



Title: **MORABIT Deliverable M3**
Editors: **Colin Atkinson, Daniel Brenner**
Contributors: **Matthias Merdes, Barbara Paech, Rainer Malaka, Lars Borner, Dima Suliman**
Document Version: **1.0**
Date: **30 November 2006**
Status: **Deliverable**

Project Partners:

<p>Heidelberg University</p>	
<p>EML Research gGmbH</p>	
<p>Mannheim University</p>	<p>UNIVERSITÄT MANNHEIM</p>

Sponsor:





Copyright MORABIT

Prof. Dr. Barbara Paech
Institut für Informatik
Neuenheimer Feld 348
69120 Heidelberg
paech@informatik.uni-heidelberg.de

Dr. Rainer Malaka
EML Research gGmbH
Villa Bosch
Schloss-Wolfsbrunnenweg 33
69118 Heidelberg

Prof. Dr. Colin Atkinson
University of Mannheim
Chair of Software Technology
A 5, 6, Gebäudeteil B, Seminargebäude
68161 Mannheim



Abstract

This document describes the MORABIT method for developing components and component-based systems using the features of the MORABIT infrastructure. As such it encompasses milestones M2 and M3 of the MORABIT project plan. The goal of the method is to provide guidelines and heuristics on how to develop and use components in the context of MORABIT, and how best to use the services offered by the MORABIT infrastructure. Since component-based software engineering inherently separates the role of component development from the role of component deployment or use, the method is accordingly organized into two main parts – one for component developers and the other for component users. The description of these two parts is preceded by an introduction to the basic component model assumed in MORABIT, a definition of the different phases of a component/systems lifecycle and a discussion of the different roles involved in MORABIT-based software engineering.

Table of Contents

Abstract	3
Table of Contents	4
PART 1: Overview	6
1 Introduction	6
2 Component Model	6
2.1 Contract-Driven Design	7
3 Life-Cycle Phases	9
4 Configuration Interface	10
5 Roles	11
6 Case Study	12
PART 2: Component Developer Method	14
1 Introduction	14
2 Component Specification	15
2.1 Structural Specification	15
2.2 Functional Specification	17
2.3 Behavioural Specification	17
2.4 Testable Interface Definition	18
2.5 Usage Profile Definition	21
2.6 Extra-Functional Requirements Definition	22
2.7 Functional Test Case Design	23
3 Architectural Design	24
3.1 Structural Design	24
3.2 Interaction Design	25
3.3 Activity Design	25
3.4 Structural Test Case Design	26
4 Test and Countermeasure Design	27
4.1 Risk and Cost Analysis	27
4.1.1 Server Analyses	27
4.1.2 Self Analysis	28
4.1.3 Ranking	29
4.2 Contract Test Case Definition	29
4.2.1 Quantitative Contract Test Case Definition	29
4.2.2 Qualitative Contract Test Case Definition	31
4.3 Countermeasure Design	36
4.3.1 Predefined Countermeasures	36
4.3.2 Custom Countermeasures	37
4.4 Test Strategy Design	38
4.4.1 Self versus Server	38
4.4.2 Test Timing Determination	38
4.5 Test Specification	40
5 Detailed Design and Implementation	41
5.1 Architecture Revision	41
5.2 Implementation Modelling	41
5.3 Implementation	41
6 Development-Time Testing	42
7 Packaging	43
PART 3: Component User Method	44



1	Introduction	44
2	Component Deployment	45
2.1	Component Instantiation and Configuration	45
2.2	Test Request Customization.....	45
3	Deployment-Time Testing	47
4	System Execution.....	48
	References	49

PART 1: Overview

1 Introduction

Component-based development is founded on the notion that new software applications are assembled from general purpose components often built by third party development organizations. It therefore assumes a fundamental separation of concerns between component developers, who create general purpose components with no knowledge of specific applications to which they may be put, and component users, who take existing, prefabricated components and assemble them into new applications. The context in which a component is executed is therefore, by definition, first determined at deployment-time and in general may change at any time thereafter.

The goal of the MORABIT method and associated infrastructure is to support the post-development, in-situ verification and validation of components and component-based systems. In other words, MORABIT focuses on the validation and measurement of the behaviour of components once they have been deployed in a specific system. We assume that components have been subjected to the usual development-time verification activities such as inspection and testing whilst being created, and thus we expect them to exhibit the level of quality which is typical in industry today.

2 Component Model

There is no single, universally agreed definition of the notion of “software component”, but according to Wikipedia, “a component is an object written to a specification”. This is a necessary condition, but does not capture the “intent” usually associated with components. Clemens Szyperski and David Messerschmitt [1] give the following five criteria for what a software component should be able to fulfil:

- Multiple-use
- Non-context-specific
- Composable with other components
- Encapsulated i.e., non-investigable except through its interfaces
- A unit of independent deployment and versioning

This characterization of software components is independent of any specific programming paradigm or language. However, in line with general development practices today, components are almost universally viewed as interacting via a message-based mechanism based on the invocation of procedures. The interaction may be synchronous, requiring the invoker to wait until the procedure has completed execution and has returned its result, or asynchronous, in which the invoker is free to proceed after invoking the message. The former represents the semantics of the classical “remote procedure call” and the latter asynchronous semantics.

Irrespective of the precise semantic of interaction, the consequence is that the functional interface between a component and its users is characterized by the set of procedures or so-called operations that it implements. Collectively, the set of operations which a component makes available (or exports), is known as the *provided interface* of the component, and characterizes the *service* that the component offers. Accordingly, in a particular interaction

(operation invocation) the invoking component is often referred to as the *client* (since it is using the offered service) and the invoked component is referred to as the *server* (since it is offering the service). Strictly speaking the roles of client and server apply to an individual operation, but it is often the case that a given component always plays the same role in its interaction with another component (i.e. client or server). It is then common to refer to the component as being “*the client*” or “*the server*” in that particular interaction. It is also common for a component to play the same role in its interaction with all other components. In this case it can be referred to as “*a client*” or “*a server*”.

However, many components play both roles. It is common for a component to be a server in its interaction with some components and a client in its interaction with others. In other words, in order to implement its own services a component will need to draw upon the services of other components. In general, therefore, a component has one or more *provided interfaces* that describe the services that it offers and one or more *required interfaces* that describe the services that it needs for other services.

2.1 Contract-Driven Design

The notion of components interacting via identifiable interfaces leads to a fundamental principle of component-based development known as **contract-driven design** [2]. This principle holds that the interfaces between components, representing the services that they use or provide, should be documented in terms of the rights and obligations of each party involved. Like parties in a legal contract, the idea is that each component involved in an interaction should know what it may expect and what it must provide in a successfully completed interaction. By the same token, the contract defines the criteria by which an interaction can be judged as having succeeded or failed.

As with other development abstractions, an important distinction in component-based development is the distinction between component types and component instances. Component types are classifiers in the UML sense. In other words, component types are templates that can be instantiated to create component instances with specified properties. Where there is ambiguity, the full terms “component type” or “component instance” are used to distinguish component types from component instances. If the term component is used without qualification, “component instance” is meant.

The development phase is mainly concerned with component types. Indeed, the goal of a component development project is to develop a component type which can be published and sold to potential users. However, component development is not exclusively concerned with component types. The required interfaces of the component type and the design of the internal architecture of the component type are usually expressed in terms of component instances (i.e. instances of other component types). This is illustrated in Figure 1 below using UML notation.

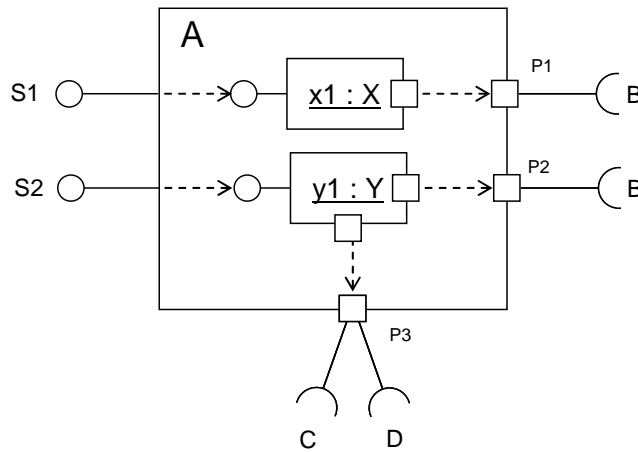


Figure 1 Provided versus Required Interfaces

This figure shows a component type, A, which has two provided interfaces S1 and S2. A realizes these services in terms of various component instance abstractions. The ports P1, P2, and P3 represent component instances which implement the interfaces B, C, and D. Ports P1 and P2 are local identifiers for component instances which both implement the interface B, while P3 is a local identifier for a component instance which implements the interfaces C and D. The figure also shows that the internal architecture of the component type A is described in terms of component instance abstractions. Thus, the service offered at interface S1 is realized by the component instance x1 of component type X, and the services offered at instance S2 are realized by the component instance y1 of component type Y.

The notions of “interface” and “service” are type level abstractions which are associated with component types, while ports are instance abstractions which provide local representations of external component instances. A given instance of a component type is “plugged” to appropriate instances of other component types at deployment-time. “Interface” and “service” are effectively interchangeable terms, and can also be used in a collective or individual sense. Thus, it is common to refer to “the” provided interface of component types such as A, meaning the collective provided interface of S1 and S2.

Contracts are associated with interfaces / services and thus are defined at the type level.

3 Life-Cycle Phases

In traditional development approaches two basic testing notions are used to check the quality of systems assembled from separate modules - the notions of “integration testing¹” and “acceptance testing”. “Integration testing” is a verification technique that focuses on the testing of successively larger groupings of modules, leading up to the system as a whole, in the context of the development environment. According to the terminology of Boehm [3], integration testing aims to verify that “we are building the system right” according to some well defined description of what the system should do. “Acceptance testing”, in contrast, focuses on validation and essentially involves the testing of a deployed instance of the system in the target execution environment before it is put into service. In Boehm’s terminology acceptance testing aims to validate that “we are building the right system” based on the expectations of the customer or users.

When systems are assembled from components at deployment-time and may have their configurations changed dynamically, these notions of testing are no longer adequate. In particular, integration testing no longer makes sense in its traditional form because the precise composition of a system is not known at development-time when integration testing is traditionally performed. The notion of testing “the system” as an integrated whole at development-time no longer applies in the traditional sense therefore. Testing at development-time is still important, but its role is to test the code that implements a component type’s provided interface in terms of representative implementations of its required interfaces. A “representative implementation” of a required component can either be a full working version of the component or a stub which mimics the component for a few chosen test cases. Since these tests are performed at development-time and are exclusively focused on verification against a specification, we simply use the term *development-time testing* for this activity.

In the context of component-based development neither the notion of integration testing nor the notion of acceptance testing is fully appropriate in its traditional form. The former is not appropriate because integration testing can no longer be fully performed at development-time as has hitherto been the case. The latter is not appropriate because the testing that is performed at deployment-time should no longer focus just on validation as has traditionally been the case. Instead, the testing activities that are performed at deployment-time also need to include tests to verify the assembly of components against the system’s specification. It therefore makes sense to combine the notions of integration and acceptance testing into a single activity known as *deployment-time testing*. Such a deployment-time, component testing activity serves the dual roles of validation and verification of the assembled system in its run-time environment.

For systems whose structure remains constant after initial deployment there is clearly no need to revalidate the system once it has been placed in service because any test cases that have already been executed will not be able to uncover new problems. However, many component-based systems do not have a constant structure. On the contrary, an important benefit of component-based development is that it allows the structure of a system to be changed while it is in service. If a change is made, then clearly tests performed at deployment-time may no longer be valid.

¹ For the purpose of this discussion we regard “system testing” as a special case of integration testing – the special case in which all the components have been combined and the final system has been assembled.

The notions of development-time and deployment-time testing are therefore not sufficient to cover the full spectrum of testing scenarios in dynamically reconfigurable component-based systems. We need to add the notion of *service-time testing* as well. Service-time tests are carried out once a system has entered service and is delivering value to users (i.e. is being used to fulfil its purpose). Deployment-time and service-time testing both take place at “run-time” in the sense that they are applied to a “running” system in its final execution environment.

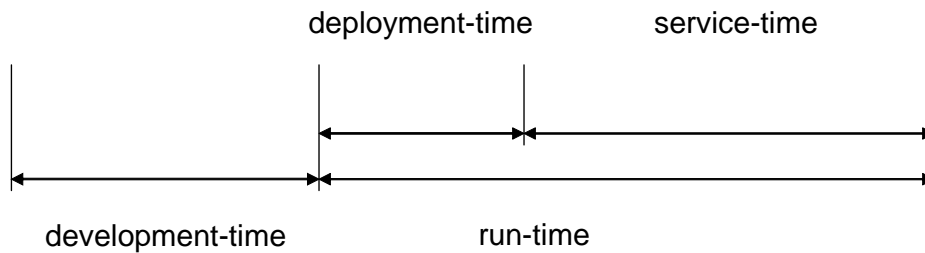


Figure 2 Life-cycle phases

The relationship and role of these different phases in the life-cycle of a component-based system are summarized in Figure 2. At the highest level of abstraction, two different phases exist, the development phase, in which the system is developed and tested using representative servers in the development environment, and the run-time phase in which an instance of the system is connected to actual servers and is running in its final execution environment. The run-time phase is divided into two subphases – the *deployment phase* and the *service phase*. In the deployment phase, the system is set up in its initial configuration and starts to run in its execution environment, but it is not yet delivering service to users. This is important because it allows testing activities to be performed under controlled conditions with known assumptions. In the service phase the system has been put into service and is delivering value to users. During service-time the assumptions that held during deployment-time may no longer be valid.

4 Configuration Interface

The configuration interface of a component defines properties of a component that are set, once and for all, when a component instance is created from a component type and deployed in a new system. Configuration properties can be “set” in a number of ways. The simplest way is that they are normal attributes which are initialized by a normal operation call immediately after the component instance is created. What sets them apart from normal properties is that they should not be changed again after initialization. However, configuration properties can be set by actually customizing the code that is loaded into memory in order to execute the component instance. Thus, if a particular capability is not wanted in a particular installation of an application, the corresponding operations and/or classes responsible for delivering the capability may be completely removed from the loaded software.

Configuration properties can be things like:

- Security level (e.g. logon required)
- Robustness level (e.g. second mail server for backup)
- ...

5 Roles

As with virtually all software methodologies, the MORABIT method foresees several different roles for human beings. Some of these are the typical roles that one finds in component-based development processes, others are more specific to the needs of run-time testing. The specific roles identified in the method are:

- **Component Developer:** the software engineer responsible for describing and developing the component including its specification, design, and implementation. Although the component developer is not responsible for developing and specifying run-time tests, s/he is also responsible for integrating any component-driven reaction to test results into the design and implementation of the component. Therefore he has to work closely with the run-time test designer.
- **Development-Time Tester:** the software engineer responsible for the testing activities applied to the component type at development-time in the development environment. This is similar to the traditional testing role in classic software engineering, and involves the definition and execution of typical defect and verification-oriented test cases. The basic goal of development-time tests is to check that the component type fulfils its provided interface. They are therefore performed before the component type is made available on the market. The only significant difference is that an instance of the component type can only be tested in association with “representative” or stub server components, since the actual servers in a deployed instance of the system are not yet known. The development-time tester is also responsible for executing and dealing with tests at development-time.
- **Run-time Test Designer:** the software engineer responsible for designing and/or selecting the run-time test cases that will be shipped with the component and for defining the run-time tests that an instance of the component type will request from the MORABIT infrastructure in a specific run-time application. The run-time test designer has to work closely with the development-time tester to determine which of the development-time test cases are suitable run-time test cases, and how they need to be modified for this purposes (if at all). He also has to work closely with the component developer to define the test requests.
- **System Deployer:** the software engineer responsible for deploying a set of component types to create the new system for a specific application. As well as instantiating and integrating a set of component instances from the set of component types, the system deployer will usually also use the test infrastructure to perform a series of run-time tests to verify and validate the system.
- **Test Administrator:** configures the infrastructure at deployment time and runs tests provided with the components to check the quality of the system. The test administrator is responsible for deciding whether the system can be put into service based on the results of the deployment time testing activities. He/she is also responsible for responding to the results of any service-time test whenever manual intervention is required.
- **System User:** An ordinary user of the services offered by a system deployed in a MORABIT enabled software environment.

- **Device Administrator:** A “senior” system user who allows or prohibits test activities on the user’s device

As is often the case in practical software development projects, several of these roles may actually be performed by the same person. In fact, in an extreme case they can all be performed by the same person. Roles that are good candidates for being performed by the same human individual are the development-time tester and the run-time test designer roles, since the run-time test designer has to become familiar with the development tests in any case, and the system deployer and service-time test administrator, since these are all concerned with the run-time execution of a set of component instances in a particular deployed scenario.

6 Case Study

The case study we use to illustrate the approach is an Auction House System whose job is to enable auction participants to interact electronically using mobile devices. Unlike fully electronic auction applications like e-bay, the users of this system need to be actually present at a physical auction. The system supports the auctioneer by allowing users to offer and bid for items and conduct payment transactions electronically.

The overall architecture of the Auction House System is illustrated schematically in Figure 3. Each of the nodes in this figure is a component that can (but need not) be executing on an independent device. Each of the edges represents a remote interaction. The central component in the system is the Auction House. This is the central server which mediates requests from Auction Participants - the clients - usually hosted on mobile devices such as notebooks or PDAs. The other components in the system assist the Auction House in delivering its services. The multi-object icons (e.g. Auction Participant, Bank, Mail Server) indicate that multiple instances of the identified component type can interact with (i.e. be connected to) an instance of the Auctions House component type during its lifetime. Thus, there can be multiple, concurrent users of an instance of the Auction House. The Activity Logger is responsible for storing a log of all the main activities in an auction, such as auction initiation, the offering of items, the placing of bids, the completion of auctions etc. Auction Manager and Participant Manager are responsible for storing the important data involved in an auction. Instances of the Mail Server component types are responsible for dispatching mails and instances of the Bank take care of handling payments.

Figure 3 depicts a sketch of a typical configuration of the system. The actual configuration of the system is highly dynamic and changes as new participants join and leave an auction. The Bank and Mail Server component instances may also change if compatibility or reliability problems are detected.

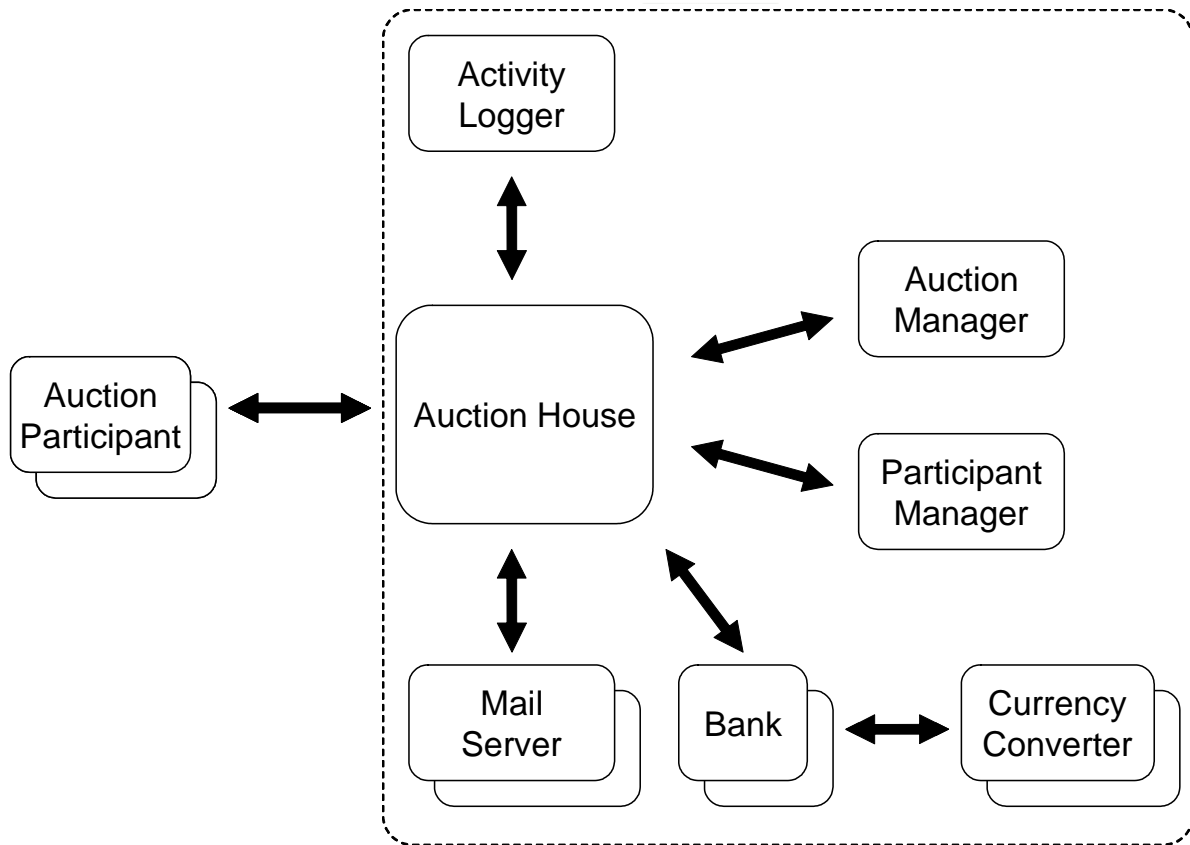


Figure 3 Auction House System architecture

Since it plays the central coordinating role in the architecture, the Auction House has a major role on the dependability of the system – that is, the ability of the system to fulfil a users expectation at any time. In fact, from the point of view of auction participants an instance of the Auction House (AH) is the system and the other component instances are hidden. Determining the reliability of the AH is a non-trivial tasks because it is itself also assembled from components at deployment-time and many of these depend on the external components to which an AH is connected at any particular point in time. Establishing whether an AH is capable of fulfilling its contract is therefore a challenging task which would take a great deal of time and effort if performed in the traditional way.

PART 2: Component Developer Method

1 Introduction

The goal of the overall MORABIT method is to make run-time testing an integral part of mainstream component-based software engineering methods. To this end it enhances a full life-cycle development method with guidelines for supporting and using the capabilities offered by the MORABIT run-time testing infrastructure. The general aim is to minimize the impact of run-time testing on the normal application development process by separating concerns for testing and functionality to the greatest extent possible. This is reflected in the separation between the roles of the component developer and the run-time test developer. Further aims are to induce as little effort as possible on the different roles.

As mentioned in the introduction, the overall MORABIT method is split into two basic parts – the Component Developer Method dealing with the construction of component types and the Component User Method dealing with the deployment and use of component instances within a specific application. This Component Developer Method is composed of six main steps indicated below. In the following sections these are presented in the general order in which they should be performed. However, this is not intended to imply that they must be performed in a strict linear sequence. On the contrary, we expect these to be applied within the context of iterative and incremental development cycles.

- 1 Component Specification**
 - 1.1 Structural Specification
 - 1.2 Functional Specification
 - 1.3 Behavioural Specification
 - 1.4 Testable Interface Design
 - 1.5 Usage Profile Definition
 - 1.6 Extra-Functional Requirements Definition
 - 1.7 Function Test Case Design
- 2 Architectural Design**
 - 2.1 Structural Design
 - 2.2 Interaction Design
 - 2.3 Activity Design
 - 2.4 Structural Test Case Design
- 3 Test and Countermeasure Design**
 - 3.1 Risk and Cost Analysis
 - 3.2 Contract Test Case Definition
 - 3.3 Countermeasure Design
 - 3.4 Test Strategy Design
 - 3.5 Test Specification
- 4 Detailed Design and Implementation**
 - 4.1 Architecture Revision
 - 4.2 Implementation Modelling
 - 4.3 Implementation
- 5 Development-Time Testing**
- 6 Packaging**

2 Component Specification

In this phase a high-level specification of the service to be realized by the component type under development is created². The purpose of this specification is to describe the basic functionality offered by the component type (the provided service) and the basic functionality that it needs from other components (the required services³). This also includes any additional operation that the component type needs to offer to support run-time testing. It also describes the non-functional properties that the component type offers and requires. Since a component type's specification defines everything that is visible about its instances at run-time, it effectively defines the contract between instances of the component and its run-time clients and servers.

The MORABIT method does not prescribe how the service specification is represented – it only specifies what information it should provide. A service specification will usually be captured in a platform independent language such as the UML, but any suitable language is acceptable. In the remainder of this document we use the UML applied according to the principles of the KobrA method [4], since this is a good match to our needs.

A KobrA service specification involves the creation of three distinct views of a service: the structural view which describes all structural information that a user of the service needs to be aware of, the functional view which describes the effects of the operations exported by the service in terms of pre- and post-conditions, and the behavioural view which describes the allowable sequences of operation invocations in terms of externally visible states and state transitions.

2.1 Structural Specification

The structural view describes the information that potential clients of the component need to be aware of when interacting with the component. Primarily, this is the types of the parameters and return values of the component's operations, but it can also include other information such as the component's position in a taxonomy.

In KobrA, the structural view of a component is represented as a class diagram, with possibly an additional object diagram. The structural class diagram for the specification of the Auction House component is illustrated in Figure 4.

² When we use the term “component” unqualified we mean “component type”

³ In general a component can support multiple services, but for simplicity we regard these as a single composite service. Thus, without loss of generality we regard components as having just one provided service. The same also holds for the required interfaces. Without loss of generality we regard components as having just one required interface.

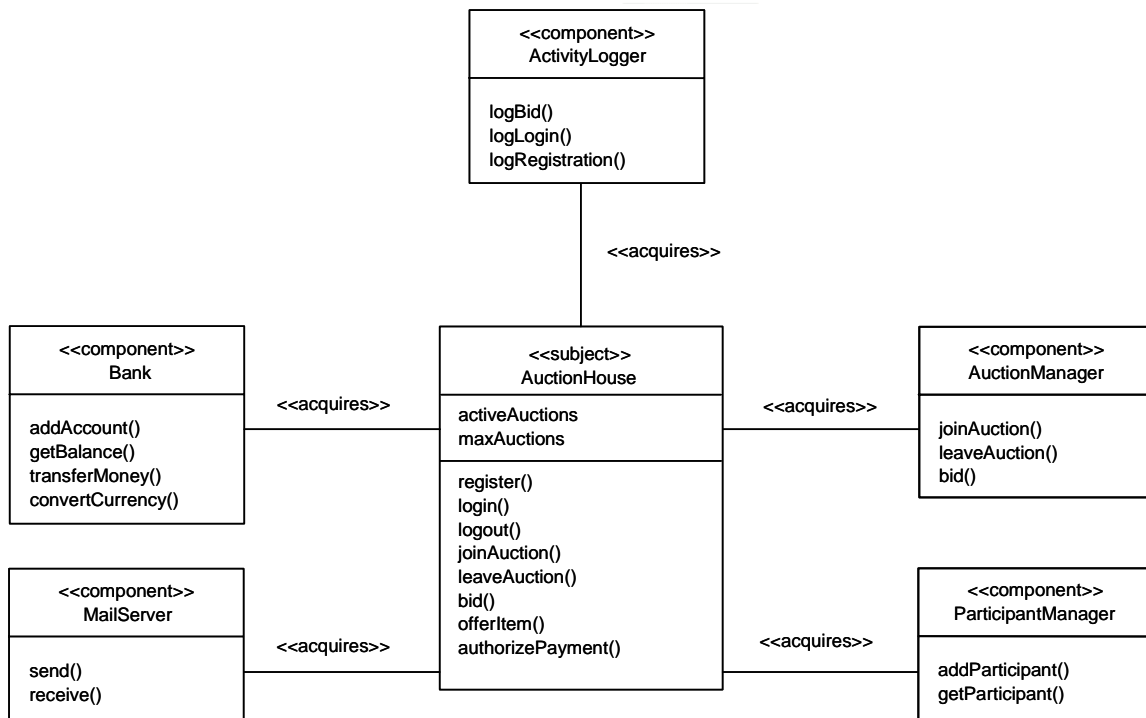


Figure 4 Auction House Specification Class Diagram

This diagram shows the AuctionHouse component type, represented as the class with stereotype <<subject>>, together with the required interface types, represented as the classes with stereotype <<component>>. The AuctionHouse class lists the operations provided by the auction component type. This includes a configuration operation setMaxAuctions() which sets the maximum number of auctions which an instance of the AuctionHouse can handle at any one time.

The class diagram can be accompanied by an object diagram which shows the services required by an instance of AuctionHouse as instances of server components.

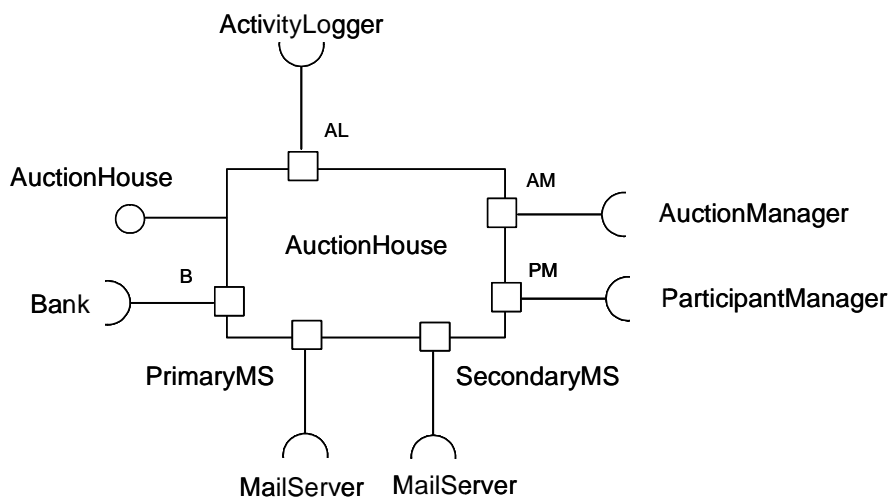


Figure 5 Structural View of the Auction House in UML lollipop notation

Figure 5 shows that an instance of AuctionHouse actually needs to be plugged to two instances of MailServer, one playing the role of the primary mail server and the other the role of secondary mail server. This diagram also shows the local names by which an instance of AuctionHouse refers to the instances of its serving components.

2.2 Functional Specification

The functional view of the component describes the set of operations offered by the component, providing a description of the effects of each operation in terms of OCL pre- and post-conditions. In general, one operation specification is created for each operation of the component type (shown for example in the structural model by the class stereotyped <<subject>>). Figure 6 shows the operation specification of the bid() operation.

Name	bid()
Description	
Constraints	the executing bidder is registered at the AuctionHouse
Receives	sessionId : String; bid : Money;
Returns	boolean
Sends	anActivityLogger.logBid(username, bid)
Reads	
Changes	
Rules	
Assumes	bidder is logged in bidder is not the initiator of the current auction
Results	if the sent bid is higher than the current highest bid, return true; otherwise return false;

Figure 6 Specification of the AuctionHouse’s bid() operation

2.3 Behavioural Specification

The behavioural view provides a description of any externally visible states that the auction house exhibits. Figure 7 shows the behavioural model for our auction house example.

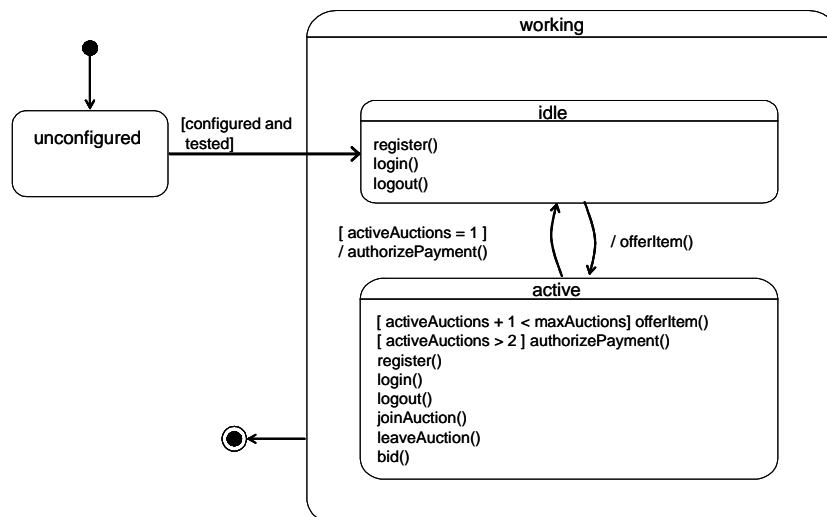


Figure 7 Auction House Behavioural Model

As with all MORABIT components, the AuctionHouse has two top level states – unconfigured and working. A component instances enters the unconfigured state when it is first created. In this state it has not been configured and has not been connected to any of its required servers. Only when the configuration operations have been executed (in this case `setMaxAuctions()`), the appropriate dependencies have been set and the component instance has been tested does the system move into the working state. This is the state in which the system is servicing user requests. The working state has two substates – idle and active. In the idle state the system is able to respond to `register()`, `login()`, and `logout()` request, but since there are no active auctions it is unable to respond to requests that relate to active auctions. Only when an `offerItem()` operation has been invoked and an auction started is an AuctionHouse instance in the fully active mode when it can respond to all actions.

2.4 Testable Interface Definition

Everything that has been defined up to this point is known as the *functional interface* of the component. This describes the behaviour and properties of the component in its role as a server. However, not all of the behaviour of the component type is represented in the functional view of the component type. Some is defined in the behavioural or structural views. This next step in the specification process is to make sure that all the behaviour and properties defined in the specification are accessible as operations, not just those operations explicitly identified and specified in the functional view. This is necessary to make instances of the component fully testable at run-time. Information in the behavioural and structural views which is not testable via the basic *functional interface* has to “functionalized” in the form of additional operations. This gives rise to the so-called *testable interface*. The functional interface and the testable interface together form the *extended interface*.

The purpose of the testable interface is to make sure all of the semantic properties defined in the service specification are accessible as operations so that they are amenable to testing at run-time. In general this can be done with information that comes from four main sources.

- 1 logical attributes of the service
- 2 logical states of the service
- 3 extra-functional requirements
- 4 configuration settings

In the case of (1), operations for setting and getting the value of each logical attribute should be added. In the case of (2), operations for setting and getting (or for confirming) the logical states of components should be added, and in the case of (3), operations for getting the value of each quality-of-service (QoS) property should be added. Such operations should include a value “undefined” representing the fact that a particular run-time environment is not able to provide the necessary API operations to realize the method. This will allow the component to be used even if the QoS measurement method cannot be supported. In the case of (4), an operation must be provided to check the value of configurable information. For example, an operation is needed to check the maximum number of auctions.

In addition, an operation should be defined for any other semantic information of any kind in the service specification that affects the run-time behaviour of the component and is in principle measurable. As with the operations defined from the explicit extra-functional

requirements, the type returned by the operation should have an “undefined” value. The motivation for such operations is that for components that are able to provide such information at run-time, the information should be accessed in a uniform and testable way. For those that cannot provide the information, the corresponding operation simply returns the “undefined” value. The client can then decide how to react to this information.

Furthermore, the testable interface could offer operations which support test isolation. These operations indicate to the infrastructure whether the component is sensitive to testing (that means test cases and business functionality cannot be performed at the same time because of the risks destroying the component state) or whether it supports the execution of test cases in parallel with the business functionality, e.g. by offering test sessions or a specific clone operation.

In this example it would make sense to define the following operations in the testable interface of the auction house:

Logical Attributes

setActiveAuctions()
getActiveAuctions()

Logical State

setIdle()
isIdle()
setActive()
isActive()
isInService()

Extra-Functional Requirements

getAllocatedMemory()

Configuration Settings

getMaxAuctions()

Test Isolation

cloneAuctionHouse()

These methods are first class citizens of the component interface, and thus need to be added to the specification views developed in the previous subsections. Each operation needs an operation specification, and should appear in the behavioural and structural views of the component type. The specification of the extended interface is known as the extended specification of the component. For example, Figure 8 shows the specification of the setActive() operation.

Name	setActive()
Description	sets the component to the Active state
Results	the component instance is in the Active State

Figure 8 Operation specification for setActive state.

The extended structural view is shown in Figure 9.

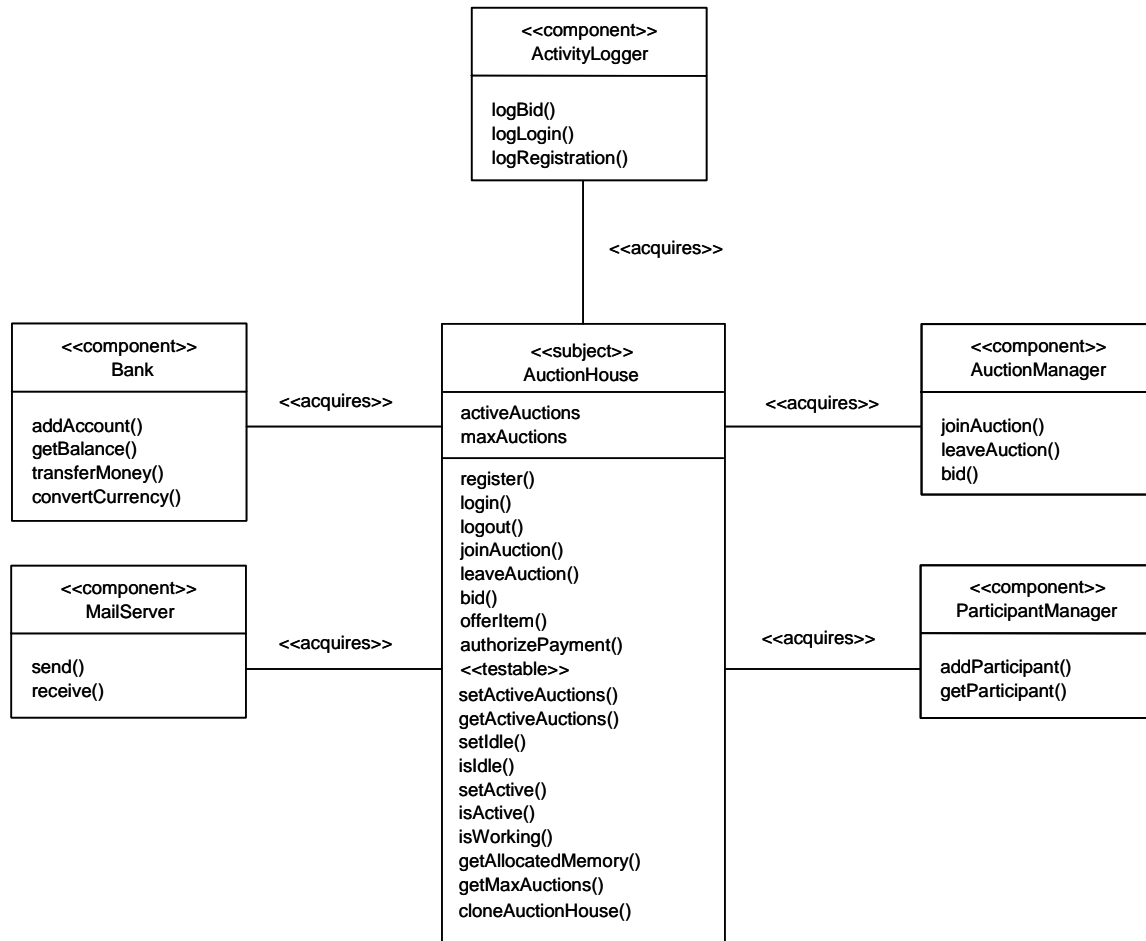
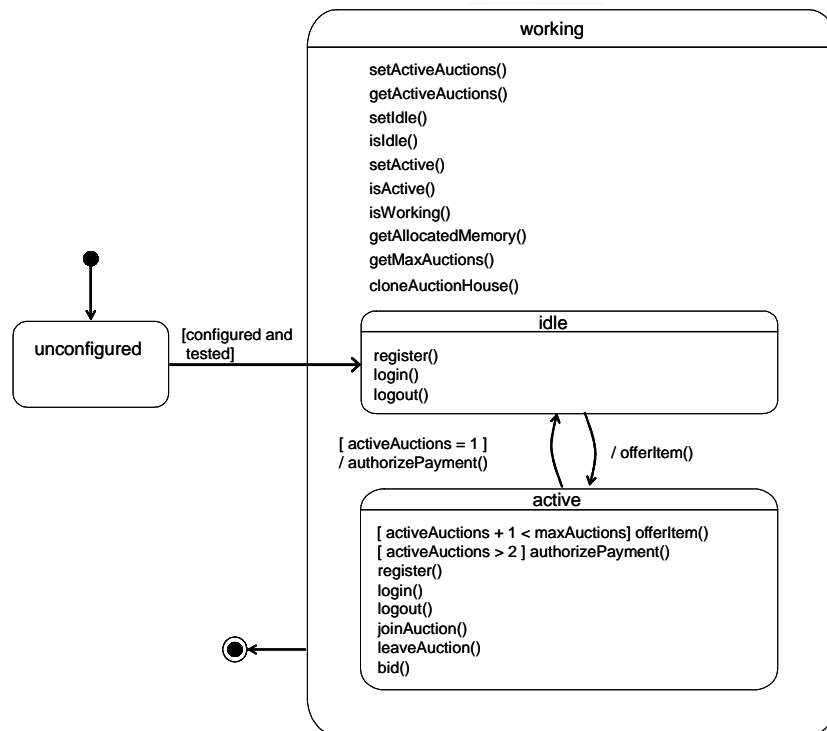


Figure 9 Extended Structural View

The extended behavioural view is shown in figure 10.



2.5 Usage Profile Definition

An important part of the specification of a component is its expected usage profile. This is composed of two parts. One part is a specification of the relative execution frequencies of the operations offered by the component, the other is a specification of the expected distribution frequencies of the values of the parameters of the operations.

The simplest way to represent the usage profile is to extend the behavioural and functional views identified above. The behavioural view can be enhanced to show execution frequencies by showing the relative probability of each of the exit transitions from each state. The sum of the probabilities of all exit transitions from a state must equal 1. This is shown in Figure 10 below.

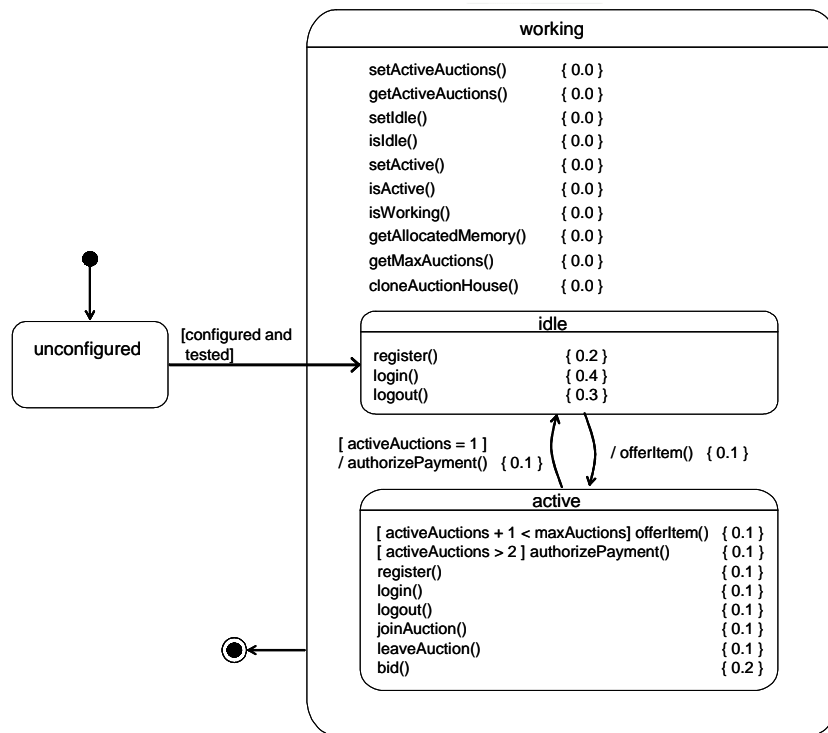


Figure 10 Usage Profile – State Machine Form

The assumed parameter value distribution is best described by adding additional information to the “receives” fields of the operation specifications, as illustrated in Figure 11. In this figure, the bracketed text after each parameter type defines the assumed distribution. Thus, sessionId is a String parameter with Strings values distributed uniformly and bid is parameter of type Money with values distributed according to a Poisson distribution.

Name	bid()
Description	
Constraints	the executing bidder is registered at the AuctionHouse
Receives	sessionId : String (Uniform); bid : Money (Poisson);
...	...

Figure 11 Extended Specification of the AuctionHouse’s bid() operation

2.6 Extra-Functional Requirements Definition

A component specification will usually have numerous extra-functional requirements in addition to its functional requirements. In general, these can simply be stated in the requirements document alongside the functional requirements discussed in the preceding subsections. In the context of MORABIT one of the most important extra-functional requirements is the reliability or equivalent the failure rate. One way to state such a requirement is to give the maximum allowed failure rate for each method, as illustrated in Figure 12 below.

operation	failure rate
ParticipantManager	
addParticipant()	0.002
getParticipant()	0.001
MailServer	
send()	0.10
receive()	0.02
sendTestMail()	0.20
ActivityLogger	
logBidder()	0.005
logLogin()	0.005
logRegister()	0.005

Figure 12 Reliability Requirements

Alternatively, it is possible to define the probability of failure on demand (POFOD) [5] for each method, This values gives the probability that any particular invocation of the method with fail.

If the POFOD and/or the likely failure rate has been defined and the relative execution frequencies have been determined, corresponding POFOD and failure rates can be calculated and specified for the component as a whole. For example, the following requirement states that no method invocation (of any method) must have a POFOD of more than 0.001

“The POFOD for any method invocation must be less than or equal to 0.001”

2.7 Functional Test Case Design

Once the core views have been developed there is enough information to develop the functional test cases included in the component’s extended interface. These are used in the development-time testing of the component. These test cases are developed according to the usual unit test adequacy criteria like structural specification coverage (e.g. coverage of the behavioural view) or error-based testing (focusing on possible output defects, e.g. input space partitioning or boundary analysis) [6]. Risk-based test case design can also be applied [7]. White-box test cases or fault-based test cases (focusing on possible software defects) are developed when the component has been coded (in Step 3.4).

3 Architectural Design

In this phase, the architecture of the component is developed and the algorithms used to realize the component’s provided interface are designed. In the KobrA method, the design of a component is known as its realization, and is documented using three views: the structural view which enhances the specification structural model with the additional information created during design including possibly new required services, the interaction model which illustrates how the component interacts with its sub-components and service supplier components to realize each of its operations, and an activity model which shows the (sub-) activities involved in realizing each operation.

This phase is not significantly affected by concerns for run-time testing. This phase can therefore be carried out according to the normal responsibility-driven design principles [8] advocated in the KobrA method and described in standard OOAD books such as Larman [9].

3.1 Structural Design

The realization structural view (illustrated in Figure 13) of the component type provides a complete picture of the components structure and dependencies. It includes the testing interface, the required interfaces and all the sub-components and classes that are needed to realize the component types.

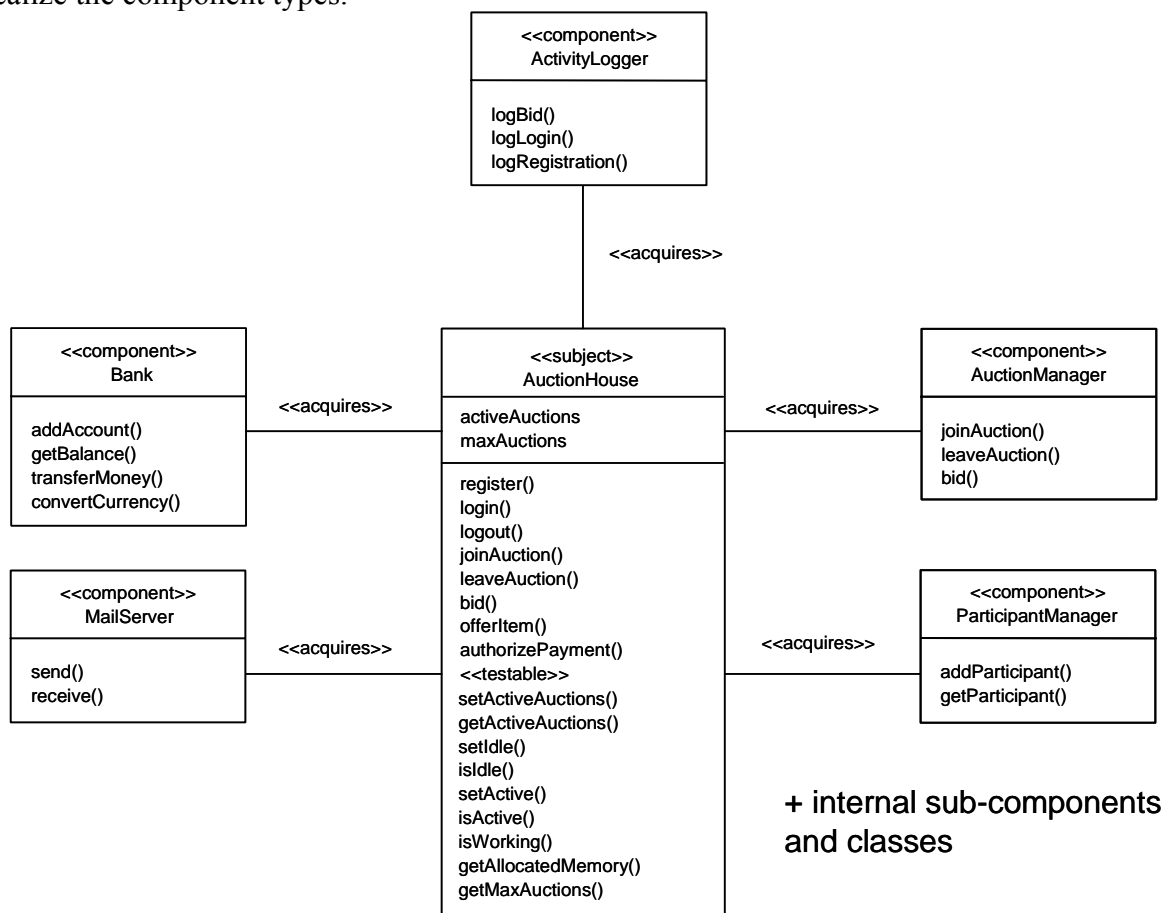


Figure 13 AuctionHouse Realization Structural Model

3.2 Interaction Design

The interaction view describes how each operation is realized in terms of interactions between an instance of the component type under development and the other components / objects. It takes the form of a UML interaction diagram of the form illustrated in Figure 14. This shows how the AuctionHouse's register operation is implemented in terms of interactions with instances of ActivityLogger, ParticipantManager and MailServer component types.

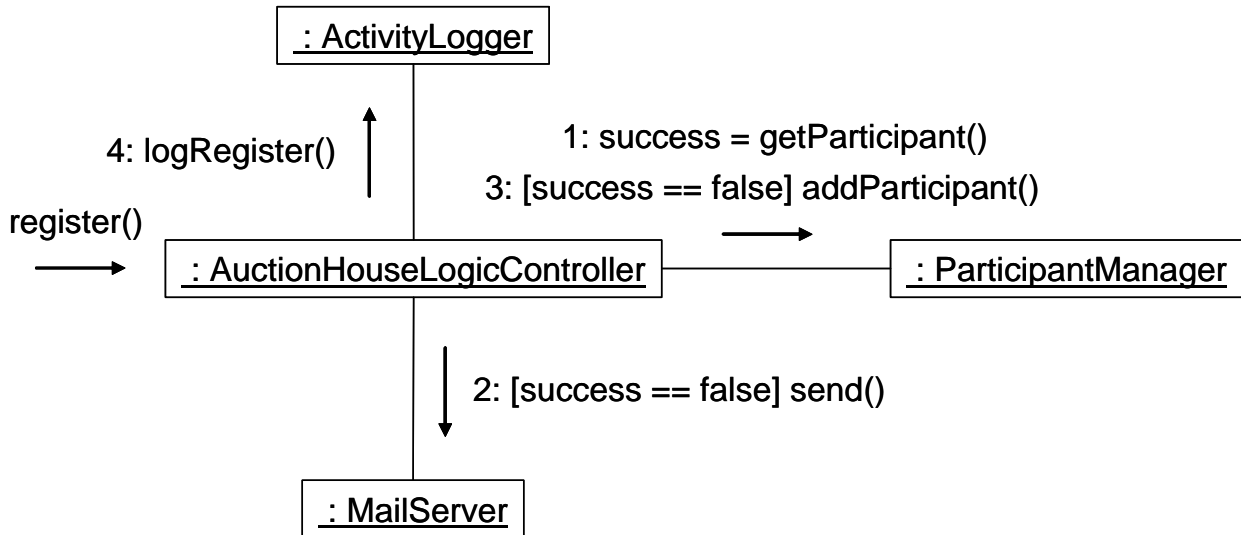


Figure 14 register() Interaction Diagram

3.3 Activity Design

The activity view provides an alternative perspective on the algorithms used to implement each operation. Instead of depicting each substep of the algorithm in the form of a message to another component or object it depicts it as an activity in an activity diagram. Figure 15 below shows the algorithm for the register operation in the previous example but in the form of an activity diagram.

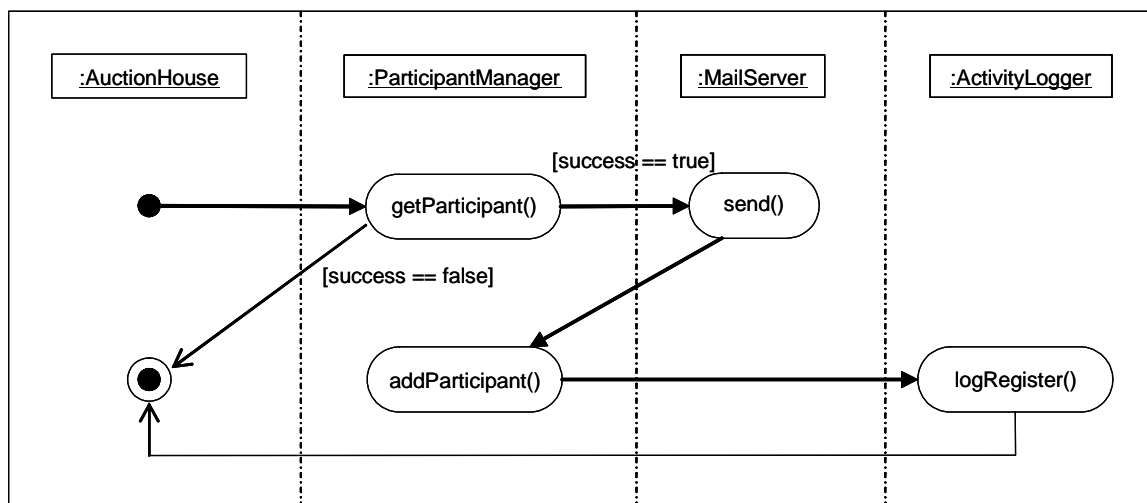


Figure 15 register() Activity Diagram

3.4 Structural Test Case Design

Once the activity view of the component has been developed there is enough information to develop the test cases based on structural coverage or fault-based testing criteria (e.g. through mutation analysis). Again risk-based testing focusing on software defects can be applied. The identified test cases augment the functional test cases developed in Section 2.7. They are also used in the development-time testing of the component.

4 Test and Countermeasure Design

The previous steps have focussed on describing and designing the services offered to clients of instances of the component type under development, including the services provided to support run-time testing. In this step, the component's use of the MORABIT infrastructure to test the quality of its servers is designed. In other words, the component type's *run-time testing strategy* is defined. An instance of the type applies this strategy at run-time to test itself and/or its required servers. In short, it determines *why* the execution of a test is needed (risks), *what* kind of test should be performed (contract test and self-test), *when* it should be performed (test timing), and *how* the infrastructure or the component type should react (countermeasure design).

4.1 Risk and Cost Analysis

Component types make use of the services of other components in order to deliver their own services defined within their public provided interface. The interaction between a component type and its servers takes place via contracts which specify each party's rights and obligation. In the MORABIT methods, contracts are defined by component specifications of the kind outlined in Part 2. The component types' contract with its client's is defined by its own specification, and the contract with its servers is defined by their specifications. We therefore assume that every server used by a component type has been specified according to the approach described in Part 2 and includes a testing interface.

The failure of a server to fulfil a contract, as perceived by the component type, can have various consequences ranging from nothing of particular concern to the complete inability of the component type to fulfil its own offered services. The goal of the first substep is to analyse the likelihood and consequences of the failures of the contracts in which the component type is involved – both as a provider and a user. The identified risks are balanced against the cost of detecting the risk and reacting to it at run-time. Based on a trade-off analysis test cases are designed for the failures with the highest risk-cost ratio.

4.1.1 Server Analyses

Initially, a risk and cost analysis is performed for each contract that the component type under development has with a server. The four key questions that must be answered when analyzing a contract are as follows:

1. what are the types of contract failures that could occur and what is the likelihood of each type occurring?
2. for each failure type, what is the consequence of that failure occurring in terms of the component type's ability to provide its own services and in terms of possible effects on the overall system in which an instance of the component might be deployed?
3. what are the estimated costs of detecting this failure in terms of the ability of the component or the system to deliver its service?
4. what countermeasures are feasible (a) in terms of changes to the realization developed in Section 3 and (b) in terms of changes to the configuration of the system in which an instance of the component type might be deployed?

Examples:

- A. Risk: The mail server does not meet its functional contract, e.g. delivers the mail to the wrong address
 - 1. Low likelihood because this is standard functionality
 - 2. Failure can be serious for high priority mails
 - 3. Estimated cost of detection is low - could be checked when the component sends a mail to itself via the mail server
 - 4. Another mail server could be chosen

- B. Risk: The mail server does not meet its reliability contract
 - 1. Fairly high likelihood because of high load
 - 2. Failure can be serious for high priority mails
 - 3. High estimated cost as this requires quantitative testing
 - 4. Use an alternative e-mail server for high priority mails

- C. Risk: The AuctionManager does not meet its reliability requirement
 - 1. Reasonable chance of occurrence due to network problems
 - 2. Failure is very serious, will stop auctions from being started
 - 3. High estimated cost as this requires quantitative testing
 - 4. Try to find a different auction manager

4.1.2 Self Analysis

Once this analysis has been performed for each of the required interfaces of the component types under development, the analysis is performed for the component types' provided interfaces as follows:

- 1. what are the types of component failures that could occur and what is the likelihood of each type occurring?
- 2. for each failure type, what is the consequence of that failure occurring in terms of the possible effects on the overall system in which an instance of the component might be deployed?
- 3. what are the estimated costs of detecting this failure in terms of the ability of the component or the system to deliver its service?
- 4. what countermeasures are feasible in terms of changes to the configuration of the system in which an instance of the component type might be deployed?

Example:

- A. Risk: Failure of the AuctionHouse to meet its reliability contract (i.e. deliver its functionality to the required level of reliability)
 - 1. Reasonable chance of occurrence due to network problems
 - 2. Failure of the AuctionHouse will reduce the reliability of the whole system as perceived by users
 - 3. Cost is high since it requires a quantitative test
 - 4. Shut down the system

4.1.3 Ranking

The risks from both server and self analysis are ranked according to their likelihood of occurrence and their potential impact taking into account the cost of detection and reaction. Thus, in the example the self-test of the auction manager failure and the contract test of the mail server functionality would be ranked highly since their likelihood and impact are both high, but the cost manageable. Also, despite the high cost, the testing of the auction managers' reliability is ranked highly because of the high impact.

4.2 Contract Test Case Definition

In this step contract test cases (CTCs) are defined for all the high ranked contract risks that the component type under development is involved in, both as a server and as client (i.e. both for the provided and required interfaces). The aim of these CTC's is to uncover at run-time a failure in a service-provider's ability to fulfil the contract expected by one of its client.

There are two basic criteria for defining CTC's depending on whether quantitative or qualitative information is required from the execution of run-time tests. Qualitative tests return binary (pass/fail) values depending on whether or not a component instance fulfils a contract as understood by the component type under development. This form of run-time test is therefore the most basic and is the most widely applicable. Quantitative tests return a numeric measurement of the level of reliability to which a component instance implements a contract as understood by the component type under development. This type of run-time test is therefore only needed if the contract has an associated reliability measure. Both kinds of tests can be defined for both kinds of services (provided and required) as explained below.

Both forms of testing rely on an ability to determine whether a particular invocation of a component's service succeeds or fails from the perspective of the invoker. There are three basic ways in which such an invocation can be judged to have failed

1. the operation completes, but returns a value that was not the expected one,
2. the operation does not complete and returns some indications to the caller that it was unable to do so (e.g. raises an exception),
3. the operation does not complete within a required period of time.

In principle, all three forms can be used in both quantitative as well as qualitative testing. However, since forms (2) and (3) do not require an expected result to be determined, they are particularly suited to quantitative testing. The creation of expected results for the first form of failure has traditionally been one of the biggest stumbling blocks to quantitative testing because it is difficult if not impossible to do automatically.

4.2.1 Quantitative Contract Test Case Definition

Quantitative contract test cases are used to determine, to a given level of confidence, whether a component instance delivers its service with a level of reliability that is lower than a given threshold. They can be designed to determine the reliability of the servers used by an instance of the component type being developed, or to determine the reliability of instance of the component type itself. A prerequisite for quantitative tests of a server is an extended specification which includes usage profile information of the kind outlined in Section 3. We therefore assume that all provided services have been specified according to the approach.

To attain a reasonable level of confidence in the reliability bound, quantitative tests usually require many more test cases than qualitative tests. The minimum number of test cases that must be executed is a function of the reliability threshold and the confidence level which is desired, as explained in Section 4.2.1.2.

4.2.1.1 Reliability Bound and Confidence Level Determination

The first step in defining quantitative tests is to decide what the reliability bound and the level of confidence should be. In some cases this may be “obvious”, but more often it is necessary to perform some kind of dependency analysis which relates the desired reliability of the component type under development to the required reliabilities of its servers. This reliability may be defined in the specification or may be determined by the developers based on judgments about the application domain and market conditions.

Once the required reliability bound of the component under development (or one of its operations) has been determined, corresponding reliability bounds for its servers can be established by analyzing the algorithms used to realize the operations of the component type. This involves two main substeps –

1. estimate the usage profiles of the component type’s servers
2. estimate the required reliability bound and confidence level

The first of these substeps is in turn achieved by performing the following three substeps –

1. for each of the operations in the component type’s functional interface, assign probabilities to the edges in the operation’s activity diagram defined in the Architectural Design phase. This results in a Markov chain for that operation.
2. for each such diagram, calculate the execution probabilities for each of the operations of the component type’s servers.
3. aggregate the results from (2) to calculate the overall execution probability of each operation of each server .

Figure 16 shows these steps applied to the register() operation of the AuctionHouse type.

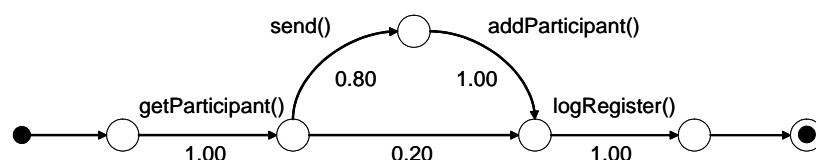


Figure 16 Markov Chain for register() algorithm

This gives a way of estimating how a server is used by an instance of the component type (i.e. each server’s actual usage profile from the point of view of the component type under development). The second step is then to use this information to derive reliability and confidence level bounds for the servers given a reliability and confidence level for the component type under development (or one of its operations). In other words, if a given operation of the component type is required to have a reliability of at least r with a confidence level of c , and given the frequencies with which the realization of this operation invokes

operations of the required servers, the minimum reliability and confidence level needed for each server can be calculated.

4.2.1.2 Test Case Definition

When a minimum acceptable reliability and confidence level has been decided for a component, the actual tests cases needed to establish whether a component delivers this minimum level of reliability can be defined, assuming that the usage profile is known. This involves two steps.

1. **Determine the number of test cases.** Following [10] the minimum number of test cases can be easily calculated by applying the formula $n \approx \ln(1 - c) / \ln(1 - f) \approx a / f$ where c is the desired confidence and f the software's probability of failing. The reliability can also be written as $r = 1 - f$. The formula can even be simplified when the number of test cases is at least 100. Then the Poisson distribution can be used where a is the parameter of the distribution.
2. **Design test cases.** In this step the required number of test cases are created. This is done by defining appropriate sequences of operations which invoke the operations of the interface with the relative frequency and the parameter value distributions defined in the usage profile. In addition, operations must be invoked with a prior state distribution which matches that in the usage profile.

4.2.2 Qualitative Contract Test Case Definition

The goal of qualitative CTC's is to uncover mismatches between the component type's understanding (i.e. interpretation) of its contracts and those of the component instances to which it is connected at run-time. The CTC's are therefore designed to maximize the chances of uncovering contract understanding mismatches at run-time. Qualitative test cases for the component's *provided* service (self-test) essentially correspond to the test cases defined in the functional and structural test case design for defect testing. Since it is assumed that the component has already been tested during development-time to detect coding , the focus is on test cases which reveal problems in the component's behaviour due to dependencies on its environment. Thus, according to the risks identified in the self analysis, test cases from the functional and structural test case design are selected which can uncover this risk. Another option is to use self-monitoring instead of self-testing. This can be realized in two ways. Either the components' code is changed so that it gathers information about its behaviour continually or – as advocated by other BIT approaches like [11] – there is a separate monitor component which collects the information about the component under test. Altogether, if the test designer wants to enable the component's self-test, he can either extend the component with test cases to be triggered by the component itself or add some code to the component which realizes monitoring.

Qualitative CTC's for the component's *required* services are intended to uncover potential inconsistencies between the component's interpretation of the contract and the run-time server's interpretation. In general, one can distinguish *static inconsistencies* due to design, implementation, and maintenance errors from *dynamic inconsistencies* due to resource errors, code corruption (e.g. through a virus), environmental incompatibilities, configuration errors, and faults. Only dynamic inconsistencies such as resource errors or configuration errors are the focus of BIT. In dynamically built systems that cannot be fully assembled at development-time, dynamic inconsistencies can also arise due to *incompatibilities between components*. A

component might not deliver its expected service because there is a misunderstanding with the client component. These errors are the focus of contract testing. The misunderstandings concern the contract between the components in terms of inputs, outputs, states, exceptions, and quality metrics. In essence, therefore, qualitative run-time tests are driven more from the perspective of validation than of verification, since the goal is to check whether one component meets another's expectations rather than its specification.

In the following we describe typical contract failures which need to be checked. We have established a taxonomy of possible factors based on the failures lists of Beizer [12], Kaner [13], and Binder [14].

4.2.2.1 Contract Misunderstanding Problems

Suppose the client component A and the server component B interact with each other at run-time. Component A and component B may not share the same understanding of the offered or required functionality for various reasons.

4.2.2.1.1 Syntactic misunderstanding

Syntactical misunderstandings are caught by the compiler in a static type system (where components are composed at deployment-time). For example: component A requires a component that offers the following operation:

```
transferMoney(int, int, double amount)
```

whereas component B offers the operation

```
transferMoney(String, String, double amount).
```

Such syntactical failures do not occur within the MORABIT environment since the components are implemented in a strongly typed language (Java) and a syntactic lookup is used by the MORABIT infrastructure. Thus, we will not consider such failures further in this document.

4.2.2.1.2 Semantic misunderstanding

Failures can also arise due to semantic misunderstandings between the components. We extend the defined failure categories presented in deliverable M2 with some heuristics to detect such failures. For each subcategory we present the definition, an example from our case study and a heuristic to detect the failure. The reaction to a failed test always depends on the criticality of the operation. Typically one should try to find another server. If there is no adequate server, either the component adapts its behaviour or shuts down (see discussion in 4.3.).

4.2.2.1.2.1 Input parameters

This includes:

1. Order of input parameters:

Definition: A component invokes an operation of another component assuming a different order of the parameters than the called components. This happens quite often especially when the parameters are of the same type.

Example: The AuctionHouse component invokes the transferMoney() operation of the Bank component with the arguments (to, from, amount) instead of (from, to, amount).

Heuristic: Define test cases for ALL operations where swapping is likely because parameters have the same type. Try all possible permutations of the parameters. If the test result is correct, then use this permutation for all future calls.

2. Invalid input values:

Definition: Parameters are initialized with invalid values

Example: The bank invokes the currency converter with invalid currency names.

Heuristic: Define test cases that call the operation with boundary (unusual) values. That means values that are not common for all components.

Different input interpretations:

Definition: The client has different assumptions about the intention of the input parameters .

Example: The participant inputs his real name to join an auction instead of the user name.

Heuristic: Define a test case that invokes an operation which uses this assumption, e.g. a test case that invokes the operation getName() of the AuctionHouse component. If no such operation is available, test the whole process, e.g. start a test auction and invoke joinAuction(name). If there are only a few alternatives, try all of them and keep the right alternative for further use.

4.2.2.1.2.2 Output parameters

A component invokes the method correctly, but the output parameters do not conform to the expectation of the component because:

1. Order of output parameters:

Definition: Similar to wrong order of input parameters, but with output parameters.

Heuristic: Define test cases for ALL operations where swapping is likely because parameters have the same type. Try all possible permutations of the parameters, by using an internal operation to check which output parameter has which expected value. If the test result is correct, then use this permutation for all future calls.

2. Invalid output values:

Definition: The output does not comply with client expectation

Example: The answer accuracy of the convertCurrency() operation of the CurrencyConverter is up to 2 digits accuracy whereas the Bank component expects 3 digits accuracy.

Heuristic: Define a test case for each operation before it is used.

3. Different output interpretations:

Definition: The caller component misinterprets the result. This will be the general case where the output is different from what the caller component expects.

Heuristic: As for invalid output values. Clearly, it is not possible to detect the problem at run-time in general. Only, if there are some well-known variants of the computation which the caller could check.

4.2.2.1.2.3 State

The component B sends an unexpected response since the state of B is not as A expects. This includes:

1. Violating the control state of the invoked component:
 Definition: Calling an operation that is not allowed in this state
 Example: The participant component invokes the operation bid() of the auction component without joining an auction
 Heuristics: Define a test case that includes a chain of activities
2. Violating the data state of the invoked component:
 Definition: Calling an operation with data which is not valid in this state.
 Example: Component Auction house uses an account number when calling transferMoney that is not valid at the Bank component.
 Heuristic: It is possible to define test cases for that, but the data state is of course dependent on the previous operation calls. Furthermore, the tested component will typically avoid to give information on its data state to the outside because of security reasons.

4.2.2.1.2.4 Exception handling

Definition: The client and the server have different understanding of exception handling.

Example: The bank checks how the currency converter reacts to a negative amount.

Heuristic: Define a test case that invokes the operation with wrong values and check the reaction.

4.2.2.1.2.5 Quality defect

Definition: The invoked component does not fulfil the quality expectations of the caller component.

Example: The response time of the auction house component for bidding is too long for the ParticipantManager component.

Heuristic: This failure can be detected with a test case that checks whether the invoked component fulfils the quality expectation. E.g., a test case that checks whether the auction house component responds within a given time interval. However, the quality is dependent on the environment so that no general conclusions can be drawn from single test cases.

4.2.2.1.2.6 Side effects

Definition: These are failures that are neither in the caller component nor in called component, but in the partner components. They usually cannot be detected directly.

Example: the component participant subscribes as a buyer to an AuctionHouse component. The AuctionHouse component sends a registration request to the ParticipantManager component, receives a confirmation, and finally sends a confirmation to the client. But the participant is not registered since the database component of the ParticipantManager component is corrupt. The client component will detect this failure only when it tries to bid within the auction.

Heuristic: Construct a test case that invokes a chain of operations. In our example the test case should invoke the operations register and bid.

4.2.2.2 Service Implementation Problems

This means a server does not fulfil its own interface specification. These problems can be the reason for every failure. We only distinguish two cases. Neither of them can be detected at run-time.

4.2.2.2.1 Concurrency defect

The required functionality is implemented correctly, but compromised by other functionalities that are provided by the component. In other words: the tested functionality returns correct results only if certain other functionalities are not running concurrently. We cannot usually detect such failures at run-time. Component A may test component B periodically and detect the failure by chance, but we cannot design a *specific* test case for such failures since the client does not know which other functionality may be involved. However, such failures can be detected by self-tests or self-monitoring.

4.2.2.2.2 Internal defect

The component B is implemented incorrectly. These failures are not treated at run-time within MORABIT since we assume that the component is sufficiently tested before its release.

4.2.2.3 Ressource problems

The resources consists of the devices on which the components run, operating systems or middleware used, and software libraries involved. Typically, for any reconfiguration of the resources regression testing is needed. Test cases can be developed to test specific resource dependencies. These can be applied as self-tests. Similarly, self-monitoring is helpful here. The client component cannot apply specific test cases as the resource dependencies of its servers are not known.

4.2.2.4 Summary

The discussion above can be summarized in the following table which relates possible failures to kinds of tests. It is important to keep in mind that usually the component cannot detect the cause of a failure. So, the test designer can design the test cases to minimize the likelihood of a certain failure (by checking for that failure). If the test passes, it has only limited evidence that the failure will not occur in the future. If the tests fails, the cause might be just anywhere. This has to be reflected in the countermeasure design in Section 4.3.

Failures	Additional self-test or monitoring component	Run-time test cases by client
Syntactical misunderstanding	no	no
Input misunderstanding	no	Specific test cases, boundary test cases
Output misunderstanding	no	Specific test cases, boundary test cases
State misunderstanding	no	Test sequences
Exception handling misunderstanding	no	Negative test cases
Quality misunderstanding	no	Quality tests
Side effects	no	Test sequences
Service implementation problems	yes	no
Environment problems	yes	no

Table 1 Qualitative Test Cases

4.3 Countermeasure Design

Once the information from the risk and cost analyses is available, it is also possible to decide if the potential countermeasures identified for each risk can be realistically realized and how. Unlike testing at development-time no “debugging process” can be performed, as neither the developer of the component nor its source code is usually available. Even if the source code is available, it is unrealistic to expect that the user of the component has enough resources (time and/or knowledge) to eliminate the detected failure. Instead, instances of the component type under development must take some form of countermeasure in the case of unfavourable test results. There are two basic ways of defining countermeasures – one way is to use the predefined countermeasures built into the infrastructure, the other way is to define a new custom countermeasure and to integrate them into the component.

4.3.1 Predefined Countermeasures

Predefined countermeasures are generic reactions that are carried out by the infrastructure and are therefore independent of specific business logic. They might involve changes at the system-level such as a reconfiguration of the component instances making up the system. Examples include (but are not limited to)

- **Shut down:** the infrastructure takes care of either shutting down a component (taking this component out of service) or the whole system (stopping the system).
- **Try next component:** the infrastructure should attempt to find an alternative provider of the service required by the instance of the component type under development.
- **Choose best component:** the infrastructure should establish which of the available servers is “best” and should return this one.

Since they are performed at the system-level, predefined countermeasures can be used to respond to failures of a component instance to fulfil its own provided contract (self-test) as well as to the failure of other component instances to meet a given component instance's expectations (contract test).

Examples:

1. The AuctionManager does not meet its reliability requirement
 - a. Response: Try next component
2. Failure of the AuctionHouse to meet its reliability contract
 - a. Response: Shut down the system

4.3.2 Custom Countermeasures

In addition to the predefined countermeasures it is also possible to define custom countermeasures which allow a component type to implement its own application specific responses to a (contract) failure. In other words, one or more of the algorithms used to realize operations of the component type can be altered to take into account information derived from the run-time tests. However, the use of such custom countermeasures should in general be minimized since it takes away some of the component user's control over the configuration of the component and its tests via test requests. It also reduces the overall visibility of the component type's behaviour. However, in some cases countermeasures can only be implemented using custom countermeasures.

Custom countermeasures are supported via the notion of "reaction modes". A reaction mode identifies a behaviour variation point which the component type uses to enact application specific countermeasures to a contract failure. Each reaction mode is coupled to a specific run-time test request, and is set by the infrastructure based on the result of the test (to either true or false). The component can then ask the infrastructure for the values of the reaction mode and perform the custom countermeasure. Thus the infrastructure is aware of when and why a particular custom countermeasure is triggered within a component instances, but is not aware of the actual nature of the countermeasure.

For example, the reaction mode "primary mail server failure" might be defined for the AuctionHouse component type in association with a failure of the primary mail server to meet its reliability. When the infrastructure detects such a failure by executing the appropriate test at the appropriate time, it will set the "primary mailer server failure" reaction mode for the AuctionHouse instance to false, thereby allowing that instance to vary its behaviour accordingly.

Whenever a custom countermeasure is added to an operation of the component under development, the corresponding diagrams developed in architectural design (see 3.2 and 3.3) must be updated to reflect the new algorithm.

Examples:

1. An AuctionHouse's primary mail server does not meet its reliability contract
 - a. Reaction Mode: "primary mailer server reliability failure"
 - b. Custom Countermeasure: use secondary mail server for high priority mail.

4.4 Test Strategy Design

The previous substeps have identified failure risks that could affect a component's ability to interact with its servers and to fulfil its contract to its clients. The tests that can be performed to uncover the corresponding problems and the countermeasures that can be taken to reduce the risks have been identified in previous steps. In this step all this information is brought together and decisions are made about which test requests should actually be executed when and with what countermeasures.

4.4.1 Self versus Server

As mentioned previously, it is possible for a component type to arrange for its own instances to be tested at run-time to check that they are fulfilling the promised contract in the context of the particular run-time environment in which they are running. Such run-time tests are referred to as self-tests. The first decision that needs to be made in test strategy design, therefore, is what mixture of self-tests and server tests is optimal.

Self-tests have the advantage that they can measure the overall behaviour of component instances, without knowledge of the details of how it is realized. Such tests have the flavour of system tests therefore. Indeed, if the component type under development essentially plays the role of "the system" because it is the outermost encapsulating component (like the AuctionHouse), then self-tests are essentially automated system tests.

The main advantage of server tests is that weaknesses in the dependencies of the component on its servers can be identified. If a self-test indicates a failure, it is not possible to identify the causing dependency. With server tests, however, it is possible to identify the failing dependencies and thus provide countermeasures specific for these dependencies.

4.4.2 Test Timing Determination

Once the type of test has been determined, the final step is to decide when a contract test should be executed in relation to the lifecycle and behaviour of the component type under development. As with countermeasures, there are two basic types of test timing strategies – predefined, infrastructure controlled timing and customized timing.

4.4.2.1 Predefined Timings

Eight distinct execution times are recognized by the MORABIT infrastructure-

- **Creation time tests:** executed when a new server of an instance of the component type under development is created.
- **Connection time tests:** executed before an instance of the component type under development is connected to a server, either via a lookup operation or via dependency injection.
- **Call time tests:** executed when an instance of the component type under development invokes an operation of one of its server components.
- **Periodic tests:** executed on a periodic basis.
- **Random time tests:** executed at random time intervals.
- **Idle time tests:** executed when an instance of the component type under development enters the idle state.
- **Topology change time tests:** executed as soon as the topology changes (i.e. certain components are replaced by others with the same provided interface). The test cases to

be executed depend on the servers which are affected by the topology change. Topology change time is analogous to connection time, but takes place once the component under development has been put into service (i.e. during the in-service phase of its life-cycle).

The choice of test time is mainly driven by the precise way in which the component type under development uses the required interface, and also depends on the nature of the intended countermeasure. However, certain combinations of test types and test times do not make sense and are ruled out as illustrated in Table 2.

		Server	Self
Test Timing	Creation	+	-
	Connection	+	-
	Lookup	+	-
	Call	+	-
	Periodic	+	+
	Random	+	+
	Idle	-	+
	Topology change	+	+

Table 2: Test execution times

Table 3 shows what kind of testing times are best suited to what kinds of predefined countermeasures. The last column with “custom” is grey because such a countermeasure can be implemented in any way chosen by the developer. “Topology change” is left out since this is more or less a combination of the others.

		Shutdown	TryNext	ChooseBest	Custom
Test Timing	Creation	-	+	+	+
	Connection	+	-	-	+
	Lookup	+	+	+	+
	Call	+	+	+	+
	Periodic	+	+	+	+
	Random	+	+	+	+
	Idle	+	+	+	+

Table 3: Test reactions

4.4.2.2 Custom Timing

When the developer has chosen to design a customized countermeasure to a run-time test it may also be useful to define a customized time for the tests. When the best reaction to a test is so tightly tied to the algorithm used to implement an operation then it is often the case that the best moment to execute the test is also intimately tied to the algorithm. To support this, the infrastructure allows an application component to issue a synchronous request (system call) for a test associated with a custom countermeasure to be executed. The component can then obtain the results of the test using the corresponding reaction mode explained in Section 4.3.

The use of custom timing has the same disadvantage as the use of custom countermeasures – it removes control and oversight of the testing process from the component user and from the



MORABIT infrastructure. However, for certain kinds of components, it represents a more understandable way of describing the overall behaviour of the system than of trying to artificially tease apart tightly coupled concerns.

4.5 Test Specification

The final step of the test and countermeasure design step of the MORABIT method is to document the decisions made in the preceding subsections in a form that the MORABIT infrastructure can understand and act upon. This is done by creating a MORABIT test request which collects together all the required information for the infrastructure including test cases, countermeasures and test strategies. More details on how test requests are specified in MORABIT are provided in the MORABIT Infrastructure Specification [15].

5 Detailed Design and Implementation

Once the strategy for using the run-time testing capabilities offered by MORABIT has been determined through step four, the design of the component type under development can be finalized accordingly. In general this involves three steps.

5.1 *Architecture Revision*

If the component type under development includes a custom reaction and/or customized test timing, it must make the appropriate “system” calls to the MORABIT infrastructure. In effect, this means that the MORABIT infrastructure becomes an additional server of the component type under development which is explicitly called from at least one of its operations. To keep the architecture views developed in the Architecture Design phase in sync it is therefore necessary to update the affected diagram(s) and algorithm(s). This will always involve a change to the structural view, since the MORABIT infrastructure needs to be shown as an explicit server of the component type under development, and also the interaction and activity diagrams of the operation realization(s) that use the infrastructure. There must be at least one invocation of the infrastructure, otherwise the component type under development cannot request a custom time test or get the information it needs to perform a custom countermeasure.

5.2 *Implementation Modelling*

Once the architectural (or high-level design) models have been brought up-to-date, the detailed design work can be performed. This is performed in the same way as any ordinary software development project. Typically it will involve the creation of detailed UML structural models showing the programming languages constructs that will be contained in the final implementation. Sometimes pseudo code is also used to provide a more detailed description of the algorithms to be implemented.

5.3 *Implementation*

The final step is the creation of the actual source code in the target programming language. If a sophisticated UML modelling tool was used to create the implementation models in the previous step, much of this can be performed automatically by the tool.

6 Development-Time Testing

In this phase the typical development-time testing activities are carried out. This is performed using the test cases identified in functional and structural tests designed in Sections 2.7 and 3.4, respectively. Since this is performed using established techniques and is not changed by the run-time testing concerns discussed in this document, we do not elaborate here. Suffice it to say that there are four main sub-activities.

1. Test case selection based on traditional testing criteria
2. Definition of “stand-in” required server components to act as stubs for the required servers at run-time. These may either be fully functional implementations of the required services respectively interfaces or stubs generated just to support this testing process (e.g. mocks)
3. Test execution
4. Location and correction of defects

At the end of this process the component is assumed to be correct with respect to the executed test cases in the context of the “stand-in” server components.

7 Packaging

The final phase in the component development process is the packaging of the component and all of the associated artefacts into a distributable format. As with the previous phase, this is not significantly affected by run-time testing issues. It essentially involves the packaging of all the appropriate artefacts (e.g. executable code, documentation, configuration files etc.) into the chosen package construct such as .jar files in the case of Java. The only thing that changes in the context of the MORABIT approach is that the test requests defined in Section 4.5 also need to be included in the package. More specifically, all the test requests for the component type's required servers and (if present) provided interface need to be included (for self-testing). Since these test requests are an integral part of component types developed using the MORABIT approach they must always be provided along with the executable code. This is the sense in which the run-time tests can be regarded as being built in to components. They are not built into run-time component instances, as such, but are built in to the component types.

PART 3: Component User Method

1 Introduction

The MORABIT Component User Method describes how software engineers should go about creating applications using MORABIT component types. A prerequisite for the use of this method is thus the existence of a run-time platform which includes the MORABIT infrastructure. At least one of the component types used to construct the application must be a “MORABIT component type” in the sense that it was created according to the MORABIT Component Development method described in the previous part of this document. However, it is not essential for every component type to be a MORABIT component type. Instances of “ordinary” component types can work with other MORABIT component types, but simply cannot make use of the MORABIT run-time testing services. However, in the following we describe the process as if all the component types are MORABIT component types. The main steps are listed below.

- 1 Component Deployment**
 - 1.1 Component Instantiation and Configuration
 - 1.2 Test Request Customization
- 2 Deployment Time Testing**
- 3 System Execution**

2 Component Deployment

The first phase of the process is to deploy all parts of the component types in the right places with the right configuration scripts and get to a point where the system is ready to run.

2.1 Component Instantiation and Configuration

The first step in the process is to unpack the executable artefacts making up each component type and install them in the run-time execution environment. In the case of Java this is the compiled binary classes (e.g. .class files). The next step is then to write any necessary configuration scripts, or code, which defines instances of component types and connects them to one another in the required starting configuration. This is sometimes referred to as dependency injection. The exact form of the configuration files and what exactly is involved depends on the particular component technology involved. If the components are POJO's then a piece of encapsulating "main program" code may need to be written to define the required application. If they are EJB's or Spring components, then the corresponding scripts must be created according to the requirements of these technologies. Also, if a component type has any configuration options, the appropriate configuration choices need to be decided for each of its instances.

Once the components have been deployed on the execution environment and the appropriate configuration scripts have been written, the components need to be made known to the MORABIT infrastructure. The details are described in the "The MORABIT Runtime Infrastructure" Guide, but basically, an XML file has to be created for each component. It contains the components name, a short description, the fully qualified class name of the implementation, the singleton property, and a list of test requests for the required servers of the component. Each test request specifies the type it is intended to test (i.e. the type of the required server of the component) and a list of test cases with their respective class names. Both, test requests and test cases again contain a name and a short description. The XML-files are then copied to the config/components directory of the respective MORABIT application. This completes the deployment of MORABIT components to the runtime infrastructure.

2.2 Test Request Customization

The original test requests built into the component types were designed based on risk analyses that were performed "out of context" – that is, without knowledge of the specific application in hand. It is quite possible that the analyses made by the developer are not optimal for the application under construction. Therefore, before they are deployed the test requests should be revised and if necessary adapted to the application in hand.

Basically, the component user must analyze each test request and decide whether the chosen component-driven and infrastructure-driven countermeasure scenarios identified by the component provider make sense in the particular application context. This means choosing any configuration options regarding the test cases with component-driven custom countermeasures (i.e. switching some of them off) and choosing the test cases with infrastructure-driven predefined countermeasures based on the needs and properties of the



specific application and of the environment. All choices are done in the context of the recommendations of the component provider.

For each test request, the component user has four options.

1. Do not deploy the test request – this is only possible for test requests featuring infrastructure-driven countermeasures.
2. Replace the test request with a new one – if a test request is either based on a totally unsuitable risk/cost analysis or embeds an unsuitable set of test cases and/or countermeasures, it can be replaced by a new one written by the component user. This obviously involves a lot of work and expertise.
3. Modify the test request – if a test request is by-and-large acceptable, but has some small things that are inappropriate, these can obviously be adapted by the component user.
4. Use unchanged – if the test request appears suitable, it can be deployed unchanged.

Test requests featuring custom countermeasures or custom test timings must be deployed since they are built into the code and the corresponding component instances will not function otherwise.

Obviously these four options involve different levels of effort and expertise. However, they also involve different levels of risk. Since they effect the ability of a component instances to fulfil its contract (or the confidence that one can have in its ability to fulfil its contract), changes to test requests, or the non-deployment of tests requests, should only be performed by expert users.



3 Deployment-Time Testing

Once the component types have been deployed (the component instances created and configured and the test requests installed) the system is ready to run. But before the system is put into service it often makes sense to run tests privately first. As explained in Section 3 of Part 1, this is referred to as deployment-time testing.

The person responsible for deployment time testing is the test administrator. Basically he has to run the system and see what happens. If the system automatically shuts down or gives back an error message, he and the system deployer need to find out what the problem is and re-perform the previous two steps.



4 System Execution

Finally, once all the previous phases have been performed and the system has “passed” the deployment time test satisfactorily the system can be put into service. At this point the only remaining human role is that of test administrator. The test administrator needs to be on hand to respond to any problems that may be detected by the MORABIT infrastructure.

References

- [1] C. Szyperski, "Component Software - Beyond Object-Oriented Programming", Addison-Wesley, 1998.
- [2] B. Meyer, "Applying Design by Contract", in IEEE Computer, IEEE Computer Society Press, 1992; 25(10), pp. 40-51.
- [3] B. Boehm, "Verifying and validating software requirements and design specifications", in IEEE Software, IEEE Press, 1984, pp. 205-218
- [4] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Müthig, B. Paech, J. Wüst, and J. Zettel, "Component-Based Product Line Engineering with UML", Addison Wesley, 2001.
- [5] I. Sommerville, "Software Engineering", 7th Edition, Addison Wesley, 2004.
- [6] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy", ACM computing surveys, 1997, 29(4), pp. 366-427.
- [7] S. Amland, "Risk-based testing: Risk analysis fundamentals and metrics for software testing including a financial application case study", Journal of Systems and Software, 2000, pp. 287-295.
- [8] R. Wirfs-Brock and A. McKean, "Object Design: Roles, Responsibilities, and Collaborations", Addison-Wesley, 2003.
- [9] C. Larman, "Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process (2nd Edition)", Prentice Hall, 2002.
- [10] W. Ehrenberger, "Software-Verifikation. Verfahren für den Zuverlässigkeitsnachweis von Software", Hanser Fachbuch, 2002.
- [11] J. Vincent, G. King, P. Lay, J. Kinghorn, "Principles of Built-In-Test for Runtime-Testability", in Component-Based Software Systems. Software Quality Journal, Springer Science +Business Media B.V., Formerly Kluwer Academic Publishers B.V, 2002, pp. 115-133.
- [12] B. Beizer, "Software testing techniques", 2. edition, International Thomson Computer Press, 1990.
- [13] C. Kaner, J. Falk, and H. Q. Nguyen, "Testing computer software", 2. edition, Wiley, 1999.
- [14] R. Binder, "Testing object-oriented systems", Addison-Wesley, 2003.
- [15] M3 IS document