Basic Concepts and Terms

1 Software Development in the Large

1.1 Software Reuse

Industrial development of software systems, often called *software development in the large*, generally needs to be guided by engineering principles similar to those in mature engineering disciplines. Informal methods, which might be appropriate for the development of simple software, cannot be employed for the development of software systems with high inherent complexity. This is one of the lessons learnt from the software crisis. The software crisis lead to the creation of the term *software engineering* to make clear that software development is an engineering discipline. A characteristic of an engineering discipline is the use of systematic, disciplined, and quantifiable methods [194]. One such a method in mature engineering disciplines is that of *reuse*: the development of a new product is based, particularly for cost-effectiveness, on prefabricated components which are tried and tested in previous projects.

Reuse can increase cost-effectiveness of software projects. Considering reusable software units as assets, cost-effectiveness can be increased by using these assets in a number of projects and sharing their development costs. Improved cost-effectiveness, however, is only one of the possible benefits motivating reuse in software projects. According to [369], other possible benefits are:

> *Increased reliability.* Reuse is supposed to have a positive effect on the quality of the software entity reused. A frequently reused software entity is expected to improve in quality and, in particular, to become more reliable, since frequent reuse is expected to reveal failures and other adverse behaviors which maybe otherwise would not be revealed when the entity is developed and tested. Even if this effect can be observed for some software entities, it cannot be generally expected, as shown in the next chapter.

Reduced process risk. Software reuse can reduce risks inherent in software projects. For instance, the cost of reusing existing software entities can usually be estimated with less uncertainty than that of developing the same software entities from scratch.

Effective use of specialists. Software reuse can also contribute to an effective use of specialists. Often, specialists are assigned within one project to development tasks that need also to be conducted within other projects. Reusable software entities can encapsulate the knowledge and experience of specialists and be reused within many projects.

Standards compliance. Reusing software in the development of a software system can also improve its standards compliance, which in turn can have other benefits, such as improved quality. For instance, usability of a software system can be increased by reusing software entities implementing a user interface that users are already familiar with.

Accelerated development. Finally, software reuse can accelerate development. A software project can be completed in less time, and development time can be saved using prefabricated software entities. The reused entities do not have to be developed from scratch, which obviously would take more development time and effort than reusing existing ones.

The potential benefits of software reuse outlined above depend on several factors. One of these factors is the size of the reused software entities in relation to the size of the system to be developed. Risks inherent in software projects, for instance, can be significantly decreased when reusing large software units. Cost estimation in the development of large software systems is more complex and thus associated with more uncertainty than in the development of small software units, whereas reuse costs estimation in both cases requires similar effort and is associated with comparable risk. Reuse of software can be classified into three categories , according to the size of the reused software entities [369]:

Application system reuse. The first category is application system reuse, which can be encountered mainly in two distinct forms. The subject of reuse can be either a complete, unchanged software system or an application family, also called *product line*, which can be adapted to certain platforms and needs.

Component reuse. The second category is component reuse. The term *component* again refers to arbitrary constituents of a system in this context. Single entities of a software system can be reused for the development of a new software system. The reused entities can range in size from single classes to whole subsystems.

Function reuse. Finally, the third category is function reuse. In contrast with application system reuse at one end of the spectrum, software entities consisting of single functions can also be subject for reuse. Mathematical

functions, for instance, have been successfully used in such a form for decades [78].

Another factor determining the extent to which software development can benefit is through the reuse of artifacts other than binary software entities. Artifacts, such as the design of a software system, can also be reused from former projects instead of being reproduced from scratch. In the case of design reuse, the design of software can be based on *design patterns* [135] which represent abstract solutions to problems encountered often in the design phase.

1.2 Abstraction Principle

Besides reuse, another method for managing complexity in software development in the large is abstraction. The abstraction principle aims at separating the essential from the non essential. In the context of software development, the essential generally refers to business logic, whereas the non essential refers to technical details. According to [158], a problem often encountered in software projects is that the development of business logic is dominated by technical details. Development of technical details can dominate the development of business logic in three distinct ways:

Firstly, a significant amount of effort is often spent on technical issues which do not contribute to the functionality of the software system. Such a technical issue is, for instance, transactions handling.

Secondly, software is often overloaded with technical details, which hinders analysis of the implemented business logic, for instance for reengineering purposes, and thus complicates its further development and maintenance.

Thirdly, separation of business logic and technical details allows their development and maintenance to be conducted by separate specialists who can contribute to quality improvements.

Various levels of abstraction can be identified in software development. Figure 1 shows some of the typical levels [158]:

Binary values. At the lowest level of abstraction are binary values, which are grouped into binary words. At the lowest level of abstraction, instructions and data are represented by binary words. For instance, the binary word 000100101010 can be an instruction to add two values.

Assembler languages. Assembler languages are at the next level of abstraction. In assembler languages, each instruction is represented by a statement instead of a binary word. For instance, adding two values can be represented as the statement `ADD A,B` instead of the binary word 000100101010.

High-level languages. High-level programming languages are at another abstraction level. High-level programming languages provide complex

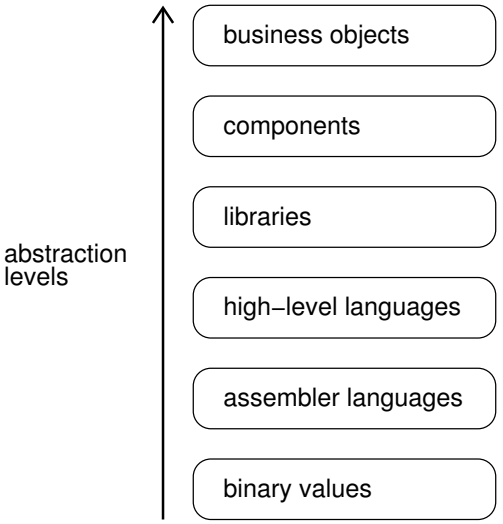

Fig. 1. Typical abstraction levels in software development

statements which are not mapped to single instructions of the underlying machine. At this level of abstraction, the underlying machine is usually completely transparent to the developer, who can thereby concentrate on the functionality of the software system to be implemented.

Libraries. Above the abstraction level of high-level programming languages are the libraries. Libraries contain solutions, either binary or source code, to problems often encountered during development. The developer of a certain system therefore does not need to implement such solutions from scratch and can instead focus on the functionality of the software system. A library can, for instance, provide solutions for data storage in the form of appropriate data structures.

Components. Components can be considered as forming the next level of abstraction. They further separate business logic from technical details.

Business objects. Finally, business objects can be found at the top level of abstraction. A business object is developed according to a business concept. Business objects represent entities in real-world such as employees, products, invoices, or payments.

2 Components as Building Blocks

In recent years, components and component-based development received much attention due to the explicit support for software reuse and the abstraction principle, two methods of software development in the large. A large num-

ber of scientific articles have been published on the subjects of components and component-based development and several component models have been released for development of component-based systems. The subjects of components and component-based development, however, are still not mature, as definitions of basic terms, such as that of a component, did not converge. The various definitions are not discussed here. According to [385], it is assumed that:

Definition 1. *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.*

A similarity between component-based and object-oriented development is the distinction between components and classes, respectively, and their instances. A component is a static description of its instances, while a class is a static description of corresponding objects. Component instances are sometimes assumed to be stateless, such as in [385], which has various consequences. One of the important consequences is that instances of a component cannot differ with respect to their output and behavior, which depend solely on the input. A component-based system thus needs to maintain a single instance of a component, which does not mean that it cannot be multithreaded. In practice, however, instances are often implemented statefully and the instances of a component are distinguished from each other with regard to their state. In some applications, it might be necessary to make an instance persistent, for example, by storing its state in a database. Such an instance exists beyond the termination of the creating process and can be identified by its state.

The type of a component is often defined in terms of the interface implemented by that component. The component type names all operations of the corresponding interface, the number and types of parameters, and the types of values returned by each operation [385]. As a component can implement several interfaces, it can also be of several types. The operations of a component are accessed through its interfaces and are implemented by its methods. Figure 2 shows a meta-model of the basic terms [158]. Note that the meta-model in Fig. 2 also indicates that a component can be implemented according to the object-oriented paradigm, which is, however, not an obligation [385].

In [158], a list of technical properties is given characterizing a component. Some of these technical properties are a direct implication of Definition 1. This list, however, also includes technical properties that do not follow from the definition but are nevertheless explained for completeness. A component is characterized by the following technical properties:

Well-defined purpose. A component needs to have a well-defined purpose. The purpose of a component is typically more comprehensive and more abstract than that of a class. A component, however, does not usually implement the entire functionality of an application, and thus needs to be embedded in an application context.

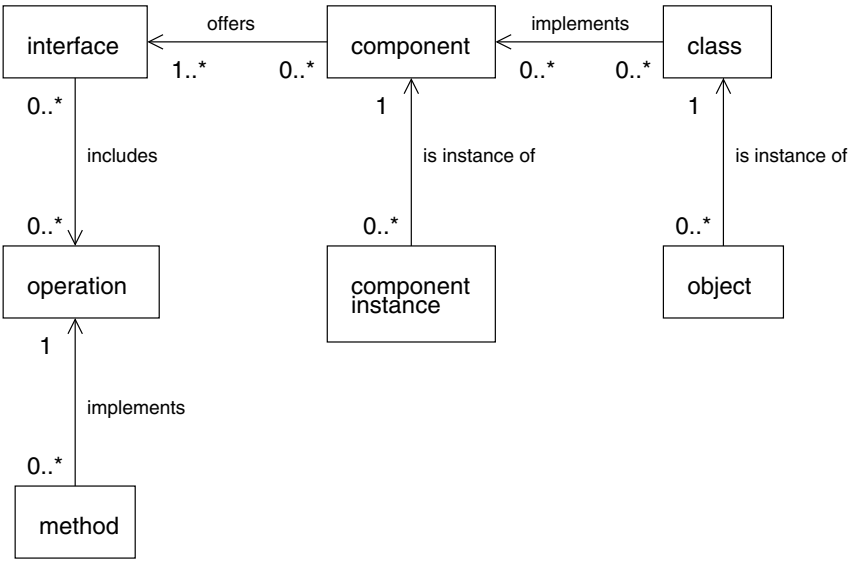

Fig. 2. Meta-model of basic terms

Context independence. A further technical property of a component is its context independence. Even though a component needs to be embedded in an application context, it should remain separated from its technical issues. Separation from technical issues of the application context concerns, for instance, issues of communication.

Portability and programming language independence. Portability and programming language independence means that a component can be implemented in an arbitrary programming language as long as the resulting binary code is compatible with the intended technical environment. Consequently, a component does not necessarily have to be implemented using object-oriented languages. Instead, a component can also be implemented using procedural or functional languages [385].

Location transparency. An important property, at least in the case of distributed applications, is that of location transparency. The quality of services provided by a component should be independent of the component's technical environment. Specifically, aspects of the component's physical location, such as whether a component is executing as a separate process or on a separate host, should be transparent to clients.

Separation of interface and implementation. The property of separation of interface and implementation requires that a component interact with its clients solely through its interfaces and that its implementation be encapsulated. The implementation of a component should be transparent to clients, which should not be able to access internal features of the

component. Separation of interface and implementation leads to explicit dependencies between a component and its context, and this in turn simplifies its reuse.

Introspection. Introspection refers to the ability of a component to provide clients meta-information describing its interfaces and functionality. In contrast to *reflection*, which also allows clients to obtain meta-information, the meta-information is usually assembled by the component developer and can thus be focused on certain composition-related aspects [151]. The meta-information provided can be explored by clients at runtime, allowing thereby, for instance, dynamic reconfiguration of a component-based system.

Plug-and-play. Furthermore, a component should possess plug-and-play capability. Installation of a component should be automatic as far as possible, and supported by the component itself. Installation tasks which can be automated are, for instance, registration of the component in naming services.

Integration and composition. A component should support integration and composition as typical properties. Component integration and composition can be distinguished in *static* and *dynamic*, where static refers to integration and composition during the development of a component-based system, and dynamic to those during runtime. A component should support both types of integration and composition.

Reuse. A component needs to support techniques allowing its reuse. One of the two methods for software development in the large is reuse as explained before. Usability of a component in software development in the large thus strongly depends on its support for techniques allowing its reuse, particularly in the application context for which it was developed.

Configuration. Closely related to the property of reuse is that of configuration. Configuration of a component can be necessary for its reuse in the development of a new system. Thus, a component needs to support configuration techniques, particularly those based on parameterization.

Approval. One of the possible benefits of reuse is that of increased reliability and thereby approval. Frequently reused software units are expected to be more reliable than newly developed ones, as they have been tried and tested in other projects. It was thus expected that reuse would make testing obsolete. Contrary to expectations, however, reuse does not generally obviate testing, as explained in the next chapter.

Binary form. Finally, components are often available in binary form only. The developer of a component often does not disclose the source code and other detailed information in order to protect intellectual property. Furthermore, disclosing such information can give competitors advantages

over the developer, which is obviously undesirable, and not only in the case of commercial components.

The technical properties illustrate the close relation between components and software development in the large. On the one hand, use of components pays off in terms of cost and time savings only if the system to be developed possesses a certain critical size, since the use of components involves a certain overhead. On the other hand, software development in large makes use of two specific methods: reuse and abstraction. Both methods are supported by components, as the above explanation of technical properties shows.

3 Component Models and Frameworks

3.1 Basic Concepts

Software development in the large in general and component-based development in particular typically involve several parties, something which requires technical standards for cooperation. In the context of component-based development, such standards and conventions are specified by component models. As with other terms in this context, the term *component model* is not defined uniquely. According to [14], it is assumed here that:

Definition 2. *A* component model *is the set of component types, their interfaces, and, additionally, a specification of the allowable patterns of interaction among component types.*

A component model can require that components of that model implement certain interfaces, and can thereby impose certain component types. Furthermore, a component model can define how components have to interact with each other in the form of allowed interaction patterns that can cover aspects such as which component types can be clients of which other component types.

The standards and conventions specified by a component model are necessary for component-based development due to the following reasons [14]:

Uniform composition. Information describing the required and provided services of each component is a necessary condition for correct interaction of two components. This information usually needs to address aspects such as the functionality of each component, how each component is located, how control flow is synchronized, which communication protocol is used, and how data is encoded.

Appropriate quality attributes. The quality of a component-based system obviously depends on the quality of its constituent individual components. The component model can specify interaction patterns necessary to set those characteristics to certain values and states that affect the quality of the component-based system to achieve a certain system quality.

Deployment of components and applications. Besides composition of components, another critical success factor for component-based development is deployment of individual components and the resulting component-based system. Specifically, standards and conventions need to be specified allowing deployment of individual components into the development environment and deployment of the resulting system into the end user environment.

The standards and conventions specified by a component model are technically supported by a *component framework* implementing the necessary technical infrastructure and providing the necessary services. A component framework thus establishes the environmental conditions for the components following a certain component model. Typical services implemented by a component framework are the following [14]:

Management of resources shared by components. One of the main services implemented by a component framework is the management of resources. For instance, component frameworks often provide load balancing services for distributing available processing capacity among components according to certain scheduling criteria.

Communication among components. Another important service provided by a component framework is facilitating communication among components. Usually, the service provided allows component communication without requiring close coupling of components, something which encapsulates technical details within components.

Management of component life cycle. A service implemented, for instance, by component frameworks for the Enterprise JavaBeans component model is management of the component life cycle. A component framework can directly work with a component and change the states of its instances according to certain criteria.

A component framework can be implemented according to various paradigms. It can be implemented as an autonomous system existing at runtime independently from the components supported, such as the component frameworks for the Enterprise JavaBeans component model. Such a component framework can be regarded as an operating system with a special purpose. A component framework, however, can also be implemented as a non-autonomous system which can be instantiated by components. An example of a non-autonomous component framework can be found in [233]. As a third alternative, a component framework can also itself be implemented as a component, which allows the composition of component frameworks [385]. In such an approach, a high order component framework might regulate the interaction of other component frameworks and might provide them with the necessary technical services. Such high order component frameworks, however, are currently not available.

3.2 Enterprise JavaBeans Component Model and Framework

One of the component models widely used in practice is the one following the Enterprise JavaBeans specification released by Sun Microsystems. The Enterprise JavaBeans specification defines the *Enterprise JavaBeans architecture* for the development of distributed, component-based client/server systems. The initial specification of the Enterprise JavaBeans was released in 1998. Since then, the specification was developed further, and several extensions were made to the initial release. In this subsection, the Enterprise JavaBeans component model and framework are explained with respect to release 2.1 of the specification, which can be found in [92]. Other examples of component models are the CORBA Component Model proposed by the Object Management Group [312] and the Component Object Model and related technologies proposed by Microsoft Corporation [283].

Component Types

One of the component types defined by the Enterprise JavaBeans specification is the *session bean*. The primary purpose of a session bean is to encapsulate business logic. A session bean is usually non-persistent, i.e., its data is not stored in a database and is lost after the session is terminated. Furthermore, a session bean cannot be accessed simultaneously by multiple clients. A session bean provides its services through specific interfaces, depending on the type of client and service. Clients that can access a session bean can be either local or remote, and the services provided can be classified as being related either to the life cycle of a particular session bean or to the business logic implemented. In this context, the main distinction between a local and a remote client is that local clients execute in the same Java virtual machine, and that the location of the session bean is not transparent for local clients, whereas a remote client can execute in a different Java virtual machine and thus on a different host, so the location of the session bean is transparent.

Figure 3 shows two session beans embedded in the component framework as specified by the Enterprise JavaBeans specification, which defines a component framework consisting of *servers* and *containers* to support the component model. A server is generally responsible for providing technical services such as persistence, transactions, and security, whereas a container provides services related to management of the component life cycle. An enterprise bean is encapsulated in a container, which is in turn embedded in a server. The Enterprise JavaBeans specification, however, does not clearly separate containers from servers, and the terms are often used interchangeably. The component framework consisting of a server and a container is transparent to the user of an enterprise bean. Initially, a client intending to access the business logic implemented by a certain session bean needs to create a reference to it. The appropriate methods can be invoked through the *home interface* of the session bean, which in Fig. 3 is called `EJBLocalHome` and `EJBHome` for the local and the remote session bean, respectively. Note that, technically, a

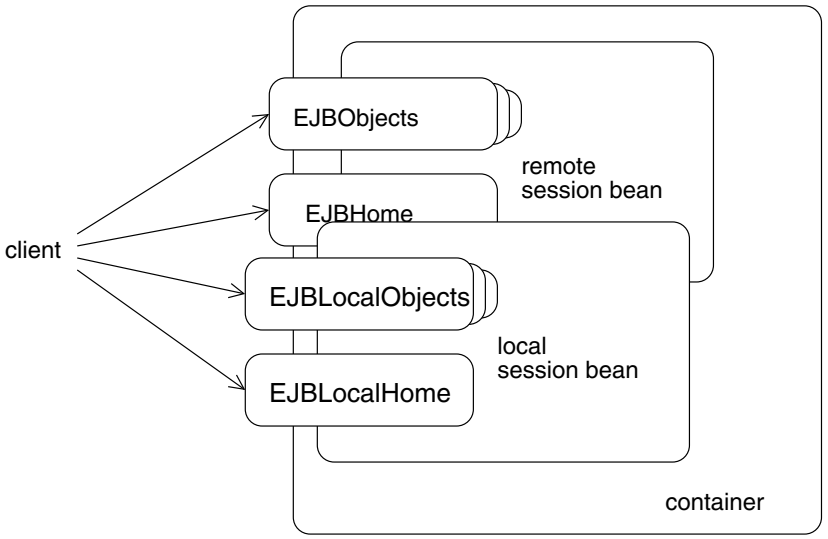

Fig. 3. Client view of session beans deployed in a container

session bean can implement both types of interfaces; however, this is rarely the case. Having obtained a reference to the session bean, methods implementing the business logic can be invoked through the *local interface* and the *remote interface*. In Fig. 3, these interfaces are called `EJBLocalObjects` and `EJBObjects`.

The second component type defined by the Enterprise JavaBeans specification is the *entity bean*. An entity bean represents a business object. In contrast to session beans, an entity bean can be persistent, i.e. data encapsulated in an entity bean can be stored in a database and can thus exist beyond the termination of the component-based system and the component framework. Database entries are mapped to entity beans through *primary keys*, which are unique identifiers of entity beans. Another distinction to session beans is that an entity bean can be accessed by several clients. As several clients might intend to modify data simultaneously, transaction techniques might be required to ensure the consistency of the data. Such techniques, however, are implemented by the server hosting the container of the corresponding entity bean, and do not need to be implemented by the entity bean itself. An entity bean provides similar interfaces as a session bean; thus, Fig. 3 is also valid, with minor changes, for entity beans.

One of the new features in release 2.0 of the Enterprise JavaBeans specification is a third component type called *message-driven bean*. Its main differences with the other component types is that it allows asynchronous computations: a message-driven bean is invoked by the container when a messages arrives. A direct consequence is that a message-driven bean is not accessed through interfaces similar to session and entity beans. Instead, a client intending to access

a message-driven bean needs to generate a message. A message-driven bean is usually stateless and thus does not need to be persistent. Consequently, individual message-driven beans usually cannot be distinguished from each other and unique identifiers are not required, as opposed to entity beans. However, a message-driven bean can process messages from several beans; thus, it is similar to an entity bean in this respect. The capability to serve several clients simultaneously necessitates transaction techniques. Such underlying techniques, however, are generally provided by the component framework, specifically, by the server.

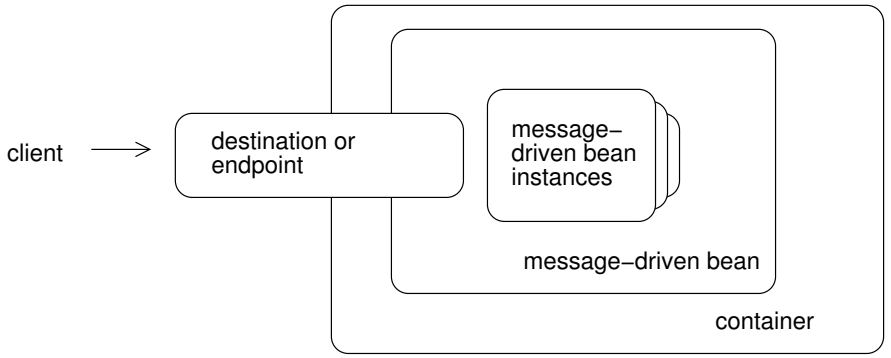

Fig. 4. Client view of message-driven beans deployed in a container

Figure 4 shows a message-driven bean embedded in the component framework. As in the case with the other two component types, the component framework consists of servers and containers. They have the same objectives, namely providing basic technical services in the case of the server and managing the life cycle of the enterprise bean, i.e., the message-driven bean.

Roles in Component-based Development

The Enterprise JavaBeans specification defines six roles that are responsible for the various tasks in the life cycle of a component-based system. The specification particularly defines contracts between the various roles, for instance, the products to be delivered by one role to another. In the Enterprise JavaBeans specification, these roles are defined as follows :

Enterprise bean provider. The enterprise bean provider is responsible for the development of beans. Tasks which particularly need to be conducted by the enterprise bean provider are the definition of the bean interfaces, development of classes implementing business logic, and composition of the bean deployment descriptor making external dependencies explicit.

Application assembler. The application assembler is responsible for packaging EJB components to larger deployable entities. The task assigned to

the application assembler is to generate files containing EJB components and information, stored in the corresponding deployment descriptors, on various technical aspects of the EJB components.

Deployer. The deployer is responsible for the deployment of the files produced by the application assembler into a specific operational environment. The operational environment is in such a case a specific component framework, i.e., an Enterprise JavaBeans server and container. The deployer needs to take into account the external dependencies of each bean, specified in the corresponding deployment descriptor by the bean provider, and also application assembly information provided by the application assembler.

EJB server provider. The main responsibility of the EJB server provider is to deliver the EJB server in which EJB containers can be embedded. The server implements technical services such as transaction management and encapsulates the implementation of the technical services from the container.

EJB container provider. In a manner to the EJB server provider, the EJB container provider is responsible for delivering the EJB container in which single beans can be embedded. Furthermore, the EJB container provider is also in charge of developing tools necessary for beans deployment and for providing runtime support for deployed beans.

System administrator. Finally, the system administrator is responsible for the configuration and administration of the operational environment including the component framework. The system administrator is usually also in charge of monitoring the deployed EJB components to avoid problems at runtime.

The Enterprise JavaBeans component model and frameworks explicitly support reuse and abstraction, two methods of software development in the large. Software reuse, on the one hand, is facilitated, for instance, by the deployment descriptor associated with each EJB component specifying its context dependencies. The abstraction principle, on the other hand, is facilitated, for instance, by the roles defined in the Enterprise JavaBeans specification. Specifically, the development of business logic is assigned to a single role, and corresponding tasks have to be carried out solely by that role. Technical and administrative tasks are assigned to other roles. The definition of the six roles contributes to an abstraction of business logic development from technical and administrative tasks during component-based development. An abstraction of business logic from technical details in the architecture of a component-based system is achieved by the component framework. An EJB component solely implements the intended business logic, whereas techniques necessary, for instance, for distribution and persistence are implemented by the component framework.

4 Commercial-off-the-Shelf (COTS) Components

In comparison to other reusable software entities, one of the distinguishing features of components, particularly those referred to as *commercial-off-the-shelf* (COTS) components, are their market-related aspects. According to [20], a COTS component has the following characteristics:

> *The buyer has no access to the source code,*
>
> *the vendor controls its development,*
>
> *and it has a nontrivial installed base.*

The term *COTS component*, however, is not uniquely defined, as a discussion of the various definitions in [291] shows, and the above characteristics do not exist to the same degree for each commercially available component. Among other factors, the existence of the above characteristics depends on the organizational relationship between the vendor of the component and its buyer. The organizational relations between the component provider and component user can be manifold, and a component can be associated with one of following categories [58]:

> *Independent commercial item.* A component can be an independent commercial item that can be purchased from a possibly anonymous component market. This category of components is referred to as *COTS components*.
>
> *Special version of commercial item.* A component can also be a special version of a commercial component. The component user might contract with the component provider to produce a customized version of a commercially available component.
>
> *Component produced by contract.* Depending on the organizational relation, a component can also be produced by contract. The component user can ask the component provider to develop a component for an agreed-upon fee.
>
> *Existing component from external sources.* A component can originate from an external source without being commercial. The component provider and component user might reside in different organizations, but the component could be one developed under a joint contract.
>
> *Component produced in-house.* Finally, a component can also be developed for a specific project. The component provider and component user might be involved in the same project or the roles of the component provider and component user can even be played by the same person.

This book focuses on COTS components. If not otherwise noted, components are assumed to have the above characteristics of COTS components, and the two terms *component* and *COTS component* are used interchangeably.

Context of the Book

1 Lack of information in development of and with components

1.1 Misperceptions in quality assurance of components

It is often argued that the quality of components will improve under certain conditions and less quality assurance will be required in the development of a component-based system to satisfy given quality requirements. The conditions under which the quality of components will improve according to this argument include [385]:

> *Frequent reuse.* Reuse is generally supposed to have a positive effect on the quality of the software entity reused. A frequently reused component is expected to improve in quality, since frequent reuse is expected to reveal failures and other adverse behavior which possibly would not be revealed when, instead of being reused, the component would be redeveloped and tested from scratch.
>
> *Competitive markets.* Competitive markets are also supposed to contribute to improvements of component quality, since quality is expected to become a success factor in such markets.

One of the implications of this argument described in the literature is that quality assurance actions are considered less important in component-based development than in software development in general. Components are supposed to possess a certain degree of quality obsoleting further quality assurance actions, and component-based systems are expected to inherit the quality of their individual constituents. However, these arguments do not take into account the following:

> Firstly, the quality of an entity, in this context a component, is defined according to common definitions, such as that in [200], with regard to

stated or implied needs. In this context, these needs are those of a particular component user. The needs of one component user might contradict those of another, and might also change after some time. Thus, even if the quality of a component might be sufficient according to the needs of a particular component user, additional quality assurance action might nevertheless be necessary prior to its use either by the same component user due to a change in requirements or by another component user due to a difference in the needs.

Secondly, competitive markets do not necessarily relieve the component user from conducting quality assurance actions. The needs of the component users might not be entirely known to the component provider so that the step taken by the component provider in order to increase quality might not be successful. Such problems are particularly caused by a limited exchange of information between the component provider and component user.

Limited exchange of information between the roles of the component provider and component user does not only limit the positive effect of competitive markets on component quality. It also relates several other issues in developing components and component-based systems and can be considered the primary factor distinguishing testing components from testing software in general.

1.2 Examples of Information Exchanged

The component provider and component user generally need to exchange information during the various phases of developing a component and a component-based system. The development of a component, if the component is not component-based and is thus not itself a component-based system, usually consists of the typical phases of software development. Software development usually comprises the phases *requirements analysis and definition*, *system and software design*, *implementation and unit testing*, *integration and system testing*, and *operation and maintenance* if it is conducted according to the waterfall model or a derivative of it [369]. The single phases might be named differently depending on the actual software process model; however, that does not affect the following explanations. During some of the phases, the component provider needs to exchange information with the component user. Such phases, for instance, are:

Requirements analysis and definition. The requirements analysis and definition phase obviously necessitates information concerning the capabilities and conditions the component needs to satisfy according to the component user's expectations.

Operation and maintenance. The operation and maintenance phase necessitates information enabling the component user to work with the component and information required for its maintenance by the component provider.

Even though a waterfall model-based software process has been assumed so far, similar patterns of information exchange can also be identified for other software process models and the results obtained also apply to them.

Information flow between component provider and component user does not occur only during the development of a component. Information often also needs to be exchanged during the development of a component-based system using the component. Even though the following explanations assume a concrete process model for component-based development, as described in [369], they are also valid for other process models for component-based development, since the phases in which information is exchanged between the two roles usually also have their counterparts in those models. The process model for reuse-oriented software development with components includes six phases, as described in [369]. These phases are *requirements specification*, *component analysis*, *requirements modification*, *system design with reuse*, *development and integration*, and *system validation*. During some of these phases, information flow between the component provider and component user can be observed. Examples of these phases are:

Component analysis. The component analysis phase requires information supporting identification of components available from the various sources, their analysis with respect to certain criteria, and finally selection of the component most suitable for the component-based system to be developed.

Development and integration. The phase of development and integration can also require technical information that the component user needs to obtain from the component provider. Such technical information might concern the interfaces of the component or the required middleware.

System validation. Information often needs to be exchanged between the two roles in the system validation phase. Such an exchange of information might concern program-based test cases generated by the component provider or meta-information supporting the component user in testing.

The above list of phases in component-based development that require exchange of information between the component provider and component user is not necessarily comprehensive. Other process models of component-based development might include other phases that also require information flow between the two roles. For instance, the process model proposed in [289, 290] for COTS-based software development defines an activity called *write glueware and interfaces* within the coding phase. This activity encompasses the development of auxiliary code necessary for integrating the various components, requiring detailed technical information which the component user might need from the component provider. However, the aim of the above list is only to show that interaction between the two roles takes place throughout the life cycles of components and component-based systems, and the flow of information is not merely one way [289, 290]. Furthermore, the various phases

of component-based development generally also encompass information exchange with roles other than the two mentioned above, such as with the end user of the component-based system. While the previous explanations focused only on communication between the component provider and component user and omitted other types of communication, it does not mean that they do not exist.

1.3 Organization Relation as an Obstacle for Information Exchange

Various factors impact the exchange of information between the component provider and the component user. The information requested by one role and delivered by the other can differ in various ways, if it is delivered at all. It can differ syntactically insofar that it is, for instance, delivered in the wrong representation, and it can also differ semantically in that it is, for instance, not on the abstraction level required. The differences might be due to various factors, one of them being the organizational relation between the two roles. With respect to the organizational relation, a component can be associated with one of the following categories: *independent commercial item*, *special version of commercial item*, *component produced by contract*, *existing component from external sources*, and *component produced in-house*. Information exchange between the component provider and the component user depends, among other factors, on the organizational relationship between them.

At one end of the spectrum, the component can be a commercial item. In this case, the quality of information exchange between component provider and component user is then often the worst in comparison to the other cases. There are various reasons for this, such as the fact that the component provider might not know the component user due to an anonymous market. In such a case, the component provider can base development of the component on assumptions and deliver only that information to the component user which is supposedly needed. Furthermore, the component might be used by several component users and the component provider might decide to consider only the needs of the majority. The specific needs of a single component user might then be ignored. Finally, the component provider might not disclose detailed technical information even if needed by the component user to prevent another component provider from receiving this information. The component provider might decide to make only that information available which respects intellectual property and retains a competitive advantage.

At the other end of the spectrum, the component can be produced in-house. The quality of information exchange between component provider and component user is then often the best in comparison to the other cases. One of the reasons for this can be the fact that the component is developed in the same project in which it is assembled. The exchange of information in both directions, from the component provider to the component user and the reverse, can take place without any incompatibility in the requested and

delivered information. Furthermore, the component provider and component user are roles, so they can even be played by the same person if the component is used in the same project in which it is developed. Information exchange would not even be necessary in that case.

1.4 Problems Due to a Lack of Information

According to [169, 170], quality assurance actions, particularly testing, that are applied to a component can be viewed from two distinct perspectives: *Component provider perspective* and *component user perspective*. Quality assurance of a component usually needs to be conducted by both the component provider and the component user. The corresponding tasks, however, differ insofar as the component provider generally needs to conduct them independently from the application context, whereas the component user of a component can concentrate on a certain application context while carrying out such tasks. The two distinct views on quality assurance of components underline this difference.

Context-Dependent Testing of a Component

One type of information required for the development of a component is an indication of the application context in which it will be used later. Such information, however, might not be available, so the component provider might develop the component on the basis of assumptions concerning the application context. The component is then explicitly designed and developed for the needs of the assumed application context, which, however, might not be the one in which it will actually be used. Even if the component is not tailored to a certain application context, but constructed for the broader market, the component provider might assume a certain application context and its development might again become context-dependent. A consequence of context-dependent development of a component can be that testing is also conducted with context dependency. A component might work well in a certain application context while exhibiting failures in another [406, 423].

The results of a case study given in [423] show the problem of context-dependent testing in practice. A component was considered a part of two different component-based systems. The component-based systems mainly provided the same functionality, but differed in the operations profiles associated with them. The operational profile of a system is a probability distribution which assigns a probability to each element in the input domain, giving the likelihood that this element is entered as input during operation, e.g., [131]. A set of test cases was generated for the first component-based system with a 98% confidence interval. The probability that an arbitrary input is tested was 98%, and both the component-based system as well as the component in its context were considered to be adequately tested. However, the fact that

a component is adequately tested in the context of one system does not generally imply that it is also adequately tested in the context of another. In the case study, the set of test cases corresponded only to a 24% confidence interval of the second system's operation profile, so occurrence of an untested input during the second system's operation was much more likely.

Observations such as those made during the case study are captured by a formal model of test case adequacy given in [350], an extension of that in [422], encompassing components and component-based systems. The formal model of test suite adequacy also considers that a component tested with a certain test suite might be sufficiently, i.e., adequately, tested in a certain application context, but not adequately covered by the same test suite in another application context. In the terminology introduced, the test suite is *C-adequate-on-*P_1, with C being the test criterion used to measure test suite adequacy and P_1 being the former application context, but not *C-adequate-on-*P_2, with P_2 being the latter application context. An exact definition of the formal framework and theoretical investigations on test suite adequacy in case of components and component-based software can be found in [350].

One of the reasons for context-dependent development of a component is often the component provider's lack of information concerning the possible application contexts in which the component might be used later. Tests conducted by the component provider might also be context-dependent, so a change of application context, which might be due to reuse of the component, generally requires additional tests in order to give sufficient confidence that instances of the component will behave as intended in the new context. Additional tests are required despite contrary claims, such as in [385], that frequently reused components need less testing. Moreover, a component that is reused in a new application context needs to be tested irrespective of its source. A component produced in-house does not necessarily need less testing for reuse than a component that is an independent commercial item [423].

Insufficient Documentation of a Component

Development of a component-based system generally requires detailed documentation of the components that are to be assembled. Such documentation is usually delivered together with the respective components, and each of them needs to include three types of information:

Functionality. The specification of the component functionality describes the functions of that component, i.e., its objectives and characteristic actions, to support a user in solving a problem or achieving an objective.

Quality. The specification of component quality can address, for instance, the quality assurance actions applied, particularly testing techniques, and the metrics used to measure quality characteristics and their values.

Technical requirements. The specification of the technical requirements of a component needs to address issues such as the resources required, the architectural style assumed, and the middleware used.

Documentation delivered together with a component and supposed to include the above specifications might, however, be insufficient for development of a component-based system. The various types of information provided by the documentation can deviate from those expected both syntactically and semantically, and may even be incomplete. This problem can be viewed from two different perspectives. On the one hand, it can be considered a problem due to lack of information. The component provider might be lacking information and might therefore not provide the information that is actually needed by the component user in the form of documentation. On the other hand, it can be considered a reification of a lack of information. Instead of assuming that the component provider is lacking information while developing the component and assembling its documentation, the component user is assumed to be lacking information while developing a component-based system using the component. According to the latter perspective, insufficient documentation is not the effect of a lack of information but its reification. However, the subtle differences of these perspectives are not further explored here.

A case study found in [139] reports several problems encountered during the integration of four components into a component-based system. The problems encountered are assumed to be caused by assumptions made during the development of the individual components. The assumptions, even those concerning the same technical aspects of the components and the component-based system, were incompatible with each other, which was the reason for what the authors call an *architectural mismatch*. As one of the solutions to tackle the architectural mismatch problem, the authors propose to make architectural assumptions explicit. These assumptions need to be documented using the appropriate terminology and structuring techniques. Another solution to problems such as those in the case study is proposed in [371]. Here, the author suggests prototyping during component-based development in order to detect potential problems.

In [289, 290], the authors propose a process model for COTS-based development that includes a specific activity to tackle problems due to insufficient documentation. The process model encompasses an activity called *COTS components familiarization*, in which components selected earlier are used to gain a better understanding of their functionality, quality, and architectural assumptions. The importance of such an activity obviously depends on the quality of the component documentation already available.

Both prototyping and familiarization require that the component under consideration is executed, which is also the main intrinsic property of testing. In fact, both can be considered testing if the term *testing* is defined more generally. The objectives of both are not necessarily related to quality assurance, but aim to obtain information which is not delivered as part of the documen-

tation. Furthermore, components delivered with insufficient documentation might also require testing in its original sense, particularly if the documentation does not include information concerning the quality assurance actions taken.

Component User's Dependence on the Component Provider

Context-dependent development and insufficient documentation of a component are two problems resulting from a lack of information that often obligate the component user to test a component before its use and the system in which the component is embedded. The component user can encounter other problems after tests are finished, particularly if the tests revealed failures. The problem which the component user can encounter is dependence on the component provider. A fault often cannot be removed by the component user, since the component user might not have the software artifacts required for isolating and removing the fault. Such artifacts include documentation, test plans, and source code of the component. Even if the required artifacts are available to the component user, debugging might be difficult or even impossible due to lack of expertise. Lack of expertise and insight of the component user might entail significant debugging costs, which can offset the benefits gained by using the component. Thus, the component user often has to rely on the component provider for maintenance and support, which the component user might not be able to influence. This can obviously entail risks in the life cycle of the developed component-based system.

The problem of dependence on the component provider can be aggravated if the component is not maintained as demanded by the component user, or if the component provider decides to discontinue support and maintenance or goes bankrupt [406, 423]. The possible financial effects of such an event is shown in [406] using a simple example. It was suggested that escrow agreements and protective licensing options be considered for the relevant artifacts of a component in order to avoid these problems. Even if the component provider accepts such an agreement, missing expertise can still hinder the component user from carrying out the corresponding tasks.

Difficulties faced by the component user because of a dependence on the component provider are not necessarily restricted to maintenance and support. Generally, several of the decisions taken by the component provider during the life cycle of the component also impact its use as part of a component-based system. Other problems which can occur due to dependence on the component provider can be [289, 290, 406]:

- Deadlines might be missed because of delays in the release of a component version,
- functionality promised might never be implemented in the component,
- modifications might have adverse effects, such as incompatibilities or even faults,

- some functionality of the component might be altered inconsistently,
- documentation might be incomplete or might not sufficiently cover modifications,
- technical support might not be sufficient.

As with context-dependent component development, the problem of component user dependence on the provider varies with the quality of information exchanged. The more the information about the component available to its user, the less dependent the user on the provider, as some maintenance and support tasks can be carried out by the user. Specifically, dependence on the component provider affects reputation of the component user. In the case of a problem, the component user's reputation will suffer even if the problem is caused by a component for which not the component user, but the provider, is responsible [423].

2 Issues in Testing Components and Component-based Systems

2.1 Objective of Testing

The intrinsic property of testing is execution of the software under test. The basic procedure of testing is to execute the software to be validated and to compare the behavior and output observed with that expected according to the specification. Although the basic procedure is clear, a common definition of the term *testing* does not exist. Several definitions can be found in the literature, all emphasizing different aspects. The definition assumed in the following is the one according to [194]:

Definition 3. [Testing *is*] *The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component.*

Testing for Quality Assurance

Even though Def. 3 does not explicitly state its objective, testing is an action of quality assurance. A definition of the general term *quality* can be found, for instance, in [200]:

Definition 4. [Quality *is*] *The totality of characteristics of an entity that bear on its ability to satisfy stated and implied needs.*

In our case, the entity referred to in the above definition is software, a component, or a component-based system, together with its documentation and data. The characteristics determining quality of software products are further refined in [208]. The quality model described is not restricted to a specific

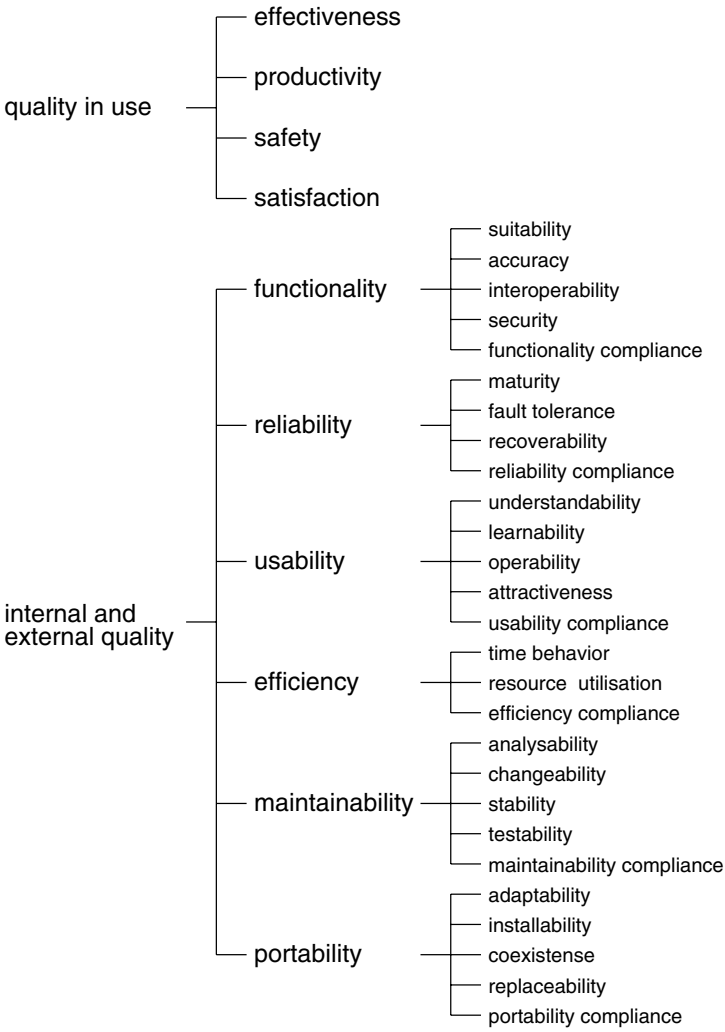

Fig. 5. A two-part quality model for software

type of software. It applies rather to software in general, to components, and to component-based systems. The general software quality model is defined according to [208] a model consisting of two parts, also shown in Fig. 5:

Internal and external quality. Internal and external quality consider software quality from the perspective of the developer. Internal quality consists of characteristics which can be observed in an internal view of the software and can thus be measured by static properties of the code, such as path length. In contrast with internal quality, external quality consists of characteristics which are observable in an external view of the software

and can be measured by dynamic properties, such as response time, during runtime of the software.

Quality in use. Quality in use considers software quality from the perspective of the user. Quality in use consists of characteristics, such as productivity, describing the extent to which software supports users in achieving their goals in a certain application context. Quality in use thus assumes an external view of software.

Specification and evaluation of software quality, for instance by testing, require that the various characteristics defining the internal and external quality and quality in use be measurable. To allow measurability, a characteristic is either further decomposed into *attributes*, which are measurable using metrics, or its measure is induced based on the measures of other characteristics. In the latter case, a characteristic used to induce the measure of another is called *indicator*. A detailed explanation of the various characteristics constituting software quality, together with the corresponding metrics, can be found in [208]. A software quality evaluation process based on the quality model described is given in [202–207]. A COTS components-specific quality model can be found in [26].

An example of a quality assurance action is that of testing. The assessment by testing typically includes executing the software, observing and measuring a characteristic or subcharacteristic using specific metrics, and comparing the observed values with those expected. Testing, however, can assess only some of the quality characteristics of the quality model introduced, mainly due to the fact that the others cannot be measured by executing the software considered. The characteristics of the quality in use model part can generally not be measured by testing, as these characteristics are subjective and depend on the application context. Similarly, the characteristics of maintainability and portability, both affecting internal and external quality, can generally also not be validated by testing, since, for instance, maintainability depends particularly on the availability of design documents which obviously cannot be assessed by an execution of the corresponding software [346]. However, efficiency of software, according to [346], can be validated by testing only if corresponding targets are specified. Similarly, usability of software can be tested only if corresponding targets are given, such as ergonomic guidelines. Functionality and reliability, however, can be validated thoroughly by testing, as explained briefly as follows [346]:

Functionality. The quality characteristic of functionality is further decomposed into the subcharacteristics of suitability, accuracy, interoperability, security, and functionality compliance, as shown in Fig. 5. Each of these subcharacteristics can be assessed by testing. For instance, suitability of software can generally be shown in the context of system testing, accuracy by determining deviations between observed and expected results, and interoperability in the context of integration testing.

Reliability. The quality characteristic of reliability is also further decomposed into subcharacteristics, as indicated in Fig. 5. These characteristics maturity, fault tolerance, recoverability, and reliability compliance can also be assessed by testing. For instance, maturity can be tested or measured by counting the number of failures within a certain time period, fault tolerance by observing the behavior in the case of failures, recoverability by measuring the effort required to recover after failures. In all of these cases, the failures which might occur and the behavior which is expected in the case of a failure need to be specified beforehand.

Testing for COTS Components Evaluation

In component-based development, testing can also have an objective other than that of quality assurance. It can aim at reducing the risks inherent in software development, which can originate from inaccurate or incomplete information concerning the individual components used. Even if reuse as one of the strategies of software development in the large, supported by component-based development, has the potential of decreasing risks inherent in software development, the use of components might introduce new risks. The process of risk management in software development encompasses several stages according to [369]:

Risk identification. The stage of risk identification includes listing the possible risks affecting the project, product, and business.

Risk analysis. The stage of risk analysis encompasses assessment of the likelihood and consequences of the risks identified.

Risk planning. The stage of risk planning involves identifying the means by which the risks identified can be tackled.

Risk monitoring. The stage of risk monitoring consists of assessing the risks and revising plans for risk mitigation as more information becomes available.

One of the difficulties the component user needs to tackle when using components is lack of information. The information which the component user needs in the risk identification and risk analysis stages of risk management can be inaccurate or incomplete. The component user might therefore not be able to appropriately carry out the successor stages. To avoid this problem, the component to be used can be evaluated by building a prototype system, as in explorative prototyping [129]. The evaluation of a component mainly aims to gain information not delivered together with the component by the component provider and to confirm the validity of the available information. Other reasons can be [90, 295] to

- identify undocumented features, e.g., hidden interfaces,
- confirm or deny the published information and specifications,

- determine how well a component fits within the system environment,
- determine possibility to mask out unwanted features and faults, e.g., with wrappers,
- ensure that unwanted features and faults do not interfere with the system's operations.

According to [61], the testing techniques often used for the purpose of evaluation are binary reverse engineering and interface probing. The former is a technique enabling the component user to derive the design structure of a component from its binary code, i.e., the component as delivered, without source code. The latter allows the component user to gain insight into functionality and limitations of a component which might not be explicitly documented.

2.2 Test Case Generation

Test case generation in the context of testing a component or a component-based system generally needs to follow the same principles as test case generation in general. One of them is to avoid input inconsistencies. A principle of test case generation is that test cases have to comprise information that avoids input inconsistencies, which means that tests carried out using the same test case must always result in the same output and behavior to ensure reproduceability. However, the information that needs to be included in a test case might be difficult to determine, particularly in the context of testing components, due to various reasons.

Firstly, the relation between input and output might not be explicit. A method of a component might compute its output using not only its arguments, but also other data, something which might not be known to the component user. The component user can suffer from a lack of information insofar that specification might be incomplete and source code not available, so the component user might not be able to determine exactly the information which must be part of the test cases.

Secondly, the behavior of a component instance and its computation of output can also depend on factors that are external to the component. For example, behavior of a component instance is often affected by the component framework in which it is embedded. The component user might not know how the component framework can impact component instance behavior, and might not be able to identify the information required to control it.

Test case generation has to be conducted, except possibly in the case of very simple software, with regard to an adequacy criterion, which can be considered another principle. An *adequacy criterion* gives for a set of test cases the degree of adequacy of that test case set, depending on the software under test and its specification [440]. A set of test cases can be considered to be adequate, and testing on the basis of this test case set to be sufficient, if the adequacy of the

test case set exceeds a predefined value. However, some problems related to adequacy criteria can be encountered in test case generation for components and component-based systems that hinder their application.

Firstly, one of the classifications of adequacy criteria provided distinguishes them with regard to the source of the information used to measure adequacy. The component user does not generally have full access to all types of information regarding a component. As a specific example, source code is often not available, and program-based adequacy criteria, which require such type of information, can generally not be computed. The problems due to a lack of information are not necessarily restricted to this category of adequacy criteria. The component user might encounter problems when trying to compute specification- or interface-based adequacy criteria if the respective information, such as unspecified functionality, is incomplete or even wrong or inaccurate, such as an operational profile assigning wrong probabilities to certain inputs.

Secondly, even with full access to the necessary information, the component user can still encounter problems in computing adequacy criteria due to their limited scalability, particularly when testing component-based systems [142]. Even if the testing of a component can in certain circumstances be considered as unit testing, adequacy criteria generally used in unit testing can only be employed to a limited extent. The adequacy criteria often used in unit testing, such as program-based and structural adequacy criteria, suffer from a lack of scalability. Such adequacy criteria can necessitate a large number of test cases for adequate testing, particularly if the methods of a component interact with each other and share global variables in the context of a component-based system.

2.3 Test Execution and Evaluation

Test execution, which is often also called testing in the narrow sense, and test evaluation are in principle conducted in the same way as for software in general. However, when testing components and systems consisting of components, difficulties can be encountered which do not exist in such a form when testing software in general. Such a difficulty is caused by the fact that some factors affecting a component's behavior might not be controllable by the tester. In this context, *controllability* of a factor refers to the tester's ability to set it to a specific value or ensuring that it satisfies a specific condition. Specifically, component-external factors, such as databases in distributed environments, might not be controllable by the tester due to access permissions, for instance.

As with testing of software in general, the output and behavior of a component instance or a system need to be observed and recorded together with all other relevant data describing the effects of the test execution. Generally, one possible way of obtaining the output and behavior is in the form

of a trace. A *trace* is a record of the execution showing the sequence of instructions executed, the names and values of variables, or both [194]. The execution of a component can be observed and traces can be built using three basic approaches [136–138]:

Framework-based tracing. The capability necessary for producing traces can be added to a component by integrating it with a framework which implements the corresponding functionality. The integration of a component with a framework can obviously be conducted only by its developer, i.e., the component provider, since source code access is mandatory for such a task.

Automatic code insertion. The above integration of the component with the framework providing the necessary functionality can be automated by using automatic code insertion techniques. Such techniques can generally be used to automatically extend the source code of the component with arbitrary statements to support trace generation. Obviously, this approach can also only be conducted by the component provider due to the necessity of source code.

Automatic component wrapping. The externally visible effects of an execution can also be observed by embedding the component into a wrapper which is capable of producing the desired traces. Externally visible effects are, for instance, the returned output and the interactions with other components. This approach considers the component as a black-box insofar that information concerning internals of the component are not required. Thus, this approach to generating traces can also be conducted by the component user.

A comparison of the three approaches to generating traces can be found in [136–138]. Generally, the component user has very limited possibilities of observing the execution of the tests if appropriate provisions were not taken by the component provider, since several of the approaches to producing traces require source code access. Even if the components used in the context of a system provide capabilities to generate traces, the traces generated might be incompatible with each other syntactically and semantically, and missing configuration possibilities can hinder the removal of such incompatibilities.

The testing of software in general and of components and component-based systems in particular is usually continued after test execution, with an evaluation of the observations. For each test conducted, the observed output is validated by means of a test oracle. The problems which need to be tackled in this context for software in general, i.e., the oracle problem, may also be encountered when evaluating the observation of component tests. Another problem in the evaluation of component test observations can be that of *fault propagation* [288]. Faulty software does not necessarily exhibit a failure even if a faulty section is executed. A failure becomes visible, i.e., is identified by a test oracle based on the observations, only if a faulty section is executed, some

variables are assigned wrong values (known as *infection*), and wrong values are propagated to an output. This problem can in principle also occur when testing software in general, but the problem that a fault is not propagated to an output can be harder to identify when testing components. The reason is that factors which hinder fault propagation might not be known to the component user due to missing insight and a lack of relevant information, such as source code.