

A different notion of component

position paper for OMER'99

Harald Störrle

Ludwig-Maximilians-Universität München
Oettingenstr. 67, 80538 München, Germany
++49-89-2178-2134
stoerrle@informatik.uni-muenchen.de

May 7, 1999

1 Introduction

Component based software development (CBSD) can greatly improve the quality and effectiveness of the development of software. This is particularly important for control software for embedded and real-time Systems (ERTSes), that are particularly sensitive to malfunctions and, once deployed, particularly expensive to modify/correct. So, unsurprisingly, methodological approaches with an emphasis on these application areas have long since been occupied with aspects of component technology. For instance, HOOD and ROOM (aka. UML/RT) provide abstractions similar to the notion of component as understood in classical software development today (i.e. subsystems, actors and capsules).

So far, however, little attention has been paid to the interactions of the development *process* and the notations available. Current component technologies define components basically as executable code only. Many (software) engineering activities, however, also require other types of information on a piece of software, that is to be reused: code reuse is only one possibility for reuse. In fact, empirical studies strongly suggest, that it is far superior to reuse documents from earlier phases of software development (see the empirical data quoted in [12, p. 83]).

Therefore, a different notion of component is proposed. Some examples are given where a CBSD-process differs from an ordinary software development process, and which informations would have to be included in the notion of component in order to support them.

2 A critique of today's component technology

Embedded systems pervade manufacturing and infrastructure. Thus, the failure of an ERTS is potentially disastrous, and so the quality requirements for the development of ERTSes are much higher than those for desktop software. Quality, however, is expensive and takes time. Components (i.e. architecture-level entities) have been proposed as a solution to this problem (see [12] on all aspects of reuse).

- Being used many times, the cost of thorough testing or even formal verification is shared among all users, and so becomes more affordable.
- Using prefabricated parts should reduce the amount of code to be written, and thus hopefully speed up development.

Current component-technologies such as Java Beans or COM, however, fail to satisfy the expectations raised. Shortcomings can be identified in three areas:

- **too small:** the abstractions they provide are too fine grained, they are on the class or package level rather than on the subsystem level. Also, no abstractions are provided for larger levels of abstraction (see [13]).
- **too late:** they are concerned with the implementation phase of SW development only. A Bean or COM-component contains the information necessary to program with

it, but not the information needed to use it in the analysis and design. However, reuse is applicable already to the documents of the analysis and design phases. And it is evident that components can be used effectively only their usage is planned early on in the development. This is already partially true for today's components, but it is inevitable for larger components (subsystem level or above). Since many authors rightfully stress that components are supposed to be self-contained entities, they should be self-contained in all phases of development, including the early ones.

- **too soft:** They do not only provide not enough information, but the little information they do provide is also too soft: anything beyond the signature of a component is an unstructured natural language comment, at best. Thus, it can not be used for semantic based tool support (such as focused retrieval and checking for compatible communication behavior).

Beyond these three issues are aspects of a larger and even more severe shortcoming, the lack of a process for CBSD. Obviously, a CBSD-process is in many respects different from a conventional development process:¹

- Apart from the desired product, the development process also produces new components, change requirements (bug fixes, improvements, etc.) for existing components and requests for new components.
- Existing components must be considered very early on in the development, i.e. there must be a search-and-evaluate-activity in the design phase, possibly already in the analysis or even earlier.
- To be practical, the search and evaluation of components has to be done with extensive machine support. Thus, the functionality, dependencies and interfaces of a com-

ponent have to be described in a way that can be exploited by tools.

Today's component technologies, however, are not intended to support these activities: they are programming components rather than software engineering components. Therefore, in this paper a new kind of components is proposed to improve in this area. For want of a better name they are called Software Engineering Components (SEC). In [2] a somewhat similar yet less powerful notion for the small to medium levels of abstraction has been proposed under the name of component package.

3 Software Engineering Components

A SEC consists of several *views*, featuring information for different development activities. Which views are present will depend on the precise purpose they are put to, and the defining characteristics one accepts for components, but in general one would expect at least the following:

- the code view represents the executable code for the component;
- a description of the functionality of the component, relevant system boundaries, and intended usage;
- the architectural representation of a component, i.e. its behavior (if it is elementary) or its internal structure, its interfaces (signature and behavior), and dependencies on other components.

For embedded systems, one might also have a deployment view, that determines what part of the functionality (i.e. which components) is to be implemented as hardware, and what part is to be implemented as software. Other possible views might be: a samples view (e.g. pointers to sample applications with the component, experience reports), a performance view (e.g. load figures relative to number of users and so on), a test view (e.g. test cases, partitions of input data, error density and so on). The notations for expressing the views are not necessarily fixed, but standard techniques such as UML probably are a good choice. Thus, a primitive realization of SECs themselves might simply be

¹Some first steps towards development process and methodologies for component based systems have been made in [3] (focus on organizational issues), [4] (proposing a fractal process model) and [2] (focus on practical questions and "process patterns"). Similar work has been done in software evolution (see [6]), a term coined by L.J. Arthur. It is currently not clear, however, how much of this prior work can be adapted to the case of component based software development.

a stereotyped UML-package diagram (see Figure 1), with a fixed set of views with (a set of) appropriate diagram types or tagged values. The individual views in turn might be implemented as follows:

- The code view would contain a set of executables for different types of environments (choices of hardware, operating system, windowing systems, class libraries and so on). Also, different revisions could be provided in order to guarantee backward compatibility. For some types of license agreements, the executables might be supplements with (excerpts of) the source code, too. The code view would also document dependencies to external software, such as operating system calls.
- The functionality view would be expressed as a use case diagram, some role and use case descriptions, and comments about intended usage (possibly supplemented by pointers to previous usages of the component). While informal, this description is not unstructured (see below).
- The architecture view is probably the most valuable view of all. The notation of the ROOM method (see [7, 5, 8] called architecture diagrams in the remainder), has been used successfully for similar purpose in the past. A ROOM-capsule is thought to represent a component, ROOM-layers are not used yet. In [9, 11, 10]² a Petri-net semantics of UML/RT and some analysis algorithms have been proposed.

Depending on the intended application, other or modified views might be useful, too. In the context of real-time applications, information on temporal behavior (e.g. response times etc.) might be added, in the context of embedded systems, the distribution of functionality on hardware or software might be worth including.

4 Some SE-activities supported by SECs

In this section, it is sketched how the additional information of SECs might be used for some different non-programming Software Engineering

²All of these papers are currently submitted, but preprint versions may be obtained from the author. They will also appear as technical reports later on.

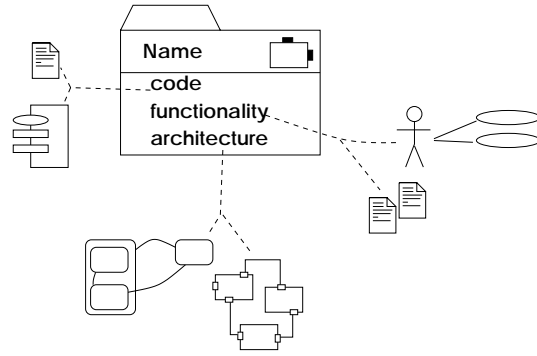


Figure 1: Template of a SE component.

activities (see [1] for some hints of other activities requiring similar support).

4.1 Design debugging

One particularly valuable potential of SECs is the possibility of *design debugging*, i.e. the simulation and analysis of designs (rather than code). Obviously, this requires a formal semantics, appropriate analysis techniques and suitable tools to go along with them. The degree of elaboration and completeness of the semantics directly depends on the precise type of analysis to be done. Among other features, they allow to check the following properties:

- Do the interfaces of two capsules' ports fit together, not just by the sets of signals they wish to exchange, but by the protocols they use?
- Does an interaction diagram really represent a sample run of a protocol?
- Is there a run where a given protocol will deadlock? Which one?
- Apart from the protocols, are there other circumstances (arguments, schedulings) where a system of components reaches an unwanted state (deadlock, livelock etc.)?

Of course, verifying these properties covers only logical errors, but these are the most expensive and difficult anyway. Also, it is clear, that for many systems properties as those mentioned are not (efficiently) decidable, so that user interaction (e.g. introduction of abstractions) is necessary. Yet, a set of useful properties can often be checked for automatically, and further additions

to this set can be expected from ongoing work. Two possible applications of design debugging techniques are sketched below.

4.2 Restructuring

Another activity is facilitated that is typical for CBSD: the *restructuring* of systems. This requires suitable equivalences or implementation relations on components to determine whether one component can be replaced by another. Consider the following example: an architectural design of a distributed system that has to be adapted to meet some load requirements. In distributed systems, response times and throughput often critically depends on the the distribution, i.e. what is done where, and how much network traffic is involved with a particular distribution. So suppose the desired speedup is to be achieved by a new distribution. As communication across distribution boundaries is asynchronous while communication within one machine can be considered to be synchronous. Thus, a new distribution of the same components will probably introduce new distribution boundaries, and thus turn some synchronous communications into asynchronous ones, and vice versa. The logical effect of this change can be examined based on the formal semantics of architecture diagrams.

4.3 Evaluation of components

Using components requires the evaluation of components at some point. In the previous sections, some formal analysis methods have been hinted at, that can help decide whether or not a component is indeed applicable in some given context. However, there is more to a component's evaluation: it also relies on non-functional aspects such as reliability, conformance to legal requirements, and temporal properties. These could be supported by including in a component experience reports, legal assessments and empirical measurements of the timing, respectively.

4.4 Support for classification and retrieval

Before one can use a prefabricated component, it has to be retrieved first. There has been a lot of work on software reuse that can be applied

here (see for instance [12]), though in the reuse-world, little or no difference is made between the different types of documents, and their semantics is not exploited in retrieval. Additional cues for classification and search can be obtained from the fact that SECs are *structured bundles* of documents rather than single documents. So, the views (and individual items in them) can act as a kind of classification facets, reducing the number of items precisely matching a query (i.e. increasing *search precision*). Thus, a greater number of rough matches may be accepted (i.e. increased *search recall*), and so the overall search quality (ratio between found matches and possible matches) can be improved. Put in a nutshell: A search guided predefined categories such as a role is better focused and yields higher retrieval precision than an unguided search.

5 Further work

We are currently investigating, what impact on the development process the activities described above will have, and what other activities call for support by semantics, algorithms and tools. As a support for case studies in this area, a prototype implementation is currently being prepared by some students at the chair of Prof. Wirsing, Univ. München.

References

- [1] Klaus Bergner, Andreas Rausch, Marc Sihling, and Alexander Vilbig. A componentware development methodology based on process patterns. In *Proceedings of the 5th Annual Conference on the Pattern Languages of Programs, (PLOP)*, 1998.
- [2] Desmond F. D'Souza and Alan Cameron Wills. *Objects, Components and Frameworks with UML. The Catalysis Approach*. Addison-Wesley, 1999.
- [3] Adele Goldberg. A reuse business model. *Software Concepts & Tools*, 19(1):11–13, 1998.
- [4] Wolfgang Hesse. Baustein-orientiert statt Phasen-zentriert: Neue Entwicklungsmethoden erfordern neuartige Vorgehensmodelle. In *Workshop Anwendungen von objektorientierten Entwicklungsstrategien und*

deren Unterstützung durch Vorgehensmodelle, 1998. Published in Rundbrief 2/1998 of the Fachausschuß 5.1 of the Gesellschaft für Informatik.

- [5] Andrew Lyons. UML for Real-Time (Overview). Technical report, ObjecTime INC., see www.objecttime.com, 1998.
- [6] Robert Moreton. *A process model for software maintenance*, pages 29–33. IEEE Computer Society Press, 1996.
- [7] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real Time Object Oriented Modeling*. Wiley & Sons, Inc., 1994.
- [8] Bran Selic and James Rumbaugh. Die Verwendung der UML für die Modellierung komplexer Echtzeitsysteme. *OBJEKTSpektrum*, (4):24–36, 1998. also published in English as: *Using UML for Modelling Complex Real-Time Systems*, technical whitepaper of Rational Software Corp. & ObjecTime Ltd.
- [9] Harald Störrle. Analyzing behavioural designs. submitted to ESEC, 1999.
- [10] Harald Störrle. Towards a denotational semantics of dynamic aspect of UML/RT. submitted to UML, 1999.
- [11] Harald Störrle. Towards a semantic for sequence diagrams. to appear in Proc. 9. GI/ITG-Fachgespräch formale Beschreibungstechniken für verteilte Systeme, 1999.
- [12] Andreas Zendler. *Konzepte, Erfahrungen und Werkzeuge zur Software-Wiederverwendung*. Tectum Verlag, Reihe Softwaretechnik, Nr. 1, 1997. Schriftenreihe des FAST.
- [13] Andreas Zendler. Foundation of an taxonomic object system. *Information and System Sciences*, (40):475–492, 1998.