and Kleppe, 1999). OCL is an expression language that allows one to describe constraints on object-oriented models.

In our component-based software development process, 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. Actually, it is may be necessary to introduce new use cases as the development process continues.

Normally different alternative sets of components may all correctly support the same application, i.e., the different sets of components produce all equivalent results. For example, we could decide to use many small components instead of fewer larger ones or vice-versa, both resulting in equivalent compositions.

Figure 6 depicts a package diagram resulting from the assignment of use cases to components. The use cases *Create Session*, *Join Session* and *Control Presence* were assigned to the *Session Manager* component. The use case *Manage Invitation* was assigned to the *Invitation Manager* component, while the use case V*ote* was assigned to the *Vote Controller* component. Because we are roughly illustrating parts of the development process, it is hard to notice that some pieces of functionality found in these use cases aim at maintaining the consistency among the replicated instances of the application (we are implicitly assuming a replicated architecture for this application). In order to handle replication, we assigned these functionalities to a single component named *Replication Manager*. We also designated a single component named *User Interface* to implement the user interface for all

components. Optionally, each component could implement its own user interface.
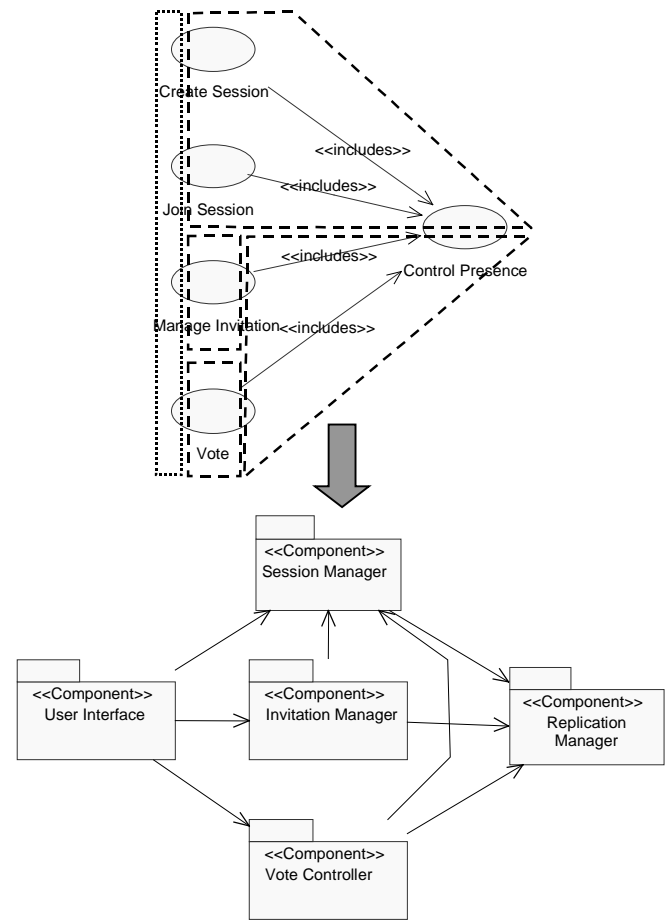

Figure 6. Assignment of use case to components.

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.

We derive the interface(s) of a component from its interaction with other components and external systems.

Figure 7 shows a package collaboration diagram at the component level.
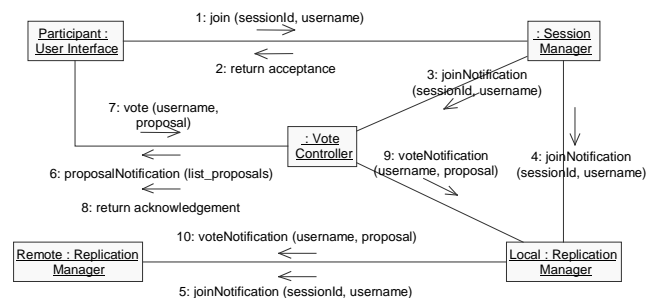

Figure 7. Package collaboration diagram.

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 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.

CONCLUSION AND FUTURE WORK

This paper presented a component-based software development process for the construction of software artefacts. According to this process, the development of an application is organised using four different abstraction levels. At each level different system views are used to capture structural, behavioural and interactional aspects of the system under development.
The three different views presented in this paper seem to be the most relevant ones for system 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, such as the Unified process (Jacobson et al., 1999) and the four deliverable process (Hruby, 1998), which normally prescribe the development of a set of objects followed by their grouping into components, our approach 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 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 recently 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 (OMG, 1999b).
Some 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 (Kobryn, C. 1999). This release aims at, amongst others, providing better support to component-based development, including CORBA, Enterprise Java Beans and DCOM.
Currently, we are applying our approach in the development of several cooperative applications, such as a chat system, a shared whiteboard and a conferencing tool. These systems will be further composed into a tele-learning environment.
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 (Quartel, 1998).

ACKNOWLEDGEMENTS

REFERENCES

Bergmans, L. 2000: Constructing reusable components with multiple concerns. International Symposium on Software *Architectures and Component Technology (SACT)*. University of Twente, Enschede, The Netherlands. To be published in M. Aksit (ed.), Kluwer.

Booch, G., Rumbaugh, J. and Jacobson, I. 1998: *The Unified Modelling Language user guide*. Addison Wesley, USA, 1998.

Guareis de Farias, C.R., Ferreira Pires, L. and van Sinderen, M. 2000: A conceptual model for the development of CSCW systems. *Fourth International Conference on the Design of Cooperative Systems*. To appear.

Hruby, P. 1998: Structuring Design Deliverables with UML. In *Proceedings of UML'98 International Workshop*, 251-260.

ISO/IEC 1995: *Open Distributed Processing – Reference Model: Part 3: Architecture*, International Standard.

Jacobson, I., Booch, G. and Rumbaugh, J. 1999: *The unified software development process.* Addison Wesley, USA.

Keller, R.K. and Schauer, R. 1998: Design components: toward software composition at the design level. In *Proceedings of the 1998 International Conference on Software Engineering*, 302-311.

Kobryn, C. 1999: UML 2001: a standardization odyssey. *Communications of the ACM*, 42(10), 29-37.

Larman, C. 1997: *Applying Uml and Patterns: An Introduction to Object-Oriented Analysis and Design*. Prentice Hall, USA.

Lewandowski, S. M. 1998: Frameworks for component-based client/server computing. *ACM Computing Surveys*, 30(1), 3-27.

Object Management Group 1999a: *Unified Modeling Language 1.3 specification*.

Object Management Group 1999b: *UML profile for modeling quality of service and fault tolerance characteristics and mechanisms*. Draft RFC, version 2.

Quartel, D. *Action relations: basic design concepts for behaviour modelling and refinement*. PhD thesis, University of Twente, Enschede, the Netherlands, 1998.

Szyperski, C. 1998: *Component software: beyond object-oriented programming*. Addison-Wesley, USA.

Warmer, J. and Kleppe, A. 1999: *The object constraint language: precise modeling with UML*. Addison-Wesley, USA.