Componentware – State of the Art 2003

Background Paper for the *Understanding Components* Workshop
of the CUE Initiative
at the Univerità Ca' Foscari di Venezia
Venice, October 7th-9th 2003

Gerd Beneken Ulrike Hammerschall Manfred Broy
María Victoria Cengarle Jan Jürjens Bernhard Rumpe
Maurce Schoenmakers

Institut für Informatik, Technische Universität München
Boltzmannstr. 3, 85748 Garching, Germany
{beneken, hammersc, broy, cengarle, juerjens, rumpe, schoenma}@in.tum.de

September 29, 2003

Abstract

Components and component-based technologies (componentware) are well-known and widely used in software development. There is a large amount of work and research in componentware. The number of available componentware approaches increases steadily and it is quite difficult to keep track of current trends in this area. In this paper, we survey the current state of the art in componentware, introduce and compare several well-known componentware approaches and classify them according to outstanding characteristics. We discuss a list of open issues in resarch and practical use of componentware and offer some proposals for further development. In our practical considerations we focus on embedded systems and business information systems because most of our partners in industry work in one of these two domains. We hope to start a broader discussion on componentware and to get a common understanding, which open issues are most important in research and industry (as a research agenda).

Contents

1 Introduction

The use of components and componentware in software development is in vogue today. Research and industry investigate and work with components and component technologies. Many approaches have been invented for component-based software engineering. Viewed from a distance, components seem to be a well-developed and ready-to-use technique. A closer look, however, shows a quite different scene. Everyone talks about components but in many cases people talk about and address different issues. Surprisingly after 30 years of research there is still no consensus about the question, what exactly a component is.

The idea behind components can be traced back to a paper, published by M.D. McIlroy [87] at the NATO conference in Garmisch in 1968 about the idea of mass-produced software components. However, since McIlroy's paper, component notions and definitions developed in various, and in some cases contradictory directions. Today, Szyperski's definition is cited most frequently [115]:

> A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

Szyperski emphasizes the composition and deployment of components. Architecture Description Languages (ADLs)[108] view components as architectural entities. Process models like [56] consider components at design time. These were three examples of different perspectives in componentware. There is no integration of approaches in componentware. The increasing number of incompatible concepts mirrors terminological inconsistencies.

In this paper, we do not intend to clarify these questions. Our goal is rather to provide the reader with a comprehensive overview of current approaches and work in progress in industry and research in the area of components and componentware. We focus on embedded systems and business information systems. That does not mean that other domains are less important.

In particular, it is our concern to initiate a discussion about open issues in componentware by providing and surveying the fundament.

1.1 Current Trends in Software Development

Current software development is characterised by many different trends. In this section we sketch some of the most important trends to motivate the success and the widespread use of componentware.

1.1.1 Complexity of Software Increases

During the recent years more and more different application domains such as mobile computing, automotive, telematics, telecommunications and e-Business emerged. In parallel, the importance and complexity of software in all domains increased. This is due to a couple of facts:

- Customers and markets require new and more complex functionality and business processes: One basic goal of successful companies is to provide customers with more online services to increase their contentment. For example, more and more automotive companies offer customers an online facility to design and order a new car, including a dependable delivery date. The mapping of such business processes to software is quite ambitious and causes high efforts in software development.
- The degree of automation in existing business processes increases. To reduce costs companies tend to automate manual tasks and integrate them with existing applications to automated (online) workflows. This causes increasing integration efforts. For example, in telematics many mobile services (such as traffic information, hotel reservation, flight booking, GPS) are combined to new and more complex applications.

- The complexity of software in embedded systems is significantly increasing. Up to 40 percent of the development effort for an upper-class car is spent in car-it (such as in drive by wire, driver assistance). Today, a car may contain up to 80 control-units that are cross-linked.
- Internationalization of business demands reliable online market places. This is a relatively new domain. Companies negotiate and trade via the internet. Online market places require complex (negotiation) algorithms in software to meet the requirements.
- The degree of cross-linkage is increasing in software systems. Questions of interest are especially the handling of system failure, unreliable network connections and communication issues.

Today software complexity is already at a frightening high level. However, it is quite probable that with the development of new business areas, also software complexity still will further increase.

1.1.2 Mass-produced Software Components are Available

McIlroy's [87] vision has largely become true. M.D. McIlroy requested in his article *Mass-Produced Software Components* at the NATO conference on software engineering in 1968:

> ... yet software production in the large would be enormously helped by the availiability of spectra of high quality routines, quite as mechanical design is abated by the existence of families of structural shapes, screws or resistors. ...

Today we work with catalogs of reusable small to large scale *components*. This fulfills a part of McIlroy's vision. Consider the Java and the C++ programming languages. Lots of reusable components enable us to develop software at a higher level of abstraction:

- Small scale programming language libraries such as *java.util* and the standard template library *STL* in C++ offer basic classes such as String, Map, List, Array, Stack, Queue etc.
- Technology abstractions such as *JDBC* or *ODBC* and the related exchangeable database drivers offer a common abstraction of proprietary database interfaces.
- Increasingly many special purpose software components of high quality such as XML-parsers (www.apache.org), X-Path analyzers, regular-expression interpreters, database access layers (www.openquasar.de), and specialized user interface components are available and in widespread use.
- Large scale standardized frameworks such as *javax.swing*, *SWT*, *MFC*, *OSF/motif* or *QT* are available.
- Large scale standardized containers such as database engines (*SQL 92*), web-browsers (*W3C-standards*) and application servers (*EJB, Servlet specifications*) are available.

During the last years, a number of market places for commercially available components showed up. Well-known examples are *www.componentsource.com* or *www.internetcomponent.com*. On the other hand, an increasing number of OpenSource communities develop freely available components that have rich functionality and a surprisingly high quality. The Jakarta project *www.jakarta.org* might be mentioned as one of the most successful examples. However, the use of ready-to-use components is not as widespread as it might be. This is mainly due to the lack of reliable standards. Documentation, component structure, interface definitions, and behaviour descriptions are proprietary. They are highly dependent on their providers.

1.1.3 Increasing Level of Standardization

Although standards for components are still missing, standardization work for software development in general is making progress. Especially in the enterprise information systems domain, standards at different level of abstraction are available.

- Standardized basic runtime environments such as Sun's Java Virtual Machine (JVM) or the Common Runtime Environment (CRE) from Microsoft.
- Standardized component frameworks such as J2EE and CORBA and their containers (EJB, CCM).
- Standardized application service interfaces that provide access to basic services such as transaction service, persistency service, event service, naming service and many more.
- Standardized application programming interfaces to technical device drivers (JDBC)

Figure 1 shows possible standardization level along an increasing level of abstraction.

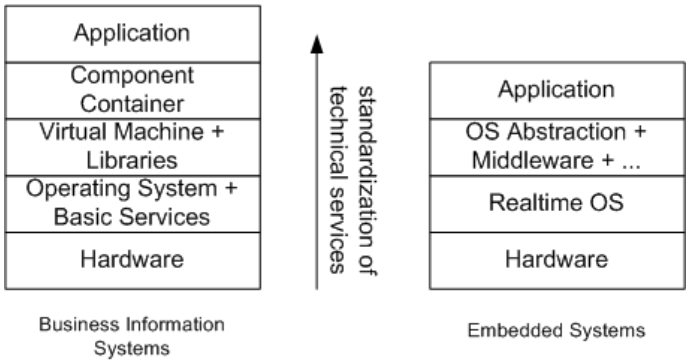

Figure 1: Standardization of Infrastructure

Several standards in software-intensive areas exist on the market. However, standards in the embedded systems domain are rare. Embedded software faces some unique constraints not found in enterprise systems:

- Limited resources: embedded devices typically have a limited amount of resources (memory, processing power, electrical power). The software has to be optimized to handle the situation.
- Real-time requirements: the majority of embedded software systems interact with their environment in some way where interactions have to happen within specific time constraints.
- Hardware integration: in many embedded systems, some interaction with hardware devices is necessary. Software has to deal with low level aspects of the hardware.
- Reliability: embedded systems are often part of safety-critical systems where failure is not acceptable.

Because of these requirements, many embedded systems are still developed manually. Standards are still rare.

Even if current standardization is still unsufficient, its necessity is well-known and standardization work is in progress.

1.1.4 Rapid Change of Infrastructure

During the recent years technical progress in software was quite overwhelming. Almost every year created its own new technologies, standards and buzzwords. Java was released in 1995, eight years later more than 3.000.000 developers used Java worldwide. SUN's component technology – Enterprise JavaBeans – was introduced in 1998, many business information systems and e-commerce applications are built upon this technology, although the standard itself is still subject to major release changes. Microsoft .NET and C# as a new programming language were presented in 2000. Currently, .Net is main competitor to EJB. XML was defined in 1998. WebServices, based heavily on XML, are still under standardization, however widely in use. On the other hand, companies that depend on the CORBA standard run into problems because of vanshing CORBA support from CORBA platform providers site.

Obviously our programming environments and technical infrastructures are changing rapidly. The currently most widely used infrastructures for distributed applications EJB and .NET are less than 5 years old! Common programming languages such as Java, VisualBasic and C# are still under development. Downward compatibility is not always guaranteed. With the upcoming WebServices and its new APIs and services major changes in architecture and design of distributed systems are likely.

These are some examples where software developement stands today and where the trend might go. In short, software development still lacks reliable standards and stable infrastructures. The next section discusses why componentware is part of a solution.

1.2 Why do we need Componentware?

Software is immaterial and can be copied almost for free. However, very early in the computer science discipline, it became clear that development of software is a costly and error prone discipline. "Software Crisis" and "Software Engineering" where terms coined already 1968 at the famous NATO conference in Garmisch.

Since then the field has made enormous technological and methodological progress: While the focus at first was in the development of single data structures and algorithms, it soon became clear that the management of the inherent complexity of many simple functionalities also has to be dealt with. Therefore, structured analysis and design techniques [120] have been developed and dominated software development for more than a decade. They have been followed by the object-oriented paradigm which did not allow any other paradigm (like functional or declarative) to become a key technique as well.

Both approaches, and various hybrids in between, have been developed to deal with specific needs and problems. Structured techniques helped to manage software development through hierarchical structures. The object paradigm allowed to increase reuse, encapsulation and independent development that for instance led to frameworks.

However, even though object oriented technologies are doing quite well in a number of ways, they also suffer from a several drawbacks:

- Objects only compose and cooperate if written in the same language.
- Object interfaces are one-directional: Only the incoming interface (import) is described, the outgoing interface (export) remains mostly implicit.
- Object cannot be deployed independently.
- Objects are tightly coupled, as they execute in the same process and data space.

Today's software systems are distributed over networks, interact and execute concurrently. Although in Nygaard's and Dahl's Simula 67 concurrency was explicitly addressed by the concept of coroutines,

most object-oriented languages do not include concepts for concurrency at all or add such concepts in brute force manner such as Java where the idea of threads is not well-integrated at all with the ideas of abstraction in classes. So far the issue of components operating concurrently in a distributed manner is not well-addressed by object orientation.

A wave of integration of applications took place, both in the business domains as well as in the embedded systems area (e.g. automotive). Now, a worldwide integration of applications can be seen in business, telecommunication and in the not too far future also in embedded area. It is therefore inevitable that systems of the future:

- run in changing and adapting environments,
- can be exchanged or newly deployed partly without failure of the whole,
- allow applications written in different languages, on different technologies, and in different operating systems to cooperate, and
- explicit interfaces allow to connect and decouple cooperating parts of the overall system.

Components are providing these additional concepts. Furthermore, through their explicit interfaces and their interoperability between different languages, they allow to

- **increase reuse** through approaches to buy commercial-off-the shelf-components instead of developing the same functionality over and over again,
- **increase quality of the result** through reusing components that have been in use already for some time and are therefore assumed to be relatively error free, and
- **increase outsourcing capabilities**, thus allowing to involve experts from various domains to derive components for their application area and combine those.

In turn, this leads to better defined and clearer software architectures with clear interfaces. Thus, a component based system can be extended more easily by new components, maintained and adapted to new requirements.

In summary these techniques will lead to a cost reduction and are therefore an important economic issue, at least for long living systems and companies with a larger landscape of integrated applications. However, the usage of component techniques is not for free. There is a considerable technical overhead to deal with, a certain architectural style has to be used, components that are bought off the shelf or reused must be adapted to the actual needs, etc. Therefore it is important to have an appropriate modelling technique at hand that allows a clear interface specification, provides a rigorous relation between interfaces and implementation and can be used for semantically sound and well-defined composition.

1.3 Overview of this Paper

The introduction motivates, why componentware is worth considering. Current trends in software industry like the increasing complexity require componentware technology.

In section 2 we survey the current state of the art in componentware, introduce and compare several well-known componentware approaches and classify them according to outstanding characteristics.

According to the classification scheme in section 2 we present open issues every class:

- Section 3 surveys formal approaches to componentware.
- Section 4 treats description techniques (especially UML) and the description of components.

- Section 5 is concerned with software architecture, scale and granularity and the composition of components.
- Components in the software life cycle are discussed in section 6. One important question is whether a component is one concept, which is refined through the phases of software development or there are different concepts in every phase.
- Toolsupport is surveyed in section 7.
- Componentware can be applied to different domains. Embedded systems, business information systems and other domains have different componentware requirements, for instance the modeling of time and concurrency. Componentware is viewed from the perspective of these domains in section 8.
- Componentware should be treated as substantial part of an engineering discipline, why and how is discussed in section 9.

2 A Survey of Current Componentware Approaches

In this section we assess the current state of the art of componentware methodologies. We define a classification based on outstanding characteristics. The classified approaches are introduced and compared and missing aspects are identified.

2.1 A Definition

A survey of componentware has to be based on a clear and reliable statement of what components and componentware are. Therefore, the first task is to find a reasonable definition of a notion of componentware. Similar to components, so far no general accepted definition exists. For our classification, we fall back on a simple but useful definition from Bergner et al. [45]:

> Componentware is concerned with the development of software systems by using components as their essential building blocks.

As a result, identifying characteristic of componentware is support of a software-component notion. According to this definition, the range of possible componentware approaches can reach from simple technical component concepts to complex component-supporting process models.

2.2 Categories of Componentware

To organize the vast range of quite different types of existing componentware approaches, it is crucial to find a classification approach. In this section we introduce a classification schema based on identifying characteristics of componentware. Bergner et al. [45] claim several important characteristics a conceptual componentware methodology should support. Figure 2 outlines these characteristics:

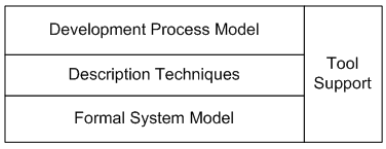

Figure 2: Componentware Characteristics according to Bergner et al.

Formal system model and theory : A well defined mathematical formalism, used to unambiguously express the basic building blocks and their relations as well as their properties: component, interface, connector and configuration.

Description technique : Textual or graphical notations hiding the underlying formalism to the user. Artifacts are diagram types for (graphically) representing static and dynamic aspects.

Development process : A component oriented process model to structure the development process. It defines tasks and their results as well as roles that are responsible for the fulfilment.

Tool support : Tools support the use of specifc description techniques and ideally the complete development process.

We use this model as a starting point for our componentware classification. Although it covers most of the characteristics, current componentware approaches incorporate, two aspects are missing:

- A well-defined approach for design and documentation of the system's architecture. This includes guidelines for refinement, component composition, quality properties and the over all structure of the system.
- The tool-support characteristic covers a wide range of products and need to be refined.

To address these aspects, we propose a refinement of the model in Figure 2 as depicted in Figure 3.

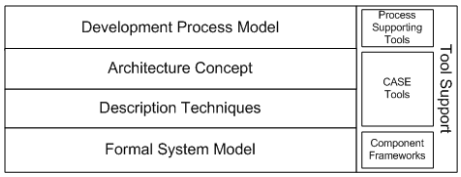

Figure 3: Extended Componentware Characteristics

The refinement of tool support shows three relevant types of tools. Each of them supports component-based software development. However, they are of different relevance for the classification scheme. On one hand, process supporting tools usually are independent of any design and programming paradigms, hence they are not considered further. CASE tools, on the other hand, can support component-based development, but depend heavily on the supported description technique and its underlying paradigm. Currently there exist a couple of competitve technical component frameworks some of which we will introduce later in this section.

The componentware categories we identified from the model are as follows:

- formal approaches
- descriptive approaches
- architecture centric approaches
- process centric approaches
- tools and technical componentware frameworks

In the following, we introduce several componentware approaches and classify them according to the identified componentware categories.

2.2.1 Formal Approaches

Formal componentware approaches are usually developed at academic and research institutes.

FOCUS [51, 52]: FOCUS is a mathematical and logical theory of the modular specification, design, refinement, and abstract interpretation of composed systems with an emphasis on component-based modular system development. It provides a clear notion of a component and ways to specify, manipulate and to compose components. This theory introduces a mathematical model of a component with the following characteristics:

- A component is interactive and concurrent.
- It interacts with its environment exclusively by its interface formed by named and typed channels. Channels are communication links for asynchronous, buffered message exchange. A component encapsulates a state that cannot be accessed from the outside directly.
- A component receives input messages from its environment on its input channels and generates output messages to its environment on its output channels.
- A component can be underspecified and thus nondeterministic. This means that for a given input history there may exist several output histories representing possible reactions of the component.
- The interaction between the component and its environment takes place concurrently in a global time frame. In the model, there is a global notion of time that applies both to the component and its environment.
- Components are specified by logical expressions relating the input and output communication histories on their channels.
- State machines describe the input and output histories by operational means.
- Compositions of components are done hierarchically and form system architectures.

Based on the ideas of an interactive component FOCUS defines forms of composition. It basically introduces only one powerful composition operator, namely parallel composition with feedback. This composition operator allows us to model concurrent execution and interaction of components within a network. For the systematic stepwise development of components FOCUS introduces the concept of refinement. FOCUS offers three refinement relations namely property refinement, glass box refinement, and interaction refinement. These notions of refinement typically occur in a systematic top down system development.

The compositionality of FOCUS guarantees that refinement steps for the components of a composed system realize a refinement step for the composed system. As a consequence, global reasoning about the system can be structured into local reasoning about the components. Compositionality relates to the notion of modularity in systems engineering.

ROOM [107]: Real-Time Object-Oriented Modeling ROOM, published in 1994, is a method for specification, design and construction of distributed real-time systems. The method comprises three elements:

- a modeling language to express high-level properties of the system,
- modeling heuristics as informal guidelines on how the modeling language can be used in particular situations, and
- a development process.

Even if ROOM seems to incorporate characteristics like other process centric approaches, there is a major difference: ROOM underlies a strict formal semantics.

Basic building blocks of ROOM are called *Actors*. Actors can be viewed as a kind of components. Typically they encapsulate an active thread or process as well as state information. Actors communicate with each other via ports. A port defines a set of messages that the actor provides or requires. In general an actor may have multiple ports. Ports at the layer boundary are called SAPs (Service Access Points). Each port has an associated protocol. For the communication between two actors, a binding has to be established between an outgoing port of one actor and a compatible incoming port of another. Compatible ports are called conjugated.

Developed before components and component-based development became buzzwords, the method incorporates many of these ideas. Even if ROOM does not support components explicitly, it can be

viewed as a classic componentware approach: Actors represent self-contained autonomous components, that can be realized as hardware components as well as as software components. Ports represent independently defined interfaces that allow components to be connected in arbitrary configuration.

It might be an old approach, successful at its time. Nevertheless, it still has a big impact on modern software development. The component concepts shipped with UML 2.0 (also called the UML RT profile) are mainly based on the ROOM concept of actors.

Critical aspects

Formal componentware approaches always have the problem not being accepted outside the research community. However, ROOM was adapted for practical use and it has been applied in many projects. Other approaches never really bridged the gap between science and industrial application. This might be due to the complexity of the underlying formal mathematical models.

2.2.2 Descriptive Approaches

The class of descriptive techniques comprises two types, graphical and textual approaches.

UML [49]: The Unified Modeling Language UML is an object-oriented graphical notation used for modeling artifacts. UML may be used in a variety of ways to support different development approaches, but in itself it does not specify any process. At the first place, UML is object-oriented even if it includes component diagrams. The component notion of the current UML version views components as physical, executable units rather than logical building blocks. UML can be used very well as design language at the class level, but the drawback is, that it is not suited as an architecture description language. This issue is addressed with the next UML version. UML 2.0 is expected to be published at the end of the year 2003. With UML 2.0 components will be design-time as well as run-time entities. The specification draft [99] of UML 2.0 states that:

> a component represents a modular part of a system, that encapsulates its contents and whose manifestation is replaceable within its environment. ... A component is modeled throughout the development life cycle and successfully refined into deployment and run-time components.

ADLs: According to [3], Architecture Description Languages ADLs are component-based techniques, intended to describe software architectures. In general, they are used to define and model system architecture prior to system implementation. In addition to identifying the components and connectors (interactions) of a system, ADLs typically address:

- Component behavioral specification: ADLs are concerned with component functionality. ADLs typically provide support for specifying both functional and non-functional characteristics of components. (Non-functional requirements include those associated with safety, security, reliability, and performance.) Depending on the ADL, timing constraints, properties of component inputs and outputs, and data accuracy may all be specified.
- Component protocol specification: Some ADLs, such as Wright [65] and Rapide [82], support the specification of relatively complex component communication protocols. Other ADLs, such as UniCon [109], allow the type of a component to be specified (e.g., filter, process, etc.) which in turn restricts the type of connector that can be used with it.
- Connector specification: ADLs contain structures for specifying properties of connectors, where connectors are used to define interactions between components. In Rapide, connector specifications take the form of partially-ordered event sequences, while in Wright, connector specifications are expressed using Hoare's Communicating Sequential Processes (CSP) language [70].

ADLs were developed to address programming in the large; they are well-suited for representing the architecture of a system or a family of systems. Most existing ADLs were developed in research

institutes. Especially the Software Engineering Institute at Carnegie Mellon University provides a multitude of different-purpose ADLs. However, until now ADLs do not play any major role in practical software engineering.

Critical aspects

Description techniques only cover a small part of a componentware technology. Especially UML might just be viewed as a vehicle to draw smart pictures of software. This is also due to the fact, that there is no formal model underlying the semiformal UML notation.

ADLs on the other hand have a formal basis in many cases, but the usage of ADLs is not wide spread. There are 5 or 6 major approaches but no conformance and no common understanding about what an ADL should incorporate. This might probably be a reason for the current restriction of ADLs to research.

2.2.3 Architecture Centric Approaches

Architecture centric approaches emphasize the architectural aspects of a component-based system. In this category we present one approach as an example.

Quasar[110]: The Quality Software Architecture Quasar has been developed at the sd&m AG since 1998. Quasar defines architectural concepts at two levels: The conceptual level includes general architectural principles and heuristics for the development of component-based systems, for example separation of technical aspects and business domain aspects. At the system level Quasar defines an explicit mapping of general concepts to a specific technical component framework. Quasar defines no explicit process, description technique, or formal system model.

2.2.4 Process Centric Approaches

Process centric approaches emphasize process models, roles, activities and workflows.

Catalysis [59]: Catalysis was one of the first component-based software engineering processes. Published in 1998 it already combines an iterative and incremental process with a component notion and UML concepts. Basic building blocks of Catalysis are objects and actions. Objects can represent everything, from business departments, machines, running software components to programming language objects. They can be structured hierarchically.

Actions are interactions that can occur between all types of objects. Similar to objects, actions can be structured hierarchically.

Zooming in and out of objects and actions defines two core principles of Catalysis: abstraction and refinement. The refinement relation enables the traceability of the model from a higher level of abstraction to a lower level. Another core principle is precision. A model has to be defined precisely, that means unambiguously at all level of abstraction. Catalysis does not propose a special process, but the use of suitable process patterns. This is to keep the approach applicable to different application domains.

In Catalysis, the component notion is not clearly defined. It is a kind of mixture between object and package. Catalysis defines a component as follows:

> ... a coherent package of software artifacts that can be independently developed, delivered as a unit and that can be composed, unchanged, with other components to build something larger.

The basic principles of Catalysis are still useful, nevertheless, the approach is difficult to understand and apply.

KobrA [41]: KobrA is a pragmatic approach for architecting large scale, component-based systems. It was published in 2001 as part of the KobrA project by Fraunhofer-Institute for Experimental

Software Engineering (IESE) and targets the development of product lines as well as single applications and systems. In KobrA, a system is viewed as a simple, generic and abstract black box. In an iterative process the level of genericity and abstraction are reduced and the system is decomposed into components.

Components are logical units of system structuring and are called Komponents (KobrA components). The approach distinguishes between Komponents (types), Komponent instances and run-time components. Komponents and Komponent instances are organized in trees: one tree of Komponent types defines the logical structure of the system, another tree of Komponent instances defines the run-time structure. The trees are developed and refined recursively during the development process. KobrA defines an informal mapping from Komponent instances to components of a technical component framework (EJB, COM).

UML Components [56]: UML Components is a process centric componentware approach with two main objectives:

- the definition of a simple process to identify and specify components and
- the definition of an application of the UML notation to support the process.

The process is an adaptation of the object-oriented Rational Unified Process (RUP) [80] to component-based development. In detail, the RUP phases *analysis* and *design* are replaced by new phases *specification*, *provisioning* and *assembly*. The approach defines activities and artifacts for component identification, interaction and specification.

UML components does not define a component notion itself, it rather refers to properties a component should incorporate. This emphasizes the pragmatism of the UML Components approach.

Business Component Factory [68]: Herzum and Sims focus on business information systems in their Business Component Approach. A business component is the core concept. A system is functionally decomposed into business components at analysis time. A business component represents an autonomous business concept through the phases of the development process. It clusters all artifacts, that are developed during the process. For example: prototypes of the user interface, entity-relationship models, use-case diagramms etc. are clusterd in a business component.

Herzum and Sims define three level of component granularity, that correspond to three types of components: Distributed components are low level components. They consist of some programming language classes and are remotely accessible. Business components consist of one or more distributed components. They may have a grapical user interface and database access. A system level component (business component system) is a group of business components, that implement a cohesive set of fuctionalities required by a business need.

The business component approach defines architectural viewpoints, that focus on different aspects of a system. Four viewpoints are defined: technical architecture viewpoint, application architecture viewpoint, project management viewpoint and functional architecture viewpoint. For example, *the functional architecture viewpoint is concerned with the functional aspects of the system, that is, with the specification and implementation of a system that satisfies the functional requirements [68].*

For a company adopting component-based development, Herzum and Sims propose a process called Rapid System Development (RSD). The process is organized v-like and has four developmend phases corresponding to four testing phases: Requirements, Analysis, Design, Implementation, corresponding to Distributed Component testing, Business Component (BC) testing, BC system testing and acceptance testing. Business components are already defined in the requirements-analysis phase or even before the start of the project. Requirements gathering and use-cases are organized around components.

Critical aspects

In general, process centric approaches cover most of the characteristics defined in the componentware model given in Figure 3. Usually they incorporate a description technique, are more or less tool

supported and define an informal mapping to existing component frameworks. But the problems are obvious:

Since the advent of UML as a standard description technique, almost all process centric approaches incorporate UML as a means of design and architecture modelling. However, UML is still (and will be at least until version 2.0) rather object-oriented than component-based. This leads to ambiguities between the process component notion and the mapping to UML constructs. The vague component notion of Catalysis [59] might be one of the victims of this mismatch. UML Components [56] has a more sophisticated solution for the problem. It uses the UML stereotype concept to define a mapping. Another often used approach is the modeling of components as objects. Since almost all approaches rely on the semiformal description technique UML, almost all approaches lack a formal system model.

All presented approaches claim to support a mapping to existing component frameworks. Usually they define an informal mapping from components to EJB or COM+. Nevertheless, the mapping does not include technical aspects of the target system. For example, to build high-quality systems with EJB that meet the performance requirements, the use of design patterns [37] is strictly recommended. A simple 1:1 mapping from logical to technical components does not lead automatically to adequate software systems [44].

2.2.5 Technical Component Frameworks

Unlike others, more conceptual approaches, technical component frameworks focus explicitly on the support of component development and the specification of concrete component runtime environments. During the recent years three major component frameworks have emerged: COM, EJB and CCM. Each of them will be introduced in this section.

COM [20]: The Common Object Model (COM), developed by Microsoft, is one of the oldest component frameworks still in use. COM, a descendant from the Object Linking and Embedding Technology OLE, was developed in 1995. Primarily intended as a simple local component technology, COM was extended in 1996 to the Distributed Common Object Model, DCOM. DCOM enables the use of COM components in a distributed environment. Currently, COM and DCOM are combined with the Microsoft Transaction Service MTS to COM+ and provide the component framework for the .NET platform. The basic idea behind COM+ is that all Microsoft deliverables, all services and applications may be packed as COM components and plugged easily into any Microsoft plattform. This leads to a very general but weak component notion: COM+ views a component as a simple binary entity with unique identifier and unchangable interface to ensure downward compatibility.

EJB [95]: In 1998, Sun Microsystems released a specification for a new component framework, called Enterprise JavaBeans (EJB). EJB is an essential part of the Java 2 Platform Enterprise Edition (J2EE). Originally defined by Sun Microsystems in coorporation with other companies, today the specification is maintained by an open organisation, the Java Community Process (JCP). The EJB specification covers several topics:

- component types and component structure
- component development and component deployment
- structure of a component runtime environment (container)
- contracts between different components
- contract between components and container
- roles in the development process

In EJB, components are called enterprise beans. An enterprise bean comprises a bean class with business logic, a home interface for the bean management and a remote interface to provide clients with access to the business logic. During the deployment process additional technical classes are added

to enable the contracts between beans and their runtime environment (the EJB-container). Finally, the bean is packaged into an archive and deployed in a container.

Additionally, the EJB standard delivers a reference implementation, a development blueprint to support the development process and a test suite for the certification process of compliant J2EE implementations.

CCM [98]: Shortly after the first stable EJB release, the Object Management Group, one of the largest organizations for open standards, recognized the increasing impact of the J2EE platform and EJB and the decreasing interest in their own open platform CORBA. CORBA, the Common Object Request Broker Architecture, is a middleware platform standard for distributed systems. Functionally, it can be compared to the J2EE platform, however without any component model. In 2000 the OMG released in response to the EJB hype, a standard for a CORBA component framework, the Corba Component Model (CCM). To gain acceptance, CCM supports interoperability with the EJB specification. For that, it defines two types of component standards:

- a basic level, more or less a copy of the EJB standard.
- an extended level with a more sophisticated component notion.

An extended CCM component may export several interfaces, so called facets, and may import several interfaces, so called receptacles. The size of components is scaleable. CCM, similar to CORBA, is based on the descriptive and also standardized interface definition language (IDL). IDL in conjunction with the communication technology, Object Request Broker, are key technologies for the platform and language independence of the CCM component framework.

Critical aspects

Current technical component frameworks are a means of a component-based implementation of distributed applications. They do not support design, architectural or process issues, and are not based on any formal model. Also, the component notion is in general weak. COM components can simply be everything with an interface. There is no conceptual integration into any development process available or at least in work. The interoperability and reusability of COM components is restricted to Microsoft platforms.

EJB components on the other hand, are an open standard and the idea is reusability in different environments. However, the EJB component notion is too restrictive, compared to conceptual component notions. For examle, EJB does not allow multiple interfaces and in general, tends to represent an object more than a component.

Compared to COM and EJB, (the extended) CCM provides the most sophisticated and complete component notion. Nevertheless, until now the CCM standard is not yet accepted in industry. Companies like IBM and IONA, the main supporters of CORBA in the 90ies, commited themselves to EJB in the meantime. Until today there is no commercial implementation of the CCM available. Some open source tools provide the basic standard level. Main argument against the extended CCM is its complexity.

EJB and COM are successful in industry, but both suffer non sufficient component notions. A component identified in system-design is larger than a class (EJB) or a remote interface (COM). The mapping of design-time components to technical component frameworks is still an issue of industrial research and development.

2.3 Classification Summary

The selected approaches represent a significant assortment of currently available componentware approaches. We chose in particular well-known or pathbreaking approaches to provide an insight into the broad spectrum of ongoing work in componentware.

The classification according to outstanding characteristics covers one essential aspect, but can be extended by a domain specific classification. Each of the approaches listed above supports at least one of two different application domains:

- embedded systems
- business information systems

To summarize this section, we present a two-dimensional table (see Figure 4) with all introduced componentware approaches classified according to characterising classes and application domains.

	process centric	**architecture centric**	**descriptive**	**formal**	**technical**
embedded systems	Catalysis			FOCUS, ROOM	
business information systems	KobrA, Catalysis, UML Components, Business Component Factory	Quasar	UML, ADLs		EJB, CCM, COM

Figure 4: Classification table

If we assume that a comprehensive componentware approach should support all characteristics at least within one application domain, the table shows that there are deficits (empty fields). Although most of the mentioned approaches support more than one characteristic (e.g. ROOM defines a description technique and a process), a sufficient overall approach is not yet available. [1] A common technique is to cover missing aspects by integrating other componentware approaches. For example, most process-oriented approaches use UML as description technique and define a mapping to the EJB component framework. However the integration of different approaches leads to an obvious problem: Each approach supports its unique component notion. Defining a mapping is difficult and usually leads to a minimal and weak component notion that fits all integrated approaches. For example,

- the modelling of a comprehensive and reasonable component notion with an inadequate non-component-oriented description technique like UML,
- the mapping of a logical component of reasonable size and several defined interfaces to an Enterprise JavaBean that supports exactly one remote interface and usually has the size of a class.

These are two examples, but there are quite more. Another general problem all approaches have, is an insufficient notion of architectural concepts. Most restrict their architectural statements to general rules for component (de)composition and do not consider over all aspects of system architecture, hierarchical structuring of components and component composition. We discuss this aspect in section 5.

[1] It may even be questionable, whether there will be such a unique process covering different domains and project sizes at all.

3 Formal Methods on Components

The formal method community was always interested in issues of composition and modularity and thus in component issues. The reason is obvious. In denotational semantics or when trying to prove properties of programs modularity is a key issue. Hence a comprehensive body of knowledge developed in foundational research. This work represents a valuable foundation for componentware.

For components the research community has to understand a lot of formal and mathematical issues. This comprises questions of the description of components, in particular, of their formal specification, formal modelling as well as the question of composition. From a methodological point of view the most interesting question is certainly how to describe component interfaces in a way such that components can easily be composed and in particular that the behaviour and the properties of a composed system can be derived exclusively from the specifications of the interfaces. This notion of modularity has interested researchers in theory and methodology since a long time starting with pioneering work by Dana Scott, Tony Hoare, David Parnas, and Edsger W. Dijkstra.

While the practical approach to componentware mainly addresses software interoperability at the signature level, the way to describe in reason on the interactive behaviours of concurrent components is still an open research issue, although quite some of the theoretical foundations are well understood by now.

In principle, there are quite different approaches to the modelling and interface specification of components. One is the idea to model the component in a state-based view. In that case we describe the interface of the component in terms of the state changes in which the component is involved. A more practical technique for doing that was developed in object orientation [91] under the term *design by contract*. Most of the work is this area is more pragmatic and a careful comprehensive theoretic foundation of this approach is still missing.

Leslie Lamport developed a theoretical based techniques in an approach called *Temporal Logic of Actions* [36, 35](TLA) where he describes components in terms of their logical properties with respect to state changes. He distinguishes between the state changes initiated by the component itself and the state changes by the surrounding system environment in the shared state space. In the end he describes safe systems by their state transition relations and the liveness issues by temporal logic.

An approach, which is not based on states but very directly addresses the issue of interfaces, is FOCUS by Manfred Broy [51]. In FOCUS the behaviour of a component is exclusively described in terms of relations on streams of events. A component is purely seen as a device that receives streams of messages, events, and signals on its input ports and generates streams of messages, events, and signals in turn on its output port. The behaviour of a component is thus a relation between input and output ports with special properties. So the interface view is a purely interactive view. Bergner et al. [46] propose a formal model for componentware that is based on FOCUS.

Also a number of other theoretical work in computer science implicitly addresses the notion of a component. A lot of the theoretical work in modelling distributed concurrent systems has implicitly a concept of components such as the approaches of CSP, Communicating Sequential Processes [69] by Tony Hoare and CCS, Calculus of Communicating Systems [97] by Robin Milner. In principle, all these approaches introduce a notion of component. The same applies to all other kinds of process algebras.

Process algebras, such as CSP and CCS, are a way to study concurrent systems in terms of their forms of composition and their algebraic properties. Algebraic laws ("equational axioms") describe the properties of the composition operators. Some architecture description languages are based on process algebras (such as Wright [65] which is based on CSP).

In particular, the work on a denotational semantics of CSP by Tony Hoare explicitly addresses the notion of modularity and of a component's interface description. In his case he developed a theory to model the semantics of CSP and to specify behaviour. In his case the research shows that even for an approach, which is very intuitive in the first sight (see [70]), it may turn out that semantic issues behind are very complicated. In the case of CSP the quite intricate concept of a failure in readiness semantics has to be used to achieve modularity.

4 Description Techniques

The most commonly used graphical description technique is the Unified Modeling Language UML. The definition of one common language was a break through in industry. The diversity of languages was replaced by just one.

The UML has a lot of critics. Two arguments are:

- *The undefined modeling language:* UML is not precise and rigorous. There is still no standardized formally founded semantics. Consistency and precision of modeling elements in diagrams cannot be properly ensured. Code generation and verification have to be done propretary by tool vendors.
- *Software should not be painted:* Structured texts, screenshots and other forms of description are still required. Diagrams with boxes and arrows are not enough to communicate to non computer scientists, who are the customers usually.

4.1 Application of Description Techniques

These two arguments illustrate two trends in research and industry:

1. The description technique is (or at least should be) precise and rigorous. A formally founded semantics exactly defines what diagrams and texts *mean*. Descriptions are complete (syntax and behaviour) at their level of abstraction. A description can therefore be unambiguously refined down to code.

   The Model Driven Architecture (see [63]) stresses the automated refinement of UML descriptions down to code. Although there have been several efforts in research like the *precise UML group*[53], there is still no formally founded *standardized* semantics of UML. Code generation and the simulation of UML descriptions are still proprietary. Tool vendors decide how descriptions are interpreted and executed.

   Autofocus [105] is a formally founded graphical description technique and tool, that does not use the UML. Autofocus is based on FOCUS (see above).

   In both examples, Autofocus and MDA, the description is viewed as an executable *model* of the intended system.

2. Code generation is not directly intended. The communication of descriptions between developers and to customers is stressed. Understandablility is more important than rigour and precision. Graphical descriptions may therefore be ambiguous and incomplete. Not the whole system is modeled, only important parts are documented. Structured texts and screen shots should also be used to specify a software system, they might even be more important than (UML) diagrams [114]. The system is implemented manually, verification is done by reviews and tests.

   The description is viewed as a *documentation* of the software under development.

4.2 Describing Components in UML

The UML has its roots in object-oriented development, most of the diagram types depict object-oriented concepts, such as class diagrams and collaboration diagrams. The current version of UML (UML 1.5) provides component diagrams, but these diagrams are focussed on deployment rather than specification time concepts. UML's next version (UML 2.0) contains some better suited diagram types.

Therefore there are different ways to specify a components with the current (OO -)UML version:

- Use UML as it is, and define a concept in which case, which diagram types should be used. This is a usage concept (see section 6.5.2). UML provides package and component diagrams that

can be used. For example a package might represent a component: The package contains all diagrams that describe a component (like a directory in a file-system). Interfaces are specified with the UML-lolipop notation and are mapped to packages. A package-diagram shows the relationships between the components.

- Define a UML (component-) dialect, with the standard extension mechanisms of the UML, such as stereotypes. Cheesman and Daniels [56] and Catalysis [59], extend the UML with component-stereotypes and interfaces-stereotypes. Medvidovic et al [89] formulate ADL-concepts using the UML extension mechanisms. Extended class diagrams are commonly used to model components and their relationships.
- UML has no diagram types for graphical user interfaces; stick figures and ovals are not enough to describe use-cases, textual descriptions are needed. These are two examples illustrate, why the UML alone is not enough to describe components. Most of the approaches add structured texts, screenshots and proprietary diagram-types to describe components [114].
- UML is basically not appropriate. A formally founded description technique better meets the requirements of componentware.

An other open issue is, what exactly constitutes a component. Is a component just a box with some lolipops or is it the set of all artefacts describing it (that also includes screenshots and textual information)? If it is a set of artefacts, how can consistence between these diagrams, graphics and texts be guaranteed?

5 Software Architecture

Just a few componentware approaches directly define software architecture concepts, that give concrete guidance to designers and architects. Most of them claim to be architecture centric. Approaches to software architecture and componentware are overlapping. Both consider the structure, the composition and the behaviour of software systems.

We think, that a software architecture approach is an integral part of an integrated componentware methodology (see section 2).

One commonly used definition of the term *software architecture* is given by Bass, Clements and Kazman [43]:

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

This notion of components is rather general and weak. A component can be everything from a database engine to a programming language class. Note, that the autors replaced *component* by *element* in the second edition of [43].

To reason about software architecture, further more expressive notions are needed. The definition mentioned above is a starting point. An approach to software architecture, which is a part of a componentware methodology should define these expressive component notions. Furthermore,

- a software architecture approach should be scoped. It defines a basic component notion and specialized domain specific notions. Software architecture defines mechanisms to combine components to a complete architecture in that domain. Domain specific component concepts are essential. For example, a software running on an embedded device in a car and a batch-job in a business information system are obviously different things - but both are called component today.

- an architecture approach provides different component notions over the phases of development and over the level of granularity to give the language of engineering more expressiveness. For example, an Enterprise JavaBean and a business subsystem are obviously different things, but both are called component today.
- an architecture approach provides concepts for the composition of components, the contract defintion between components and a technical infrastructure.

In the following different component notions are discussed. Scale and granularity, technical and business issues and composition of components are considered. Section 6 surveys components over the phases of development. Section 8 gives some impressions of domain specific componentware.

5.1 Scale and Granularity

Distinct Level of Granularity

Large scale object-oriented software systems consist of several hundreds (thousands) of classes. Components should be a means of structuring large systems. In systems of that size obviously more than one level of granularity is needed. One basic question in many appoaches is: Are there a few, discrete levels of refinement or is refinement to arbitrary depth allowed?

Herzum and Sims [68] propose three distinct level of refinement in business information systems: Distributed Components, Business Components and System-Level Components (see Figure 5).

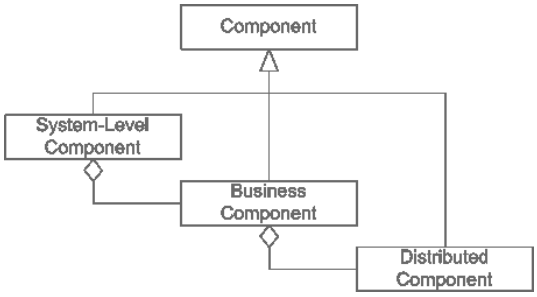

Figure 5: Discrete levels of refinement [68]

An approach with discrete levels of refinement allows a detailed specification of components' properties. Components at a certain level can be defined early in the development process.

- Distributed component (e.g. a EJB-session bean plus a few extra classes): A distributed component is a small component, that consists of some programming language classes and has a network interface. Enterprise JavaBeans, COM or CORBA are used to implement such a distributed component. Note, that a distributed component not necessarily consists of just one Enterprise Bean or IDL implementing class. Several packaged classes or beans may constitute a distributed component.
- Business component (e.g. a customer management component). A business component consists of distributed components. It represents an autonomous business concept as physical unit that can be independently deployed. Typical business components have a graphical user interface (and its specification), an application layer and database access (plus the database schema). Business components are defined during requirements gathering.
- System level component. A system level component consists of business components. It represents a cohesive set of functionality required by a business need.

Named Level of Granularity

Stüutzle [113] discusses three levels of component granularity. He allows components at one level to contain components from the same level of granularity:

Basic component : A basic component has arbitrary dependencies to other basic components. It is not executable on its own. For example, a programming language class is a basic component.

Assembly : An assembly consists of basic components or other assemblies. It has cleary defined dependencies from/to other assemblies. It is not executable. For example, the customer management component in a business information system is an assembly.

System : A system consists of assemblies. It has minimal dependencies to other components and is executable.

Refinement to Arbitrary Depth

KobrA [41] and Catalysis [59] are general purpose methodologies. They use recursive refinement to arbitrary depth. Concerning the system size and other factors, the business analyst or the software architect decide which level of refinement should be used.

5.2 Technical and Business Components

Siedersleben applies in his approach [111] the principle *separation of concerns*. He separates technical software from business software and defines a classification scheme for software in business information systems. Four categories of software according to thier dependencies and their implemented concepts are defined:

0-software depends on nothing. Examples are class libraries such as STL in C++ or java.util. 0-software is ideally reusable, but it is of no use its own.

A-software implements a business concept and depends on that concept only. Examples are classes such as customer, invoice or contract.

T-software implements a technical concept and uses a lower level technical infrastructure. For example, a database access layer that uses JDBC is T-software.

AT-software implements technical and business concepts at the same time, and depends therefore on both. Hence AT-software has to be modified all times, when business or technical requirements change.

We can apply software-categories to componentware. **0-components, T-components, A-components** and **AT-components** can be distinguished. This sets up a quality criterion and design heuristics for software architectures: AT-components should be avoided.

Siedersleben further formulates the principle: *separation of technical and business concepts* in the Quasar (QUAality Software ARchitecture) [110] approach. This scheme also applies to software architecture. Two different views on a system architecture can be definend:

The T-Architecture defines everything related to the technical platform (such as .NET or J2EE) and the cross-cutting technical principles, such as security and persistency. The A-Architecture comprises business components. Only business concerns are incorporated. Figure 6 depicts A- and T-architecture.

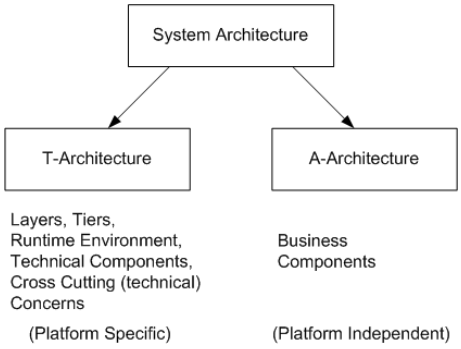

Figure 6: Technical Architecture and Business Application Architecture

5.3 Component Composition

Dependent and self-contained composition

Dependencies are a critical factor in software development. The more dependencies are introduced into the system, the harder it is to change and to maintain. All componentware approaches claim to reduce maintenance cost and to lower dependencies.

There are two different approaches in the composition of a system of components:

- A component knows the business concepts or/and the interfaces of the components it uses. The component imports these concepts as references in its documentation or as direct includes of the specified interfaces.
- Compontents are self-contained concepts and units of software. Appart from general stable business concepts and technical infrastructure, a component defines all concepts it uses on its own. All interfaces the component provides and requires are a part of the component. The component defines the interfaces it expects from other components to provide, independent of the interfaces these components actually provide. The self-contained components are connected through extra channels (i.e. adapters), that provide a mapping between the interfaces and may define further contracts between the components.

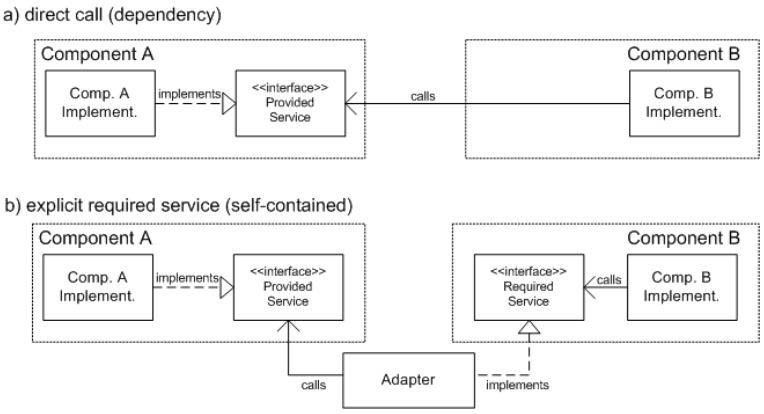

Figure 7: Direct calls and explicitely required services

The two types of component development and composition differ in their introduced dependencies. The first and commonly used concept introduces dependencies between the components. If business requirements change, changes in interfaces and therefore in the providing and using components are

likely. The second concept (e.g. provided by Rausch [102]) reduces dependencies, if interfaces change, only the adapters (mappings) have to be changed.

5.4 Contracts

Contracts belonging to Classes

Meyer [91, 92] introduced the notion of contracts between classes. Contracts are defined with invariants, pre- and post-conditions between components (classes).

Rausch [102] defines two kinds of interfaces of a component: provided and needed interfaces. The needed interfaces are syntactically specified. The behaviour is specified with invariants, pre- and post-conditions. Contracts between components are modeled as *signed contract*.

Contracts as separate Constructs

Andrade and Fiadeiro [38] define contracts between components explicitely apart from the components. They separate between service providers (the components) and the mechanisms (contracts/connectors) through which the behaviour of these components is coordinated to fulfil business requirements. Andrade and Fiadeiro use a logical interception mechanism: The applier directly calls the provider, the contract mechanism intercepts the call.

5.5 Infrastructures

Only in a few domains components directly run on a hardware. Usually at least a thin operating system controls the ressources such as memory, processes and IO-channels. Modern technical infrastructures such as EJB-Containers or the .NET-Framework provide a large set of predefined technical services such as transaction management, persistency or security.

The contracts between the components and the infrastructures (such as an operating system) are often not explicitely defined. For example: A component should define memory- and CPU-consumption in an enviroment, where these ressources are limited.

Contracts to an EJB-Container are defined declaratively, there is no direct method call that could be intercepted. A deploymend descriptor defines aspects such as persistency, transaction handling and security.

6 Development Processes

The different representations of components in a software development process are discussed in this section. The relationship of these representations is an open issue.

A detailed discussion of all tasks in component development would fill a complete book (perhaps part two of Szyperski's book [116]), therefore we will present only one task in component development in more detail, the validation and verification of components.

6.1 Component Representations

Components can be introduced into every phase of a software development process (see Figure 8). Their representation and meaning depend on the phase and the level of detail we concern. In the analysis phase a component might be represented as a text block and additional business process diagrams whereas the deployment-time representation might be a library file such as a JAR or a DLL file.

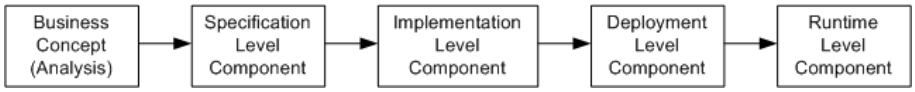

Figure 8: Components and Phases in the Software Life-Cycle

The relationship between the components over phases of development is seen differently by contemporary approaches. Some approaches define an integrated component notion that covers all phases of development, others focus on few phases of the development process or define different component notions for every phase.

In the following the component notions and representations over the phases of software development processes are described. We consider the level of coarse grained components that represent independent business concepts. We consider six phases:

1. Analysis time where requirements are gathered
2. Modeling time where a model of the system is created (i.e. the specification is done)
3. Implementation time where the model is implemented on a specific platform
4. Verification time where system-tests and other verification is done
5. Deployment time where the system is shiped to the customer
6. Runtime where the system runs in the customer's enviroment

6.1.1 Analysis Time Components

Herzum and Sims [68], Levi and Arsanjani [81], start with the *decomposition of the business domain* into business components, *before* requirements are gathered. Both approaches use business processes and other information from the problem domain to identify components. Both assume that coarse grained business concepts and processes are already defined.

For example, Levi and Arsanjani *divide the domain into functional areas based on department boundaries, business process boundaries and value chains.* They call this first phase *Domain Decomposition* (functional areas are another word for components, that is more accepted by business analysts). In a second step, the subsystem analysis, the identified functional areas are broken down further into their constituent business processes or functional areas. Further steps define the functions and services of the identified components through Goal-Service-Graphs [81].

6.1.2 Modeling Time Components

In the specification/design phase a model of the system is created. The model is described by a textual or graphical notation such as UML. Most of the contemporary approaches are based on UML. For example, UML is used by [56, 59, 41].

Components are represented in UML, structured texts or other description techniques at modelling time (see section 4 for a discussion on the representation of components in UML).

An open issue at modeling time is: Is there just one model of a system (component) that is refined down to code or are there some distinct models, that describe different abstractions? For example, the Model-Driven-Architecture (e.g. [63]) defines two different models of a system: a platform independent model (PIM) that contains only the business part and the platform specific model (PSM), which is an amended mapping of the business part onto a technical platform. Whereas most of the contemporary approaches refine their models down a certain level of abstraction or even down to code.

If the Model-Driven-Architecture becomes broadly accepted the notion of specification time components must be refined to PIM and PSM components.

6.1.3 Implementation Time Components

All industry approved programming languages are not component-based, they are either object oriented or procedural. Therefore components have to be expressed at the implementation level in terms of these programming languages, e.g. in ADA, COBOL, VisualBasic and Java.

A component might be represented by a single programming language class. A large program comprises easily hundreds of classes, therfore it is obvious that a programming language class is by far to small to represent a coarse grained business component.

Programming languages and their programming environments provide mechanisms to bundle several classes, functions and other constructs into a logical unit. There are three basic mechanisms:

- physical organization of code in directories, i.e. a directory represents a component. For example java-packages are a means of organizing code in directories.
- logical organisation of code in namespaces, i.e. a namespace represents a component. Namespaces are directly implemented in C# and C++. Java provides packages that represent both namespace and physical organization.
- logical organisation of code in a tool based representation, such as a project in the IDE, or a target in a make-file.

The interfaces of components are represented by means of the programming language, such as interfaces in java and abstract base classes in C++.

In the implementation-phase the specification is implemented on a specific technical platform, such as J2EE, .NET or in a CORBA-Environment. These platforms provide technical component notions. These notions are defined by SUN, Microsoft and others. Usually one notion is defined for the graphical user interface and one for the business functions. Some notions are listed below:

- a remotely accessible interface (or service). The interface could be defined in an interface definition language, such as CORBA-IDL or COM-IDL. Similar representations could be a WebService definition in XML or a programming language interface, such as an Java RMI interface.
- a technical unit, that is managed by an infrastructure. Enterprise JavaBeans provide technical components, such as EntityBeans and SessionBeans. These units are managed by the EJB-container. These units are fine grained, e.g. an EntityBean instance roughly represents a single row in a database-table.
- a unit of composition with a simple common access-protocol. For example, JavaBeans are building blocks in graphical user interfaces.

Currently there is no common understanding, which programming-language and technical constructs actually represent a component. SUNs (Enterprise) JavaBeans and programming language classes are roughly on the same level of granularity. Therefore they should not be used to represent a business component directly. A business component may rather be a set of these constructs [44].

6.1.4 Verification-Time Components

In the verification phase quality criteria of a software are measured. Test specifications are executed manually or implemented as unit-tests. Test protocols document executed tests. Business components are units of verification, therfore test specifications, unit-tests and protocols can be structured according to the given component structure. Tests verify critera of an executable software.

Quality critera, such as maintainability or understandability of the documentation cannot be measured with tests. These critera are measured by reviews of the system structure, the code and the documentation.

Review protocols, test protocols and other measurement results are a part of the component representation at verification time.

6.1.5 Deployment-Time Components

A component is a unit that can be independently deployed. This phrase is a part of the majority of component-definitions.

Libraries are units of independent deploymemt. A component may be represented by a library or a set of libraries deployed together. For example, a component can be represented by DLLs (Dynamic Link Library), a .NET assembly or a JAR-package (java).

The interfaces in the libraries are defined in a binary form, compiled for a hardware or in a machine independent intermediate language. In some cases contracts with infrastructures (like an EJB-Container) are described in extra text-files (deployment descriptors). For example, these text-files define the object/relational mapping, transaction and security attributes.

6.1.6 Runtime-Time Components

The notion of runtime-components is an old argument between the different componentware approaches. Some discussed questions are:

- Has a component an externally observable state at runtime?
- Is there one or an arbitrary number of components at runtime? Or to put it in an other way: Is there a difference between a component type and component instances?
- Is a component an autonomous unit of execution? Does it run by executing an own process (thread)?

The first two quesitions are closely related since if a component has no externally observable state, instances cannot have an externally observable identity, and therfore they cannot be logically distinguished.

These questions have consequences for the early phases of development and on the systems architecture. A development approach should answer these questions before any analysis or design is done.

6.2 Relationships of the Phases

As we have shown above there are several possible representations of business components through the phases of development. How do these representations relate?

The opinions differ on that matter:

- A (business-) component is an autonomous concept with different representations over the various phases of development. An analysis-time component and the deployed libraries are basically the same.
- In every phase of development different concepts are concerned, for instance from the problem and the solution domain. Therefore a business concept in the analysis-phase may be mapped to several deployment units or a set of concepts is mapped to one deployment unit. Or only phase or a few aspects are considered, for example, only component assembly and the deployment.

Integrated Concept

Herzum and Sims [68] provide an integrated business-component notion. The components identified before the requirements are gathered are also units of deployment to the customer. This idea has several implications:

- A component is more than a jar-file or a code block. It consists of all artefacts that are created during the development process: documents that describe the business concept, test specifications, user-interface prototypes and so on. All artefacts are packaged in the component, and deployed to the customer.
- Because a component constitutes a business concept, it usually has a (graphical) user interface and database access in a business information system.
- A component is coarse grained. Therfore it can be used as unit of project planning, risk analysis and configuration management.
- Technical components such as EJBs are mere parts of one component implementation and a connection to the technical infrastructure such as an application server.

Cheesman and Daniels [56] do not define component at all, instead they define the forms a component can take: A component specification, component implementation, installed component and component (runtime-) object. In addition, a component interface is a part of a component specification.

Focus on Assembly and Deployment

Szyperski's component definition, has a strong focus on deployment and assembly. He states that *A software component is what is actually deployed* [116]. Documentation aspects or any business aspects, such as a human understandable usage description or business autonomy are not mentioned. With this definition also component markets and component reuse can be concerned, whereas the approaches mentioned above rather focus on the development of new software.

Note that marketplaces for components are actually available. Only technical components (see below) such as GUI-controls or logging facilities are sold there, no substantial reuse of business components (e.g. some banking components) takes place. The idea of just assembling pre-defined components is currently not carried out (and might never be).

6.3 Verification & Validation for Componentware

According to the IEEE Standard Glossary of Software Engineering Terminology (IEEE-Std-610.12-1990):

1. *Error* is the difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition.
2. *Fault* is a defect in a hardware device or component (short cut or broken wire), or an incorrect step, process, or data definition in a computer program.[2]
3. *Failure* is the inability of a system or component to perform its required functions within the specified performance[3] requirements.

Errors, faults and failures are *defects*. A defect is defined as a product anomaly, according to the IEEE standard 100-1992. Validation and verification primarily aim at discovering errors in a system.

Verification and validation denote two different activities. While validation addresses the relationship between a model and the intention ("are we building the correct thing?"), verification is concerned with the question of a (software) system accurately implementing the specified model ("are we building the thing correctly?"). These activities form part of the development process because this is imperfect. For long time, the common industrial validation and verification practice has been testing,

[2] This definition is mainly used in fault tolerance discipline. In common usage, the terms error and bug are used to express this meaning.

[3] *Performance* is the degree to which a system or component accomplishes its designated functions within given constraints, such as speed, accuracy, or memory usage.

simultaneously neglected by the formal methods community because of an erroneous perception of it as innecessary, since formally developed (i.e., formally validated/verified) systems be "correct by construction". Through formal methods, namely, the correctness of program elements is in principle provable using mathematical techniques. But results from this field unfortunately have had only limited impact on the practice of the industry [94]. Therefore, and considering the complexity of today's systems, it is improbable that the development process become reliable enough to turn testing redundant. Testing, therefore, as well as verification and validation, are inseparable from the development process, and many methods aim at converting them both as effective and as efficient as possible.

Componentware provides the foundations for building reusable components and assembling them into systems. A component is a reusable piece of software with clearly defined interfaces [39]. These may include precise definitions of (mutual) obligations and in this case are also called contracts. Applying reuse techniques in the software construction activities without also applying the same techniques in verification and validation is possible but much less effective. Coordinating reuse across both types of activities is much more effective and efficient [86].

The validation and verification of the behaviour of each component can be undertaken using conventional techniques. A distinguished task is an additional contract verification/validation in order to ensure that interacting components abide by their contracts; see [42]. These activities are also called "validation and verification from the component producer resp. consumer perspective"; see [119]. Once the necessary components as puzzle pieces are gathered for the assembling of the target system,[4] validation and verification of their integration has to be carried out in order to assess the correctness and dependability of the smaller units in a new environment (in a similar way as deployment testing); see [47]. Here the principle of encapsulation should not be violated. The difficulty of this task is increased when OTS (off the shelf) components are used.

It is desirable that component designs are available for composition. In this case, design verification and validation may also be compositional; see [62]. They might, moreover, be tool supported; see [57]. In this way, errors could be exposed earlier in the development process, which is advantageous not only because of cost reasons.

Modern efforts towards reliable component-based development take advantage of the experiencies with free, source-code-available products. Some of these tools are ambitious, of astonishing quality, and have been selected over commercial products by for-profit as well as administrative organizations. [94] adjudges this success to the power of social processes: those tools have been developed by volunteers and examined by devotees all over the world, who tested them, report defects and suggest improvements, in an effort of unprecedented magnitude. There it is proposed, in a similar vein as [72], a collaborative, federative effort including, among other things, the development and adaptation of components, and the elaboration of verification and testing technologies and of metrics.

Summarising, the challenge resides in the development of quality assurance methods that do not only check components *inside* but moreover work *on top* of them, not only after implementation but also at the design phase, and this in a modular, language-independent, compositional fashion. A starting point for a mathematically precise formalism can be given by institutions; see e.g. [40].

7 Toolsupport for Componentware

Componentware related tools provide an essential support for component engineering. They can range from text based generators to graphical diagram painting tools. The vast variety of tools as depicted in Figure 9 help to accomplish a wide area of tasks within the component engineering processes. They support executing the development process itself, modelling components and component architectures, specifying with various description techniques, implementing, deploying and monitoring components. Furthermore there are also tools which support the integration of different component-based applications and orchestrate the various components that participate in possibly distributed

[4] Gaps between components might either be filled by other components or give rise to the requirements for the development of a new component. This component must likewise undergo the quality assurance procedure as any previously developed component does.

business processes.

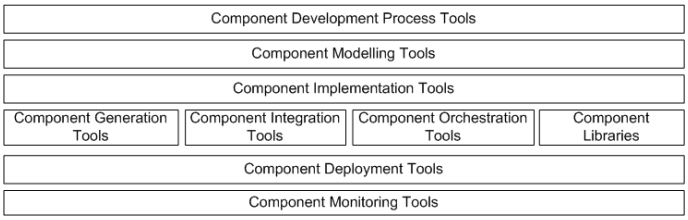

Figure 9: Componentware Related Tools

We now describe in more detail the different categories of tools:

Component Development Process Support Tools

These tools should support the process of componentware engineering. While one can define componentware specific processes [48] there are almost no specific tools to support componentware specific processes. It seems often sufficient to adapt existing process management tools to the componentware related process at hand. Techniques for creating an adaptable engineering process for componentware are so called process patterns [45]. Processes are composed out of instantiations of single patterns where each pattern describes the context wherein it can be applied, the problem it solves with prerequisites, constraints, and implications. Finally it presents a solution as a set of activities with their dependencies. A tool to support this is LISA [19].

Component Modeling Tools

Components can be modeled using case tools based on standard UML [118] like Rational Rose [23], Together [28], Telelogic TAU [26] and ArgoUML [4] just to name a few. Another OMG standard for developing component models based on UML is EDoc [61].

Component architectures are sometimes also modeled using an architecture description language (ADL), for an overview see [90]. For some of the various ADLs there exists tool support, for example ACME Studio [1].

Because ADLs are neither wide spread nor standardized probably the UML will develop itself more towards an architecture modeling language and so the tools will incorporate such modelling features in future.

Component Generation Tools

On the one side many UML modeling tools also evolve into architecture modeling tools. Sometimes they allow model execution but even more often they support code generation, for example for EJB [24]. Often code generation is supported by so called wizards. Unfortunately this is often too inflexible and cannot be automated. The so called Model Driven Architecture (MDA) approach became popular recently [96]. Using MDA, independent models are transformed with platform independend profiles to concrete platform specific models or code. As the transformation descriptions are not yet standardized many of the UML tools mentionend above implement proprietary profile and generation capabilities.

Also gaining in popularity is generation based on metadata information that is added to the objects of a partial implementation. This is done by adding descriptive metadata to the source code itself or in form of external XML files. A tool then generates missing parts and platform specific component descriptions based on this meta data together with transformation profiles. The advantage in this case is that there is just one central textual source code which is often easier to maintain than graphical specified architectural models. Examples using the meta data attributes approach are Micrsoft C#

attributes [8], XDoclet [33] and EJBGen [5] for Java. For Java meta data attributes are current standardized by the Java Community Process [16].

Other component generation approaches are based on models in a XML format called XMI [34] and XML transformations, they allow to combine the MDA with the meta data based approach mentioned above by generating from UML models in form of XMI code with meta data attributes, which is then further handled by the meta data based generators. An example for this is AndroMDA [2]

Component Implementation Tools

Besides generation also a so called Intergrated Development Environment (IDE) can provide valuable support during component implementation. Examples for popular IDEs are Eclipse [10], IDEA [13], Borland Enterprise Studio [11], Visual Studio .Net. and Bowstreet [6]. Web service specific IDEs are provided by CapeClear and Systinet [25].

Component Deployment Tools

Besides implementation also deployment of components e.g. in distributed systems is an issue. Various frameworks exist to simplify installation and deployment in different runtime environments. Java Web Start [15] is such an example which allows the installation of Java applications over the internet. Also most java application servers support dynamic deployment of standardized component packages for web or business components as .war and .jar files.

Component Monitoring Tools

After component deployment monitoring tools can be used to manage the deployed component. For example most java application servers implement today the J2EE Management Standard [18] which allows to manage deployed components remotely using so called Java Management Extensions (JMX) [14]. Management tools like AdventNet ManageEngine JMX Studio then can be used to track the performance of single components.

Component Integration Tools

Besides supporting design, development and operation of components a whole set of tools also exists for component integration. These so called Enterprise Application Integration (EAI) tools, for example from Tibco [27], WebMethods [30] or PolarLake [22] simplify the integration of components of existing applications. Basically they make business data accessible from various sources, and integrate them in new business processes presented via different types of user interfaces. A standardized infrastructure to integrate systems on the business process level for java based EAI systems which is called Java Business Integration (JBI) is currently developed [17].

Component Orchestration Tools

For business processes which span multiple distributed systems like in B2B systems with automated interfaces, there are various standards and related tools available to standardize the orchestration of the participating components. Often these components present their services online as so called web services which exchange messages with each other. Examples for such standards on a busines process level are ebXML [55], RosettaNet [104], OAGIS [21] and BPML [7]. BizTalk [117] is a tool in this area. On a finer grained implementation level there are lots of web service orchestration "standards" like WS-BPEL/BPEL4WS [29],WSCL [32] and WSCI [31]. Collaxa [9] is an example for a WS-BPEL related tool.

Component Online Libraries

Components become more and more available as web services and subscribed and integrated over the internet. Such services can be found at different web service networks like Grand Central Communications [12], XMethods or Salcentral. Beside this online form, also component libraries are still available where component are provided as binaries or source code.

Now that we have provided a short overview about the different tool categories and named a few examples it becomes obvious that there are far too many tools to mention here. In fact it becomes clear that tools for designing, implementing, integrating and managing componentware are already widespread but are still enhanced especially towards supporting online integration and coordination of distributed components.

8 Domain Specific Componentware

8.1 Scoping Componentware

Components are viewed as general building blocks of an architecture or a software-design. Everything from a database-engine to a programming language class may be called a component.

Business information systems and embedded systems differ in their componentware requirements. Several approaches focus on one of these domains. ROOM [107] is specialized in real-time systems, the Business Component Approach [68] and Quasar [110] focus on business information systems.

Embedded systems are usually hard real-time systems with a number of tasks running in parallel. The modeling of time and concurrency is a critical task in building embedded software. On the other hand the timing constraints in business information systems are softer and concurrency is managed by the transactional infrastructure. Whereas modeling a graphical user interface is the crucial task in specifying business information systems, because the user interface design is a means of communication with the users.

8.2 Componentware for Embedded Systems

This section will be added by Alexander Pretschner

8.3 Componentware for Critical Systems

The high quality development of critical systems (be it dependable, security-critical, real-time, or performance-critical systems) is difficult. Many critical systems are developed, fielded, and used that do not satisfy their criticality requirements, sometimes with spectacular failures.

Systems whose correct functioning human life and substantial commercial assets depend on need to be developed very carefully. Systems that have to operate under the possibility of system failure or external attack need to be scrutinized to exclude possible weaknesses.

Part of the difficulty of critical systems development is that correctness is often in conflict with cost. Where thorough methods of system design pose high cost through personnel training and use, they are all too often avoided.

On the other hand, there is an increasing interest in the use of componentware due to a possibility for savings from potential reuse.

This raises the question whether the componentware approach can be used fruitfully in the critical systems area.

Beyond the general arguments about savings by reuse, the componentware approach offers an interesting opportunity for high-quality critical systems development that is feasible in an industrial context.

- If reusable critical components can be identified, they can be developed at a high standard of quality, and possibly even certified by official authorities.
- If a suitable methodology for component-based construction of critical system exists, these critical components can be employed safely and securely within the system context.

Under these two provisos, the componentware approach has the potential not only to reduce costs, but at the same time to increase the quality of critical systems software. This observation prompts some challenges one has to overcome to exploit this opportunity, which include the following:

- Adaptation of an appropriate notion of component to critical system application domains.
- Correct use of critical components in the system context and the application domain.
- Conflict between flexibility and level of criticality guarantees when defining components.
- Improving tool-support for component-based development of critical systems.

Surprisingly few attempts have thus far been started to address these challenges, explicitly dealing with component-based development of critical systems: [85] points out some problems with using commercial-off-the-shelf (COTS) software for security-critical systems. On the tool-side, the CASE tool AutoFocus [106] based on the component-based design methodology in [50] has been used for critical systems development for example in [67]. Component-based approaches to developing critical systems with the Unified Modeling Language can be found for example in [75, 77, 76]. [66] considers components in the context of specifying safety-critical embedded systems with Statecharts and Z. The integration of mixed-criticality software components is treated in [60]. [84] considers testing component-based safety critical software. [73] treats the verification of requirements specifications using composition and invariants. Several of the contributions in [78] also consider components. To be able to use an component-based approach to developing critical systems, one needs knowledge on to what extent criticality requirements are preserved by composing system parts. For example, [88, 74] examine composability of security requirements.

9 On a Discipline of Component Engineering

Software engineering is an engineering science, very much like electrical, chemical and mechanical engineering. A considerable portion of the daily work in these discipline is not based on scientific results but on experience, and thus fuzzy and imprecise. A software engineer needs engineering methods that include expertice and provide concrete guidelines for the daytime work. Some of the following questions should be answered by a component engineering methodology:

- What are the characteristics of a "good" component?
- How (and when) is a problem (system) decomposed into components?
- How is a required functionality mapped to components?
- What are the criteria to decompose a system into components?
- How does a good component interface look like?
- How can a system be composed of existing components, that are reused?

We do not have any software-engineering algorithm or methodology that produces adequate components or architectures automatically. It is probably not possible to create proper components and a good architecture only by using some tools and a more or less adaptable development process. Human intuition and experience as well as scientific-methodology are required:

- Design heuristics and rules of thumb are a less formal way to support designers. Parnas formulated a strong heuristic 30 years ago: the principle of information hiding [100].
- Quality criteria and component metrics are the basis for the quality assessment of designs and are a basis for reusable (trusted) components, that are traded in component market places [93].
- Application concepts, design rules and cookbooks provide conrete help. They form the craftsmanship's part of software engineering. For example, a description technique defines a common language for engineers and their customers, but it is useless without the knowledge how it is applied to a given problem. Usage concepts as provided by some methodologies [59, 56] show how a description technique is properly applied.
- Patterns and templates provide reusable designs. In other engineering disciplines nearly no design is built from scratch. Approved designs are copied and adjusted to a given problem. The design-patterns movement [64, 54] with catalogs of reusable micro-architectures are a good foundation. Patterns for complete software architectures are still an area of industrial research [110].
- Standards provide measurable norms for development processes, quality improvement and designs. In software-engineering lots of national and international standards such as ANSI, DIN and ISO are available. Independent organizations also define standards such as IEEE and the OMG.

9.1 Design Heuristics and Rules of Thumb

How is a system decomposed into components? How can these components be made resistant even to unanticipated changes?

Many approaches give just vague ideas, how to do the decomposition and how to define good component's interfaces. Although experience and some intuition are required to do good designs, basic guidelines and can be formulated as heuristics or as rules of thumb.

Example: Decomposition Heuristics

Parnas provided criteria for the decomposition of a system into modules. We present the *information hiding* as an example for a well known design heuristic:

Implement parts, that independently change in separate modules [101] and hide a difficult design decision or a part likely to change in a separate module [100]. For every module (or component) a simple question must be answererd: *What is the secret of that module/component?*

Collecting Heuristics

Heuristics are abstractions of experience formulated in natural language. Heuristics are not generally appliable laws or rules, they may be wrong in certain settings. Therefore, they must be applied with judgement. Rechtin describes heuristics as a basis for practitioners:

> All professions and their practitioners have their own kits of tools, physical and heuristic, selected from their own and others' experiences to match their needs and talents.

Rechtin summed up a collection of heuristics in the appendix of [103], these heuristics are based on a collection of student heuristics in systems architecting and from subsequent studies. For example:

> Do not slice through regions where high rates of information exchange are required.

Rules of Thumb

Rules of thumb are abstracted experiences, like heuristics. They can formulate evaluation criteria for component oriented designs. Herzum and Sims [68] provide some rules of thumb with which components can be evaluated. For example:

> A distributed component has on average 10 business language classes, but a few may have up to 200. If there are fewer than 5 or more than, say, 150, there may be problems.

9.2 Quality Criteria and Metrics

Meyer formulates the *Grand Challenge of Trusted Components* [93]. *A trusted component is reusable and has specified and guaranteed property qualities.* Meyer proposes a framework for a *Component Quality Model*, that takes properties of the design, behaviour and documentation into accout. It devides properties of interest into five categories: Acceptance, Behaviour, Constraints, Design and Extension (ABCDE) [93].

For example, in the behaviour category, properties such as examples, usage documentation, preconditioned, full postconditions and observable invariants are listed. These properties of a component are assessed.

A detailed common quality model for components is desireable, because it enables a structured assessment of components. Components, that are traded, might be certified against such a quality model. Other engineering disciplines have defined such quality standards decades ago.

9.3 Application Concepts and Cookbooks

While heuristics capture the *art of engineering*, application concepts, guidelines, and cookbooks cover the craftsmanship's part. Application concepts give a detailed description, how a method is applied to a concrete problem.

For example: The UML is a standardized, general purpose modeling language. But there is no common understanding how it should be applied. An organization that uses UML and the common modeling tools should define a usage concept for the UML and the tools. If such concepts do not exist, designers will not be able to work together, because everyone has her own understandig, how UML and the tools should be used. Long learning curves and errorneous designs are a second consequence of absent usage concepts. Cheesman and Daniels provide such an application concept for the UML [56].

Example: Architectural Views

Architectural views are can be seen as a application concept for modeling languages. A system is described from different perspectives. Approaches to software architecture usually define a set of views on the architecture [71, 112, 79, 68, 58].

Each view defines a set of aspects of an architecture and diagram types or textual specifications that describe these aspects. Views are usually focussed on the concerns of certain stakeholders.

Soni, Nord and Hofmeister describe architectures with views (structures) of systems [112]. They define four views: conceptual, module, execution and code. For example, *the conecptual structure describes the system in terms of its major design elements and the relationships among them. This is a very high-level and domain-specific structure of the system, using elements and relationships specific to the domain.*

Cookbooks and Tutorials

Cookbooks, tutorials and guidelines provide simple and concrete "recepies" for common problems. Tutorials are focused on beginners, they ease learning a new methodology, tool or programming language. Tutorials provide simple examples, that can be used as starting points for designs.

Cookbooks and focus more experienced software-engineers. The J2EE patterns [37] can be seen as cookbook for J2EE design. Common technical and design problems are discussed and concete solutions are presented.

9.4 Patterns and Templates

Confronted with a design problem, an engineer searches for an existing solution to a similar problem. The solution is copied and applied to the actual problem. The description of abstract solutions to common design-problems is the basic idea of design-patterns.

Patterns

In the 90ies the patterns movement started with the book by Erich Gamma et al [64] on design patterns. The basic idea is to provide a catalogue with simple, approved, reusable micro-designs. Only a few classes are involved in design-patterns, most of the patterns stress decoupling. Buschmann et al. [54] formulate a few patterns with a higher abstraction, such as layered architecture and pipes and filter. Buschmann calls them architecture patterns. Both catalogues are helpful to design small and medium scale architectures, and internal designs of larger components respectively.

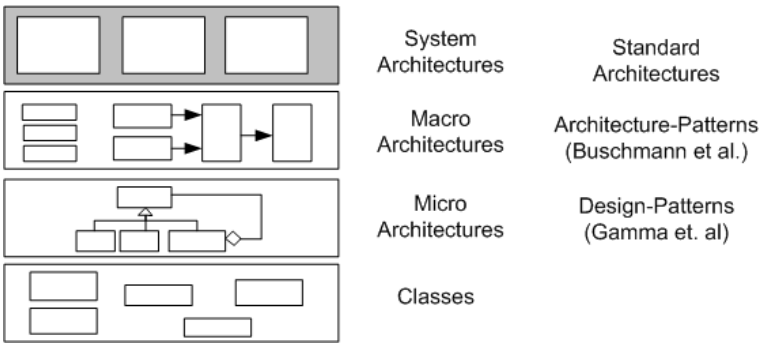

Figure 10: Design- and Architecture Patterns

Component-oriented designs at a higher level of abstraction need more specific help, taking the problem domain into consideration. Figure 10 shows the relationships of the pattern catalogues.

Reference Architectures

For every problem domain suitable standard components can be defined. The QUASAR architecture [110] names a set of standard components in business information systems, such as dialog, screen, dialog memory, business entity, business facade, etc. Figure 11 illustrates, how QUASAR defines an application component. It consists of use-case implementations and business entities and their managers, just the interfaces of the use-cases are exported to other components. The application component has its own island of data in the database.

Other efforts in this area are the architectural blueprints by SUN and IBM. J2EE patterns [37] present a list of standard components, with a focus on J2EE technology.

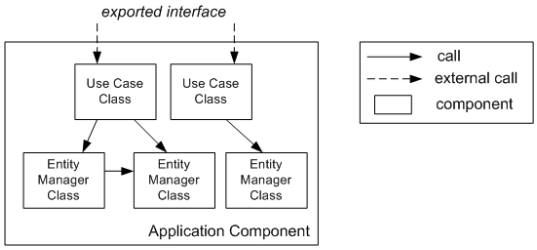

Figure 11: Structure of an application component

9.5 Standards

ANSI, ISO and DIN-standards (DIN are German industrial standards) are the basis of mechanical, chemical and electrical engineering. Nearly all standard elements, such as nuts and bolts, integrated circuits, and machine elements are standardized. Notions and their relationships, metrics, processes, notations are also standardized.

In software-engineering there are a some standards such as ANSI/IEEE 1471 (Software-Architecture), UML 2.0 (description techniques), ISO 15288 (Systems Engineering, System Life Cycle Processes) and others.

For example, the ANSI/IEEE 1471 defines practices for architecture description [83]. It defines the basic notions such as architecture, view and stakeholder and thier relationships. But nether a specific set of views (like in [79]) nor standard-architecture is defined.

In componentware some industry specifications (de facto standards) such as EJB or .NET exist. Basic architectural components are not yet standardized like the machine elements in the DIN-standards. For a mature engineering discipline a broad coverage with standards is essential.

10 Conclusion

Componentware is a broad subject. We have cited only a small amount of literature concerned with componentware and reached more than 100 citations. A simple classification scheme is essential not to loose the overview.

We have proposed a simple scheme in this paper with five categories:

Formal system model and theory : A well defined mathematical formalism, used to unambiguously express the basic building blocks and their relations as well as their properties: component, interface, connector and configuration.

Description technique : Textual or graphical notations hiding the underlying formalism to the user. Artifacts are diagram types for (graphically) representing static and dynamic aspects.

Software architecture : A well-defined approach for design and documentation of the system's architecture. This includes guidelines for refinement, component composition, quality properties and the over all structure of the system.

Development process : A component oriented process model to structure the development process. It defines tasks and their results as well as roles that are responsible for the fulfilment.

Tool support : Tools support the use of specifc description techniques and ideally the complete development process.

Considering the main characteristics of some well known componentware approaches we described these approaches and assigned them to one of the five categories.

In spite of the huge number of researchers and publications and the still increasing interest in industry there is a huge amount of open issues, inconsistencies and a babel of component notions. A number of open issues are enumerated in this paper (see sections 3 to 9) . We started a discussion on some questions. In our oppinion the most important issues are:

- A broadly accepted formal system model and theory to achieve rigour and precision in description techniques and software architecture.
- A set (taxonomy) of different component notions and methodology concerning the problem domain (embedded, critical, and other domains, see section 8), scale and granularity (see section 5) and the phases of the software life-cycle (see section 6). Different things should be named differently to enhance expressiveness.
- An engineering like consideration of componentware. Heuristics, rules of thumb, usage concepts and quality measures should be made explicit. A broad coverage with standards and templates is essential for an engineering discipline (see section 9).

We hope that this paper is a starting point for a broader discussion of componentware. It can be extended to a link-collection and overview of current research and industrial practice. A research and cooperation agenda might be a result as well.

References

[1] Acmestudio. http://www-2.cs.cmu.edu/ acme/AcmeStudio/AcmeStudio.html.

[2] Andomda. http://www.andromda.org/.

[3] Architecture description languages. http://www.sei.cmu.edu/str/descriptions/adl_body.html.

[4] Argouml. http://argouml.tigris.org/.

[5] Bea, ejbgen an enterprise javabeans 2.0 code generator. http://edocs.bea.com/wls/docs81/ejb/EJBGen_reference.html.

[6] Bow street. http://www.bowstreet.com.

[7] Bpml, business process management language. http://www.bpmi.org/bpml-spec.esp.

[8] C sharp programmer's reference, attributes tutorial. http://msdn.microsoft.com/library/default.asp.

[9] Collaxa. http://www.collaxa.com.

[10] Eclipse. http://www.eclipse.org/.

[11] Enterprise studio. http://www.borland.com/.

[12] Grand central communications. http://www.grandcentral.com/.

[13] Idea. http://www.intellij.com/.

[14] Java management extensions (jmx). http://java.sun.com/products/JavaManagement/.

[15] Java web start. http://java.sun.com/products/javawebstart/.

[16] Jsr 175, a metadata facility for the java(tm) programming language. http://www.jcp.org/en/jsr/detail?id=175.

[17] Jsr208, java(tm) business integration. http://www.jcp.org/en/jsr/detail?id=208.

[18] Jsr77, j2ee(tm) management. http://www.jcp.org/en/jsr/detail?id=77.

[19] Living software development process support tool. http://processpatterns.informatik.tu-muenchen.de/.

[20] Microsoft component object model (com). http://www.microsoft.com/com/.

[21] Oagis. http://www.openapplications.org.

[22] Polarlake. http://www.polarlake.com/.

[23] Rational rose. http://www.rational.com/products/rose/index.jsp.

[24] Rational xde. http://www.rational.com/products/xde/.

[25] Systinet. http://www.systinet.com.

[26] Telelogic tau uml suite. http://www.telelogic.com/products/tau/uml/.

[27] Tibco business works. http://www.tibco.com/solutions/products/active_enterprise/bw/default.jsp.

[28] Togehter. http://www.borland.com/together/index.html.

[29] Web services business process execution language. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel.

[30] webmethods integration platform. http://www.webmethods.com/cmb/solutions/integration_platform/.

[31] Wsci, web service choreography interface. http://www.w3.org/TR/2002/NOTE-wsci-20020808/.

[32] Wscl, web services conversation language. http://www.w3.org/TR/2002/NOTE-wscl10-20020314/.

[33] Xdoclet, attribute-oriented programming. http://xdoclet.sourceforge.net/.

[34] Xmi, xml metadata interchange. http://www.omg.org/technology/documents/formal/xmi.htm.

[35] Martín Abadi and Leslie Lamport. Composing specifications. Technical report, DEC SRC, 130, Lytton Av, Palo Alto, Ca 94301, 1990.

[36] Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–535, May 1995.

[37] Deepak Alur, John Crupi, and Dan Malks. *Core J2EE Patterns*. Prentice Hall, 2003.

[38] Luis F. Andrade and José Luiz Fiadeiro. Supporting architecture-based evolution. *submitted (www.fiadeiro.org/jose/papers/contracts.pdf)*, 2003.

[39] Dirk Ansorge, Klaus Bergner, Bernd Deifel, Nicolas Hawlitzky, Christoph Maier, Barbara Paech, Andras Rausch, Marc Sihling, Veronika Thurner, and Sascha Vogel. Managing componentware development – software reuse and the V-modell process. In Matthias Jarke and Andreas Oberweis, editors, *Advanced Information Systems Engineering*, volume 1626 of *Lecture Notes in Computer Science*, pages 134–148. Springer Verlag, June 1999.

[40] M. Arrais and José Luiz Fiadeiro. Unifying theories in different institutions. In M. Haveraaen, O. Owe, and O-J. Dahl, editors, *Recent Trends in Data Type Specification*, volume 1130 of *Lecture Notes in Computer Science*, pages 81–101. Springer-Verlag, 1996.

[41] Colin Atkinson et al. *Component-based Product Line Engineering with UML*. Addison-Wesley, 2001.

[42] Mike Barnett and Wolfram Schulte. Spying on components: a runtime verification technique. In Dimitra Giannakopoulou, Gary T. Leavens, and Murali Sitaraman, editors, *Specification and Verification of Component-Based Systems (SAVCBS Workshop affiliated with OOPSLA'01, Proceedings)*, volume ISU TR #01-09a, pages 7–13. Department of Computer Science, Iowa State University, 2001.

[43] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[44] Gerd Beneken and Manfred Schamper. Komponenten mit j2ee patterns (components with j2ee patterns). *Java Spektrum*, (2), 2002.

[45] K. Bergner, A. Rausch, M. Sihling, and A. Vilbig. Putting the parts togehter: Concepts, description techniques, and development process for componentware. In *33rd Hawaii International Conference on System Sciences volume 8*, 2000.

[46] K. Bergner, A. Rausch, M. Sihling, A. Vilbig, and M. Broy. *A Formal Model for Componentware*. 2000.

[47] Klaus Bergner, Heiko Lötzbeyer, Andreas Rausch, Marc Sihling, and Alexander Vilbig. A formally founded componentware testing methodology. In *First International Workshop on Automated Program Analysis, Testing and Verification (ICSE'00, Proceedings)*, June 2000.

[48] Klaus Bergner, Andreas Rausch, Marc Sihling, and Alexander Vilbig. Componentware – methodology and process. In *CBSE '99 Proceedings of the International Workshop on Component-Based Software Engineering*. IEEE, 1999.

[49] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 1999.

[50] M. Broy. A logical basis for component-based systems engineering. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. IOS Press, 1999.

[51] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, and R. Weber. The design of distributed systems - an introduction to FOCUS. Technical report, Technische Universität München, Institut für Informatik, Sonderforschungsbereich 342: Methoden und Werkzeuge für die Nutzung paralleler Architekturen TUM-I9202, 1992.

[52] Manfred Broy and Ketil Stølen. *Specification and Development of interactive Systems*. Springer, 2001.

[53] Jean-Michel Bruel, Tony Clark, Andy Evans, Robert France, Kevin Lano, Stuart Kent, Ana Moreira, and Bernard Rumpe. The precise uml group. http://www.cs.york.ac.uk/puml/.

[54] F. Buschmann, R. Meunier, H. Rohnert, M. Stal, and P. Sommerlad. *Pattern Oriented Software Architecture*. Wiley, 1996.

[55] UN/CEFACT Business Process Team and OASIS. Business process specification schema, July 2001. Version 1.01.

[56] John Cheesman and John Daniels. *UML Components, A Simple Process for Specifying Component-Based Systems*. Addison-Wesley, 2001.

[57] Siobhan Clarke and John Murphy. Verifying components under development at the design stage: A tool to support the composition of component design models. In *International Workshop on Component-Based Software Engineering (ICSE'98, Proceedings)*, April 1998.

[58] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2003.

[59] D.F. D'Souza and A.C. Wills. *Objects, Components and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1998.

[60] B. Dutertre and V. Stavridou. A model of noninterference for integrating mixed-criticality software components. San Jose, CA, January 1999.

[61] Uml profile for enterprise distributed object computing specification, February 2002.

[62] Bernd Finkbeiner and Ingolf Krüger. Using Message Sequence Charts for component-based formal verification. In Dimitra Giannakopoulou, Gary T. Leavens, and Murali Sitaraman, editors, *Specification and Verification of Component-Based Systems (SAVCBS Workshop affiliated with OOPSLA'01, Proceedings)*, volume ISU TR #01-09a, pages 32–45. Department of Computer Science, Iowa State University, 2001.

[63] David S. Frankel. *Model Driven Architecture*. Wiley, 2003.

[64] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA, 1994.

[65] D. Garlan and R. Allen. Formalizing architectural connection. In *Proceedings of the 16th International Conference on Software Engineering. Sorrento, Italy, May 16-21*, pages 71–80. CA: IEEE Computer Society Press, Los Alamitos, 1994.

[66] W. Grieskamp, M. Heisel, and H. Dörr. Specifying Safety-Critical Embedded Systems with Statecharts and Z: An Agenda for Cyclic Software Components. *Science of Computer Programming*, 2000.

[67] J. Grünbauer, H. Hollmann, J. Jürjens, and G. Wimmel. Modelling and verification of layered security protocols: A bank application. In *SAFECOMP 2003*, Lecture Notes in Computer Science, Edinburgh, GB, 23-26 September 2003.

[68] Peter Herzum and Oliver Sims. *Business Component Factory: A Comprehensive Overview of Component-Based Development for the Enterprise*. Wiley, 2000.

[69] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[70] C.A.R. Hoare, S.D. Brookes, and A.W. Roscoe. A theory of communicating sequential processes. Technical monograph prg-21, Oxford University Computing Laboratory Programming Research Group, Oxford, 1981.

[71] Christine Hofmeister, Robert Nord, and Dilip Soni. *Applied Software Architecture*. Addison-Wesley, 1999.

[72] Workshop on Systems Engineering and the Information Infrastructure (final report), July 1996.

[73] R. Jeffords and C. Heitmeyer. A strategy for efficiently verifying requirements specifications using composition and invariants. In *European Software Engineering Conference/ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2003)*, Helsinki, Finland, September 1-5 2003.

[74] J. Jürjens. Secure information flow for concurrent processes. In C. Palamidessi, editor, *CONCUR 2000 (11th International Conference on Concurrency Theory)*, volume 1877 of *Lecture Notes in Computer Science*, pages 395–409, 2000.

[75] J. Jürjens. UMLsec - Presenting the Profile. In *Sixth Annual Workshop On Distributed Objects and Components Security (DOCsec2002)*, Baltimore, MD, March 18-21 2002. OMG. Half-day tutorial.

[76] J. Jürjens. Developing safety-critical systems with UML. In P. Stevens, editor, *UML 2003 – The Unified Modeling Language*, Lecture Notes in Computer Science, San Francisco, Oct. 20–24 2003. 6th International Conference.

[77] J. Jürjens. *Secure Systems Development with UML*. 2003. In preparation.

[78] J. Jürjens, V. Cengarle, E. Fernandez, B. Rumpe, and R. Sandner, editors. *Critical Systems Development with UML*, number TUM-I0208 in TUM technical report, 2002. UML'02 satellite workshop proceedings.

[79] Philippe Kruchten. Architectural blueprints - the 4+1 view model of software architecture. *IEEE Software*, 12(6):42–50, 1995.

[80] Philippe Kruchten. *The Rational Unified Process, An Introduction*. Addison-Wesley, 2nd edition, 2000.

[81] Keith Levi and Ali Arsanjani. A goal-driven approach to enterprise component identification and specification. *Communications of the ACM*, 45(10):45–52, 2002.

[82] David C. Luckham and other. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21:336–355, 1995.

[83] Mark Maier, David Emery, and Rich Hilliard. Software architecture: Introducing ieee standard 1471. *IEEE Computer*, 34(4):107–109, 2001.

[84] J. May. Testing the reliability of component-based safety critical software. In S. Thomason, editor, *20th International System Safety Conference*, pages 214–224. System Safety Society, August 2002.

[85] G. McGraw and J. Viega. Why cots software increases security risks. In *ICSE Workshop on Testing Distributed Component-Based Systems*, May 1999.

[86] John D. McGregor. Verification and validation issues and implications for reuse. In *Workshop on Verification and Validation of Models and Simulations (Foundations'02, Proceedings)*, October 2002.

[87] D. McIlroy. Mass-produced software components. In *Software Engineering, NATO Science Committe report*, pages 138–155, 1968.

[88] J. McLean. A general theory of composition for a class of "possibilistic" properties. *IEEE Transactions on Software Engineering*, 22(1):53–67, 1996.

[89] Nenad Medvidovic, David S. Rosenblum, David F. Redmiles, and Jason E. Robbins. Modeling software architectures in the unified modeling language. *ACM Transactions on Software Engineering and Methodology*, 11(1):2–57, 2002.

[90] Nenad Medvidovic and Richard N. Taylor. A framework for classifying and comparing architecture description languages. In M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 60–76. Springer–Verlag, 1997.

[91] Bertrand Meyer. Applying design by contract. *IEEE Computer*, pages 40–51, May 1992.

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

[93] Bertrand Meyer. The grand challenge of trusted components. In *Proceedings of the 25th International Conference on Software Engineering ICSE*, pages 660–667, Portland, Oregon, 2003.

[94] Bertrand Meyer, Christine Mingins, and Heinz Schmidt. Trusted components for the software industry. *IEEE Computer*, May 1998.

[95] SUN Microsystems. Enterprise javabeans(tm) specification 2.1 proposed final draft 2. http://java.sun.com/webapps/download/Display, 03.06.2003.

[96] Joaquin Miller and Jishnu Mukerji. Mda guide version 1.0. Technical report, mai 2003.

[97] R. Milner. *A Calculus of Communicating Systems, Lecture Notes in Computer Science 92*. Springer, 1980.

[98] OMG. Corba components. http://www.omg.org/docs/formal/02-06-65, 2002.

[99] OMG. Unified modeling language: Superstructure. citeseer.nj.nec.com/weck96do.html, 2003.

[100] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[101] D.L. Parnas, P. Clements, and D.M. Weiss. Enhancing reusability with information hiding. In *Proceedings of ITT Workshop on Reusability in Programming*, Stratford, CT, 1983.

[102] Andreas Rausch. Design by contract + componentware = design by signed contract. *Journal of Object Technology*, 1(3), 2002.

[103] Eberhardt Rechtin and Mark Maier. *The Art of Systems Architecting*. CRC Press, 2nd edition, 2000.

[104] Rosettanet homepage - standards, 2002. http://www.rosettanet.org/standards.

[105] Bernhard Schätz and Franz Huber. Integrating formal description techniques. In *[?]*, pages 1206–1225, 1999.

[106] Bernhard Schätz and Franz Huber. Integrating formal description techniques. In *[?]*, pages 1206–1225, 1999.

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

[108] Mary Shaw and David Garlan. *Software Architecture, Perspectives of an Emerging Discipline*. Prentice Hall, Upper Saddle River, NJ, 1996.

[109] Mary Shaw and other. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21:314–335, 1995.

[110] Johannes Siedersleben. Quasar: Die sd&m standardarchitektur. Technical report, sd&m AG, 2002.

[111] Johannes Siedersleben, Gerhard Albers, Peter Fuchs, and Johannes Weigend. Segregating the layers of business information systems. In Patrick Donohoe, editor, *Software Architecture, TC2 First Working IFIP Conference on Software Architecture (WICSA1), 22-24 February 1999, San Antonio, Texas, USA*, 1997.

[112] Dilip Soni, Robert L. Nord, and Christine Hofmeister. Software architecture in industrial applications. In *International Conference on Software Engineering*, pages 196–207, 1995.

[113] Rupert Stüetzle. *Wiederverwendung ohne Mythos: Empirisch fundierte Leitlinien für die Entwicklung wiederverwendbarer Software*. PhD thesis, Technische Universität München, 2002.

[114] C. Stutz, J. Siedersleben, D. Kretschmer, and W. Krug. Analysis beyond uml. pages 215–218, 2002. In [**?**].

[115] C. Szyperski and C. Pfister. Workshop on component-oriented programming, summary. In M. Mühlhäuser, editor, *Special Issues in Object-Oriented Programming - ECOOP 96, Workshop Reader*, Heidelberg, 1997. dpunkt Verlag.

[116] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component Software, Beyond Object-Oriented Programming*. Addison-Wesley, 2nd edition, 2002.

[117] B. Travis. *XML and SOAP Programming for BizTalk Servers*. Microsoft Press, 2000.

[118] OMG Unified Modeling Language Specification 1.4, 2002.

[119] Christina Wallin. Verification and validation of software components and component-based software systems. In Ivica Crnkovic and Magnus Larsson, editors, *Building Reliable Component-Based Software Systems*. Artech House Publishers, 2002.

[120] Edward Yourdon. *Techniques of Program Structure and Design*. Prentice-Hall, Englewood Cliffs, NJ, 1975.