MSc Distributed Computing Systems Engineering

Department of Electronic & Computer Engineering

Brunel University

Extensible framework for testing distributed object-oriented systems

Melanie Cossy

09/00

A Dissertation submitted in partial fulfilment of the requirements for the degree of Master of Science

MSc Distributed Computing Systems Engineering

Department of Electronic & Computer Engineering

Brunel University

Extensible framework for testing distributed object-oriented systems

Melanie Cossy

Supervisor:
Mr. P. Van Santen

09/00

A Dissertation submitted in partial fulfilment of the requirements for the degree of Master of Science

Abstract

This dissertation discusses the main issues involved in testing CORBA-based distributed system with client-server architectures. It outlines important testing objectives and testing techniques.

In the first part, an analysis of existing testing and monitoring tools determines advantages and drawbacks of different methodologies.

In the second part a testing methodology is proposed and the implementation of a prototypical testing framework is described. The testing framework was developed during the practical part of my dissertation. The capabilities of the framework and of the adopted testing methodology are evaluated and its architectural design is explained. The steps necessary to use the framework for testing CORBA-based systems are enumerated.

The final part dissertation summarises my gained experiences in the field of testing object oriented distributed systems and gives an outlook to further trends.

Table of contents

1 Summary

More and more areas of our environment are controlled by software. We expect the software to be correct, this means that it works as it should. This confidence only has a solid basis if the software has been tested thoroughly. To ensure software quality, it is necessary to test sufficiently – but what is meant by the word *sufficiently*? Rigorous and extensive testing is required especially for software systems that need to be highly reliable or available. But testing is expensive in terms of manpower and money. Therefore it is important that the testing process is effective in finding possible defects and efficient in performing test cases quickly and cheaply.

Software architectures have changed rapidly in the past – mainly because of new computer technologies and an unbelievable increase of available computing power. It is expected that this process will continue in the future. New forms of software architectures make new testing methodologies necessary. Software testing has to keep pace with the changing nature of the software itself.

Software systems have become increasingly important in the last few years and a general trend is to interconnect systems to share resources and information. A wide range of distributed systems has been built up and it is clear that this field will grow even further.

Testing a distributed system is complex and traditional testing techniques cannot cover the new issues caused by the distributed nature of such a system. In this dissertation the effectiveness of new testing methodologies is analysed with respect to the issues caused by the special nature of distributed systems.

2 Introduction

Testing is a skill. For any software system the number of possible test cases is immense, but practically not all test cases can be executed and a reduced number of test cases must be sufficient to find most of the software defects. Therefore the selection of the test cases to run is difficult and demanding – both experiment and experience is needed. To select test cases at random is normally not the most effective approach to testing. A methodical approach is needed and it must be defined what exactly is a *good* test case.

2.1 Test cases

There are five characteristics that a good test case should possess:

1. Representative
   A representative test case covers more than only one special feature of the system. This helps to reduce the total number of test cases required.
2. Maintainable
   A maintainable test case can be adopted when the software under test changes.
3. Extensible
   An extensible test case has the ability to scale up when the system under test grows.
4. Effective in detecting defect
   This attribute gives information about whether or not a test case may find defects.
5. Economic
   The attribute economic considers the costs to execute a test case and to analyse the result afterwards.

2.2 Test activities

There are many ways how testing can be carried out. But always five core test activities can be identified that have to be carried out. These activities may be considered as the development process of a test case. The figure below shows their sequential order.

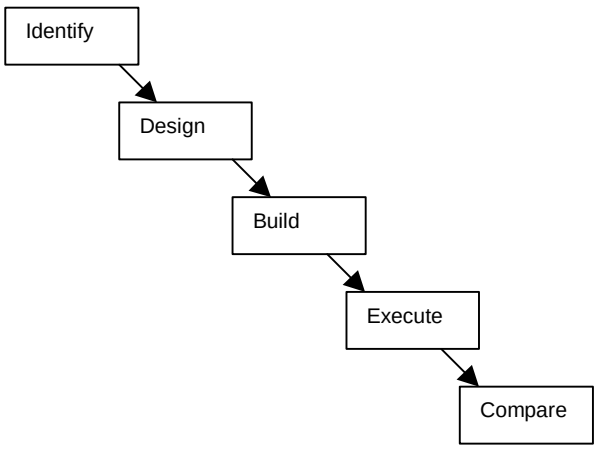


figure 2.2-1 sequential order of the five core test activities

2.2.1 Identify test conditions

The first activity determines *what* should be tested. There are different test categories like functional tests, performance tests or security tests and each category has various features that must be verified by testing. These features are called test conditions.

Special techniques help testers to derive test conditions in a systematic way. Well-known testing techniques are equivalence partitioning, boundary value analysis or cause-effect graphing.

After identifying and describing the test conditions, they should be prioritised.

2.2.2 Design test cases

Test case design determines *how* the found test condition may be tested. A test case consists of a sequence of test instructions. For each of them the test case must specify the input values, the expected outcomes and the environmental conditions. In addition it is important to consider the state of the system under test, because in dependence of this state the expected outcomes may vary widely. The expected outcomes include data items that are output, created, changed, deleted or updated during test execution. But also data items that are not modified must be specified as expected outcome. If the expected outcome is not defined in advance, then a person with adequate knowledge of the system under test must carefully examine the first actual outcome. If the first outcome is correct, it can be used as future expected outcome.

Normally every test case is part of a larger set of test cases. During the execution of one test case it may be assumed that other aspects of the system are already tested and work correctly. A hierarchy of test cases is needed so higher level test cases can use trusted functionality verified by lower level test cases executed before.

2.2.3 Build test cases

Test cases are implemented as test scripts. A test script contains the test instructions and the test data often in a formal syntax such that it can be used for automated testing. Some test cases require special preconditions, before the test case can run. These preconditions – as for example special environmental conditions – must also be part of the test script, which must list all necessary set-up or clear-up procedures.

The test inputs and expected outputs may be included as part of the script, or may be stored outside the script in a separate file or database. Especially for automated comparison it may be useful to hold the expected outcomes in a separate file.

2.2.4 Execute test cases

The software under test is executed following the instructions of a test script. There are two extremes of test execution: manual testing and automated testing. Both forms have advantages and disadvantages and most companies have chosen a test execution process that lies between the two extremes.

Test scripts can only be executed after the system under test has been implemented. The three activities before test execution – namely identification, design and building – can be performed earlier, parallel to the development of the software system.

2.2.5 Compare test outcomes with expected outcomes

The actual outcome of each test case must be checked to determine whether the software under test worked correctly or not. This check can be done informally by a human tester who verifies the actual outcome on basis of his experience, or it may be realised with an automated tool that rigorously and exactly compares actual outcome with expected outcome.

In most cases it can be assumed that the software is correct, if the actual outcomes and expected outcomes are the same. But if actual and expected outcome do not match it cannot be directly inferred that the software is incorrect. There are various possible reasons that must be considered: perhaps the expected outcome is incorrect or the test environment was not set up correctly or the test was not adequately specified.

This shows the difference between comparing and verifying. A tool is able to compare one set of test outcome with another. It can determine any differences but it cannot say whether or not a test outcome is correct. This is verification and only a human tester can insure that a software system in fact is correct.

2.3 Software lifecycle and testing

A widely accepted software lifecycle model is the V-model. The simplified V-model below illustrates the different lifecycle stages and shows that each development phase has a corresponding testing phase.

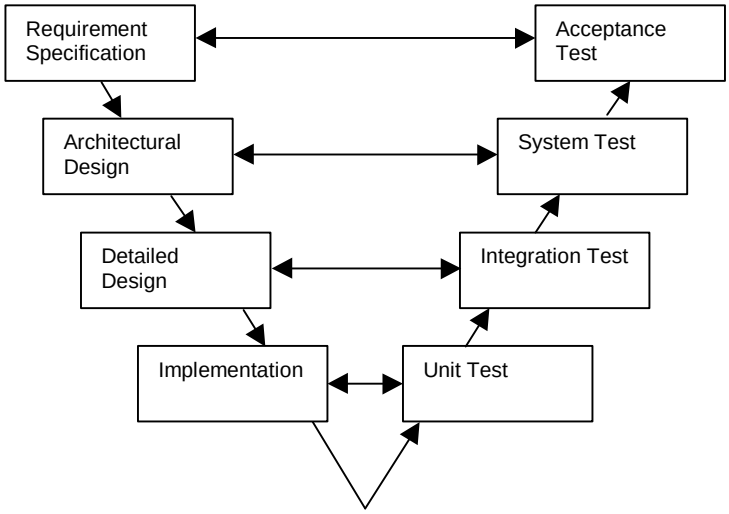


figure 2.3-1 software lifecycle model

The four test phases intend to find defects in their corresponding development phase. For example test cases of the acceptance test will be designed such that defects in the requirements can be found.

Testing is normally considered as the final step of the software development process, but – as already mentioned in section 2.2.4 – only for test execution it is necessary that the software has been written yet and there are more testing activities than just running tests.

A general principle is that test activities should take place as early as possible, because later found defects are more expensive to fix. In addition early found defects can be corrected before they may propagate.

Many test activities do not have to wait until the software has been written; they can be done as soon as the information, which they are based on, becomes available. For example the design of acceptance test cases can be done as soon as the requirements are specified. This implies that test cases may be designed in reverse order to their execution –unit test cases can only be designed last but they run first.

2.4 Tools for lifecycle testing

Tool support is available for testing in every stage of the software lifecycle. The different types of tools with their position within the lifecycle are shown in the figure below.

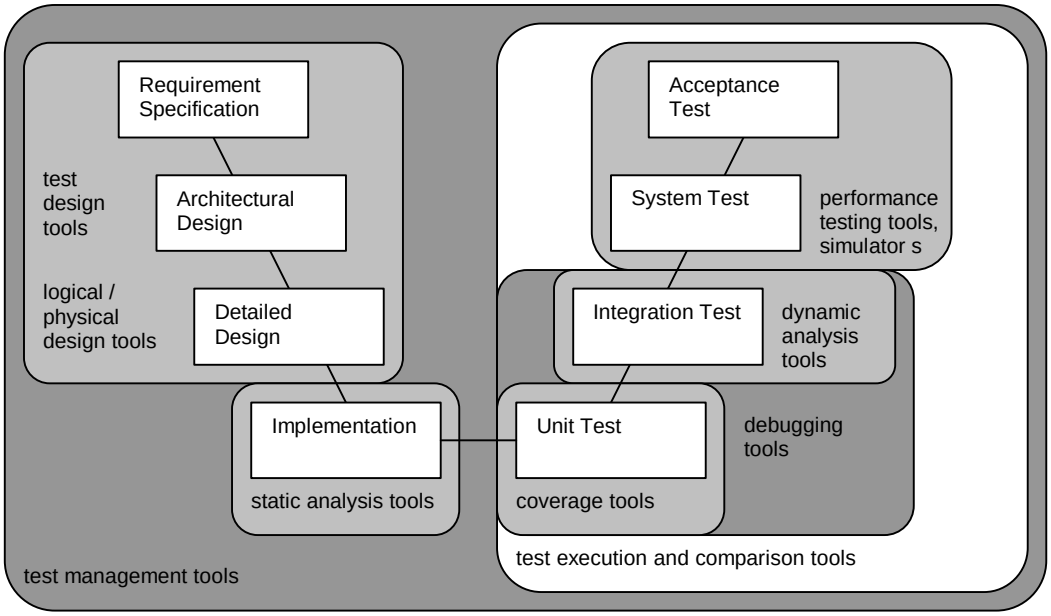


figure 2.4-1 tools for lifecycle testing

2.4.1 Test management tools

There are many types of test management tools for different purposes. They are intended to keep track of what tests have to run and assist in test planing. Also defect tracking tools and the broad rage of requirement management tools can be considered as part of this category. Requirement management tools help to trace requirements to design, code and test cases.

2.4.2 Test design tools

Two classes of test design tools help to derive test inputs:

- Logical design tools use specifications, interface definitions or source code to derive test inputs.
- Physical design tools work with existing data.

An example a physical design tool is a tool that extracts random data records from a database.

2.4.3 Static analysis tools

Static analysis tools analyse the source code without executing it. They efficiently detect certain types of defects and can be used to calculate various metrics that give information about the code complexity.

2.4.4 Coverage tools

Coverage tools indicate the code parts of the system under test that have been exercised by a set of test cases. Especially when testing safety-critical systems coverage tools are used to analyse the executed branch coverage.

2.4.5 Debugging tools

Debugging is not really a testing activity. Testing is intended to identify defects and debugging to remove them. However debugging tools are often used for testing purposes. They enable the tester to step through the code when trying to isolate low-level defects.

2.4.6 Dynamic analysis tools

Dynamic analysis tools assess the system while it is running. They help to detect run time errors, as for example memory leaks and racing conditions.

2.4.7 Simulators

Simulators are used to replace the actual environment of the system under test. They help to test the software in situations, which are hard or expensive to provoke in the real word – for example when testing the reactions of the system in emergency situations.

2.4.8 Performance testing tools

The term performance testing covers the various forms of capacity, volume and stress testing. Performance testing tools very often measure the time period that between two events, for example the mean response time under typical workload conditions. Load testing tools can be used to generate specific system workload to represent typical or maximum levels.

2.4.9 Test execution and comparison tools

Test execution and comparison tools make it possible to execute tests automatically and to compare test outcomes with expected outcomes. Such tools can be used at any level: unit, integration system or acceptance testing. A typical example for a test execution and comparison tool is a capture replay tool.

2.5 Test case generation

In chapter 2.2 five testing activities are described: identify test conditions, design test cases, build test cases, execute test cases and compare test case outcomes with expected outcomes. The first two activities have a main influence on the quality of the test cases.

Chapter 2.3 describes a V-model of the software lifecycle with its different stages. On the left-hand side of the V-model are the stages: requirement specification, architectural design, detailed design and finally implementation. As explained each development phase has its partner for testing on the right-hand side of the V-model. In dependence of the V-model stage the first two testing activities – test cases identification and design – are based on different inputs.

Acceptance test cases are based on the specifications as input. System test cases are based on interfaces as input and unit test cases are based on the code. Integration test cases may be designed on basis of code or interfaces.

2.5.1 Code-based

Code-based test case design generates test cases by examining the structure of the source code. The various paths through the code are analysed to find all sequential code segments and all branches at decision points. With this information the required logical conditions for the execution of each path segment can be determined.

Code based testing is normally executed in conjunction with coverage measurement to identify which path segments have been covered during test execution.

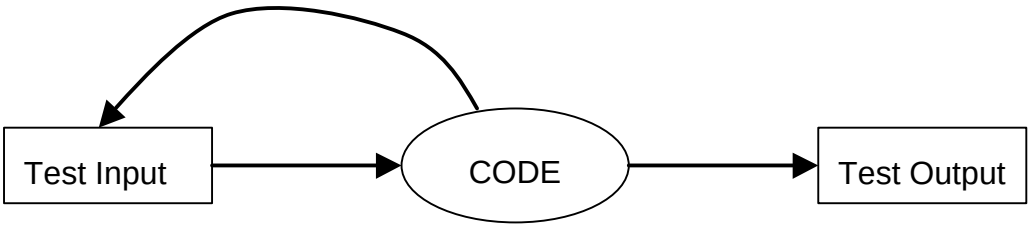


figure 2.5.1-1 code-based test case design

The code-based approach can be used to generate test inputs, but does not help in predicting correct test outcomes. To decide whether the actual outcomes are correct, expected outcomes are required to compare with.

Another problem of code-based testing is that it can only be used to test the existing code. Faults caused by missing code cannot be detected.

2.5.2 Interface-based

Interface-based test cases generation does not look at the source code itself; it only considers the interfaces between the different parts of the system.

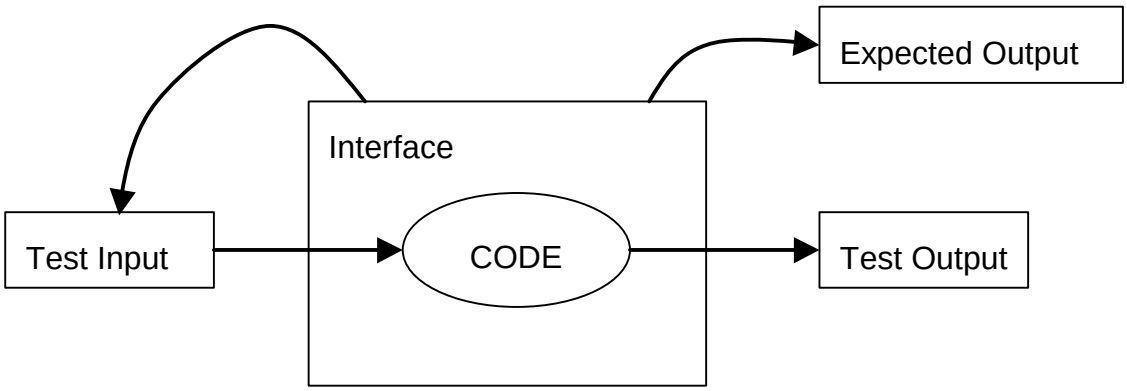


figure 2.5.2-1 interface-based test case design

Interfaces can be grouped into two categories:

- Interfaces between the software system and its environment.
- Interfaces between – possibly distributed – software components.

A typical interface between a software system and its environment is a graphical user interface with various controls – as menus, buttons or check boxes. The correctness of such GUI controls must be checked manually or with the help of a special testing tool.

Interfaces between two or more software components are normally well-defined method invocations with input parameters and return values. Testing can be performed by selecting each parameter and change its value or type in some way to see if the invoked component can correctly deal with this or detect the change. This type of is known as mutation testing.

With an interface-based approach expected outcomes can be generated, but only in a very general sense. For example a correct result may be “mutation can be handled”. This approach therefore has a partial test oracle and can be used to identifying some types of defects.

An advantage of interface-based testing against code-based testing is that it does not only check the existing parts of the system, it also allows to prove that “everything that should be there is there”.

2.5.3 Specification-based

For specification-based testing the specification of a system is analysed to extract the information necessary for the test case design. A specification may be written in natural language or it may contain data in form of tables or states charts. If a tool has to analyse the specification, it is necessary to write it down in a structured way.

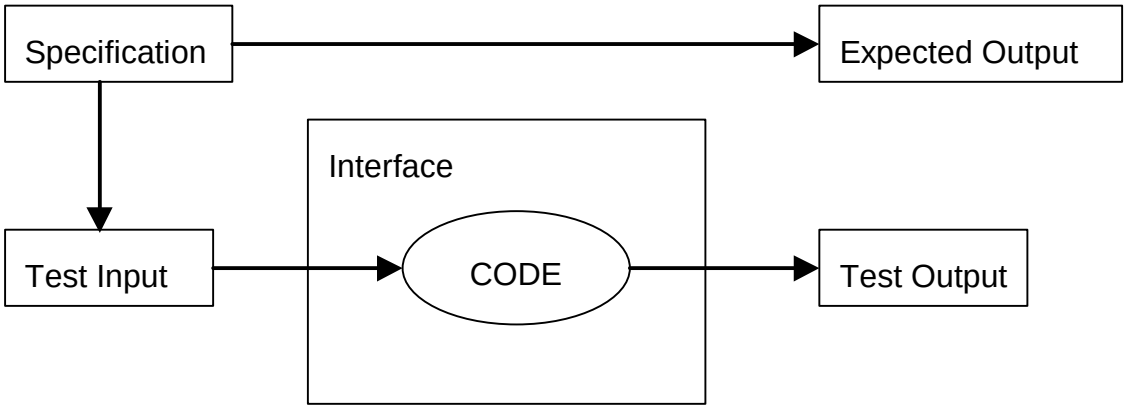


figure 2.5.3-1 specification-based test case design

An example of test case design is the building equivalence classes: the allowable ranges of the input values are used to determine boundary values, valid values and invalid values as test input.

The main benefit of the specification-based approach is that expected test outcomes can be generated – if they are defined correctly in the specification.

For specification-based testing it is essential to validate that the specification is correct by searching for specification defects such as ambiguities and omissions.

3 Survey - two testing tools

This chapter summarises the features of two existing testing tools for Java programs. In the recent years many different testing tools became commercially or freely available and nearly all of them have slightly different purposes and intentions.

Perhaps the best-known testing framework for Java programs is JUnit. As its name implies, it is intended for unit testing and does not specially consider problems arising by the distributed nature of a system. But the internal structure of JUnit can be considered as a basic pattern and shows how good testing frameworks can be build up.

The other testing framework has the long name Parallel Architecture for Component Testing. Its focus lays on testing distributed systems that use Java RMI to communicate.

3.1 JUnit

JUnit was developed by two of the best known personalities in the object-oriented field: Kent Beck and Erich Gamma [3]. It is a well-designed testing framework with the intention to improve programming productivity by writing self-testing code.

JUnit consists of three packages, namely *Textui*, *Ui* and *Framework* and each package contains a set of classes.

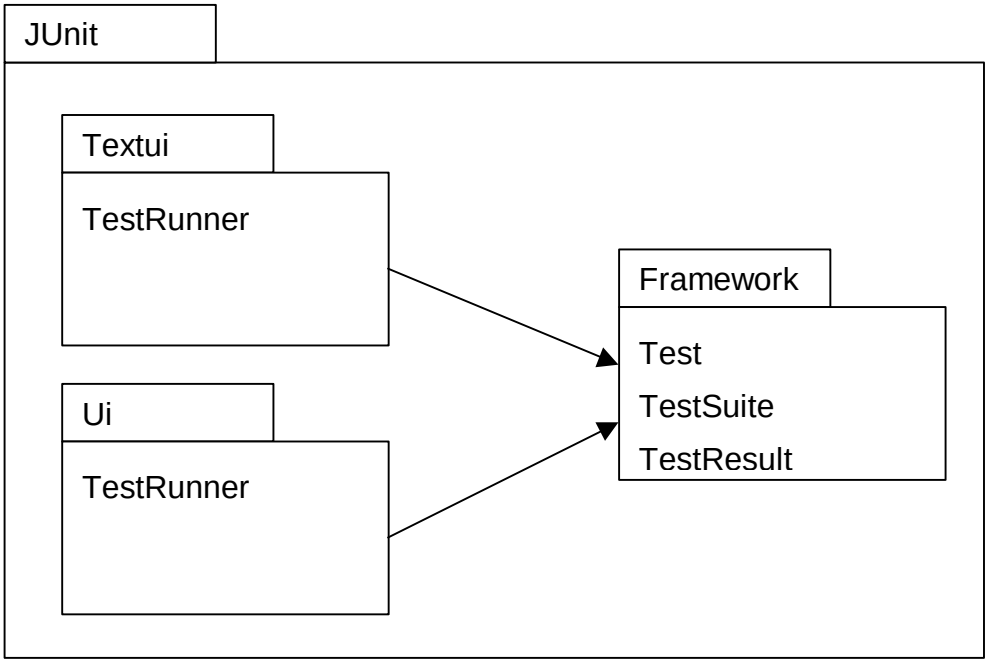


figure 3.1-1 packages of JUnit

The figure above shows the dependencies between the packages. *Textui* and *Ui* depend on the package *Framework*. They both contain a *TestRunner* class and are the user interface to the tester. The difference between them is that only *Ui* provides the tester with a graphical user interface.

3.1.1 Internal structure

The main package of JUnit is the *Framework* package. The figure below shows its class diagram.

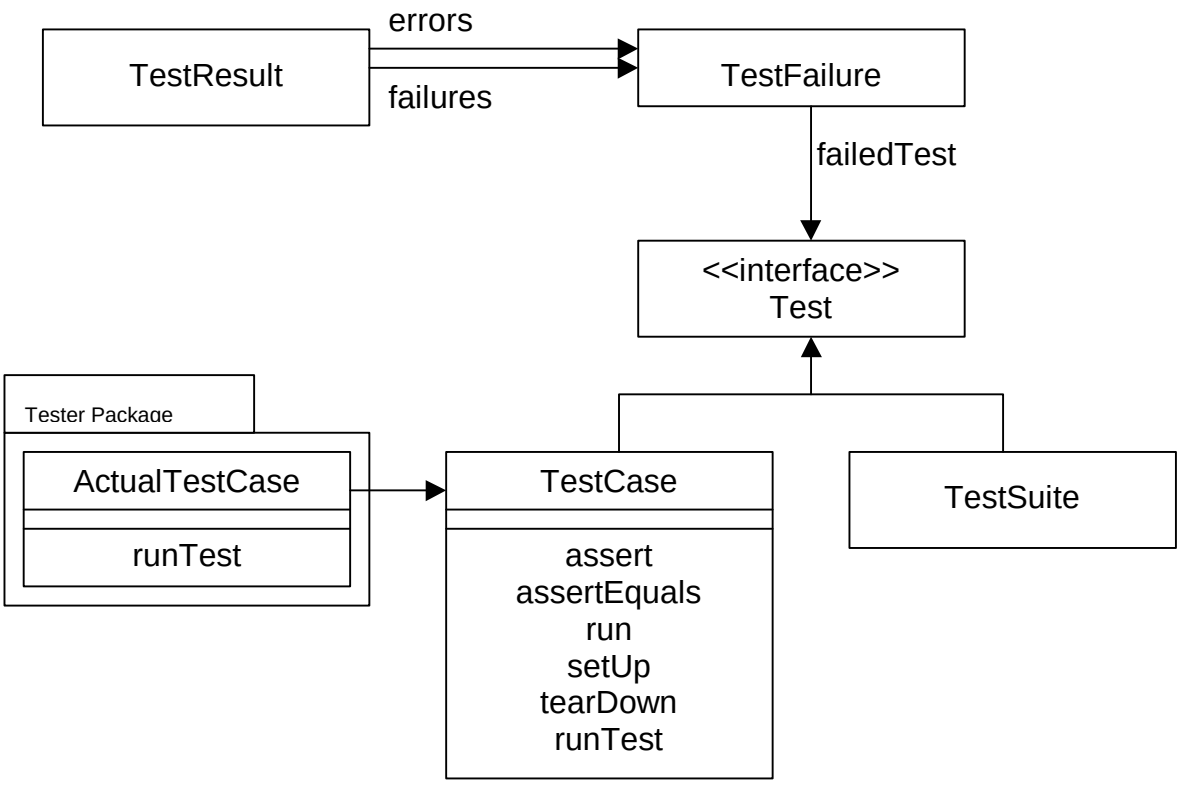


figure 3.1.1-1 package Framework

A *TestCase* class represents a single test. Its most important method is the *run* method, inherited from the *Test* interface. This method executes the test case by invoking the *setUp*, *runTest* and *tearDown* methods in sequence. The *runTest* method is defined as abstract and must be implemented by a subclass written by the tester. It contains the code that executes the actual testing procedure.

In the figure the *ActualTestCase* class represents the test case written by the tester. Normally a test case is written for every single class of the system under test. The *runTest* method of the *ActualTestCase* class invokes methods of the system under test and then uses the *assert* and *assertEquals* methods to check expressions or values.

The *setUp* and *tearDown* methods are defined as blank methods and can be overwritten in the *ActualTestCase* class to set up or tear down a user defined test fixture.

The results of a test case are stored in an instance of the *TestResult* class. The *TestResult* class contains two collections of type *TestFailure* - one collection is called *failures* and the other is called *errors*. An error is defined as an unhandled exception that normally would had led to a crash. A failure contains details of user defined assertions that have failed. The *TestFailure* class records the occurred failures or errors together with information about the test that generated them.

3.1.2 Running tests

The user-defined test cases can be collected into test suites. A test suite can contain either single test cases or entire test suites. Running a test suite causes all the nested test cases to run recursively.

This architectural feature is achieved by applying the so-called composite pattern: the classes, *TestCase* and *TestSuite,* both implement the *Test* interface*.* This pattern makes it unnecessary to distinguish between the two different classes.

Calling the *run* method of the *TestRunner* class in the package *Ui* or *Textui* starts a test. *TestRunner* creates a new *TestResult* object and then calls the *run* method of its *TestSuite*, passing *TestResult* as argument. *TestResult* collects all failures and errors that may occur during any of the tests of the suite. The *run* method of the main *TestSuite* invokes the *run* methods of all contained tests.

The *assert* method may be called more than once during the test execution to check parameter values. If a value is not correct then the *assert* method throws an *AssertionFailed* exception, which is caught and added as a failure to the *failures* collection of *TestResult*. If a not handled exception is raised then an error is added to the *errors* collection of *TestResult*. During a successful test run, no assertion fails and all raised exceptions are handled.

3.2 Parallel architecture for component testing

PACT is an abbreviation for Parallel Architecture for Component Testing, a testing framework designed by John D. McGregor [4]. PACT is a framework that assists in selecting and executing test cases by providing a test execution environment.

PACT enables the tester to monitor and control distributed systems based on Java RMI by instrumenting individual classes. The instrumentation method is independent of the chosen distribution technique and can be applied with the Java RMI as well as with CORBA based systems.

3.2.1 Types of faults

McGregor defines tree types of faults that are specific to distributed systems:

Infrastructure Faults

Network failures and network latency – the amount of time a message needs to pass from the sender to the receiver – may cause infrastructure faults. If a network problem causes a distributed system to crash or to behave abnormally, it is an infrastructure fault of the system. The system cannot prevent a remote machine from failing, but it must anticipate such failures and react with recovery sequences.

McGregor points out, that it is important to test this failure recovery code as rigorously as the main logic of the program. A problem is that network latency can vary widely and this variability can cause different results when the same test is executed more than one time.

Concurrency Faults

Concurrency faults cover problems resulting from the synchronisation of multiple independent tasks. Due to race conditions deadlock, livelock or incorrect results can occur. Essentially these are problems of order.

All concurrency faults have a common characteristic: their occurrence is non-deterministic; and this makes them difficult to detect.

Distribution Faults

Distribution faults can be caused by

- objects that are required but not registered
- servers that are not implementing a requested operation
- incorrect permissions on remote machines

Dynamically loadable components increase the complexity of testing for distribution faults.

In addition to these three faults caused by the distributed nature of a system, there are still remaining all those faults that are associated with traditional systems.

3.2.2 Finding faults

Many faults specific to distributed systems are produced when events occur in an unexpected order. The problem is that the order of events is influenced by external factors and therefore cannot be predicted easily. For example network latency on one branch of a network can cause messages to arrive in a different order than anticipated.

Because of non-deterministic faults, McGregor therefore proposes the following strategy for finding faults in a distributed system:

“Investigate the effects of varying the sequence of events on the target object”

Executing the software so that events occur in various different orders and monitoring the respective outcome helps to verify that the system works correctly.

There are tree techniques for realising this verification:

Logging exercised paths

The paths exercised are logged and after a large number of executions the logged data is analysed to determine which different orders have occurred.

Modify application

The application under test can be modified to guarantee a specific order of execution.

Artificial delays

Construction of a test environment so that artificial delays can be injected into the communication between components.

3.2.3 Internal structure

The testing framework that McGregor describes is based on Java RMI as distribution model. The tool is intended to test a component of a distributed system in its operational context. This has the advantage that it is not necessary to stub all calls to other objects. In addition testing in the operational context is particularly important to achieve reliable test results.

The main objective of PACT is to provide the tester with fine-grained control over the interactions that occur between the objects under test by reflecting on the interfaces of the objects. The tool allows the tester to invoke individual methods.

Another important objective of PACT is to avoid the modification of the system under test. The wrapper pattern is used to add the functionality needed for testing to the normal behaviour of the encapsulated objects. The wrapper pattern involves two objects: the wrapper object and the wrapped object.

The wrapped object must be a component that provides a public interface. Such components are called servers, and their interface methods can be invoked by other components, so-called client. The server interface is made available through the Java naming service “rmiregistry”. The naming service allows clients to locate the server.

The wrapper around the server provides the original interface of the server, but it also possesses additional interfaces. The original interface of the server is still available through the naming service and other objects see the wrapper as the server under test. The additional interfaces of the wrapper support testing behaviours, as for example methods for manipulating logs.

The main activities of the testing process are organised as follows:

1. **Creation of the wrapper class**
   The *Reflection* API of Java is used to obtain the interface of the component under test. The instantiated objects from the generated wrapper class will include the interface of the component under test and can be used to encapsulate an instance of the component under test.
2. **Replacing the object under test**
   The object under test is replaced with its wrapped counterpart. The wrapped object under test (WOUT) is registered in the registry under the name of the object under test.
3. **ObjectBrowser**
   With the help of a graphical user interface – the ObjectBrowser – the tester can monitor and control the WOUT. The ObjectBrowser presents a list of all the objects registered on the machine to which the tester has connected.
4. **Selecting object**
   The tester selects a WOUT and its interface methods are listed. Then the tester chooses a method, provides the values of each parameter and finally invokes the method. The tester must ensure that the pre-conditions for the method invocation are satisfied, otherwise the behaviour of the object under test cannot be predicted.
5. **Result**
   The return value of the method invocation will be displayed. A watch window presents the values of the internal object parameters allowing the tester to verify that the actual values are correct.

As explained above the ordering of events is essential during the testing process of a distributed system. To cover different orderings the tester can specifically send messages in different orders to the WOUT. In addition the wrappers can be used to introduce artificial delays. The WOUT normally directly forwards the incoming messages to the object under test. Delaying the delivery - for example until another specific message is received and forwarded - provides a way for systematic coverage of all possible orderings.

3.2.4 Critical evaluation

The PACT framework is intended to validate the correctness of a specific class under test - not the remainder of the system. The approach cannot be used to ensure the correct behaviour of the whole system.

The main advantage of this approach is that a tested class does not need to be changed to add or to remove test-related behaviour.

Insufficiencies of the PACT framework are:

1. The testing process cannot be executed automatically. For large-scale systems it is important to invoke methods systematically. Therefore test protocols, sequences of method invocations that implement some meaningful behaviour, are needed.
2. Another problem is caused by methods that require not only primitive types but also other objects as parameters. A Helper class must be created for each class that is used as parameter. This process requires additional effort.

4 Methodology for testing distributed systems

It is impossible to define generally how a distributed system looks like. There are too many kinds of distributed systems, which differ in their architectural software designs as well as in their hardware platforms and have a wide range of different purposes. This diversity implies that it is impossible to define a general testing framework that allows testing all of them. Before starting with the development of a testing framework, it must be defined for what kind of distributed system the testing framework is intended to be applied. The testing framework that is described here, in my dissertation, is intended to test client-server systems that use CORBA as middleware with Java as implementation language.

4.1 System under test

A CORBA-based client-server system consists of service requestors, called clients, and service providers, called servers. Servers and clients are isolated by well defined encapsulating interfaces. Requests can be made by using statically defined invocation interfaces or by dynamically creating invocation structures. The object request broker (ORB) provides the mechanisms needed to make requests and to receive responses transparently in heterogeneous environments. The middleware CORBA offers additional services to support basic functions for using and implementing distributed objects.

The language used to describe interfaces, which the clients call and the servers implement, is called Interface Definition Language (IDL). The CORBA middleware provides IDL compilers that can be used with various programming languages like C, C++ or Java to generate appropriate language constructs from the IDL description of an interface.

The server and client components of a distributed system may be running on various machines with different hardware platforms and operating systems connected by a network. To test such systems it is necessary that the testing framework has the ability to run on different hardware platforms with different operating systems. The fact that in a distributed system not all server or client components must be written in the same programming language also must be address during the design of a testing framework.

The testing framework presented in the following chapters is written in Java. It can be used to test distributed systems independently of the chosen operating systems and hardware platforms as long as the components of the system under test are written in Java.

4.1.1 System components

A distributed system S can be described as a set of individually addressable components C. The number of components that belong to the system in n.
C can be defined with the following equation:

$$C = \{C_1, C_2, ..., C_n\}, n \geq 2$$

The system S is said to be dynamic if the number of components can change during system execution. If n does not change the system is considered static. A CORBA-based system can be static or dynamic.

A component could be a client, a server or both. Each server presents at least one interface. An interface consists of one or more methods that can be invoked by clients.

A method is specified through:

- the name of the method
- the parameters passed between server and client
- the return type of the method
- the possibly raised exceptions

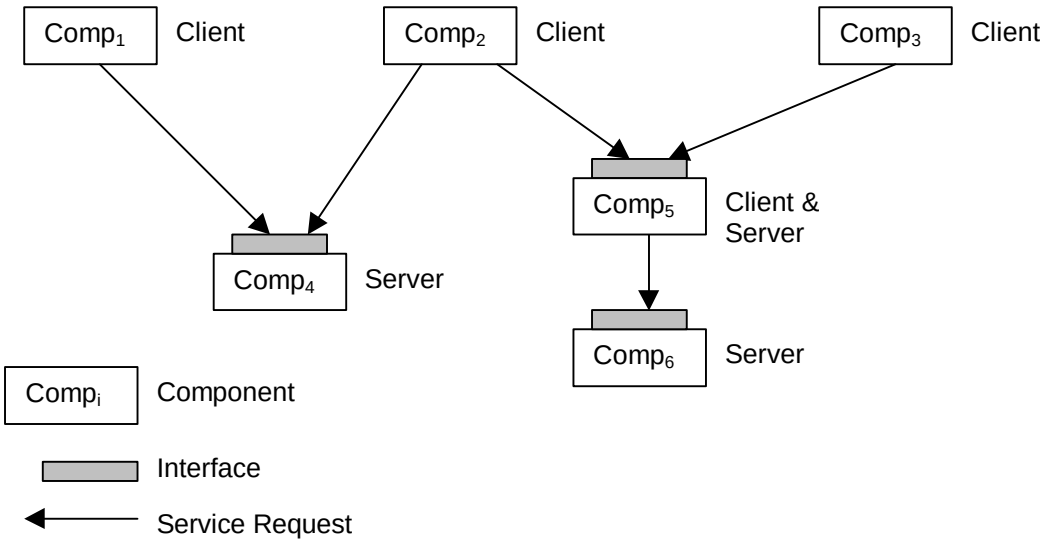


figure 4.1.1-1 components in a distributed system

4.2 Objectives of the testing framework

Distributed systems with client-server architectures cause new difficulties during the testing process. This paragraph describes the new testing issues that must be satisfactorily addressed by the testing framework.

The aim of the testing framework is to enable the tester to efficiently test a distributed system by monitoring the system behaviour and by injecting typical faults. This implies that the testers must be able to monitor and control system execution. The basic requirements, that the testing framework must fulfil, can be grouped into three groups:

- test monitoring
- test controlling
- test measuring

Monitoring, controlling and measuring is realised at interface level. This approach can take advantage of the fact that in CORBA-based systems, interfaces between components are exactly specified with an Interface Definition Language.

4.2.1 Requirements – testing framework

Test monitoring

The testing framework should provide a graphical user interface to enable the tester to monitor the system behaviour. Monitoring the system behaviour consists of:
- visualising the system behaviour in the presence of a known input sequence
- visualising the system behaviour in the presence of injected faults

The tester needs the ability to reduce the amount of data to a manageable size by monitoring only specific parts of the system.

The selection of system parts that should be monitored is done on the level of the component interface definitions. A list of all interfaces in the system must be presented to the tester. The following information should be extracted from the interface definitions:
- names of methods
- return types
- parameter types
- exceptions that can be thrown

Test controlling

The testing framework should provide a graphical user interface to enable the tester to control the system behaviour. To control the system behaviour the tester needs the ability to:
- activate testing services
- define filters for specific system monitoring
- inject different kinds of predefined faults

Injectable faults should be simulated delays, component crashes or raised exceptions.

Filters should be available for interface methods, parameters, return values and exceptions.

Test measuring

The testing framework should assist the tester in the analysis of the monitored data:
- to measure the test adequacy and determine the achieved coverage
- evaluate the system performance

The interface definitions should be used as basis for coverage and performance measurements.

A centralised data storage must be provided to store the data monitored in distributed locations.

The system performance can be determined by measuring
- the instantaneous and average load distribution
- the maximal, minimal and average response times

4.2.2 Requirements – testing methodology

The aim of a testing framework is to define a methodology to test single components as well as the whole system.

To guide the tester three questions should be answered by the testing methodology:

1. If the tester defines a test set T_{Comp} to test a component $Comp_i$ of S, how can the tester assess the adequacy of T_{Comp}? [1]
2. If the tester defines a test set T_S to test a system S, how can the tester assess the adequacy of T_S?
3. The system S is required to be fault-tolerant. What methodology is appropriate to test the fault-tolerance capabilities of S?

To assess the adequacy of a test set T the testing methodology should define a test adequacy criterion.

This leads to another problem: How can be evaluated whether an adequacy criterion is useful or not? The methodology itself must be evaluated. The results of such an evaluation may be that the chosen methodology must be redefined to improve its capabilities to test that a system under test fulfils its requirements or its capabilities to test for fault tolerance.

4.3 Distributed components under test

When testing a component C of a distributed system, the tester needs a methodology to evaluate the test adequacy of the applied test set T_{Comp}. In order to ensure that the component under test behave exactly the same way during test execution as during normal system execution, it is also necessary to consider dependencies between components.

4.3.1 Test adequacy criteria

The various software components of a distributed system normally are implemented by different persons or even bought from external software companies. The tester usually has access to the source code of in-house developed components, but the source code of externally supplied components may be not available. Depending on the availability of the source code, different testing techniques have to be applied for component testing. In the literature it is normally differentiated between two main testing techniques: black-box testing and white-box testing. If the source code is available, white-box testing can be executed otherwise only black-box testing is possible.

Two common test adequacy criteria for white-box testing are:

- **Decision coverage**
  The intention of the tester is to generate a test set that covers all decisions in the source code of a component. A test set is adequate, if a predefined percentage of all decisions have been exercised by the tests.

[1] The tester defines test cases as sequences of input values and various test cases may be grouped to a test set.

- **Path coverage**
  The intention of the tester is to generate a test set that covers all paths in the source code of a component. A test set is adequate, if a predefined percentage of all paths have been exercised by the tests.

The problem of weight-box testing methodologies is their lack of scalability. Decision coverage and path coverage are criteria that do not scaled up with the increasing size and complexity of the components. Tools supporting weight-box testing are useful to assess test sets for relatively small components, but they are not cost-effective when used to test large components. The problem is that these criteria are based on low-level elements and the number of possible ways to reach a low-level element explodes with an increasing component size.

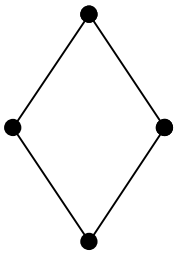
To illustrate this problem, the number of possible path through a component is calculated in the following example.

Example:

Inside the component C the functions f and g are called.

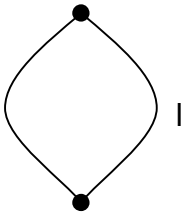
code of function f:

```
void f () {
      ...
      if ( ) {
            ...
      }
      else {
            ...
      }
}
```



code of function g:

```
void g () {
      ...
      while () {
            ...
      }
      ...
}
```



n_C: number of possible path through the component C

n_f: number of possible path through the function f

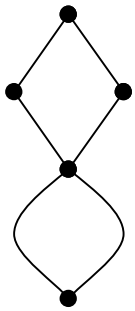
n_g: number of possible path through the function g

l: number of loop repetitions

The function f is simple and it can be easily seen that n_f is equal to 2. The number of path through g depends on the maximum and minimum number of loop repetitions that can occur during run-time. Under the assumption that the while-loop is executed at least once and maximal 6 times n_g can be calculated as:

$n_g=1^1+1^2+1^3+1^4+1^5+1^6=6$ (with $0 < l \leq 6$)

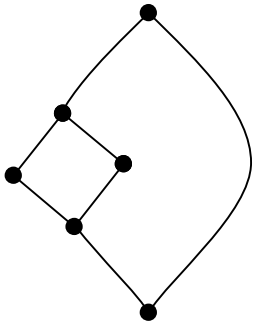
If inside the component C the two functions f and g are called one after the other the number of possible path through C can be determined with the following rule:



The number of possible path after the sequential composition of two or more functions can be calculated by multiplying the number of possible path through the individual functions.

If the two function f and g are called sequentially:
$n_c = n_f \times n_g = 2 \times 6 = 12$

If function f is called inside of the while-loop of g, the number of possible paths through the component increases drastically.



If the two function f and g are called in a nested way:
$n_c = 2^1+2^2+2^3+2^4+2^5+2^6=2+4+8+16+32+64=126$

This example shows that the number of possible paths explodes to an infeasible number for calculations that are done with realistic numbers of decision elements and loop counts.

The size of weight-box coverage domains usually increases exponentially to an increase of component size. Therefore the amount of test cases required to fulfil a test adequacy criterion is typically far higher than what is possible with the resources available. In addition it becomes more and more difficult to design test cases that cover low-level elements, because of the test cases must pass a complex nested structures before reaching a desired low-level element. Both problems make the generation of new test cases to improve a test set with respect to weight-box coverage criteria to a difficult task. For large software components it is actually impossible a to obtain high coverage for low-level elements as path or decisions. Therefore it may become necessary to choose higher-level elements as basis for the determination of the test adequacy.

4.3.2 Dependencies between components

A client component calls interface methods of server components. If the component under test is a client that needs services of other components, it is necessary that the invocation interfaces of these services are available during testing. One possibility is to use stubs that act as placeholders for the server components and provide the interfaces expected by the component under test. However when stubs are used during testing it is impossible to create the same conditions as in an actual system environment.

The problem is that deadlocks and race conditions very often only occur when the component under test is under special environmental conditions. Even though correct behaviour is exhibited when the component is executed with stubs, the behaviour may not be correct when real server components are used.

Another problem arises if a server component is multithreaded, because different numbers of clients may cause different testing results. It is possible that errors remain hidden in the code when only one client is used and only one server thread is

executed. However, with many clients several threads may be started and race conditions may show the error.

The need for realistic environmental conditions during testing and the dependencies between components make it necessary to test components together. A simple approach is to start with components, which do not require any services, and go on with components that only use services of already tested components. However this is only possible if the system under test has a layered architecture with a bottom-most layer of components that do not depend on any other components. But distributed systems very often do not fit into such a layered architecture. If dependencies between components form cycles, it is not clear which component must be tested first. All components should be tested together.

4.4 Distributed systems under test

Section 4.3.1 considers the problem of finding a test adequacy criterion that scales with the increasing size of the components under test. The same problem arises when testing a complete distributed system. It is important to choose a test adequacy criterion that is scalable with the number of components the system under test consists of.

4.4.1 Redundant testing

The most common approach to test distributed systems consists of two main steps: First the different components are tested separately and then - during integration test - all components are tested together. For large distributed systems however very often more than only one integration step is necessary. Before the final integration of a system is possible, the tester needs to check the correctness of smaller subsystems to reduce the integration test complexity to a manageable size.

Very often different test adequacy criteria are used during the different testing steps. The test adequacy criterion applied when testing components separately is not used, when the entire system is tested. This leads to the problem that the test adequacy reached during component testing cannot be directly reused in the following testing step. To avoid redundant testing during the integration of the components, the tester must be able to identify how much extra testing is needed during integration to ensure a sufficient software quality.

Finding out how much testing really is necessary and how much of the testing is redundant, is especially important when testing reused software components. The advantage of reusing already coded software is that the implementation phase gets shorter, but software reuse may cause problems during system integration. New failures may arise, because reused components are originally designed for similar applications with nearly – but not exactly – the same specifications. Therefore it is necessary to test reused components.

4.4.2 Distributed monitoring and control

An important aspect of testing is monitoring and controlling the system execution. Distributed systems consist of multiple computers connected by a network. This makes the test monitoring and test control mechanisms complex, because distributed monitoring and control is needed. In addition the monitoring and control mechanisms must be scalable and configurable such that they can be adapted to the testing methodology and test adequacy criteria used.

Very often a distributed system consists of many components, which together create an enormous amount of data that have to be collected and processed. The distributed data collection and storage mechanisms need to be designed such that they can handle this huge amount and that they take care of possible buffer overflows.

It is important to ensure that the monitor and control mechanisms as well as the data collection and storage mechanisms do not cause bottlenecks during the testing process.

The automation of the monitoring and control mechanisms gets important when the testing framework is used for regression testing. If the system under test evolves steadily, automation is necessary to retest it efficiently.

4.4.3 Concurrency

A testing framework for distributed systems must provide monitoring mechanisms capable to detect concurrency problems. In a distributed system concurrent processing occurs permanently and in dependence of environmental conditions concurrency can lead to problems caused by race conditions or deadlocks. Random testing may or may not reveal such problems. Therefore a good testing strategy is required to detect race conditions and deadlocks in the code. A control mechanism is needed to reproduce system failures. This is a difficult task because of the non-determinism involved.

The tester needs to have full control over the system in order to test for concurrency related problems, otherwise he is unable to reproduce execution behaviour.

Deadlock:
The figure 4.4.3-1 shows how concurrent processes may cause deadlock situations. The components P, Q and R are three servers that are supposed to be single threaded.

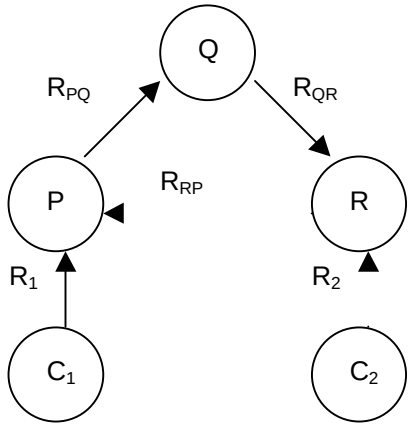


figure 4.4.3 –1 deadlock situation

In the example the following sequence of requests is visualised:

1) Client C_1 makes a request R_1 to P.
   Client C_2 makes a request R_2 to R.

2) During the processing of request R_1 P makes a request R_{PQ} to Q.
   During the processing of request R_2 R makes a request R_{RP} to P.
   P is busy because of request R_1, therefore R must wait until P gets free again.

3) During the processing of request R_{PQ} Q makes a request R_{QR} to R.
   R is busy because of request R_2 and waiting for P because of request R_{RP}.
   P does not get free before Q returns but Q has to wait for R.

The two initial requests R_1 and R_2 from Client C_1 and C_2 cause a circular-waiting situation - a deadlock, all tree servers are waiting for each other.

Race conditions:
Race conditions that cause erroneous system behaviour can be described basically as unexpected execution order. Race conditions can occur in different situations; one example is the CORBA one-way calls, which are delivered with best-effort semantics. If a client makes several one-way calls to a server, the order in which they arrive is undefined.

When testing for concurrency related problems multi-threaded servers must be considered carefully, because multi-threading implies more concurrency and therefore race conditions or deadlocks may occur more easily.

If threads are used, testing for thread-safe properties is important.

4.4.4 Testing for performance

In section 4.4.2 it is pointed out that the tester must have the ability to control the system under test. In addition the tester may wish to control the environmental conditions of the system under test to evaluate the system performance. Stress testing helps to ensure that the system performs well under different load factors. This is necessary because components may perform well under small workload, but may perform poorly or even crash under high workload. Therefore the usage of the various system services, the number of request to the different server components and the environmental conditions have to be metered.

4.5 Fault tolerance

Distributed systems are often required to be fault tolerant to different kind of faults. They need to tolerate component faults as well as faults in the environment in which the components execute. An example for an environmental fault is the following situation: A client makes a one way call, but the request does not arrive at the server because of a network breakdown. A common component fault occurs if a client that makes a request to a server and gets blocked forever because the server crashes.

Some distributed system require strict fault tolerance because any failure in the system would lead to catastrophic results that may cause enormous costs or danger for human life. If strict fault tolerance is required faults must be handled immediately through backup components or alternative routing mechanisms without system shutdown. Most of the systems however do not require such a strict form of fault tolerance. System breakdowns are acceptable, if they are rare and short. The system is only required to be available most of the time. The required level of fault tolerance must be known during the tests for fault tolerance.

To observe the effects of failures occurring in individual components or in the underlying communication mechanism the entire system in its normal execution environment must be considered. The ability of a system to handle faults depends on

the correctness of its fault-handling code. It is important to test the fault-handling code, but the problem is that in a testing environment the fault-handling code gets rarely executed, because of that faults must be triggered directly.

Fault injection testing can be used to test for fault tolerance by injecting faults into the system under test and observing the resulting behaviour. Faults are injected through so called fault-sites. Dedicated fault injection code is inserted into a program such that a fault is triggered when execution control reaches the fault-site.

Two difficulties arise in the context of fault injection:

- Reachability
  Adding fault injection code to a program is not enough to cause a failure. It is necessary that the fault-site is reached and executed during the test.
- Latency
  Although a fault has been triggered the tester might not see any results. Very often execution control has already passed through several modules or components before the triggered fault gets observable. The time period before a fault manifests itself is undetermined.

To test a system for fault tolerance the tester firstly adds the fault injection code to the program and then ensures that the execution control reaches the point where the fault is injected. A system is fault tolerant if it detects the fault and handles it correctly.

While testing a system for fault tolerance the tester may wish to inject different kinds of faults. Having a generic set of faults for distributed systems helps to do fault injection efficiently. In an ideal situation the tester only selects a fault from a list of typical faults, determines the fault-site and then the injection of the fault is performed by a dedicated fault injection mechanism of a testing framework.

Fault coverage is used for assessing the test adequacy with respect to fault tolerance. Fault coverage is defined as the percentage of faults triggered compared with the number of injected faults. If the fault coverage is considered together with code coverage it results in a 2-dimensional diagram visualising the testing effort.

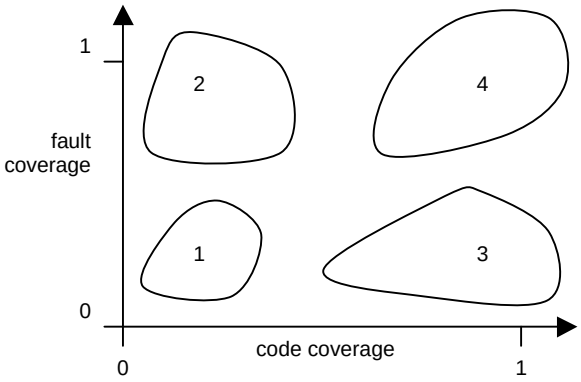


figure 4.5-1 two-dimensional coverage diagram

Figure 4.5-1 shows four main regions inside the coverage diagram. The regions help to evaluate given test sets.

- In region 1 testing is considered inadequate because only low code coverage and low fault coverage is achieved.

- In region 2 fault coverage is high but code coverage is low. Highly available systems require a testing process that ensures good fault tolerance mechanisms that mask errors. Therefore the fault coverage for such systems must be especially high. The problem of region 2 is that new faults may arise in uncovered code and these faults must be handled by perhaps already over-exercised fault handlers.
- Region 3 is an unsafe region: test sets with high code coverage but low fault coverage are inadequate for systems that must be fault tolerant.
- The safest region is indicated by 4, which has both high code coverage and high fault coverage.

4.5.1 Fault injection

As said before a list of typical faults is needed to efficiently perform fault injection testing. Studying typical faults also improves the software development process. It can prevent the recurrence of known problems and helps during the design of fault handlers. During the testing process the classification of faults can be used to test the system against identified faults. If generic faults can be determined the testing framework is able to use them for efficient fault injection testing.

Distributed systems may fail in many ways and failures can have different causes, severity, and cost of recovery. This variety of possible faults requires very different fault injection techniques that range from hardware fault injection at pin level of chips, to software fault injection in software programs.

Most commercially available fault injection tools can only perform low-level fault injection by simulating or provoking hardware errors. Fault injection at hardware level requires special hardware support, which makes the fault injection testing to an expensive process. The emphasis of low-level fault injection tools lies in the test throughput, the number of tests performed. The tools do not check which fault handlers get exercised or consider the underlying software architecture.

Various studies have been done to analyse faults [7]. Results indicate that the causes of problems have shifted from hardware to software over the years. In addition it has been shown that the number of module internal faults is less than the number of module interface faults. Especially in the later phases of the software life cycle module interface faults tend to dominate over other faults. The existing studies have categorised and listed typical faults for nearly all common software systems. For distributed systems various faults at the level of component interfaces have been identified. The tester can use the list to perform fault injection testing.

The table below shows examples for typical faults that may be injected at interface level. Possible system reactions after the fault has been triggered are listed in the corresponding right column.

Injected fault	Observed behaviour
server crash	client hangs
invalid return value	error message on GUI
access delay	wrong updates

The testing framework described in this dissertation uses the interfaces between components as fault-site. During testing more and more faults get trigged and the likelihood of finding unhandled problem increases. The raised exceptions or returned error codes are indicators that helps to show how well a distributed system reacts to faulty components.

Scaling is an issue for fault injection as well as for monitoring and control. A testing framework used for fault injection testing must scales up for large distributed systems. A problem for fault injection testing is that for many distributed systems the correct behaviour is not clearly defined for different circumstance. Very often only the acceptable behaviour is specified and it is difficult to build an oracle that detects deviation from the correct behaviour.

4.5.2 Typical faults of CORBA based distributed systems

CORBA is a standardised middleware for building distributed systems. A systematic analysis of typical errors and failures[2] in CORBA-based systems can be realised by examining of the specifications of CORBA. The analysis helps to design testing tools and to understand the system behaviour if one or more components fail.

Sources of errors

There are various reasons for errors made during the design and coding phases of CORBA-based systems. Errors result from mistakes made by the system developers, but an examination of the CORBA standards reveals typical errors creeping into the code. Some of the errors may only occur in conjunction with special implementation languages.

- **Errors in the descriptions of the CORBA interfaces**
  The interface definition language is used to describe interface methods. Every method specification consists of the method name, parameters and zero or more exceptions raised. Parameters can be defined as `in`, `out` or `inout`. When a parameter is declared as `in`, its value remains unchanged for the client. A parameter specified as `out` changes its value upon return. A common mistake is to specify wrongly a parameter as `in` instead of `out` or vice versa.

  Example:
  Correct: `void transfer (in float add, out float sav);`
  Incorrect: `void transfer (in float add, in float sav);`

  The two lines above show a correct and an incorrect IDL-description of the method `transfer`, which is called by a client to transfer money. The corresponding implementations differ in the parameter `sav`. When the method invocation returns to the client, the value `sav` obtained from the first implementation gives the new and correct savings, while the second implementation causes an incorrect result.

- **Errors in the server implementation**
  The next likely source of errors are the server implementations of the defined CORBA interfaces. All possible programming errors could occur during the server implementation phase. Errors could range from mistakes during the interpretation

[2] The difference between errors and failures is that errors creep into a system during design or coding and failures arise during the execution of a system.

of the interface description to erroneous loop bounds, array errors, range errors, etc. The following example shows a mistake during the interface interpretation. An interface method `sort` is defined as follows:

```
void sort(in sequence arr, in long n);
```

The number n may be intended to range from 0 to (n-1), but during the implementation the developer can also assume the range 0 to n.

- **Errors in the client code**
  Like server components, client components can contain any kind of programming errors. Of particular interest are errors that occur if methods implemented by other components are invoked incorrectly. A typical error is that parameters are passed in wrong order. If the parameter types are different, the compiler can detect the error [3], but if parameters of the same type are passed in wrong order compilers cannot detect the error. The example below shows the method `transfer` that can be called with incorrectly ordered parameters.

  Correct:
```
cout << "Enter amount to be transferred" << endl;
cin >> add;
account_obj->transfer(add, s);
cout << "New balance is -" << endl;
cout << " Savings " << s << endl;
```

  Incorrect:
```
cout << "Enter amount to be transferred" << endl;
cin >> add;
account_obj->transfer(s, add);
cout << "New balance is -" << endl;
cout << " Savings " << s << endl;
```

Note:
Syntax errors are not part of this paragraph because compilers can detect them.

Sources and types of failures

Various failures can arise during the execution of distributed systems. If components are executing on different machines connected by a network, network related problems like delays due to congestion or lost packets can occur, communication links can brake and computers may crash. The result visible to a system user is that clients do not get a correct response in time.

Server may fail and clients must be prepared to handle such situations. If a server throws an exception, the client must be able to react gracefully.

Security and authentication becomes an important issue, if computers are connected to the internet. Distributed systems interconnected around the word have to deal with processes whose primary intention is to cause damage. Both clients and servers have to be aware of security issues.

[3] Some data type errors may or may not be detected if the target language is C or C++.

4.6 Test adequacy criteria

The testing framework presented in this dissertation uses interface-based test adequacy criteria. Interface-based test adequacy criteria have the advantage that they are at a higher level than the traditional test adequacy criteria. This minimises the size of the coverage domains and reduces testing to what is necessary

If interface-based test adequacy criteria are used, precise definitions of all component interfaces must be available. To understand the components and their interfaces, it is necessary to analyse the system under test.

4.6.1 Architectural description

According to Shaw and Garlan [8], "The architecture of a software system defines that system in terms of computational components and interactions among those components". The software architecture of a system describes the different system components and explains how these components are put together.

Architectural descriptions help the software developer in a number of ways:

- During software development, deployment and maintenance an architectural system view is necessary to understand the system, its components and their interactions:
- The software architecture helps to match system requirements with the behaviour of the actual implementation.
- If a single component has to be modified, the effect of the modification on other components can be estimated through analysing the interconnections of the software architecture.
- Architectural views are useful during the evaluation of performance and reuse issues.
- Knowing the software architecture helps to understand the complexity of a system in terms of concurrency and resource sharing.
- Monitoring and fault injection mechanisms need the architectural system view to set up points for observing events and attaching faults.

In order to obtain architectural information very often the source code can be analysed. However, in the context of distributed systems, static analysis cannot be used to gather all information needed, because clients may not know all servers at compile time. Services may be discovered at runtime and requests for services may be realised through dynamic invocation interfaces. For distributed systems only dynamic analysis can be used to extract the complete system architecture with all components and interfaces.

4.6.2 Interface-based test adequacy criteria

The distributed components of CORBA-based systems have interfaces that are defined in IDL-files. The interface definitions contain the declarations of all interface methods and exception signatures. This information can be used to determine the test adequacy criteria AC_{met} and AC_{ex} based on method and exception coverage of a single server component.

$$AC_{met} = MethodCoverage = \frac{NumberOfMethodsExecutedByTheComponent}{NumberOfMethodsDefinedInTheComponentInterface}$$

$$AC_{ex} = ExceptionCoverage = \frac{NumberOfExceptionsThrownByTheComponent}{NumberOfExceptionsDefinedInTheComponentInterface}$$

However the coverage of methods and exceptions is not enough. The server implementation uses the parameters passed by the interface methods and the executed code paths depend on the values and types of these parameters. If a method is executed only once it is impossible to detect all hidden faults. Hence it is necessary to execute the methods with different parameter values.

Mutation techniques allow improving the fault detection efficiency. Mutation operators can be used to modify method parameters and return values in such a way that typical server or client failures are provoked.

Examples for mutation operations are:

- swapping parameters that have the same data type
- increment, decrement operators of simple data types
- set parameters to their boundary values
- nullify object references

A mutant is created if a mutation operation is applied to an original parameter of a method invocation. Mutation of complex data types can be done by applying mutation operators on their simple data types.

The test adequacy of a test set with respect to mutation coverage can be determined as follows:

$$AC_{mut} = MutationCoverage = \frac{NumberOfMutationsApplied}{TotalNumberOfMutations}$$

4.6.3 Overall test adequacy criterion

To determine the test adequacy achieved for the complete distributed system an overall coverage criterion that includes coverage domains from the client components as well as the server components must be defined. In this paragraph is explained how such an overall test adequacy criterion can be determined for a client-server system.

Server components must be tested first. Once all server components have been tested individually and the test set for each server has satisfied the interface-based adequacy criteria AC_{met}, AC_{mut} and AC_{ex}, the distributed system can be integrated. After integration the tester can create test sets for the client components, which need the server components to execute. As client components do not implement any interfaces, interface-based adequacy criteria cannot be used. Therefore the test cases used to test clients must satisfy a traditional test adequacy criteria. The test adequacy criteria for client components can be:

- AC_{dec} - decision coverage
- AC_{path} - path coverage

If all individual server components and client components are tested sufficiently the tester tries to achieve test adequacy with respect to an overall criterion for the entire system.

Each server component implements an interface. The notation C(*interface*, *domain*) defines the coverage domain of a server component with a defined interface-domain combination. The domain can be either method coverage, exception coverage or mutation coverage.

$$C_{Server} = \bigcup_{dom \in \{C_{met}, C_{ex}, C_{mut}\}} cup_{k=1}^{n} C(I_k, dom)$$

Decision coverage and path coverage are domains for client components.

$$C_{Client} = \bigcup_{dom \in \{C_{dec}, C_{path}\}} Client$$

For an entire system the overall coverage domain can be defined as:

$$C_{System} = C_{Client} \bigcup C_{Server}$$

To evaluate the goodness of test adequacy criteria, their effectiveness in finding errors must be considered.

If in each component are E_i errors, then the complete system contains

$E = \sum_{i=1}^{n} E_i$ errors.

In the following considerations E_{Client} represents the number of errors found by using only the coverage elements C_{Client} and E_{all} is the number of errors revealed by using C_{System}.

By comparing E_{all} with E_{Client}, the defined overall test adequacy criterion can be evaluated:

$E_{client} \leq E_{all}$ is a hypothesis that must be confirmed.

C_{System} uses a larger number of coverage elements and it is likely that more errors will be revealed with C_{System} as with C_{Client}. But the costs of a testing process using C_{System} are much higher than the costs caused by a test process that only uses C_{Client}.

If C_{Server} only performs redundant testing the assumption that E_{client} is smaller than E_{all} may be wrong. If this is the case the tester of a distributed system only has to develop test sets for the client components by using traditional test adequacy criteria.

However if $E_{client} \leq E_{all}$ is true, the relative cost of a test process using C_{System} can be determined.

An issue that is not considered by the defined overall coverage domain C_{System}, is that the various clients of a distributed system must be tested in parallel, because interleaving requests and parallel client actions may also cause system failures.

4.7 Summary

As part of my dissertation work a prototypical testing framework for CORBA-based distributed system was developed. Finding a test methodology that specially focuses on distribution issues and identifying common faults that can occur in distributed systems were activities during the first phase of my dissertation. This chapter has discussed these issues and has described a testing methodology that specially considers concurrency issues and scalability problems as well as a fault injection technique to evaluate the fault-tolerance properties of a distributed system.

The next chapter is analyses the design and implementation of the prototypical testing framework, which realises the chosen monitoring and fault injection methodology.

The usage of the testing framework is explained with the help of a sample application in chapter 6.

5 Architecture of the testing framework

The proposed testing framework allows monitoring the individual components of the system under test, to control the test execution and to meter the services offered by the system under test. The monitoring of the system under test is realised by observing the messages passed between its distributed components. Controlling the test execution means that the tester is able to specify and execute user actions during the test execution. Metering refers to a measurement of the usage and performance of the system components.

The following goals played a major role, during the architectural design of the testing framework

- ease of use
- non-obtrusiveness of the testing framework
- completeness of the information obtained via monitoring
- timeliness of the information obtained via monitoring
- scalability of the testing framework with respect to the number of components

The final version of the testing framework will consists of five main modules:

1. Decentralised listeners are used as distributed buffers to store temporarily the monitored method invocations.
2. A central test control communicates with the decentralised listeners and polls the buffered data.
3. A database system stores the monitored data.
4. A data processor analyses the monitored data to present different views that help to improve the test process.
5. A parser automatically instruments the Java components of the system under test.

The first version of the testing framework implements the modules 1 and 2. The architecture of the first version of the testing framework, its classes and their interactions are described in this chapter.

5.1 Main components

The testing framework consists of decentralised and centralised components. Figure 5.1-1 shows an overview of the implemented first version of the testing framework. Decentralised components are visualised through dashed lines and centralised components are indicated through dotted lines.

Wrappers and listener objects are decentralised. They are associated with a component of the system under test and for every component of the system under test exists an instance of them. They are created automatically via so called service loaders during the start-up of a their associated component.

The test controller is a centralised component. Only one instance of it is needed.

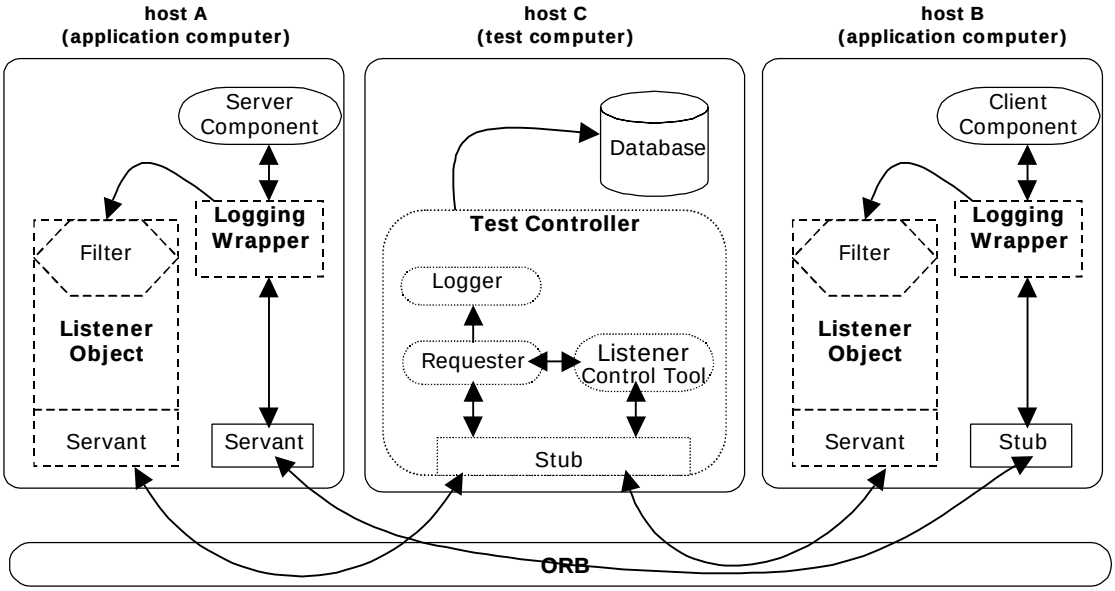


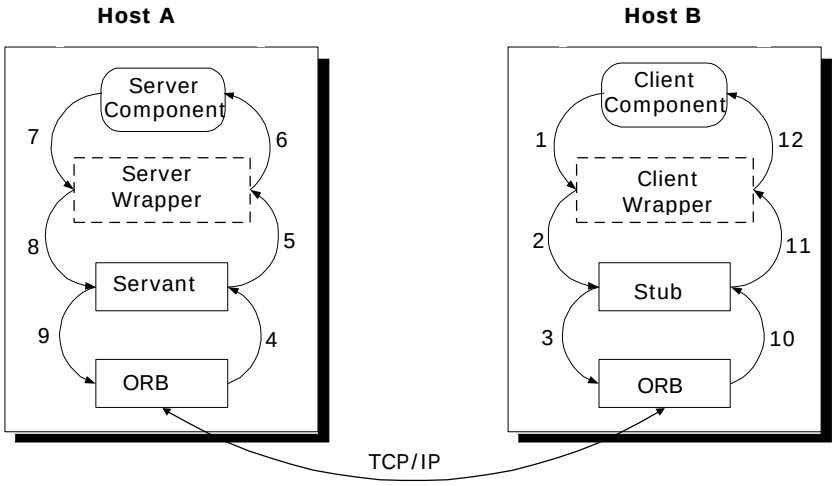
figure 5.1-1 overview of the main components of the testing framework

The following sub-chapters of 5.1 give an overview about the functionality of the three main components of the testing framework: logging wrapper, listener object and test controller. After that from chapter 5.2 onwards the internal design of these components and of the testing framework is described.

5.1.1 Logging wrapper

The data, which is gathered by the testing framework, consists of component interface related information, like method signatures and exceptions. Specialised components, called wrappers, are used to get this information. They act as a door for the components of the system under test. For every CORBA interface there are two wrappers: one on the client side and one on the server side. The wrapper on the client side belongs to the stub and the wrapper on the server side belongs to the servant of the interface. A client request passes the client side wrapper before being sent to the server. And before it arrives at the server components it has to pass another “door”, the server side wrapper.

The figure 5.1.1-1 shows what happens if a client component calls a CORBA interface method of a server component. On the way from the client to the server the request message has to pass first the client wrapper and then the server wrapper. When the response message is sent back by the server to the client, happens the same, but in reverse order.



5.1.1-1 the wrapper mechanism

When the request or replay message of a method invocation pass through a wrapper all information about the method invocation is available and therefore it is possible to implement a simple wrapper that logs method parameters and return value. With more sophisticated wrappers the tester can determine in addition the time needed for the execution of a request or even change parameters and return values, throw exceptions or delay the passed messages.

The wrappers used by the testing framework must implement the string `myCorbaInterfaces`. This string describes details about the methods of the wrapped interface. The source listing below shows an example:

```
private static String myCorbaInterfaces =
"[ClientReceptionWrapper]void messageIn( String theSender, String
                                                    theMessage )\r\n"+
"[ClientReceptionWrapper]boolean signOn( String clientID )\r\n"+
"[ClientReceptionWrapper]boolean signOff( String clientID )";
```

During start-up of the testing framework each wrapper sends its string `myCorbaInterfaces` to the local listener object. During test execution the wrappers are used to log all method invocations that take place. The logged details about incoming and outgoing requests or responses are sent to the local listener objects of the testing framework.

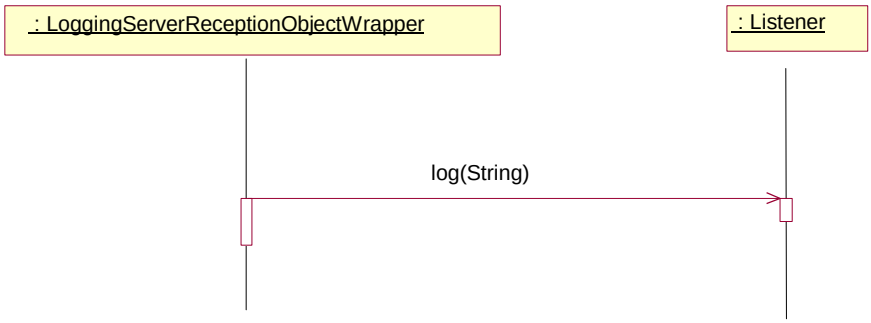


figure 5.1.1-2 communication between the wrapper and the local listener

5.1.2 Listener object

In addition to the decentralised interface wrappers every computer used by the system under test hosts a listener object. The listener object acts as a data buffer for interface wrappers. During start-up it receives information about the interface methods and later during test execution it stores details of requests and responses. The wrappers send the monitored data to the local listener object and the listener object buffers the information about the monitored method invocation.

To reduce the amount of test data the details logged by the wrappers primarily have to pass a filter such that only the actually wanted information is stored. The listener object has a filter for each interface method of its component. When a method is invoked and the data is sent to the listener object, the listener checks if there is a filter attached to the method. If the received data passes the filter, it is stored in the buffer of the listener object.

The listener object has a CORBA interface such that a centralised test controller hosted by a dedicated test computer can request the buffered data. The interface also allows to get information about the momentarily set filters, the tester can modify the filter configuration and it is possible to clear the buffer of the logged data.

5.1.3 Test Controller

To monitor and control the system under test there is a central component called test controller. This component is not associated with a special component of the system under test. The test controller has a graphical user interface to display the data logged during test execution. It allows the tester to determine the test coverage by viewing the executed methods, the exceptions thrown and the used parameters. To generate a hierarchical view of the system under test the test controller has the ability to analyse the name space of the system under test.

Listener control tool

The listener control tool provides a graphical user interface to specify filters. It enables the tester to set and get filter configurations, reset the local listeners, start a test run and stop a test run. Dedicated dialog windows help to perform these control actions.

Requester

Like the listener control tool the requester is a sub-component of the test controller and is located on the test computer. The requester contains a thread that can be started and stopped by the tester through the listener control tool. When the thread is running it cyclically polls all local listeners to collect their buffered data. After receiving the data, the requester sends it to the logger.

Logger

In the current version of the testing framework the logger stores the gathered data in log files – in the future a central database system will be used.

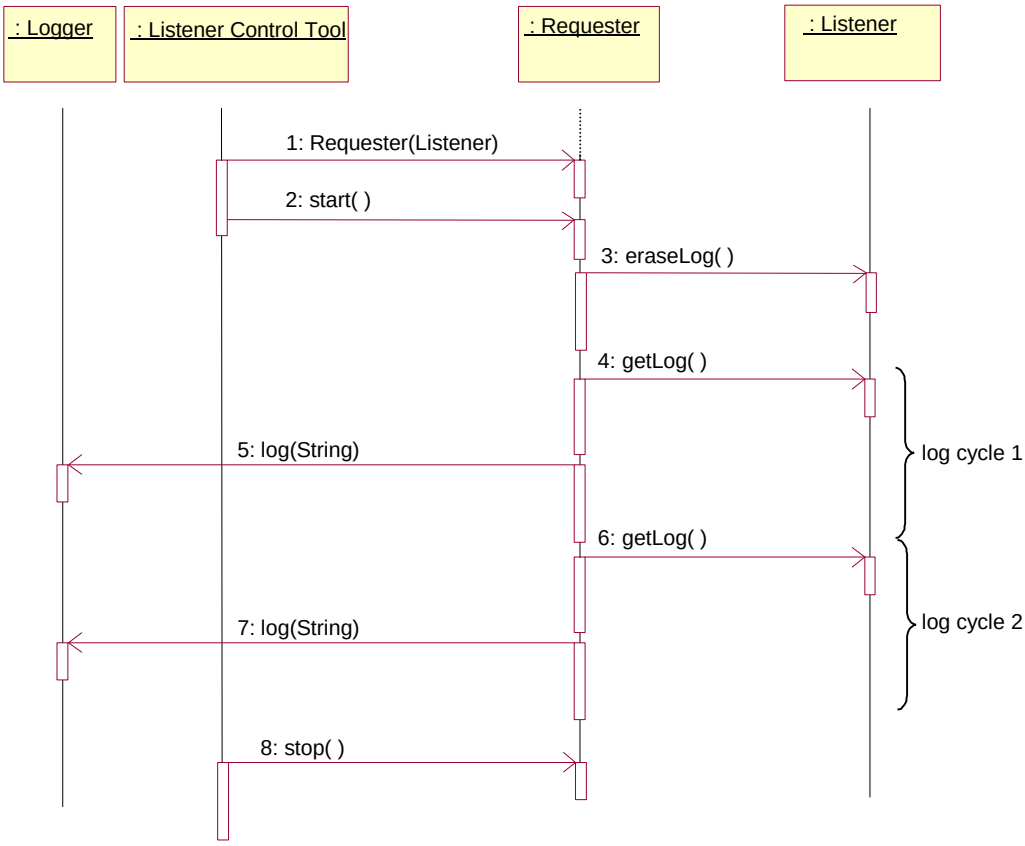


figure 5.1.3-1 the test control polls the local listener objects

5.2 Logging wrapper

This chapter describes in detail the feature of the logging wrappers, which are used by the testing framework to monitor the system execution. Logging wrappers realise defined activities when a client invokes a method on a bound object or when a server receives an operation request. In the first case the logging wrappers are invoked before the operation request is marshalled and it possible that the logging wrappers interfere before the operation request is sent across the network.

In the context of the first version of the testing framework the logging wrappers have two main tasks:

- They log information about the operation requests issued by a client and received by a server.
- They measure the time required for operation requests to complete.

5.2.1 Typed and un-typed wrappers

The VisiBroker CORBA implementation offers two kinds of object wrappers: typed wrappers and un-typed wrappers. It is possible to use both wrapper types within a single application. The table on the next page summarises and compares the features of the two object wrappers.

Features	Typed wrapper	Un-typed wrapper
The wrapper receives all method arguments passed to the stub.	Yes	No
The wrapper can return control to the caller without actually invoking the stub or the server implementation.	Yes	No
One wrapper can be used for all methods of a CORBA interface.	No	Yes

The testing framework uses typed wrappers, because for testing purposes it is important to have access to all method arguments.

5.2.2 Generation

To use typed or un-typed object wrappers the option `-obj_wrapper` must be chosen when invoking the `idl2java` compiler. With this option the compiler generates an object wrapper base class for the compiled interface with the name `<interface_name>ObjectWrapper`. In addition the compiler generates a `Helper` class with methods for adding or removing the object wrappers.

5.2.3 Implementation

The steps needed to implement a logging wrapper are listed below:

1. Identify the CORBA interfaces for which logging wrapper are needed.
2. Execute the `idl2java` compiler with the option `-obj_wrapper`.
3. Derive the logging wrapper class from the object wrapper base class `<interface_name>ObjectWrapper` generated by the IDL compiler.
4. Implementation the logging mechanism needed to monitor all existing interface methods.

The implementation of a logging wrapper defines what happens when an interface method is invoked. A logging wrapper can implement any kind of activity for the different methods of the wrapped interface.

5.2.4 Registering the logging wrappers

Client-side logging wrappers can be registered by invoking the method `addClientObjectWrapperClass`. Similarly server-side wrappers are registered by invoking the method `addServerObjectWrapperClass`. Both registrations are realised with the `Helper` class generated by the IDL compiler. Client-side and server-side wrappers must be registered after `ORB.init()` is called, but before any objects are bound or any server implementation services a request.

The following code listing shows the section of the file `Init.java`, in which the logging wrappers are created.

```
// install the LoggingServerReceptionObjectWrapper
owrp.TypedWrappers.LoggingServerReceptionObjectWrapper.setListener
                                                    ( theListener );
Class c =
     owrp.TypedWrappers.LoggingServerReceptionObjectWrapper.class;
Comm.ServerReceptionHelper.addClientObjectWrapperClass(orb, c);
Comm.ServerReceptionHelper.addServerObjectWrapperClass(orb, c);

// install the LoggingClientReceptionObjectWrapper
owrp.TypedWrappers.LoggingClientReceptionObjectWrapper.setListener(
                                                          theListener );
Class cc =
     owrp.TypedWrappers.LoggingClientReceptionObjectWrapper.class;
Comm.ClientReceptionHelper.addClientObjectWrapperClass(orb, cc);
Comm.ClientReceptionHelper.addServerObjectWrapperClass(orb, cc);
```

The ORB keeps track of any logging wrapper that has been registered. When a client invokes the `bind` method to bind to a wrapped object, the necessary object wrapper is created. If a client binds to more than one instance of a wrapped object, each instance will have its own wrapper. If a server creates more than one instance of a wrapped object, a dedicated wrapper will be created for each instance.

5.2.5 Service initializer

Using the following command-line parameter when starting the system components activates the logging wrappers:

`-DORBservices=owrp.Init`

The logging wrappers are defined in the package `owrp` that includes the service initializer, called `Init.java`. This initializer is invoked automatically during the initialisation of the ORB if `-DORBservices=owrp.Init` is specified on the command-line when starting the client or server components of the system under test.

The service initializer is used by the testing framework to start all decentralised objects necessary for testing purposes that are associated with the distributed components of the system under test.

The service initializer executes the following steps:

1. create a local listener object
2. register the logging wrappers of the CORBA interfaces
3. start the local listener

5.3 Test controller

The test controller consists of various classes. The diagram below shows these classes and their relationships in the logical design of the test controller.

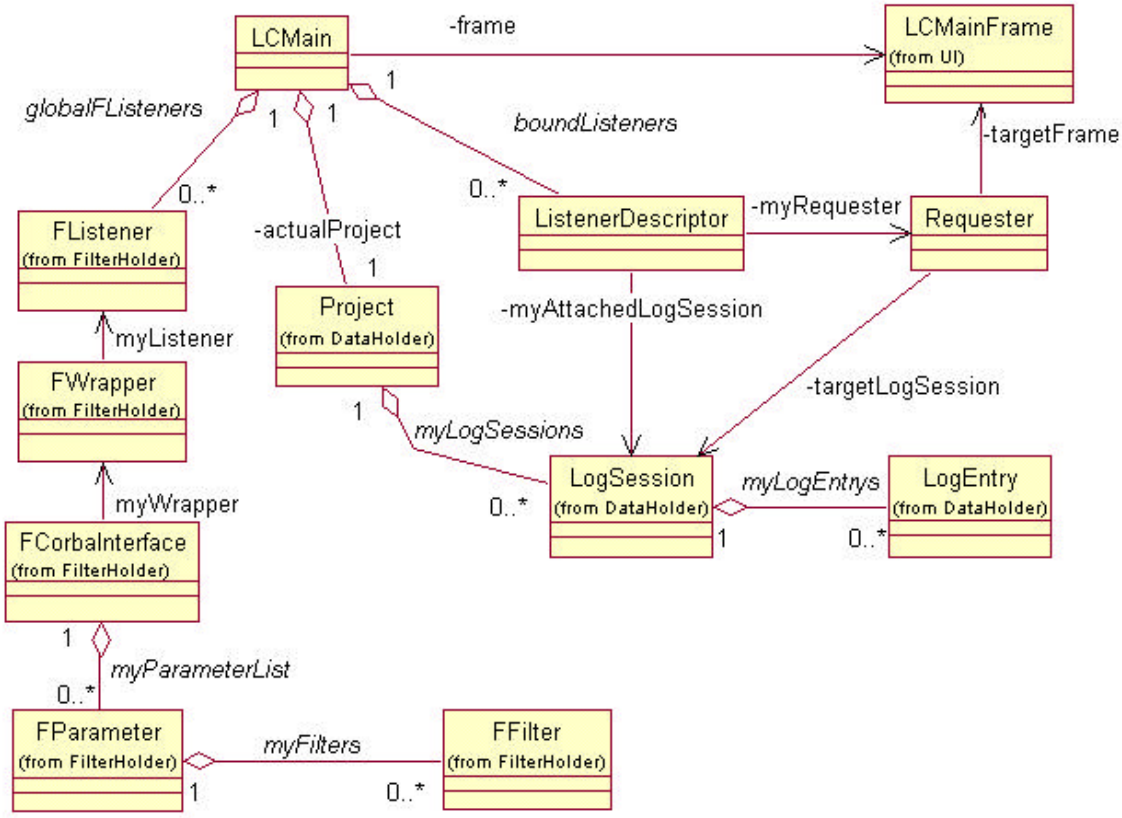


figure 5.3-1 overview of the classes of the test controller

The class diagram 5.3.-1 represents the class structure of a test controller. Each of the different classes is specified in the following paragraphs of this chapter.

The different class specifications describe the properties, operations and relationships of the individual class in the context of the test controller. The properties and operations are listed inside the rectangles representing the classes in the class diagram. Relationships describe how closely two classes are related. They are visualised through the different relationship arcs between two classes.

5.3.1 LCMain

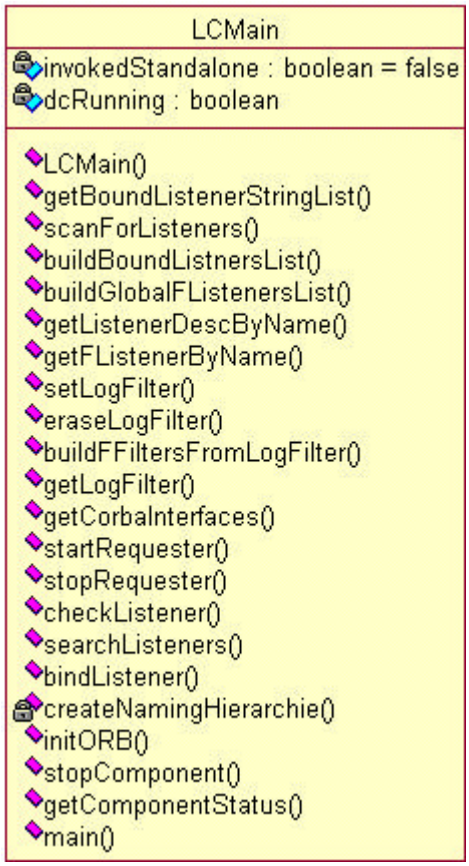


figure 5.3.1-1 functions and attributes of the class LCMain

The class LCMain has an aggregate relationship with three classes, namely FListener, Project and ListenerDescriptor. LCMain is the aggregate class at the client end of the aggregate relationship. This signifies that LCMain contains instances of FListener, Project and ListenerDescriptor.

The aggregate relationship is used to show that an instance of LCMain has the ownership of its FListener, Project and ListenerDescriptor and the relationship names identify the type or purpose of the relationships:

- globalFListeners
  The relationship globalFListeners specifies the cardinality 0..* for the supplier Flistener and the cardinality 1 for the client LCMain. This shows that LCMain contains an instance of FListener for all listeners available in the system.
- actualProject
  The object Project associated with LCMain represents the actual valid test project.
- boundListeners
  The relationship between LCMain and ListenerDescriptor is similar to the relationship between LCMain and FListener. LCMain own a ListenerDescriptor for every bound listener.

5.3.2 ListenerDescriptor

ListenerDescriptor is a central representation of the distributed listener objects. It creates a Requester for its listener and passes the actual LogSession to the Requester.

figure 5.3.2-1 functions and attributes of the class ListenerDescriptor

5.3.3 Requester

Requester is a thread, which frequently polls the distributed listener objects and stores the strings logged by the listener objects in the object LogSession and in the LogStringList of the window LCMainFrame.

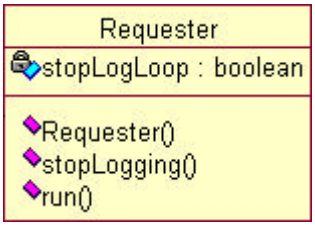


figure 5.3.3-1 functions and attributes of the class Requester

5.3.4 Test data

The test data is organised as a simple tree. The tester creates a new test project or chooses an existing one, which becomes the current test project. The current test project is represented by an instance of the class Project. It contains various object of type LogSession and every LogSession object owns an unlimited number instances of LogEntry.

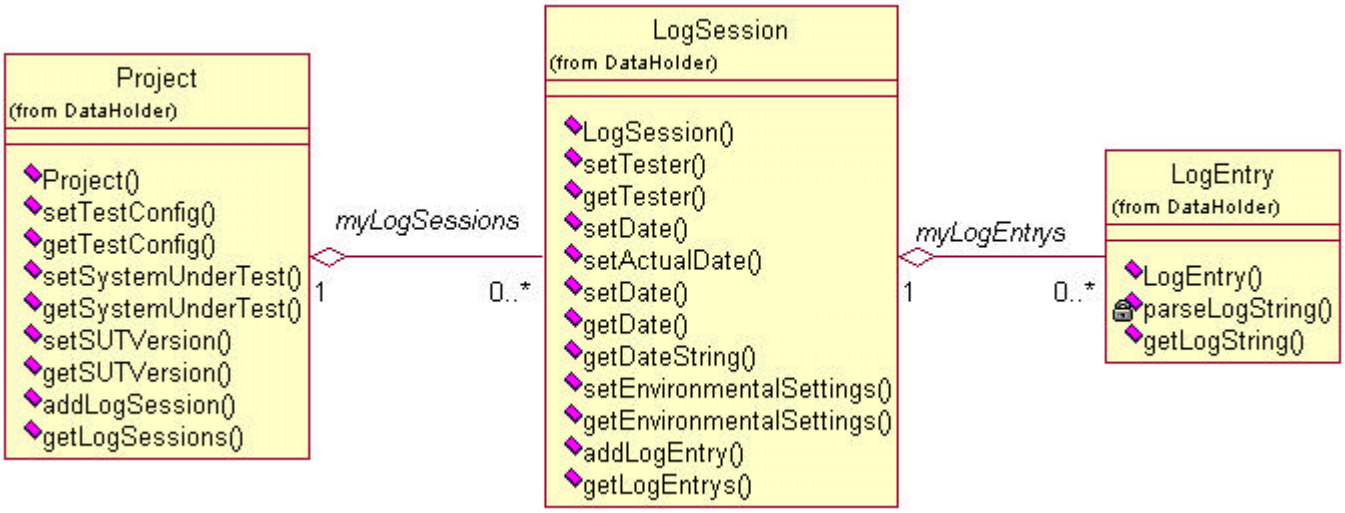


figure 5.3.4-1 tree organising the test data

5.3.5 Filter Hierarchy

The testing framework uses the following filter hierarchy:

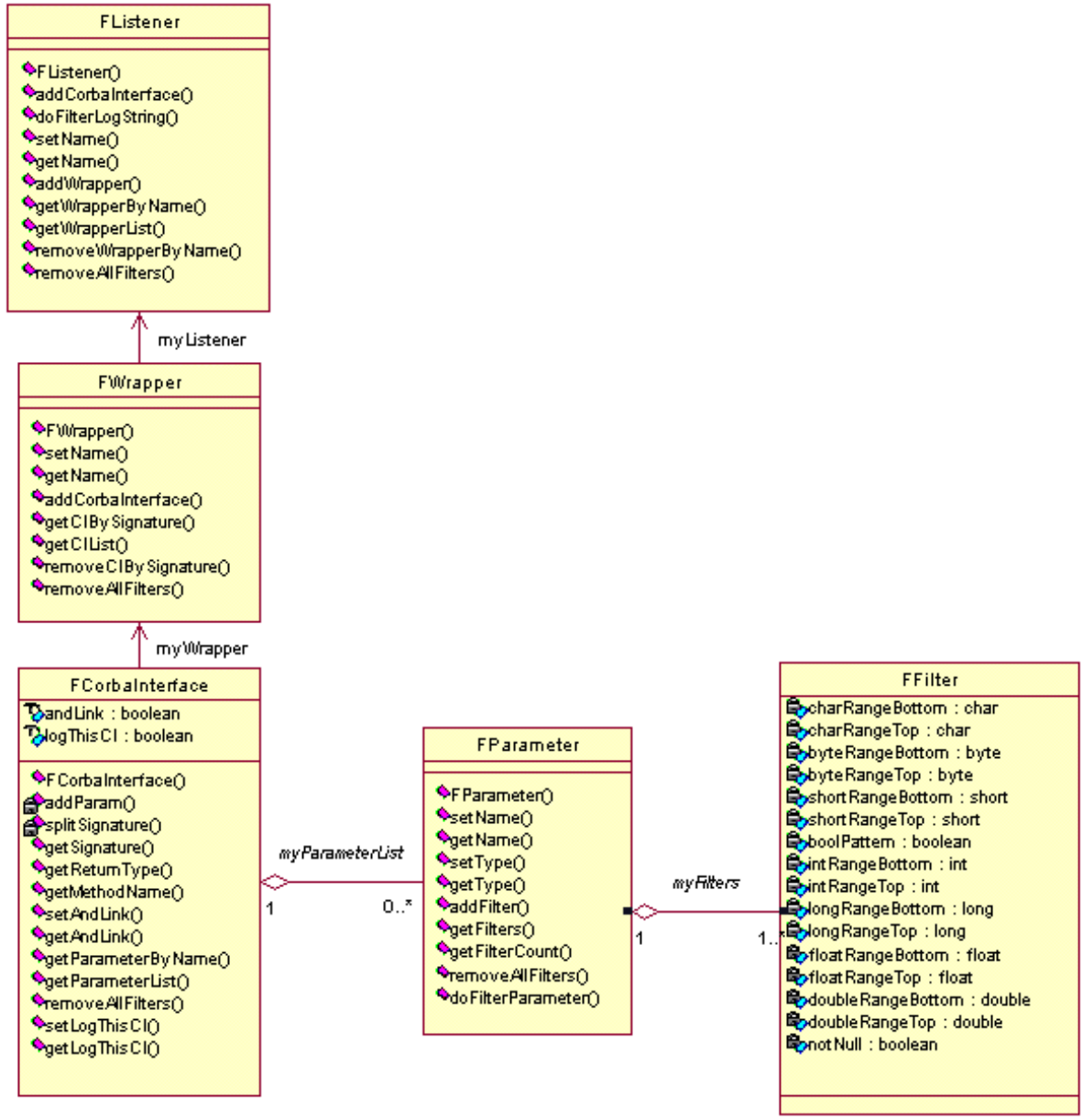


figure 5.3.5-1 filter hierarchy

FListener

Every component of the system under test has is own listener.

FWrapper

Every CORBA interface of the system under test is wrapped by its own dedicated wrapper.

FCorbaInterface

Every interface method of the system under test has a virtual representation through an instance of FCorbaInterface.

The class FCorbaInterface owns a list of FParameter objects that are used to set and modify the filters applied on the different elements of the interface method.

The various filters of an interface method can be joined by an AND- or an OR-relationship.

FParameter

Various elementary filters can belong to one parameter. The complex filter FParameter contains all elementary filters, called FFilter, of one parameter.

The FFliter objects are applied together: If the parameter matches only one of the elementary filters, it is said that the parameter matches complex filter FParameter.

FFilter

Instances of the class FFilter are used to determine whether a parameter matches a filter or not:

- A numerical parameter (e.g. byte, short, int, long, float, double) matches a filter, if its actual value lies inside a specified range.
- A string or a char matches a filter, if it contains a special string pattern.
- A boolean matches a filter, if it has the same value (false or true) as defined by the filter.

For non-primitive parameter types FFilter can only differentiate between Null and not Null.

5.4 Listener object

The Listener objects are located on every machine hosting a component of the system under test. The class diagram below visualises the relationship of Listener with the other classes, which belong to the testing framework and reside on distributed computers.

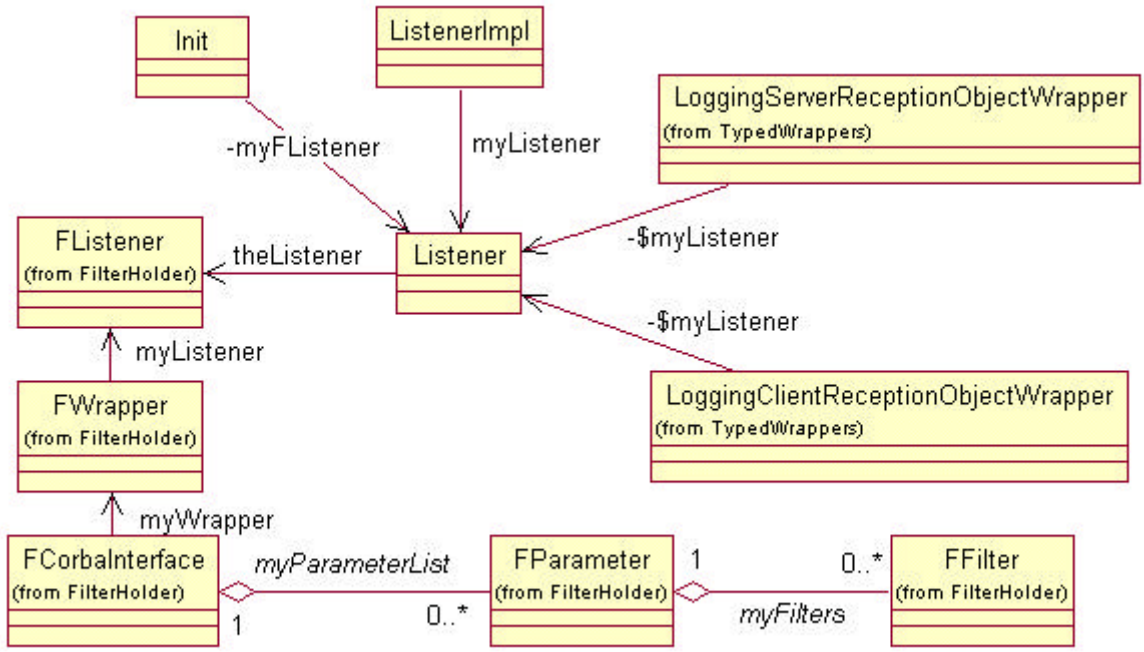


figure 5.4-1 overview of the classes related to the Listener

Like the class diagram of the test controller figure 5.4-1 shows classes FListener, FWrapper, FCorbaInterface, FParameter and FFilter, which have been described in paragraph 5.3.5. The functionality of the other classes is explained now.

5.4.1 Init

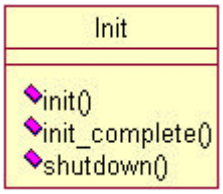


figure 5.4.1-1 functions of the class Init

The class Init is used to initialise the local wrappers at system start-up.

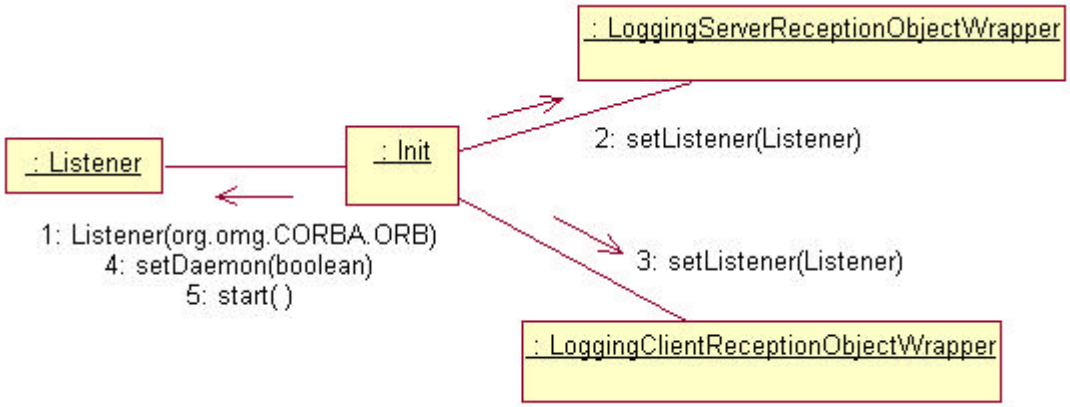


figure 5.4.1-2 collaboration diagram: initialisation of the wrappers

5.4.2 Listener

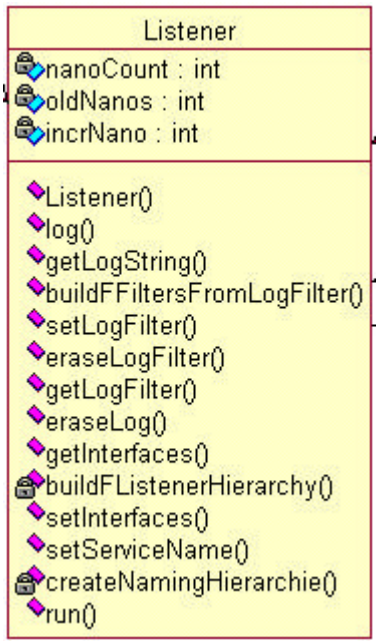


figure 5.4.2-1 functions and attributes of the class Listener

The class Listener has the method setLogFilter. The collaboration diagram below shows what happens in the Listener if SetLogFilter is called.

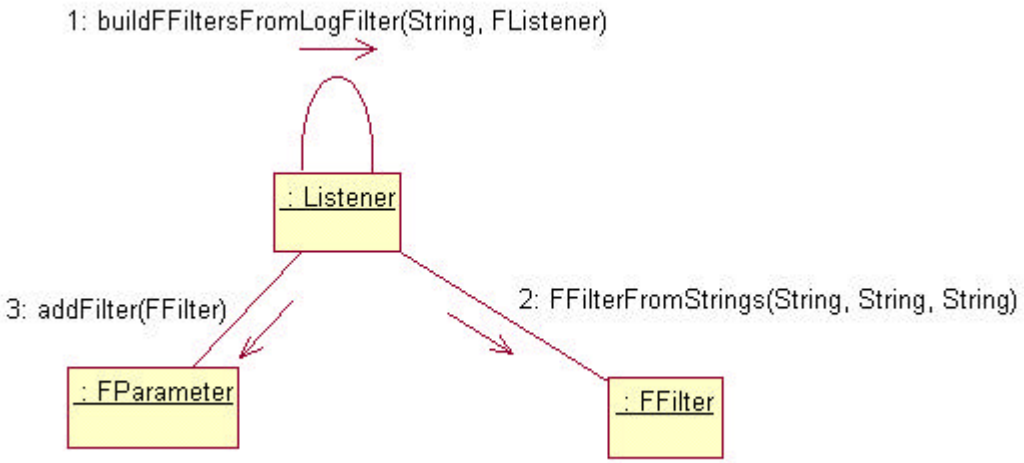


figure 5.4.2-2 collaboration diagram: method SetLogFilter

Another method of Listener is setInterfaces. If the method setInterfaces is evoked, the filter hierarchy is build up. The interfaceDefinitionString is passed to the FListener object and the objects FWrapper, FCorbaInterface and FParameter are created.

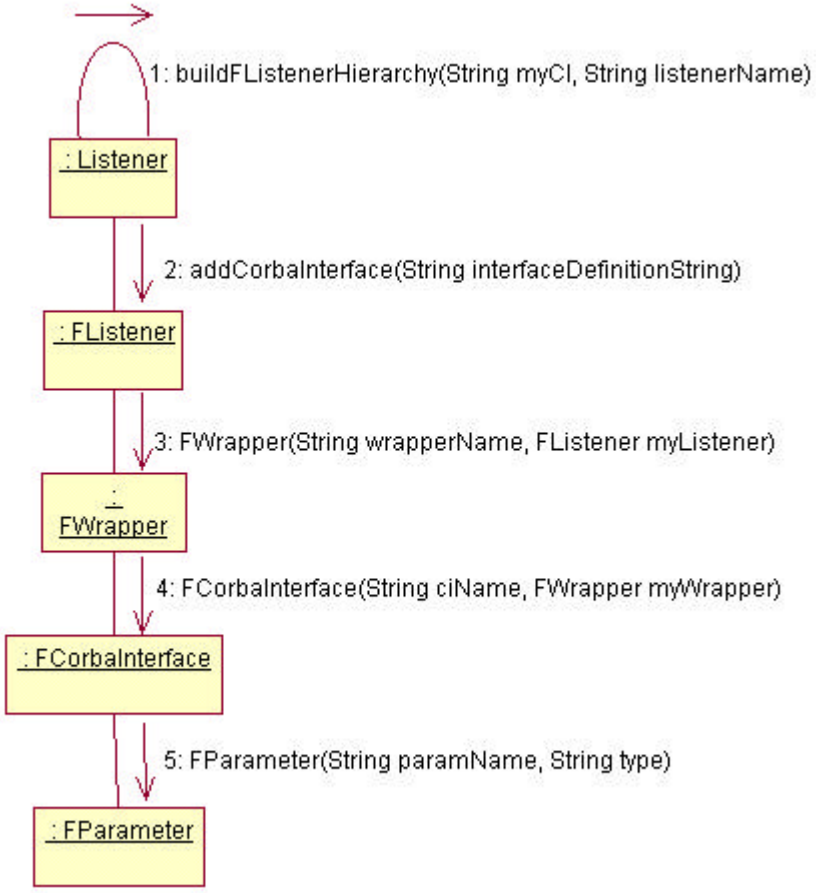


figure 5.4.2-3 collaboration diagram: method SetInterfaces

5.4.3 ListenerImpl

figure 5.4.3-1 functions of the class ListenerImpl

ListenerImpl is the implementation of the CORBA interface Listener.

```
public class ListenerImpl extends ListenerPOA
{
     ...
}
```

The CORBA interface methods of Listener and their usage are listed below:

- setLogFilter(String)
  command to set the log filter in the listener
- getLogFilter()
  command to get the log filter from the listener
- getInterfaces()
  command to get the wrapped CORBA interfaces from the listener
- eraseLog()
  command to erase the saved log entries in the listener
- getLogString()
  command to get the saved log entries from the listener

The following format is used to exchange information about the CORBA interfaces:

- [WrapperName]returnType methodName(TypeA varA, TypeB varB,...)

The following format is used to exchange the logged data:

- Primitive Java data types like byte, char, double, float, int, long, short and boolean are logged directly.
- Strings are transferred without the symbol ". The real length of a string must be added at the end of the string:
  [String(length)]
- for other data types it is only differentiated between null and not null
  if(tmp != null)
      objAdr = new String("notnull");
  else
      objAdr = new String("null");

Example:
SERVER[ClientReceptionWrapper: PREmethod]->calling void messageIn([theSender,String(7)]clientA, [theMessage,String(5)]hallo);

The format used to exchange data about filters is described in the figure below:

```
// format of filterString (1..5 --> one Filter)
// 1 "[WrapperName]ReturnType methodName( TypeA varA, TypeB varB, ... )"\r\n
// 2 "TypeA"\r\n
// 3 "varA"\r\n
// 4 [RangeBottom OR StringPattern OR BooleanPattern OR "notNull" OR "Null"]\r\n
// 5 [RangeTop]\r\n
// 1 "[WrapperName]ReturnType methodName( TypeA varA, TypeB varB, ... )"\r\n
// 2 "TypeB"\r\n
// 3 "varB"\r\n
// 4 [RangeBottom OR StringPattern OR BooleanPattern OR "notNull" OR "Null"]\r\n
// 5 [RangeTop]\r\n
//  ...
//  or, to log nothing from a CorbaInterface
//  "[WrapperName]ReturnType methodName( TypeA varA, TypeB varB, ... )"\r\n
//  "-"\r\n
//  "-"\r\n
//  "-"\r\n
//  "-"\r\n
//  or, to erase all Filters
//  ""
```

figure 5.4.3-2 filter string format

5.4.4 LoggingServerReceptionObjectRapper

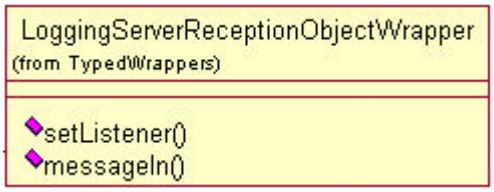


figure 5.4.4-1 functions of the class LoggingServerReceptionObjectRapper

The logging wrapper LoggingServerReceptionObjectRapper extends the wrapper base class ServerReceptionObjectWrapper that the IDL compiler generates automatically during the compilation of the interface ServerReception. Its functionality is described in detail through explanations in the complete source code listing below:

```
public class LoggingServerReceptionObjectWrapper extends
                            Comm.ServerReceptionObjectWrapper
{
```

A logging wrapper always belongs to an interface. The string myCorbaInterfaces describes the names, parameters and the return values of the interface methods.

```
  private static String myCorbaInterfaces =
                      "[ServerReceptionWrapper]void messageIn
                      ( String theSender, String theMessage )";
```

Every logging wrapper needs a listener object to which it can send the logged data.

```
  private static Listener myListener;
```

The method setlistener is called at the beginning of the testing phase to initialise the attribute myListener with the actual listener object. Then the logging wrapper sends its method description to the listener.

```
public static void setListener( Listener theListener )
{
  myListener = theListener;
  myListener.setInterfaces( new String( myCorbaInterfaces ) );
}
```

The method messageIn is the wrapped CORBA interface method. The logging wrapper has to trace all invocations of this method.

```
public void messageIn (java.lang.String theSender,
                       java.lang.String theMessage)
{
```

When the method invocation arrives, the wrapper sends a string that describes the name and parameters of the invocation to its listener.

```
String logname = new String( this.getLocation().toString() );
long m_Time;
long diff;
myListener.log(
  logname +
  "[ServerReceptionWrapper: PREmethod]"+
  "->calling void messageIn([theSender,String("+
  theSender.length()+
  ")]"+
  theSender+
  ", [theMessage,String("+
  theMessage.length()+
  ")]"+
  theMessage+
  " )"
);
```

Then the actual time is determined and the real server implementation of the interface method is called.

```
m_Time = System.currentTimeMillis();
super.messageIn( theSender, theMessage );
```

The time needed by the server to execute the method is determined.

```
diff = System.currentTimeMillis() - m_Time;
```

Finally the logging wrapper sends another string to its listener. Now the string contains in addition to the information of the first string the time needed by the server (the value of diff) and if there is a return value it can also be transmitted to the listener.

```
myListener.log(
      logname +
      "[ServerReceptionWrapper: POSTmethod]" +
      "->called void messageIn( [theSender,String("+
      theSender.length()+
      ")]"+
      theSender+
      ", [theMessage,String(" +
      theMessage.length()+
```

```
            ")]" +
            theMessage+
            " )" +
            ",Duration:" +
            diff+
            "ms"
    );
  }
}
```

5.4.5 LoggingClientReceptionObjectRapper

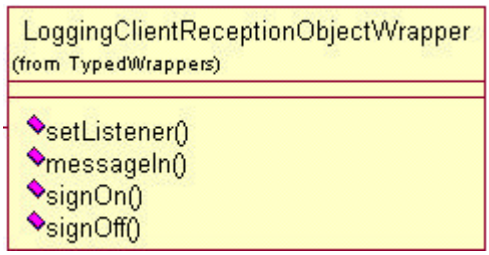


figure 5.4.5-1 functions of the class LoggingClientReceptionObjectRapper

The functionality of the class LoggingClientReceptionObjectRapper is very similar to the functionality of the class LoggingServerReceptionObjectRapper, which is described in detail in chapter 5.4.4.

6 Usage of the testing framework

This chapter describes the usage of the testing framework. In 6.1 some general aspects of the testing framework and an exemplary system under test are introduced. In 6.2, 6.3 and 6.4 I am describing how to use the testing framework during three distinct phases of the testing process namely pre-test-phase, testing-phase and post-test-phase.

Illustrations in form of screen shots are used to visualise the features of the testing framework.

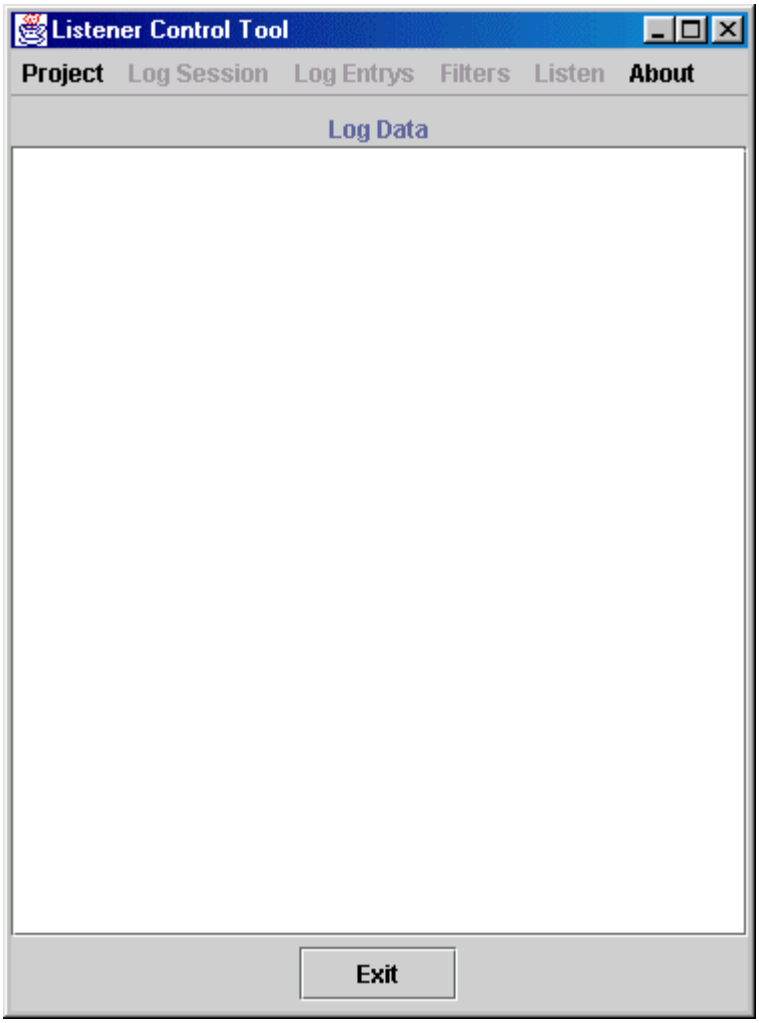


figure 6-1 main window of the testing framework

6.1 Introduction

The testing framework is intended for distributed systems and helps the tester to perform system integration tests. Its focus lies on the interfaces between the distributed components of the system under test. The testing framework itself is written in Java and it can be used to test distributed systems that consist of Java components and use the VisiBroker CORBA implementation.

During the description of the usage of the testing framework some terms are used with a special meaning more than once. Their signification in the context of the testing framework is explained in this paragraph.

- A **project** is used to denote a test configuration. It defines the following configuration items: name of the test configuration, name of the system under test and version of the system under test.
  A project can have several sessions with the same project settings.
- A **session** is used to denote a test run executed by a named person at a specified time, with defined environmental conditions.
- **Filters** allow to monitor the test execution based on a selection made by the tester in order to determine different coverage values, evaluate system performance or view only special method invocations.

6.1.1 System under test

The best way to explain the usage of the testing framework is to choose an exemplary system under test. In the following paragraphs a chat system for distributing messages between different chat clients is used as example system. The chat system consists of at least three components:

- `ChatSystemServer`
  One component that receives and sends messages to the registered chat clients.
- `ChatSystemClient`
  Two or more components that have an user interface to broadcast messages.

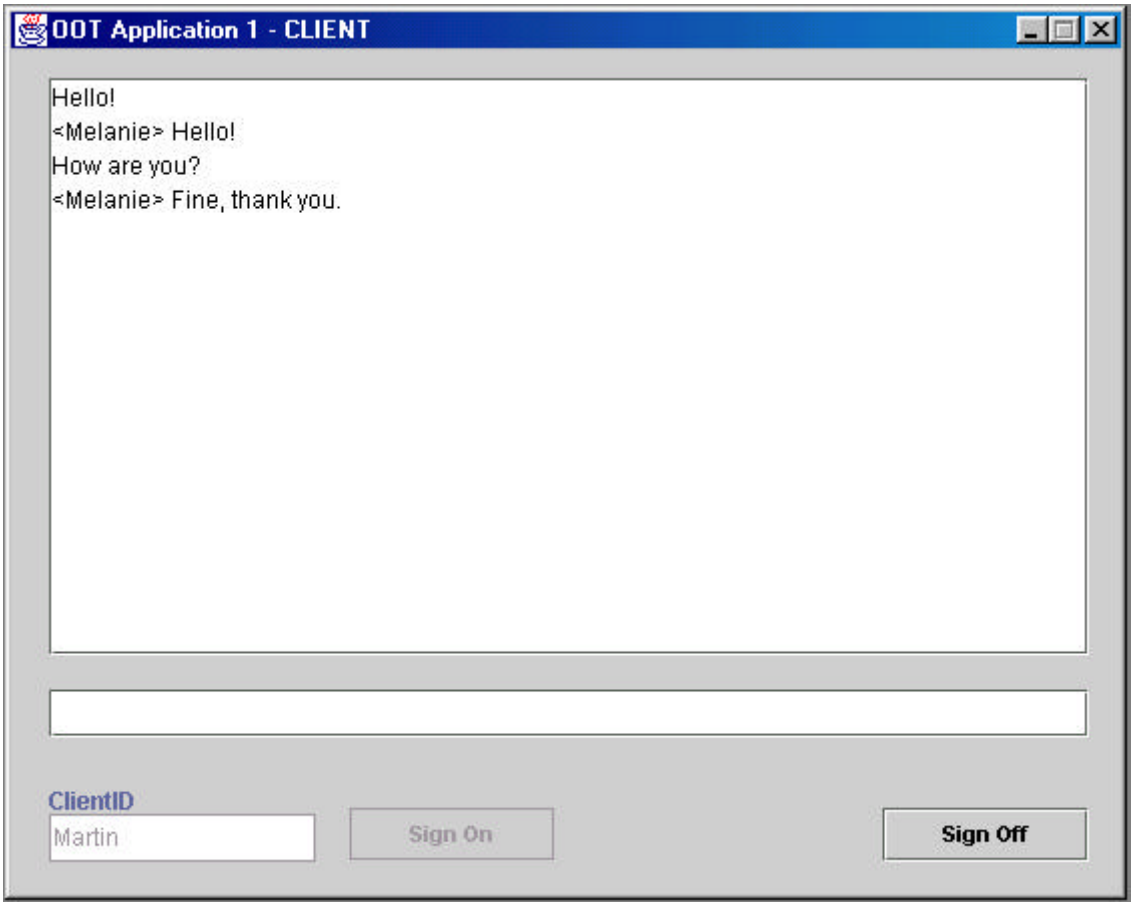


figure 6.1.1-1 user interface of the `ChatSystemClient`

The CORBA interface `ClientReception` is provided by the `ChatSystemServer` and it is used by the `ChatSystemClients`. The interface methods of the `ChatSystemServer` are defined in the file `Comm.idl`.

```
interface ClientReception
  {
    exception AlreadySignedOn{};
    boolean signOn( in string clientID ) raises( AlreadySignedOn );
    boolean signOff( in string clientID );
    oneway void messageIn( in string theSender,
                           in string theMessage );
  };
```

The CORBA interface `ServerReception` is provided by the `ChatSystemClients` and the ChatSystemServer uses it. The interface of the `ChatSystemClients` is also described in the `Comm.idl` file.

```
interface ServerReception
  {
    oneway void messageIn( in string theSender,
                           in string theMessage );
  };
```

The interfaces `ClientReception` and `ServerReception` contain the method `messageIn`. The method `messageIn` is defined as a one way function. This has the advantage that the calling component does not block until the invoked method has been executed.

The figure below visualises what happens if an already registered chat client “Melanie” broadcasts the message “Hello”.

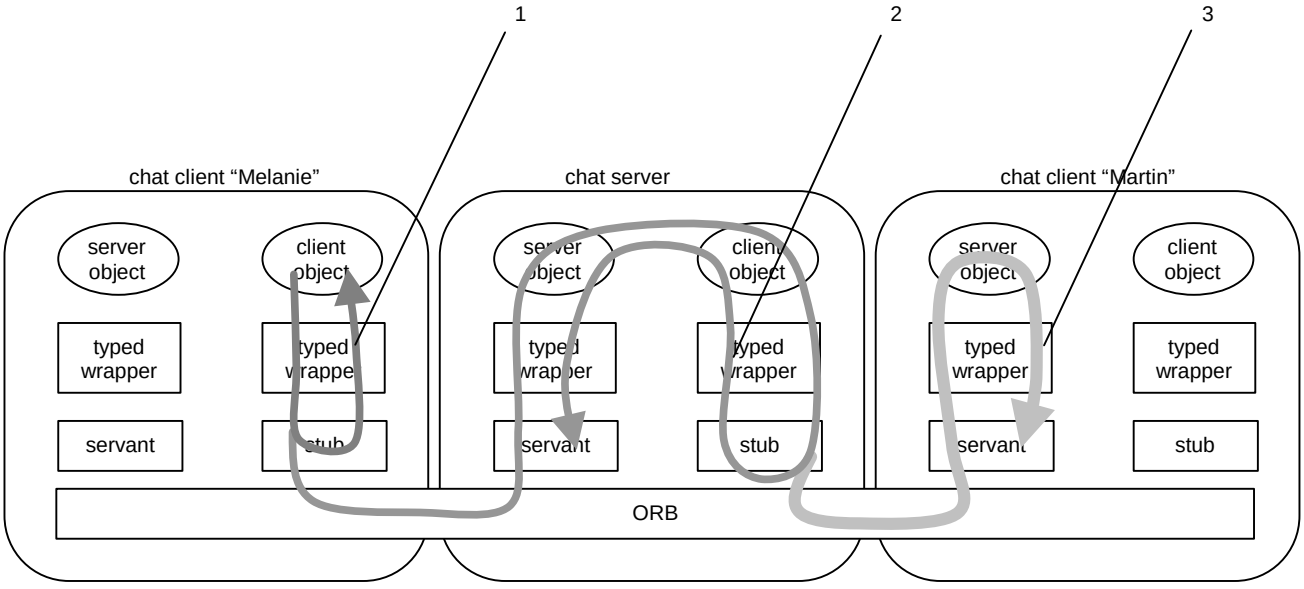


figure 6.1.1-2 invocation of an interface method

1. The chat client “Melanie” calls the CORBA interface method `messageIn` of the chat server. The method invocation returns immediately as soon as the invocation has been accepted by the ORB.

2. The chat server receives the message “Hello” from the chat client “Melanie” through its CORBA interface method `messageIn`. During the execution of the method implementation the chat server send the received message “Hello” to all registered chat clients by calling the CORBA interface methods `messageIn` of the chat clients.
3. The chat client “Martin” receives the message “Hello” from the chat server and displays the message in its GUI.

6.1.2 Packages

The class-files of the system under test are organised in the package `ChatSystemClient` and `ChatSystemServer`. The classes generated automatically by the IDL compiler are in the package `Comm`. To ensure this structure the IDL file is organised like in the code listing below:

```
module Comm
{
  interface ...
  {
     ...
  };
  interface ...
  {
     ...
  };
}
```

The classes needed for testing purposes are organised in the package `owrp` and `ListenerControl`.

The package `owrp` has three sub-packages, namely `FilterHolder`, `ListenerComm` and `TypedWrappers`, which contain the classes of the testing framework that are needed for every component of the system under test. The components of the system under test may reside on distributed computers and every component needs its dedicated wrapper, a local listener and the filter objects.

The centralised part of the testing framework is implemented in the package `ListenerControl`. The sub-package `Ui` contains all classes needed to offer the tester a graphical user interface to control the testing process. The classes needed to store test configurations and the data logged during the test process can be found in the sub-package `DataHolder`.

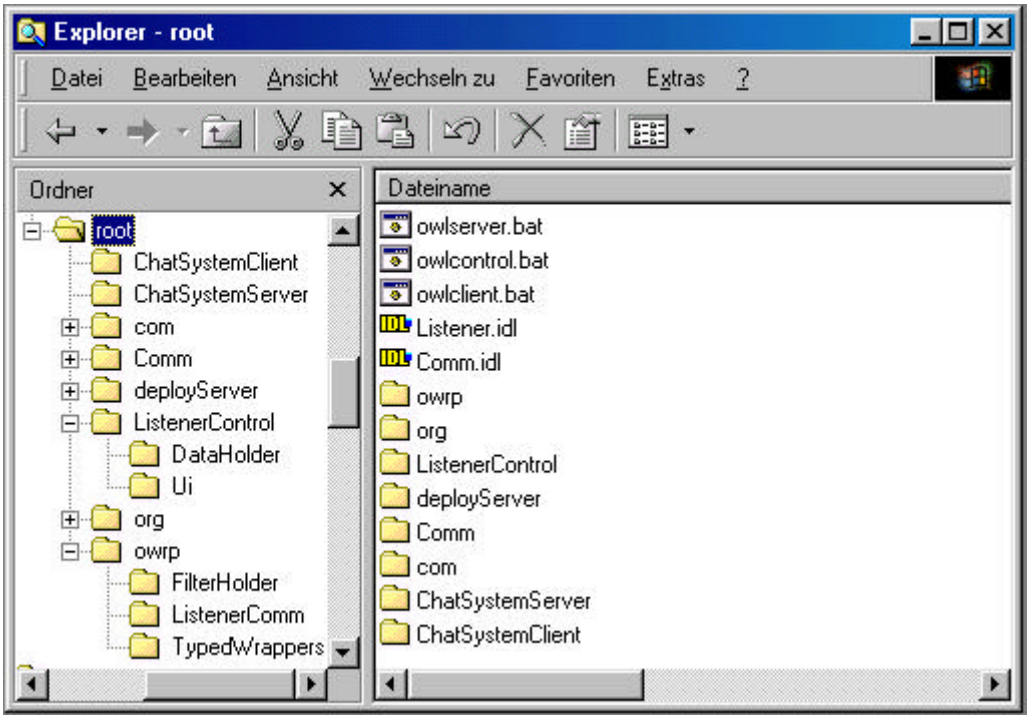


figure 6.1.2-1 packages of the testing framework and the system under test

6.1.3 Start-up

The tree testing phases, pre-test-phase, testing-phase and post-test-phase, are outlined in the following paragraphs. But before starting with the explanation of the pre-test-phase, I describe shortly how the testing framework together with the example application must be installed.

Before the testing framework can be used, it must be ensured that the VisiBroker CORBA middleware is already installed, that JDK 1.2.2 is available on the system and that the environment variable `PATH` is are appropriately set.

```
SET PATH= c:\programme\jdk1.2.2\bin;
          C:\PROGRAMME\INPRISE\VBROKER\BIN;
```

The following two CORBA services must be started on at least one machine connected to the network:

1. VisiBroker Smart Agent
2. VisiBroker Naming Service

It is important to start the Smart Agent first.

In the directory “ChatSystem” there are two batch files that can be used to start the components of the system under test. The file `owlserver.bat` helps to start a chat server and every time `owlclient.bat` is used a new chat client is created.

Both batch files need an additional parameter that is necessary to identify the component interfaces during the testing process. If the following command lines below are entered, a chat server and two chat clients are created.

1. command line: `owlserver sa`
   to start the chat server with the parameter sa

2. command line: `owlclient ca`
   to start the first chat client with the parameter ca
3. command line: `owlclient cb`
   to start the second chat client with the parameter cb

The batch file `owlclient.bat` contains the following string:

```
vbj  -DORBservices=owrp.Init -DSVCnameroot=NameService
     -DServiceName=%1 ChatSystemClient.ClientMain
```

The command `vbj` is used to start the Java class `ClientMain` that is located inside the package `ChatSystemClient`. In addition the batch file provides various parameters:

- name of the service loader of the testing framework
  `-DORBservices = owrp.Init`
  The service loader is called `Init` and is located inside the `owrp` package.
- name of the used naming service
  `-DSVCnameroot = NameService`
  This parameter is necessary because the system under test and the testing framework are using the same naming service and the local listener of the testing framework relay on the ORB instance created by the components of the system under test.
- name of the local listener
  `-DServiceName=%1`
  This parameter is used to identify the CORBA interface of the local listener objects. `%1` signifies that the parameter supplied when calling the batch file is used.

The batch file `owlserver.bat` contains a very similar command line. It is important that every component of the system under test is started with a different `ServiceName`, because every local listener must have a different name.

The figure below summarises the necessary steps to start the system under test:

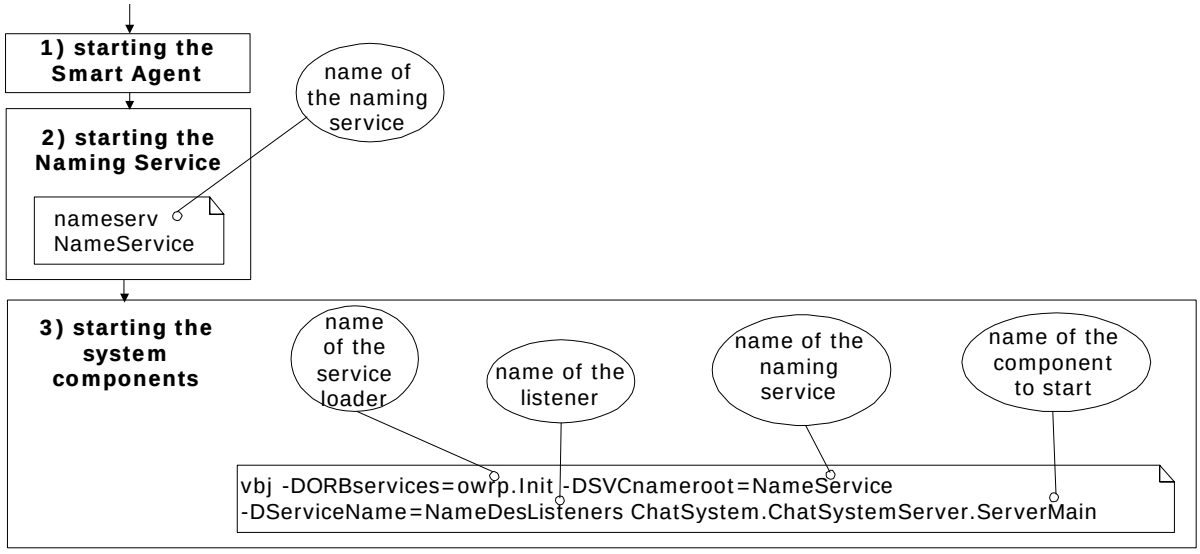


figure 6.1.3-1 steps to start the system under test

The graphical user interface of the testing framework itself can be started through the batch file `owlcontrol.bat` that contains the command line: `vbj ListenerControl.LCMain`

6.2 Pre-test-phase

During the pre-test-phase the tester prepares the system under test by generating or writing the extra code needed for testing purposes. Then the tester sets up a test project, defined filters for the logging data and selects faults to be injected.

6.2.1 Instrumentation

The so-called object wrappers described in chapter 5.2 are used to instrument the CORBA interfaces for testing purposes. When executing the `idl2java` compiler with the option `-obj_wrapper` the compiler automatically generates object wrapper base classes for each CORBA interface and additional helper classes with methods for adding or removing object wrappers.

In the current implementation of the testing framework the instrumentation of the wrappers has to be done manually but in the future the testing framework will be extended by a parser and instrumenter to automate this task.

The source code listing in chapter 5.4.4 shows the instrumentation of the `ServerReception` interface.

6.2.2 Projects

The following steps are related to the handling of test projects:

Create a new project

To create a new test project the tester has to enter the name of the test configuration, the name of the system under test and the version of the system under test .

A distributed system consists of several distributed components with at least one CORBA interface. When creating a new project to test such a distributed system, the testing framework has to gathers details of the CORBA interfaces. Important information items about an interface are:

- the defined methods
- the names and types of parameters
- the types of the return values
- the exceptions that may be raised by the methods

To create a new project the tester has to execute the following steps:

1. click on “Project” followed by “New”
2. enter the name of the test configuration, the name of the system under test and the version of the system under test

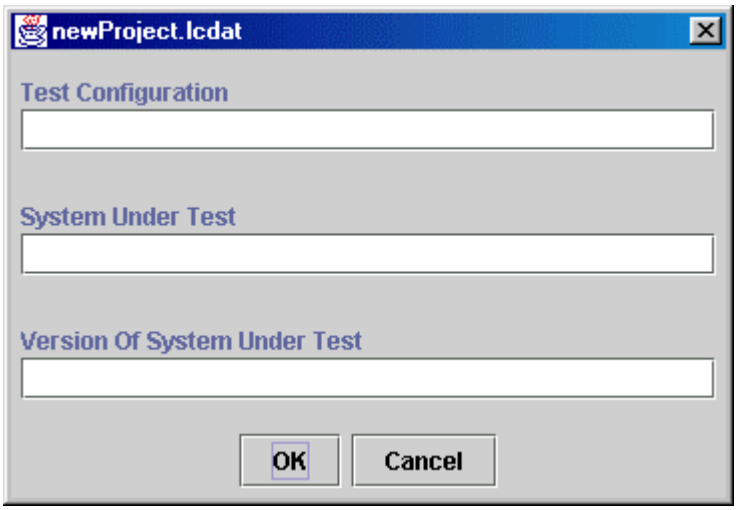


figure 6.2.2-1 first window during the creation of a new project

3. click on “OK” and a second window appears
4. enter the name of the testing person and the environmental settings

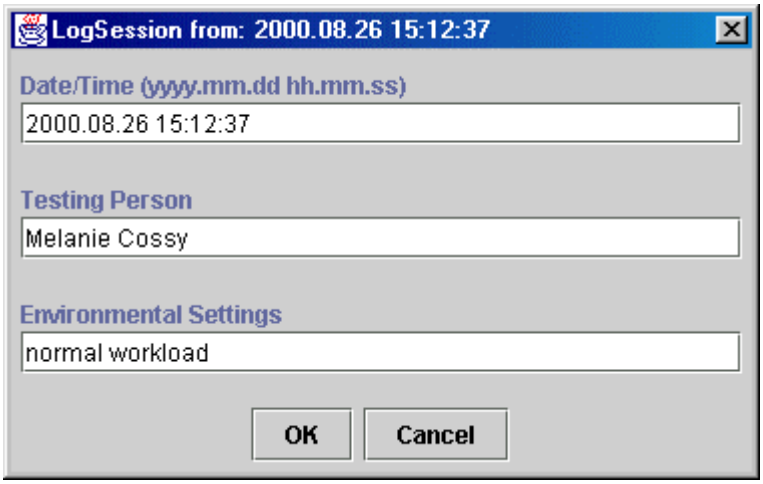


figure 6.2.2-2 second window during the creation of a new project

5. click on “OK”

Save a project

The project configuration can be saved in a file to re-use the project settings later during the execution of other tests. To save a project the tester has to:

1. click on “Project” followed by “Save As”
2. type a file name without extension

Open a project

A saved project configuration file can be opened later. To do this the tester has to:

1. click on “Project” followed by “Open”
2. type in or browse for a project configuration file

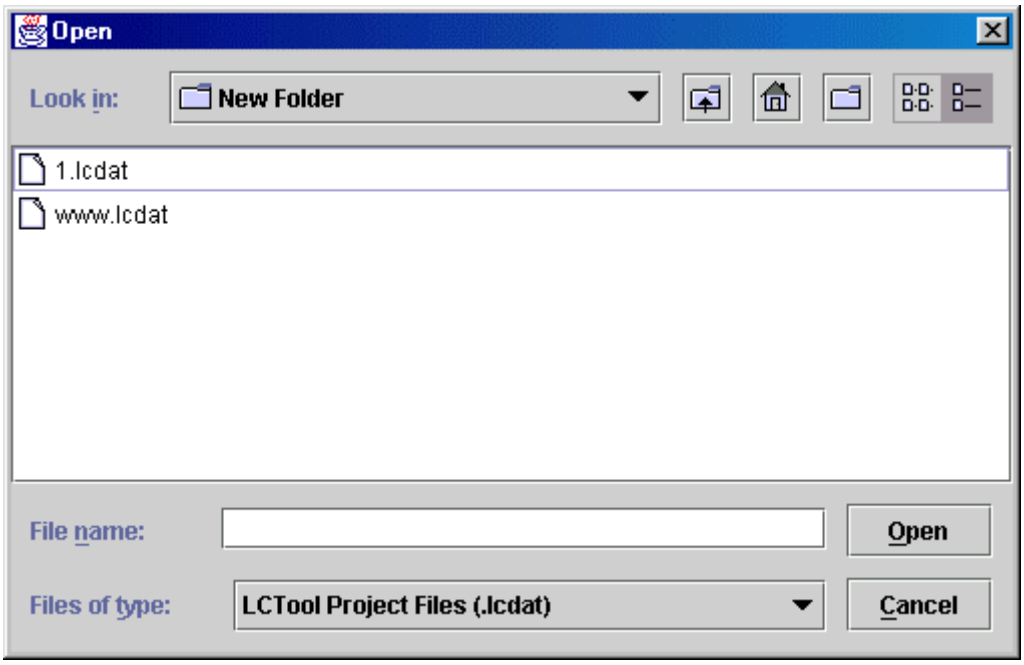


figure 6.2.2-3 dialog window to open a project

6.2.3 Hierarchy of the system

The tester can view the system under test in a hierarchical manner. After clicking on “Filters” followed by “Properties” a window appears that shows the different components, the CORBA interfaces and all defined interface methods.

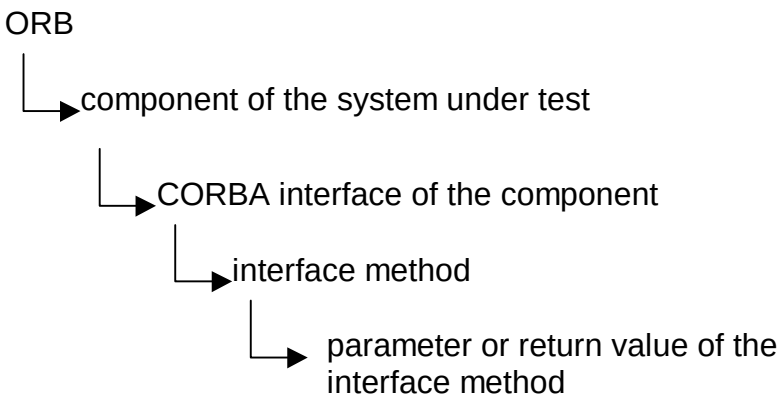


figure 6.2.3-1 hierarchical organisation of the system under test

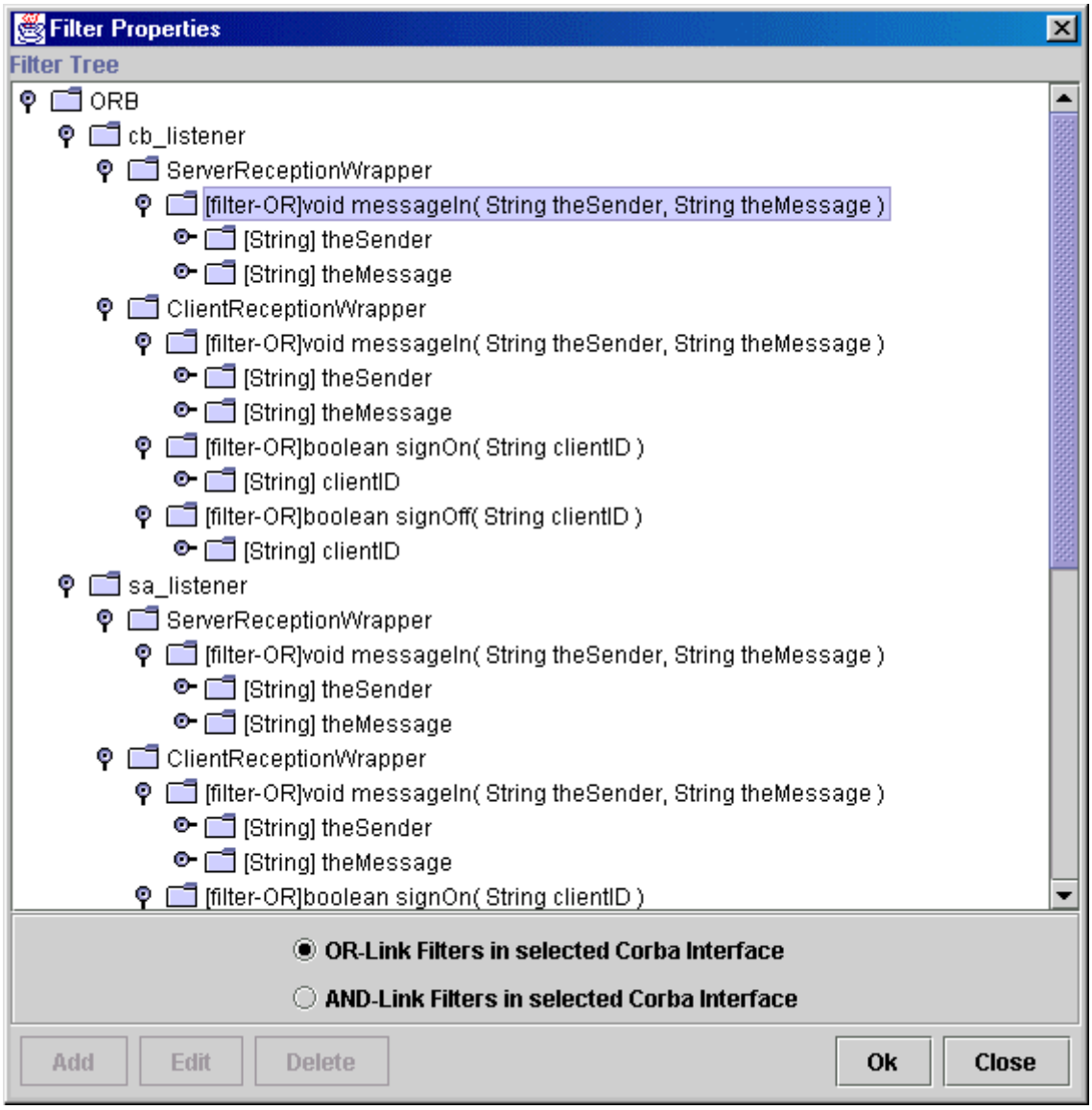


figure 6.2.3-2 hierarchical system view

6.2.4 Filters

Filters are necessary to reduce the amount of gathered data to what is necessary. The tester can place filters to determine specifically what information he wants to trace during the test execution. In the hierarchical system view the tester can set and modify filters that specify parameter ranges and exceptions to be observed. Filters are used to determine whether or not a method is ever executed with specific parameters or whether or not a defined exception is throw. The actual filter configuration can be saved and retrieved later.

Set filter properties

The tester defines the filters based on what he wants to test. The figure below shows the dialog window that helps the tester to filter a parameter of type string. In the title bar of the dialog window the name `theSender` and the type `String` of the chosen parameter is shown. The manually entered string pattern `M*` signifies that only those method invocations will get logged where the name of the sender starts with an `M`.

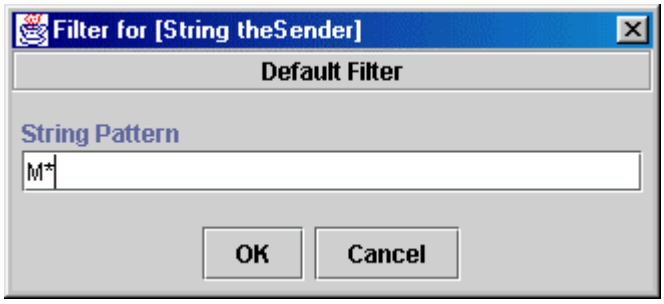


figure 6.2.4-1 dialog window to define the filter of a string

The following steps must be executed to define the filter properties of a parameter:

1. click on “Filters” followed by “Properties” to see a hierarchical view of the system under test
2. highlight the chosen parameter by clicking on it
3. click on the button “Add”
4. a dialog window appears to define the filter properties of the highlighted parameter
5. define specific filter properties or click on the button “Default Filter” to chose a filter that matches all possible parameter values
6. click “OK” and the defined filter property appears in the hierarchical view below its parameter
7. if necessary repeat the steps 2 to 6 for the other parameters
8. click on the button “OK”

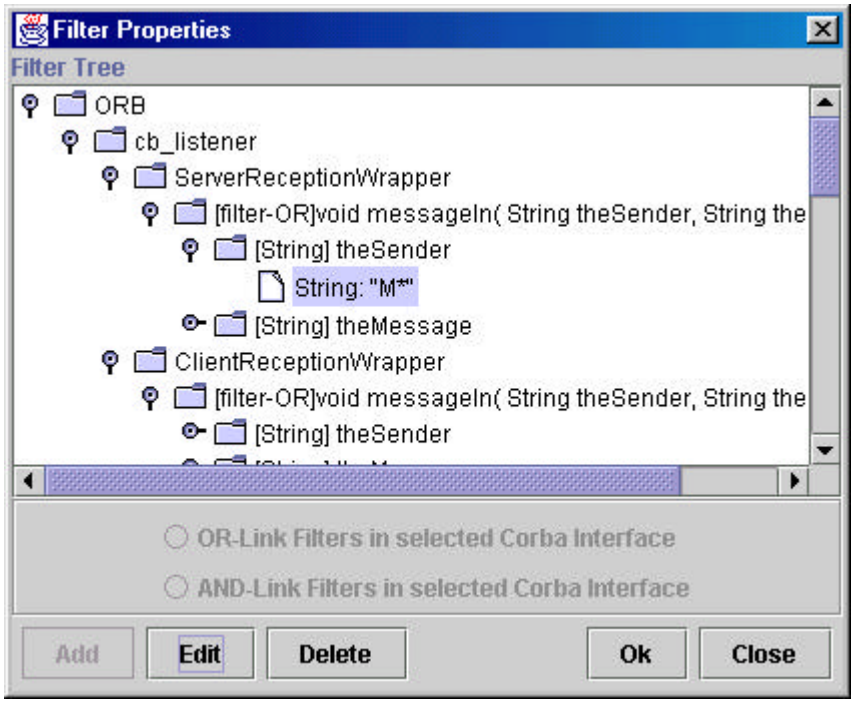


figure 6.2.4-2 hierarchical view with the defined filter

6.2.5 Selecting the faults to be injected

This feature is not implemented in the actual version of the testing framework, yet. But fault selection will work very similar to the definition of filter properties. The testing framework will present a list of faults and indicate possible locations for injection, such that the tester will be able to introduce faults into the system and observe the fault handling capability of the system.

The locations for injection will be the interface methods between components and the planned faults for injection are:

- Mutation
  The tester can mutate parameters of interface methods
  - by swapping two parameters
  - by incrementing or decrementing return value and parameters.
- Delays
  The execution of a method invocation can be delayed by inserting waiting times.
- Throw exceptions
  The tester can force components to throw exceptions ignoring what happens during the real execution of the method invocation.

6.3 Testing-phase

After the pre-test-phase comes the testing-phase. During the testing-phase the tester monitors the system execution to determine coverage details and system characteristics.

A test run is initiated by starting the distributed listeners of the testing framework. These listeners reside on the same machine as the distributed components of the system under test and therefore are called local listeners. The wrappers of the CORBA interfaces send the monitored data to their local listener, whenever an interface method is invoked. The global test control requests the monitored data from the local listeners at fixed time intervals.

6.3.1 Log session

A test project has at least one log sessions. Every time a new project is created the tester has to enter the name of the testing person and the environmental settings for the first log session (figure 6.2.2-2).

New log session

During the testing process new log sessions can be added to the current test project by executing the following steps:

1. click on “Log Session” followed by “New” and “Yes”
2. enter the name of the testing person and the environmental settings of the new log session
3. click “OK”

Opening an earlier session

Executing the following steps can reopen previously saved log sessions:

1. click on “Log Session” followed by “Choose Actual”

2. a dialog window appears, which shows all log sessions belonging to the current test project

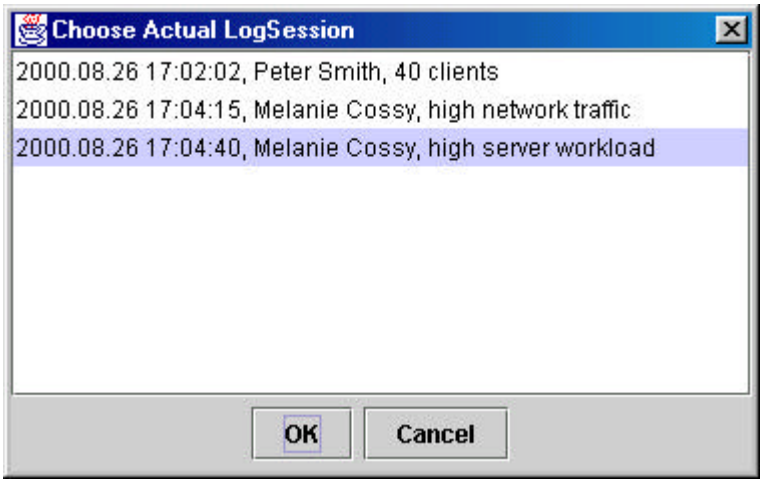


figure 6.3.1-1 dialog window with all available log sessions

3. select one of the listed log sessions
4. click “OK”

6.3.2 Viewing the traced data

Every time an interface method is invoked or an exception is thrown, the wrapper of the appropriate interface is sending a log string its local listeners. The local listener receives the log string and checks if the method invocation matches the filter defined for that method. If yes, a message is displayed in the main window of the testing framework.

After starting the system under test and testing framework the filters are configured such that **no** method invocation get displayed in the main window. The tester selects the specific methods he wants to trace by defining the appropriate filter properties explicitly.

Figure 6.3.2-1 shows the main window of the testing framework displaying four invocations of the method `messageIn`. Every method invocation is represented by two lines.

The following steps have been executed to get this screenshot:

1. click on the menu item “Filters” followed by “Properties”
2. a window showing the hierarchical system view appears
3. search for “cb_listener”, the listener of the component “cb”[4]
4. below “cb_listener” there is the interface “ServerReceptionWrapper” with the method “messageIn” and the parameter ”theMessage”
5. highlight the parameter ”theMessage”
6. click on the button “Add” and a dialog window appears
7. click on the button “Default Filter”

[4] “cb” is the parameter, which must be defined when starting the system under test (6.1.3 Start-up)

8. click on the button “OK” and return to the hierarchical system view
9. search for “ca_listener”, the listener of the component “ca”
10. below “ca_listener” there is the interface “ServerReceptionWrapper” with the method “messageIn” and the parameter ”theMessage”
11. repeat the steps 5 to 8
12. click on “OK” and close the window showing the hierarchical system view
13. in the main window of the testing framework click on the menu item “Listen” followed by “Start”
14. now all messages sent between the two chat clients “ca” and “cb” get logged and displayed in the main window of the testing framework

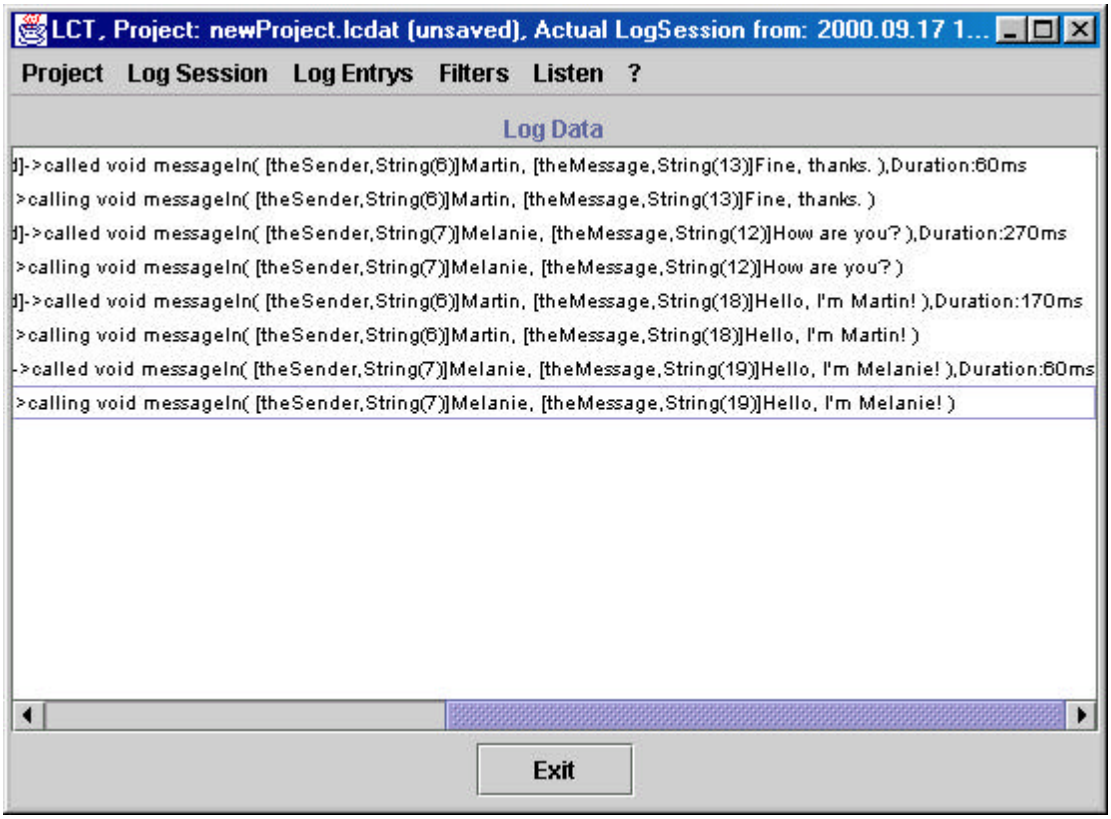


figure 6.3.2-1 monitored data

In the current version of the testing framework there are two ordering options for the monitored method invocations, called log entries, available. Log entries can be sorted by time or by wrapper.

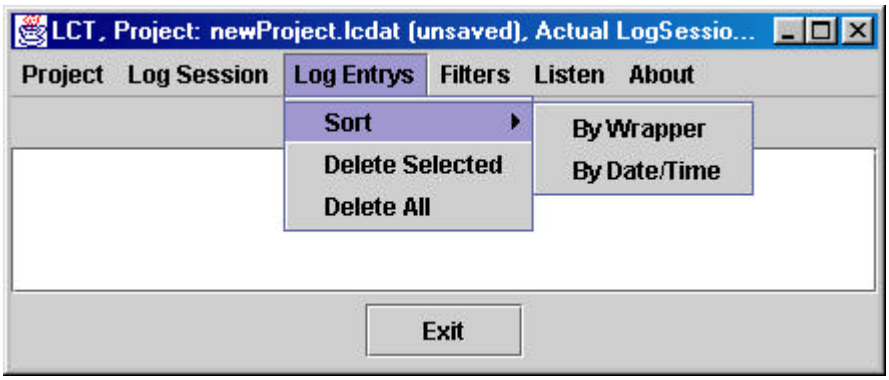


figure 6.3.2-2 available menu options for viewing the data

Other forms to visualise the monitored data - for example tables, graphs and charts - will be implemented in the next version of the testing framework.

6.4 Post-test-phase

The post-test-phase deals with the analysis of all the data gathered and saved. It must be possible to analyse the method, exception and fault coverage easily by presenting the data in different coverage related views. Therefore it will be necessary that the next version of the testing framework uses a database to store the monitored data, such that the data in the database can be analysed with the help of database queries.

It must be possible to display the gathered data in the following forms:

- a bar chart that displays the method coverage[5] of the whole system
- bar charts that display the method coverage[5] component-wise
- bar charts that display the method coverage[5] of the interfaces
- a list of all methods that have been covered and a list of all method that need to be covered[6]
- a pie chart that shows profiling information in terms of time spent in the different components
- a pie chart that shows profiling information in terms of time spent during the execution of the different interface methods
- a graph that shows how many times the different interface methods have been executed
- a graph that shows how many times exceptions have been raised

---

[5] when determining coverage values it is important to consider which kind of filters have been used during the test run

[6] helpful for test set enhancement

7 Conclusion

During my dissertation I have developed a testing framework that can be used to monitor, meter and control distributed system during the testing process. The testing framework itself can only be used to test CORBA-based systems written in Java, but the adopted testing methodology can be applied to test also other kinds of distributed systems that have components with well defined interfaces, such as DCOM or Java RMI.

7.1 The testing framework

The proposed testing framework is non-obtrusive as the implementation of the system under test remains untouched. The extra code needed for testing purposes is added without having to modify the system under test nor the interface definitions. The testing framework uses so called wrapper objects, which can be generated easily from the component interface descriptions with the help of the CORBA IDL compiler.

The framework provides a mechanism by which the tester can define exactly which method invocations should be logged. Though filters the tester can specify data values, number ranges or string patterns for the parameters of all interface methods. In addition AND as well as OR conditions are available to allow fine-grain control over the testing process.

7.2 The testing methodology

The main activities of the adopted testing methodology are outlined below:

1. Identify the CORBA interfaces of the components.
   a) Identify the methods and exceptions that are present in the interfaces.
   b) Identify the parameters in the methods.
2. Create a test set based on the requirements of the component and trace the system execution during the testing process.
3. Remove any error revealed in the process of testing.
4. Improve the test set by adding new or modifies test cases to increase all interface-based coverage measures.

Special activities are needed to test a distributed application for fault tolerance:

1. Identify generic faults and inject them into the system under test.
2. Create a test set based on the requirements for fault tolerance and execute the system under test against the test set.
   a) Check if a fault has been triggered.
   b) Trace the system execution after a fault has been triggered.
   c) Observe the behaviour for fault tolerance
3. If the behaviour does not conform to the requirements for fault tolerance, make appropriate changes to the fault-recovery code
4. Improve the test set to increase the number of triggered faults.

7.3 Known problems

The testing framework that has been presented in this dissertation is a prototype and it has some inadequacies that are described below:

- no database
  There is no database connected to the testing framework such that the test data gathered during the various log sessions of the different projects can be stored. The monitored test data is stored in text files. The project and session configurations are saved in a Java `serializable` file.
- no fault injection can be performed
  Although the logging wrappers could easily be implemented such that they allow fault injection, the GUI of the testing control tool does not support this.
- insufficient views
  There is no graphical visualisation to present the gathered test data.
- manual instrumentation
  The IDL files of the system under test are compiled with a special compiler option to generate wrapper base classes. The tester uses these wrapper base classes to implement manually the specialised logging wrappers needed by the testing framework.
- no tool tips
  The GUI of the test control tool has no help features.
- no filters for complex parameters
  Only for primitive parameter types the tester can define special filters. For complex types like classes, there is only a filter, which differentiates between "null" and "not null", available.
- accuracy of time measurement
  Sometimes it takes only a few milliseconds to execute an interface method. To determine such al small execution time accurately the available clock is not fine granular enough.

7.4 Future Work

The next version of the testing framework should address the identified problems of the current testing framework. In addition it should give the tester more control over the system under test. The tester should be able to invoke interface methods directly and to pause or resume server components.

In the next version of the testing framework it should be possible to monitor the system behaviour exactly by observing distributed state transitions with or without the presence of injected faults. The tester should be able to specify and recognise global events that are conjunctions of local events occurring in distributed components.

The costs of test execution will be another main issue in the future and possibilities to automate the test execution must be evaluated.

Further research must be done in the field of requirement coverage. The question is how system requirements can be included into the test adequacy criteria. Special facilities are needed to specify requirements such that it is possible to measure the requirement coverage.

Bibliography

[1] S.Ghosh and A.P.Mathur

On Sources of Errors and Failures in Distributed Systems

In *Proc. International Conference on Software Engineering, Hyderabad, India*, pages 348-355, December 20-22 1997

[2] M.C.Hsueh, T.K.Tsai and R.K.Iyer

Fault Injection Techniques and Tools

IEEE Computer, pages 75-82, April 1997

[3] E.Gamma, R.Helm, R.Johnson, J.Vlissides

Design Patterns: Elements of Reusable Object-Oriented Software

Addison-Wesley, 1994

[4] John D.McGregor

Parallel Architecture for Component Testing

Journal of Object-Oriented Programming, May 1997

[5] R.Carver and K.C.Tai

Use of Sequencing Constraints for Specification-Based Testing of Concurrent Programs

IEEE Transaction of Software Engineering, June 1998

[6] Mark Fewster and Dorothy Graham

Software Test Automation

Addison-Wesley; 1999

[7] R. Chillarege and N. Bowen

Understanding Large System Failures --- a Fault Injection Experiment

19th International Symposium Fault-Tolerant Computing, 1989

[8] Mary Shaw and David Garlan

Software Architecture: Perspectives on an Emerging Discipline

Prentice Hall, 1996

[9] Inprise Corporation
VisiBroker for Java - Programmer’s Guide
1996