Testing in the Component Age

Mario Winter

University of Applied Sciences Cologne
Faculty of Computer Science and Eng. Sciences
Campus Gummersbach, Am Sandberg 1
D-51643 Gummersbach
winter@gm.fh-koeln.de

Abstract: At the end of the last century, quality and especially reusability problems of object-oriented software cropped out. As a remedy, nowadays component based software development resounds throughout the developer communities. Nevertheless the special aspects of testing component based systems often remain overlooked.

After characterizing the main differences between object-oriented and component-based software, this paper firstly surveys some basic concepts of component development and software testing. Then applicable techniques for specification and black-box testing of components are depicted, and particularly contract-based test case specification for component interfaces is emphasized. On this groundwork, some new testing levels and testing roles which have to be played in component based software development are proposed. The paper ends with some prospects on appropriate testing tools.

1 Introduction

At the end of the last century, quality and especially reusability problems of object-oriented software cropped out. As a remedy, nowadays component based software development resounds throughout the developer communities. According to Szyperski, „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“ [Sz02].

Components can, but must not be implemented in an object-oriented programming language. Since most components actually are implemented with object-oriented technologies, the notions of a component and a class should be distinguished carefully. To this end, some major differences between components and classes may help:

- Components are bigger than classes. Classes can constitute a component, but not vice versa. A class must adhere to a programming language standard; a component must follow a component standard (which should be independent of a particular programming language). Classes can inherit features from other classes, whereas (normally) there are no inheritance relationships between components.

- Components live in component environments, whereas classes (resp. their instances, aka objects) live in runtime environments. In addition to (object-oriented) runtime environments, component environments in most cases deliver middleware functionality like transaction management, persistency, distributed systems lookup and communication services, and security awareness.

- Components are constituted by several artefacts: sources, binaries, interface (specifications), configuration and deployment descriptors. Their documentation normally includes business knowledge, whereas the documentation of classes mostly comes into the form of a technical API-description (application programming interface).

- Components can be delivered as single entities, are configurable, and composable with other components. Normally, components are distributed in binary form, and the source code isn't available to component customers (at least in case of commercial of the shelf components, COTS).

- Component based systems most often are distributed systems. Thus asynchronous communication, remote procedure calls (RPCs), Internet protocols and other technologies are of concern.

The sequel of the paper is organized as follows. Chapters 2 and 3 survey basic concepts of component-based development and software testing. Chapter 4 depicts applicable techniques for specification and black-box testing of components, emphasizing test case specification based on design-by-contract for component interfaces. On this groundwork, chapter 5 proposes some new testing levels and testing roles which have to be played in component based software development. The paper ends with some prospects on appropriate testing tools.

2 The Component Age

Components are with us for a long time. For example, audio and other home entertainment equipment normally is assembled by specialized components like amplifier, tuner, CD/DVD-Player, speakers, and so on. These components (most often) can be composed simply by interconnecting some wires. Moreover, in the case of such components we wonder if they don't fit together.

Though software components started to play their role in software development some ten years ago, we have to admit that in the case of software components we wonder if they fit together. In the sequel I survey some component architectures in order to demonstrate that most often the technical complexity of the component interfaces and the lack of precise specifications not only hinder the composability, but also the testability of software components.

2.1 Component Architectures

Commercial component architectures are specified by e.g. the Corba Component Model (CCM) of the Object Management Group [OMG], Suns Enterprise Java Beans (EJB) included in the Java Enterprise Edition (J2EE) [EJB] and Microsoft's COM+ and .NET technologies [COM]. These architectures differ regarding their support of operating systems and programming languages, communication mechanisms, persistency, transaction management, security, and directory and naming services.

Common to CCM and EJB is the container programming model, where a special middleware, called the container, realizes the environment in which the components reside. Normally the container itself is hosted in an application server responsible for the runtime environment of the constituted business applications. Fig. 1 sketches the resulting architecture.

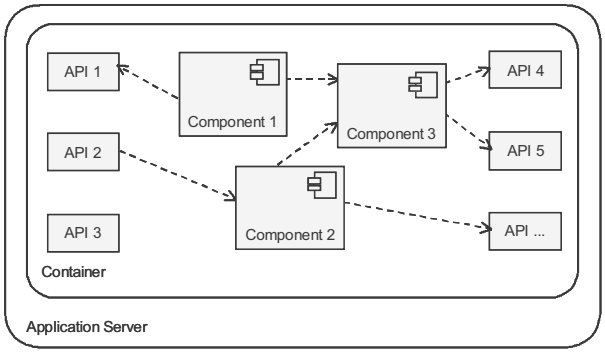

Figure 1: Component programming model

Because of the similarity of EJB and CCM and the heterogeneous, technical diversity and complexity of „company-grown" architectures like COM/COM+, the sequel of this paper is written with an EJB-like component architecture in mind.

2.2 Roles in Component-Based Software Development

Regarding the different concerns of the main architectural elements mentioned above, the following roles have to be played in component based software development [EJB]:

- The application server provider is a specialist in the area of distributed transaction management, distributed objects, and other lower-level system-level services. A typical application server provider is an OS vendor, middleware vendor, or database vendor. Often the application server provider is assumed to be the same vendor than the container provider (c.f. below).

- The container provider delivers the component environment together with tools supporting the development, assembly, packaging, and deployment of components. His expertise is system-level programming, his focus is on the development of a scalable, secure, transaction-enabled container that is integrated with an application server. The container provider typically provides support for versioning and updating the installed components.

- The component provider develops components that implement business tasks or business entities and thus is typically an application domain expert. He is responsible for the code that implements the component's business logic, the definition of the components interfaces, and the component's deployment descriptor. The deployment descriptor includes the structural information (e.g. the name and the interfaces of the component) of the component and declares all the component's external dependencies (e.g. the names and types of resources that the component uses).

- The component deployer is an expert at a specific operational environment and is responsible for the deployment of components. He has to resolve the components external dependencies. For example, the component deployer is responsible for mapping the security roles defined by the application assembler to the user groups and accounts that exist in the operational environment in which the components are deployed. He uses tools supplied by the container provider to perform his deployment tasks.

- The application assembler is a domain expert who composes applications that use components. He works with the component's deployment descriptor and its business related interfaces.

- The system administrator is responsible for the configuration and administration of the enterprise's computing and networking infrastructure that includes the application server and the container as well as underlying databases and other services. The system administrator is also responsible for overseeing the well being of the deployed component based applications at runtime.

3 Software Testing at a Glance

Since humans tend to err, software, being among the most complex human artefacts, was, is, and – so far as foreseeable by me – will be deficient. Though the prevention of errors by far would be the most effective means to achieve software quality, we humbly have to cope with our limitations in applying any methods and techniques, thus post hoc quality assurance will stay with us. This section sketches some terminology on software testing as one of the bullets against defective software [Ve04].

Firstly, an *error* is a human action that produces an incorrect result. An error may lead to a *fault*, which is a flaw in a component or system, e.g. an incorrect statement or data definition that can cause a component or system to fail to perform its required function. A fault, if encountered during execution, may cause a *failure* of the component or system, i.e. an observable deviation of the actual behaviour from the expected behaviour.

Secondly, *software testing* is the process of checking software products. It aims at

- verifying that the product satisfies the specified requirements,
- demonstrating that it is fit for purpose and
- detecting defects of the product.

Frankly, software testing aims at checking that the software does what it should do and that it doesn't do what it shouldn't.

Lastly, the *test process* consists of all life cycle activities concerned with checking software products and related work products. The fundamental test process comprises *test activities* like planning, specification, execution, recording and checking for completion. When linked to some responsibilities in a project and organised and managed together, a coherent group of test activities is called a *test level*. Examples of test levels are component test (aka unit test), integration test, system test, and acceptance test.

Testing may or may not depend on executing the software. If it doesn't, one conducts *static testing* at specification or implementation level, e.g. reviews or static code analysis. If it does, we talk about *dynamic testing* which executes the software by stimulating it through some input, observing its actual behaviour, and comparing the observed behaviour with the expected one (Fig. 2). In this context, a *test case* is a set of input values, execution assumptions, expected results and execution effects, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement.

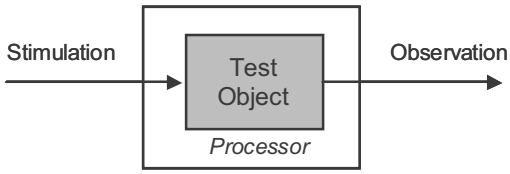

Figure 2: Dynamic Testing

Test cases are specified by using some test design technique. In *black-box testing*, the input values and execution assumptions are derived from the specification, while in *white-box testing* the implementation is used as well. In both cases, the expected results and execution effects are derived by the specification, since it is the specification that tells what the software should and shouldn't do.

4 Specification and Testing of Components

In this section some specification techniques for components together with appropriate testing techniques are assembled.

4.1 Specification Techniques

An audio equipment component's specification tells us about acceptable input signals (voltage and amperage limits) and output signals the component will deliver. In addition to those functional specifications, some non-functional, global operating conditions (temperature, humidity and so on) often are specified – all independently of the components implementation.

According to Bertrand Meyer [Me03], for each software components interface we need the same three kinds of specification elements:

- The *precondition* (of an operation) states the properties that must hold whenever the operation is called. It refers to input-parameters of the operation und the state of the component before the operation is activated.

- The *postcondition* (of an operation) states the properties that the operation guarantees when it returns (assuming its precondition was satisfied). It refers to output-parameters of the operation und the state of the component after the operation is completed, and may refer to the input parameters and the initial state, as well.

Both preconditions and postconditions describe properties of individual operations, but often there are more general properties that one wishes to specify.

- To this end the *invariant* (of the component) expresses global properties of all instances of a component, which must be preserved by all operations. It refers to the state of the component and must be satisfied before and after each execution of an operation

Together, pre- and postconditions of all operations of a component interface together with the general properties stated by the invariant answer the questions of "What does the component expect?" "What does it deliver?" and "What does it maintain?" Bounded to the components interfaces, they resemble a contract between the component and its clients, saying: "If my clients promise to call me with the precondition satisfied then I, in return, promise to deliver a final state in which the postcondition (and my invariant) is satisfied".

Sometimes the behaviour of a component (resp. its instances) is specified by a UML *state chart* [OMG], depicting the (sets of) a components properties which may be observable at some point of time, and transitions between states, which are triggered by some events and my lead to some actions of the component before it comes to rest in some – not necessarily other – state. Here the states resemble the components invariant, whereas the events often are guarded by preconditions and the actions are described by postconditions.

Interaction diagrams bear the possibility not only to specify the behaviour of a single component, but the interplay of several components including interactions with the container. An interaction diagram shows how (instances of) components work together in some scenario by sending messages, or activating operations, over time, thus providing a "grey-box" view of a component based system.

To specify the functionality offered by a component based application system in a "black-box" view, *use cases* are a popular means. Each use case specifies a unit of useful functionality that the component based system provides to its users, i.e., a specific way of interacting with the system[OMG]. This functionality, which is initiated by an actor, must always be completed for the use case to complete. It is deemed complete if, after its execution, the system will be in a state in which no further inputs or actions are expected and the use case can be initiated again or in an error state.

4.2 Testing Techniques

Choosing an appropriate testing technique depends on several questions, among them are:

- Which quality should be achieved (risks, budget, …)?
- Which errors are most prominent?
- Who specifies the tests (developer, assembler, domain expert, …)?
- Which information is at hand (specification, code, …)?

As already seen in section 2.2, we have to distinguish the component developer, who knows the implementation, from the component user (component deployer, application assembler), who only has the component specification at hand.

The component developer may use the whole bunch of testing techniques known from the testing literature. He may use control-flow based white-box techniques, which have proofed of value both in theory and practice. Here the degree of executed statements, branches, and paths of the code is an objective measure of test coverage. A big pro of these techniques is the great many of commercial testing tools available for many programming languages and platforms.

Both the component deployer and the application assembler have no access to the components code. Since they do not know how the component is implemented, they are restricted to black-box testing techniques, which suppose a specification of the components interface, including functional and non-functional aspects.

Which black-box testing technique may be used is primarily determined by the nature of the available specification. According to section 4.1, table 1 list some kinds of specifications and some testing techniques and coverage criteria. In the sequel of this section I sketch these testing techniques in some more detail.

Table 1: Component Specification and Testing Techniques

Specification	Type	Concrete Syntax	Method / Coverage Criteria
Contract based (declarative)	Formal language	UML-OCL	All conditions, MC/DC, ...
	Formal theory	Object-Z, VDM	All clauses
	Informal (natural language)	API	All functions
State based (operational)	Diagram with formal semantics	UML state chart	All states, all transitions, n-Paths, ..., all paths
Interaction based (operational)	Message based	SDL MSC,UML sequence diagram	All messages, all nodes, all branches, ..., all paths
	Structure based	UML communication diagram	All messages, all links
Function based (declarative/ operational)	Informal (natural language)	UML use case diagram	Normal flow, all alternate flows, ..., all flows
	Diagram with formal semantics	UML activity diagram	All actions, all transitions, n-Paths, ..., all paths
		UML sequence diagr.	All messages, all nodes, all branches, ..., all paths

The goal of the contract-based interface testing is the verification of all operations offered by the component under test (CUT) [Wi01]. As a prerequisite the interface(s) offered by the CUT have to be specified by contracts, i.e. pre- and postconditions and invariants. On the one hand the conformance of the CUT's realization w.r.t. its specification is verified by test cases, which demonstrate that the CUT delivers the specified results if used with its preconditions satisfied. On the other hand calling the operations with some precondition violated tests the robustness of the CUT. Testing coverage can be measured by the degree of (atomic) conditions evaluated to true resp. false and the exceptions thrown.

State based testing has a long tradition especially in software testing. Considering the specifying state chart one tries to stimulate the CUT such that e.g. all states and all state transitions are covered. In practice state charts mainly are used to specify the container related life cycle of components, since they tend to grow beyond limits if used to specify business logic. Thus state based testing is mainly conducted in testing the component-container interface.

In interaction based testing test cases are specified from interaction diagrams, i.e. sequence and communication diagrams. Interaction based testing bears similarity to control flow based testing, since the nodes (objects, operation activations) and the branches (guarded messages, repetitions) should be covered [Fr02].

Use case based testing verifies that the expected flow and the alternative flows specified in the use case are delivered correctly. Since use cases normally are described rather informally, the specification of concrete test cases with particular values for inputs and expected outputs may be hindered. This can be handled by augmenting the use case specifications by activity diagrams depicting the whole possibility of scenarios and/or by sequence diagrams, each of which depicts one single, concrete scenario for the use case [KSW01].

5 Testing Levels in Component-Based Development

Besides the classical testing levels mentioned in chapter 3, some new testing levels have to be considered in component-based development. In the sequel I describe container testing and component testing, the latter of which can be divided further into deployment testing, component-container testing, and component-application testing.

5.1 Container Testing

First of all the container itself, being the environment in which components come into life, has to be tested. Among the testing goals are container standard conformity, performance, and robustness. Generic components without real business logic or a set of off the shelf components may be used to test the containers deployment facilities. Mainly the container provider does container testing. Nevertheless, the system administrator has to assure that each new version of a container type already used in his organization and especially each new container type considered to be used is installed into the actual application server and tested against an application- or business-specific operational profile. Besides the obligatory functional tests also performance, load, stress and vulnerability testing has to be conducted by him.

5.2 Component Testing

Even testing a component in isolation is more than traditional unit testing, because – as seen in chapter 2 – a component is assembled from several "units"(classes, modules, …) and lives in a complex environment. Component testing focuses on the deployability of the component, the component-container interface, and the component-application interface.

Deployment Testing

The component provider validates deployability and standard life cycle adherence of his components w.r.t. all container types the components are specified for. Deployment testing should demonstrate that the deployment scripts work and the deployment descriptors are interpreted right by all containers.

Component-Container Testing

Component-container testing focuses on the standard life cycle as specified e.g. by state charts in the component standard. Here reusable, generic test cases for every component type may be specified and automated. Component deployability test cases as well as component-container test cases developed by the component provider may be delivered together with the component s.t. the component deployer can reuse them, too.

Component-Application Testing

As in any other software development project, the component provider conducts functional testing of his components business logic. Since the business logic is accessible only through the components public interfaces and mainly be used by applications comprising the component, in component-based development this test level can be called component-application testing. The component provider possesses the components sources, so besides black-box testing he also has white-box testing techniques at hand. As a drawback, the component provider only can guess the real usage of his components, s.t. he is only able to specify rather generic component-application test cases.

From the component provider's point of view, specification based black-box test cases constitute some sort of acceptance test, which must be passed by the component before it may be delivered. On the other hand, both component deployer and application assembler would gain from reusing these tests in order to validate that the component behaves as expected also in its new working context. So these tests ideally should be delivered together with the component e.g. as executable code or as built in self tests.

The application assembler specifies black-box test cases to validate all functions of the components used in his application. These test cases are derived from the application specific requirements on the component. Since such specifications are needed to select the appropriate components, the application assembler may assemble automated application-component test suites a priori to aid and accelerate the component selection process.

Component-application tests mainly have to be executed inside of a container, since it provides the services needed by the components (c.f. section 2.1). Additionally, appropriate test drivers have to be developed e.g. using servlets, JSP, or HTML. In case of the EJB container bundled within the IBM Websphere application server, a generic driver comprising a GUI for manual tests is comprised.

Unfortunately, component-application testing inside a "real" container and application server often isn't applicable, since the repeated deployment of "fresh" components for every test case may consume too much time. Also the usage of a container test stub is impractical because its development would cost nearly as much as the development of a „real" container [Li02]. As a remedy one may factor out business operations not relying on container functionality into some helper classes, thus using the component only as a façade [Ga95] to these business operations. So at least the business logic may be tested outside the container, s.t. deploying fresh components is circumvented. As a side effect, also white-box tests become feasible. This (rather extreme) approach is called the box metaphor, since the components represent only a tier layered above and using the business logic tier, which is independent of the component tier and thus of any component technology. It is applicable especially for stateless components, the behaviour of which is independent of their history. For other types of components the container dependencies may be to complex to apply the box metaphor.

Table 2: Component Testing Levels and Roles

	Container Testing	Component Deployment Testing	Component-Container Testing	Component-Application Testing
Container Provider	X	O	O	
Component Provider	X	X	X	O
Component Deployer		X	X	
Component Assembler			O	X
System Administrator	X	O	O	

Table 2 depicts which testing level pertains to which role in component-based development. X denotes primary testing responsibility, O secondary one.

6 Component Testing Tools

Last but not least I consider specific requirements on testing tools for component-based development. One target for automation is deployment, a complex and error-prone process which has to be redone for every test run (at least if the component's code, the deployment script, or the deployment descriptor has been changed). Here scripting tools like e.g. Ant have proved of value [HL02][ANT].

For unit testing at the component provider's side, conventional testing tools may be considered, since e.g. Enterprise Java Beans in many aspects are like „normal" Java classes and can be tested without being deployed into a container. In practice tools like JUnit [JUN] and mock objects [MOC] often are used (for details refer e.g. to [Li02]).

Also for component-based system testing one would not expect special requirements on testing tools, since system testing makes no assumptions about the internals of the system under test. In this case, the complexity caused by components reduces to the (re-) deployment of components.

Special requirements on testing tools emerge in component-container testing, because the interactions of the component and the container are not observable with conventional means. Using a stub container isn't a remedy (c.f. section 5.2) and especially wouldn't allow any reliable statement on the interoperability of the component with the real container used in the production system. At least for Enterprise Java Beans a very promising approach is the one followed by Cactus [CAC], an open source testing framework extending JUnit.

In order to observe the interactions between component and container, in Cactus so-called wrapper objects for the most prominent interfaces (request, response, session) are provided. If needed, these objects can be augmented with additional information, manipulated, and read by the tester. Though this approach requires special, container related knowledge on the testers side, it alleviates the adjustment of relevant component states and the testing of interactions and should pay off in many cases.

7 Conclusions

In this paper I considered some testing aspects of component-based systems. Besides diving into the literature on testing component based software ([GTW03] is an excellent starting point), the following points seem mandatory to me in order to do successful component testing.

- Firstly, specify your requirements on prospective components as precise as possible and derive appropriate automated test cases.

- Standardize your infrastructure (application server provider / versions, container provider / versions, application architecture (product lines)).
- Ask the component provider for component specification and test cases.
- Automate your tests (regression testing will come).
- Ask for component maintenance and evaluation arrangements.
- If in doubt: ask for source code disclosure agreements.
- Educate your people in component modelling, architecture, and design, especially in design by contract and in black-box testing techniques.

Literature

[Fr02] Fraikin, F.: SeDiTeC - Testen auf der Basis von Sequenzdiagrammen. Softwaretechnik-Trends, Vol. 21, Nr. 2, 2002.

[GTW03] Gao, J., Tsao, H., Wu, Y.: Testing and Quality Assurance for Component based Software. Artech House, London, 2003.

[Ga95] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: Design Patterns. Addison Wesley, Reading, Mass., 1995.

[HL02] Hightower, R.; Lesiecki, N.: Java Tools for Extreme Programming. John Wiley & Sons, 2002.

[KSW01] Kösters, G., Six, H.-W., Winter, M.: Coupling Use Cases and Class Models as a Means for Validation and Verification of Requirements Specifications. Requirements Engineering, Vol. 6, Nr. 1, Springer Verlag, London, 2001; pp. 3–17.

[Li02] Link, J.: Unit Tests mit Java, dpunkt-Verlag, 2002.

[Me03] Meyer, B.: The Grand Challenge of Trusted Components. Proc. 25th Int. Conf. on Software Engineering, Portland, Oregon, May 2003.

[Sz02] Szyperski, C.: Component Software - Beyond Object-Oriented Programming. 2nd Ed. Addison-Wesley / ACM Press, 2003.

[Ve04] Van Veenendaal, E. (Ed.): Glossary of terms used in software testing. Version 0.2, Int. Software Testing Qualification Board, 2004.

[Wi01] Winter, M.: Testfallermittlung aus Komponentenschnittstellen. Proc. Imbus QS-Tag 01, Nürnberg, 2001.

[Wi02] GI-TAV Arbeitskreis „Testen objektorientierter Programme“: Test von Komponenten. Proc. 18. GI-TAV-Workshop, 20. und 21. Juni 2002, Hasso Plattner Institut, Universität Potsdam

Links to the web:

[ANT] http://jakarta.apache.org/ant/

[CAC] http://jakarta.apache.org/cactus/

[COM] http://www.microsoft.com/com

[EJB] http://java.sun.com/products/ejb

[JUN] http://www.junit.org/

[MOC] http://mockobjects.sourceforge.net/

[OMG] http://www.omg.org