Executive Summary:
Software Evolution in Componentware –
A Practical Approach*

Andreas Rausch
rausch@in.tum.de

Institut für Informatik
Technische Universität München
D-80290 München
http://www4.in.tum.de

10th June 1999

Abstract

Industrial software projects are not based on a top-down development process relying on refinement but use an evolutionary approach. In this paper, we present the basic concepts of a suitable overall componentware methodology with respect to software evolution. We clarify the difference between refinement steps and evolution steps in an document-based development process. Based on this, we introduce the concept of *Requirements/Assurances Contracts* to explicitly model the dependencies between the development documents. This helps developers to track and manage the software evolution process.

Keywords: Software Evolution, Componentware, Description Techniques, Process Model, Software Engineering

1 Introduction

Most of today's software engineering methodologies are based on a top-down development process, e.g. the Object Modeling Technique (OMT) [RBP+91], the Rational Unified Process (RUP) [Iva99], the Objectory Process [Jac92], or the German standard for information systems development in the public service, called V-Modell [IAB98]. The basic idea behind those is: During system development a model of the system is build and step-wise refined. A refinement step adds addional properties of the desired system to the model. At last the model is a sufficient fine, consistent, and correct representation of the desired system. It may be directly implemented or even generated. Surely, all of these processes support local iterations, for instance the RUP allows developers to have interations during analysis, design or implementation. But the overall process is still based on refinement steps of the underlaying system model. In formal approaches, like ROOM [Bra94] or FOCUS [BDD+92, Bro98] is the concept of refinement even more strict.

These kind of process models involve some severe drawbacks: Initially, the customer often does not know all relevant requirements, cannot state them adequately, or even states inconsistent requirements. Consequently, many delivered systems do not meet the customer's expectations. In addition, top-down development leads to systems that are very brittle with respect to changing requirements because the system architecture and the involved components are specifically adjusted to the initial set of requirements. This is in sharp contrast to the idea of building a system from truly reusable components, as the process does not take already existing components into account with respect to reuse. Beyond this, lifecycle and maintenance are not supported in these approaches. This is exremly critical as, for instance, nowadays maintenance takes about 80 percent of the IT budget of europe's companies in the average and 20 percent of the user requirements are obsolete within one year (cf. [Neu99]).

However, there are some approaches available with a more sophisticated support of software evolution, like for instance Rapid Prototyping [Bis92] or reuse resp. component based approaches [BRSV98a, BRSV98b, ABD+99, Szy97]. Nevertheless, software evolution as a basic concept is currently not well supported. In our opinion, this is partly due to the

* This paper originates from the research project A1—Methods for Component-Based Software Engineering, a part of Bayerischer Forschungsverbund Software-Engineering (FORSOFT), supported by Siemens ZT.

lack of a suitable overall componentware methodology with respect to software evolution. Such a methodology should at least incorporate the following parts, as illustrated in Figure 1:

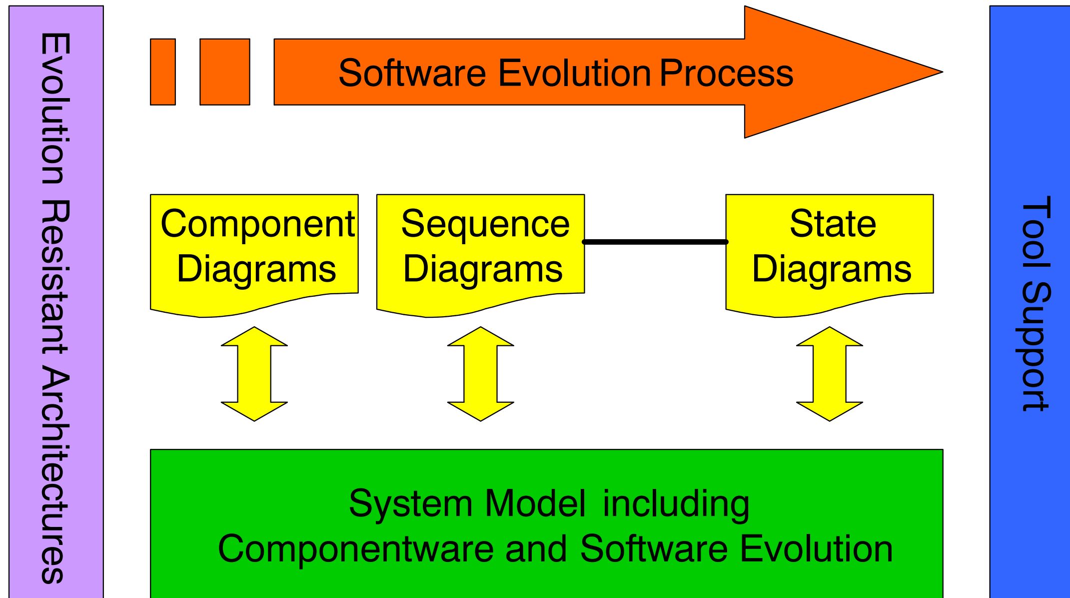

Figure 1: Framework of Software Evolution in Componentware

System Model: A well-defined conceptual framework for componentware and software evolution is required as a reliable foundation. It contains the basic definitions, notion, and concepts, e.g. for evolution, a component, or an interface.

Description Techniques: Based on the system model, desciption techniques for componentware are needed. Therewith, developers can model and document the evolution of a single component resp. a whole component system. As the description techniques are based on the system model we can state well-defined consistency criteria between descriptions of different evolution steps. We may be able to verify the correctness of evolution steps with the help of specialized tools.

Software Evolution Process: Development should be organized according to a process model tailored to software evolution resp. componentware evolution. This includes guidelines for the usage of the description techniques and reasonable evolution steps.

Evolution Resistant Architectures: To minimize the costs of software evolution systems should be developed on evolution resistant architectures. Such architectures contain a common basic infrastructure for components, like DCOM [Bro95], CORBA [OH97], or Java Enterprise Beans [Jav99]. But even more important are business oriented standard architectures, which are evolution resistant. Whereas the latter one relay on the former one.

Tools: At last, all former parts should be supported by tools. Instances of the system model are stored in repositories and changed by editors, which support the description techniques. An overlying workflow tool organizes the software evolution process. Programming tools support evolution resistant architectures.

In the following sections we will discuss these aspects in detail. In Section 2 we introduce our view of a software evolution process and of development documents. We introduce the concept of *Requirements/Assurances Contracts* to model explicitly the dependencies between the development documents. A short conclusion ends the paper.

2 Development Documents and Process

Usually, during the development of a system, various *documents* are created. Such a document is seperate unit that describes a certain aspect of, or 'view' on the system under development. In componentware we typically have the following kinds of documents:

Structural Documents describe the structure of the of a system resp. component. The structure of a component consists of its subcomponents and the connections between the subcomponents and the supercomponent. Usual structural documents describe the static structure of the system (e.g. aggregation or inheritance in UML Class Diagrams [OMG99]). More sophisticated onces can also describe the structural changes over time (e.g. Architecture Description Languages [BDRS97]).

Interface Documents describe the interfaces of components. Currently most interface descriptions (e.g. CORBA IDL [OH97]) only allow to specify the syntax of component interfaces. The behavior of interfaces is usual described in prose. Enhanced descriptions use pre- and post-conditions, e.g. Eifel [Mey97].

Protocol Documents describe the interaction between a set of components. Typicial interactions are message exchange, data exchange, or dynamic changes in the connection structure. Examples of protocol descriptions are: Sequence Diagrams in UML [OMG99] or Extended Event Traces [BHKS97].

Implementation Documents describe the implementation of a component. Programm code is the most popular kind of those descriptions, but we can also use automatons, like in [Rum96, Har88]. Especially in componentware the implementation of a component can be (recursively) described by a set of structural, interface, protocol, and implementation documents.

During development we describe a system by sets of those documents. Let DOC be the infinite set of all documents. Let SYS be the infinite set of all systems. We can assign a semantic to a given set of documents via the semantic function sem as follows:

$$sem : \mathcal{P}(DOC) \rightarrow \mathcal{P}(SYS)$$

Intuitively, sem calculates for a given set of development documents the set of systems that are described by these documents or—in other words—that 'fulfill' the specification given by the development documents. Current development methods are based on refinement as already outlined in Section 1. Figure 2 illustrates three refinement steps. A refinement step (e.g. from Doc_{t_1} to Doc_{t_2}) is formally correct if the set of systems described by Doc_{t_1} is a subset of the systems described by Doc_{t_2}: $sem(Doc_{t_2}) \subseteq sem(Doc_{t_1})$. Formal methods like for instance FOCUS allow us to prove the sematical correctness of those refinement steps [BDD+92, Bro98, Rum96]. More informal methods provide tools that perform syntactical correctness checks of refinement steps.

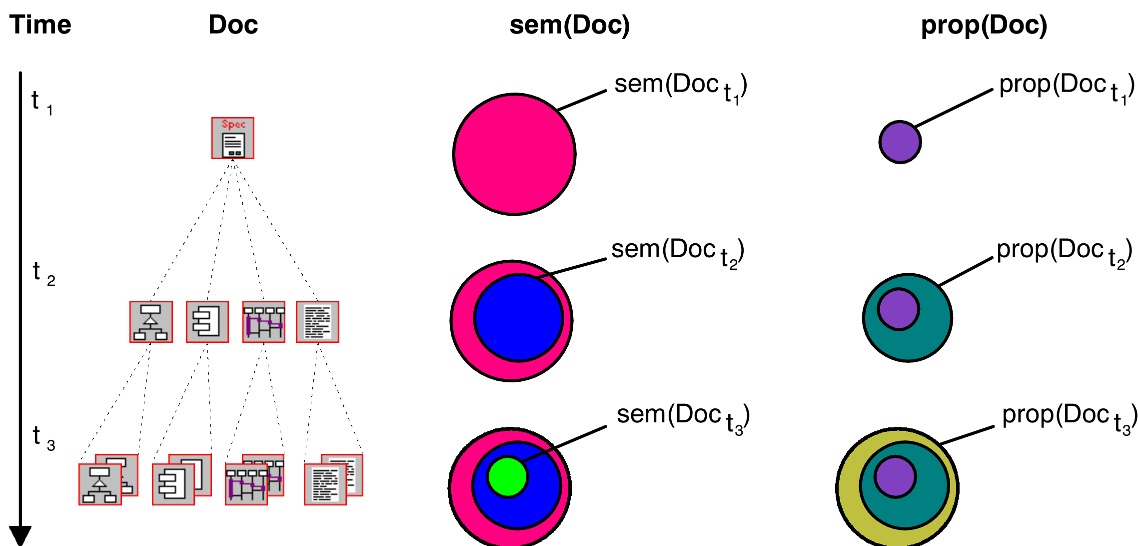

Figure 2: Refinement during System Development

As Figure 2 illustrates, a correct refinement steps adds additional properties to the set of documents describing the desired system. Let $PROP$ be the infinite set of all properties. Thus, we can define another semantic function which assignes to a given set of documents a set of properties characterizing the system:

$$prop : \mathcal{P}(DOC) \rightarrow \mathcal{P}(PROP)$$

Again, a refinement step is formally correct, if the condition $prop(Doc_{t_2}) \supseteq prop(Doc_{t_1})$ holds, or more intuitively, if the new set of properties are consistent with the former one. Nevertheless, the pure top-down approach based on refinement has several severe drawbacks (Section 1). For that reason development steps in practice are not refinement steps but evolution steps. Hence, tools based on formal or informal methods can not help developers in a reasonable way.

Figure 3 shows three typical evolution steps during system development. An evolution step in our sense are changes in the set of development documents within a certain time step. Neither refinement (first condition) nor abstraction (second condition) are evolution steps. But the described properties of the system before the evolution step and after the evolution step must have a common base (third condition). Formally, an evolution step is given by the following function:

$$change : \mathcal{P}(DOC) \rightarrow \mathcal{P}(DOC), \ \ where \ \forall Doc \in \mathcal{P}(DOC) :$$
$$prop(Doc) \not\supseteq prop(change(Doc)) \wedge prop(change(Doc)) \not\supseteq prop(Doc) \wedge prop(Doc) \cap prop(change(Doc)) \neq \emptyset$$

Note, this definition of an evolution step is strongly simplifying. We have to refine this definition and come up with evolution rules for a single document of a certain type. A full treatment is beyond the scope of this executive summary, as the resulting formulae are rather lengthy.

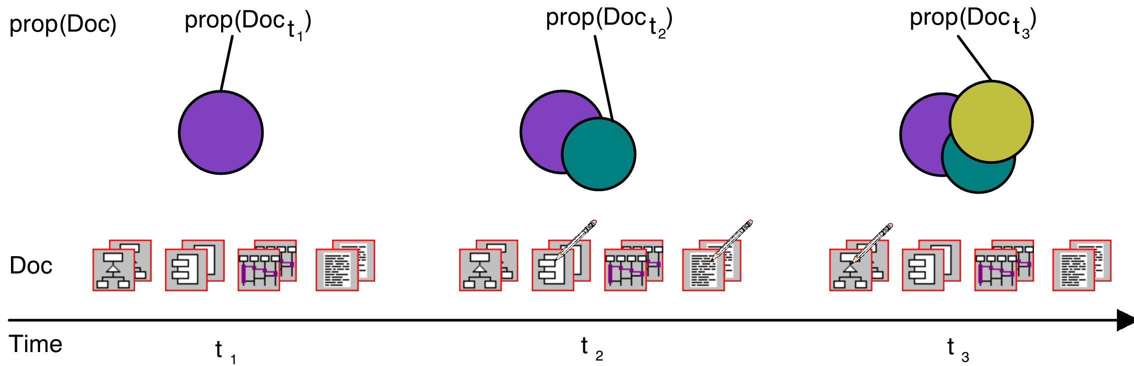

Figure 3: Evolution during System Development

As illustrated in Figure 3, the basic condition of classical refinement based approaches $prop(Doc_{t_2}) \supseteq prop(Doc_{t_1})$ does not longer hold. However, for the development of complex systems, we still need methodical guideline and help. Obviously, we need new techniques to prove the 'correctness' of those evolution steps. For that reason we claim that an evolution based development methodology has to include guidelines and tools to

1. calculate and visualize for each evolution step the dismissed properties of the system (formally given by $prop(Doc_{t_1}) \setminus prop(Doc_{t_2})$),
2. calculate and visualize for each evolution step the new properties of the system (formally given by $prop(Doc_{t_2}) \setminus prop(Doc_{t_1})$), and
3. validate the correctness of this evolution step by specialized tools or developers.

To reach this goal we have to make the dependencies between the development documents more explicit. Currently, in description techniques or programming languages the modelable dependencies between different documents are extremly rudimentary. For instance, in UML [OMG99] designers can only specifiy the relation *uses* between documents or in Java [Fla96] programmers have to use the *import* statement to specify that one document relays on another.

If a document changes—via an evolution step—the consequences for documents that relay on the evolved document are not clear at all. Normally, the developer, who causes the evolution step, has to to check whether the other documents are still correct or not. As the concrete dependencies between the documents are not explicitly formulated, the developer has usually to go into the details of all concerned documents.

To avoid these drawbacks, we propose to model the dependencies between the different documents more concretely within description techniques tailored to the special needs of software evolution. These description techniques enable the designer to model the required and assured properties of a certain document explicit within this document. Such description techniques must be strongly structured. They should have at least two additional parts capturing the set of required and assured properties:

Requirements: In the requirements part the designer has to specify the properties the document needs from its enviornment.

Assurances: In the assurances part the designer describes the properties the document assures to its enviornment, assuming its own requirements are fulfilled.

Once these aditional aspects are specified, the designer can explicitly state the dependencies between the documents by specifing for each document the assurances that guarantee the requirements. We call such explicit formulated dependencies *Requirements/Assurances Contracts*. Figure 4 illustrates the usage of those contracts. The three development documents include the additional requirements (white bubble) and assurances (black bubble) parts. Developers can explicitly model the dependencies between the documents by requirements/assurances contracts shown as double arrowed lines.

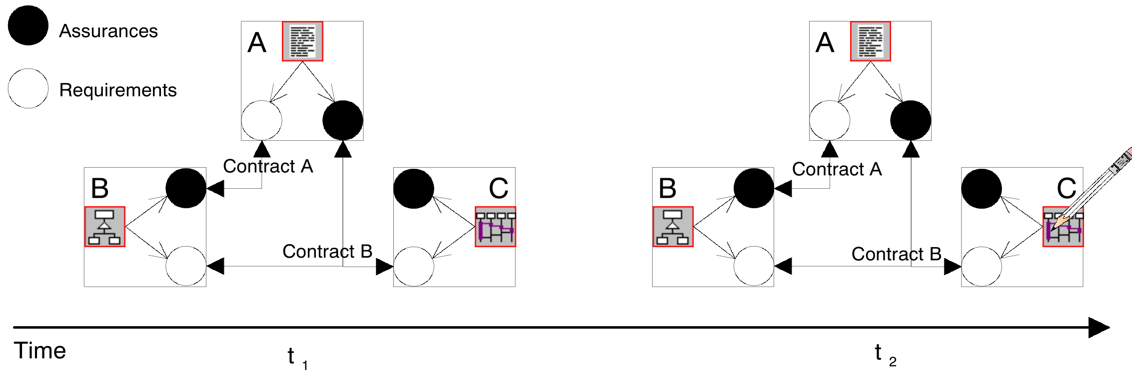

Figure 4: Requirements/Assurances Contracts between Documents

In the case of software evolution the designer only has to re-check whether requirements of documents, that relay on the assurances of the evolved document, are still guaranteed. For instance, in Figure 4 document `C` has changed over time. The designer has to validate whether `Contract B` still holds. More exactly, he has to check whether the requirements of document `C` are still satisfied by the assurances of document `A` or not. The advantages of requirements/assurances contracts come only fully to validity if we have adequate description techniques to specifiy the requirements and assurances within documents.

Therefore, we propose description techniques similar to those well-known in the field of 'design by contract' [Mey97] or 'contracts in object-oriented systems' [HHG90]. Design by contract is based on pre- and post-conditions that are restricted to the scope of a single object. In [HHG90] contracts are used to specify the collaborations between objects. These approaches take neither evolution nor componentware into account. Nevertheless, the description techniques can be reused to specifiy requirements and assurances in the proposed contracts. Another related approach are 'Reuse Contracts' [LSM97]. There, evolution conflicts in the scope of inheritance and call structures are discussed, but not conflicts in component collaborations, as sketched by the proposed requirements/assurances contracts. We still have to elaborate a consistent set of description techniques to support evolution in componentware at the best.

3 Conclusion and Further Work

In this paper, we have outlined the overall idea of an evolution based development methodology for componentware: it consists of a system model, description techniques, a software evolution process, evolution resistant architectures, and tools. During software evolution, a set of development documents are created. The refinement approach has severe drawbacks, for instance it does not support maintenace or change of user requirements. For that reasons we discussed the basic concepts of a development process based on software evolution. To provide guidelines and help for developers we introduced the concept of requirements/assurances contracts to model explicitly the dependencies between development documents. Thus, in case of the evolution of a single development document, we can point out the effects for the rest of the system.

The full version of the paper includes a more detailed discussion of the presented evolution development process. Additionally we provide more sophisticated graphical description techniques. A complete development example shows these description techniques in practice. Finally, we present mappings into technical componentware infrastructures like CORBA, DCOM, or Java Enterprise Beans. Further work has to be done concerning a complete embedding of the concept of evolution into a formal componentware methodology. Moreover, we have to provide mappings from the presented description techniques into this formal model. At last, we have to develop tool support and provide a set of evolution resistant architectures.

Acknowledgements

We thank Klaus Bergner, Manfred Broy, Marc Sihling, and Alexander Vilbig for interesting discussions and comments on earlier versions of this paper.

References

[ABD+99] Dirk Ansorge, Klaus Bergner, Bernd Deifel, Nicholas Hawlitzky, Andreas Rausch, Marc Sihling, Veronika Thurner, and Sascha Vogel. Managing componentware development – software reuse and the V-Modell process. In *Proceedings of CAiSE '99*, Lecture Notes in Computer Science. Springer-Verlag, 1999.

[BDD+92] Manfred Broy, Franz Dederichs, Claus Dendorfer, Max Fuchs, Thomas Gritzner, and Rainer Weber. The design of distributed systems - an introduction to FOCUS. Technical Report TUM-I9203, Technische Universität München, Institut für Informatik, January 1992.

[BDRS97] Manfred Broy, Ernst Denert, Klaus Renzel, and Monika Schmidt (eds.). Software architectures and design patterns in business applications. Technical Report TUM-I9746, Institut für Informatik, Technische Universität München, 1997.

[BHKS97] Manfred Broy, Christoph Hofmann, Ingolf Krüger, and Monika Schmidt. Using extended event traces to describe communication in software architectures. In *Asia-Pacific Software Engineering Conference and International Computer Science Conference, Hong Kong*. IEEE Computer Society, 1997.

[Bis92] Bischofberger W. R. and Pomberger G. *Prototyping-Oriented Software Development - Concepts and Tools*. Springer-Verlag, 1992.

[Bra94] Bran Selic and Garth Gullekson and Paul T. Ward. *Real-Time Object-Oriented Modeling*. Wiley & Sons, 1994.

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

[Bro98] Manfred Broy. A logical basis for modular systems engineering. Internal paper, 1998.

[BRSV98a] Klaus Bergner, Andreas Rausch, Marc Sihling, and Alexander Vilbig. A componentware development methodology based on process patterns. Technical Report I-9823, Technische Universität München, Institut für Informatik, 1998.

[BRSV98b] Klaus Bergner, Andreas Rausch, Marc Sihling, and Alexander Vilbig. An integrated view on componentware - concepts, description techniques, and development process. In Roger Lee, editor, *Software Engineering : Proceedings of the IASTED Conference '98*. ACTA Press, Anaheim, 1998.

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

[Har88] D. Harel. On Visual Formalisms. *Communications of the ACM*, 31(5):514–531, May 1988.

[HHG90] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. *ECOOP/OOPSLA '90 Proceedings*, pages 169–180, October 1990.

[IAB98] IABG. Das V-Modell, `http://www.v-modell.iabg.de/`, 1998.

[Iva99] Ivar Jacobson and Grady Booch and James Rumbaugh. *The Unified Software Development Process*. Addison Wesley, 1999.

[Jac92] I. Jacobson. *Object-Oriented Software Engineering — A Use Case Driven Approach*. Addison-Wesley, 1992.

[Jav99] JavaSoft. Enterprise JavaBeans website, `http.//java.sun.com/products/ejb/`, 1999.

[LSM97] Carine Lucas, Patrick Steyaert, and Kim Mens. Managing software evolution through reuse contracts. Technical Report vub-prog-tr-97-01, Vrije Universiteit Brussel Faculteit Wetenschappen, BELGIUM, 1997.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

[Neu99] Gregor Neumann. 500 Europa: Der Club der Innovatoren. *Information Week*, pages 10–12, January 1999.

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

[OMG99] OMG. OMG Unified Modeling Language Specification. Version 1.3, Object Management Group, 1999.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. PhD thesis, Technische Universität München, 1996.

[Szy97] Clemens Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison Wesley Ltd., 1997.