International Journal of Software Engineering and Knowledge Engineering
Vol. 0, No. 0 (0000) 000–000
© World Scientific Publishing Company

COMPONENT-BASED SOFTWARE ENGINEERING

WILHELM HASSELBRING
Software Engineering Group, Computer Science Department, University of Oldenburg
PO Box 2503, D-26111 Oldenburg, Germany

With component-based software engineering, it is expected that software systems can be created and maintained at lower costs and with increased stability through reuse of approved components in flexible software architectures. We survey the state-of-the-art of this approach, hereby discussing issues such as reuse, architecture, various forms of component integration, component middleware, and impact on the development process.

Keywords: Software Components, Software Reuse, Software Architecture, Component Integration.

1. Introduction

A software component can be defined as an executable unit of code that provides a set of services through specified interfaces:

> "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 third-party composition." [1]

The kind of 'unit' is important: As a technical goal, minimal coupling with the outside world and maximal cohesion inside the unit is desirable. However, component-based software engineering has impact from both a managerial and a technical perspective.

1.1. *Managerial Goals for Component-Based Software Engineering*

The managerial goals of component-based software engineering can be identified as follows:

Cost Reduction: One important goal of any development process is to be cost-effective in developing the software system.

Ease of Assembly: Components need to be designed in a manner that facilitates the subsequent assembly process when components are assembled together to develop software systems.

Reusability: This addresses the component's potential for reuse in multiple applications. In component-based software engineering, reusability is the extent to which a component is (re)used during the assembly process in developing software systems.

Customization and Flexibility: In component-based software engineering, when a set of components are made available to the application developers, they can customize their systems by assembling components according to their specific requirements.

Maintainability: Maintainability is the ease with which software features can be added, removed, or modified in a component; according to new and emerging requirements.

1.2. *Technical Features of Component-Based Systems*

The technical features of component-based systems can be summarized as follows:

Coupling (inter-relatedness among components)

In component-based software engineering, coupling for a component is defined as the extent to which that component is coupled with other components. Low coupling is desired.

Cohesion (strength of association among elements within a component)

Cohesion refers to the strength of association of elements within a system. In component-based software development, cohesion of a component is the extent to which its contained elements are inter-related. High cohesion is desired.

Number and Size of Components (number of components in a system, complexity)

In component-based software engineering, the number of components used to realize a particular system is an important design parameter. The trade-off between many small components and a few large components must be considered in component and system design.

1.3. *Chapter Overview*

The chapter starts with a look back at object-oriented development as one basis for components. A major part then discusses software reuse and software architectures. Various forms of component integration, such as integrating legacy systems and enterprise/office application integration are discussed subsequently. Component middleware, the impact on the development process, and questions of finding the right components are discussed before we summarize the chapter.

2. Object-Oriented Development as a Basis for Components

The basic idea of object-oriented development is that a software system consists of a set of interacting objects [2]. A class describes the structure and behavior of objects. Objects encapsulate information (information hiding). This capability is highly relevant for component-based systems: client components are not interested in the internals of server components, only the services that are specified in their interfaces should be required for using those components. Therefore, objects are good candidates for components, provided their granularity is sufficient.

Object-Oriented *analysis* aims at understanding the application domain and identifying requirements. Object-Oriented *design* and *implementation* aim at achieving the identified requirements in a particular environment. Table 1 illustrates the basic similarities and differences between objects and components (based on [1]). As objects get instantiated, there needs to be a ‘template’ for their creation, which is their class. The creation mechanism can be regarded as an *object factory* that produces individual objects with own identity and (persistent) state. Opposed to that, components are deployed at some sites. Several objects could be managed by deployed components. Granularity (size of components) is an important issue within this context.

Objects	**Components**
An object is a unit of instantiation; it has a unique identity.	A component is a unit of independent deployment.
An object has state; this state can be persistent state.	A component has no persistent state.
An object encapsulates its state and behavior	A component is a unit of third-party composition.

Table 1: Basic similarities and differences between objects and components.

Object-oriented programming languages usually do not support explicit 'uses' relationships as they are available with module definition languages [3] and module interconnection languages [4, 5]. This restriction has good reasons: 'uses' defines implementation rather than abstraction in object-oriented programming. However, for composing components, 'uses' relationships are relevant. Similar to using a class in object-oriented programming, with component-based software engineering it is relevant what a component offers, not how it realizes offered services. Anyway, it is possible to realize components without employing object techniques. A well-organized library of functions in a procedural setting is a good basis for component-based development, too.

It is important to note that inheritance is problematic for component composition, because it breaks encapsulation. Inheritance is an 'open' relation among classes: the inheriting class usually has full access to the internals of the base class. This allows for extensibility and incremental construction of classes. Conversely, uses relationships among classes are 'closed' relations: only the public interfaces are shared, the internal implementations is encapsulated to decouple the classes. This distinction is also called the open/closed principle [2]. With inheritance, the fragile base class problem may emerge: the question whether a base class can evolve without breaking independently developed subclasses. This problem may cause unacceptable dependencies (coupling) among components. As discussed earlier, in component-based software engineering, low coupling and high cohesion are desired. For a more detailed discussion of the fragile base class problem refer to [1, Section 7.4].

3. Reuse of Components and Software Architectures

We can distinguish two classes of concerns based on whether components are used as a design philosophy independent from any concern for reusing existing components, or seen as off-the-shelf building blocks used to design and implement a component-based system.

Reuse on the design level plays an important role in component-based software engineering. On the programming level, reuse is usually accomplished by means of high-level programming language constructs, function libraries, or object-oriented class frameworks. For well-understood domains, generators [6] may be used for assembling applications by automatically composing pre-written components. On the design level, design patterns and established software architectures are essential. However, the borderline between design and programming cannot always be pinpointed precisely.

3.1. *Domain Engineering*

The inability of software developers to achieve low costs, high productivity, and consistent quality has often been attributed to a lack of software reuse. Re-creating

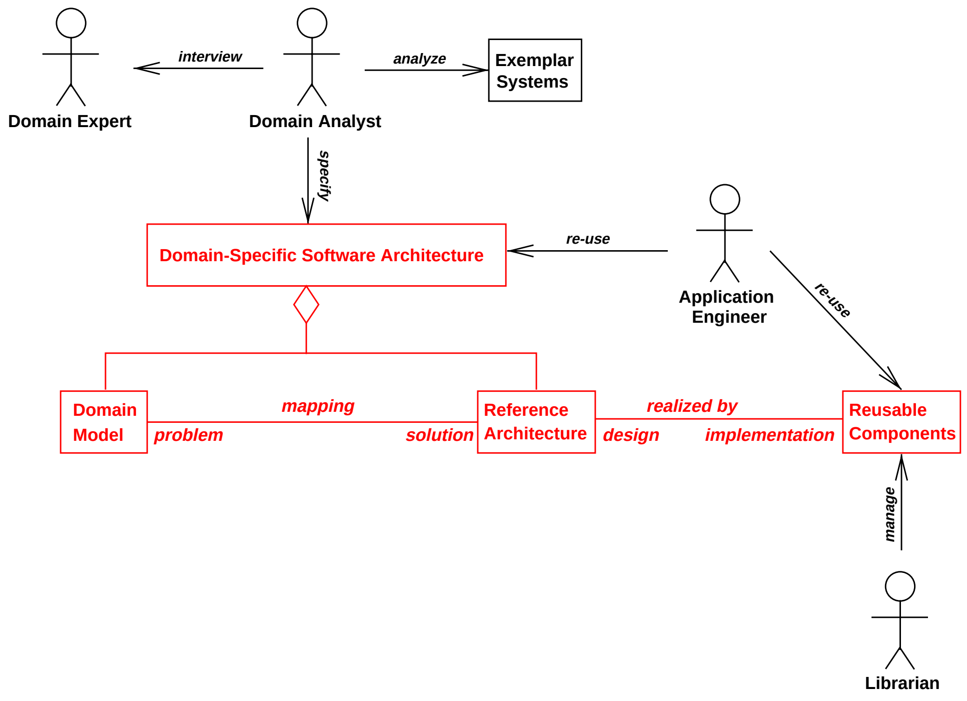

Figure 1: Relations between some roles and artifacts in the DSSA engineering process. Hollow diamonds indicate part-of relations. We use the UML notation for actors to model the roles [12].

similar systems without capturing components for future development wastes time, money, and human resources. While a universal software reuse solution may prove ineffective, significant improvements can be realized by focusing on well-understood domains. Architectures support domain-specific reuse by serving as frameworks for understanding families of systems, which may be called *product lines* [7, 8, 9].

Domain engineering [10] is an activity for building reusable components, whereby the systematic creation of domain models and architectures is addressed. Domain engineering aims at supporting *application engineering* which uses the domain models and architectures to build concrete systems. The emphasis is on reuse and product lines. The Domain-Specific Software Architecture (DSSA) engineering process was introduced to promote a clear distinction between domain and application requirements [11]. A Domain-Specific Software Architecture consists of a domain model and a reference architecture, and guides in reusing components as modeled in Figure 1. Appropriate management of component libraries is essential for successful reuse. The DSSA process consists of domain analysis, architecture modeling, design and implementation stages as illustrated in Figure 2.

Domain models represent the set of requirements that are common to systems within a specific domain. Usually, those systems can be grouped into product lines [7, 8, 9], for instance for the insurance or banking domain. There may be many domains, or areas of expertise, represented in a single product line and a single domain may span multiple product lines. *Domain analysis* is the process of identifying, collecting, organizing, and representing the relevant information in a domain, based upon the study of existing systems and their development histories, and knowledge

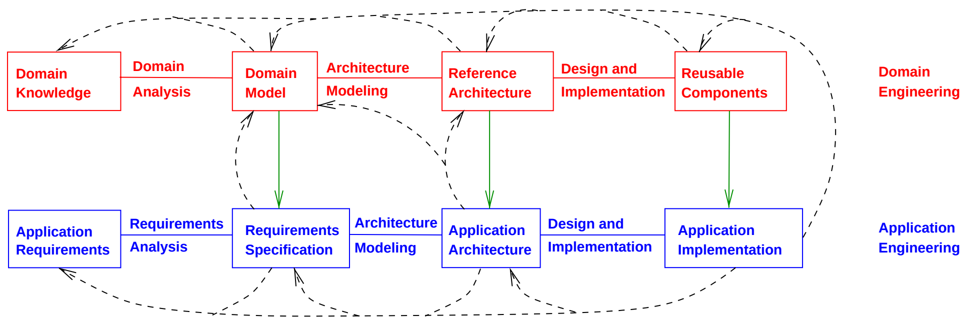

Figure 2: The DSSA engineering process. In application engineering, software systems are developed from reusable components created by a domain engineering process. As indicated by the dashed arrows, various forms of feedback are possible.

captured from domain experts. Product lines cover both commonality and variance for a family of systems. Figure 1 illustrates the relations between some roles (domain experts, domain analysts, application engineers, and librarians) and the artifacts in the DSSA engineering process. Reference architectures are the structures used to build systems in a product line. The domain model characterizes the *problem space*, while the *reference architecture* addresses the *solution space*, as illustrated in Figure 1.

In *application engineering*, a developer uses the domain models within the product line to understand the capabilities offered by the reference architecture and specifies a system for development. The developer then uses the reusable components to build the system. An architectural model is developed from which detailed design and implementation can be done. Application engineers use the domain models with the users to elicit the requirements for the planned software systems. By so doing, the models cover the user's needs in terms of existing models. Those needs not covered by a domain model are new requirements. The domain analysts may choose to update a domain model with the new requirements. As indicated by the dashed arrows in Figure 2, various forms of feedback are possible.

3.2. *Software Architectures for Structuring Components*

For large, complex software systems the design of the overall system structure (the software architecture) is a central problem. The *architecture* of a software system defines that system in terms of components and interactions/connections among those components [13, 14, 15, 16]. It is not the *design* of that system which is more detailed. The architecture shows the correspondence between the requirements and the constructed system, thereby providing some rationale for the design decisions. Different views on component-based software architectures may be distinguished:

- Design-time: This includes the application-specific view of components, such as functional interfaces and component dependencies.
- Compose-time: This includes all the elements needed to assemble a system from components, including generators and other build-time services; a component framework may provide some of these services.
- Runtime: This includes frameworks and models that provide runtime services for component-based systems.

These characteristics suggest that components are complex design-level entities, that is, both abstractions and implementations. Some authors, additionally, consider all artifacts that are produced during the different phases of the development process as components (for instance, parts of design specifications) [17]. However, in this chapter we restring ourselves to software components. Anyway, specifications should be structured in modular ways, too.

Architectures can themselves be reusable assets, but they also support the reuse of design and code components by standardizing interfaces, protocols, and the packaging of functionality at a high level of abstraction, to be further refined at more detailed levels. Standardization facilitates smooth integration of both off-the-shelf and custom components during initial development and subsequent maintenance phases.

3.3. *Object-Oriented Application Frameworks and Design Patterns*

Object-oriented application frameworks are class hierarchies plus models of interactions which can be turned into complete applications through various kinds of inheritance and usage associations [18, 19]. An object-oriented framework defines a class structure as well as an interaction model for cooperating objects involved. Thus, besides a generic architecture, also controls for specific applications are offered; the places where specific functionality has to be added are predefined. Design patterns often guide the construction and documentation of frameworks [20]. Frameworks can offer reusable components in the domain engineering process.

Design patterns [21] are "micro-architectures" while software architectures are more coarse-grained designs. A design pattern describes a family of solutions to a recurring problem. Patterns form larger wholes like pattern languages, systems or handbooks when woven together so as to provide guidance for solving complex problem sets. Patterns express the understanding gained from practice in software design and construction. Design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context. The patterns community catalogs useful design fragments and the context that guides their use. They do not make special distinctions between architectural patterns and patterns for code. An organized collection of related patterns for a particular application domain can be called a *pattern language*.

All aspects of software systems, their development and their deployment are suitable topics of individual patterns or comprehensive pattern languages. Patterns might be so specific as to name particular objects, their responsibilities, and interaction. A well-known pattern of this kind is, for example, the Observer pattern from [21]. It supports keeping co-operating components consistent, with help of a change propagation mechanism.

The run-time architecture of a framework is characterized by an inversion of control: event handler objects of the specific application are invoked via the framework's reactive dispatching mechanism. In general, event-based interaction is very important for component-based systems. With traditional imperative and functional programming, the control flow is usually organized within a hierarchical call graph. With event-based interaction, the control flow is not restricted to hierarchies allowing for a loose coupling among (autonomous) components. For instance, the CORBA event services [22] may be used to provide synchronous or asynchronous transfer of objects using event channels to decouple the communication between distributed objects. Consumer objects can either receive notification of events that concern them (*push* model) or can connect to the event channel to wait for their events (*pull* model). Figure 3 illustrates both mechanisms, where suppliers deposit information at the event channel and consumers fetch them in different ways. The event service can be implemented as a specialized CORBA object which means that it can be used by multiple suppliers and consumers simultaneously. In effect, this means that multiple suppliers can pass information to multiple consumers using

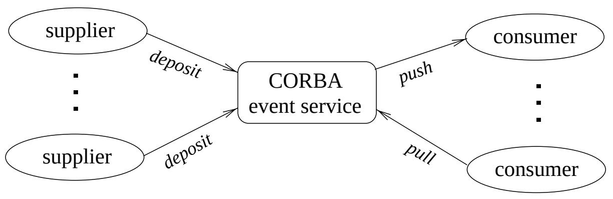

Figure 3: Decoupled push and pull communication with CORBA event services.

the same event channel without any supplier or consumer having direct knowledge of each other. Suppliers and consumers may register and unregister with an event channel with no consequences to either other suppliers who could be providing events or consumers who may be listening for events. Event channels provide a decoupling of the consumer and producer components.

4. Component Integration

Components need to be integrated to constitute usable software systems. Various forms of component integration, such as integrating legacy systems and enterprise/office application integration are discussed in the following subsections.

4.1. *Integrating Legacy Systems into Component-Based Systems*

There is often no time and justification to replace existing legacy systems. New functionality must be integrated with other components, existing applications and data sources. Component-based software engineering aims at building applications that are adaptable to business and technology changes, while retaining legacy applications and legacy technology as far as possible and reasonable. The speed of business and technology change does not allow time for total replacement, therefore, evolution and migration of legacy and new application systems is required [23]. Migration and evolution aim at protecting existing investments and enabling rapid response to the changing user requirements.

When the legacy components are information systems, an often occurring requirement is that the systems to be integrated are to remain autonomous [24]. Preexisting applications (legacy systems) must still be able to use their local data without modification. In this way (financial) investment can be preserved and a smooth migration towards modern systems can take place. A federated database system is an integration of such autonomous database systems, where both local applications and global applications accessing multiple database systems are supported [25]. The notion of 'federation' is originally a political term: several states join together and constitute a federal system in which each state retains its autonomy up to a certain degree. This idea of federation can be transferred to the integration of preexisting information systems which could have been developed independently (autonomously) within different departments of an enterprise.

On the technical level, wrappers [26] that provide unified interfaces are an established technique for accessing legacy systems in a component-based setting. Here, the question arises, how to freeze interfaces and let the components evolve without further interaction. For enabling such an evolution, a modular design [27] of the overall system is required. The interfaces between the independent components form *contracts* among the involved parties. Those contracts have to be fulfilled. With advanced interface definition languages that define, for instance, pre- and post-conditions for services, a middleware system could check the compliance with the contract. As an example, the Eiffel language supports the *design by contract* principle, where such conditions are checked by the runtime system [2].

4.2. *Enterprise Application Integration*

With Enterprise Application Integration the goal is to integrate independent Enterprise Resource Planning (ERP) systems. This is usually achieved by means of some kind of messaging services. Even the SAP R/3 approach, which basically aims at enterprise integration via one single database (no borders between enterprise units), meanwhile acknowledges the fact that specific services are required for integrating autonomous ERP systems, both within and across enterprises [28]. TSI Software's Mercator product, for instance, specializes on pre-built application adapters, data transformations, and messaging services for the ERP systems SAP R/3 and PeopleSoft.

The deployment of ERP systems often requires the business processes to re-engineer to align with the ERP system. However, it is usually unacceptable to require the business to change to the applications functionality; instead the information architecture should align with the business organization. Component-based software engineering aims at supporting the business processes with flexible component architectures that are readily adaptable to changing requirements.

For Enterprise Application Integration, applications need to *understand* the data provided by other applications, for instance a common understanding is required of what a person's bank account is. Standardization of message formats and message contents plays an important role in that context [29]. Meanwhile, XML [30] emerges as the standard for defining the syntax of data structures to be transferred over the Internet. In order to provide interoperability across implementations, the concrete syntax *and* the semantics of standardized messages must be defined. Traditional EDI (Electronic Data Interchange) is often being re-examined to define the *meaning* of the transferred data, and XML is employed as the practical foundation in which to structure this information [31].

A technical and organizational question is where to store and manage schemas and Document Type Definitions (DTDs) for XML messages. XML.org, for instance, aims at being an independent industry portal for the standardization of XML applications in electronic commerce, whereby it serves as a reference for XML DTDs. BizTalk.org is a competing industry initiative started by Microsoft, and mySAP.com is SAP's approach. These initiatives are highly relevant for integrating enterprise applications of dissimilar organizations.

4.3. *Office Application Integration*

With Enterprise Application Integration, mainly the 'backoffice' part of software systems is addressed. Another domain is to integrate software systems on the personal desktop computer. For instance, [32] report on the development of a dynamic fault-tree analysis tool, which is an assemblage of commercial off-the-shelf tools. The composed tool integrates Visio Corporation's drawing application Visio Technical, Microsoft's Word, and Microsoft's Internet Explorer into one user interface. Technically, Microsoft's OLE is used to drive the components through their application programming interfaces and Microsoft's Active Document Standard is used to integrate the individual user interfaces. A consistency maintenance engine has been built to keep the graphical (Visio) and textual (Word) tools consistent.

In general, such Office Application Integration works well with a small number of (coarse-grained) components. If those components are from one vendor, the integration task is manageable (meanwhile, Visio has been acquired by Microsoft). Anyway, the approach has serious limitations:

- It is usually limited to the integration on personal computers. Heterogeneous environments are not supported.

- It is limited to a small number of components. These components are usually already complex software systems.

- New versions of the components may require re-building the composed application.

However, if it works, powerful applications may be constructed with limited effort.

5. Middleware for Component-Based Software Engineering

Primarily, component-based software engineering is a fundamental approach for structuring systems in a modular way. To support the actual construction of component-based systems, middleware for connecting components provides a technical foundation. Currently, the three major middleware approaches for component-based software engineering are:

- The Common Object Request Broker Architecture (CORBA) of the Object Management Group (OMG) [33].
- Sun's Java Beans and Enterprise Java Beans [34].
- Microsoft's Component Object Model (COM+) [35].

In addition, XML can be combined with such object-oriented middleware, whereby data is encoded in XML and transferred via the middleware system. An example is the Tigra architecture which combines XML with CORBA [36].

When employing such middleware, the following architectural concepts are relevant:

- Components that are the building blocks.
- Connectors that provide the 'glue' among the components.
- Containers that deliver some non-functional quality services to contained components. Examples are persistence, transactions, and security.

Using such middleware, programmers are expected to be able to get their applications running quicker. In addition, these technologies often provide graphical development tools to drag and drop any component into their applications, where (in the most optimistic situation) they are included in a plug-and-play way. As a result, programmers are allowed to focus their efforts on their particular application problems.

To allow third-party composition of components, standards for the technical infrastructure are important. Basically, tow approaches to standardization are followed:

- Implementation follows standardization (the OMG way)
- Standardization follows implementation (the Microsoft way)

The first approach puts emphasis on well-defined interfaces, while the second puts emphasis on available implementations.

Middleware Integration addresses the syntactical level ('plumbing' and 'wiring') while Enterprise Application Integration also addresses a semantical level. The borderline between Enterprise Application and Middleware Integration cannot always be pinpointed precisely. For instance, the Object Management Architecture of the OMG defines the Object Request Broker, which can be deployed for middleware integration, and also high-level services (e.g., business objects) that address Enterprise Application Integration.

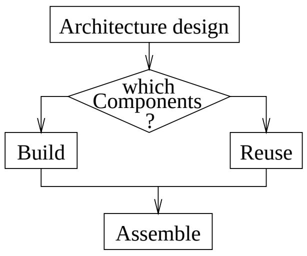

Figure 4: The selection process for components.

6. Impact on the Development Process

Component-based software engineering changes the focus of software engineering from one of traditional system specification and construction to one requiring simultaneous consideration of

- the system context (system characteristics such as requirements, cost, schedule, operating and support environments),
- and also the capabilities legacy systems and products in the marketplace for components. Figure 4 illustrates that selection process for components.

Essentially, the development process changes from a top-down partitioning of the problem towards a bottom-up assembly of available components. In a comprehensive development process both basic approaches should be combined [37].

The impact of this fundamental change is profound. Not only must engineering activities such as requirements specification change to support simultaneous consideration of system requirements and the marketplace, but so must acquisition processes and contracting strategies. For example, integration contractors and commercial product vendors must be treated as partners and rewarded for identifying the best value to be achieved through the (re-)use of (third-party) component products.

The design, development, and maintenance of component-based systems, and the migration of legacy systems towards components are complex. New products and technologies constantly emerge into the marketplace. The vendors of existing products work to differentiate their products from those of competitors. This leads to a marketplace characterized by a vast array of products and product claims, extreme quality and capability differences between products, and many product incompatibilities, even when they adhere to the same middleware standards.

7. Finding Components

For organizations designing and implementing a component-based system, or upgrading a legacy system with of-the-shelf components, the current market state presents a number of challenges. It is difficult to discover the actual technical capabilities of a product or set of competing products, since there is no objective forum for product evaluation. Once individual products are selected, it is difficult to identify and resolve mismatches between products, and to avoid becoming dependent on the products of a single vendor or set of vendors. Equally difficult but necessary is the ability to forecast what technologies and products will be relevant over the life of the system. Professional library management is a prerequisite for finding the right components. As illustrated earlier in Figure 1, a separate role — the librarian — is responsible for managing the reusable components in a way that the application engineers are able to effectively and efficiently reuse those (high quality) components.

The question arises, how domains and components can be described for retrieval. Important in this context are classification techniques. Classification aims at grouping 'things' into classes with common characteristics. Basically, two approaches to classifying classes exist:

- hierarchical enumeration of classes, and
- grouping on the basis of features (facets).

Faceted classification assigns attributes to software components [38]. These facets are then used for retrieving components with required features.

When organizing a library of components (in a hierarchy or by facets), it is important to use appropriate terms for the components and their attributes. The term 'ontology' has been used in many ways and across different communities. In general, each person has her individual view on the world and the things she has to deal with every day. However, there is a common basis of understanding in terms of the language we use to communicate with each other. Terms from natural language can therefore be assumed to be a shared vocabulary relying on a (mostly) common understanding of certain concepts with only little variety. This common understanding relies on the idea of how the world is organized. We often call this idea a 'conceptualization' of the world. Such conceptualizations provide a terminology that can be used for communication.

The main problem with the use of a shared terminology according to a specific conceptualization of the world is that much information remains implicit. When a mathematician talks about a binomial he has much more in mind than just the formula itself. He will also think about its interpretation (the number of subsets of a certain size) and its potential uses (e.g., estimating the chance of winning in a lottery). Ontologies have set out to overcome the problem of implicit and hidden knowledge by making the conceptualization of a domain (e.g., mathematics) explicit. Ontologies are explicit specifications of conceptualizations [39].

An ontology is used to make assumptions about the meaning of a term available. It can also be seen as an explication of the context a term is normally used in. There are many different ways in which an ontology may explicate a conceptualization and the corresponding context knowledge. The possibilities range from a purely informal natural language description of a term corresponding to a glossary up to strictly formal approaches with the expressive power of full first order predicate logic or even beyond.

Standardized terminologies and ontologies could be helpful, but so far no generally accepted standards exist. The standardization of domain-specific CORBA facilities, for instance, is an initiative in the right direction. Within the standardization efforts of the Object Management Group Domain-Specific Task Forces, several Business Object Facilities are standardized. The scope of the OMG's Financial Domain Task Force (CORBAfinancials Task Force), for instance, comprises financial services and accounting. The Insurance Working Group is part of the OMG's Financial Domain Task Force. The Insurance Working Group aims at developing domain-specific interfaces to enable insurance companies and other financial institutions to leverage purchased componentry and integrate their data. Another area is addressed by the Manufacturing Domain Task Force whose goals are interoperable manufacturing domain software components through CORBA technology. More information is available at `http://www.omg.org/`. Such standards are also useful for semantic interoperability of information system components [29].

8. Summary

Despite the fact, that the discussion on component-based software development started some decades ago [40], we seem today not even have realized the full potential of this software construction approach; not only for technological reasons. In the beginning there was an emphasis on compositional reuse, with libraries of components which were supposed to be interconnected. During the early 1990s the reuse community found out that this did not lead to much, i.e., the cost of finding, adapting and integrating small components did not pay off [41]. Meanwhile the focus is very much on product lines, component frameworks, and software architectures.

Beside the various technological deficiencies accompanied with many decisions to take such as:

- Which component architecture to take?
- Which components are most appropriate?
- Which infrastructure/middleware is needed to develop, deploy, and maintain component-based software systems?
- How to develop components, if no reusable components are available?
- How to test components and component-based systems?
- How can component mismatches [42] be rectified in a system?
- How can we engineer system attributes such as reliability, security, and performance in spite of decreasing control over individual system components?
- How do we integrate component products with the legacy code that continues to provide the core of many systems?

Beside that, organizational issues need to be taken into account, for instance:

- The right granularity of the components for successful marketing.
- Searching for components and finding the right ones.
- Describing the functionality of components (semantics).
- Versioning and configuration management for components.
- Buying components, not building them yourself. The 'not-invented-here' syndrome comes into play: programmers often do not trust the code of others.

Many problems to be solved are non-technical! Thus, in designing and constructing a component-based system, or in modifying a legacy system to take advantage of component architectures, an organization should find answers to those questions. It is also important to realize that only in the long run, investments in reusable components will generate profits.

Anyway, component-based software is a promising technology for increasing developer productivity and software quality. Unfortunately, many available components do not meet the users' expectations, even when they are based on popular component middleware, such as JavaBeans, COM+, or the CORBA Component Model. The reason is simple: middleware component models only provide the infrastructure that forms the basic building blocks for large-scale systems. The internal design of components, particularly the aspects related to component evolution, component configuration, distribution, concurrency, and scalability, are not addressed by most conventional component models. Handling these issues effectively remains the responsibility of component developers and assemblers.

The promises of component-based software engineering are cost reduction and increased quality through reuse of proven components, as well as better maintainability through flexible software architectures. Possible pitfalls for establishing component-based software engineering are performance issues (modular systems are often less resource efficient than monolithic systems), security, safety, trust in third-party components, and configuration management of complex componentized systems.

References

1. C. Szyperski. *Component Software*. Addison-Wesley, Harlow, England, 1998.
2. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, NJ, second edition, 1997.
3. C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice-Hall, Englewood Cliffs, NJ, 1991.
4. R. Prieto-Diaz and J.M. Neighbors. Module Interconnection Languages. *Journal of Systems and Software*, 6(4):307–334, November 1986.
5. M.D. Rice and S.B. Seidmann. A formal model for module interconnection languages. *IEEE Transactions on Software Engineering*, 20(1):88–101, January 1994.
6. D. Batory, G. Chen, E. Robertson, and T. Wang. Design wizards and visual programming environments for GenVoca generators. *IEEE Transactions on Software Engineering*, 26(5), May 2000.
7. R. Macala, L. Stuckey, and D. Gross. Managing domain-specific, product-line development. *IEEE Software*, pages 57–66, May 1996.
8. D. Dikel, D. Kane, S. Ornburn, W. Loftus, and J. Wilson. Applying software product-line architecture. *Communications of the ACM*, 30(8):49–55, August 1997.
9. J. Bosch. *Design & Use of Software Architectures: Adopting and evolving a product-line approach*. Addison-Wesley, Harlow, England, 2000.
10. J.L. Diaz-Herrera. Domain engineering. In *Handbook of Software Engineering and Knowledge Engineering*, volume I. World Scientific Pub., 2001.
11. R.N. Taylor, W.J. Tracz, and L. Coglianese. Software development using domain-specific software architectures. *ACM SIGSOFT Software Engineering Notes*, 20(5):27–38, December 1995.
12. G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language User Guide*. Object Technology Series. Addison-Wesley, Reading, MA, 1999.
13. R. Kazman. Software architecture. In *Handbook of Software Engineering and Knowledge Engineering*, volume I. World Scientific Pub., 2001.
14. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, Reading, MA, 1998.
15. M. Shaw and D. Garlan. *Software architecture: perspectives on an emerging discipline*. Prentice Hall, 1996.
16. L. Barroca, J. Hall, and P. Hall, editors. *Software Architectures: Advances and Applications*. Springer-Verlag, London, 2000.
17. S. Yacoub, H. Ammar, and A. Mili. Characterizing a software component. In *Proc. Second International Workshop on Component-Based Software Engineering*, Los Angeles, CA, May 1999. (available at http://www.sei.cmu.edu/cbs/icse99/).
18. W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, Wokingham, England, 1995.
19. M.E. Fayad and D.C. Schmidt. Object-oriented application frameworks. *Communi-*

cations of the ACM, 40(10):32–38, October 1997.

20. S. Srinivasan. Design patters in object-oriented frameworks. *IEEE Computer*, 32(2):24–32, February 1999.
21. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA, 1995.
22. T.J. Mowbray and R. Zahavi. *The Essential CORBA: Systems Integration Using Distributed Objects*. Wiley, New York, 1995.
23. M.L. Brodie and M. Stonebraker. *Migrating Legacy Systems: Gateways, Interfaces and the Incremental Approch*. Morgan Kaufmann, San Francisco, CA, 1995.
24. W. Hasselbring. Information system integration. *Communications of the ACM*, 43(6):32–38, June 2000.
25. A. Sheth and J. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, 1990.
26. M.T. Roth and P.M. Schwarz. Don't scrap it, wrap it! A wrapper architecture for legacy data sources. In *Proc. 23rd International Conference on Very Large Data Bases (VLDB'97)*, pages 266–275. Morgan Kaufmann, 1997.
27. D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.
28. R. Munz. Usage scenarios of DBMS. Keynote Address at VLDB'99, September 1999. (available at `http://www.dcs.napier.ac.uk/~vldb99/IndustrialSpeakerSlides/`).
29. W. Hasselbring. The role of standards for interoperating information systems. In K. Jakobs, editor, *Information Technology Standards and Standardization: A Global Perspective*, pages 116–130. Idea Group Publishing, 2000.
30. S. McGrath. *XML by example: building e-commerce applications*. Prentice Hall, Upper Saddle River, NJ, 1998.
31. W. Hasselbring and H. Weigand. Languages for electronic business communication: State of the art. *Industrial Management & Data Systems*, 2000. (in press).
32. D. Coppit and K.J. Sullivan. Multiple mass-market applications as components. In *Proc. 22th International Conference on Software Engineering*, pages 273–282, Limerick, Ireland, June 2000. IEEE Computer Society Press.
33. K. Seetharaman. The CORBA Connection. *Communications of the ACM*, 41(10):34–36, October 1998.
34. R. Monson-Haefel. *Enterprise JavaBeans*. O'Reilly, 2nd edition, 2000.
35. G. Eddon. COM+: The evolution of component services. *IEEE Computer*, 32(7):104–106, July 1999.
36. W. Emmerich, E. Ellmer, and H. Fieglein. Tigra — An architectural style for enterprise application integration. In *Proc. 23th International Conference on Software Engineering*, Toronto, Canada, May 2001. IEEE Computer Society Press.
37. W. Hasselbring. Top-down vs. bottom-up engineering of federated information systems. In *Engineering Federated Information Systems (Proc. EFIS'99)*, pages 131–138. Infix-Verlag, 1999.
38. R. Prieto-Diaz. Implementing faceted classification for software reuse. *Communications of the ACM*, 34(5):88–97, 1991.
39. B. Chandrasekaran, J.R. Josephson, and V.R. Benjamins. What are ontologies, and why do we need them? *IEEE Intelligent Systems*, 14(1):20–26, 1999.
40. M.D. McIlroy. Mass produced software components. In P. Naur and B. Randell, editors, *Proc. NATO Conference on Software Engineering*, pages 88–98, Garmisch, Germany, October 1968.
41. E.-A. Karlsson, editor. *Software Reuse: A Holistic Approach*. John Wiley & Sons, 1995.

42. D. Garlan, R. Allen, and L. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, November 1995.