# FAT:
# A Framework for Automated Regression Testing of Protocol Stacks

**MASTER OF ENGINEERING THESIS**

**Karl Magnus Nilsen**

December 15th, 2003

**UNIVERSITY OF TROMSØ**
**DEPARTMENT OF COMPUTER SCIENCE**

# Abstract

Software systems today are becoming larger and more complex, resulting in a growing need for good and efficient testing routines. An approach used by several software developers is to automate the test process. Test automation has the benefits that it reduces the time of the testing process and that automated tests are more accurate and precise than manual tests.

Manufacturers who wish to develop products using the Bluetooth technology, the Bluetooth logo and trademark has to go through a qualification program. This program is expensive, thus the manufacturer has incentives to make sure that the product is well tested before sending it to qualification. A Bluetooth stack is an example of a product that must be qualified. An automated tool for testing of Bluetooth stacks is therefore desired.

FAT is a framework that provides functionality to write and execute tests on a Bluetooth stack. The framework makes use of the ability to stitch generic test layers in-between the layers of the stack. These test layers can operate on messages passing through the stack. Our test layers provide an API to insert, modify, copy and delete messages. FAT introduces a test system client (TSC) where a tester can write tests and choose tests for execution. The tests are written in Java, where each test is a single method. The tester uses the test layer's API to interface with the stack. The communication mechanism between the TSC and the test layer is XML-RPC. The TSC may therefore be executed on a different node than the stack itself.

This thesis motivates FAT, and describes how the framework is designed and implemented.

# Preface

This thesis presents the final work of my education at the University of Tromsø. The thesis is a joint project between the University and ObexCode AS.

ObexCode AS is a worldwide leading vendor of short-range connectivity products and solutions. ObexCode is an enabling company, which means that they deliver key components to other companies in the ad-hoc wireless business. Such components will include IrDA and Bluetooth stacks, object exchange and synchronization layers. The name ObexCode can be read as “development of systems and application for technologies that have OBEX in common.” OBEX refers to the standard IrOBEX, which is currently used by IrDA, Bluetooth, SyncML, WAP and 3G. The headquarters is located in Tromsø. The company also has offices in Oslo and Shanghai.

ObexCode is currently involved in several different development projects. The company has during these, and previous projects, discovered the need for standardized test processes, which can be automated. One of the current projects is the development of a Bluetooth stack. The framework described in this thesis is meant to be used for regression testing during the development of the Bluetooth stack.

The ideas and main principles of the framework were already adopted by ObexCode before the whole framework was completed.

# Acknowledgements

First of all, I would like to thank my supervisors, Dag Brattli, Åge Kvalnes and Haakon Bryhni for supervising and guiding me on this thesis. Special thanks to Dag for helpful hints and suggestions of the design and implementation of the system and interesting discussions about the subject of this thesis. Special thanks also to Åge for helpful comments and guidelines during the writing process.

Special thanks also go to Frank Ronny Larsen. Without his help, the implementation process probably would have taken considerably longer time. I don't think there is anything this man does not know about programming! He has also reviewed my thesis and provided me with useful comments.

I will also thank my co-students Rune Devik and André J. Henriksen for interesting discussions about different subjects related to the thesis. These discussions have made several things clearer and I feel this have improved my thesis. Rune has also reviewed parts of my thesis and contributed with useful comments. They have both also contributed to a nice and attractive working environment. It has made the days working with the thesis a real pleasure.

Thanks to Jan Fuglesteg for providing us with office supplies and coffee. The importance of the coffee shall not be underestimated!

I will also thank my whole family for supporting me during my studies in Tromsø and especially during the work with the thesis. It has been very important to me.

Tromsø, December 15th, 2003.

Karl Magnus Nilsen

# Table of Contents

# Table of Figures

# Chapter 1

# Introduction

## 1.1 Background

Bluetooth [Bluetooth SIG, 1999] is a low-power, short-range wireless technology that provides links between mobile computers, mobile phones and other portable handheld devices. Bluetooth was originally developed for replacing cables when connecting devices like mobile phones, headsets and computers. The specification of the Bluetooth technology is developed, published and promoted by the Bluetooth Special Interest Group (SIG).

Manufacturers who wish to develop products using the Bluetooth technology, the Bluetooth logo and trademark has to go through a qualification program. The qualification process tries to verify if the product conforms to the Bluetooth specification. The specification is detailed and the qualification process is therefore a time consuming and expensive task. The manufacturer does not wish to go through the qualification process more than once. Hence the quality of the product should be as good as possible before sending the product to qualification.

The development of software is a process with many steps from the beginning to the final deployment of the system. Testing is one of the final steps towards a complete system. The purpose of the testing phase is to verify that the product lives up to its requirements, and is therefore a crucial part of the process. Because of the importance of testing, software developers should have incentives to develop good test routines.

Software systems have a tendency to become larger and more and more complex. As the systems grow the test process takes more time. In the software business, as in most industries, time is money. One of the most effective ways to save time is to automate a process. This is a known fact in many industries. Instead of letting people do a job manually, machinery, computers or robots can do the job for us or aid us with the job, and in most cases they will do the job not only faster but better than us. If the test process can be automated it could save considerable time for the testers.

If a test is to be carried out it has to be specified in some way. This may not be a straightforward task. It would help the test engineers if a template exists that tells how a test should be specified. This will also standardize the test process in the company or development group, and will eventually probably save time when the test engineers get used to the template and the standard process. In addition to this, a standard test specification will make it a lot easier to automate the test process.

Another fact is that many test systems today offer one or just a few types of tests within the same system. This means that if a test engineer wants to perform different types of tests (e.g. conformance and performance) on the same system under test, he or she must use different test systems, which probably have different ways of specifying tests. It is not a desirable situation to use many different systems when you might as well could use a single system. The difference in the specifications might also lead to confusion. A test system that can handle many types of tests will ease the test process.

When a system has been made and is ready for delivery, the software developer has to convince its customer that the system works as it should. The customer has to trust the developer to test the system properly. Often the customer does not have the resources or knowledge to test the systems themselves to verify that it works correctly. But if the test process is automated and simplified in such a way that the customers easily can run some tests on the new system, the customer has a kind of guarantee on the system. The customer can verify that the system works properly and buy the product without being insecure of the quality of the system.

A common approach for test automation is to design a framework that offers functionality to write and execute tests. This thesis describes FAT, a framework for automated regression testing of protocol stacks.

## 1.2 Problem Definition

The goal of this project is to design and implement a framework for automated regression testing of protocol stacks developed within the ObexCode network protocol development framework (NPDF).

We divide our main goal into three sub-goals that must be addressed and examined:

- The proposed test environment shall be implemented in a prototype which will interface with the stack under test through a defined API, preferably possible to execute on a different processing node than the stack itself.
- Computation and memory footprint of the testing framework on the stack under test should be minimal, because stacks run on devices with little memory and poor computation capabilities.
- The framework should allow for the testing of Bluetooth stacks developed within NPDF.

## 1.3 Method and Approach

Computer Science can be separated into three major paradigms that provide a context for the definition of the discipline. The ACM Task Force [Denning et. al., 1989] has given the following description of the three paradigms:

The first paradigm is theory. It is rooted in the mathematical sciences. The process is to define the objects of study, hypothesize possible relationships among them, determine whether the relationships are true, and to interpret the results. Mathematicians will say that science advances only on a foundation of sound mathematics.

The second paradigm is abstraction. It is rooted in the experimental scientific method and is the bedrock of natural sciences. Scientists say that scientific progress is achieved primarily by formulating hypotheses, and systematically constructing models and design experiments, to verify and validate the hypotheses.

The third paradigm is design. It is rooted in engineering and consists of a process to construct a system to solve a given problem. Engineers say that progress is achieved primarily of posing problems and systematically following the design process to construct systems that solve them. The design process consists of stating requirements, stating specifications, designing and implementing the system before testing the system.

The design paradigm will be used in this thesis. The reason is that the main task of the thesis is to construct an actual system that can perform automatic regression testing of protocol stacks. The design approach is also a natural choice since it is used by the company that has defined the thesis.

## 1.4 Limitations

The Bluetooth stack that the framework shall evaluate is currently in the process of being implemented. This implies that it will not be possible for the framework to evaluate a full stack. The parts that are implemented are however enough to verify the functionality of FAT. But the lack of a full stack will limit the possible experiments that may be done to evaluate the performance of the framework.

## 1.5 Outline of the Thesis

The thesis is organized as follows:

- Chapter 2: Gives an overview of the theory that works as background material for the thesis, including a description of related work.
- Chapter 3: Gives a short introduction to the architecture, covering the most central parts of the framework.
- Chapter 4: Describes the design and implementation of FAT.
- Chapter 5: Presents the experiments and results that are performed on the system.
- Chapter 6: Summarizes, discusses and concludes the work presented in the thesis.

# Chapter 2

# Background and Related Work

There exists a large number of testing methodologies and frameworks. In this chapter we present a few selected methodologies and frameworks that we consider prominent and relevant to our work. In our presentation we focus on the main aspects of each approach, and examine in particular regression testing, conformance testing and test automation. We also examine existing frameworks, which are similar to our framework.

## *2.1 Bluetooth*

This section explains important concepts of the Bluetooth technology including a brief description of the Bluetooth protocol stack. Furthermore, it describes the Bluetooth qualification process and the Bluetooth test specifications.

### 2.1.1 Bluetooth – An Introduction

Bluetooth is a low-power, short-range wireless technology that provides links between mobile computers, mobile phones and other portable handheld devices. Bluetooth was originally developed for replacing cables when connecting devices like mobile phones, headsets and computers. Bluetooth has since evolved into a wireless standard for connecting electronic devices to form personal area networks (PANs) as well as ad hoc networks [Dideles, 2003].

Bluetooth operates on the unlicensed Industrial Scientific Medical (ISM) band at 2.4 GHz, which ensures worldwide communication compatibility. However, since the ISM band is open, several unpredictable sources of interference must be dealt with. To minimize the risk of such interference, Bluetooth uses a Frequency Hopping Spread Spectrum (FHSS) technology. Using FHSS, Bluetooth devices multiplex the sending of packets over multiple frequencies.

The link bandwidth offered by Bluetooth is 1 Mbps, but with overhead, and due to asynchronous channels, the maximum link bandwidth in one direction is 721 kbps, while 57.6 kbps in the opposite direction. The alternative is a 432.6 kbps symmetric link. The typical communication range for Bluetooth is 10m, but up to 100m is possible depending on the power class of the device [Dideles, 2003].

The Bluetooth technology was conceived at Telefonaktiebolaget LM Ericsson in Sweden in 1994. At this time they started a project to study the feasibility of a low-power and low-cost radio interface to eliminate cables between mobile phones and their accessories. The inventors understood that the technology was more likely to be widely accepted and thus more powerful if it was adopted and refined by an industry group that could make an open specification. The Bluetooth Special Interest Group (SIG) was therefore formed in 1998. The founding companies of the SIG are Ericsson, Intel, IBM, Nokia and Toshiba. Later other companies have joined the SIG [Miller and Bisdikian, 2001].

The Bluetooth technology is named after the Danish king Harald Blåtand. During his reign he tried to unite Denmark and Norway. For a technology with its origin in Scandinavia, and with the purpose of unify multinational companies, it seemed appropriate to name it after a king who united countries. Blåtand translates loosely to "Blue Tooth" [Miller and Bisdikian, 2001].

### 2.1.2 The Bluetooth protocol stack

The Bluetooth protocols define procedures for connections and data exchange between Bluetooth devices.

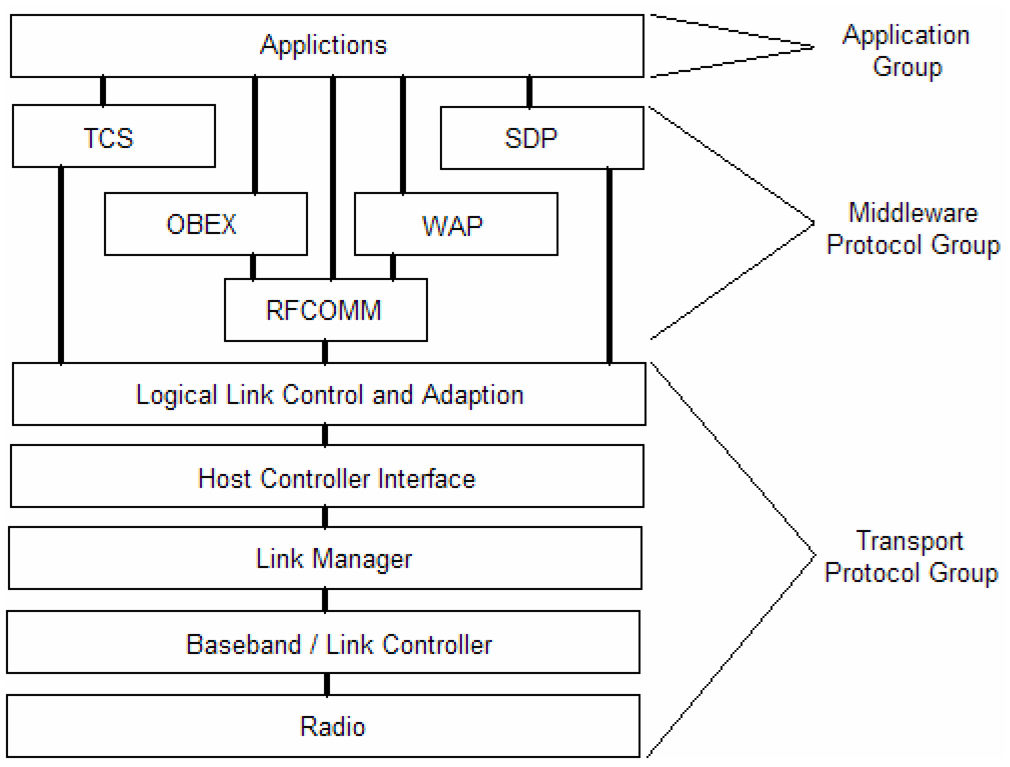

Applictions
Application Group
TCS
SDP
OBEX
WAP
RFCOMM
Middleware Protocol Group
Logical Link Control and Adaption
Host Controller Interface
Link Manager
Baseband / Link Controller
Radio
Transport Protocol Group

**Figure 1 - The Bluetooth Protocol Stack**

The elements of the stack are logically partitioned into three groups:

- The transport protocol group
- The middleware protocol group
- The application group

The transport protocol group contains the protocols that enable Bluetooth devices to locate each other, and that are responsible for the creation, configuration and management of physical and logical links. The protocols in this group are the radio, the baseband/link controller, the link manager, the logical link and adaptation and the host controller interface.

The middleware protocol group contains additional transport protocols needed to enable existing and new applications to operate over Bluetooth links. The group contains both third-party and industrial standard protocols, as well as protocols developed by the SIG specifically for Bluetooth wireless communication. The former group includes internet-related protocols (TCP, IP, PPP), WAP and OBEX, which is adopted from IrDA. The latter group contains RFCOMM, TCS and SDP.

The application group consists of the applications that make use of Bluetooth links. These applications could either be unaware of Bluetooth transports, such as a web browsing client, or be are aware of Bluetooth wireless communication, such as applications that use the telephony control protocol for controlling telephony equipment.

In the remainder of the section we present a brief description of each of the protocols and layers in the Bluetooth stack.

*The Radio Layer*

The Bluetooth radio layer is designed to make it optimal for use with the Bluetooth protocol stack. The radio part of the specification contains mostly design specifications for Bluetooth transceivers. The transceiver design is motivated by the requirement to allow development of high-quality, low-cost transceivers that comply with the various 2.4 GHz ISM band regulations around the world. Different regulations in different countries imply that the Bluetooth radio can operate over 79 or 23 channels, each one of which is 1 MHz wide.

*The Baseband/Link Controller Layer*

The Baseband Layer (BL) determines and instantiates the Bluetooth air-interface. It defines how devices search for other devices, and how they connect to them. In particular, BL defines the master and slave roles for devices: the device that initiates a connection becomes the master of the link and the other becomes slave. The layer also defines rules for sharing of the air-interface, so that several devices can use the technology simultaneously. It defines how the frequency-hopping sequences used by communicating devices are formed. It also defines various packet types supported for synchronous and asynchronous traffic and packet processing procedures such as encryption, error detection and correction, packet transmission and retransmissions.

*The Link Manager Layer*

The Link Manager Protocol (LMP) is used to negotiate the properties of the Bluetooth air-interface between devices. This negotiation includes authentication where the communicating devices uses a challenge-response approach. If authentication fails, the link managers may sever the link between the devices and thus denying any communication between them. The Link Manager also negotiates bandwidth allocation to support a desired grade of service for data traffic and periodic bandwidth reservation to support audio traffic. Finally it supports power control by negotiating low activity Baseband modes of operation.

*Host Controller Interface Layer*

The Host Controller Interface (HCI) has been developed to ensure interoperability between different host devices and Bluetooth modules. A host device is a device that is enabled with Bluetooth communication, and contains the upper layers of the stack (from L2CAP and upwards). A Bluetooth module is a package consisting of the lower layers, Radio, Baseband and Link Manager. Both the host devices and the Bluetooth modules may come from different vendors. To provide interoperability between different devices and modules, the HCI layer provides a common interface for accessing the lower layers of the stack regardless of the physical interface that connects the host to the module. The HCI layer is not a required part of the specification. For tightly integrated embedded systems the HCI layer may not be required.

*The Logical Link Control and Adaptation Layer*

The Logical Link Control and Adaptation Protocol (L2CAP) layer shields higher layer protocols and applications from the details of the lower-layer protocols. L2CAP supports protocol multiplexing, in order to support sharing of the air-interface between different protocols and applications. The L2CAP layer also supports segmentation/reassembly of large packets used by higher layers into smaller packets for the lower layers. Finally, it also negotiates a level of service between two devices. The regulation of service is done by exercising admission control for incoming traffic, and coordination with lower layers to maintain the desired level of service.

*The RFCOMM Layer*

The serial port is a common communication interface used by communicating devices today. To ease the integration with legacy software, the RFCOMM layer implements a serial port abstraction. An application can use RFCOMM very much like a standard wired serial port to accomplish scenarios such as synchronization, dial-up networking and others without significant changes to the application.

*The SDP Layer*

The Service Discovery Protocol (SDP) is the protocol that enables Bluetooth devices to discover and learn about the services offered by other devices. It also defines a way for devices to describe the services that they provide to other devices. This protocol is motivated by the fact that ad-hoc networks, like a network of Bluetooth devices, do not have a static configuration of services like traditional networks. A dynamic discovery protocol is therefore required.

*IrDA Interoperability Protocol Layers*

The Infrared Data Association (IrDA) has defined protocols for exchange and synchronization of data in infrared networks. Some of these protocols are adopted by the SIG because of the similarities between the Bluetooth and IrDA protocols, applications, and usage scenarios. The Object Exchange (OBEX) protocol is such a protocol. OBEX is a session protocol for peer-to-peer communication. OBEX defines the syntax and semantics of data that is sent between devices. The protocol is used for exchange of well-defined objects such as electronic business cards (vCard format), e-mail or other messages (vMessage format), calendar entries (vCal format) and others. Another IrDA-defined protocol, Infrared Mobile Communications (IrMC) enables synchronization of these same objects.

*Networking Layers*

Bluetooth uses a peer-to-peer network topology rather than a LAN style topology. But the technology allows Bluetooth devices to connect to other networks through a dial-up connection or via a network access point. If a dial-up connection is established to an IP-network, standard Internet protocols such as TCP, UDP, HTTP can be used to interact with the external network. The device may also connect to an IP-network through an access point using the Internet Point-To-Point (PPP) protocol. When this connection is established, the regular Internet protocols can be used to interact with the network. The Wireless Application Protocol (WAP) can also be used to interact with the network.

*TCS Layer*

One of the properties of Bluetooth technology is the ability to transfer voice traffic as well as data traffic. The Telephony Control Specification (TCS) layer is designed to support telephony functions. The TCS protocol includes call control functions, group management functions and a method for devices to exchange call signalling information without actually placing a call or having a call connection established.

*Applications*

The application group refers to software that is placed above the protocol stack as it is defined by the SIG. This software may be developed by device manufacturers or independent software vendors. The SIG does not define application protocols or APIs. Instead there are Bluetooth profiles, which define how to build interoperable applications that address various usage cases. The look and feel of these applications are however not defined in the specification, so in this area the application software developers have the ability to differentiate their products from others, and add extra features without violating the interoperability guidelines described by the profiles.

### 2.1.3 The Bluetooth product qualification process

Manufacturers who wish to develop products using the Bluetooth technology, the Bluetooth logo and trademark has to go through a qualification program. The Bluetooth SIG has delegated the responsibility for the qualification program to the Bluetooth Qualification Review Board (BQRB).

The qualification process is an expensive and time-consuming process. A company that develops Bluetooth products has to pay a significant amount of money to the BQRB to go through the process. In addition it may take a while before the BQRB has finished the job. Because of this, a company has an incentive to be as certain as possible that their product will be approved on the first attempt at the BQRB. If it fails, they have wasted a lot of money and their product may be delayed, which will also lead to loss of money. That is why companies should try to develop good testing routines themselves so that the qualification will be just a verification of their own testing. Automated conformance testing of the Bluetooth test specification may be one approach to develop better testing routines.

To emphasise and further motivate the need for our system, the remainder of this section is devoted to a detailed description of the Bluetooth qualification process.

The Bluetooth Qualification Review Board (BQRB) is responsible for the qualification process, and the Bluetooth Qualification Administrator (BQA) administers the process. Figure 2 shows the structure of authority delegation of the Bluetooth qualification process.

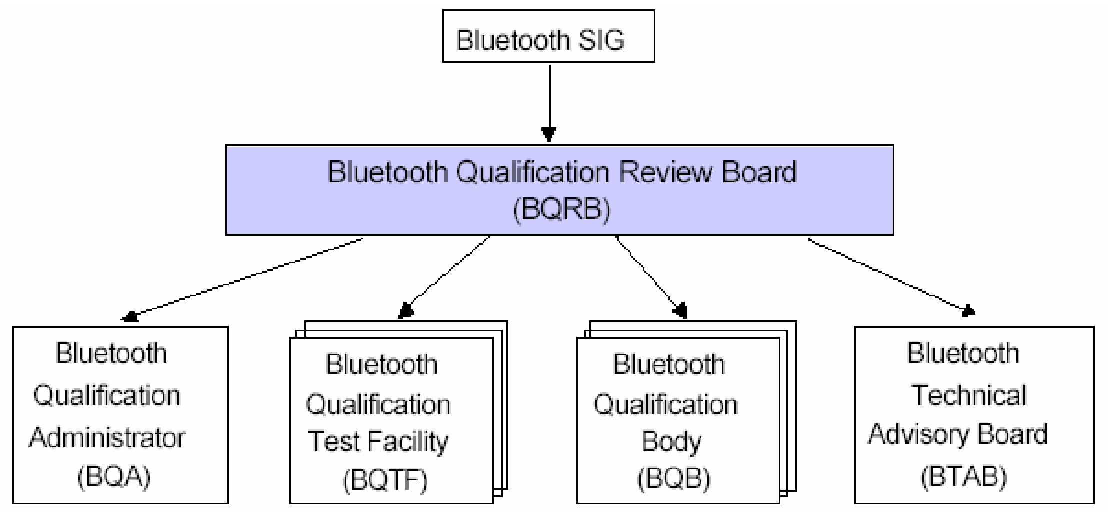
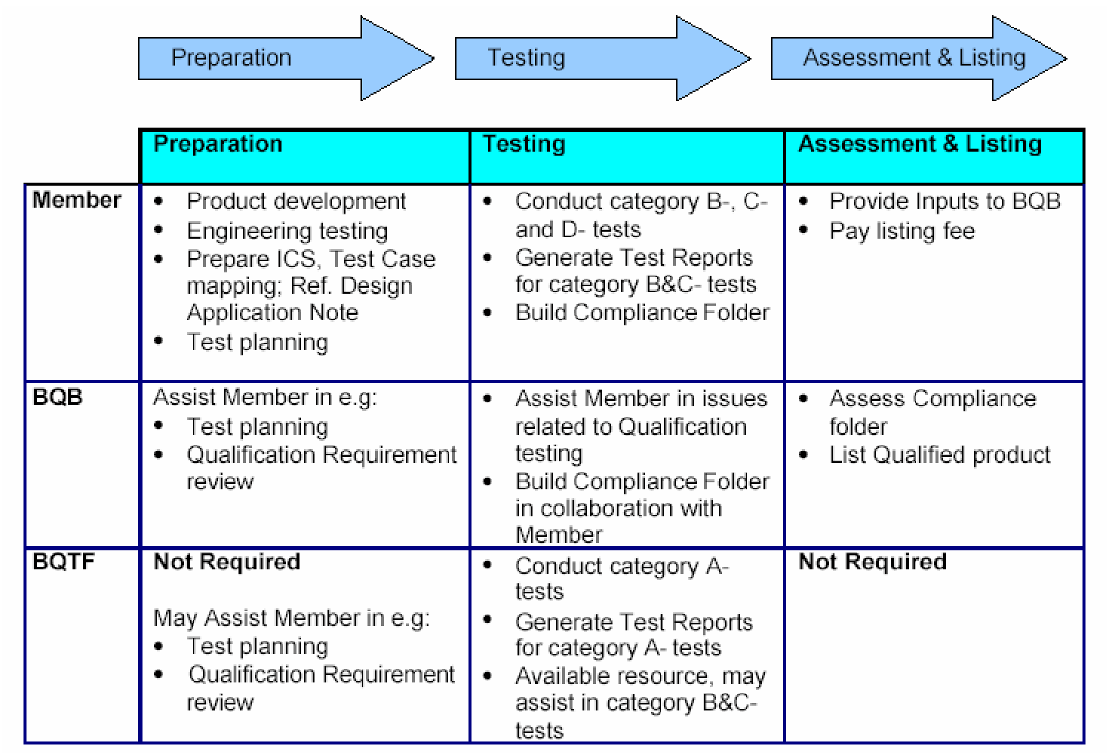

Bluetooth SIG
Bluetooth Qualification Review Board (BQRB)
Bluetooth Qualification Administrator (BQA)
Bluetooth Qualification Test Facility (BQTF)
Bluetooth Qualification Body (BQB)
Bluetooth Technical Advisory Board (BTAB)

**Figure 2 - Authority delegation for the Bluetooth qualification process (figure taken from [PRD, 2002])**

The qualification program is designed to protect the Bluetooth brand by promoting interoperability, declaring product capabilities, and defining a level of performance. To initiate the process the manufacturer has to become a Bluetooth member. There are two member types, Associate Member and Adoptive Member. One becomes a member by executing the applicable Bluetooth Agreement, which can be accessed from the Bluetooth web site. After becoming a Member, the manufacturer can select a BQB (Bluetooth Qualification Body). This is a person that will assist the manufacturer through the rest of the qualification process. The Member has to prepare a compliance folder, which contains test reports, test plans, technical product descriptions, user manuals, Protocol Implementation Conformance Statement (ICS) and Declaration of Compliance (DoC). This compliance folder will be used by the BQB as an objective evidence of compliance to the Bluetooth specification. Testing may also be performed at a Bluetooth Qualification Test Facility (BQTF).

The BQTF then provides the test report to the BQB for review. If the product is approved, the product will be ready for listing. The product will then be listed on the Bluetooth Qualification Product web site, along with relevant information/documents such as pre-tested components information, compliant portion declaration, etc [Fischer and Chin, 2003]. The Bluetooth Technical Advisory Board (BTAB) is a forum consisting of all BQBs and BQTFs. This forum is responsible for advising the BQRB on technical matters concerning test requirements, test cases, test specifications and test equipment. Figures 3 and 4 give an overview of the entire qualification process and especially which responsibilities the member, the BQB and the BQTF have in the different stages of the process. The flowchart shows the roles of the member, the BQB, the BQTF and the BQA during the process.


Preparation
Testing
Assessment & Listing

| | Preparation | Testing | Assessment & Listing |
|---|---|---|---|
| Member | • Product development<br>• Engineering testing<br>• Prepare ICS, Test Case mapping; Ref. Design Application Note<br>• Test planning | • Conduct category B-, C- and D- tests<br>• Generate Test Reports for category B&C- tests<br>• Build Compliance Folder | • Provide Inputs to BQB<br>• Pay listing fee |
| BQB | Assist Member in e.g:<br>• Test planning<br>• Qualification Requirement review | • Assist Member in issues related to Qualification testing<br>• Build Compliance Folder in collaboration with Member | • Assess Compliance folder<br>• List Qualified product |
| BQTF | **Not Required**<br><br>May Assist Member in e.g:<br>• Test planning<br>• Qualification Requirement review | • Conduct category A- tests<br>• Generate Test Reports for category A- tests<br>• Available resource, may assist in category B&C- tests | **Not Required** |

**Figure 3 - Process for Bluetooth Product Specification (figure taken from [PRD, 2002])**

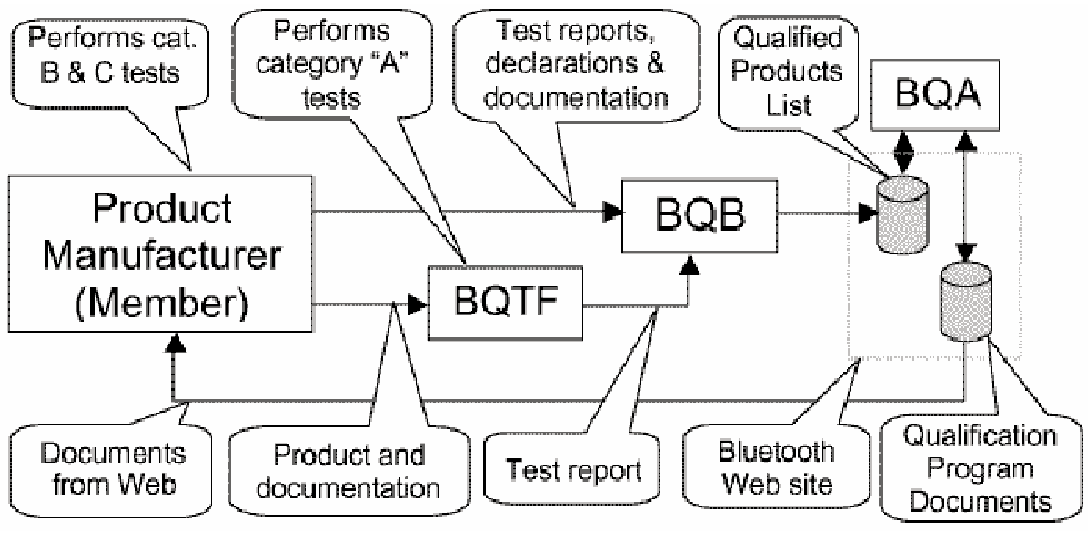

Performs cat. B & C tests
Performs category "A" tests
Test reports, declarations & documentation
Qualified Products List
BQA
Product Manufacturer (Member)
BQB
BQTF
Documents from Web
Product and documentation
Test report
Bluetooth Web site
Qualification Program Documents

**Figure 4 - Bluetooth Qualification Process Flowchart (figure taken from [PRD, 2002])**

### 2.1.4 The Bluetooth test specifications

The Bluetooth test specifications describe test cases for each protocol layer of the Bluetooth stack and each defined *profile*. A Bluetooth profile represents a usage model that the device under test is likely to use. The Bluetooth SIG has defined a series of such profiles to ensure interoperability. The test cases in the specification form the basis for conformance and interoperability testing of Bluetooth implementations. The conformance test cases are found in all protocol and in some profile specifications. These test cases are also called the Bluetooth Conformance Statements.

## *2.2 Testing*

In the following sections we first present a brief motivation for the need for software testing. We then present several testing methodologies, including the important aspects of test automation.

### 2.2.1 Motivation

In practically any kind of engineering activity, testing is used to verify the correctness of the built product. Therefore one can say that testing is one of the oldest forms of verification. Testing is also an important part of the software development process. Different testing techniques are used to improve the quality of systems and to make sure that the system acts the way it is supposed to. An ideal test is a test that succeeds only when a program contains no errors [Goodenough and Gerhart, 1975]. The ultimate goal of software testing is to help developers construct systems with high quality [Harrold, 2000].

As software systems mature, there is a tendency that the cost of maintaining them increases. The normal experience from development processes is that the cost of software maintenance will eventually become the major part of the total development cost. Up to two thirds of the overall cost can be traced back to software maintenance [Rothermel and Harrold, 1996]. A large percentage of the maintenance is due to testing [Wolverton, 1974] [Ramamoorthy and Ho, 1975]. Figure 5 shows the cost of the different phases of general software development [Boehm, 1987]. As the figure shows, testing takes up a significant amount of the total time.

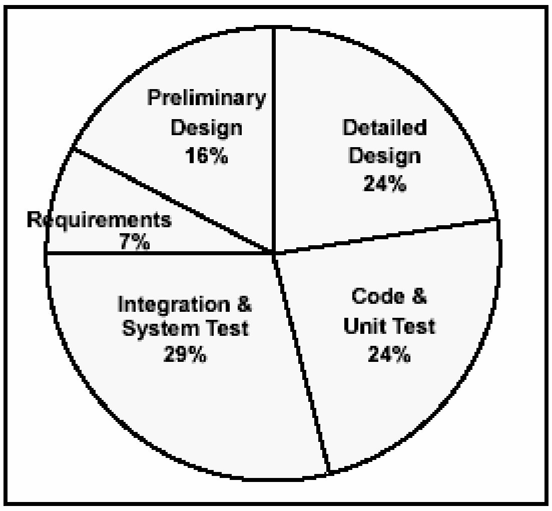


**Figure 5 - Cost of Software development (figure taken from [Boehm, 1987])**

In the future the testing process will take up even more of the maintenance costs, as the software becomes more pervasive, and is used to perform even more critical tasks. This new complex software will require even higher quality, which again requires more testing. With such high costs connected to testing, it is clear that efficient testing methods are needed to save time and money.

### 2.2.2 Positive and negative testing

Positive and negative testing are two complementary views on how to improve the quality of a software system. Positive testing tries to verify that a system conforms to its stated requirements. The requirements are a possible source to the design of the test cases. The positive testing process must be performed to determine if the system has the functionality that is required. A system that passes such a test will often be shipped to a customer because the positive testing process is likely to be an indication of the quality of the product [Engels et al., 1997].

Negative testing is to test that a system does not do what it is not supposed to do. This often means to test that a system works properly even if an unexpected event should occur. Negative testing is often used to test aspects of the system that are not well documented, and outside the scope of the requirements specification [Watkins, 2001]. While the test cases for positive testing is limited to the requirements of the system, negative testing has no such limitations. The possible amount of test cases for negative testing can grow without limits. If one wishes to use negative testing, it is important to choose the most relevant test cases and not use much time and effort on finding all or most of them. This is a problem pointed out by Dijkstra as he states: "Program testing can be used to show the presence of bugs, but never to show their absence." [Dijkstra et al., 1972] In other words: You can never prove that your system can handle all types of failures, but if you can prove that some of the most important of these failures is resolved, it is good enough for most customers/users.

### 2.2.3 Black box testing

There are two general methods of testing programs: black box and white box testing. Black box testing can be done without any knowledge of the internals of the system under test. The main goal is to check which output the system provides to a certain input. The focus often lies on requirements, i.e. the system's functionality. One can in this way verify that the system does what it is supposed to, without saying anything about how the requirements are resolved inside the system. The test cases for black box testing must be designed based on the external behaviour of the system [Myers, 1979].

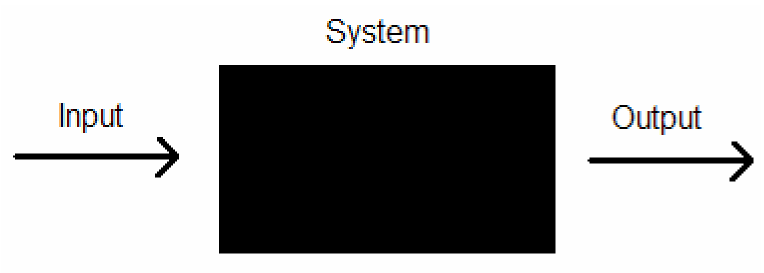


**Figure 6 - Black box testing**

### 2.2.4 White box testing

White box testing is often called glass box testing, because the test cases are designed with the knowledge of how the system under test is constructed. Compared to black box testing it means that you now are able to open the black box and test the mechanisms it is made of. When white box testing a system, you test each part of the implementation, that is, you try to execute each line of code given a set of inputs.

Testing in this way will find out if the logic of the code lines is implemented correctly [Myers, 1979].

### 2.2.5 Regression testing

The main goal of regression testing is to determine whether new errors have been introduced to a modified program. During software development, the code is constantly modified and tested. When new code is added or existing code is modified, the previously tested code should still work correctly. Regression testing is an expensive activity. It can in fact account for up to half of the cost of software maintenance [Rothermel and Harrold, 1997].

During regression testing there often exists a test suite with tests that can be rerun after a modification. Exactly which tests to run is a question that has lead to two different strategies of regression testing; the straightforward *retest-all strategy* and the more sophisticated *selective strategy*. The retest-all strategy re-runs all the tests in the suite. As such, for each modification of the code, all the tests in the suite are run. This approach will however most likely lead to many unnecessary tests, especially if a modification is minor. The retest-all strategy may therefore waste both time and resources. The selective strategy takes advantage of the fact that a modification often has an impact only on a few parts of the code. If there are no dependencies between the modified code and other parts of the code, these other parts need not be retested. Selective re-runs can as such save significant time and resources. Here we can see an analogy between retesting and recompilation. The *make* [Make, 2002] tool recompiles only source files that have changed and those files that depend on the changed files. Retesting is however a harder task than recompiling. This is because the dependencies between a test unit and the program entities it covers are harder to identify than dependencies between a program and its source files, which is specified in build scripts or makefiles [Chen et al. 1994]. The selective approach leads to two main problems: the problem of selecting tests from an existing test suite, and the problem of determining where additional tests may be required [Rothermel and Harrold, 1997].

### 2.2.6 Conformance testing

Conformance testing is the process of determining whether the implementation of a system meets the standards or specifications it was designed to meet. The motivation for conformance testing originates from the development of different implementations of given standards. International standards exist for many areas of computer systems. An example is communication protocols. These standards are important since their purpose is to guarantee that different systems can work together even if they are implemented in different ways. To make sure that an implementation of a protocol meets all the protocol’s requirements, the implementation must be tested against these requirements [Sarikaya et. al., 1986]. This testing process is called conformance testing. Since the specifications of a protocol often are well defined, it is possible to write test cases that can be standardized. By working with the testing methodology in parallel with the standards itself, the quality of the testing methodology can be comparable with the quality of the protocol standards itself [ETS, 1995]. With a well-defined testing methodology for conformance testing, the testing will give an even better guarantee of the quality of the product. An example of such a testing methodology is The Bluetooth Conformance Statements, which is the specification for conformance testing of implementations of the Bluetooth stack.

Conformance testing typically uses the black box testing technique because the test cases originate from a specification, and it is therefore the functionality that is tested. How the protocols are implemented doesn't matter as long as the implementation meets the specified requirements.

### 2.2.7 Interoperability testing

Interoperability testing is the process of testing whether the device under test can communicate successfully with other devices, preferably developed to the same standard. A standard may be implemented in several ways, so even if a device passes a conformance test, it does not necessarily interoperate with other devices because the different implementations may lead to conflicts. And two devices that interoperate may not have passed a conformance test. So the conformance test does not say anything about the interoperability between different implementations of the same standard. However, a passed conformance test will increase the possibility that two devices interoperate, since they both have correctly implemented a standard, but there is no guarantee that interoperability between the two systems are present. This is why we need interoperability testing.

Interoperability testing is a very important process, and it gets more important when one implements a standard that already has a lot of other implementations. Your product is probably worth less if it does not interoperate with the other products on the market, even if it has the most elegant and efficient implementation of them all. Interoperability testing may also be a time consuming process, since the tests has to be performed between several systems. Ten systems will require ten conformance tests, but the same ten systems require 90 interoperability tests [Kindrick et al., 1996].

### 2.2.8 Performance testing

Performance testing is the process of testing the performance of a system with respect to different criteria. The criteria can include user response times, system response times, external interface response times, CPU utilization, memory utilization, throughput etc. The most complete definition of performance would be to rate the effectiveness of the total system including the users [Browne, 1976]

Performance testing is often a problem area because system performance is frequently poorly specified [Watkins, 2001]. This may lead to poor, or in the worst case, no performance testing of an application under test. The focus is usually on the functionality tests. It seems fair that software developers prioritize the functionality tests, such as conformance testing and interoperability testing. A product that does not meet its requirements has less value if not any value at all. But often it seems that the primary problems that projects report after a release are not system crashes or incorrect system responses, but rather system performance degradation or problems handling required system throughput [Vokolos and Weyuker, 1998]. This is especially true for fault tolerant systems since performance is often degraded in such systems at the presence of faults [Huslende, 1981].

It is therefore essential to have some kind of a performance model when conducting performance testing. This model should define the test environment, the requested performance requirements of the system, and how the system can be tested. Based on this model a set of test cases can be made. The performance requirements should be provided in a concrete and verifiable manner, such as in a separate requirements or specification document, and might be provided in terms of throughput or response time. Since performance requirements must be included for average system loads and peak loads, it is important to specify those as early as possible, preferably in the requirements document [Vokolos and Weyuker, 1998].

The use of benchmarks is a traditional way of performing performance testing. A benchmark is a workload that can be used to obtain comparative performance measurements of different systems [Hitti and Joslin, 1965]. To test the system, it is simply run on these benchmarks. The challenge with benchmarks is to construct a benchmark that can act as similar to the natural environment as possible. There are two important aspects with this challenge: How will we know what a representative workload really is, and should the workload reflect an average workload or a very heavy or stress load. In both cases someone must have knowledge of the system and the environment it is run in. This person must make decisions on how the system most likely is used. Earlier versions of the system, historical usage data and similar systems can be of significant help here. A well known problem with benchmarks is that system manufacturers may design their systems such that they perform optimally when compared to a widely accepted benchmark, while the performance in real life may not be prioritized. It may result in systems with performance which is not as good as the tests say.

### 2.2.9 Fault tolerance testing

Fault tolerance testing is the process of testing how a system behaves under faulty conditions. Fault recovery testing is the process of verifying that following an error or exception, the system can be restored to a state where it can continue to perform successfully.

Fault injection techniques are a useful way of testing the adequacy of fault tolerance mechanisms, examining coverage of error detection schemes and studying system behaviour under faulty conditions [Gunneflo et. al., 1989]. Fault injection is simply a technique where faults are inserted into the system on purpose. In this way the system can be studied to see how it performs under faulty conditions.

Experiments based on fault injection techniques can be employed to achieve two separate objectives regarding the validation of fault tolerant computing systems: Fault forecasting and fault removal. Fault forecasting is to perform experiments that rate the effectiveness of various dependability mechanisms or to study system behaviour under faulty conditions. Fault removal attempts to eliminate the presence of faults [Arlat et al. 1991].

Fault tolerance testing is typically performed as white box testing. This is because it is much easier to insert faults to the system if you have access to the internals of the system. Most systems are designed to not let a user insert faults during ordinary usage. Faults often arise within the system and therefore it might be hard to insert faults by doing black box testing.

### 2.2.10 Reliability testing

Reliability testing is to test the robustness and reliability of a system under typical usage. The goal is to test whether the system will remain reliable in its intended environment over a required period of time. In addition to test the stability of the system, the data produced from reliability testing will form a basis for a statistical product capability assessment. This can make it easier for customers, who have specified stability requirements, to check if the system meets their requirements.

There exist two types of reliability testing: *integrity testing* and *structural testing*. Integrity testing is to verify the system's robustness and compliance to language, syntax and resource usage. An example is to execute a unit of a system repeatedly to ensure that there are no memory leaks. Structural testing is to verify that the system adheres to its design and formation. An example is to ensure that all links are connected, appropriate content is displayed and there is no orphaned content in a Web-enabled application [Watkins, 2001].

An example of automated integrity testing is the concept of *test monkeys* [Marsaglia and Zaman, 1993]. A test monkey is a kind of test which tries to discover what a user might do to a program. The term test monkey comes from the idea that if you have many monkeys typing a keyboard for while, some of them might hit a combination that may have a serious impact to the program. In other words it is a randomized way of getting different user inputs to a program. The test monkey may then reveal bugs that appear from user inputs that the test designers may not have thought of.

### 2.2.11 Test automation

As stated earlier in the thesis, testing in general, and regression testing in particular, is difficult and time-consuming. The process of testing is often done manually. This manual work does not have to be necessary when we have computers to help us. An approach often used in most industries, when trying to reduce costs and ease the work, is to automate the costly and difficult process [Ramamoorthy and Ho, 1975].

The main points one may achieve from test automation are:

- Speed
- Efficiency
- Accuracy and Precision
- Relentlessness

Speed is maybe the most obvious advantage. It takes some time to write the test, but once this is done, you may run the test over and over again in very short time, much faster than a manual test. Test automation can make the whole test process more efficient since the time used for running test cases is reduced. The extra time earned can be used to write more or better test cases. A tester is human and humans make mistakes. The accuracy and precision of your testing might be slightly worse after running many tests manually and you will probably make some mistakes. An automated test tool will always perform the same tests with the same accuracy each and every time. Finally, a test tool never gets tired, like a manual tester may do. It can keep on running for as long as you like and it will never give up [Patton, 2001].

The challenges with automated testing is how to feed input to the implementation under test (IUT), how to capture the IUT's output and how to evaluate this output.

There are several ways to feed test data to an IUT. One approach is to load test data from data files, which gives an opportunity to test the core functionality in detail, but not the user interface. Batch files can be used to run the program's commands and give input to the program. Almost all aspects of the program can be tested with batch files. Keyboard capture and replay is a technique that records all your keystrokes, mouse positions and mouse clicks. If you want to run a test many times you just record all your input actions the first time using a capture/replay program. Then you can run the very same test over and over again.

To evaluate the test you have to capture the IUT's output in a useful format. The capture may be done in several ways. A straightforward approach is to save to file everything that the IUT can write to disk. For output that is not supposed to be written to disk, like output intended for a printer, redirection of the output to a disk file is recommended. Then you can capture output you normally would not see, like the control characters sent to the printer. You may also send output to a remote computer through a network interface. The remote computer may then save the data on disk. Finally you may take a snapshot of the screen or active windows for later evaluation.

When you have captured the output it must be evaluated to check if it is the desired output. One technique for output evaluation is to find a reference program that already does what the IUT does. The output of the two programs may be compared to verify correct behaviour of the IUT. A similar approach is to construct a program similar to the IUT which works in parallel with the IUT, and is supposed to give the same output as the IUT. The outputs of the two programs may be compared. You may also build a library of correct outputs. When you create a new test case, you add the correct output to the library. The output of the IUT will be compared to this library. A final approach is to capture all outputs whether they are bad or good in separate files. Then investigate the files and mark them failed or passed depending on the result of the test. The next time the tests are run, the system flags the files where there are results that differ from the last run. These files are the ones that should be investigated. They will either show that previous correct tests now show failure or that a previous failed test now runs correctly or that a new bug has been introduced [Kaner et al., 1993].

There are many interesting challenges in the area of test automation, and there exists many systems today that offer automation of tests.

### 2.2.12 More testing techniques

Many other testing techniques exist. Here is a brief summary of some of these techniques as described in [Myers, 1979] and [Watkins, 2001].

Configuration/Installation testing is used to ensure that a system is correctly installed. This includes checking that appropriate files and connections have been created or loaded, system defaults are correctly set and interfaces to other systems/devices are working.

Documentation and help testing is to check the user documentation and help system information for conformance to the requirements specification document. This is often an overlooked aspect since it is thought to be outside the scope of the testing process. But this may be vital for new or naive users, who need to trust the documentation to be correct.

Security testing is to ensure that the features implemented in a system provide the required level of protection. The security requirements may specify the level of confidentiality, availability and integrity of the software. Security testing is mainly concerned with establishing the degree of traceability from the requirements through to implementation, and in the validation of those requirements.

Stress testing examines the system’s ability to perform correctly under instantaneous peak loads with the aim of identifying defects that appear only under such adverse conditions. Simulation is often used in stress testing since it can be hard to test under the conditions required for stress testing, e.g. it can be difficult to get a large number of users to log on to a system simultaneously.

Usability testing is to test how well a system can be used. Software usability is becoming increasingly important. Users are becoming increasingly sophisticated in their expectations of what a user interface should do and how it should support their activities. And there are of course the users who are unfamiliar with computer systems but still are expected to be able to use a particular application with minimal or no guidance or training. The techniques used in usability testing are among others conformance checks, where the application is tested against agreed user interface standards, user-based surveys, where psychometric testing techniques are used to analyze user perceptions of the system, and usability testing, where users are asked to perform a series of specified business tasks on the system to test the usability goals or requirements of the system.

Volume testing examines the system’s ability to perform correctly using large volumes of data with the aim of identifying defects that appear only under such conditions.

## 2.3 Existing Systems

There has been other work in the area of test automation and test case specification. The following sections will give an overview of some of this work. Section 2.3.1 will give a detailed description of the Tree and Tabular Combined Notation, which is an important framework for conformance testing of communication systems. Section 2.3.2 gives a brief introduction to one of many commercial test systems for Bluetooth implementations. Section 2.3.3 describes JUnit, a Java framework for regression testing.

### 2.3.1 Tree and Tabular Combined Notation (TTCN)

The ISO/IEC 9646 is a seven-part standard which defines a framework and methodology for conformance testing of implementations of OSI and ITU protocols. In [ISO/IEC 9646-3, 1998], the third part of the standard, the Tree and Tabular Combined Notation (TTCN) is described. The TTCN is a standard defined by the ISO for specification of tests for communication systems. TTCN has been chosen by the Bluetooth SIG as a preferred standard for specifying protocol and profile tests of Bluetooth implementations. A TTCN-specified test suite is a collection of various test cases together with all of the declarations and components needed [IEC, 2003].

The motivation for the ISO/IEC 9646 is conformance and interoperability testing. But the framework itself does not focus much on interoperability testing. Instead they see conformance testing as a road towards interoperability. With a huge number of protocols and vendors, interoperability sure is an issue, but with such a high number of implementations, the job of testing the interoperability between them can be too much even for the most eager tester. If all vendors have to pass a standard test suite, this can ease the interoperability testing. A passed conformance test does not guarantee interoperability, but it increases confidence. This is why the framework focuses on conformance testing. Issues like performance, reliability, fault tolerance, efficiency, etc are not taken care of in this framework [Graney, 2000]. The standard is divided into seven parts:

1. General Concepts
2. Abstract Test Suite Specification
3. Tree and Tabular Combined Notation (TTCN)
4. Test Realization
5. Conformance Assessment Process
6. Protocol Profile Test Specification
7. Implementation Conformance Statement

The focus in this chapter will be on part three, the Tree and Tabular Combined Notation. The TTCN is described in [Telelogic, 2001] and [IEC, 2003].

As mentioned earlier the framework described by the ISO/IEC 9646 is developed with conformance testing in mind. This is why one of their basic premises is that the implementation of the protocol, the implementation under test (IUT), is a black box. The conclusions that can be drawn about conformance of an IUT will be made by observing and controlling the events that occur at the lower and upper service interfaces of the IUT. These interactions take place at points of control and observation (PCO) and are expressed in terms of protocol data units (PDU) embedded in abstract service primitives (ASP). The test components which communicate with the IUT via the PCOs at the lower interface are called the lower tester (LT). The test components which communicate with the IUT via the PCOs at the upper interface are called the upper tester (UT). The Master Test Component (MTC) is a test component that always has to be present in the system. It is responsible for coordinating and controlling the test and for setting the final verdict of the test. Communication between test components both in the LT and the UT is achieved via coordination points (CP). Communication between the LT and UT is achieved by test coordination procedures (TCP). The LT is more complex than the UT. This is because it is responsible for the control and observation of the PDUs embedded in the ASPs that it sends and receives.

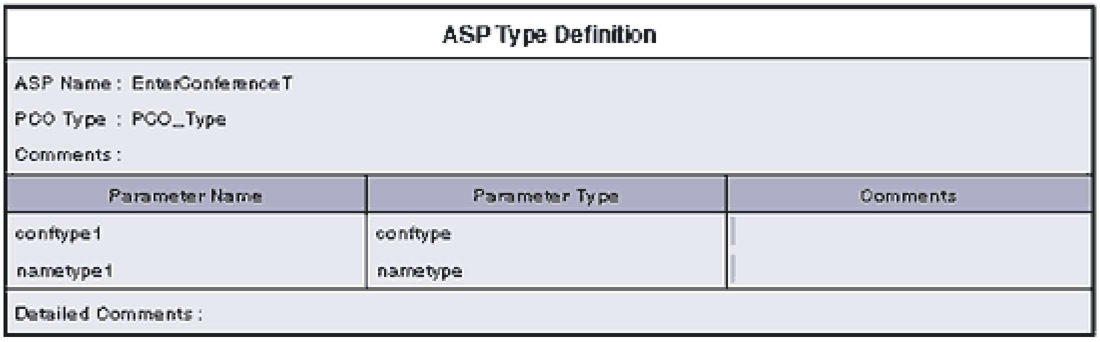
To test the IUT the sequences of interactions, or test events, need to be specified. A sequence of such events that specify a complete test purpose is called a test case. A set of test cases for a particular protocol is called a test suite. The TTCN is a notation for specification of test cases that is abstracted away from the architecture of any real test system that these test cases may be run on. These abstract test cases contain the necessary information to fully specify the test purpose in terms of the protocol that is to be implemented. This doesn’t mean that the notation itself is abstract. The definition of TTCN has become very precise, with regard to both syntax and operational semantics, and is now close to a programming language. The common TTCN notation is a graphical notation (TTCN-GR) where all information is presented using tables.

A TTCN test suite consists of four major parts:

1. Suite overview part
2. Declarations part
3. Constraints part
4. Dynamic part

The suite overview part is basically a documentation of the test suite. It contains a table of contents and a description of the test suite. The purpose of the suite overview is to document the test suite and increase readability and clarity.

The declarations part is used for declaring types, variables, timers, points of control and observation (PCO) and test components. All the types that that are used in the test suite are declared here. TTCN has been constructed to interface with the Abstract Syntax Notation One [ISO/IEC 8824, 1990]. There is no clear boundary between the TTCN an ASN.1 types. The distinction is there because there shall be possible to build types that can be used in parts of the protocol specifications that normally not use ASN.1. Hence the types are declared using either TTCN or ASN.1 type notation. Declaring of types in TTCN or ASN.1 is done in a graphical table instead of in a file. TTCN supports several built-in types, like INTEGER and BITSTRING. Most of these types are a subset of the ASN.1 built-in types and they are compatible with their ASN.1 counterparts. TTCN also allows the user to construct own types from the built-in types. There are specific tables for the definition of user-defined types.



ASP Type Definition

ASP Name : EnterConferenceT

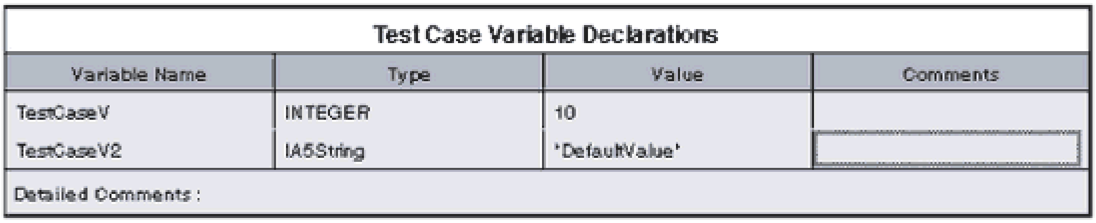
PCO Type : PCO_Type

Comments :

| Parameter Name | Parameter Type | Comments |
|---|---|---|
| conftype1 | conftype | |
| nametype1 | nametype | |

Detailed Comments :

**Figure 7 - Example of an ASP type definition (figure taken from [IEC, 2003])**

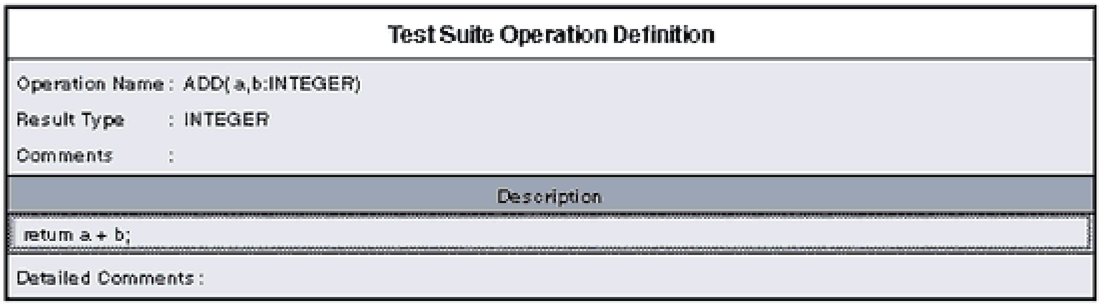


Test Case Variable Declarations

| Variable Name | Type | Value | Comments |
|---|---|---|---|
| TestCaseV | INTEGER | 10 | |
| TestCaseV2 | IA5String | 'DefaultValue' | |

Detailed Comments :

**Figure 8 - Example of test case variable definition (figure taken from [IEC, 2003])**



Test Suite Operation Definition

Operation Name : ADD( a,b:INTEGER)

Result Type : INTEGER

Comments :

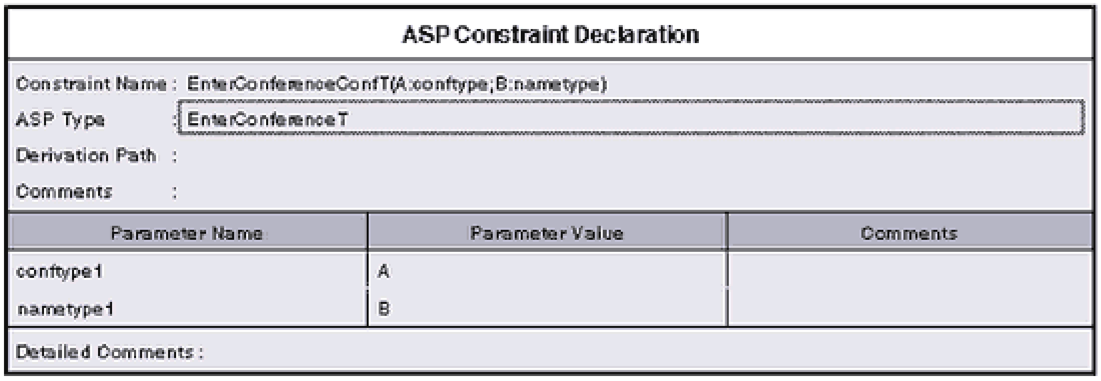
Description

```
return a + b;
```

Detailed Comments :

**Figure 9 - Example of test suite operation definition (figure taken from [IEC, 2003])**

The constraints part is used for describing the values sent or received. The structured types, PDUs and ASPs defined in the declarations part, are used as models to describe the messages sent on the PCOs. The instances used for sending must be complete, but for receiving there is the possibility to define incomplete values using wild cards, ranges and list. Constraints may be reused. They can be parameterized and the actual value can be assigned dynamically to the specific component stated for the argument.



**ASP Constraint Declaration**

Constraint Name : EnterConferenceConfT(A:conftype;B:nametype)
ASP Type : EnterConferenceT
Derivation Path :
Comments :

| Parameter Name | Parameter Value | Comments |
|---|---|---|
| conftype1 | A | |
| nametype1 | B | |

Detailed Comments :

**Figure 10 - Example of ASP Constraint Declaration (figure taken from [IEC, 2003])**

The dynamic part contains descriptions of the tests. The test description is an overview of the actual execution behaviour of the test suite. The dynamic part is created in a hierarchical and nested manner. The building blocks are test groups, test cases, test steps and test events. Three different types of tables exist for behaviour descriptions, Test Case Dynamic Behaviour, Test Step Dynamic Behaviour and Default Dynamic Behaviour.

To describe the test behaviour of the various test components TTCN uses a behaviour tree. Protocol specifications often use state diagrams or state tables to describe the behaviour of the protocol. Test cases are derived from these specifications. But since conformance testing is concerned with observing and controlling sequences of interactions at service interfaces it is more appropriate to use a tree to specify the test system behaviour. This tree has branches for all the possible sequences of interaction that may occur between any two given protocol states. This tree is called a behaviour tree. The tree structure is represented by using increasing levels of indentation to indicate progression into the tree with respect to time.

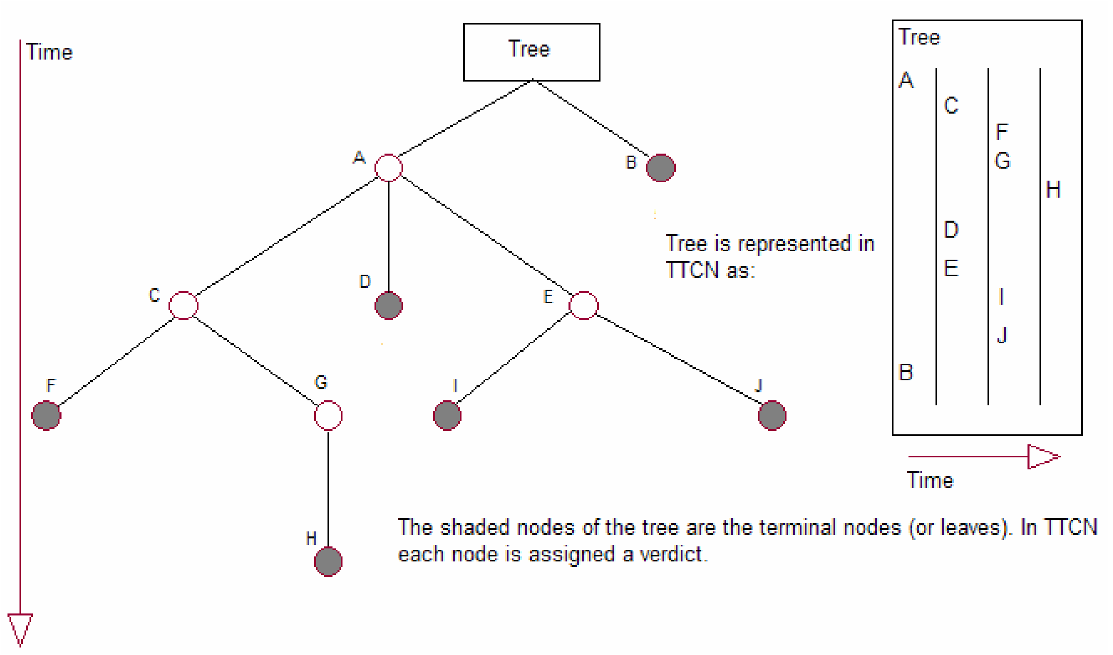

Time
Tree
A
B
C
D
E
F
G
H
I
J
Tree is represented in TTCN as:
Tree
A
C
F
G
H
D
E
I
J
B
Time
The shaded nodes of the tree are the terminal nodes (or leaves). In TTCN each node is assigned a verdict.

**Figure 11 - The behaviour tree represented in TTCN using indentation (figure taken from [Telelogic, 2001])**

A node in the behaviour tree is called a behaviour line. The behaviour lines are specified in dynamic behaviour tables (as described earlier). A sequence of one or more statements, together with the indentation information in a behaviour line is called a statement line. In figure 12, the light shading indicates a behaviour line while the dark shading represents a statement line.

| Nr | Label | Behaviour Description | Constraints Ref | Verdict | Comments |
|---|---|---|---|---|---|
| 1 | | A | | | |
| 2 | | C | | | |
| 3 | | F | | | |
| 4 | | G | | | |
| 5 | | H | | | |
| 6 | | D | | | |
| 7 | | E | | | |
| 8 | | I | | | |
| 9 | | J | | | |
| 10 | | B | | | |

**Figure 12 - The body of a dynamic behaviour table (figure taken from [Telelogic, 2001])**

The behaviour of the test system is expressed using statements. Statements can be split into three different types; events, actions and qualifiers. Some statements will be successful, i.e. *match*, depending on the occurrence of certain events. There are two types of events; input events and timer events. Some statements will always be successful, i.e. execute. These statements are called actions. Statement lines may include a qualifier statement, i.e. a Boolean expression. These statements are called qualifiers.

A set of statement lines at the same level of indentation, and in the same branch of the tree, are called a set of alternative statement lines or just alternatives. Execution of the behaviour tree starts at the root of the tree. The first set of alternatives is repeatedly looped. Each alternative is evaluated in the order of its appearance in the set. This continues until a statement line is successfully executed or matched. If a statement line is successful then the next set of alternatives is entered, and the process is repeated. Execution stops when a leaf of the tree is reached. A final verdict will also halt the execution.

The TTCN is a detailed and well defined notation for conformance test specification. When comparing TTCN to FAT we find that our framework does not have the intention to create a new test specification notation/language. FAT will try to make use of an already existing general-purpose programming language, Java, to investigate whether it is suitable for writing automated test cases. TTCN focuses on conformance testing and does that by performing black box tests. FAT also uses a black box testing technique similar to TTCN for conformance testing. But it tries to do more than just conformance testing and uses a white box testing technique to achieve e.g. fault tolerance testing.

### 2.3.2 IVT BlueTester

BlueTester [BlueTester] is a test tool for Bluetooth implementations. It is based on the Bluetooth specification and its main goals are to offer conformance testing through the module BlueTester for Conformance, and interoperability testing through the module BlueTester for Interoperability.

BlueTester offers a graphical user interface to the tester. The test cases are represented using TTCN and implemented in C. The Conformance test case procedures are viewed using a Message Sequence Chart (MSC). MSC shows the test procedure graphically by viewing the message flow sequence. Test results are compared against the expected results. These results are also logged for later use. The logging may be done in a complete and detailed fashion such that it may be used in an official Bluetooth qualification process. The test cases implemented by the program are easy to upgrade against the official Bluetooth specification, which makes the program always up to date with the latest standard.

The Interoperability test cases are based on the Bluetooth Profile Specification. The test case scripts execute these test cases and log the results in a way that is similar to the Conformance test cases. That means that the test logs for the interoperability tests may also be used as a basis for qualification of Bluetooth products. Similar to the Conformance test cases, the test cases for the Interoperability tests may easily be upgraded so that the test program always is up to date with the latest specifications.

Compared to FAT, this system has pre-defined tests for conformance and interoperability, while FAT lets the tester write his, or her, own tests and not just only for conformance and interoperability. BlueTester has a more detailed view of the outcome of the test process including a more detailed logging of test results than our system. BlueTester may only be used against Bluetooth implementations while FAT may be used against other protocol stacks that have similar architectures as the Bluetooth stack.

### 2.3.3 JUnit

JUnit [JUnit] is a regression test framework written by Erich Gamma and Kent Beck. It is used for writing unit tests in Java. JUnit is based on the Smalltalk testing framework [Beck] which has formed a basis for many testing frameworks based on programming languages like CUnit (C), CppUnit (C++), JUnit (Java), PhpUnit (PHP) and PyUnit (Python). We have chosen to describe JUnit here since our framework uses Java to specify tests.

JUnit is Open Source Software, released under IBM's Common Public License Version 0.5 and hosted on SourceForge [SourceForge]. JUnit defines how to structure test cases and provides the tools to run them. It offers functionality for evaluating the results of the tests so that the tester does not have to do this job manually. It also offers functionality to run more than one test at a time.

Creating a test is quite simple:

1. Implement a subclass of TestCase.
2. Create a constructor which accepts a String as a parameter.

3. Write a method that sets expected values from the test, and runs the method(s) that is to be tested.
4. If you want to check a test value, call assertTrue() with the boolean that is true if the test succeeds.

An example showing such a test method is shown here. This method tests if the sum of two Money objects with the same currency contains a value which is the sum of the values of the two Money objects. All examples in this section are taken from [Beck and Gamma].

```
public void testSimpleAdd() {
    Money m12CHF= new Money(12, "CHF");
    Money m14CHF= new Money(14, "CHF");
    Money expected= new Money(26, "CHF");
    Money result= m12CHF.add(m14CHF);
    assertTrue(expected.equals(result));
}
```

There are some constructs that are useful when writing several tests. One such is a *Fixture*. A Fixture is used when writing tests that use the same set objects. These set of objects is the Fixture. A Fixture is instantiated using the setUp() method and the tearDown() to release the resources allocated in setUp(). An example Fixture is shown here:

```
public class MoneyTest extends TestCase {
    private Money f12CHF;
    private Money f14CHF;
    private Money f28USD;

    protected void setUp() {
        f12CHF= new Money(12, "CHF");
        f14CHF= new Money(14, "CHF");
        f28USD= new Money(28, "USD");
    }
}
```

To write and run test cases against a Fixture one has to write a public test case method in the Fixture class and then create an instance of the same class with the method name as a parameter. Here is an example:

```
public void testMoneyMoneyBag() {
    // [12 CHF] + [14 CHF] + [28 USD] == {[26 CHF][28 USD]}
    Money bag[]= { f26CHF, f28USD };
    MoneyBag expected= new MoneyBag(bag);
    assertEquals(expected, f12CHF.add(f28USD.add(f14CHF)));
}
```

An instance of MoneyTest is created like this:

```
new MoneyTest("testMoneyMoneyBag")
```

When the test is run, the name of the test is used to look up the method to run. If there are several tests they should be organized into a *suite*. A suite is used to run several test cases together. To run a single test case, execute:

```
TestResult result= (new MoneyTest("testMoneyMoneyBag")).run();
```

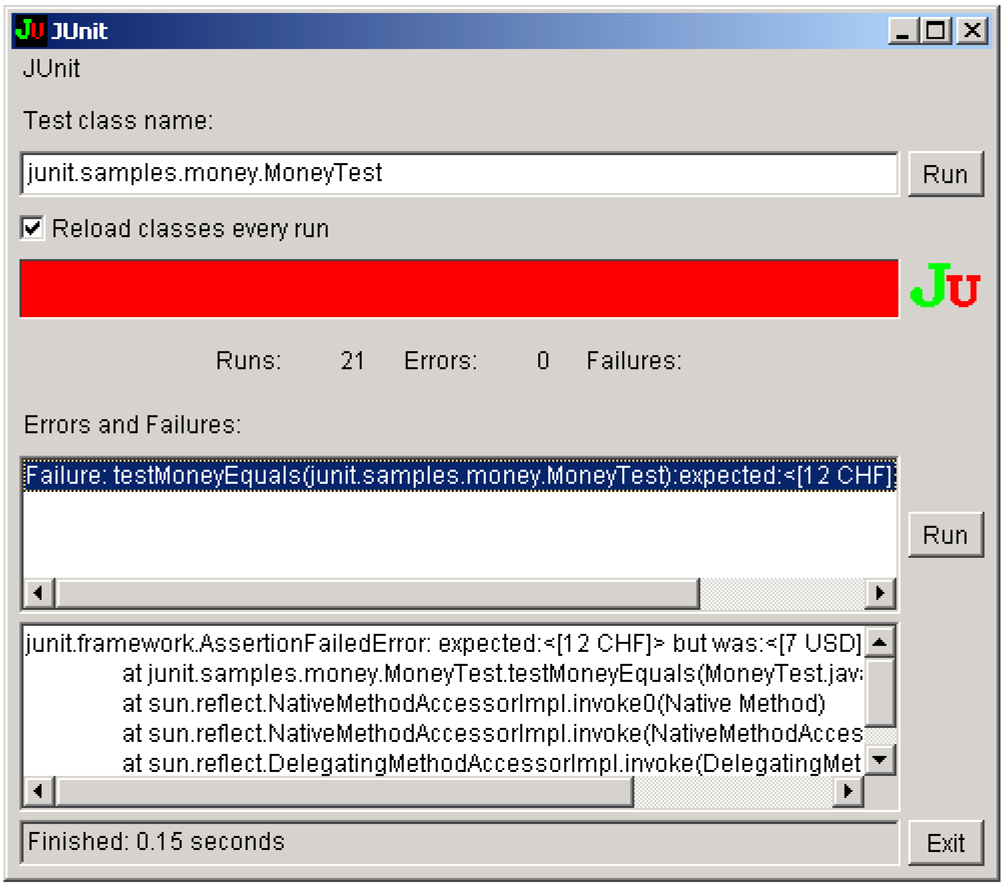
To create a suite of two test cases which can be run together, execute:

```
TestSuite suite= new TestSuite();
suite.addTest(new MoneyTest("testMoneyEquals"));
suite.addTest(new MoneyTest("testSimpleAdd"));
TestResult result= suite.run();
```

To run tests and collect the results, one has to make the test suite available to a TestRunner tool with a static method suite() that returns a test suite, like this:

```
public static Test suite() {
    TestSuite suite= new TestSuite();
    suite.addTest(new MoneyTest("testMoneyEquals"));
    suite.addTest(new MoneyTest("testSimpleAdd"));
    return suite;
}
```

There is both a graphical and textual version of the TestRunner tool. The graphical version is shown in figure 13.



junit.samples.money.MoneyTest

Runs: 21 Errors: 0 Failures:

```
Failure: testMoneyEquals(junit.samples.money.MoneyTest):expected:<[12 CHF]
```

```
junit.framework.AssertionFailedError: expected:<[12 CHF]> but was:<[7 USD]
        at junit.samples.money.MoneyTest.testMoneyEquals(MoneyTest.jav
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAcces
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMet
```

Finished: 0.15 seconds

**Figure 13 - JUnit graphical TestRunner tool.**

From figure 13 we can see that JUnit distinguishes between *failures* and *errors*. Failures are anticipated and checked for with assertions in the code. Errors are unanticipated problems.

Compared to FAT, JUnit also uses Java to specify tests. Our framework does not have quite as sophisticated functionality with Fixtures and suites, but has simplified assertion functionality like JUnit. Our framework offers the ability to test other systems through XML-RPC and not only systems residing on the same node as the framework itself. This is an advantage when performing tests of stacks that are to be executed on devices which have limited capacity to run other applications, like a test system.

# Chapter 3

# Architecture

The purpose of this chapter is to give an introduction to the architecture of FAT. The most important concepts of the design of the framework are included.

## 3.1 Overview

FAT provides a framework for automated regression testing of Bluetooth stacks. Both black box and white box testing are supported. To achieve white box testing, e.g. for fault tolerance testing, there must be some way for the tester to inspect the messages passing through the stack. If the stack layers are bound together with pointers to their upper and lower layers, it is possible to insert a layer between the original layers that can inspect messages passing through the layer. Such a layer will make it possible to insert, modify, copy and delete messages. With the help of these four operations it is possible to write and execute many different tests, both black box and white box tests.

Figure 14 shows the overall picture of the architecture of FAT.

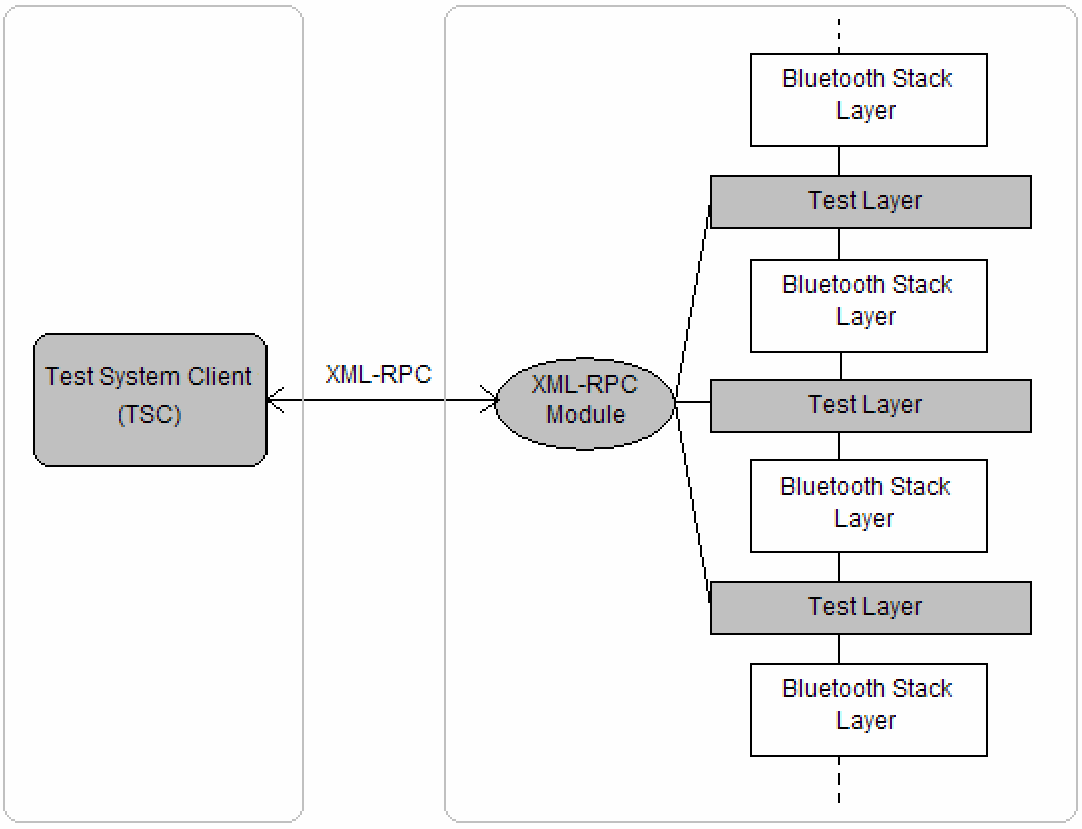


**Figure 14 – FAT architecture**

FAT consists of two main components:

- Test System Client (TSC)

- Test Layer Component

The TSC is the client side of the framework. Here is where the tests are written and executed. The Test Layer Component is the component that makes it possible to inspect messages passing through the stack. The Test Layer Component consists of one of more test layers and an XML-RPC module, which provides an interface towards the TSC. The communication mechanism between the TSC and the test layer is XML-RPC, which makes it possible to have the two main components residing on different nodes.

The components are further explained in the following sections.

## 3.2 The Test Layer Component

The Test Layer Component consists of the test layer(s) and the XML-RPC module.

### 3.2.1 The Test Layer

The test layer provides an API towards the TSC to insert, modify, copy and delete messages. The four methods offered by the API are:

- `Put`
- `Get`
- `Copy`
- `Delete`

`Put` inserts a message into the stack. The message is constructed by the TSC and inserted into one of the layers in the stack, including the test layer. Get takes a message out of the stack and sends it to the TSC. This method is used to modify a message. By using Put on the received message, the TSC can return the message to the stack after modification. Copy copies a message by both sending the message to the TSC and forwarding it up or down the stack. Delete removes a message from the stack by refraining from forwarding it. To know when to do any of these operations, the test layer has four modes, which decide which action to take with incoming messages.

The test layer is bound together with the rest of the stack using the binding mechanisms of the ObexCode stack development framework. The test layer is treated just like a regular stack layer, which means it has the same internal structure as other layers with pointers to the upper and lower layer, and methods for sending and receiving messages.

### 3.2.2 The XML-RPC Module

The XML-RPC module exports the methods that the TSC uses to cooperate with the test layer. There is one method for each of the methods of the test layer's API described in section 3.2.1. In addition there is one method that registers XML-RPC clients and returns a handle that must be used as first parameter with all calls to the module.

## 3.3 The Test System Client

The Test System Client (TSC) lets the tester write tests and execute them on a Bluetooth stack modified to include FAT test layers. The TSC provides the tester with a GUI where implemented tests may be chosen for execution. All test results are logged and written to file.

The test procedures are specified as ordinary Java methods. The test methods are organized by their category in different classes. Each test is represented in the system as an object of their corresponding class. The test classes share some functionality. This shared functionality is placed in a class which acts as a superclass for the test classes. The class hierarchy is shown in figure 15:

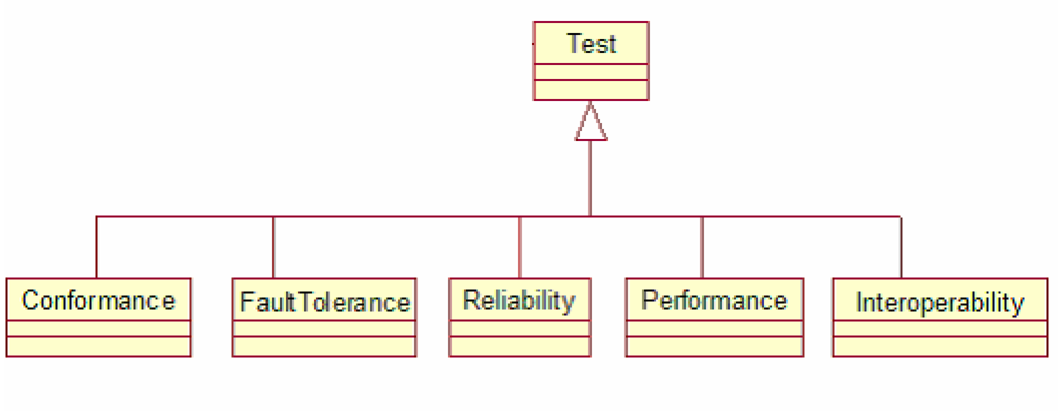


**Figure 15 - The class hierarchy of the test classes**

The TSC communicates with the test layer through XML-RPC. It contains therefore an XML-RPC module which acts as an XML-RPC client. XML-RPC is chosen as the communication mechanism because of the ability to let modules implemented in different languages easily communicate with each other. By choosing XML-RPC the design choices of the TSC could be made more independently from the implementation it is supposed to test.

## 3.4 Summary

This chapter has described the architecture of FAT. The framework consists of two main components: The Test System Client (TSC) and the Test Layer Component. The main idea of the architecture is to insert test layers between the original stack layers of the stack that is to be tested. These test layers may be used to insert, modify, copy and delete messages passing through the stack.

# Chapter 4

# Design and Implementation

## 4.1 Introduction

This section presents an overview of the components of FAT. The section also describes the design rationale behind FAT.

### 4.1.1 Overview

Figure 16 shows an overview of FAT's components. The grey boxes represent the FAT components.

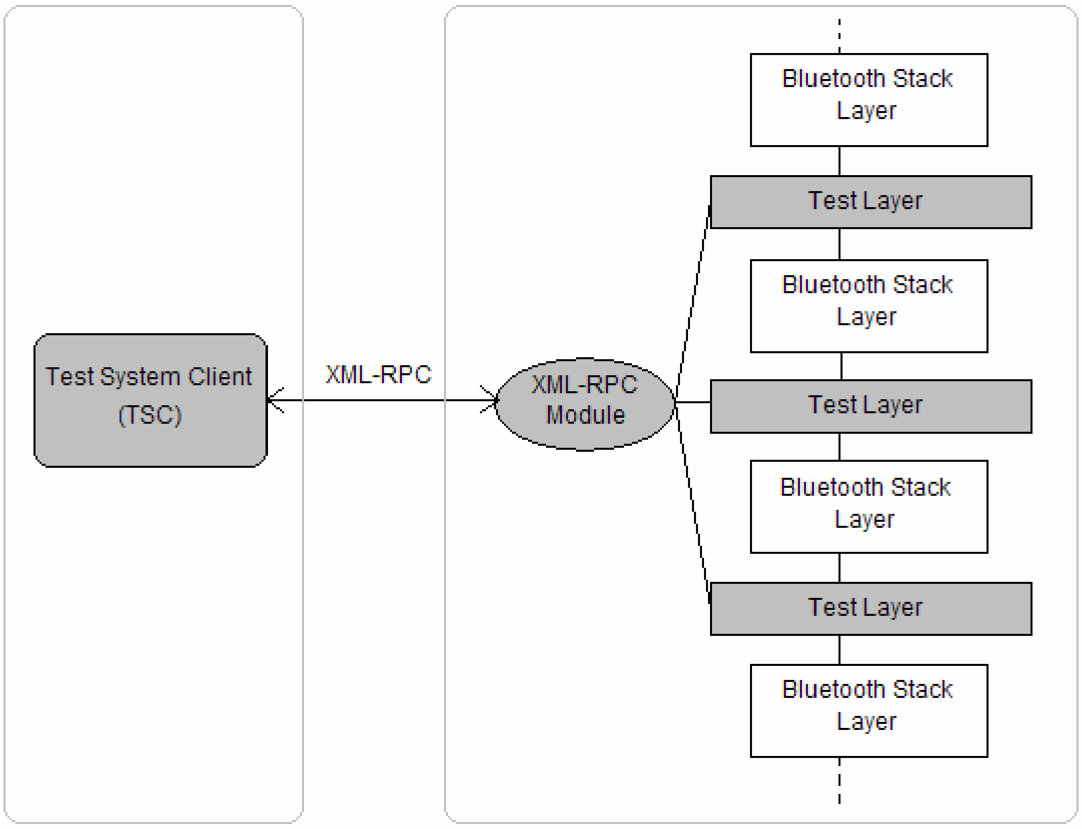


**Figure 16 - System overview**

- The Test System Client (TSC) is the client part of the framework. A tester will use this component to write tests and to execute them on a stack.
- The XML-RPC Module is the component that exports the methods that the TSC uses to interface with the stack.
- The Test Layer is a layer inserted between two layers in a stack. It provides functionality to the TSC to insert, modify, copy and delete messages passing through the test layer.
- The communication mechanism between the TSC and the stack is XML-RPC, which makes it possible to let the TSC run on a different node than the stack.

### 4.1.2 Approach

The different testing techniques discussed in previous chapters are often performed as either black box or white box testing. Conformance and interoperability testing are typically black box tests, while reliability and fault tolerance testing typically are carried out as white box tests. Black box testing a system is often straightforward, because one typically feeds the IUT with input data, collects the output data and finally checks if these data are accepted. The chief advantage of black box testing is that it does not require knowledge of the internals of the system, and that it can be easily automated. For our system the latter advantage is of importance. To black box test a protocol stack for conformance it is sufficient to call the methods that the stack is supposed to export and check the output.

To be able to perform white box testing of the protocol stack, which is the preferred method when performing fault tolerance and reliability testing, there must be some way for the tester to modify the internals of the stack. One such approach is to insert or modify the messages flowing through the IUT. One way of doing such a modification is to modify the stack-code itself. The test engineer has to go through the stack-code and insert code which changes the messages sent internally in the stack. This way the test engineer can observe the behaviour of a modified system and check if this behaviour is as expected. This is probably the most straightforward way of achieving message changes.

However, code modification is not an easy task. First of all, as a tester you must have good knowledge of the code in order to know where to insert your own test code. If you don’t know the code, test modification could be a time consuming task. This approach is also messy and it may be hard to keep control of what you have and what you have not tested. The original program code could easily ‘disappear’ among all the test code that is injected. The risk of modifying existing code, which you are not supposed to modify, is also present. Another fact is that this approach can not easily be automated. The automation of the test process is a main goal for this system. A final fact is that program code that is changed, e.g. by inserting test code, is not the same code as it was before. Just a simple thing like adding a single print-statement to the program code, can make the program behave in a totally different way, e.g. due to changed memory references.

If the test system shall perform white box testing it must be done in a different way than the code-modification approach. The approach that eventually will be chosen must primarily lend itself easily to automation. If we are satisfied with modifying the messages when they are in transition between the layers, there is an easy and efficient way of inspecting, creating, modifying or deleting messages. To achieve this, the stack layers have to have an interface for sending and receiving messages between them which can be used by a dedicated test layer. This test layer can be a generic layer, which receives messages sent from one layer, takes some or no action on the message, and sends the message to the next layer. The first layer will have no idea that its message is being hijacked, because it just sends the message without further control. The next layer just receives a message, regardless of who sent the message. What we get here is a layer that we can place in-between the original layers of the stack, a layer that has the same send and receive interface as the stack layers. This layer can then provide the functionality we requested, like inspecting, creating, modifying and deleting messages.

An interesting consequence of using layers like this is that the conformance tests we described earlier in the section, where the stack's exported methods are used to test functionality, can be done using these layers instead. Most of the methods that will be tested in such a test are actually just messages of some form that are sent through the stack. Since the layers will offer the possibility to insert messages, the messages that correspond to the stack's exported methods can be made and inserted to the stack instead of calling the methods directly.

The system is therefore designed in such a way that all tests use these generic test layers. This is done to provide one view of the entire test system, so all tests can be specified in a similar manner. One of the main points of our system is to construct a framework for testing where different types of tests can be specified using the same methodology. The design with generic test layers which can create, copy, modify and delete messages internal to a protocol stack, will provide such a commonality to all types of tests.

## 4.2 The Generic Test Layer

This section describes the test layer which provides functionality for testing of protocol stacks. The section starts by describing the particular stack FAT is designed for. We then continue with a description of the design and implementation of the test layer.

### 4.2.1 ObexCode Protocol Stack Development Framework

FAT is designed to test protocol stacks developed within the ObexCode network protocol development framework (NPDF). Although we are addressing the ObexCode Bluetooth stack in this thesis, FAT should be applicable as a testing framework for any protocol stack developed within NPDF. In the remainder of this section we summarize parts of NPDF particularly relevant for FAT.

Each layer in a NPDF stack has two pointers, one that points to the layer above itself, the `up` pointer, and another that points to the layer below, the `down` pointer. To stitch layers together, the binding functions `oc_bind_up` and `oc_bind_down` are used. These functions rearrange the `up` and `down` pointers of the layers involved in the call. Both functions take two parameters. `oc_bind_up` takes a pointer to itself and a pointer to the layer above it. It sets its own `up` pointer to point to the layer above it. The above layer's `down` pointer is set to the initiating layer. The `oc_bind_down` does exactly the same, except it sets its own `down` pointer and the other layer's `up` pointer. The binding is done at start-up time, when the stack is configured. A layer is created using the function `oc_<layer_name>_new`. It returns a handle to itself. The binding of layers is shown in figure 17.

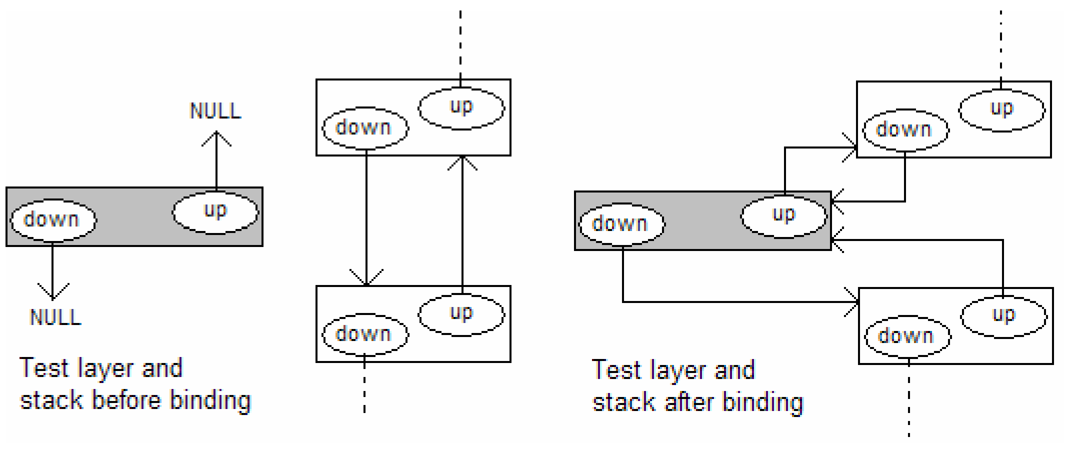


**Figure 17 - Binding of layer to ObexCode stack**

The format of the messages sent through the stack is simple. A message contains an event code which identifies the particular type of message.. A message also contains a handle to the source layer, and to the destination layer. The event code, together with the source handle, is used by the receiving layer to decide how to process the message. The message also contains a pointer to a data buffer. This data buffer contains the actual message data.

The messages that are sent between layers are sent using the function `oc_call_msg`. The function takes four parameters: The layer the message is sent from, the destination layer, an event code and the message data. Usually, when a message is sent down the stack, the destination layer is set to the layer pointed to by the `down` pointer. And for messages going up the stack the destination layer is set to be the layer pointed to by the `up` pointer.

When `oc_call_msg` is called, a message is constructed from the parameters, and sent to the destination layer. The function `oc_msg_proc` is the function that is called in the receiving layer when a message is received. It basically checks the event code and source layer of the message and takes the appropriate action based on these two parameters. The message flow is shown in figure 18.

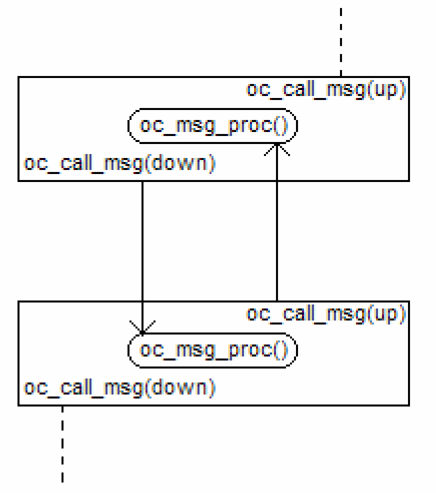


**Figure 18 - Message flow through ObexCode stack**

The stack that FAT has been tested against contains only the HCI layer from the Bluetooth specification. Below the HCI layer in this stack is a layer called

AsyncIO, which will act as a Bluetooth module. Although simplistic, these two layers are sufficient to verify the basic functionality of FAT.

### 4.2.2 The Test Layer

The purpose of the test layer is to enable the Test System Client (TSC) to insert, modify, copy and delete messages sent through an ObexCode NPDF stack layer. A graphical representation of the API is shown in figure 19.

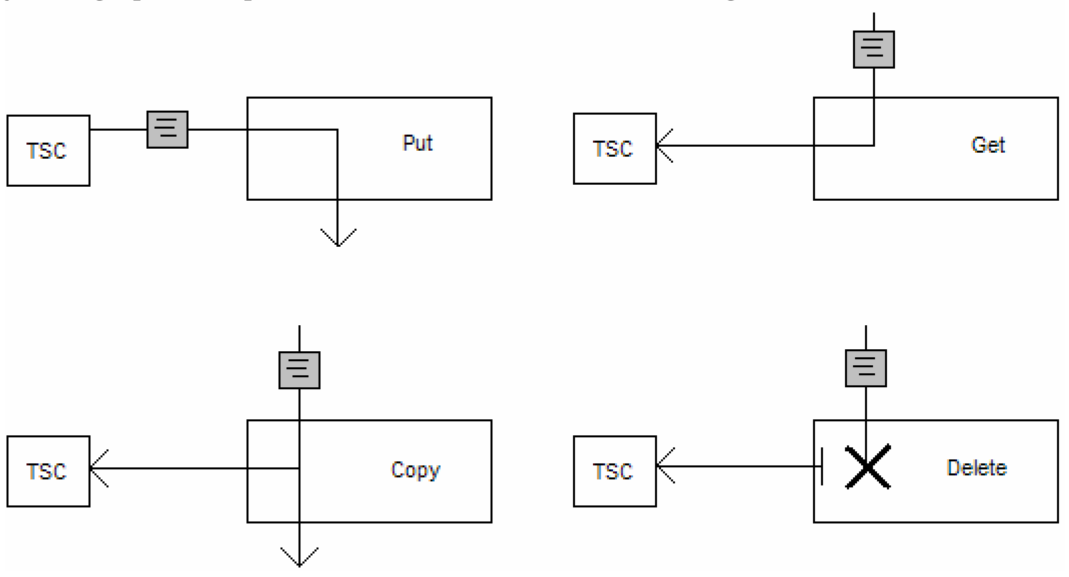


**Figure 19 - Test Layer API**

- Put provides the functionality to insert messages into the stack.
- Get lets the TSC modify messages. Get takes a message out of the stack and sends it to the TSC. The TSC must do a Put call to insert the modified message back to the stack.
- Copy sends an incoming message both to the TSC and further down, or up the stack, depending on where it originally came from.
- Delete removes messages by refraining from sending incoming messages to the next layer in the stack. It sends a confirmation message to the TSC when the operation is done.

The test layer has to know when to do any of these operations or do nothing at all. For the test layer it is not enough to just look at the event code when deciding the appropriate action to take with the message. It needs to know which messages to copy, modify or delete. To help the test layer with this decision the layer defines four modes:

- Get mode
- Copy mode
- Delete mode
- Normal mode

These four modes correspond to the four possible outcomes of a message arrival. The Get mode is for the modify situation, where a message is taken out of the stack and returned to the TSC for modification. There is one mode for copy, where an incoming message is copied and sent to the TSC, and also sent further down or up the stack depending on where it originated. When a message is taken out of the stack or copied, the event code and the actual data that the message carries is taken out and sent to the TSC. The third mode is when a message is deleted, which means that an incoming message is simply not forwarded at all. The fourth mode is when we have a ‘normal’ situation. In this mode incoming messages are just forwarded to the next layer. This mode is for messages that are not modified in any way by the test layer, and for messages that are inserted to the test layer from the TSC.

The layer is initialized with the Normal mode. To change mode, the TSC has to make a call to the test layer. The XML-RPC module which belongs to the test layer, exports four methods, `Put`, `Get`, `Copy` and `Delete`, which correspond to the API defined by the test layer. These are the methods that are used by the TSC to communicate with the test layer. Basically, `Get`, `Copy` and `Delete` sends a message to the test layer telling the layer to change mode to Get mode, Copy mode and Delete mode, respectively. Hence after a `Get` call the test layer is in Get mode and knows that the next incoming message shall be sent to the TSC and not forwarded to the next layer, as it would in the Normal mode. The test layer remains in this new mode until it has done the appropriate action with the first message that arrived after the mode change. The mode is then set to Normal. The test layer is implemented such that when it is in a mode other than Normal it takes the next incoming message and does the appropriate action, before resetting the mode to Normal.

The functionality described in this section could easily be improved by looking at the event code and the content of the incoming message and then decide to take some action or just forward it. This approach could be useful if you are sending a lot of messages through the stack but you are interested in one particular type for e.g. copying. Another improvement is that the test layer could return to Normal mode after a certain number of messages have passed and not just after one message, such as in the current implementation. This approach is useful when one wishes to copy all or some number of messages flowing through the stack. Both of these approaches require extra parameters to be added to the `Get`, `Copy` and `Delete` methods, that is one parameter for the event code(s) to look for and one parameter for the number of messages to examine. These improvements are currently not implemented in the test layer, but they are implemented by the exported methods of the XML-RPC module. These methods include an event code parameter and a number of messages parameter that can be used by the test layer if it implements the required functionality. Another improvement to the test layer could be to have multiple modes, which means that the layer could be in both delete and copy mode simultaneously.

The way the test layer works now, the `Get`, `Copy` and `Delete` methods just change the test layer mode and then returns to the TSC with no value. It is when a message is inserted to the stack that the result of these methods is actually returned. Another approach could be to use blocking calls that waits for messages to arrive, wakes up and does the appropriate action, and then returns to the TSC with the result. This approach requires the ability to make asynchronous calls from the TSC. A problem with this approach is that it might have to return more than once to the TSC, e.g. if several messages are to be copied. This could be solved by saving the copied messages in the layer and returning them to the TSC when all copying is done. In the current implementation however, this problem would not occur since only one message at a time is copied.

An interesting approach, which could make the test layer handle almost any situation required, is if the test layer could offer an API to the TSC to dynamically download code. Dynamically downloading of code is a well known technique for making a system more flexible and to increase its performance. Examples of down-loadable systems are SPIN [Bershad et. al., 1995] and U-Net/SLE [Oppenheimer and Welsh, 1997] [Welsh et. al. 1998]. By downloading code dynamically, a tester could control the behaviour of the test layer in greater detail than before. It may also reduce the traffic between the TSC and the test layer when e.g. modifying messages since the messages now may be evaluated at the test layer.

A light-weight alternative to a code down-load approach is using packet filtering technologies such as the Berkeley Packet Filter (BPF) [McCanne and Jacobson, 1993]. BPF defines filters at kernel level which can filter incoming packets. This approach is an alternative to a common approach for network monitoring where a user level network monitor gets copies of all incoming packets. Often a network monitor is interested in just a subset of the packets, so if incoming packets are investigated first at kernel level, and only the interesting packets are forwarded to the network monitor, it will improve performance. By introducing some kind of filtering mechanism in the test layer, we could increase the functionality and improve the performance of FAT.

The test layer contains some state. It is in the current implementation reduced to a variable telling which mode the layer is in. But the improvements mentioned earlier in the section requires more state to be present, like the number of messages to be treated by the layer and which event codes to be treated. Practically any statistics of messages passing by the layer may also be saved in the layer. Such statistics may be the number of messages passed, types of messages passed, timestamps of the messages passed, etc. To get this information from the test layer, new methods for the TSC are required. Such methods will have to send a message to the test layer requesting some information and will receive this information in return. Figure 20 shows an example of a method, `GetStats()`, which returns the message statistics of the test layer to the TSC.

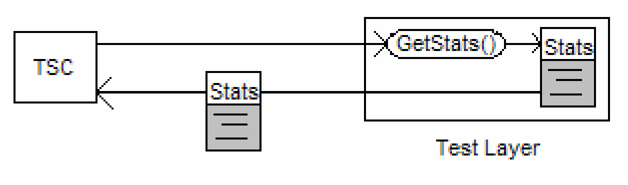


**Figure 20 - Get message statistics from test layer**

The test layer in its present form is very thin. It hardly contains any state, and the functionality implemented is small and simple. The reason for this is that Bluetooth is a technology designed for devices which often have few resources, such as memory and processing power, than ordinary computers. Adding layers to the stack should preferably introduce as little extra load on the system as possible. Therefore the test layer is designed such that it should be possible to do as much as possible for the TSC by adding as little as possible to the stack. Because of the desire to have a thin layer, the improvements described in this section may not be desirable features.

Saving messages and statistics about them will require too much memory. Instead all messages have to be sent to the TSC right away, like it is done in the current implementation. Message statistics could also be sent straight to the TSC without saving, if statistics is to be implemented later. The same problem applies to downloading of code. There might not be enough memory to store the downloaded code. The low processing power might also be an issue when introducing more functionality to the stack by downloading code.

### 4.2.3 The XML-RPC Module

The FAT architecture describes the Test System Client, an external client where tests are written end executed. To let this client execute tests on a stack, a communication mechanism between the client and the test layers is required. Since the client and the test layers may be running on different platforms, the communication mechanism must handle conversion of data types, e.g. between little and/or big-endian. We have therefore based our communication mechanism on XML-RPC [XML-RPC, 1999].

XML-RPC is a remote procedure call protocol. It uses HTTP as the transport protocol and XML as message encoding. An XML-RPC message is a HTTP-POST request. The body of the request is in XML. The server executes the procedure called from the request and the return value(s) are also returned as XML. XML is chosen as message encoding because it is a standard that is widely accepted and supported by many platforms which makes it possible to make procedure calls between a wide range of platforms. Figure 21 gives an overview of the process of making an XML-RPC call.

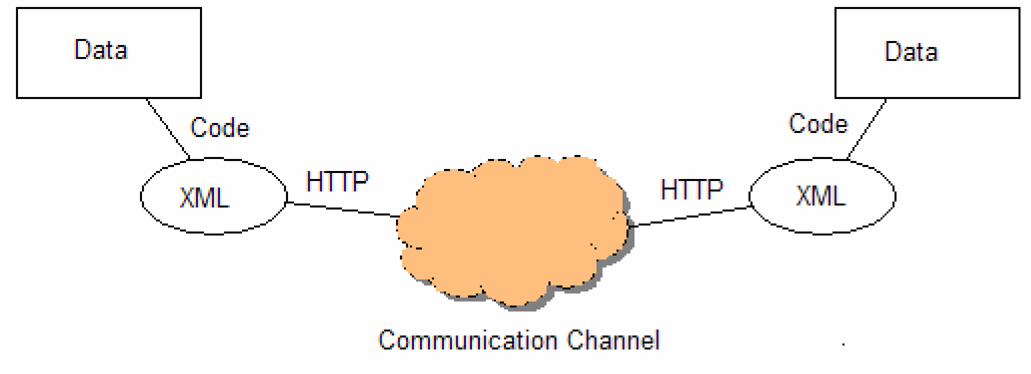


**Figure 21 - Overview of XML-RPC**

From figure 21 we can see that a program that wishes to make an XML-RPC call has to encode the procedure call into an XML document. This document is passed as a HTTP-POST request over some communication channel, like a serial cable or a network connection. The receiver has to decode the XML document and execute the call locally. The return messages are encoded into an XML-RPC message which is sent back to the caller as the return value for the original request. The caller decodes the XML response and can receive the return value.

A key advantage with XML-RPC is that applications implemented in different languages can easily communicate with each other. The motivation for using XML-RPC as communication mechanism is rooted in this property. Using XML-RPC, the TSC may be implemented in any programming language supporting XML-RPC and it will make the framework more generic. But XML-RPC is expensive because it consumes much memory. This disadvantage is however evaluated against the advantage of XML-RPC and found to be tolerable. Another advantage of choosing XML-RPC is that the ObexCode NPDF relies on XML-RPC as its external communication mechanism. The fact that the stack already uses XML-RPC is a good reason for choosing it as communication mechanism between the test system and the test layer.

The XML-RPC module exports four methods: `Put`, `Get`, `Copy` and `Delete`, which corresponds to inserting, modifying, copying and deleting messages. With the help of these four methods, the test system may perform the four actions desired. An example of an XML-RPC request to `Copy` is shown below:

```
POST /RPC_TEST HTTP/1.0
User-Agent: Frontier/5.1.2 (WinNT)
Host: cs.uit.no
Content-Type: text/xml
Content-length: 181


<?xml version="1.0"?>
<methodCall>
   <methodName>Copy</methodName>
   <params>
      <param>
         <value><int>13934442</int></value>
      </param>
      <param>
         <value><int>41</int></value>
      </param>
      <param>
         <value><int>1</int></value>
      </param>
   </params>
</methodCall>
```

The body of the message shows the method name and three parameters. This information is all the XML-RPC module needs to know to execute the `Copy` function. Further explanation of the `Copy` method and its parameters is presented later in the section. The module exports another method, `New`. This method has nothing to do with the test layer. It is a method that registers XML-RPC clients, and it must be called by all clients before making a call to the other exported functions. `New` returns a handle that must be used as first parameter to all the other exported methods.

The XML-RPC module communicates with the test layer and the other layers in the stack using the same mechanisms as the test layer, i.e. `oc_call_msg` for sending messages and `oc_msg_proc` for receiving messages. The only difference is that the module is not bound together with the stack layers like the test layer. That is why we call it a module rather than a layer since it is not part of the stack. The module needs to contain pointers to the layers that it will send messages to, since it does not have the `up` and `down` pointers of the stack layers. Most of the messages will be sent to the test layer(s), but the `Put` method may insert messages in any layer of the stack, so pointers to all relevant layers need to be initialized when the module is instantiated. Received messages mostly come from the test layer(s) but if a message has been inserted to another layer, this layer may return a message to the XML-RPC module.

The `oc_msg_proc` unpacks the data from these messages and return the data to the TSC. Following is a description of the four exported methods.

The `Put` method inserts a message to a layer in the stack. The message to be inserted is partly constructed by the TSC and partly by the `Put` method. The client constructs the data part of the message which is one of the parameters to `Put`. The other parameters are the handle returned from `New`, the event code the message shall use and an ID telling which layer to insert the message. `Put` unpacks the parameters and uses the message data and the event code as parameters to `oc_call_msg`, together with a pointer to the module and the destination layer which is decided from the layer ID. A message is now constructed and sent to the destination layer.

The `Get` method sets the mode of the test layer to modify mode. The parameters of the method are the handle returned from `New`, the event code of the message to be modified and the number of messages to be modified. The last two parameters are packed as the message data of the message to be sent to the test layer. The functionality of selecting which and how many messages to get is, as described in section 4.2.2, not implemented in the test layer, but later implementations can make use of data load following the message from `Get`. The module has defined three event codes, one for each of the calls `Get`, `Copy` and `Delete`. The event code for `Get` is used as the event code in the call to `oc_call_msg`, so that the test layer can recognize this as a `Get` message.

`Copy` and `Delete` are almost similar to `Get`. They take the same parameters and send the same data to the test layer. The only difference is the event code of the message sent to the test layer.

### 4.2.4 Summary

This section has described the test layer that act as an interface towards the stack that is to be tested. To perform white box testing in a way that can be easily automated, one can insert test layers in-between the original stack layers and insert, modify, copy and delete messages that pass through the test layer. The test layer opens also for black box testing of the system. An XML-RPC module is the communication interface towards the TSC. The XML-RPC layer exports four methods: `Put`, `Get`, `Copy` and `Delete`. With the help of these four methods it should be possible for the TSC to execute a significant number of relevant tests.

## *4.3 The Test System Client*

This section describes the Test System Client (TSC). The chapter reveals the design and implementation of the different modules of the test system. It starts with a short introduction and continues with an analysis of the system as a whole. The following sections describe each of the modules in more detail.

### 4.3.1 Introduction

The TSC implements the client component of the FAT framework. Users interact with the TSC to specify and execute specific tests. The responsibility of the TSC can be summarized like this:

- Specify a test
- Execute the test
- Log and view the test result

The design and implementation will reveal how these requirements are realized in the TSC. The TSC has been divided into four modules:

- Test module
- GUI module
- Control module
- XML-RPC module

The Test module is the part where the tests are specified. The GUI module represents the system's graphical user interface towards the user. The Control module is responsible for the TSC's internal interaction between the GUI module and the Test module and for logging of test results. The XML-RPC module represents the communication interface towards the IUT.

The TSC is implemented in Java using Java$^{TM}$ 2 SDK, Standard Edition Version 1.4.1 [Java]. Java is an object-oriented, high-level language. Java programs are running on top of the Java Virtual Machine (JVM). Because of the JVM, Java programs may run on almost any platform, which makes the language highly portable.

The TSC is implemented in Java because of the language's high level of portability, which makes it easy to run the system on almost any kind of processing node in any environment. Another reason for choosing Java is that it seems to be a good language for defining tests. The tests that are to be executed must be represented in some way. There exist several description languages with the purpose of defining tests. Instead of using a test specification language or developing a new specification language, we wanted to investigate the properties of a high-level language like Java, to check if it is suited for specifying tests. The reason for this is that Java is a relatively easy, yet powerful programming language. There should be enough properties to the language to define tests. Another reason is that the TSC itself is written in Java. A test written in Java will easily interoperate with a system also written in Java.

The TSC may be run on the same node as the IUT but it should also be possible to run it on a different node. This is done in order to make it possible to evaluate the IUT on the target device. This will not have any practical influence on the system since the XML-RPC is used anyway.

The rest of the chapter will describe the design and implementation details of the TSC. The Unified Modelling Language (UML) [UML, 2003] has been used to describe the different parts of the system. UML is chosen because it is a standardized and well known graphical language for visualizing, specifying, constructing and documenting artifacts of object oriented systems.

### 4.3.2 Analysis

The design of the TSC also contains a brief analysis of the system. The analysis is supposed to give an overview of the system without too many details.

In the analysis of the TSC, two actors and four classes are defined. The actors are the Test Engineer and the Implementation Under Test.

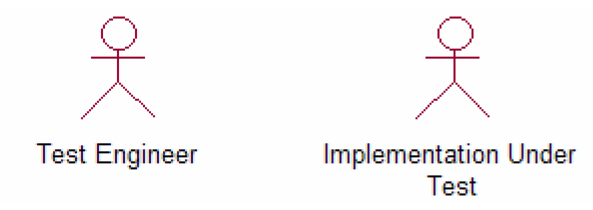


**Figure 22 – Actors of the test system**

- Test Engineer – The person that will use the TSC to test an implementation.
- Implementation Under Test – The implementation that is to be tested. This actor will correspond to the test layers and the protocol stack described in section 4.2.

There are two boundary classes, one entity class and one control class in this model. The relationship between the classes is shown in figure 23.

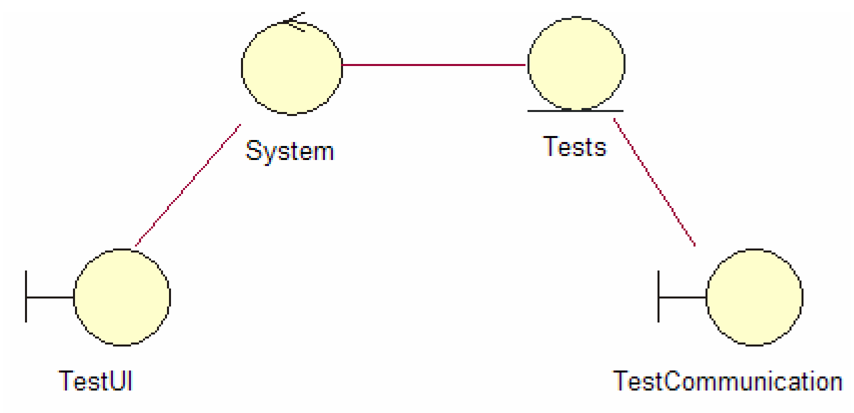


**Figure 23 - Analysis classes of the test system and their relations**

- TestUI – The interface towards the Test engineer. The Test engineer communicates with the TSC through this class.

- System – Controls and coordinates the information sent and received between the Test Engineer and the rest of the system.
- Tests – Contains the actual tests that are to be executed on the Implementation under test.
- TestCommunication – The interface towards the Implementation under test. Responsible for the sending of tests and receiving of test results between the TSC and the Implementation under test.

There is practically only one overall workflow for this system. This workflow is where the Test engineer chooses a test to be carried out, and then the system executes this test and then logs and views the results to the Test engineer. This workflow is shown in figure 24.

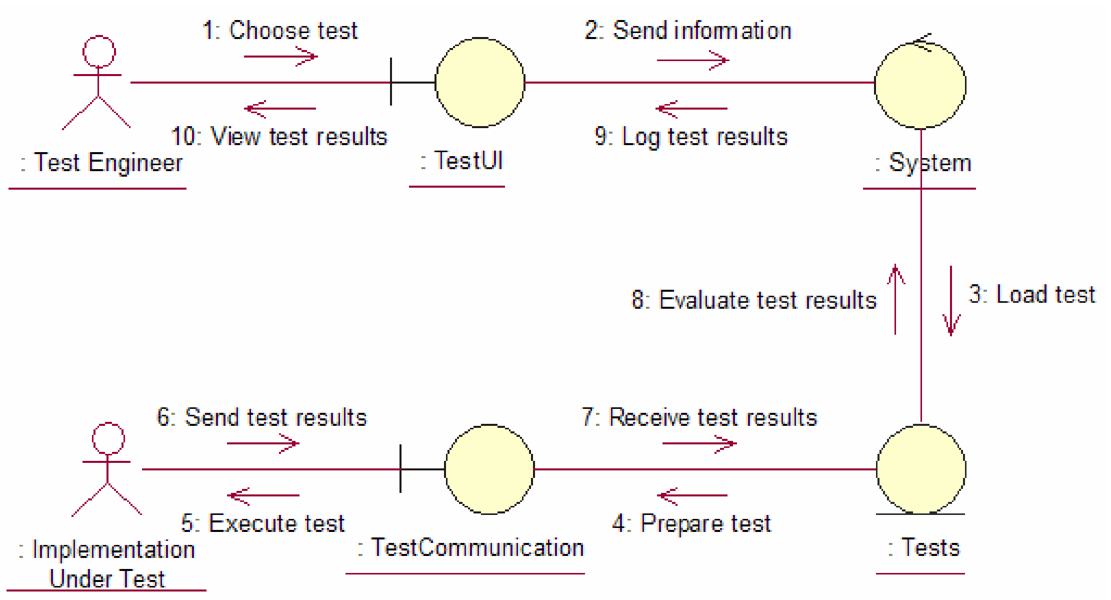


**Figure 24 - Collaboration diagram for standard workflow**

The Test engineer chooses which test to execute through the TestUI class (1). The TestUI class informs the System class that a test is chosen (2) and the System class loads the chosen test from the Tests class (3) The Tests class then prepares the test to be executed (4) before it asks the TestCommunication class to execute the test on the Implementation Under Test (5). The TestCommunication class sends the test to the IUT for execution (5) and receives the results from the IUT (6) when the test is finished. The test results are handed to the Tests class (7) for evaluation (8). The System class logs the results (9) for later use, and lets the TestUI view the results to the Test engineer (10).

## 4.3.3 Test Module

This section describes the fundamentals of the design and implementation of the Test Module.

### 4.3.3.1 Introduction

The Test Module corresponds to the Tests analysis class. The module's task is to represent the actual tests the TSC provides. The main challenge of the TSC is how the actual test shall be designed and implemented.

As described in section 2.2.11, there are three important challenges of software automation: How to feed the IUT with test data, how to capture the IUT's output and how to evaluate the output. The system has to solve all three challenges.

In section 2.2.11 three techniques for feeding test data are described, data files, batch files and keyboard capture and return. The last technique have not been found to be suitable for this system because it would need a separate capture/replay program and we think it is better to define the tests beforehand instead of recording a manual test first. The two other techniques have formed a basis for our approach, although none of them have been used in their pure form. For data input there are no separate data files. Instead the input data is defined in the test procedure itself together with the functionality that will execute the test. This approach may look similar to the batch file approach, but it is not quite the same as our test procedure not will run the IUT's own commands but the set of commands for test message treatment offered by the test layer.

Because of the XML-RPC communication mechanism between the TSC and the IUT, the capture of test results will be a simple method call return. The results have to be saved for further investigation and the most straightforward way to achieve this is to save all return values to file which is the same approach as described in section 2.2.11. But what may differ from the approach described in section 2.2.11 is that the TSC evaluates the output immediately when the remote call returns, and not after the output has been saved. To be able to do this the potential outputs must be known beforehand and set in the test procedure. There are generally three potential outcomes of a test: The test passed, the test failed or the test result is inconclusive, which means that something makes it impossible to determine the outcome of the test. By setting pass, fail and inconclusive verdicts in the test procedure, the test output can be evaluated right away by comparing the output against the predefined verdicts. This approach corresponds to the approach of building a library of correct outputs and doing a comparison against the actual output, as described in section 2.2.11. When the test results are written to file, the result of the evaluation will also be included. The log file is then just used as a log, and not for evaluation, since this is already done. Of course the results may be evaluated manually too, but that is outside the scope of the test system.

The approaches described here opens for a very compact solution for writing a test. Since both the test data and the evaluation verdicts must be set in the test procedure, test specifications does not have to be spread between separate data files, evaluation libraries and test procedures, which makes the task of writing a test more surveyable. That in turn will hopefully make the writing task easier, which is a main concern for FAT.

The next sections will describe the class design and implementation details of the TSC.

#### 4.3.3.2 Class Design

The design classes realizing the Tests analysis class are constructed as a hierarchy. There is a lot of overlapping functionality between the tests. Thus a hierarchy of classes is created where the lower classes share the functionality of the higher classes. This will make TSC smaller and more perspicuous, and it will probably be easier to design and implement new tests.

There are five main test groups. These five groups correspond to the five main testing techniques this system is supposed to provide, namely Conformance, Fault Tolerance, Reliability, Performance and Interoperability testing.

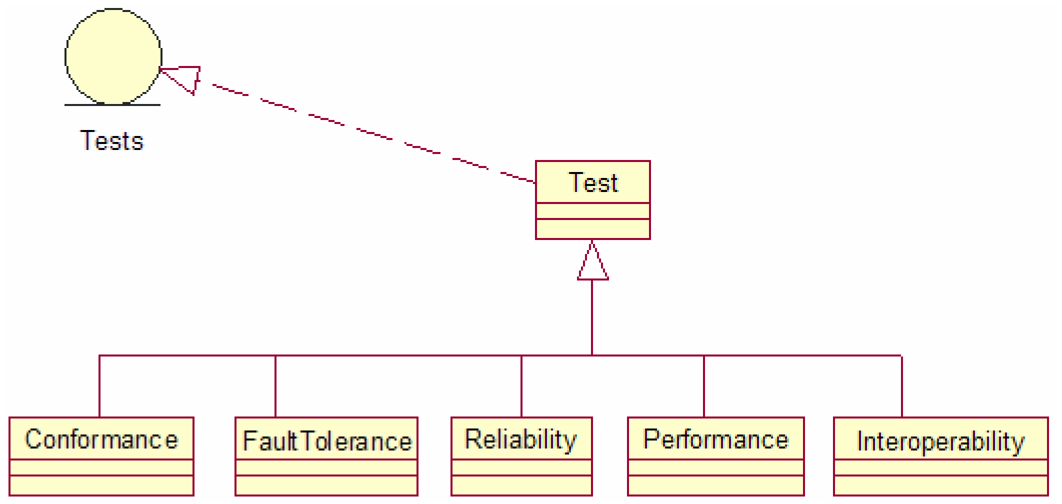


**Figure 25 - Design class realization of the Tests analysis class**

Figure 25 shows the hierarchy of the design classes realized from the Tests analysis class. Common properties for the test classes are:

- A test ID.
- A test procedure.
- A pass verdict, a fail verdict and an inconclusive verdict.
- A brief test description (optional)

A test is represented as an object of its corresponding class. All the properties of the test are encapsulated in the object. The test procedures will be implemented in the five subclasses.

The Test classes use the functionality from the MyXmlRpcClient class to communicate with the IUT. In addition to this the class offers functionality to and cooperates with the Control class.

#### 4.3.3.3 Implementation Details

Because of Java's object oriented nature and class organization, the design classes have acted as a blueprint for the implementation classes for the TSC.

The test classes are implemented with the same hierarchical structure as described in the Design section. The challenge of this system is how a test should be represented in the code. Since Java is the chosen language, it is natural to make use of the object oriented nature of the language. Hence the tests will be represented as objects. Each test is supposed to be represented as a single object of the class it naturally belongs to, e.g. a conformance test will be instantiated as an object of the Conformance class. As stated in section 4.3.3.1, the design of the test specification opens for a compact solution for writing the test procedure. This property has lead to a compact implementation solution. Each test procedure is represented in a single method. The advantage with such an approach is that a tester does not have to modify any other code than what he or she writes in the test method. The return value from the test method is a string containing the test result including the evaluation of the test. The method name shall be identical to the ID of the test. This is because the tests that are implemented are dynamically instantiated as objects at start up time. To achieve this, the system instantiates one object for each method it finds in the test classes. During the instantiation the object's test ID is set to the method name.

Each test class implements a Vector which contains all the instantiated test objects of the class. It is populated at start up time when the objects are instantiated. Since the Vector is static, it can be accessed by all the other classes. Placing the test objects in lists is done because it provides easy access to the objects for classes that need access to them, e.g. the GUI module, which needs to know the test ID and description for each test.

To show how the TSC may be used to write and execute tests, a few tests have been implemented. The implementation of these tests can be found in the appendix under the file "Test.java". Each test uses some of the functionality offered by the test layer. Together, the tests verify all of the test layer's functionality and hence act as a proof-of-concept to the framework. Some of the tests are evaluated (passed, failed or inconclusive) to show that evaluation is possible in tests such as these.

Five conformance tests, two fault tolerance tests and one performance test are implemented. We have chosen to focus on those three categories of tests and not on reliability tests and interoperability tests. The reason for this focus is that we feel it is most important to verify that the methods of the test layer works as they should, and not to implement many different categories of tests. Those three categories are also the test types that were most wanted by ObexCode in the first place. Thus we have used conformance, fault tolerance and performance tests to verify the test layer's functionality.

The three conformance tests, TP_HCI_BV_10_C, TP_HCI_BV_14_C and TP_HCI_BV_15_C test the `New` method, the `Put` method and the `Copy` method, respectively. TP_HCI_BV_10_C calls the test layer's `New` method and gets a handle in return. TP_HCI_BV_14_C inserts a HCI `Connect` message to the HCI layer using `Put`. TP_HCI_BV_15_C copies a message, using `Copy`, which has been inserted to the stack using `Put`. The two fault tolerance tests TP_HCI_BV_02_FT and TP_HCI_BV_04_FT test the `Get` and `Delete` methods of the test layer. TP_HCI_BV_02_FT uses `Get` to take a message out of the stack when it passes the test layer and then uses `Put` to insert the same message back to the test layer for further processing. TP_HCI_BV_04_FT uses `Delete` to remove a message from the stack. The five tests described verify the API of the test layer. In addition to these tests there is a simple performance test, TP_HCI_BV_06_P, which measures the time to do a HCI `Connect` using `Put`. Finally, to show that the TSC can be used to interface with the exported methods of the stack directly and not just through the test layer, two conformance tests TP_HCI_BV_22_C and TP_HCI_BV_23_C are implemented. These tests make calls to `InquiryReq` and `ConnectReq` which is exported methods of the HCI layer.

The superclass implements the functionality that is shared among all the subclasses. In addition to this, it is responsible for the instantiation of the test objects. Each of the subclasses may also implement functionality that is special for this class, e.g. the Performance class may implement timing methods that are used to measure the performance of the system. Such helper methods have to be marked in some way, e.g. with a prefix, so that they can be identified as helper methods and not test methods when the system instantiates the test objects.

The only information a tester needs to have of the TSC is the format/API of the test objects and the helper methods available. If the tester has this knowledge he or she can start writing tests.

#### 4.3.3.4 Summary

This section has described the Test Module which is a realization of the Tests analysis class. The module is responsible for containing the actual tests that the TSC shall execute. Because of the compact test procedure design, a test procedure is implemented as a single method. This is done to provide simplicity when writing tests.

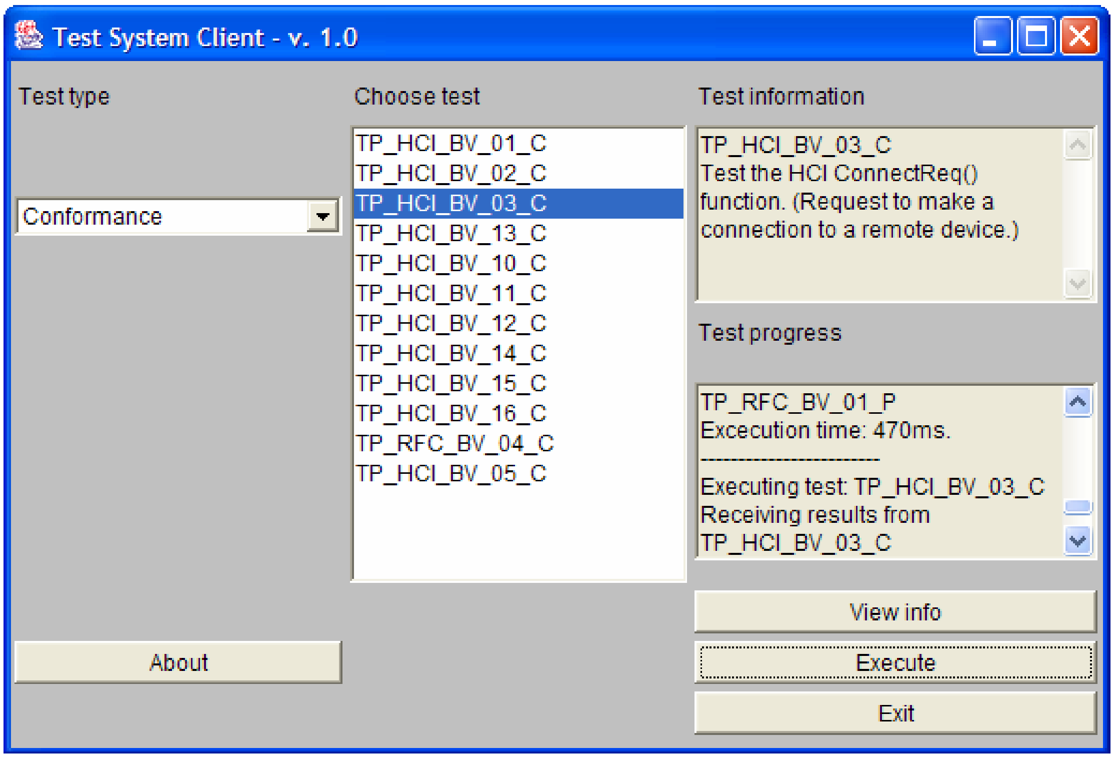
### 4.3.4 GUI Module

This section describes the fundamentals of the design and implementation of the GUI Module.

#### 4.3.4.1 Introduction

The GUI Module corresponds to the TestUI analysis class. The GUI Module is responsible for giving the tester an interface towards the TSC. This corresponds to the requirements of giving the tester the opportunity to execute a test and to view the test results, as discussed in section 4.3.1.

The interface towards the user is a graphical user interface. The alternative was a textual or command line interface. The graphical interface was chosen because it will provide better usability for the tester. The window that will appear when the test program is started will look something like figure 26.



TP_HCI_BV_01_C
TP_HCI_BV_02_C
TP_HCI_BV_03_C
TP_HCI_BV_13_C
TP_HCI_BV_10_C
TP_HCI_BV_11_C
TP_HCI_BV_12_C
TP_HCI_BV_14_C
TP_HCI_BV_15_C
TP_HCI_BV_16_C
TP_RFC_BV_04_C
TP_HCI_BV_05_C

TP_HCI_BV_03_C
Test the HCI ConnectReq() function. (Request to make a connection to a remote device.)

TP_RFC_BV_01_P
Excecution time: 470ms.
-------------------------
Executing test: TP_HCI_BV_03_C
Receiving results from
TP_HCI_BV_03_C

**Figure 26 - Example screenshot of the GUI of the test system**

The window contains buttons for execution of tests and for exiting the program. It also contains a button for viewing information about chosen tests. There are also two text areas, one for viewing test information and one for viewing the test progress and results. A choice list of available test types is presented together with a list of actual tests. The list shows the implemented tests of the test type chosen in the choice list. It is also possible to select more than one test for execution from the test list. This is a useful feature, since a tester this way can execute many tests without having to start each test manually.

### 4.3.4.2 Class Design

The analysis class TestUI is realized through the design classes MainGUI and Listener. Figure 27 shows this realization and the relationship between the classes.

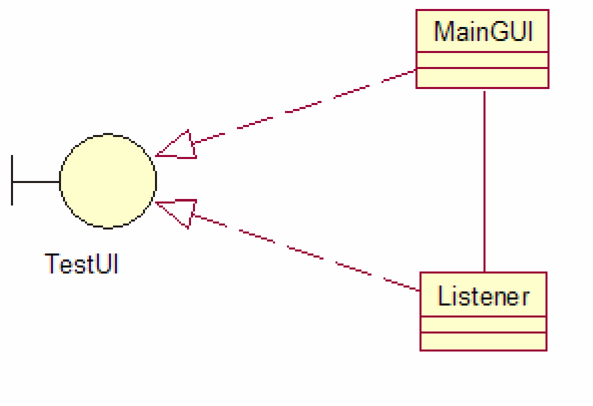


**Figure 27 - Design class realization of the TestUI analysis class**

The TestUI class is the interface towards the Test Engineer and it provides the communication between the engineer and the system. The MainGUI class shall provide the graphical user interface that the Test Engineer uses to communicate to the system. The MainGUI class must offer the possibility to display the tests that may be executed and offer functions to execute the tests. In addition the test results shall be displayed.

The Listener class is the class that listens to and reacts to events from the MainGUI class. An event is a button click or a change in a list of a graphical user interface. An event is supposed to trigger some action to the system. The actions that the Listener class has responsibility to trigger are:

- execute a test
- view available tests
- view test information
- view test progress
- to close the program

Except for the closing of the program, these events will have to make the class contact other parts of the system, since this functionality lies within other classes. Thus the Listener class communicates with the rest of the system, which in this case is the Control class, and the MainGUI class communicates only with the Listener class.

#### 4.3.4.3 Implementation Details

The GUI is implemented in one class, *MainGUI*. The class is implemented using the built-in java.awt toolkit. The listener class is implemented in the class *Listener*. The class takes care of all events in the GUI, both the action events and the item events using the built-in interfaces *ActionListener* and *ItemListener*. The *Listener* class communicates with the *Control* class for execution of the chosen tests. The *Listener* class uses the test Vectors described in section 4.3.3 to access the test objects. This is necessary for populating the GUI’s test list and for viewing the test information.

#### 4.3.4.4 Summary

This section has described the design and implementation of the GUI module. The module provides the tester with a graphical user interface where it is possible to choose tests for execution. The ability to choose more than one test at a time for execution is important since it means that a tester can execute many tests after another without having to start each test manually.

### 4.3.5 Control Module

This section describes the fundamentals of the design and implementation of the Control Module.

#### 4.3.5.1 Introduction

The Control Module corresponds to the System analysis class. The main task for the Control module is to identify which tests have been chosen for execution and to start executing these tests. Additionally the module is responsible for the logging of test results.

As discussed in section 4.3.3.1 the test results must be captured somewhere. The solution in TSC is to write the test results to file. This opens for some interesting design choices. One subject is how the test result should be presented. As described in section 4.3.3.3 the test methods return a string with the result of the test, including the evaluation of the result. This is basically everything we need to know about the test. Hence the string that is returned from the test method is captured in the Control Module and written to file together with the ID of the test. The ID is included of the obvious reason that we shall be able to know which test gave these results. The time the test finished is also included in the file just for convenience.

Here is an example of a test result file:

```
TP_HCI_BV_16_C
Mon Dec 01 13:34:43 CET 2003

Result:
0
The test is passed.
```

Another issue is to determine if several test results should be written to the same file, or if each test is assigned a separate file. The solution for the TSC is one file per test. The reason for this approach is that it might be more surveyable than having all the test results in one file. One could consider a few test results per file but this might not be any more surveyable than the other approaches. The implementation section says more about the file naming of the log files.

The following sections deal with the class design and implementation of the Control Module.

### 4.3.5.2 Class Design

The Control class is the design class realization of the System analysis class. The realization is shown in Figure 28.

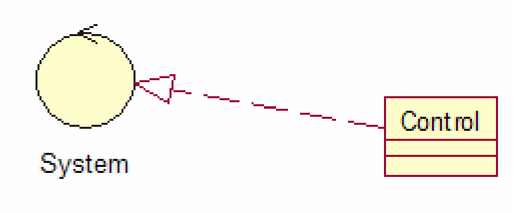


**Figure 28 - Design class realization of the System analysis class**

The responsibility of the Control class is to be the link between the GUI part and the Test part of the system. In more detail, the following are the responsibilities of the class:

- create a log file where test results can be logged
- identify which test object is chosen
- tell the Test class to execute the chosen test
- write test results to file and send the results to the GUI

The Control class offers functionality to and cooperates with the Test classes and the Listener class.

#### 4.3.5.3 Implementation Details

The class *Control* implements methods that coordinate the events from the GUI Module and the Test Module.

The logging is implemented in this class. As discussed in section 4.3.5.1 each test result is logged in a separate file. To separate the log files from each other, each file is named with the test ID of the test the results in the file originates from, which might cause a problem when there is more than one execution per test. To solve the problem, a timestamp value is added to the filename. The format of the filename is as follows:

```
results_<testID>_<timestamp>.txt
```

#### 4.3.5.4 Summary

This section has described the design and implementation of the Control Module. The responsibility of the module is to be the link between the GUI Module and the Test Module. In addition it is responsible for the logging of test results. These results are written to file where each test is written into a separate file.

### 4.3.6 XML-RPC Module

This section describes the fundamentals of the design and implementation of the XML-RPC Module.

#### 4.3.6.1 Introduction

In section 4.2.3 the test layer's XML-RPC module is described. This module is the module that receives the requests, or remote calls from the test system. The TSC XML-RPC Module is the client side of the communication mechanism.

The responsibility of the XML-RPC module is to provide the Test Module with functionality to contact the IUT. The next sections describe the details of the design and implementation of the Module.

#### 4.3.6.2 Class Design

The MyXmlRpcClient class is the design class realization of the TestCommunication analysis class. The realization is shown in Figure 29.

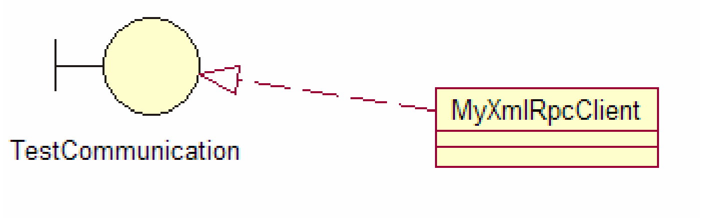


**Figure 29 - Design class realization of the TestCommunication analysis class**

The responsibility of this class is to offer functionality to make XML-RPC calls to an XML-RPC server. An object of this class shall represent an XML-RPC client.

The functionality offered by the MyXmlRpcClient class is used by the Test classes. This is for communication with the IUT.

#### 4.3.6.3 Implementation Details

The XML-RPC client that communicates with the IUT is implemented here. To implement this client, an implementation of XML-RPC from Web Services Project @ Apache [Apache] is used. The package *org.apache.xmlrpc* provides the functionality needed to implement a Java XML-RPC client. An object of the MyXmlRpcClient class is instantiated with the URL of the XML-RPC server as parameter. The class contains one method, which executes the remote call. The parameters for this method are the remote function name represented as a string and the remote function's parameters represented as a Vector. The return value is a string with the actual return value of the remote call unless an exception has been thrown. In the latter case the return value is a string representation of the exception.

#### 4.3.6.4 Summary

This section has described the design and implementation of the XML-RPC module. The responsibility of the module is to provide the communication mechanism to the IUT for the TSC.

### 4.3.7 Miscellaneous

There is one more class in the TSC. This class is called Start. The only task of the Start class is to start the TSC by running the main method. In this method the Test objects are initialized and the GUI is started. When the Test objects are initialized, and the GUI is started, the program is ready for use.

### 4.3.8 Summary

This section has described the test system client (TSC). The main task of the TSC is to specify a test, execute a test, and log and view the test result. The TSC is divided into four modules where the Test module is the part where the tests are written. A test is written in Java. The communication mechanism towards the IUT is XML-RPC. An XML-RPC client is therefore implemented as part of the TSC.

# Chapter 5

# Experiments

This chapter describes experiments that we have performed in order to evaluate FAT.

## 5.1 Introduction

The test layer introduces an additional layer to the stack. An interesting metric is how much latency the extra layer introduces, and what impact the TSC has on the performance of the stack. This might be important for time-critical applications that will use the stack. The results of these experiments may help a tester decide whether it is possible to use FAT when testing the stack with certain time-critical applications.

There are in particular two interesting scenarios we want to examine. These two scenarios are further described in section 5.3. Before we explain the experiments we describe the hardware and system characteristics of the platform that is used for the experiments in section 5.2.

## 5.2 Test Platform Characteristics

When performing the experiments, two computers were used. One was used to run the Bluetooth stack with an inserted test layer and the other was used to run the TSC. The following are the characteristics of the computer running the stack:

- Processor: AMD Athlon XP 2500 +, 1.8 GHz
- Memory: 1 GB RAM
- Operating System: Linux, version 2.4.20-8

The following are the characteristics of the computer running the TSC:

- Processor: Mobile AMD Athlon XP 2400 +, 1.8 GHz
- Memory: 512 MB RAM
- Operating System: Windows XP Home Edition, version 2002

The two computers were connected to the same 100 Mbit LAN switch.

The experiments also include a Bluetooth device, in this case a mobile telephone. This phone is used to establish a connection to.

## 5.3 Experiments

Two experiments were performed. The first experiment examined the delay introduced by the test layer when forwarding a message. The other experiment examined the delay introduced when a message is sent to the TSC for modification.

### 5.3.1 Delay introduced when forwarding a message

In a situation where the test layer receives a message which it is not supposed to do anything with, i.e. it is in Normal mode, the layer forwards the message to the next layer. We would like to know the extra time the message uses when the test layer just forwards the message to the next layer in the stack.

The experiment was performed by measuring the time it takes from the message is sent from the upper layer and until it is being sent from the test layer. The situation is shown in figure 30.

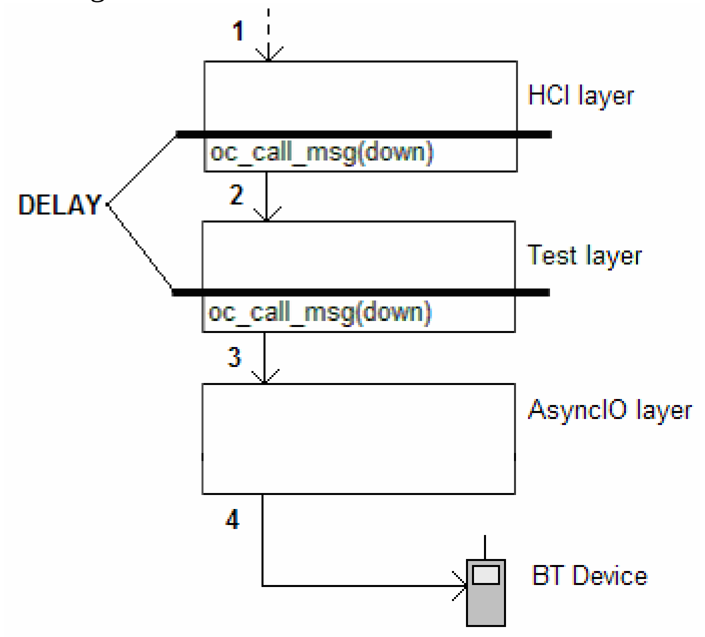


**Figure 30 – Measuring delay introduced when forwarding a message**

To measure the time, the C-function `gettimeofday()` was used. This function returns the number of seconds and microseconds since the Epoch. The message sent through the stack was a HCI-Connect message. It was inserted into the HCI layer using FAT's `Put` method. The upper layer of figure 30 is as such the HCI layer. The experiment was performed 30 times to get a reliable average value.

### 5.3.2 Delay introduced when modifying a message

The second experiment measures the delay introduced by both the test layer and the TSC when a message is taken out of the stack and sent to the TSC for modification, before inserting it back to the test layer. The message data is not actually modified in the TSC, it is just bounced back to the test layer.

The experiment is performed in a similar fashion as the above experiment. It is performed by measuring the time it takes from the message is sent from the upper layer to when it is being sent from the test layer. This experiment is shown in figure 31.

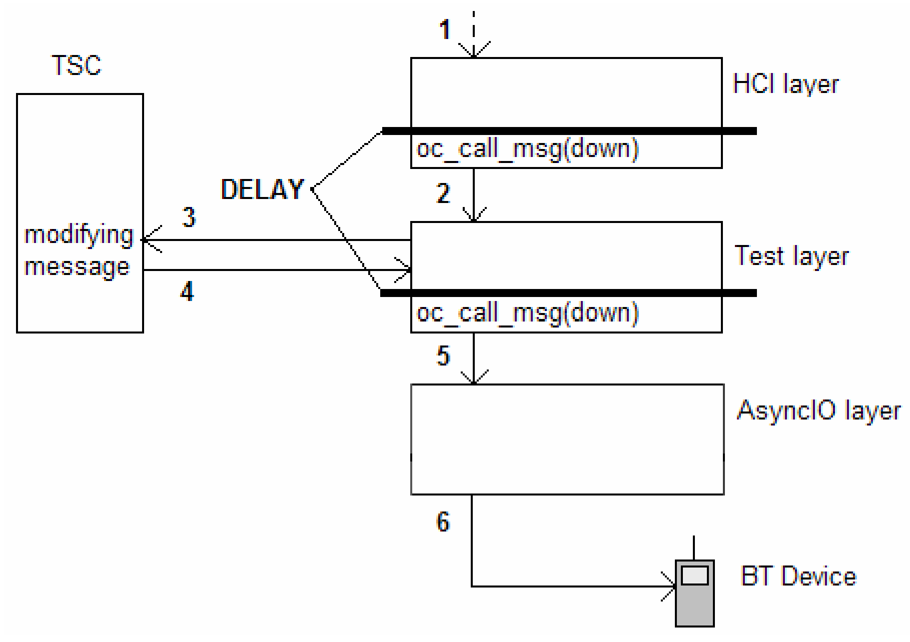


**Figure 31 - Measuring delay introduced when modifying a message**

The experiment is performed using the same methodology as described in section 5.3.1.

## 5.4 Results

For comparison, we measured the time a HCI-Connect message uses through the HCI layer. These experiments were performed using the same methodology as the other experiments. The result of the tests is presented in figure 32. All measurements are in microseconds. The figure shows the average values of the experiments together with the standard deviation of the results.

| | *Average* | *Stand. dev.* |
|---|---:|---:|
| Forwarding | 68.8 | 0.5 |
| Modifying | 113 140 | 20 524 |
| Through HCI | 277 | 6.3 |

**Figure 32 - Experiments results**

We can conclude that for forwarding, the test layer introduces overhead in the order of 25% of the time the HCI layer uses to process a Connect request. This delay is acceptable if we assume that the HCI layer is representative for all layers in the stack. If that is the case, we may insert test layers between all layers in the stack and get a total overhead that is in the order of 25 % of the time it takes to send one message through the stack. Most of the delay is probably due to the extra call to `oc_call_msg()`. Such a call takes about the same time as the measured time for forwarding. Thus the delay is really just an extra function call, which implies that the test layer therefore should not have any significant impact on the performance of the stack. The implemented HCI layer is also a thin layer, i.e. it does not do much. When other layers are implemented, they will probably do more than this HCI layer, which will make the order of the delay of the test layer less.

Compared to forwarding, the overhead introduced when modifying messages is significantly higher. It takes about 400 times longer to send the message to the TSC for modification and back to the test layer, than the HCI layer uses to process it. A higher delay is however expected, since the experiment involves network traffic, and not just internal processing. The high delay might not however reduce the bandwidth of the stack. The stack, including the sending to and from the TSC, will act as a pipeline that may have the same capacity as the stack alone. This will, however depend on the amount of processing overhead when sending and receiving messages to the TSC, and the amount of data sent from the Bluetooth Application. An experiment to verify this statement can only be done on a stack that is fully implemented, hence it is not evaluated further in this thesis.

An interesting curiosity is that it was a significant difference between measurements made when we measured the delay of the first message sent after start up of the stack, and when we already had sent a message through the stack. This extra delay was however removed when we let the process sleep for five seconds before we started measuring the time. The results were now the same as the results when messages already had been sent through the stack. The delay introduced could be due to preemption interrupts. By letting the process sleep just before starting the experiment, a full timeslot is available when starting the execution of code involved in the experiment.

## 5.5 Summary

The experiments performed on FAT investigate the delay the test layer and the TSC introduce to the stack. Two experiments were performed. The first experiment measures the delay introduced when forwarding a message through the test layer. The second experiment measures the delay introduced when sending a message to the TSC for modification, before it is returned to the stack. The results show that the delay introduced when forwarding a message is about 25 % of the time the HCI layer uses to process a message. The delay introduced when modifying a message is about 400 times the time the HCI layer uses to process a message.

# Chapter 6

# Discussion and Conclusion

This chapter summarizes FAT. It also evaluates the work done, shows aspects of future work and concludes the thesis.

## 6.1 Summary of the Thesis

This thesis has described FAT, a framework for automated regression testing of protocol stacks.

FAT is designed to be used to test protocol stacks developed within the ObexCode network protocol development framework (NPDF). A Bluetooth stack, which is under development, is used as the basis for the development of the framework.

FAT makes use of the ability to stitch test layers between the NPDF layers. Such a test layer can insert messages to the stack, or modify, copy or delete the messages passing through it. The test layer offers an API with these four functions. The API is used by a client called the Test System Client (TSC). In the TSC the tests are written. A test procedure is written as a single Java method. The TSC offers a GUI to the tester. This GUI makes it possible to choose among the implemented tests and to execute them on the stack. The communication mechanism between the test layers and the TSC is XML-RPC. That is why an XML-RPC module is connected to a test layer. The module realizes the API offered by the test layer with four methods that are exported, for use by the TSC. These methods are `Put`, `Get`, `Copy` and `Delete`, corresponding to inserting, modifying, copying and deleting messages.

The test layer does not provide much functionality because it is desirable to keep it as thin as possible. The reason for this approach is that a Bluetooth stack is often executed on devices with a small amount of resources, such as memory and processing power. In other words: We want to put as little extra load on the stack as possible. The functionality offered by the test layer should however be enough to write many useful tests.

Five conformance tests, two fault tolerance tests and one performance test are implemented. These tests are implemented to show that the functionality of the test layer can be used to write tests. Together the tests verify the correctness of the test layer's API.

The experiments show that the test layer introduces low overhead when forwarding messages. When a message is modified however, the delay is significant. But this delay might not have much to say for the bandwidth of the stack, as it probably will act as a pipeline when several messages pass through it.

## 6.2 Evaluation

The problem definition in chapter 1 state that the purpose of this project is to construct a framework for automatic regression testing of protocol stacks developed within the ObexCode network protocol development framework (NPDF). In this section we evaluate the goals of the project.

In chapter 2 we first described the Bluetooth technology. Then we motivated the need for testing, before we described some of the important testing methodologies, including test automation. Existing systems and frameworks for test automation concluded the chapter. This background laid the foundation for our own work.

Chapter 3 described the architecture of the framework. The architecture realized the goals described in section 1.2. It laid the foundation for the design and implementation of the framework

Chapter 4 described the design and implementation of the framework. One goal from the problem definition was that the test environment shall be implemented in a prototype, which will interface with the stack under test through a defined API. This API was implemented as four methods in a test layer. The test layer is a layer that may be stitched in-between the layers in a NPDF stack. The test layer provides methods for inserting, modifying, copying and deleting messages. The same goal also proposes that the prototype shall be able to execute on a different processing node than the stack itself. Because of this goal we designed and implemented a Test System Client (TSC) where the actual tests are specified. The TSC communicates with the test layer through XML-RPC, which makes it possible to run it on another node than the stack.

Another goal from the problem definition was that the computation and memory footprint of the framework on the stack under test should be kept at a minimum. This goal was realized by adding just the absolute necessary functionality to the test layer. The four methods of the test layer are simple, but provide the tester with powerful tools for testing a stack.

The last goal, which stated that the framework shall be constructed to interface to an NPDF Bluetooth stack, was realized using a Bluetooth stack that is under development by ObexCode. Both the test layer and the TSC worked as expected together with this stack.

All the goals described in section 1.2 are fulfilled. In addition to the realization of the goals, we believe that FAT may be used to test other stacks developed within the NPDF. The test layer is very simple and general, and does not depend much on the particular stack it has been designed for. FAT is, due to time limits, not tested against other stacks, but we believe only minor changes to the test layer are necessary to use it.

## 6.3 Discussion and Future Work

Although FAT fulfils all the goals of the project, there are still aspects of the framework that can be improved.

In section 4.2.2 several possible additions to the test layer's functionality were discussed. These additions include evaluation of messages based on content and event codes, multiple modes, code downloads and packet filters. They were all rejected due to our goal of placing as little extra load on the stack as possible. But if we allow a "thicker" test layer, many of these approaches are very interesting. For further details on these approaches, see section 4.2.2.

Our framework makes it possible to do simple performance testing. This is limited to time measurements from the TSC. If we insert more functionality to the test layer, we could improve the possible performance tests. This functionality includes timestamps of messages passing the test layer, or different statistics used for throughput testing, like message counters etc. How to deal with statistics in the test layer is further discussed in section 4.2.2.

On the client side there is also room for improvement. Parts of the code used to write tests are common for many of the tests. This code should be placed in a library of helper methods, which will make the test writing task both faster and easier. In the current implementation only a few such methods exists. The lack of helper methods is partly due to the fact that only a few tests have been implemented, and therefore the need for many such methods have not been discovered yet. As more tests are implemented, more possible helper methods are discovered and needed.

FAT is currently designed and implemented for stacks developed within the NPDF. This approach makes it difficult to use FAT to test stacks that do not use NPDF, without modifications. An alternative to FAT would be a framework that is completely generic. Such a framework would above all need to specify a packet format that can be used to investigate packets in the stack. The fact that different stacks operate with different packet formats is one of the major challenges of such a framework. An own programming language could be developed for writing of tests. This language should have special features designed specially for writing tests. The ideal case would be if one could define an abstract specification of the correct behaviour of a stack. This specification could be loaded by the framework which could check the stack's behaviour towards the specification. Such an approach is described in [Rodrigues et. al., 2001]. The complexity of such an approach is discouraging, but the paper provides a proof of concept for a realization.

## 6.4 Conclusion

We have designed and implemented FAT, a framework for automated regression testing of protocol stacks developed within the ObexCode network protocol development framework (NPDF). The framework enables a tester to write tests that can be executed on a Bluetooth stack. The test results are evaluated and logged on the framework's client. Example test implementations show that it is possible to perform conformance, fault tolerance and performance testing using FAT. The framework has been evaluated against a Bluetooth stack, but it should be possible to use the framework on other stacks developed within the NPDF.

# Chapter 7

# References

[Apache]
Web Services @ Apache homepage – http://ws.apache.org.

[Arlat et. al., 1991]
Arlat, J., Crouzet, Y. and Laprie J-C. Fault Injection for the Experimental Validation of Fault Tolerance. In *Proceedings of the Annual Esprit Conference (Esprit'91),* Brussels, 1991, pp. 791-805.

[Beck]
Beck, K. *Simple Smalltalk Testing: With Patterns*. First Class Software, Inc.

[Beck and Gamma]
Beck, K and Gamma, E. JUnit Cookbook.
http://junit.sourceforge.net/doc/cookbook/cookbook.htm.

[Bershad et. al., 1995]
Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M. E., Becker, D., Chambers, C. and Eggers S. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating System Principles.* 1995.

[BlueTester]
IVT BlueTester. International Validation & Testing Corporation.
http://www.ivtcorporation.com/products/tester/newtester.htm

[Bluetooth SIG, 1999]
Bluetooth Special Interest Group, *Specification of the Bluetooth System*, version 1.0B, volumes 1 and 2. December 1999.

[Boehm, 1987]
Boehm, B. W. Improving Software Productivity, *IEEE Computer*, Vol. 20, No. 9, Sep. 1987, pp. 43-57.

[Browne, 1976]
Browne, J. C. A Critical Overview of Computer Performance Evaluation. In *Proceedings of the 2nd International Conference on Software Engineering*. October 1976.

[Chen et al., 1994]
Chen, Y. F., Rosenblum, D. S. and Vo, K. P. TestTube: A system for Selective Regression Testing. In *Proceedings of the 16th International Conference on Software Engineering*, May 1994, pp. 211-222.

[Denning et al., 1989]
Denning P. J. (Chairman). Computing as a Dicipline. *Communications of the ACM*, vol. 32, no 1, January 1989, pp. 9-23.

[Dideles, 2003]
Dideles, M. Bluetooth: A Technical Overview. *Crossroads – The ACM student magazine, Networking*, Issue 9.4, Summer 2003, pp. 11-17.

[Dijkstra et al., 1972]
Dijkstra, E. W., Dahl, O. J. and Hoare, C. A. R. *Structured Programming*. Academic Press, London and New York, 1972.

[Engels et al., 1997]
Engels, A., Feijs, L. and Mauw, S. Test Generation for Intelligent Networks Using Model Checking. In *Proceedings of the third international workshop on tools and algorithms for the construction and analysis of systems (TACAS' 97),* Springer-Verlag, April 1997, vol. 1217 of Lecture Notes in Computer Science, pp. 384-398.

[ETS, 1995]
ETS (European Telecommunication Standard). *Methods for Testing and Specification (MTS); Protocol and profile conformance testing specifications; Standardization methodology*. European Telecommunication Standards Institute, April 1995.

[Fischer and Chin, 2003]
Fischer, K and Chin, J. Bluetooth Conformance and Interoperability Testing. *Conformity*. February 2003, pp. 28-33.

[Goodenough and Gerhart, 1975]
Goodenough, J. B and Gerhart, S. L. Toward a Theory of Test Data Selection. In *Proceedings of the International Conference on Reliable Software*, 1975, pp. 493-510.

[Graney, 2000]
Graney, M. TTCN, Protocol Testing on Steroids! *IEEE P802.15 Working Group for Wireless Personal Area Networks (WPANs),* doc.: IEEE 802.15-00/063r0. March 2000.

[Gunneflo et. al., 1989]
Gunneflo, U., Karlsson, J. and Torin J. Evaluation of Error Detection Schemes Using Fault Injection by Heavy-Ion Radiation. *Proceedings of the 19th International Symposium on Fault-Tolerant Computing (FTCS-19)*, Chicago, 1989, pp. 340-347.

[Harrold, 2000]
Harrold, M. J. Testing – A Roadmap. In *Future of Software Engineering. 22nd International Conference on Software Engineering*. June 2000.

[Hitti and Joslin, 1965]
Hitti, R. F. and Joslin, E. O. Application Benchmarks: The Key to Meaningful Computer Evaluations. In *Proceedings of the 1965 20th National Conference*. August 1965.

[Huslende, 1981]
Huslende, R. A Combined Evaluation of Performance and Reliability for Degradable Systems. In *Proceedings of the 1981 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. September, 1981.

[IEC, 2003]
IEC: *Tree and Tabular Combined Notation*. International Engineering Consortium, 2003.

[ISO/IEC 8824, 1990]
ISO/IEC 8824. *Abstract Syntax Notation One (ASN.1)*, 1990.

[ISO/IEC 9646-3, 1998]
ISO/IEC 9646-3. *Information technology – Open Systems Interconnection – Conformance testing methodology and framework – Part 3: The Tree and Tabular Combined Notation (TTCN)*, 1998.

[Java]
Java homepage -http://www.java.sun.com.

[JUnit]
JUnit homepage - http://www.junit.org.

[Kaner et al., 1993]
Kaner, C., Falk, J. and Nguyen, H. Q. *Testing Computer Software,* Second Edition. Van Nostrand Reinhold, 1993.

[Kindrick et al. 1996]
Kindrick, J. D., Sauter, J. A. and Matthews, R. S. Improving Conformance and Interoperability Testing. *StandardView*. Vol. 4, No 1, March 1996, pp. 61-68. ACM Press.

[Make, 2002]
GNU Make Documentation, make version 3.80, July 2002.

[Marsaglia and Zaman, 1993]
Marsaglia, G. and Zaman, A. Monkey Tests for Random Number Generators. *Computers and Mathematics with Applications,* 1993, 26(9), pp. 1-10.

[McCanne and Jacobson, 1993]
McCanne, S. and Jacobson, V. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the 1993 Winter USENIX Technical Conference,* San Diego, California, January 1993,

[Miller and Bisdikian, 2001]
Miller, B. A and Bisdikian, C. *Bluetooth Revealed*, Prentice Hall, 2001.

[Myers, 1979]
Myers, G. J. *The Art of Software Testing*. Wiley-Interscience, 1979.

[Oppenheimer and Welsh, 1997]
Oppenheimer, D. and Welsh M. User Customization of Virtual Network Interfaces with U-Net/SLE. *Technical Report CSD-98-995*, University of California, Berkeley, 1997.

[Patton, 2001]
Patton, R. *Software Testing*. Sams Publishing, 2001.

[PRD, 2002]
*Bluetooth Qualification Program Reference Document* v. 1.0, document no. 8.B.124/1.0. February 2002.

[Ramamoorthy and Ho, 1975]
Ramamoorthy, C. V. and Ho, S. F. Testing Large Software With Automated Software Evaluation Systems. In *Proceedings of the International Conference on Reliable Software*. 1975.

[Rodrigues et. al., 2001]
Rodrigues, R., Castro, M. and Liskov, B. BASE: Using Abstraction to Improve Fault Tolerance. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, October 2001, pp. 15-28.

[Rothermel and Harrold, 1996]
Rothermel, G. and Harrold, M. J. Analyzing Regression Test Selection Techniques. *IEEE Transactions on Software Engineering*, v 22, n 8, August 1996, pp. 146-156.

[Rothermel and Harrold, 1997]
Rothermel, G. and Harrold, M. J. A Safe, Efficient Regression Test Selection Technique. *ACM Transactions on Software Engineering and Methodology*, Vol. 6, No. 2, April 1997, pp. 173-210.

[Sarikaya et. al., 1986]
Sarikaya, B., Bochmann, G., Maksud, M. and Serre, J. Formal Specification Based Conformance Testing. In *Proceedings of the ACM SIGCOMM Conference on Communications Architectures & Protocols*, 1986, pp. 236-240.

[SourceForge]
SourceForge homepage: http://sourceforge.net/

[Telelogic, 2001]
*Telelogic Tau 4.2 TTCN Suite Methodology Guidelines*, chapter 1, Telelogic AB, March 2001.

[UML, 2003]
UML – *Unified Modelling Language Specification v. 1.5 (formal/03-03-01)*. Jan. 2003.

[Vokolos and Weyuker, 1998]

Vokolos, F. I. and Weyuker, E. J. Performance Testing of Sofware Systems. In *Proceedings of the First International Workshop on Software and Performance*, 1998, pp. 80-87.

[Watkins, 2001]
Watkins, J. *Testing IT – An Off-the-Shelf Software Testing Process*, Cambridge University Press, 2001.

[Welsh et. al. 1998]
Welsh, M., Oppenheimer, D. and Culler, D. E. U-Net/SLE: A Java-Based User-Customizable Virtual Network Interface. In *Proceedings of Java for High-Performance Network Computing Workshop at EuroPer'98,* Southampton, UK, September 1998.

[Wolverton, 1974]
Wolverton, R. W. The Cost of Developing Large-Scale Software. *IEEE Trans. Computers*, June 1974, pp 615-636.

[XML-RPC, 1999]
XML-RPC specification. UserLand Software. June 1999.

# Appendix

The appendix lists the source code of FAT.

The following source code files are listed with page numbers.

```
/**
 * \file     oc_test.h
 * \author   Karl Magnus Nilsen <karlm@obexcode.com>
 * \date     07.12.2003
 *
 * Copyright (C) 2003, ObexCode AS, All Rights Reserved.
 */

#ifndef OC_TEST_H
#define OC_TEST_H

#include <test/oc_test.h>
#include <core/oc_core.h>

/** Test message events */
enum {
  EV_TEST_SEND_PUT_REQ              = 0x030,
  EV_TEST_RECV_PUT_COMPLETE         = 0x031,
  EV_TEST_SEND_GET_REQ              = 0x032,
  EV_TEST_RECV_GET_COMPLETE         = 0x033,
  EV_TEST_SEND_COPY_REQ             = 0x034,
  EV_TEST_RECV_COPY_COMPLETE        = 0x035,
  EV_TEST_SEND_DELETE_REQ           = 0x036,
  EV_TEST_RECV_DELETE_COMPLETE      = 0x037,
};

/** Test layer modes*/
enum {
  NORMAL_MODE = 0x00,
  GET_MODE    = 0x01,
  COPY_MODE   = 0x02,
  DELETE_MODE = 0x03,
};

/** Test layer struct*/
struct oc_test {
  OC_Layer layer;

  OC_Handle up;       /**< Handle to upper layer */
  OC_Handle down;     /**< Handle to lower layer */
  OC_Handle test;     /**< Handle to xml-rpc module */
};
typedef struct oc_test OC_Test;

/**
 *  struct OC_TestPutReq
 */
struct oc_test_put_req {
  OC_Handle up;       /**< Handle to client layer */
  char * message;     /* The message that is to be inserted to the
stack.*/
};
typedef struct oc_test_put_req OC_TestPutReq;

/**
 *  struct OC_TestGetReq
 */
struct oc_test_get_req {
  OC_Handle up;       /**< Handle to client layer */
  uint8 event;        /**< Event code for message to be fetched.*/
  int num_msg;        /**< Number of messages to be fetched.*/
};
typedef struct oc_test_get_req OC_TestGetReq;

/**
 *  struct OC_TestGetReqComplete
 */
struct oc_test_get_req_complete {
  OC_Handle up;       /**< Handle to client layer */
  uint8 event;        /**< Event code of the message */
  uint16 start;       /**< Start index of data buffer */
  uint16 len;         /**< Length of data in data buffer */
  uint8 buf[1];       /**< The data buffer */

};
typedef struct oc_test_get_req_complete OC_TestGetReqComplete;

/**
 *  struct OC_TestCopyReq
 */
struct oc_test_copy_req {
  OC_Handle up;       /**< Handle to client layer */
  uint8 event;        /**< Event code for message to be copied.*/
  int num_msg;        /**< Number of messages to be copied.*/
};
typedef struct oc_test_copy_req OC_TestCopyReq;

/**
 *  struct OC_TestCopyReqComplete
 */
struct oc_test_copy_req_complete {
  OC_Handle up;       /**< Handle to client layer */
  uint8 event;        /**< Event code of the message */
  uint16 start;       /**< Start index of data buffer */
  uint16 len;         /**< Length of data in data buffer */
  uint8 buf[1];       /**< The data buffer */
};
typedef struct oc_test_copy_req_complete OC_TestCopyReqComplete;

/**
 *  struct OC_TestDeleteReq
 */
struct oc_test_delete_req {
  OC_Handle up;       /**< Handle to client layer */
  uint8 event;        /**< Event code for message to be deleted.*/
  int num_msg;        /**< Number of messages to be deleted.*/
};
typedef struct oc_test_delete_req OC_TestDeleteReq;

/**
 *  struct OC_TestDeleteReqComplete
 */
struct oc_test_delete_req_complete {
  OC_Handle up;       /**< Handle to client layer */
  int del_status;     /**< The status of the delete operation */
};
typedef struct oc_test_delete_req_complete OC_TestDeleteReqComplete;


#define OC_TEST(ptr) ((OC_Test *) ptr)

#define oc_test_bind_up(self, above) \
      OC_TEST(self)->up = (above)
#define oc_test_bind_down(self, below) \
      OC_TEST(self)->down = (below)

/* Exported function */
OC_Handle oc_test_new(void);


#endif /* OC_TEST_H */
```

```
/**
 * \file     oc_test.c
 * \author   Karl Magnus Nilsen <karlm@obexcode.com>
 * \date     07.12.2003
 *
 * Copyright (C) 2003, ObexCode AS, All Rights Reserved.
 */

#define OC_DEBUG_LEVEL 4
#include "oc_config.h"

#include <core/oc_core.h>
#include <blue/oc_hci.h>

#include "oc_test.h"
#include "oc_test_xmlrpc.h"

/* Internal prototypes */
static void oc_test_msgproc(OC_Handle handle, OC_Msg *msg);

/* The test layer has four modes, depending on the operation to do
with the next message*/
static int mode;

/* Initializes test layer */
OC_Handle oc_test_new(void)
{
  OC_Test *self;

  self = malloc(sizeof(OC_Test));
  if (!self)
    {
      OC_ERROR(("Could not initialize test layer!"));
    }

  self->up = NULL;
  self->down = NULL;

  /* Mode initially set to NORMAL.*/
  mode = NORMAL_MODE;

  oc_layer_init(self, oc_test_msgproc);
  OC_DEBUG_SET_NAME(self, "OC Test Layer");

  return self;
}


/* Handles incomming messages */
static void oc_test_msgproc(OC_Handle handle, OC_Msg *msg)
{
  OC_Test *self;
  OC_TestGetReqComplete get;
  OC_TestCopyReqComplete copy;
  OC_TestDeleteReqComplete delete;
  OC_Msg sms;
  int del_status;
  MsgBuf *buf;


  OC_DEBUG_ENTER(0, "oc_test_msgproc");
```

```
self = (OC_Test *) handle;

if(mode == NORMAL_MODE)
  {
    switch (msg->event) {

    case EV_SEND_DATA_REQ:
    /* Receive messages from upper layer */
    /* Send this message to lower layer */
    oc_call_msg(self, self->down, EV_SEND_DATA_REQ, msg);

    break;

    case EV_RECV_DATA_IND:
    /* Receive messages from lower layer */
    /* Send this message to upper layer */
    oc_call_msg(self, self->up, EV_RECV_DATA_IND, msg);

    break;

    case EV_TEST_SEND_GET_REQ:
    /* Prepare the layer for get mode */
    mode = GET_MODE;

    break;


    case EV_TEST_SEND_COPY_REQ:
    /* Prepare the layer for copy mode */
    mode = COPY_MODE;

    break;


    case EV_TEST_SEND_DELETE_REQ:
    /* Prepare the layer for delete mode */
    mode = DELETE_MODE;

    break;


    default:

    break;
    }
  }

/* Wait for incoming message and get it */
else if(mode == GET_MODE)
  {
    OC_INIT_AUTO_MSG(sms);

    /* Get MsgBuf from msg.*/
    buf = oc_msg_get_data_buf(msg);

    /* Get vital data.*/
    get.event = msg->event;
    get.start = buf->start;
    get.len = buf->len;
    memcpy(get.buf, oc_msg_get_start(msg), oc_msg_get_len(msg));
```

```
      get.up = self;

      /* Pack response message into sms*/
      oc_msg_set_data_buf(&sms, (MsgBuf *) &get);

      /* Send responses to test-xmlrpc layer*/
      oc_call_msg(self, self->test, EV_TEST_RECV_GET_COMPLETE, &sms);


      /* Reset mode*/
      mode = NORMAL_MODE;
    }

  /* Wait for incoming message and copy it */
  else if(mode == COPY_MODE)
    {
      OC_INIT_AUTO_MSG(sms);

      /* Get MsgBuf from msg.*/
      buf = oc_msg_get_data_buf(msg);

      /* Get vital data.*/
      copy.event = msg->event;
      copy.start = buf->start;
      copy.len = buf->len;
      memcpy(copy.buf, oc_msg_get_start(msg), oc_msg_get_len(msg));

      copy.up = self;

      /* Pack response message into sms*/
      oc_msg_set_data_buf(&sms, (MsgBuf *) &copy);

      /* Send original message further down the stack */
      if(msg->event == EV_SEND_DATA_REQ)
      {
        oc_call_msg(self, self->down, EV_SEND_DATA_REQ, msg);
      }

      /* Send original message up the stack */
      else if(msg->event == EV_RECV_DATA_IND)
      {
        oc_call_msg(self, self->up, EV_RECV_DATA_IND, msg);
      }

      else
      {
        /* Do nothing */
      }

      /* Send responses to test-xmlrpc layer*/
      oc_call_msg(self, self->test, EV_TEST_RECV_COPY_COMPLETE,
&sms);

      /* Reset mode*/
      mode = NORMAL_MODE;
    }

  /* Wait for incoming message and delete it */
  else if(mode == DELETE_MODE)
    {
      OC_INIT_AUTO_MSG(sms);

      /* Response is an integer telling the status of the delete
operation.*/
      delete.del_status = 0;
      delete.up = self;

      oc_msg_set_data_buf(&sms, (MsgBuf *) &delete);

      /* Send responses to test-xmlrpc layer*/
      oc_call_msg(self, self->test, EV_TEST_RECV_DELETE_COMPLETE,
&sms);

      /* Reset mode*/
      mode = NORMAL_MODE;
    }

  /* This should not happen...*/
  else
    {
      OC_ERROR(("'else' in oc_test.c is triggered...."));
    }

  OC_DEBUG_EXIT();
}
```

```
/**
 * \file     oc_test_xmlrpc.h
 * \author   Karl Magnus Nilsen <karlm@obexcode.com>
 * \date     07.12.2003
 *
 * Copyright (C) 2003, ObexCode AS, All Rights Reserved.
 */

#ifndef OC_TEST_XMLRPC_H
#define OC_TEST_XMLRPC_H

#include <core/oc_core.h>

#include <xmlrpc/oc_xmlrpc.h>

/* Exported methods. */
static int New(OC_Handle handle, OC_Msg *msg);
static int Put(OC_Handle handle, OC_Msg *msg);
static int Get(OC_Handle handle, OC_Msg *msg);
static int Copy(OC_Handle handle, OC_Msg *msg);
static int Delete(OC_Handle handle, OC_Msg *msg);

/* Constants representing layers. Used to send messages from Put() */
#define LAYER_HCI   0x00
#define LAYER_TEST  0X01

/* XML-RPC module struct*/
struct oc_test_xmlrpc{
  OC_Handle test;
  OC_List services;
  OC_Handle hci;     /* Handle to HCI layer*/
};
typedef struct oc_test_xmlrpc OC_TestXmlRpc;

struct oc_test_xmlrpc_service {
  OC_XmlRpcService service; /* Superclass, must be first */
};
typedef struct oc_test_xmlrpc_service OC_TestXmlRpcService;

void oc_test_xmlrpc_init(OC_Handle xmlrpc, OC_Handle test, OC_Handle
hci);

#endif /* OC_TEST_XMLRPC_H */
```

```
/**
 * \file     oc_test_xmlrpc.c
 * \author   Karl Magnus Nilsen <karlm@obexcode.com>
 * \date     07.12.2003
 *
 * Copyright (C) 2003, ObexCode AS, All Rights Reserved.
 */

#define OC_DEBUG_LEVEL 4
#include "oc_config.h"

#include <core/oc_core.h>
#include <xmlrpc/oc_xmlrpc.h>

#include <blue/oc_hci.h>

#include "oc_test.h"
#include "oc_test_xmlrpc.h"

static OC_Hci *hci_xmlrpc;
static OC_Test *test_xmlrpc;

/* Handles incoming messages */
static void oc_test_xmlrpc_msgproc(OC_Handle handle, OC_Msg *msg)
{
  OC_XmlRpcService *service = (OC_XmlRpcService *) handle;
  OC_HciConnectionComplete *con;
  OC_TestGetReqComplete *get;
  OC_TestCopyReqComplete *copy;
  OC_TestDeleteReqComplete *delete;
  OC_Msg *ret_msg;
  int *del_status;

  int len;

  OC_DEBUG_ENTER(0, "oc_test_xmlrpc_msgproc");

  switch (msg->event) {
  case EV_HCI_RECV_CONNECT_COMPLETE:
    /* The result of a hci connect request*/
    con = (OC_HciConnectionComplete *) oc_msg_get_data_buf(msg);

    oc_xmlrpc_pack_integer(service->params, con->status);
    oc_xmlrpc_pack_integer(service->params, con->con_handle);
    oc_queue_msg(service, service->session, EV_XMLRPC_SEND_RESPONSE,
service->params);

    break;

  case EV_TEST_RECV_GET_COMPLETE:
    /* Return the message that is requested.*/
    get = (OC_TestGetReqComplete *) oc_msg_get_data_buf(msg);

    oc_xmlrpc_pack_integer(service->params, get->event);
    oc_xmlrpc_pack_integer(service->params, get->start);
    oc_xmlrpc_pack_integer(service->params, get->len);
    oc_xmlrpc_pack_base64(service->params, (uint8 *)get->buf, get-
>len);

    oc_queue_msg(service, service->session, EV_XMLRPC_SEND_RESPONSE,
service->params);
```

```
    break;

  case EV_TEST_RECV_COPY_COMPLETE:
    /* Return a copy of the message that is requested.*/
    copy = (OC_TestCopyReqComplete *) oc_msg_get_data_buf(msg);

    oc_xmlrpc_pack_integer(service->params, copy->event);
    oc_xmlrpc_pack_integer(service->params, copy->start);
    oc_xmlrpc_pack_integer(service->params, copy->len);
    oc_xmlrpc_pack_base64(service->params, (uint8 *)copy->buf, copy-
>len);

    oc_queue_msg(service, service->session, EV_XMLRPC_SEND_RESPONSE,
service->params);

    break;


  case EV_TEST_RECV_DELETE_COMPLETE:
    /* Return an identifier for how the operation went.*/
    delete = (OC_TestDeleteReqComplete *) oc_msg_get_data_buf(msg);

    oc_xmlrpc_pack_integer(service->params, delete->del_status);

    oc_queue_msg(service, service->session, EV_XMLRPC_SEND_RESPONSE,
service->params);

    break;


  default:

    break;
  }

  OC_DEBUG_EXIT();
}


/**
 * Initializes the Test XML-RPC module, and exports all methods.
 *
 * @param xmlrpc OC_Handle to OC_XmlRpc instance.
 * @param test OC_Handle to OC_Test driver.
 * @param hci OC_Handle to OC_Hci driver.
 *
 * @return OC_Handle to new OC_HciXmlRpc instance.
 */
void oc_test_xmlrpc_init(OC_Handle xmlrpc, OC_Handle test, OC_Handle
hci)
{
  OC_DEBUG_ENTER(0, "oc_test_xmlrpc_init");

  test_xmlrpc = test;
  hci_xmlrpc = hci;

  /* Export all required functions */
  oc_xmlrpc_export(xmlrpc, "/test", "New",     &New);
  oc_xmlrpc_export(xmlrpc, "/test", "Put",     &Put);
  oc_xmlrpc_export(xmlrpc, "/test", "Get",     &Get);
  oc_xmlrpc_export(xmlrpc, "/test", "Copy",    &Copy);
  oc_xmlrpc_export(xmlrpc, "/test", "Delete",  &Delete);

  OC_DEBUG_EXIT();
}

/**
 * Get handle.
 *
 * @param handle
 * @param msg
 *
 * @return Handle to client
 */
static int New(OC_Handle handle, OC_Msg *msg)
{
  OC_TestXmlRpcService *self;
  int ret = TRUE;
  OC_UNUSED_AVOID_WARNING(handle);

  OC_DEBUG_ENTER(0, "oc_test_xmlrpc_new");

  self = malloc(sizeof(OC_TestXmlRpcService));
  if (!self) {
    ret = -ENOMEM;
    goto out;
  }
  oc_layer_init(&self->service.layer, &oc_test_xmlrpc_msgproc);
  OC_DEBUG_SET_NAME(self, "Test XML-RPC Service");

  oc_xmlrpc_service_add(handle, self);
#if 0
  oc_timer_msg_init(&self->lease_timer, self,
EV_HCI_XMLRPC_LEASE_TIMEOUT);
  oc_timer_add(&self->lease_timer,
OC_MSECS_TO_TICKS(HCI_XMLRPC_LEASE_TIMEOUT));
#endif

  /* Pack return parameters */
  oc_msg_reset(msg);
  oc_xmlrpc_pack_integer(msg, (int) self);
 out:
  OC_DEBUG_EXIT();
  return ret;
}


/**
 * Inject a message to a stack layer.
 *
 * @param handle Handle to Test driver instance
 * @param msg Message containing parameters for the call.
 *
 * @return
 */
static int Put(OC_Handle handle, OC_Msg *msg)
{
  OC_TestXmlRpcService *self;
  OC_XmlRpcService *service;
  OC_HciConnectReq req;
  OC_Msg sms;
  int event;
  int layer_id;
  int ret = 0;
  uint16 len;
  char message_p[64];

  uint16 buf_len;
  uint16 buf_start;
  MsgBuf msg_buf;
  uint8 *buf;

  OC_DEBUG_ENTER(0, "Put");

  OC_INIT_AUTO_MSG(sms);

  /* Checks if this handle is a legal handle*/
  service = oc_xmlrpc_service_find(handle, msg);
  if (!service) {
    OC_ERROR(("Handle does not exist!"));
    ret = -EINVAL;
    goto out;
  }
  self = (OC_TestXmlRpcService *) service;

  /* Set test layer's pointer to the xml-rpc layer */
  test_xmlrpc->test = self;

  /* Unpack layer identifier*/
  ret = oc_xmlrpc_unpack_integer(msg, &layer_id);

  if (ret) {
    OC_ERROR(("Invalid arguments!"));
    goto out;
  }

  /* Unpack the event code for this message*/
  ret = oc_xmlrpc_unpack_integer(msg, &event);

  if (ret) {
    OC_ERROR(("Invalid arguments!"));
    goto out;
  }

  switch(layer_id)
    {
    case LAYER_HCI:

      /* Unpack the test message */
      len = 64;
      ret = oc_xmlrpc_unpack_base64(msg,  (uint8 *) &req.bd_addr,
&len);

      if (ret) {
      OC_ERROR(("Invalid arguments!"));
      goto out;
      }

      oc_xmlrpc_service_prepare_async(handle, service, msg);

      req.up = self;
      oc_msg_set_data_buf(&sms, (MsgBuf *) &req);
      /* Send message to HCI layer*/
      ret = oc_call_msg(self, hci_xmlrpc, event, &sms);

      break;

    case LAYER_TEST:

      /* Unpack the length of the data of this message*/
      ret = oc_xmlrpc_unpack_integer(msg, &buf_len);

      if (ret) {
      OC_ERROR(("Invalid arguments!"));
      goto out;
      }

      /* Unpack the start index for the data in the data buffer*/
      ret = oc_xmlrpc_unpack_integer(msg, &buf_start);

      if (ret) {
      OC_ERROR(("Invalid arguments!"));
      goto out;
      }

      /* Unpack the data */
      len = 64;
      buf = malloc(buf_len);
      ret = oc_xmlrpc_unpack_base64(msg,  (uint8 *) buf, &len);

      if (ret) {
      OC_ERROR(("Invalid arguments!"));
      goto out;
      }

      /* Set the values for the MsgBuf*/
      msg_buf.start = buf_start;
      msg_buf.len = buf_len;
      memcpy(&msg_buf.buf[buf_start], buf, len);

      oc_xmlrpc_service_prepare_async(handle, service, msg);

      /* Set the the message's message buffer*/
      sms.buf = &msg_buf;

      /* Send message to Test layer*/
      ret = oc_call_msg(self, test_xmlrpc, event, &sms);

      break;

    default:

      break;
    }


#ifdef OC_CONFIG_DEBUG
  if (ret) {
    OC_DEBUG(0, ("Test is busy!"));
  }
#endif /* OC_CONFIG_DEBUG */
```

```
  /* No errors have occured by now, then we know the operation is in
   * progress */
  if (ret == 0) {
    ret = -EINPROGRESS;
  }
 out:
  OC_DEBUG_EXIT();
  return ret;

}

/**
 * Get a message from a stack layer and delete the message.
 *
 * @param handle Handle to Test driver instance
 * @param msg Message containing parameters for the call.
 *
 * @return
 */
static int Get(OC_Handle handle, OC_Msg *msg)
{
  OC_TestXmlRpcService *self;
  OC_XmlRpcService *service;
  OC_TestGetReq req;
  OC_Msg sms;
  int ret = 0;

  OC_DEBUG_ENTER(0, "Get");

  OC_INIT_AUTO_MSG(sms);

  service = oc_xmlrpc_service_find(handle, msg);
  if (!service) {
    OC_ERROR(("Handle does not exist!"));
    ret = -EINVAL;
    goto out;
  }
  self = (OC_TestXmlRpcService *) service;

  /* Set test layer's pointer to the xml-rpc layer */
  test_xmlrpc->test = self;


  /* Unpack the event code*/
  ret = oc_xmlrpc_unpack_integer(msg, &req.event);

  if (ret) {
    OC_ERROR(("Invalid arguments!"));
    goto out;
  }

  /* Unpack the number of messages to get */
  ret = oc_xmlrpc_unpack_integer(msg, &req.num_msg);

  if (ret) {
    OC_ERROR(("Invalid arguments!"));
    goto out;
  }

  oc_xmlrpc_service_prepare_async(handle, service, msg);
  req.up = self;
  oc_msg_set_data_buf(&sms, (MsgBuf *) &req);


  /* Send message to Test layer*/
  ret = oc_call_msg(self, test_xmlrpc, EV_TEST_SEND_GET_REQ, &sms);

#ifdef OC_CONFIG_DEBUG
  if (ret) {
    OC_DEBUG(0, ("Test is busy!"));
  }
#endif /* OC_CONFIG_DEBUG */

  /* No errors have occured by now, then we know the operation is in
   * progress */
  if (ret == 0) {
    ret = -EINPROGRESS;
  }
 out:
  OC_DEBUG_EXIT();
  return ret;
}

/**
 * Get a copy of a message from a stack layer.
 *
 * @param handle Handle to Test driver instance
 * @param msg Message containing parameters for the call.
 *
 * @return
 */
static int Copy(OC_Handle handle, OC_Msg *msg)
{
  OC_TestXmlRpcService *self;
  OC_XmlRpcService *service;
  OC_TestCopyReq req;
  OC_Msg sms;
  int ret = 0;


  OC_DEBUG_ENTER(0, "Copy");

  OC_INIT_AUTO_MSG(sms);

  service = oc_xmlrpc_service_find(handle, msg);
  if (!service) {
    OC_ERROR(("Handle does not exist!"));
    ret = -EINVAL;
    goto out;
  }
  self = (OC_TestXmlRpcService *) service;

  /* Set test layer's pointer to the xml-rpc layer */
  test_xmlrpc->test = self;

  /* Unpack the event code */
  ret = oc_xmlrpc_unpack_integer(msg, &req.event);

  if (ret) {
    OC_ERROR(("Invalid arguments!"));
    goto out;
  }

  /* Unpack the number of messages to copy */
  ret = oc_xmlrpc_unpack_integer(msg, &req.num_msg);

  if (ret) {
    OC_ERROR(("Invalid arguments!"));
    goto out;
  }

  oc_xmlrpc_service_prepare_async(handle, service, msg);

  req.up = self;
  oc_msg_set_data_buf(&sms, (MsgBuf *) &req);

  /* Send message to Test layer*/
  ret = oc_call_msg(self, test_xmlrpc, EV_TEST_SEND_COPY_REQ, &sms);


#ifdef OC_CONFIG_DEBUG
  if (ret) {
    OC_DEBUG(0, ("Test is busy!"));
  }
#endif /* OC_CONFIG_DEBUG */

  /* No errors have occured by now, then we know the operation is in
   * progress */
  if (ret == 0) {
    ret = -EINPROGRESS;
  }
 out:
  OC_DEBUG_EXIT();
  return ret;
}

/**
 * Delete a message.
 *
 * @param handle Handle to Test driver instance
 * @param msg Message containing parameters for the call.
 *
 * @return
 */
static int Delete(OC_Handle handle, OC_Msg *msg)
{
  OC_TestXmlRpcService *self;
  OC_XmlRpcService *service;
  OC_TestDeleteReq req;
  OC_Msg sms;
  int ret = 0;


  OC_DEBUG_ENTER(0, "Delete");

  OC_INIT_AUTO_MSG(sms);

  service = oc_xmlrpc_service_find(handle, msg);
  if (!service) {
    OC_ERROR(("Handle does not exist!"));
    ret = -EINVAL;
    goto out;
  }
  self = (OC_TestXmlRpcService *) service;

  /* Set test layer's pointer to the xml-rpc layer */
  test_xmlrpc->test = self;

  /* Unpack the event code */
  ret = oc_xmlrpc_unpack_integer(msg, &req.event);

  if (ret) {
    OC_ERROR(("Invalid arguments!"));
    goto out;
  }

  /* Unpack the number of messages to delete */
  ret = oc_xmlrpc_unpack_integer(msg, &req.num_msg);

  if (ret) {
    OC_ERROR(("Invalid arguments!"));
    goto out;
  }

  oc_xmlrpc_service_prepare_async(handle, service, msg);

  req.up = self;
  oc_msg_set_data_buf(&sms, (MsgBuf *) &req);

  /* Send message to Test layer*/
  ret = oc_call_msg(self, test_xmlrpc, EV_TEST_SEND_DELETE_REQ,
&sms);


#ifdef OC_CONFIG_DEBUG
  if (ret) {
    OC_DEBUG(0, ("Test is busy!"));
  }
#endif /* OC_CONFIG_DEBUG */

  /* No errors have occured by now, then we know the operation is in
   * progress */
  if (ret == 0) {
    ret = -EINPROGRESS;
  }
 out:
  OC_DEBUG_EXIT();
  return ret;
}
```

```
/**
 * \file     blue.c
 * \author   Dag Brattli <dag@obexcode.com>
 * \author   Karl Magnus Nilsen <karlm@obexcode.com>
 * \date     07.12.2003
 *
 * Copyright (C) 2003, ObexCode AS, All Rights Reserved.
 */

#define OC_DEBUG_LEVEL 4
/* Core includes */
#include <core/oc_core.h>
#include <util/oc_util.h>

/* System includes */
#include <stdio.h>
#include <stdarg.h>

#include <blue/oc_hci_bluez.h>

#include <test/oc_test.h>
#include <linux/oc_async_io.h>

#ifdef OC_CONFIG_DEBUG
# define MAX_FUNCTION_COUNT 500
static char *funcs[MAX_FUNCTION_COUNT];
static uint32 func_counts[MAX_FUNCTION_COUNT];
int8 indentlevel = 0;
static uint32 call_counter = 0;
#endif /* OC_CONFIG_DEBUG */

static DebugInfo debug_info;

static volatile int running;
static OC_Timer timer;

/**
 * Function debug_print:
 * @type: Type of message to print.
 * @fmt: Format string
 * @args: Argument list
 *
 * Debug print function. This function simply just prints to standard
 * error.  But it could do more or less anything.  Imagination is the
 * limit. :)
 */
void debug_print(int type, const char CODE *fmt, va_list args)
{
  struct timeval time;
  char *hilite = "";
  char *str = NULL;
  uint32 s;
  uint8 log = FALSE;
  FILE *fp;
#ifdef OC_CONFIG_DEBUG
  int i;
#endif /* OC_CONFIG_DEBUG */

  fp = stderr;

  oc_gettimeofday(&time, NULL);
  switch (type) {
  case OC_TYPE_DEBUG:
    str = "DBG";
    log = FALSE;
    break;
  case OC_TYPE_WARNING:
    hilite = "\e[31m";
    str = "WRN";
    break;
  case OC_TYPE_ERROR:
    hilite = "\e[30;41m";
    str = "ERR";
    break;
  case OC_TYPE_INFO:
    hilite = "\e[33m";
    str = "INF";
    break;
  }

  s = (time.tv_sec) % 86400;
  fprintf(fp, "%s%02d:%02d:%02d.%06u ", hilite,
        s / 3600, (s % 3600) / 60,
        s % 60, (uint32) time.tv_usec);

  if (str)
    fprintf(fp, "%s ", str);

#ifdef OC_CONFIG_DEBUG
  if (indentlevel > 0) {
    for (i=0; i<indentlevel-1; i++) {
      fprintf(fp, "| ");
    }
    fprintf(fp, "[%d] ", indentlevel);
  }
#endif /* OC_CONFIG_DEBUG */
  if (args)
    vfprintf(fp, fmt, args);
  else
    fprintf(fp, "%s", fmt);

  fprintf(fp, "\e[37;0m\n");
  fflush(fp);
}

#ifdef OC_CONFIG_DEBUG
/**
 * Function debug_enter:
 * @name: Name of this block (usually the function name)
 * @print: TRUE if print is enabled.
 *
 * Enter debug block. Print the name of this block and increase
 * indentlevel by 1.
 */
void debug_enter(const char CODE *name, int8 print)
{
  indentlevel++;

  if (print) {
    debug_print((int) OC_TYPE_DEBUG, name, NULL);
  }
```

```
}

/**
 * Function ocd_dbg_exit:
 *
 * Exit debug block. Decrease indentlevel by 1.
 */
void debug_exit(void)
{
  if (indentlevel > 0)
    indentlevel--;
}
#endif /* OC_CONFIG_DEBUG */


/**
 * Function main:
 *
 * Main function that starts the server.
 */
int main(int argc, char **argv)
{
  OC_Handle hci;
  OC_Handle test;
  OC_Handle xmlrpc;

  OC_Test *self;
  OC_Hci *hci_l;
  OC_AsyncIo *async_io;

#ifdef OC_CONFIG_DEBUG
  int i;

  debug_info.enter_callback = &debug_enter;
  debug_info.exit_callback  = &debug_exit;

  for (i = 0; i < MAX_FUNCTION_COUNT; i++) {
    funcs[i] = NULL;
    func_counts[i] = 0;
  }
#else
  debug_info.enter_callback = NULL;
  debug_info.exit_callback  = NULL;
#endif /* OC_CONFIG_DEBUG */
  /* The last part of the debugging system is always enabled now. */
  debug_info.print_callback = &debug_print;
  OC_DEBUG_INIT(&debug_info);

  /* Initialize the system */
  oc_mainloop_init();
  oc_timer_init(&timer, NULL);

  /* Initialize hci layer */
  hci = oc_hci_new(NULL);

  /* Initialize test layer*/
  test = oc_test_new();

  self = (OC_Test *)test;
  hci_l = (OC_Hci *)hci;
  async_io = hci_l->down;
  /* Insert test layer into stack*/
  oc_test_bind_up(self, hci_l);
  oc_test_bind_down(self, async_io);

  /* Initialize XML-RPC layer */
  xmlrpc = oc_xmlrpc_new(7888);
  oc_test_xmlrpc_init(xmlrpc, test, hci);

  /* Start the Select Loop */
  running = 1;

  oc_mainloop_run(&running);

  return 0;
}
```

```
/**
 * \file     Start.java
 * \author   Karl Magnus Nilsen <karlm@obexcode.com>
 * \date     07.12.2003
 *
 * Copyright (C) 2003, ObexCode AS, All Rights Reserved.
 */

/**
 * This class starts the show.
 *
 * Initializes the test objects and starts the GUI.
 */

class Start
{
    public static void main(String []args)
    {
      MainGUI m = new MainGUI();

      Test.initTests();
      m.initMain();
    }
}
```

```
/**
 * \file     MainGUI.java
 * \author   Karl Magnus Nilsen <karlm@obexcode.com>
 * \date     07.12.2003
 *
 * Copyright (C) 2003, ObexCode AS, All Rights Reserved.
 */

/**
 * The GUI and Listener classes of the test system.
 *
 * The GUI class represents the GUI of the program.
 * The Listener class contains methods which react to events in the
GUI.
 */

import java.awt.*;
import java.awt.event.*;
import java.util.Vector;

class MainGUI extends Frame
{
    Choice testTypeList = new Choice();
    List chooseTestList = new List(16, true);
    TextArea infoTA = new TextArea(null, 5, 20,
TextArea.SCROLLBARS_VERTICAL_ONLY);
    TextArea progressTA = new TextArea(null, 5, 20,
TextArea.SCROLLBARS_VERTICAL_ONLY);

    /* Initialize main window.*/
    void initMain()
    {
      Label testType = new Label("Test type");
      Label chooseTest = new Label("Choose test");
      Label testInfo = new Label("Test information");
      Label progress = new Label("Test progress");
      Button about = new Button("About");
      Button viewInfo = new Button("View info");
      Button execute = new Button("Execute");
      Button exit = new Button("Exit");

      /* Populate choice lists */
      testTypeList.insert("Choose test type", 0);
      testTypeList.insert("Conformance", 1);
      testTypeList.insert("Reliability", 2);
      testTypeList.insert("Fault tolerance", 3);
      testTypeList.insert("Performance", 4);
      testTypeList.insert("Interoperability", 5);

      /* Create listeners for the GUI components*/
      Listener aboutListener = new Listener(this, Listener.ABOUT);
      Listener viewInfoListener = new Listener(this,
Listener.VIEW_INFO);
      Listener executeListener = new Listener(this,
Listener.EXECUTE);
      Listener exitListener = new Listener(this, Listener.EXIT);
      Listener testTypeListListener = new Listener(this,
Listener.TEST_TYPE_LIST);
      Listener chooseTestListListener = new Listener(this,
Listener.CHOOSE_TEST_LIST);
```

```
      about.addActionListener(aboutListener);
      viewInfo.addActionListener(viewInfoListener);
      execute.addActionListener(executeListener);
      exit.addActionListener(exitListener);
      testTypeList.addItemListener(testTypeListListener);
      chooseTestList.addActionListener(chooseTestListListener);

      /* Using a GridBagLayout to place the GUI components*/
      GridBagLayout gbl = new GridBagLayout();
      GridBagConstraints gbc = new GridBagConstraints();
      gbc.insets = new Insets(2,2,2,2);
      gbc.fill = GridBagConstraints.HORIZONTAL;
      gbc.weightx = 1.0;

      infoTA.setEditable(false);
      progressTA.setEditable(false);

      /* Adding GUI components*/
      this.setSize(900, 500);
      this.setTitle("Test System Client - v. 1.0");
      this.setBackground(Color.LIGHT_GRAY);
      this.setLayout(gbl);
      this.addComponent(this, gbl, gbc, 0, 0, 1, 1, testType);
      this.addComponent(this, gbl, gbc, 1, 0, 1, 1, chooseTest);
      this.addComponent(this, gbl, gbc, 2, 0, 1, 1, testInfo);
      this.addComponent(this, gbl, gbc, 0, 1, 1, 1, testTypeList);
      this.addComponent(this, gbl, gbc, 1, 1, 1, 3, chooseTestList);
      this.addComponent(this, gbl, gbc, 2, 1, 1, 1, infoTA);
      this.addComponent(this, gbl, gbc, 2, 2, 1, 1, progress);
      this.addComponent(this, gbl, gbc, 2, 3, 1, 1, progressTA);
      this.addComponent(this, gbl, gbc, 0, 5, 1, 1, about);
      this.addComponent(this, gbl, gbc, 2, 4, 1, 1, viewInfo);
      this.addComponent(this, gbl, gbc, 2, 5, 1, 1, execute);
      this.addComponent(this, gbl, gbc, 2, 6, 1, 1, exit);
      this.addWindowListener(new WindowAdapter(){public void
windowClosing(WindowEvent e){System.exit(0);}});

      this.setVisible(true);
    }

    /* Function to add components in a grid bag layout.*/
    void addComponent(Container cont, GridBagLayout gbl,
GridBagConstraints gbc, int x, int y, int xx, int yy, Component comp)
{
      gbc.gridx = x;
      gbc.gridy = y;
      gbc.gridwidth = xx;
      gbc.gridheight = yy;
      gbl.setConstraints(comp, gbc);
      cont.add(comp);
    }

}

class Listener implements ActionListener, ItemListener
{
    /* Identifiers for the GUI components which requires a Listener*/
    static final int ABOUT            =  0;
    static final int VIEW_INFO        =  1;
    static final int EXECUTE          =  2;
    static final int EXIT             =  3;
    static final int TEST_TYPE_LIST   =  4;
    static final int CHOOSE_TEST_LIST =  5;

    MainGUI main;
    int choice;

    /* Current choice in choice list.*/
    static String strChoice = null;

    /* Listener objects are instantiated with a GUI component ID*/
    public Listener(MainGUI m, int c)
    {
      main = m;
      choice = c;
    }


    /* Reacts to action events in the GUI*/
    public void actionPerformed(ActionEvent e)
    {
      switch(choice)
          {
          case ABOUT:
            System.out.println("About");
            break;

          case VIEW_INFO:
            updateInfoField();
            break;

          case EXECUTE:
            executeTests();
            break;

          case EXIT:
            System.out.println("Exiting Test System Client");
            System.exit(0);
            break;

          case TEST_TYPE_LIST:

            break;

          case CHOOSE_TEST_LIST:
            updateInfoField();
            break;


          default:
            System.err.println("This should not happen! An event that
doesn't exist occured...");
            System.out.println(choice);
            System.exit(0);
            break;
          }
    }

    /* Reacts to the events on the choice list.*/
    public void itemStateChanged(ItemEvent e)
    {
      strChoice = (String)e.getItem();
      if(strChoice.compareTo("Conformance") == 0)
          {
            updateTestList(Conformance.testList);
          }
      else if(strChoice.compareTo("Reliability") == 0)
          {
             updateTestList(Reliability.testList);
          }
      else if(strChoice.compareTo("Fault tolerance") == 0)
          {
            updateTestList(FaultTolerance.testList);
          }
      else if(strChoice.compareTo("Performance") == 0)
          {
            updateTestList(Performance.testList);
          }
      else if(strChoice.compareTo("Interoperability") == 0)
          {
            updateTestList(Interoperability.testList);
          }
      else if(strChoice.compareTo("Choose test type") == 0)
          {
            /* Do nothing - (This is just an information field, not a
test type*/
          }
      else
          {
            System.err.println("This should not happen! An event that
doesn't exist occured...");
            System.out.println(choice);
            System.exit(0);
          }
    }

    /* Updates list of available tests.*/
    void updateTestList(Vector v)
    {
      /* Removes old elements in list before inserting new
elements.*/
      main.chooseTestList.removeAll();
      for(int i=0; i<v.size(); i++)
          {
            Test tmp = (Test)v.get(i);
            /* Puts the avalable tests into the list of tests.*/
            main.chooseTestList.add(tmp.testID, i);
          }
    }

    /* Updates the test info field.*/
    void updateInfoField()
    {
      String choice = new String();

      /* Clearing text area.*/
      main.infoTA.setText("");

      if(strChoice.compareTo("Conformance") == 0)
          {
            findInfo(Conformance.testList);
          }
      else if(strChoice.compareTo("Reliability") == 0)
          {
            findInfo(Reliability.testList);
          }
      else if(strChoice.compareTo("Fault tolerance") == 0)
          {
            findInfo(FaultTolerance.testList);
          }
      else if(strChoice.compareTo("Performance") == 0)
          {
            findInfo(Performance.testList);
          }
      else if(strChoice.compareTo("Interoperability") == 0)
          {
            findInfo(Interoperability.testList);
          }
      else
          {
            /* Do nothing*/
            System.out.println("Nothing is chosen.");
          }
    }

    /* Fetches the correct information from the chosen tests.*/
    void findInfo(Vector v)
    {
      String []selected = main.chooseTestList.getSelectedItems();

      for(int i=0; i<selected.length; i++)
          {
            for(int j=0; j<v.size(); j++)
                {
                  Test tmp = (Test)v.get(j);
                  if(tmp.testID.compareTo(selected[i]) == 0)
                      {
                        main.infoTA.append(tmp.testID + "\n");
                        main.infoTA.append(tmp.description + "\n\n");
                      }
                }
          }
    }

    /* Executes the chosen tests and displays the progress and
results.*/
    void executeTests()
    {
      /* Get selected tests*/
      String []selected = main.chooseTestList.getSelectedItems();
      String results;

      if(selected.length == 0)
          {
            main.progressTA.append("No tests chosen.\n---------------
---------\n");
          }
      else
          {
            for(int i=0; i<selected.length; i++)
                {
                  main.progressTA.append("Executing test: " +
selected[i] + "\n");
```

```
                  results = Control.executeTest(strChoice,
selected[i]);
                  main.progressTA.append("Receiving results from " +
selected[i] + "\n");
                  main.progressTA.append(results + "\n");
                  main.progressTA.append("------------------------
\n");
                }
          }
    }
}
```

```
/**
 * \file     Control.java
 * \author   Karl Magnus Nilsen <karlm@obexcode.com>
 * \date     07.12.2003
 *
 * Copyright (C) 2003, ObexCode AS, All Rights Reserved.
 */

/**
 * The Control class is coordinating the events of the system.
 *
 */

import java.util.*;
import java.io.*;

class Control
{
    /* Create log file.*/
    static FileWriter createLogFile(String testID)
    {
      FileWriter logFile = null;

      try
          {
            /* Include test ID and current time (in ms) in log file
name.*/
            Date currTime = new Date();
            long currTimeMillis = currTime.getTime();
            String fileName = "results_" + testID + "_" +
currTimeMillis + ".txt";
            File file = new File(fileName);

            /* Checks if log file has been created. If not, try again
in 100 ms. Retry max 10 times.*/
            int i = 0;
            while((file.createNewFile()) == false && i < 10)
                {
                  System.err.println("Failed creating log file.");
                  Thread.sleep(100);
                  i++;

                  currTime = new Date();
                  currTimeMillis = currTime.getTime();
                  fileName = "results" + currTimeMillis + ".txt";
                  file = new File(fileName);
                }
            /* If unable to create log file after ten attempts,
quit.*/
            if(i >= 10)
                {
                  System.err.println("\nCould not create log file.");
                  System.exit(0);
                }

            logFile = new FileWriter(fileName, true);
          }
      catch(IOException e)
          {
            System.err.println(e);
            System.exit(0);
          }
      catch(InterruptedException e)
          {
            System.err.println(e);
            System.exit(0);
          }

      return logFile;
    }

    /* Executes a test.
     *
     * This method is called from the Listener class when a test is
     * chosen to be executed.
     *
     * testType - The test type
     * testID - The id of the test to be executed.
     */
    static String executeTest(String testType, String testID)
    {
      Test test = null;
      String ret = null;

      FileWriter logFile;

      /* Find the test object that is called.*/
      if(testType.compareTo("Conformance") == 0)
          {
            test = (Conformance)findTest(Conformance.testList,
testID);
          }
      else if(testType.compareTo("Reliability") == 0)
          {
            test = (Reliability)findTest(Reliability.testList,
testID);
          }
      else if(testType.compareTo("Fault tolerance") == 0)
          {
            test = (FaultTolerance)findTest(FaultTolerance.testList,
testID);
          }
      else if(testType.compareTo("Performance") == 0)
          {
            test = (Performance)findTest(Performance.testList,
testID);
          }
      else if(testType.compareTo("Interoperability") == 0)
          {
            test =
(Interoperability)findTest(Interoperability.testList, testID);
          }
      else
          {
            /* The test selected does not exist.*/
            System.err.println("Control.executeTest - Can not execute
0 tests.");

            System.exit(0);
          }

      /* Run the test object's test method*/
      ret = test.runTest(testType, testID);

      /* Write result of the test to log file together with the ID of
the test
       * and the time it was executed.
       */
      try
          {
            Date d = new Date();
            logFile = createLogFile(testID);
            logFile.write(testID + "\n");
            logFile.write(d.toString()+ "\n\n");
            logFile.write("Result:" + "\n");
            logFile.write(ret + "\n");
            logFile.close();
          }
      catch(IOException e)
          {
            System.err.println(e);
          }

      /* Returns the string that is to be displayed to the user in
the GUI.*/
      return ret;
    }

    /* Finds the correct test object based on the testID.*/
    static Test findTest(Vector v, String testID)
    {
      Test test = null;
      for(int i=0; i<v.size(); i++)
          {
            Test tmp = (Test)v.get(i);
            if(tmp.testID.compareTo(testID) == 0)
                {
                  test = (Test)v.elementAt(i);
                }
          }
      return test;
    }
}
```

```
/**
 * \file     MyXmlRpcClient.java
 * \author   Karl Magnus Nilsen <karlm@obexcode.com>
 * \date     07.12.2003
 *
 * Copyright (C) 2003, ObexCode AS, All Rights Reserved.
 */

/**
 * The MyXmlRpcClient class represents an XML-RPC client.
 *
 * Used to contact the IUT.
 * Uses the XML-RPC implementation from Apache
 */

import java.util.*;
import org.apache.xmlrpc.*;
import java.io.*;

class MyXmlRpcClient
{
    String server;

    /* Objects are invoked with a server name.*/
    public MyXmlRpcClient(String s)
    {
      server = s;
    }

    /* Executes the XML-RPC call.
      *
      * funcName - The remote function
      * paramList - A vector containing the remote function's
parameter(s)
      */
    Object executeXmlRpc(String funcName, Vector paramList)
    {
      Object response;

      try
          {
            XmlRpcClient conn = new XmlRpcClient(server);

            response = conn.execute(funcName, paramList);
          }

      catch(XmlRpcException e)
          {
            System.err.println(e);
            response = new String(e.toString());
          }

      catch(IOException e)
          {
            System.err.println(e);
            response = new String(e.toString());
          }
      return response;
    }
}
```

```
/**
 * \file     Test.java
 * \author   Karl Magnus Nilsen <karlm@obexcode.com>
 * \date     07.12.2003
 *
 * Copyright (C) 2003, ObexCode AS, All Rights Reserved.
 */

/**
 * The Test class is the Superclass for the test classes, that is,
the classes that implement
 * the actual tests.
 *
 * The class hierarchy:
 * Test
 *   |
 *   |---Conformance
 *   |---FaultTolerance
 *   |---Reliability
 *   |---Performance
 *   |---Interoperability
 *
 * This class is responsible for the common properties of the test
classes.
 */

import java.util.Vector;
import java.util.Hashtable;
import java.lang.reflect.Method;
import java.lang.reflect.Constructor;
import java.lang.Thread;

class Test
{
    String testID;             /* The test id, also the name of
method that implements the test.*/
    String description;        /* A brief description of the test.*/
    Object passV;              /* The test's Pass verdict.*/
    Object failV;              /* The test's Fail verdict.*/
    Object inconclusiveV;      /* The test's Inconclusive verdict.*/

    /* Hashtable with descriptions of the tests.*/
    static Hashtable descrList = new Hashtable();

    /* Test objects are initalized with the test ID and the test
description*/
    public Test(String id, String d)
    {
      testID = id;
      description = d;
    }

    /* Initializes test description list.
     *
     * This feature is optional. A test does not need a description.
     */
    static void initDescrList()
    {
      /* Set test descriptions here.
       *
       * The key is the test ID, i.e. the method name of the
implemented test.
       */
      Test.descrList.put("TP_HCI_BV_22_C", "Test the HCI InquiryReq()
function.");
      Test.descrList.put("TP_HCI_BV_23_C", "Test the HCI ConnectReq()
function.");
      Test.descrList.put("TP_HCI_BV_02_FT", "Put a message into the
stack and get it when it reaches the test layer.");
      Test.descrList.put("TP_HCI_BV_10_C", "Test the Test New()
function.");
      Test.descrList.put("TP_HCI_BV_14_C", "Try to make a connection
using Put().");
      Test.descrList.put("TP_HCI_BV_15_C", "Copy a message.");
      Test.descrList.put("TP_HCI_BV_04_FT", "Delete a message.");
    }

    /* Initializes the test objects.
     *
     * Uses the method name as test ID and puts the tests in the test
vectors for each class.
     */
    static void initTests()
    {
      Class c;
      Method []methods;
      Test tmp;
      String method;
      String descr;

      /* Initialize test description list.*/
      initDescrList();

      /* Initiates test objects and inserts them in the vector of
their corresponding class.*/
      try
          {
            c = Class.forName("Conformance");
            methods = c.getDeclaredMethods();
            for(int i=0; i<methods.length; i++)
                {
                  method = methods[i].getName();
                  descr = (String)Test.descrList.get(method);
                  if(descr == null)
                      {
                        tmp = new Conformance(method, "This test does
not have a description.");
                      }
                  else
                      {
                        tmp = new Conformance(method, descr);
                      }
                  Conformance.testList.add(i, tmp);
                }

            c = Class.forName("Reliability");
            methods = c.getDeclaredMethods();
            for(int i=0; i<methods.length; i++)
                {

                  method = methods[i].getName();
                  descr = (String)Test.descrList.get(method);
                  if(descr == null)
                      {
                        tmp = new Reliability(method, "This test does
not have a description.");
                      }
                  else
                      {
                        tmp = new Reliability(method, descr);
                      }
                  Reliability.testList.add(i, tmp);
                }

            c = Class.forName("FaultTolerance");
            methods = c.getDeclaredMethods();
            for(int i=0; i<methods.length; i++)
                {
                  method = methods[i].getName();
                  descr = (String)Test.descrList.get(method);
                  if(descr == null)
                      {
                        tmp = new FaultTolerance(method, "This test
does not have a description.");
                      }
                  else
                      {
                        tmp = new FaultTolerance(method, descr);
                      }
                  FaultTolerance.testList.add(i, tmp);
                }

            c = Class.forName("Performance");
            methods = c.getDeclaredMethods();
            for(int i=0; i<methods.length; i++)
                {
                  method = methods[i].getName();
                  descr = (String)Test.descrList.get(method);
                  if(descr == null)
                      {
                        tmp = new Performance(method, "This test does
not have a description.");
                      }
                  else
                      {
                        tmp = new Performance(method, descr);
                      }
                  Performance.testList.add(i, tmp);
                }

            c = Class.forName("Interoperability");
            methods = c.getDeclaredMethods();
            for(int i=0; i<methods.length; i++)
                {
                  method = methods[i].getName();
                  descr = (String)Test.descrList.get(method);
                  if(descr == null)
                      {
                        tmp = new Interoperability(method, "This test
does not have a description.");
                      }
                  else
```

```
                      {
                        tmp = new Interoperability(method, descr);
                      }
                  Interoperability.testList.add(i, tmp);
                }

          }
        catch(Exception e)
            {
              System.err.println(e);
            }
    }


    /* Run the test method.
     *
     * type - The test type/class of the test.
     * id - The test ID.
     */
    String runTest(String type, String id)
    {
      String res = null;
      try
          {
            Class c = Class.forName(type);
            Method m = c.getMethod(id, null);
            res = (String)m.invoke(this, null);
            return res;
          }
      catch(Exception e)
          {
            System.err.println(e);
          }
      return res;
    }

    /* Get handle from test layer.
     *
     * All exported methods from the test layer needs a handle as
first parameter.
     *
     * client - The connection to the test layer's XML-RPC interface
     */
    Integer getHandle(MyXmlRpcClient client)
    {
      String funcName = "New";
      Vector paramList = new Vector();
      System.out.println("Starting XML-RPC call - New()");
      Integer handle = (Integer)client.executeXmlRpc(funcName,
paramList);
      System.out.println(handle);
      System.out.println("XML-RPC call returned - New()");

      return handle;
    }

    /* Set a test's Pass verdict.
     *
     * verdict - The Pass verdict, represented as a String.
     */
    void setPassVerdict(Object verdict)
    {
      this.passV = verdict;
    }

    /* Set a test's Fail verdict.
     *
     * verdict - The Fail verdict, represented as a String.
     */
    void setFailVerdict(Object verdict)
    {
      this.failV = verdict;
    }

    /* Set a test's Inconclusive verdict.
     *
     * verdict - The Inconclusive verdict, represented as a String.
     */
    void setInconclusiveVerdict(Object verdict)
    {
      this.inconclusiveV = verdict;
    }
}

/**
 * Contains the Conformance tests.
 */
class Conformance extends Test
{
    static Vector testList = new Vector();

    public Conformance(String id, String d)
    {
      super(id, d);
    }

    /* Test TP_HCI_BV_10_C
     *
     * Test the Test-xmlrpc-layer's New() function.
     * Function - New()
     */
    public String TP_HCI_BV_10_C()
    {
      /* The return value.*/
      String ret;
      /* The remote function name.*/
      String funcName;
      /* The parameter list for the remote call.*/
      Vector paramList;
      /* The handle for the remote call.*/
      Integer handle;
      /* The returned object from the call.*/
      Object retVal;

      MyXmlRpcClient client = new
MyXmlRpcClient("http://matrix.tromso.obexcode.vpn:7888/test");

      /* Set pass, fail and inconclusive verdict.*/
      this.setPassVerdict(new Integer(0));           /* Will receive
zero.*/
```

```
      /* Get handle - Execute call to New() with empty parameter
list.*/
      funcName = "New";
      paramList = new Vector();
      System.out.println("Starting XML-RPC call - New()");
      handle = (Integer)client.executeXmlRpc(funcName, paramList);
      System.out.println(handle);
      System.out.println("XML-RPC call returned - New()");

      ret = handle.toString();

      return ret;
    }


    /* Test TP_HCI_BV_14_C
     *
     * Try to make a HCI-connection to a BT device using Put().
     */
    public String TP_HCI_BV_14_C()
    {
      /* The return value.*/
      String ret;
      /* The remote function name.*/
      String funcName;
      /* The parameter list for the remote call.*/
      Vector paramList;
      /* The handle for the remote call.*/
      Integer handle;
      /* The returned object from the call.*/
      Object retVal;

      MyXmlRpcClient client = new
MyXmlRpcClient("http://matrix.tromso.obexcode.vpn:7888/test");

      /* Set pass verdict.*/

      this.setPassVerdict(new Integer(0));           /* Will receive
zero.*/

      /* Get handle - Execute call to New() with empty parameter
list.*/
      handle = getHandle(client);

      funcName = "Put";
      paramList = new Vector();
      paramList.add(handle);
      paramList.add(new Integer(0x00));       /* Layer ID*/
      paramList.add(new Integer(0x05));       /* Event code
(EV_HCI_SEND_CONNECT_REQ = 0x05)*/
      /* Connect to address 4d:de:15:d9:0a:00 */
      byte [] message = new byte[6];
      message[0] = (byte)0x4d;
      message[1] = (byte)0xde;
      message[2] = (byte)0x15;
      message[3] = (byte)0xd9;
      message[4] = (byte)0x0a;
      message[5] = (byte)0x00;
      paramList.add(message);

      System.out.println("Starting XML-RPC call - Put()");
      retVal = (Integer)client.executeXmlRpc(funcName, paramList);

      System.out.println("XML-RPC call returned - Put()");

      /* Evluating result*/
      if(retVal.getClass().toString().compareTo("class
java.lang.Integer") != 0)
          {
            /* Have received an unexpected class in return*/
            ret = retVal.toString() + "\nThe test is inconclusive.";
          }
      else if(retVal.equals(this.passV))
          {
            ret = retVal.toString() + "\nThe test is passed.";
          }
      else
          {
            ret = retVal.toString() + "\nThe test failed.";
          }


      ret = handle.toString();

      return ret;
    }


    /* Test TP_HCI_BV_15_C
     *
     * Put a message into the stack and copy it when it reaches the
test layer.
     * Like TP_HCI_BV_02_FT this method suffers from the lack of
support for sending
     * arrays in the XML-RPC server used. Check the comments on
TP_HCI_BV_02_FT in
     * class FaultTolerance for more information.
     * When a message is copied, only the data buffer is sent to the
client.
     */
    public String TP_HCI_BV_15_C()
    {
      /* The return value.*/
      String ret;
      /* The remote function name.*/
      String funcName;
      /* The parameter list for the remote call.*/
      Vector paramList;
      /* The handle for the remote call.*/
      Integer handle;
      /* The returned object from the call.*/
      Object retVal;

      MyXmlRpcClient client = new
MyXmlRpcClient("http://matrix.tromso.obexcode.vpn:7888/test");

      /* Get handle */
      handle = this.getHandle(client);

      funcName = "Copy";
      paramList = new Vector();
      paramList.add(handle);
      paramList.add(new Integer(0x05));     /* Event code */
      paramList.add(new Integer(0x01));     /* Number of messages to
copy */
      System.out.println("Starting XML-RPC call - Copy()");
      retVal = client.executeXmlRpc(funcName, paramList);
      System.out.println("XML-RPC call returned - Copy()");

      funcName = "Put";
      paramList = new Vector();
      paramList.add(handle);
      paramList.add(new Integer(0x00));       /* Layer ID*/
      paramList.add(new Integer(0x05));       /* Event code
(EV_HCI_SEND_CONNECT_REQ = 0x05)*/
      /* Connect to address 4d:de:15:d9:0a:00 */
      byte [] message = new byte[6];
      message[0] = (byte)0x4d;
      message[1] = (byte)0xde;
      message[2] = (byte)0x15;
      message[3] = (byte)0xd9;
      message[4] = (byte)0x0a;
      message[5] = (byte)0x00;
      paramList.add(message);

      System.out.println("Starting XML-RPC call - Put()");
      byte [] retMsg = (byte [])client.executeXmlRpc(funcName,
paramList);
      System.out.println("XML-RPC call returned - Put()");

      ret = retMsg.toString();
      return ret;
    }


     /* Test TP_HCI_BV_22_C
     *
     * Test the request to run an inquiry about Bluetooth devices in
the vicinity.
     * This method does not use the test layer, but is used to show
that this program
     * can be used to test the functionality of the exported
functions of the HCI layer.
     * Function - InquiryReq()
     */
    public String TP_HCI_BV_22_C()
    {
      /* The return value.*/
      String ret;
      /* The remote function name.*/
      String funcName;
      /* The parameter list for the remote call.*/
      Vector paramList;
      /* The handle for the remote call.*/
      Integer handle;
      /* The returned object from the call.*/
      Object retVal;

      MyXmlRpcClient client = new
MyXmlRpcClient("http://matrix.tromso.obexcode.vpn:7899/hci");

      /* Get handle - Execute call to New() with empty parameter
list.*/
```

```
      handle = getHandle(client);

      /* Call InquiryReq(), handle as parameter.*/
      funcName = "InquiryReq";
      paramList = new Vector();
      paramList.add(handle);
      System.out.println("Starting XML-RPC call. - InquiryReq()");
      retVal = client.executeXmlRpc(funcName, paramList);
      System.out.println("XML-RPC call returned - InquiryReq()");

      ret = retVal.toString();
      return ret;
    }

    /* Test TP_HCI_BV_23_C
     *
     * Test the request to make a connection to a remote device.
     * This method does not use the test layer, but is used to show
that this program
     * can be used to test the functionality of the exported
functions of the HCI layer.
     * Function - ConnectReq()
     */
    public String TP_HCI_BV_23_C()
    {
      /* The return value.*/
      String ret;
      /* The remote function name.*/
      String funcName;
      /* The parameter list for the remote call.*/
      Vector paramList;
      /* The handle for the remote call.*/
      Integer handle;
      /* The returned object from the call.*/
      Object retVal;


      MyXmlRpcClient client = new
MyXmlRpcClient("http://matrix.tromso.obexcode.vpn:7899/hci");

      /* Get handle - Execute call to New() with empty parameter
list.*/
      handle = getHandle(client);

      /* Call ConnectReq() function with handle as first parameter.*/
      funcName = "ConnectReq";
      paramList = new Vector();
      paramList.add(handle);
      /* Connect to address  */
      byte [] message = new byte[6];
      message[0] = (byte)0x4d;
      message[1] = (byte)0xde;
      message[2] = (byte)0x15;
      message[3] = (byte)0xd9;
      message[4] = (byte)0x0a;
      message[5] = (byte)0x00;
      paramList.add(message);

      System.out.println("Starting XML-RPC call. - ConnectReq()");
      retVal = client.executeXmlRpc(funcName, paramList);
      System.out.println("XML-RPC call returned - ConnectReq()");
      ret = retVal.toString();
      return ret;
    }
}

/**
 * Contains the Fault Tolerance tests.
 */
class FaultTolerance extends Test
{
    static Vector testList = new Vector();

    public FaultTolerance(String id, String d)
    {
      super(id, d);
    }

    /* Test TP_HCI_BV_02_FT
     *
     * Insert message to the stack and take it out from the test
layer.
     * Then return the same message to the test layer where it will
be forwarded.
     *
     * This method suffers from the lack of support for sending
arrays from the XML-RPC server.
     * When a message is taken out of the stack, the message data,
the length of the data, the start
     * index of the data in the data buffer and the message's event
code is supposed to be returned to
     * the XML-RPC client. Right now it is only possible to send one
of these return values. We have chosen
     * to write code for the future situation where array-sending is
supported. This code is, however,
     * commented out, but shall be used when the functionality
required is present.
     * We have chosen to receive the actual message data, and hard
code the other parameters, just to
     * try to test the functionality of the test layer.
     */
    public String TP_HCI_BV_02_FT()
    {
      /* The return value.*/
      String ret;
      /* The remote function name.*/
      String funcName;
      /* The parameter list for the remote call.*/
      Vector paramList;
      /* The handle for the remote call.*/
      Integer handle;
      /* The returned object from the call.*/
      Object retVal;

      MyXmlRpcClient client = new
MyXmlRpcClient("http://matrix.tromso.obexcode.vpn:7888/test");

      /* Get handle */
      handle = this.getHandle(client);

      funcName = "Get";
      paramList = new Vector();
      paramList.add(handle);
      paramList.add(new Integer(0x05));
      paramList.add(new Integer(0x01));
      System.out.println("Starting XML-RPC call - Get()");
      retVal = client.executeXmlRpc(funcName, paramList);
      System.out.println("XML-RPC call returned - Get()");

      funcName = "Put";
      paramList = new Vector();
      paramList.add(handle);
      paramList.add(new Integer(0x00));       /* Layer ID*/
      paramList.add(new Integer(0x05));       /* Event code
(EV_HCI_SEND_CONNECT_REQ = 0x05)*/
      /* Connect to address 4d:de:15:d9:0a:00 */
      byte [] message = new byte[6];
      message[0] = (byte)0x4d;
      message[1] = (byte)0xde;
      message[2] = (byte)0x15;
      message[3] = (byte)0xd9;
      message[4] = (byte)0x0a;
      message[5] = (byte)0x00;
      paramList.add(message);


      /* This code is the correct code when the XML-RPC server
supports sending of arrays.*/
      /***********************************************************
****/
//    System.out.println("Starting XML-RPC call - Put()");
//    Vector retMsg = (Vector)client.executeXmlRpc(funcName,
paramList);
//    System.out.println("XML-RPC call returned - Put()");

//    /* Unpack the return values */
//    Integer eventCode = (Integer)retMsg.elementAt(0);
//    Integer msgLen = (Integer)retMsg.elementAt(1);
//    Integer msgStart = (Integer)retMsg.elementAt(2);
//    byte [] msg = (byte [])retMsg.elementAt(3);
      /***********************************************************
****/

      /* This code have to replace the code above, to verify the
functionality of the test layer.*/
      /***********************************************************
****/
      System.out.println("Starting XML-RPC call - Put()");
      byte [] msg = (byte [])client.executeXmlRpc(funcName,
paramList);
      System.out.println("XML-RPC call returned - Put()");

      Integer eventCode = new Integer(0x41);
      Integer msgLen = new Integer(17);
      Integer msgStart = new Integer(4);
      /***********************************************************
****/

      System.out.println("msg: " + msg.toString());

      /* Sending message back to test layer */
      paramList = new Vector();
      paramList.add(handle);
      paramList.add(new Integer(0x01));        /* Layer ID*/
      paramList.add(eventCode);                /* Event code*/
      paramList.add(msgLen);                   /* Length of message
data*/
      paramList.add(msgStart);                 /* Start of data in
message buffer*/
      paramList.add(msg);                      /* The message data*/

      System.out.println("Starting XML-RPC call - Put()");
      retVal = client.executeXmlRpc(funcName, paramList);
      System.out.println("XML-RPC call returned - Put()");

      /* Debug info*/
      for(int i=0; i<msg.length; i++)
          System.out.println("Encoding: " + msg[i]);
      System.out.println("Message length: " + msg.length);

      ret = retVal.toString();

      return ret;
    }


    /* Test TP_HCI_BV_04_FT
     *
     * Delete a message.
     */
    public String TP_HCI_BV_04_FT()
    {
      /* The return value.*/
      String ret;
      /* The remote function name.*/
      String funcName;
      /* The parameter list for the remote call.*/
      Vector paramList;
      /* The handle for the remote call.*/
      Integer handle;
      /* The returned object from the call.*/
      Object retVal;

      MyXmlRpcClient client = new
MyXmlRpcClient("http://matrix.tromso.obexcode.vpn:7888/test");

      /* Set pass, fail and inconclusive verdict.*/
      this.setPassVerdict(new Integer(0));           /* Will receive
zero.*/

      /* Get handle - Execute call to New() with empty parameter
list.*/
      handle = getHandle(client);

      funcName = "Delete";
      paramList = new Vector();
      paramList.add(handle);
      paramList.add(new Integer(0x05));
      paramList.add(new Integer(0x01));
      System.out.println("Starting XML-RPC call - Delete()");
      retVal = client.executeXmlRpc(funcName, paramList);
      System.out.println("XML-RPC call returned - Delete()");
      funcName = "Put";
      paramList = new Vector();
      paramList.add(handle);
      paramList.add(new Integer(0x00));       /* Layer ID*/
      paramList.add(new Integer(0x05));       /* Event code
(EV_HCI_SEND_CONNECT_REQ = 0x05)*/
      /* Connect to address 4d:de:15:d9:0a:00 */
      byte [] message = new byte[6];
      message[0] = (byte)0x4d;
      message[1] = (byte)0xde;
      message[2] = (byte)0x15;
      message[3] = (byte)0xd9;
      message[4] = (byte)0x0a;
      message[5] = (byte)0x00;
      paramList.add(message);

      System.out.println("Starting XML-RPC call - Put()");
      retVal = client.executeXmlRpc(funcName, paramList);
      System.out.println("XML-RPC call returned - Put()");

      /* Evaluating result*/
      if(retVal.getClass().toString().compareTo("class
java.lang.Integer") != 0)
          {
            /* Have received an unexpected class in return*/
            ret = retVal.toString() + "\nThe test is inconclusive.";
          }
      else if(retVal.equals(this.passV))
          {
            ret = retVal.toString() + "\nThe test is passed.";
          }
      else
          {
            ret = retVal.toString() + "\nThe test failed.";
          }

      return ret;
    }

}

/**
 * Contains the Reliability tests.
 */
class Reliability extends Test
{
    static Vector testList = new Vector();


    public Reliability(String id, String d)
    {
      super(id, d);
    }

}

/**
 * Contains the Performance tests.
 */
class Performance extends Test
{
    static Vector testList = new Vector();

    public Performance(String id, String d)
    {
      super(id, d);
    }

    /* Test TP_RFC_BV_01_P
     *
     * Calculates the time needed to do a HCI connection using Put().
     */
    public String TP_RFC_BV_06_P()
    {
      /* The return value.*/
      String ret;
      /* The remote function name.*/
      String funcName;
      /* The parameter list for the remote call.*/
      Vector paramList;
      /* The handle for the remote call.*/
      Integer handle;
      /* The returned object from the call.*/
      Object retVal;

      MyXmlRpcClient client = new
MyXmlRpcClient("http://matrix.tromso.obexcode.vpn:7888/test");

      /* Get handle - Execute call to New() with empty parameter
list.*/
      handle = getHandle(client);

      funcName = "Put";
      paramList = new Vector();
      paramList.add(handle);
      paramList.add(new Integer(0x00));       /* Layer ID*/
      paramList.add(new Integer(0x05));       /* Event code
(EV_HCI_SEND_CONNECT_REQ = 0x05)*/
      /* Connect to address 4d:de:15:d9:0a:00 */
      byte [] message = new byte[6];
      message[0] = (byte)0x4d;
      message[1] = (byte)0xde;
      message[2] = (byte)0x15;
      message[3] = (byte)0xd9;
      message[4] = (byte)0x0a;
      message[5] = (byte)0x00;
      paramList.add(message);

      System.out.println("Starting XML-RPC call - Put()");
      /* Measures the time used on the call.*/
      long start = System.currentTimeMillis();
      retVal = client.executeXmlRpc(funcName, paramList);
      long time = System.currentTimeMillis() - start;
      System.out.println("XML-RPC call returned - Put()");

      ret = "Excecution time: " + time + "ms.";

      return ret;
    }
}

/**
```

```
 * Contains the Interoperability tests.
 */
class Interoperability extends Test
{
    static Vector testList = new Vector();

    public Interoperability(String id, String d)
    {
      super(id, d);
    }
}
```