

Introduction

Component-based software development and software testing are two subdisciplines of what today is generally understood as software engineering. Software engineering is a discipline that attempts to bring software development activities, and testing is part of that, more in line with the traditional engineering disciplines such as civil engineering, mechanical engineering, or electrical engineering. The main goal of software engineering is to come up with standard ways of doing things, standard techniques and methods, and standard tools that have a measurable effect on the primary dimensions that all engineering disciplines address: cost and quality. Software engineering is still in its relative infancy compared with the more traditional engineering disciplines, but it is on its way.

Component-based software development directly addresses the cost dimension, in that it tries to regard software construction more in terms of the traditional engineering disciplines in which the assembly of systems from readily available prefabricated parts is the norm. This is in contrast with the traditional way of developing software in which most parts are custom-designed from scratch. There are many motivations for why people allocate more and more effort toward introducing and applying component-based software construction technologies. Some will expect increased return on investment, because the development costs of components are amortized over many uses. Others will put forward increased productivity as an argument, because software reuse through assembly and interfacing enables the construction of larger and more complex systems in shorter development cycles than would otherwise be feasible. In addition, increased software quality is a major expectation that people are looking for under this subject. These are all valid anticipations, and to some extent they have yet to be assessed and evaluated. Heineman and Councill [89] present a nice collection of articles that address these issues, and many of the discussions in this field are about the economics of software components and component-based software construction. However, this is not the subject of this book.

Testing directly addresses the other dimension, quality, and it has turned out that it must be dealt with somewhat differently in component-based software engineering compared with traditional development projects. This is because component-based systems are different and the stakeholders in component-based development projects are different, and they have contrasting points of view. How component-based systems are different and how they must be treated in order to address the challenges of testing component-based systems are the main subjects of this book. In the following sections we will have a look at the basics of component-based software engineering and what makes testing so special in this discipline, and we will look at the role that the Unified Modeling Language (UML), which emerges as a major accepted standard in software engineering, plays under this subject.

1.1 Component-Based Software Development

1.1.1 Component Definition

The fundamental building block of component-based software development is a component. On first thought it seems quite clear what a software component is supposed to be: it is a building block. But, on a second thought, by looking at the many different contemporary component technologies, and how they treat the term component, this initial clarity can easily give way to confusion. People have come up with quite a number of diverse definitions for the term component, and the following one is my personal favorite:

A component is a reusable unit of composition with explicitly specified provided and required interfaces and quality attributes, that denotes a single abstraction and can be composed without modification.

This is based on the well-known definition of the 1996 European Conference on Object-Oriented Programming [157], that defines a component in the following way:

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

In this book I chose a somewhat broader terminology that avoids being independently deployable, since I am not specifically restricting the term component to contemporary component technologies such as CORBA, .NET, or COM, which provide independent execution environments. In this respect, I see the term component closer to Booch's definition, or the definition of the OMG:

A component is a logically cohesive, loosely coupled module [20].

A component is a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces [118].

But there are many other definitions that all focus on more or less similar properties of a component, for example:

A software component is an independently deliverable piece of functionality providing access to its services through interfaces [24].

A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard [89].

From these definitions it becomes apparent that components are basically built on the same fundamental principles as object technology. If we consider an object-oriented language as the only deployment environment, we can also say that objects are independently deployable within this environment. The principles of encapsulation, modularity, and unique identities that the component definitions put forward are all basic object-oriented principles that are subsumed by the component paradigm [6]. However, the single most important feature of a component, in my opinion, is that it may be reused in a number of different contexts that have initially not been anticipated by the component producer. Only the component user decides whether it may be fit for a particular purpose, and this, in fact, is the single most important distinguishing factor of component-based development with respect to the traditional approaches. From the component definitions, we can actually derive a number of important properties for software components according to [24, 66, 89, 158]:

- Composability is the primary property of software components as the term implies, and it can be applied recursively: components make up components, which make up components, and so on.
- Reusability is the second key concept in component-based software development. Development for reuse on the one hand is concerned with how components are designed and developed by a component provider. Development with reuse on the other hand is concerned with how such existing components may be integrated into a customer's component framework.
- Having a unique identity requires that a component should be uniquely identifiable within its development environment as well as its runtime environment.
- Modularity and encapsulation refers to the scoping property of a component as an assembly of services that are related through common data. Modularity is not defined through similar functionality as is the case under the traditional development paradigms (i.e., module as entity with functional cohesion), but through access to the same data (i.e., data cohesion).

- Interaction through interface contracts; encapsulation and information hiding require an access mechanism to the internal workings of a component. Interfaces are the only means for accessing the services of a component, and these are based on mutual agreements on how to use the services, that is, on a contract.

In the following paragraphs we will take a closer look at the concepts that the component definitions put forward.

1.1.2 Core Principles of Component-Based Development

Component Composition

Components are reusable units for composition. This statement captures the very fundamental concept of component-based development, that an application is made up and composed of a number of individual parts, and that these parts are specifically designed for integration in a number of different applications. It also captures the idea that one component may be part of another component [6], or part of a sub-system or system, both of which represent components in their own right. As a graphical representation, composition maps components into trees with one component as the root of the parts from which it is composed, as shown in Fig. 1.1.

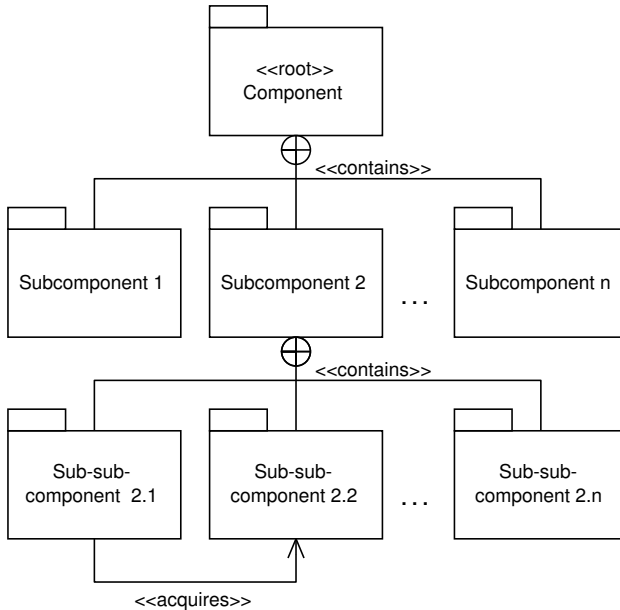


Fig. 1.1. Composition represented by a component nesting tree, or a so-called component containment hierarchy

Component Clientship

An additional important concept that is related to component composition and assembly is clientship. It is borrowed from object technology and subsumed by the component concept. Clientship or client/server relationship is a much more fundamental concept than component composition. It represents the basic form of interaction between two objects in object-oriented systems. Without clientship, there is no concept of composition and no interaction between components. Clientship means that a client object invokes the public operations of an associated server object. Such an interaction is unidirectional, so that the client instance has knowledge of the server instance, typically through some reference value, but the server instance needs no knowledge of the client instance. A clientship relation defines a contract between the client and the server. A contract determines the services that the server promises to provide to the client if the client promises to use the server in its expected way. If one of the two parties fails to deliver the promised properties, it breaks the contract, and the relation fails. This typically leads to an error in the clientship association. A composite is usually the client of its parts. A graphical representation of clientship forms arbitrary graphs, since clientship is not dependent on composition. This is indicated through the `«acquires»`-relationship in Fig. 1.1. It means that **Sub-subcomponent 2.1** acquires the services of **Sub-subcomponent 2.2**, thereby establishing the client/server relationship. Clientship between contained components in a containment hierarchy is represented by the anchor symbols in Fig. 1.1. The minimal contract between two such entities is that the client, or the containing component, at least needs to invoke the constructor of its servers, or the contained components.

Component Interfaces

A component's syntax and semantics are determined through its provided and required interfaces. The provided interface is a collection of functionality and behavior that collectively define the services that a component provides to its associated clients. It may be seen as the entry point for controlling the component, and it determines what a component can do. The required interface is a collection of functionality and behavior that the component expects to get from its environment to support its own implementation. Without correct support from its servers at its required interface, the component cannot guarantee correct support of its clients at its provided interface. If we look at a component from the point of view of its provided interface, it takes the role of a server. If we look at it from the point of view of its required interface, the component takes the role of a client. Provided and required interfaces define a component's provided and required contracts. These concepts are illustrated in Fig. 1.2.

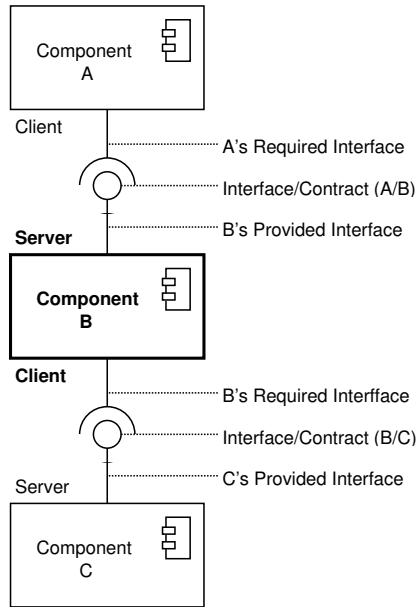


Fig. 1.2. UML-style representation of components with provided and required interfaces

Quality Attributes

Quality attributes have the same meaning for the non-functional aspects of a component that interfaces have for the functional and behavioral aspects of the component. Quality attributes define additional requirements of components, such as dependability and performance.

Quality Documentation

The documentation can be seen as part of a component's specification, or a refinement of its specification. Sometimes, a pure specification may be too abstract, so that it is difficult for users to see and understand how a component's operations may be called or can be applied. It is particularly useful in order to document how sequences and combinations of operation invocations add to the overall behavior. A documentation provides a deeper insight into how a component may be used in typical contexts and for typical usage profiles that the provider of the component had anticipated.

Persistent Component State

An additional requirement of Szyperski's component definition [157] that is often cited in the literature is that a component may not have a persistent

state. It means that whenever a component is integrated in a new application, it is not supposed to have any distinct internal variable settings that result from previous operation invocations by clients of another context. This requires that a runtime component, a so-called component instance, will always be created and initialized before it is used. However, this is not practical for highly dynamic component systems such as Web services, which may be assembled and composed of already existing component instances that are acquired during runtime. It is not a fact that because components have persistent states they cannot be integrated into a running system. It is a fact that they must have a well-defined and expected persistent state so that they can be incorporated into a running application. The fact that a component may already have a state must be defined a priori, and it is therefore a fundamental part of the underlying clientship relation between two components. The invocation of a constructor operation, for example, represents a transition into the initial state of a component. This is also a well-defined situation that the client must know about in order to cooperate with the component correctly. In this respect, it may also be seen as a persistent state that the client must be aware of.

1.1.3 Component Meta-model

The previous paragraphs have briefly described the basic properties of a component and component-based development. The following paragraphs summarize the items that make up a component and draw a more complete picture of component concepts. I also introduce the notion of a UML component meta-model, which will be extended over the course of this book, to illustrate the relations between these concepts.

Figure 1.3 summarizes the concepts of a component and their relations in the form of a UML meta-model. It is a meta-model, a model of a model, because it does not represent or describe a physical component but only the concepts from which physical components are composed. The diagram defines a component as having at most one provided interface and one required interface. These two interfaces entirely distinguish this component from any other particular component. The provided interface represents everything that the component is providing to its environment (its clients) in terms of services, and the required interface represents everything that the component expects to get from its environment in order to offer its services. This expectation is represented by the other associated (sub-)components that the subject component depends upon, or by the underlying runtime environment.

Provided and required interfaces must be public, as indicated through the UML stereotype `<<public>>`. Otherwise we cannot sensibly integrate the component into an application, because we do not know how the component will be connected with other components in its environment. Provided and required interfaces are also referred to as export and import interfaces.

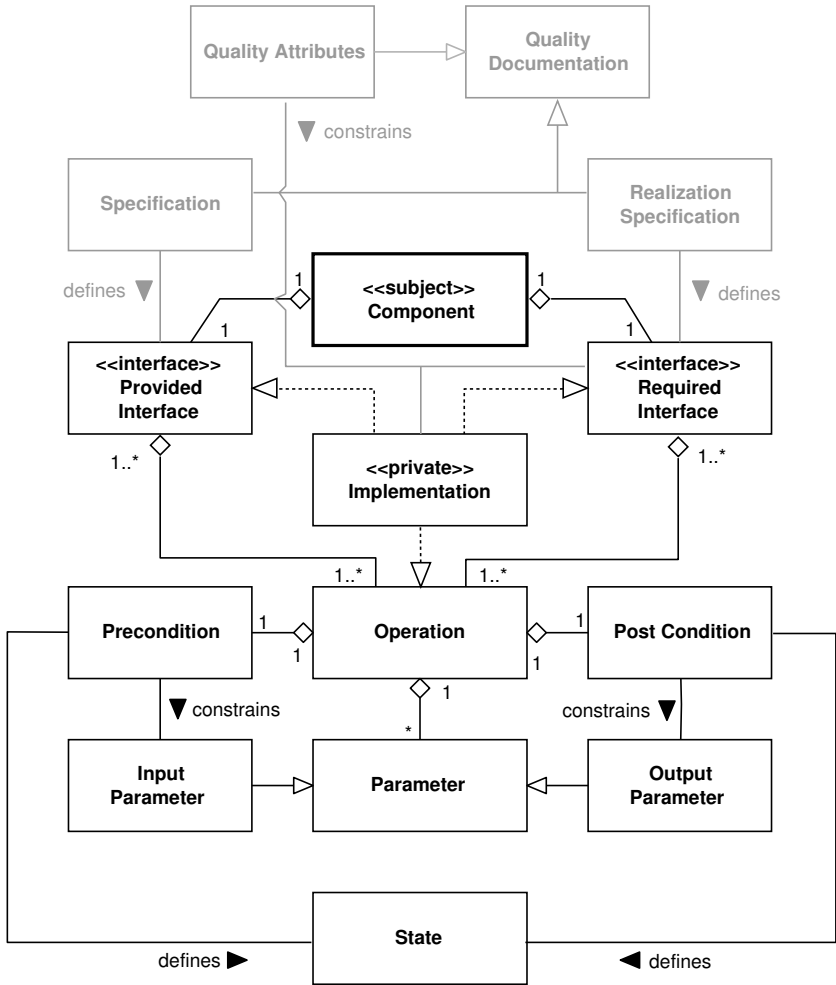


Fig. 1.3. Component meta-model

The implementation of a component realizes its private design. This is the collection of algorithms by which a component achieves its functionality, its internal attributes, internal and external operations, and operation calls to its associated (sub-)components. The implementation is hidden inside the encapsulating shell of the component, and is arbitrarily exchangeable through any other implementation that realizes the same external features.

Provided and required interfaces comprise operations. The operations in the provided interface are access points that the client of the component can use to control its functionality, and the operations in the required interface are the access points to other associated components that the subject component

depends on or to the underlying runtime system. The functionality of an operation depends on pre and postconditions. A precondition is an invariant that must be true or fulfilled in order for the operation to guarantee the postcondition. A precondition constrains the input parameters of an operation and defines an initial state of the component that must be valid before the operation may be invoked. A postcondition constrains the output parameters of an operation to the guaranteed values and defines a final state of the component that becomes valid after operation invocation.

The combination of precondition, operation invocation with input parameters, and postcondition, with output parameters, represents a transition from one state to another. A state is a distinct combination of a component's internal attribute values that are constantly changed through the operations. These attributes may be owned by the component, or by any subordinate component that in itself may be seen as an attribute of the superordinate component. States are not directly visible outside a component's encapsulation boundary because they are determined through internal attributes. A state manifests itself through differing observable external behavior of a component only if the same operation is invoked; an operation may not be invoked at all under the conditions of a certain state. For example, the `storeItem()` operation of a storage component behaves differently if the storage is already completely used up. In this case the component won't store any more items.

So far we have looked only at the parts of a physical component in Fig. 1.3. A physical component is an executable binary version of the component. But a component is not really usable if it does not come with some additional description or documentation, which can be seen as the logical part of the component. Additional descriptive artifacts are the specification and realization of the component that represent its quality documentation and a specification of the quality attributes. A specification of a component is a collection of descriptive documents that define what a component can do. It comprises descriptions of the externally provided and required operations with their behavior and pre and postconditions and exceptions. A realization defines how a component implements its functionality, e.g., in terms of interaction with other subordinate components and additional algorithmic information. The quality documentation defines quality attributes that the component is abiding with, plus the quality attributes that it expects from its associated subordinate components. The quality attributes constrain the internal implementation of the component as well as the required interface. This means that a component can only accept an associated server component as long as it provides not only the expected function and behavior but, additionally, the expected quality features.

1.1.4 Component Engineering vs. Application Engineering

Component-based software construction may be subdivided into two distinct development activities: component engineering and application engineering.

While component engineering is concerned with the development of components as individual building blocks in component-based software construction, application engineering is concerned with the assembly and integration of these building blocks into new software applications. Component engineering is primarily performed by the provider of a component, and application engineering is mainly carried out by the component user. I have referred to these activities above respectively as development for reuse and development with reuse.

In its purest form, component-based development is only concerned with the second item, development with reuse (component integration), representing a bottom-up approach to system construction. This requires that every single part of the overall application is already available in a component repository in a form that exactly maps to the requirements of that application. Typically, this is not the case, and merely assembling readily available parts into a configuration will likely lead to a system that is not consistent with its original requirements.

Because the shape of the building blocks that are assembled and integrated during application engineering is so essential to component engineering, development for reuse (component engineering) is an important activity that needs also to be considered in the context of component-based development. After all, component engineering is initially responsible for how suitably and easily components can be composed and reused.

But there is another important dimension to the activities of component engineering and application engineering. Application engineering deals with how a system is decomposed into finer-grained parts that are individually controllable. Component engineering deals with how individual components will fit into the logical decomposition hierarchy that application engineering has come up with. Both are interdependent, and component-based development methods must deal with these two orthogonal approaches.

Component-based development is usually a mixture of top-down decomposition and bottom-up composition. In other words, the system is decomposed into finer-grained parts, that is, subsystems or components, and these are to be mapped to individual prefabricated building blocks. If no suitable components are found, decomposition is continued. If partially suitable components are found, the decomposition is repeated according to the needs of the candidate component. A found suitable component represents a feasible and acceptable solution for the entire system or the subsystem considered. The whole process is iterative and must be followed until all requirements are mapped to corresponding components or until the system is fully decomposed into the lowest desirable level of abstraction. If suitable third-party components are found, they can be composed to make up the system or subsystem under consideration. Such a process is always goal-oriented in that it accepts only components that are fit for the purpose of the system. It means that only these parts will be selected that somehow map to the system specification. The

outcome of such a development process is usually a heterogeneous assembly consisting of combinations of prefabricated parts plus implementations.

All these considerations fall into the scope of component-based development methods that provide guidelines on what, when, and how such activities have to be carried out during component-based software construction. The next chapter (Chap. 2, “Component-Based and Model-Driven Development with UML”) is devoted entirely to how component-based development should ideally be done in the context of a development method that follows the ideas of model-driven software construction. Development methods provide the framework for applying modern software engineering principles, and they are also responsible for integrating dynamic quality assurance techniques, the main subject of this volume.

1.2 Component-Based Software Testing

Component-based software development has been and still is considered to be the primary technology to overcome the software crisis [23, 89, 159]. Its main idea is to build new software products by reusing readily available parts, rather than by developing everything from scratch. The expected savings in product development are based on the assumption that software reuse has a much higher return on investment than pure software development. This is certainly true for product development, because parts of one system whose investment has already been amortized and written off are used again in another system without any extra cost. However, this is not entirely true. The assumption becomes a reality, and a component-based development project becomes a success, only if the costs of adapting all components to their new environment, and of integrating them in their new context is much lower than the whole system being developed from scratch. The cost of assembling and integrating a new product also includes the assuring of its quality.

The expectation that software developers and software development organizations place in component-based software engineering is founded on the assumption that the effort involved in integrating readily available components at deployment time is less than that involved in developing code from scratch and validating the resulting application through traditional techniques. However, this does not take into account the fact that when an otherwise fault-free component is integrated into a system of other components, it may fail to function as expected. This is because the other components to which it has been connected are intended for a different purpose, have a different usage profile, or may be faulty. Current component technologies can help to verify the syntactic compatibility of interconnected components (i.e., components that they use and that provide the right signatures), but they do little to ensure that applications function correctly when they are assembled from independently developed components. In other words, they do nothing to check the semantic compatibility of interconnected components so that the individual

parts are assembled into meaningful configurations. Software developers may therefore be forced to perform more integration and acceptance testing to attain the same level of confidence in the system's reliability. In short, although traditional development-time verification and validation techniques can help assure the quality of individual units, they can do little to assure the quality of applications that are assembled from them at deployment time.

In the previous two paragraphs, I have already presented some ideas on the challenges that we have to consider and address in component-based software testing. In the following subsection we will have a closer look at the typical problems that emerge from this subject.

1.2.1 Challenges in Component-Based Software Testing

The software engineering community has acknowledged the fact that the validation of component-based systems is different from testing traditionally developed systems. This manifests itself in the number of publications in that field, in people's interest in the respective forums and workshops on the topic, and, in the production of this book; references include [65, 67, 77, 175, 176]. Component-based software testing refers to all activities that are related to testing and test development in the scope of a component-based development project. This comprises tests and testing activities during component engineering, carried out by the provider of a component, as well as all testing activities during application engineering, carried out by the user of the component. Here, it is important to note that the two roles, component provider and user, may be represented by the same organization, i.e., in case of in-house components that are reused in a number of diverse component-based development projects.

A test that the provider performs will typically check the internal workings of a component according to the provider's specification. It can be seen as a development-time test in which the individual parts, classes, or subcomponents of the tested component are integrated and their mutual interactions are assessed. It concentrates on assessing the component's internal logic, data, and program structure, as well as the correct performance of its internal algorithms. Since the provider assumes full internal knowledge of the component, it can be tested according to typical white box testing criteria [11]. This component test will likely concentrate on the individual services that the component provides and not so much on the combination or sequence of the services. The aim of the component provider is to deliver a high-quality reusable product that abides by its functional and behavioral specification.

A second test that the user of the component will typically carry out is to assess whether the module fits well into the framework of other components of an application. This sees a component as a black box entity, so typical black box testing techniques [12] will be readily applied here. The aim of the component user is to integrate an existing component into the component

framework, and assess whether both cooperate correctly according to the requirements of the user. For his or her own developments the user will assume the role of the producer and carry out typical producer tests. This is the case, for example, for adapter components that combine and integrate different third-party developments and enable their mutual interactions. Since the user assumes white box knowledge of these parts of an application they can also be tested based on typical white box testing criteria.

The following list summarizes the most important problems that people believe make component-based software testing difficult or challenging [66]:

- Testing of a component in a new context. Components are developed in a development context by the provider and reused in a different deployment context by the user. Because the provider can never anticipate every single usage scenario in which the component will be deployed by a prospective user, an initial component test can only assess the component according to a particular usage profile, and that is the one that the provider can think of. Components are often reused in contexts that the provider could have never imagined before, thus leading to entirely different testing criteria for the two stakeholders [175, 176]. I believe this is the primary challenge in component-based software testing, and the most important subject of Chap. 4, “Built-in Contract Testing”, and the entire book is more or less devoted to this problem.
- Lack of access to the internal workings of a component [86, 175], which reduces controllability and testability. Low visibility of something that will be tested is a problem. Testing always assumes information additional to what a pure user of an entity is expecting. For most components, commercial ones in particular, we will not get any insight except for what the component’s interface is providing. One part of the technology that I introduce in this book is devoted to increasing controllability and observability and, as a consequence, testability of components. This is the second most important subject of Chap. 4, “Built-In Contract Testing.”
- Adequate component testing [142]. This is concerned with the questions of which type of testing, and how much testing a component provider should perform, and which type of and how much testing a component user must perform to be confident that a component will work. Unfortunately, I have no answer to this, and this book concentrates only on which testing criteria may be applied in a model-based development approach. This is the subject of Chap. 3, “Model-Based Testing with the UML.” Defining adequate test sets (in particular), and adequate testing (in general) are still challenging subjects, and this represents one of the most significant problems in testing. There is not much empirical evidence of which testing criterion should be applied under which circumstances.

I believe the most fundamental difference between traditional testing and the testing of component-based systems lies in the different view points of the stakeholders of a software component, the provider and the user. This identi-

fies the fundamental difference in the validation of traditional software systems and component-based developments. While in traditional system development all parts are integrated only according to a single usage profile, in component-based development they are integrated according to differing usage profiles that depend on their integrating contexts. In the first instance, system developers can integrate two parts of a system according to a single well-defined and known interface. Even if the two parts do not match, they can be adapted to their respective contexts and fully validated in their particular predetermined association. In the second instance however, this is not feasible. Neither the client nor the server role in an association can be changed, or will be changed, under the component paradigm. Here, component adaptation according to a different context is restricted to inserting an adapter between the two roles. But even if the adapter translates the interactions between client and server correctly, there is no guarantee that the semantic interaction between the two will be meaningful. The two deployed and integrated components have been initially developed in complete mutual ignorance for each other, so although a component is alright in one deployment context, e.g., at the component vendor's site, it may completely fail in another deployment context, e.g., in the component user's application. And this is not a problem of the individual components, because they may be individually perfect, but of their interaction, or their interaction within a new context. Even if such components have been developed by the same organization, and the development teams have full access to all software artifacts, something can go horribly wrong, as the ARIANE 5 crash 1996 illustrates.

1.2.2 The ARIANE 5 Failure

The first launch of the new European ARIANE 5 rocket on June 4th, 1996, ended in failure, and, consequently, in the rocket's destruction about 40 seconds after it had taken off. This is probably the most prominent real-world example to illustrate the fundamental problem in testing component-based systems. What had caused the failure was later identified through an extensive investigation which was published in a report by the inquiry board commissioned through the European Space Agency (ESA) [104]. In addition, the failure was analyzed by Jézéquel and Meyer [97].

The inquiry board found out that the on-board computer had interpreted diagnostic bit pattern from the inertial reference system as flight data, and wrongly adjusted the flight angle of the rocket. This had caused the rocket to veer off its intended flight path and disintegrate in increased aerodynamic loads. The inertial reference system did not send correct attitude data to the on-board computer due to a software exception which was caused through a conversion of a 64-bit floating point value into a 16-bit integer value. The floating point value was greater than what could be represented by the integer variable which created an operand error in the software. The error occurred in a component that was taken from the ARIANE 4 system and reused within

the ARIANE 5 system, and that performed meaningful operations only before lift-off. According to ARIANE 4 specifications, the component continued its operation for approximately 40 seconds after lift-off and provided data representing the horizontal velocity of the rocket in order to perform some alignments. However, this was not a requirement of ARIANE 5. The exceptionally high values that caused the operand error were due to the considerably higher horizontal velocity values of the much more powerful ARIANE 5. The requirement for having the inertial reference component continue its operation for some time after lift-off came from the particular preparation sequence of ARIANE 4, which did not apply to the newer ARIANE 5.

1.2.3 The Lessons Learned

We can now have a closer look at why this is the typical failure scenario in component-based software construction. The reused component coming from the ARIANE 4 was obviously alright, and compliant with its specification. The ESA had used that component successfully and without any trouble for years in the ARIANE 4 program. So they could claim reasonably that this was a highly dependable piece of software that may also be used successfully in the new ARIANE 5 program. Or could they not?

Apparently, something went wrong because the importance of the context in component-based development was not considered. In this case the context is the integrating system (the new ARIANE 5) that exhibits an entirely different usage profile of the component from the original system (the old ARIANE 4). The much higher velocity of the new rocket generated values in the original inertial reference component that the designers of that component had not anticipated or considered at the time. At the time of its initial development, the ARIANE 4 engineers did not anticipate a new rocket that would require such velocity numbers. In other words, the new system exhibited a usage profile of the component that its original development team did not take into consideration at the time. The developers implemented the component in a way that provided more than enough margin for the failure of the application under consideration.

It is unlikely that contemporary component-based software testing technologies, some of which will be introduced in this book, could have prevented the crash. At that time, ESA's quality assurance program was aimed only at hardware failures, so that they would not have identified the software problem. Well, they could have, if they had considered software failures as a problem at the time, and in fact they do that now, and tested the system properly. But it clearly illustrates the effect that the integration of a correct and dependable component into a new context may have on the overall application. The fact that components may be facing entirely different usage scenarios, and that they have to be tested according to every new usage scenario into which they will be brought, is the most fundamental issue in testing component-based systems.

1.3 Model-Based Development and Testing

Modeling and the application of graphical notations in software development have been receiving increasing attention from the software engineering community over the past ten years or so. The most prominent and well known representative of graphical modeling notations is the Unified Modeling Language (UML) [118]. This has actually become a de-facto industry standard; it is readily supported by quite a number of software engineering tools, and the amount of published literature is overwhelming. The UML is fostered and standardized by the Object Management Group (OMG), a large consortium of partners from industry and the public sector (e.g. universities and research organizations) worldwide. The OMG's mission is to define agreed-upon standards in the area of component-based software development that are technically sound, commercially viable, and vendor independent. The UML, among other standards that the OMG has released over the years, is one of their earliest products, with its initial roots in the object-oriented analysis and design methods from the late 1980s and early 1990s [60]. It is a graphical notation, or a language, and not a method. I will introduce a method, the Kobra method [6], that is based on the UML, in Chap. 2. The UML can be used for specifying and visualizing any software development artifact throughout an entire software development project. The initial overall goals of the UML are stated as follows [44], and the language addresses these topics sufficiently:

- Model systems with object-oriented concepts.
- Establish an explicit coupling between conceptual as well as executable artifacts.
- Address the problems of scalability and complexity.
- Create a notation that can be used by humans as well as by machines.

In particular, the last item is being tackled more and more by researchers in the domain of generative programming that is based on, or follows the fundamental concepts of, Model-Driven Architectures (MDA) [22, 79]. The basic idea is to develop a system graphically in the form of UML models, pretty much according to the ideas of computer-aided design in mechanical engineering, and then translate these models into an executable format. The correlative technology for this second step in mechanical engineering is computer-aided manufacturing. Generative programming is not as simple as it may sound, and I will present some of the challenges of this subject in Chap. 2, but only marginally. I will give an overview on the primary concepts of the UML in Chaps. 2 and 3, and I will use the UML to specify the examples throughout the book.

1.3.1 UML and Testing

Testing activities that are based on models, or use models, are becoming increasingly popular. We can see this in the number of publications that have

been emerging over the last few years, for example [1, 87, 88, 100, 119], to name only a few. UML models represent specification documents which provide the ideal bases for deriving tests and developing testing environments. A test always requires some specification, or at least a description or documentation of what the tested entity should be, or how it should behave. Testing that is not based on a specification is entirely meaningless. Even code-based, or so-called white box testing techniques, that initially only concentrate on the structure of the code, are based on some specification. The code is used only as a basis to define input parameter settings that lead to the coverage of distinct code artifacts. In Chap. 3, “Model-Based Testing with the UML,” I will give a more extensive overview of these topics. Models are even more valuable if UML tools that support automatic test case generation are used. In general, we can discriminate between two ways of how to use the UML in tandem with testing activities; this is further elaborated upon in the following subsections, and in Chaps. 3 and 4:

- Model-based testing – this is concerned with deriving test information out of UML models.
- Test modeling – this concentrates on how to model testing structure and test behavior with the UML.

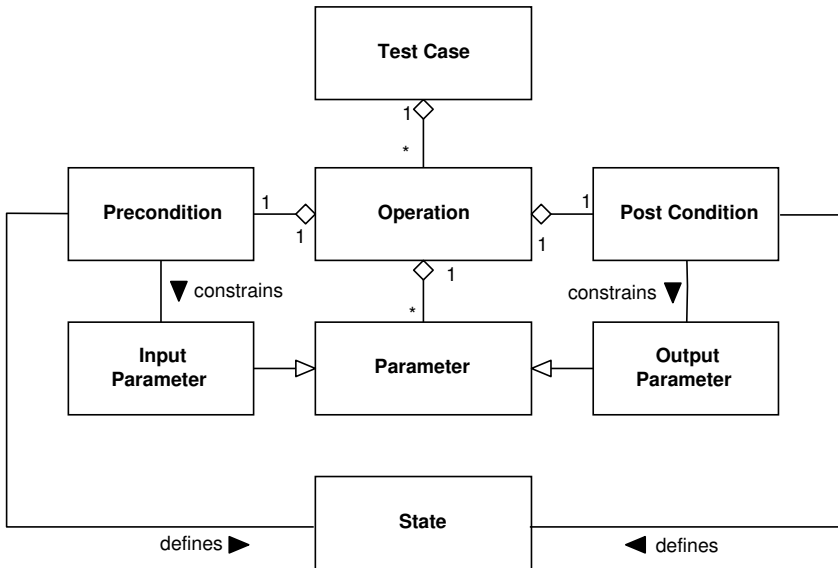


Fig. 1.4. UML-style representation of the concepts of a test case

1.3.2 Model-Based Testing

The UML represents a specification notation, and testing is the using or the applying of the concepts of a specification. A test case for a tested component, for example, comprises one or more operations that will be called on the tested object, a precondition that defines constraints on the input parameters for the test and determines the initial state of the object, and a postcondition that constrains the output of the tested operation, and defines the final state of the object after test execution. These concepts are depicted in Fig. 1.4, and it becomes apparent that this maps exactly to the lower part of the component meta-model depicted in Fig. 1.3 on page 8. So, we can map the concepts of a component exactly to the concepts of a test case for a component. Although, for a test we will need these concepts twice, once for the specification of what should happen in a test and once for the observation of what really happens. A validation action can then be performed to determine whether the test has failed or passed, and this is called the verdict of a test. The concepts of a test case are therefore a bit more complex, but the UML Testing Profile defines them sufficiently [120]. Hence, the UML readily provides everything that is necessary to derive tests and test cases for a component, and it even provides sufficient information to define entire application test suites. This will be described in Chap. 4, “Built-in Contract Testing,” and Chap. 5, “Built-In Contract Testing and Implementation Technologies.”

1.3.3 Test Modeling

Test cases or a test suite represent software. Any software, whether it performs any “normal functionality” or whether it is especially crafted to assess some other software, should be based on a specification. The UML is a graphical specification notation, and therefore it is also adequate for the specification of the test software for an application. Why should we apply a notation for testing that is different from the one we use for developing the rest of a system?

In general, the UML provides everything that is required to devise a testing framework for components or entire applications. However, there are some very special concepts that are important for testing, as I have stated in the previous subsection. These very special concepts are provided by the UML Testing Profile that extends the meta-model of the core UML with testing artifacts and testing concepts [120]. Chapters 3 and 5 concentrate on this topic, among other things.

1.4 Summary and Outline of This Book

This book addresses two of the three primary challenges in testing component-based software systems that I have identified in Sect. 1.2:

- Lack of access to a component’s internal implementation, and as a consequence low observability, low controllability, and, thus, low testability of components.
- Testing of a component in a new context for which it had not been initially developed.

This book focuses on built-in testing for component-based software development in general, and built-in contract testing and related technologies for component-based development in particular. These technologies can, if they are applied wisely and organized intelligently, provide feasible solutions to these typical challenges. The fundamental idea of built-in testing is to have the testing for a software system directly incorporated into the system, although this may not always be the case, as we will see later on. Because testing, if it is viewed in that way, will be an essential part of an application, it must be integrated seamlessly into the overall development process of an organization. After all, testing in this respect is just another development effort that extends a component, or an assembly of components.

Since component-based system development and the UML go so well together, it is logical to set up the testing activities on this notation as well, and, additionally, to have it supplement an existing mainstream component-based development method, such as the KobrA method [6]. UML and testing are a perfect fit, and the component-based development method provides the common framework for putting the three main subjects of this book together: components, modeling, and testing.

The next chapter, Chap. 2 on “Component-Based and Model-Driven Development with the UML,” introduces the KobrA method as the adhesive framework that incorporates all the other technologies. The chapter gives an overview on the KobrA method. It explains its phases, core development activities, and describes its artifacts that are mainly UML models. The chapter will also introduce an example that is more or less used consistently throughout the entire book to illustrate all the essential concepts and technologies.

The next chapter, Chap. 3 on “Model-Based Testing with the UML,” describes why and how the UML and testing are a perfect fit. It briefly compares the more recent model-based testing techniques with the more traditional testing criteria, and then covers the two main areas extensively: model-based testing and test modeling. Under the first topic, I will introduce the diagram types of the UML (version 2) and investigate how these can be good for deriving testing artifacts. Under the second topic, I will introduce the recently published UML Testing Profile and investigate how this may be used to specify testing artifacts with the UML.

In Chap. 4, “Built-in Contract Testing,” which represents the main technological part of the book, we will have a look at what built-in testing means, and where it is historically coming from. Here, we will have a closer look at the two primary challenges in component-based software testing that this book addresses, and I will show why and how built-in contract testing presents

a solution for tackling these. I will introduce the model of built-in contract testing, and describe extensively how testing architectures can be built and how modeling fits under this topic. This chapter concentrates on how built-in contract testing should be treated at the abstract, modeling level. The chapter also comprises the description of the built-in contract testing development process that can be seen as part of, or as supplementing, the development process of the Kobra method.

Chapter 5, on “Built-in Contract Testing and Implementation Technologies,” looks at how the abstract models that have been defined in Chap. 4 and represent the built-in contract testing artifacts can be instantiated and turned into concrete representations at the implementation level. Here, we will have a look at how built-in contract testing can be implemented in typical programming languages such as C, C++, or Java, how contemporary component technologies, including Web services, affect built-in contract testing. An additional section is concerned with how built-in testing can be realized through existing testing implementation technologies such as XUnit and TTCN-3 [63, 69].

Chapter 6, on “Reuse and Related Technologies,” concentrates on how built-in testing technologies support and affect software reuse as the most fundamental principle of, or as the main motivation for, applying component-based software development. It illustrates how the built-in contract testing artifacts can be used and reused under various circumstances, how they support earlier phases of component-based development, in particular component procurement, and how they can be applied to testing product families as generic representatives for software reuse at the architectural level.

Chapters 2 to 6 concentrate only on testing functional and behavioral aspects of component-based software engineering. Additional requirements that have to be assessed in component-based development belong to the group of quality-of-service (QoS) attributes, and Chap. 7, “Assessing Quality-of-Service Contracts,” focuses on these. It gives an overview on typical QoS attributes in component contracts in general and characterizes timing requirements in component interactions, so-called timing contracts, in particular. It concentrates primarily on development-time testing for component-based real-time systems.

Additionally, I give an outline of an orthogonal built-in testing technology, built-in quality-of-service testing, that is typically used to assess QoS requirements permanently during the runtime of a component-based application.



<http://www.springer.com/978-3-540-20864-8>

Component-Based Software Testing with UML

Gross, H.-G.

2005, XVIII, 316 p., Hardcover

ISBN: 978-3-540-20864-8