Computer Science

Christer Hermansson

Fredrik Malmqvist

regTUX

A Test Tool for Command Line Interface Regression Test

Bachelor's Project

2004 : 23

regTUX

A Test Tool for Command Line Interface Regression Test

Christer Hermansson

Fredrik Malmqvist

This report is submitted in partial fulfillment of the requirements for the Bachelor's degree in Computer Science. All material in this report which is not my own work has been identified and no material is included for which a degree has previously been conferred.

Christer Hermansson

Fredrik Malmqvist

Approved, 2004-06-03

Advisor: Katarina Asplund

Examiner: Donald F. Ross

Abstract

Ericsson AB develops GSN nodes, which target the GPRS and UMTS networks. In order to facilitate the development process, they requested a test tool built. This tool, later named regTUX, should be used in order to check input/output consistency on the nodes in question. Having the possibility to run readily made test files via the tool would increase the efficiency of the implementation process substantially. The purpose of these files is to inform the developers of any changes within the node in a fast and controlled way. It was decided that the tool should be built in Java with the purpose of making it as modular as possible. Having a stable tool that would not cause problems on the nodes was also a part of the requirements from Ericsson. The thesis presents the construction of regTUX, as well as a deeper discussion regarding the usage of the tool and its interface. It also accounts for the primary concerns that arose during the implementation process with special regards to Java functionality. Furthermore, the thesis discusses the creation of tests for a specific area of Ericsson's business, namely Performance Monitoring. A presentation of the test suites constructed follows a brief introduction to the area itself. The main focus of the thesis, however, is the construction of the regTUX tool. This tool enables Ericsson to run the pre-constructed tests within their internally developed environments, as well as on the actual nodes. The tool can also easily be distributed to third-party software developers.

Contents

List of Figures

1 Introduction

Ericsson AB develops so-called GSN nodes, which target both the GPRS[1][6] and UMTS[2][12] networks. Via these nodes, Ericsson is able to run its implementations of the mentioned networks. In order to facilitate the development of such nodes, Ericsson AB has internally developed a Command Line Interface (CLI) to be run in their Solaris environment. Being a part of the nodes themselves, this CLI has been primarily developed for the potential customers of the nodes. This CLI resembles the shell prompt in a normal UNIX environment, with the difference that the CLI is configured for different commands than the normal shell. These commands manipulate data within the various nodes in different ways. When inputting data into their nodes via the CLI, the node returns output of different kinds. The developers at Ericsson would like to be able to notice any change in behavior within these nodes in an easy fashion. They therefore request a test tool built that has the ability to record input and result from the user. We named this tool regTUX (REGression Tool for UniX). They would like the tool to be able to inform them about any changes that might have occurred after a system upgrade. It is also required of the tool that it produces documentation that is easily read by people involved in the development process, which is why any produced files should adhere to XML-standards[5]. Because of the possibly vast amount of data that the program should be able to handle, Ericsson has also requested that the program should take its information from readily made files. Although they would like the possibility to input commands one at the time, it is vital for their every day use that data can be inputted via files. These files also enable the company to build test suites for a particular area of their production. Using these files to check the consistency of their nodes should give them an easier development process, as far as debugging goes.

Ericsson also wished that a test suite for one of their development areas, Performance Monitoring (PM)[2], should be constructed. This should be done by using the program in order to run consistency tests directly on their target environment in Gothenburg. Using a specification of Performance Monitoring, they asked us to build this test suite according to what we feel are the areas within PM that ought to be included in such a consistency check.

[1] General Packet Radio Service. Standard for sending data via packet-switched mobile networks.
[2] Universal Mobile Telecommunications System. European standard for the commonly recognized 3G networks.

This task enabled us to use our test tool in a practical way to get a better feeling of how it will work in real life. Thus, this task also provided us with a hint of how the test tool could be improved. For instance, the different printouts could be refined, which greatly increased the tool's functionality. At first, a number of test suites had to be constructed in order to accomplish the task. PM is a wide area and there was not enough time for creating test suites for the whole area of PM. But that was not the purpose of this task even if Ericsson would have appreciated it.

Each packet switching node within the GPRS and UMTS networks are configured by the above mentioned CLI. Each of the available CLI commands corresponds to a given syntax consisting of a command name, a parameter name[3] as well as a parameter value[4]. When inputting these into the system, the system outputs either the requested information or an error message. However, when the system is upgraded, the CLI might change slightly with regards to input/output consistency. This is a problem not only for Ericsson, but also for companies developing third-party software to be run on the nodes in question. The tool should be able to reflect possible changes via an input file consisting of commands to be run on the nodes. The tool will then inform the user of any and all changes related to the commands in said file. This outputted file will be useful when debugging the application, as it is readable in a normal text editor. Being so, it enables the developer to find inconsistencies in the program in an easy fashion.

This thesis consists of seven Sections and two appendices. After this introduction follows Section 2, which discusses the purpose with the thesis work. Section 3 is a brief overview of Performance Monitoring, which is the area of the tests mentioned in this Section. Section 4 presents all prerequisites and requirements that Ericsson put forward during the course of the project. Following that Section is a detailed discussion of the structure of the tool, called *Program Construction*, which is denoted Section 5. Section 6 introduces the program interface and displays examples of how the program looks when running. Finally, Section 7 is called *Implementation Process*, and it outlines the work carried out in order to complete the tool.

Appendix A contains a class diagram of all the classes and the following appendix, Appendix B, shows the usage text for the program.

[3] An instruction to the tool denoting the task to be performed. See PARAMETERS in appendix B.
[4] A file required by the task to be performed. See FILETYPES in appendix B.

2 Purpose

The purpose of the thesis work was to build a test tool for command line interface regression test. The first part of this Section describes the purpose of the tool itself, while the following Section accounts for the purpose of the test suites we were asked to build.

2.1 Test Tool

The purpose of the test tool has been briefly described in the introduction. The introduction states that the tool should be developed in order to facilitate the development of a GPRS or UMTS node via Ericsson AB's internally developed Command Line Interface. Ericsson felt that it would be a useful part of their development to be able to establish if their CLI had changed between implementations. By using the tool to ensure this, Ericsson can also make certain that the underlying functionality of the nodes has not been altered. In order to do this, they requested that a tool be developed that could compare CLI-output from a given input.

Apart from using this tool in order to facilitate their own development, the tool can be used to aid developers of third-party software. Generally speaking, developers of third-party software implement their products expecting very specific output from the nodes in question. Although they would not benefit from the tool in the same way as Ericsson would, they would still find it very useful. While Ericsson plan to use the tool in order to check their implementation, as mentioned in Section 1, a third-party developer would use the tool somewhat differently. Ericsson re-implements parts of their nodes on occasion. They make these changes for a number of reasons; perhaps the most common one is in order to increase the nodes' performance. When such a release reaches the third-party developer, they would like to make sure that the output they expect has not been changed. It is important to note, however, that the third-party software developers would not use the tool themselves. Ericsson would ensure that the necessary information would be available for such developers.

2.2 Test Suites

The test suites mentioned in Section 1 was primarily created in order to familiarize us with the usage of the program itself. Constructing these tests enabled us to understand more about the program's interface, and gave us the opportunity to optimize the interface itself.

A secondary reason for creating these test files was to give Ericsson a small test suite to work with, hence allowing them to evaluate the tool upon its completion. Regardless of the fact that the test themselves would most likely be less important for Ericsson's development, they would carry significance when assessing the importance of the program to their development. Below follows an example of a command file that carries out a test upon the node. The resulting test file has been checked according to a test specification, and the test file can be used in a consistency check. This test file does a simple check of which commands are available to the user of the node interface. The origins of these tests can be found in an internal document at Ericsson[3], which specifies tests to be run upon the nodes in question.

```
# NDPGSN_1_0_R2E02_040414201354
# Test_id: PXM7/A1.1

# Purpose:
# Check that the Node doesn't have any set command
# for Measurement type. According to the requirement
# WPP-PM-007, it must not be possible to edit the
# settings for a measurement type.

# Expected result:
# A list of all commands, sorted by application.
# This list must not include any set commands for
# measurement types.

# Received result:
# Expected. Test file adheres to standard.

gsh list_sort_cmds
```

Figure 2:1: A command file for test suite.

A third reason for creating the test files also arose at the end of the project. Since Ericsson requested of us that we hold a presentation of the tool in early June, we also needed some data to use in this presentation. Although the presentation does not constitute a physical requirement on the tool itself, it is nonetheless an important part of the development process. It is important that we are able to give a presentation to Ericsson's staff, thereby enabling them to put the tool to use in their development process.

3 Background

This Section gives an introduction to Performance Monitoring, the area for which Ericsson required the test suites to be built. Although Ericsson could have chosen almost any area for which the tests should be constructed, they decided to choose Performance Monitoring. We therefore decided to give a brief introduction of the chosen area in this Section.

3.1 Performance Monitoring

Ericsson is giving its operators support to monitor the performance of a packet switching node, which means to set up measurement jobs on different counters (named gauges and counters) in the node. When a measurement job is active the counter values of the measurement will be collected for survey. All types of measurement jobs can be configured via the Command Line Interface (CLI), and from that interface operators can do the following:

- List all measurement types
- Modify control values for measurement types
- Create, delete, list and copy measurement jobs
- Suspend and resume measurement jobs
- Demand logging of counter's and gauge's values to file

A graphical user interface, called PXM[2] also exists for configuring the measurements. However, any work on this graphical user interface is out of scope of this thesis. Consequently, we were not asked to implement any graphical user interface on the test tool itself.

By using the counters in this node, the user is given the possibility to monitor all traffic on the node. Performance Monitoring provides the possibility to configure in order to achieve

optimal performance. When setting up counters, the user is left with a vast amount of available statistical monitoring. The user has the option of setting alarms to go off when, for instance, values exceed predefined boundaries. The options for presenting the statistics to the user are also substantial, thus providing easily overviewed data regarding the node's performance.

4 Prerequisites and Requirements

In order for Ericsson to find the tool useful it had to conform to a number of requirements. As is common when developing a product, these requirements were given to us in a formal request from the company. This Section of the report aims to outline these requirements in a fashion that is more easily overviewed.

4.1 Requirement Specification

This Section presents the requirements that Ericsson had on the tool. Each subsection discusses one requirement, as well as states the formal requirement from Ericsson. This formal requirement is stated exactly as it was stated in the original requirement specification.

4.1.1 Requirement One

Firstly, Ericsson had to stipulate which target environment the tool was to be run upon. This formed into requirement one:

> The tool shall run in a Solaris environment. The tool shall behave as a normal UNIX command.

This requirement forced us to investigate the target environment at TietoEnator, where we were to perform the development process.

4.1.2 Requirement Two

Because of the vast amount of data that is to be processed by the tool, it is important that the user has the possibility to input data via pre-constructed files. This also enables the company to utilize readily made test suites, whose purpose is to run conforming tests on a specific part of the developed system.

> The tool shall use a file as input. The file shall include command name, parameter name, parameter value, output and return code. The file shall be possible to include comments. The input file shall be possible to view and edit with a normal text editor.

This file, although easily overviewed, is not necessarily easy to construct. Therefore, it was desired from the start that the program could construct such a file, given the commands that were to be included in the file. At this point, it became crucial to distinguish between a *test file* and a *command file*. It was stipulated that a test file should denote a file, normally created by the program, containing all of the information mentioned in the second requirement. This file should adhere to common XML-standard, in order for a user to easily understand it. A command file, however, was decided to represent a file containing only commands that were to be executed by the test tool. In order to facilitate the reading of these files, we also agreed to include support for comments as well as for blank lines.

4.1.3 Requirement Three

Apart from the possibility to use a test file as input, as in Section 4.1.2, Ericsson asked for the possibility to input one command at the time, as one might do at a normal UNIX-prompt. This feature should also include the possibility to store these commands in a command file, in the requirement called *text file*.

> The tool shall be able to create an initial input file. A text file containing CLI commands shall be a possible source for creating an input file. An optional feature might be to "record" a command sequence from the CLI interface and generate an input file.

Meeting this requirement meant that the tool was to be able to build a test file from inputted commands, as previously stated in Section 4.1.2.

4.1.4 Requirement Four

It is also important for Ericsson to stipulate the conditions for running the program. Since it is supposed to be used in order to test their internally developed nodes, they are interested in very specific information. When using the tool to test an application, the developer is

interested in knowing what, if anything, went wrong, and why. In order to assist the development process, and make it a smoother one, it is vital that the tool presents its information in a useful way. At the earlier stages of this project, it was not entirely clear how this was to be done. However, after communicating closely with people at Ericsson, and after having tested the program on their nodes, we feel that we now know how the developers want this information presented to them.

The tool shall evaluate each CLI command output with the output specified in the input file. If the evaluation results in a difference the user shall be notified with the test case and difference.

4.1.5 Requirement Five

Standard usage of the tool is to run the tool with a test file as input. The tool extracts the commands from the file, executes these in the target environment, and compares the results with the file. The user has little control over the program flow at this point in the process. Ericsson therefore requested the possibility to run the program in what they call *interactive mode*. Interactive mode means that the user can administrate the handling of the files themselves. If the program finds a difference somewhere in the program, the user is alerted in the usual way. What differs, however, is that the tool prompts for information on how to continue. This mode enables developers to filter out normal deviations from the output in an easy fashion.

The tool shall be possible to run in an interactive mode. By interactive mode means that the user shall be able to take a decision how to continue at every difference. The tool shall not run in interactive mode as default.

4.1.6 Requirement Six

The requirements in Section 4.1.1 to 4.1.5 were the formal requirements on the tool from the beginning. However, Ericsson had one more requirement on the tool. Since this tool is to be run on GPRS-nodes both internally as well as externally, they required that the tool be robust.

The tool could under no circumstances cause a crash on any of the nodes. This meant that we had to implement the tool with the aspect of security from the very beginning.

The tool shall be easy to use and maintain. The tool shall be robust and give informative and relevant error messages if something goes wrong.

4.1.7 Informal Requirements

Apart from the requirements in Section 4.1.1 to 4.1.6, Ericsson only demanded that the source code, the design specification, and a user manual should be delivered to them at the completion of the project.

4.2 Language Requirements

As far as programming language goes, Ericsson had no specific requirements. They only required that the tool could be run on their nodes. Instead they requested that we recommended which programming language to use. We discussed different options regarding which language was best suited for this particular assignment. We broke the discussion down to four different options. The reason for choosing between the four following languages was that we felt familiar with C, C++ and Java, while Erlang[1] is an internal Ericsson language, which Ericsson recommended that we considered using. However, our recommendations were as follows:

1. *Erlang:* We felt that the time required in order to fully comprehend the language itself would be far too great. Thus, we felt that it would be impossible to perform the given task within the given time frame if we chose this particular language.

2. *C:* The biggest advantage with C is that UNIX is written in this language, enabling us to get closer to the operating system. We believed, however, that we would get this advantage even if we used C++.

3. *C++:* Has the advantage mentioned above, plus the added bonus that it gives us an easier compilation process compared to Java, which is an interpreted

language. Furthermore, C++ enables us to benefit from object-oriented programming, allowing us to organize the system in a better way. The downside, nevertheless, is that we would have to build the system completely from the ground up.

4. *Java:* This language gives us a vast amount of pre-programmed code. Thus, Java presents us with the possibility to build the system specified, in a faster and more secure way. The major downside with Java is that we would have to find a way to compile the system for the specific operating system. The nodes do not contain a Java Virtual Machine, thus forcing us to compile for the target environment only.

Based on the discussion above, we chose to recommend that Java be used for the development of the tool. Our primary concern was that the nodes do not contain a Java Virtual Machine, and we did not now whether we could find a native compiler to use. Ericsson gave us permission to start the development in Java, well aware of the difficulties we had presented.

After a week of implementation, Ericsson decided to install a Java-plug-in[8] on the nodes, which gave us the possibility to run the tool on the nodes after all. Unfortunately, the plug-in was limited to version 1.3, which gave us fewer API-methods to work with than the most recent API, version 1.4.2[9]. We learned to deal with this situation and decided to use Java as the primary language for building the tool. We have used the API specification for 1.4.2 as the primary source of information. However, we have limited us to the use of the methods that existed in 1.3.

5 Program Construction

This Section presents the way in which we decided to construct the test tool. It accounts for the different options available to the user, as well as the design itself. The first part of this Section discusses the programming environment, and the tools used in the development process. Subsection 2 is an overview of the program design process, after which a Section on program usage follows.

As mentioned in Section 4.1, there are two types of files in this system, command files and test files. The command parameters allowed in the tool are all in some way connected to which of these files are affected. To make it easier to overview the program it has been separated into five different modes of usage. These modes are the same ones that accounts for the interface description in Section 6. Each of these modes conforms to one parameter in the usage text, which can be found in appendix B.

Following the description of the different modes is an overview of the Java classes that constitute the tool. Each class is only briefly described, while a more detailed description of how the classes are implemented can be found in appendix A. In appendix B a more detailed description of how to use the program can be found. This appendix accounts for the usage of the tool, for instance which parameters can be used with which commands. This usage text is also incorporated into the program, in order to give the user an quick guide to program usage.

5.1 Programming Environment

It was not specified in the requirement specification which tools to use, thus we had a wide choice when we were to choose the programming environment. The tools we have used to complete this project are the following:

- Sun Solaris 5.8 (Operating System)
- Xemacs 21.4.15 (Source Code Editor)[14]
- NetBeans IDE 3.6 (Integrated Development Environment)[10]
- Rational Rose Enterprise Edition (UML Design Tool)[11]

- Microsoft Word (Documentation)

5.2 Program Design

Once the first analysis of the task was finished we generalized it into classes. The most important parts of the system, such as command executions, command comparison and an input handler were made into separate classes, namely `CmdExecute`, `CompareContainer` and `InputHandler`.

Another vital section of the system is file handling and due to the two types of files which are intended to be a part of the system, we generalized these types into two classes, namely `TestFileReader` and `InputFileReader`. To construct test files we suggested a class called `TestFileWriter`.

In order to establish a program, which would not crash during runtime, we generalized a class `Interruptor`. This class would be used for termination purposes. In Figure 5.1, a view over the program at this point in the implementation process is displayed.

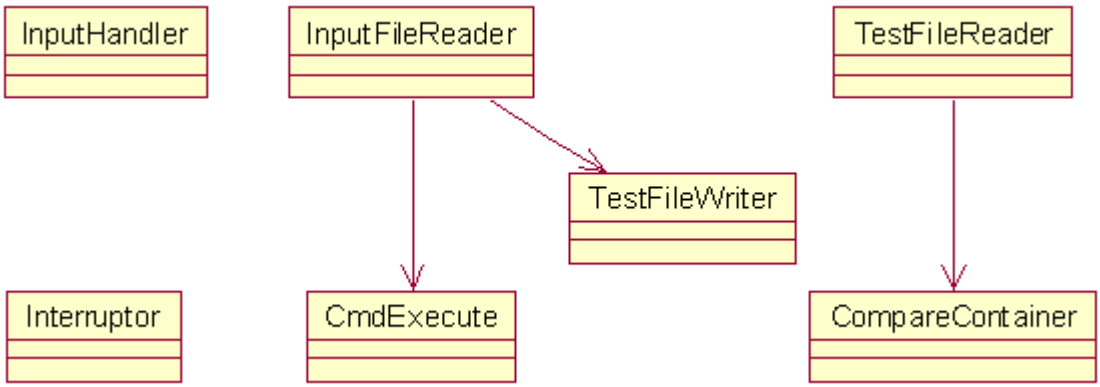


Figure 5:1: Snapshot of the implementation process.

During realization of the classes, we considered to construct a class that would divide responsibility for the classes using file checking to new help classes. Our consideration resulted in a class called `FileChecker`.

We suggested to implement a feature were a user can execute single commands in a shell. This suggestion was accepted, implemented, and became dependent of the same classes as `InputFileReader`.

Problems with the returned output from executions, described in Section 7, forced us to implement a separate thread in order to handle the given execution results.

Late in the implementation process each class printouts were moved to a new class called `Print`. This approach made the printouts much easier to manage, which turned out to be necessary, as is described in Section 7.

The final view of the design is found in Appendix A.

5.3 Class description

This Section presents an overview of the different classes that constitute the program. Each class is described briefly, and its function is stated.

5.3.1 TestTool

Class `TestTool` consists of the main function, which initializes the class InputHandler (5.3.2). The class also contains a method for terminating the allocated resources, thus cleaning up processes.

5.3.2 InputHandler

The class InputHandler handles any input from the user and switches the execution flow according to the input. All modes described in Section 5.5 are initialized by this class. All logic concerning mode 5.5.2 through 5.5.5 is also found here. While the other classes contain methods that enables them to perform their specific task, this class governs the execution flow in accordance with the user's choice. All tasks that have to be performed in order for the program to do its job are carried out by the logic in this class.

5.3.3 CmdListen

Class `CmdListen` writes a prompt in which the user can execute and record a single command through using `CmdExecute` and `TestFileWriter`.

5.3.4 InputFileReader

The class `InputFileReader` reads a command file consisting of CLI commands to be executed. It can also compare the result of this input with the input stored in a test file.

5.3.5 TestFileReader

Class `TestFileReader` simply reads a test file and compares different executions in the file with the current execution.

5.3.6 FileChecker

This class, `FileChecker`, prompts the user to take action if the file already exists. Also when the execution of a command proves inconsistent, it provides the user with the possibility to store differences in a new test file.

5.3.7 TestFileWriter

Class `TestFileWriter` writes the result of an execution into test files in well-formed XML.

5.3.8 CmdExecute

`CmdExecute` is the class that executes commands sent from `CmdListen` and `InputFileReader`. `CmdExecute` is also responsible for writing output to a test file after each executed command. This is done via `TestFileWriter`.

5.3.9 CompareContainer

`CompareContainer` assists `TestFileReader` in the comparison phase; this is where the actual comparison takes place.

5.3.10 Interruptor

Class `Interruptor` is a separate thread, making sure that the executions terminates safely within a period of time.

5.3.11 OutputStreamHandler

The class `OutputStreamHandler` is a separate thread, taking care of the output the executions produce.

5.3.12 Config

Class `Config` makes it possible for a user to change specific values such as the maximum time for executions to run.

5.3.13 Print

The class named `Print` serves the system with appropriate printouts.

5.4 Program Usage

Usage of the tool varies slightly depending on what kind of information one would like to receive from the tool itself. However, two main areas of consistency checks can be seen.

Firstly, one can use the tool in order to find out which output is generated given a collection of inputs. In order to do this, the user provides the tool with what is called a command file[5]. This file is run by the tool, which produces a so-called test file[6]. Figure 3.1 displays an example of a test file and a command file, although the commands used within the nodes are more complicated than the ones in the example.

```
which java
sleep 2
```

Example of a command file.

```
¤testFile¤
#Do not edit above this
line.
<COMMAND>
<INPUT>which java</INPUT>
<RETCODE>0</RETCODE>
<OUTPUT>
/usr/bin/java
</OUTPUT>
</COMMAND>

<COMMAND>
<INPUT>sleep 2</INPUT>
<RETCODE>0</RETCODE>
<OUTPUT>
</OUTPUT>
</COMMAND>
```

Example of a corresponding test file.

Figure 5:2: Example of a command file and a test file.

[5] A file consisting of command lines that are to be run by the tool.
[6] A file, in XML standard, containing output and return code received from a command execution.

Using a command file to build a test file is useful primarily for building test suites for different areas of development. Nevertheless, it also has a place at developers of third-party applications, who can use this feature in order to find out if the interface has changed in any way concerning the specific commands they use in their applications.

Secondly, the tool has another main feature, mostly used in the development of the nodes. This feature is to be able to check a test file for consistency. When Ericsson has built test suites, the company can at a later time use this test file to check its consistency. When giving the tool a test file as input, the tool runs all commands in the file and checks the returned output against the output in the file. If any differences are found, the user is notified of these in a useful manner.

Ericsson spends much time on testing implementations. A simple code change might not only affect the module whose code has been changed. Therefore they have to go through related test suites for the product once again for any kind of version update. This is why they requested a test tool that will make development more time efficient.

5.5 Program Modes

This Section describes the responsibilities of the five modes in the program. For a more formal description, see appendix B. Also note that some of the modes can be run in interactive mode. Interactive means that a separate file is created alongside the consistency check. If any inconsistencies are found, the user will have the choice to save this new test file, which has the same name as the executed file, with the suffix _versionX. X is the lowest number, starting at 0, which denotes a file name not already in use.

5.5.1 Mode One: Record Single Commands[7]

The first mode in the tool, to record single commands, enables the user to execute single commands in run-time from a shell prompt. Every command that is written will be recorded, which means that the input, along with the results, will be written to a test file. Even if the command does not exist, the program will handle the error message. The user inputs a file to write executions to when running the command as a parameter value. This test file will after execution contain the given input, the return code and the received output. If the user specifies

an already existing file to write to, the program prompts the user for a decision on how this situation should be handled. The user's options are to either overwrite the file, append the results to the file, or simply to exit the tool. In order to make the file easy to read and easy to maintain it is structured in well-formed XML.

5.5.2 Mode Two: Record Lists of Commands[8]

Mode two, record lists of commands, allows a user to build a test suite. Given a file containing a collection of input, the tool executes one command at a time, in the same fashion as in mode 1. The results are then written to a test file specified by the user. This mode has the possibility of taking more than one command file as input, thus building a test file with all the input from the files.

5.5.3 Mode Three: Compare Results[9]

When the implementation of a node has been altered, Ericsson would like to check their nodes for inconsistencies, which is why this mode, compare results, was implemented. In order to check the consistency from two different executions with the same inputted command an already generated test file is needed. The test file is traversed and each command is executed once again. The new output is compared with the old output and if the executions are found to be inconsistent, the program displays the differences on the screen. By default the program just displays any inconsistency, but the user also has the possibility of running this mode in interactive mode (see Section 5.5).

5.5.4 Mode Four: Compare Results with Command File[10]

If the user wants to check parts of a test files, mode four is used. The mode compares the results from executing commands from a command file and compares them with what is stored in a test file. This mode resembles the mode in Section 5.5.3 somewhat, with the difference that the tool takes a command file and compares the execution with an already existing test file. This mode can also be run in interactive mode.

---

[7] Conforms to parameter `-record`.
[8] Conforms to parameter `-buildfile`.
[9] Conforms to parameter `-check`.
[10] Conforms to parameter `-compare`.

5.5.5 Mode Five: Printing Input from Test File[11]

The last mode is a simple printout function, supporting the user with the possibility to display all input from a generated test file. This feature is needed due to the vast amount of information that can be stored in a test file. Listing the input found in such a file can be interesting for debugging purposes.

[11] Conforms to parameter `-printinput`.

6 Tool Interface

This Section refers to the requirements from Section 4.1 in order to make the reader familiar with the program's interface. The Section is divided into three main parts: Normal function, control structures and parameter error handling. The control structure Section also discusses the only program option, namely to run the tool in interactive mode. Note that the tool uses a shell script called `tt` in order to launch the program into the Java Virtual Machine (JVM). This shell script simply starts the tool itself by taking the parameters inputted and calling the class `TestTool`. The script has only the following two lines of code:

```
#!/bin/sh
java -jar tt.jar $@
```

Figure 6:1: Test tool shell script.

The design of the program, which enables this interface to function, is further presented in Section 5. The five modes presented in this chapter are also further explained in Section 5.5.

6.1 Normal Function

As all software, the tool expects the user to enter input in a certain way. If the user conforms to these rules a specific result can be expected. If the user does not, the program handles resulting errors. This Section discusses those cases in which the user has entered the input in accordance with the usage text. In this Section it is also presupposed that there are no conflicts with existing files. In order to read about such conflicts, see Section 6.2.

6.1.1 Task One: Build Test Files via Prompt

The first task required of the program can be derived from the latter part of requirement three in Section 4.1.3.

An optional feature might be to “record” a command sequence from the CLI interface and generate an input file.

This requirement leads to the first task for the tool, to record the result of a single command. The user enables this by inputting the following command at the prompt:

```
tt -record fileName
```

This command presents the user with a prompt in which it is possible to enter commands that are to be executed. The parameter value `fileName` denotes the test file where the results are to be saved. In accordance with Ericsson’s wishes, the result of the command is printed both to a test file, and to the screen. Printing to the screen enables the user to see if something is inconsistent in an easier fashion. Otherwise the user would have had to open the test file after each command. Below you can find an example of a user running this task. The user executes one command, `which java`, and then exits the program by typing `exit` at the prompt.

```
seksux025+94> ./tt -record fileName

regTUX - Version 1.0

Input commands to execute, 'exit' terminates test tool.
regTUX>which java
<OUTPUT>
        /usr/bin/java
</OUTPUT>
regTUX>exit
bye...

seksux025+95>
```

Figure 6:2: Task one interface, build test file via prompt.

6.1.2 Task Two: Build Test Files from Command Files

The second task required of the program can be derived from the first part of requirement three in Section 4.1.3.

> A text file containing CLI commands shall be a possible source for creating an input file.

In order to accommodate this requirement, a second mode had to be created. This mode, record a list of commands, enables the user to input a command file into the program in order to build a test file. The program executes all commands in the file and writes the result to a test file. The syntax for starting this mode is:

```
tt -buildfile testFile commandFiles
```

When using this command, the user is required to supply at least two parameters. The first one denotes the test file that the results are to be written to, while the following parameter values represent the command files containing all the commands to be executed. After conferring with Ericsson it was decided that the result of each execution should not be written to the screen. This decision was reached because the command files can potentially become rather large, and the program execution would be very hard to monitor in such cases. In the example below the user inputs two command files with a number of commands into the tool.

```
seksux025+95> ./tt -buildfile testFile cFile1 cFile2

regTUX - Version 1.0

**************************************
Processing: [ cFile1 ]
File: [ cFile1 ] completed

**************************************
Processing: [ cFile2 ]
File: [ cFile2 ] completed

seksux025+96>
```

Figure 6:3: Task two interface, build test file from command file.

When handling files, the program has to perform checks on these files. For instance, it is not desirable that a file is overwritten without the users consent. For more information on this see Section 6.2, Control Structures.

6.1.3 Task Three: Check Test File for Inconsistency

The third task, as well as the fourth, required of the program can be derived from the requirement four in Section 4.1.4. This Section presents the third task, namely to check an already existing test file for inconsistencies.

> The tool shall evaluate each CLI command output with the output specified in the input file. If the evaluation results in a difference the user shall be notified with the test case and difference. file.

In order to do this, the program executes all input found in a test file. This collection is the same as the result of printing all input from a file, presented in Section 6.1.5. After the execution, the tool compares the received result with the result that has previously been stored in the file. To activate this mode the user types:

```
tt -check fileNames
```

The parameter values in this command are test files the user wishes to check. In order to facilitate for the user regarding the messages that appear on the screen, a new tag was introduced. This inconsistency tag only exists on-screen. An example can be seen below.

In the example the user checks two files for inconsistencies. The first file is consistent, while the other displays one output inconsistency and one return code inconsistency. After conferring with Ericsson, we reached the conclusion that the display should be formatted in the way the example illustrates.

```
seksux025+98> ./tt -check fileName1 fileName2

regTUX - Version 1.0


****************************************
Processing: [ fileName1 ]
File: [ fileName1 ] completed

****************************************
Processing: [ fileName2 ]
<TIMEOUT>
        Command: sleep 40
        Timeout: 30 seconds
</TIMEOUT>
<INCONSISTENCY>
        Command:      which java
        Execution:    [0] /usr/bin/java
        Test File:    [0] /usr/bin/jva
        Diff at line:  1
</INCONSISTENCY>
<INCONSISTENCY>
        Command: rolf
        Return Code Exec: [1]
        Return Code File: [0]
</INCONSISTENCY>
File: [ fileName2 ] completed

seksux025+99>
```

Figure 6:4: Task three interface, check test file for inconsistency.

6.1.4 Task Four: Check Test File via Command File

As mentioned in Section 6.1.3, requirement four has resulted in two tasks for the tool to perform. The second task is derived from requirement four, and the fourth task to be discussed in this Section, is to compare the executions from a command file with the results stored in a test file. This task is useful when the user wishes to test a part of a larger system. The tool enables the user to use a test file for a larger system, checking only the desired commands.
Entering the following at the prompt starts a task of this type:

```
tt -compare testFile commandFile
```

The file denoted by the argument `commandFile` is executed in the same manner as when a new test file is built (see Section 6.1.2). Instead of storing the results from this execution in a new test file, the tool compares the result from said execution with the contents of the test file denoted by the argument `testFile`, much like in Section 6.1.3. In the example below the user checks the contents of a command file against that of a test file. This test results in one inconsistency, namely that the command `ls` is inconsistent.

```
seksux025+75> ./tt -compare testFile commandFile

regTUX - Version 1.0


****************************************
Processing: [ commandFile ]
<INCONSISTENCY>
        Command:      ls
        Execution:    [0] testFile
        Test File:    [0] usage
        Diff at line:  8
</INCONSISTENCY>
File: [ commandFile ] completed

seksux025+76>
```

Figure 6:5: Task four interface, check test file via command file.

This output is formatted in a similar way as the output received in Section 6.1.3. This is the case because Ericsson wanted the two to be interchangeable, as far as user interaction is regarded. This was agreed upon because the two tasks are essentially the same, apart from the source of the commands to be executed.

6.1.5 Task Five: Print All Input in Command File

The next task required of the program was not originally part of Ericsson's requirements on the tool. Nevertheless, after testing the program a few times the need for the ability to print all input from a collection of test files arose. Ericsson felt that printing the input would be a useful feature, and it was added to the list of requirements. To print all input in a test file the user enters:

```
tt -printinput testFile
```

When this command is entered, the tool traverses the specified test files looking for the input tag. When an input tag is found, the tool prints the input to the screen via `stdout`. In the example below, the user wishes to print all the input from two test files to the screen.

```
seksux025+32> ./tt -printinput testFile1 testFile2

regTUX - Version 1.0


****************************************
Processing: [ testfile1 ]
<INPUT> head file1
<INPUT> wc file1
<INPUT> noCmd
File: [ testfile1 ] completed

****************************************
Processing: [ testfile2 ]
<INPUT> which java
<INPUT> sleep 40
<INPUT> cat file1
File: [ testfile2 ] completed

seksux025+33>
```

Figure 6:6: Task five interface, print all input in command file

This task does not include executing any of the commands, it simply states which commands exists in the file.

6.1.6 Task Six: Print Usage Manual

The task discussed in this Section was, like the one in Section 6.1.5, not part of the original requirements. Ericsson wanted a usage text printed on-screen, much like the man pages within the UNIX systems. For a complete overview of this usage text see Appendix B. The usage text is invoked by simply entering:

```
tt
```

Ericsson later asked us to make the usage text available when entering a command with a parameter, but no parameter values, as in the example found below.

```
tt -compare
```

Printing usage text in this case has been extended to cover all parameters. The usage text covers all cases discussed in Section 6.1, enabling the user to get a quick overview of how the program is used. The text does not, however, cover any other aspects of the program than the usage.

6.2 Control Structures

This Section covers the control structures found in the program. All of these are connected to some type of file, since files are the central part of the tool. First a description of the interface for interactive mode is presented, after which the more basic controls of the files themselves follow.

6.2.1 Interactive Mode

A further description of the usage of interactive mode can be found either in Section 5.5 or in the usage text in Appendix B. In short, interactive mode enables the user to decide how to act if a test file is found to contain inconsistencies. This option can be invoked when using the modes discussed in Section 6.1.3 or 6.1.4. A user invokes the option by typing one of the two commands:

```
tt -check -interactive testFiles
tt –compare -interactive testFile commandFile
```

If an inconsistency should be found in a test file, a prompt appears where the user has to decide on how to continue. The user has two choices, either to save the result to a new file or not. In the example below the user decides to save the inconsistencies.

```
seksux025+12> ./tt -check –interactive testFile1 testFile2

regTUX - Version 1.0


**************************************
Processing: [ testFile1 ]
<INCONSISTENCY>
        Command:      which java
        Execution:    [0] /usr/bin/java
        Test File:    [0] /usr/bin/jva
        Diff at line:  1
</INCONSISTENCY>

regTUX> File inconsistent: (s)tore, (i)gnore?
regTUX>s
Stored in file: testFile1_version0

**************************************
Processing: [ testFile2 ]
File: [ testFile2 ] completed

seksux025+13>
```

Figure 6:7: Interactive mode interface.

6.2.2 File Incompatibilities

When entering the files that are to be used as values to the parameters, one rule is always to be adhered to. This rule is that if a test file is required the test file is always the first parameter value. There are no circumstances under which a command file precedes a test file. If the user fails to follow this standard, the program would execute every line found in the test file. This behavior is not desirable, which is why a test file always has the following header:

```
¤testFile¤
#Do not edit above this line.
```

Figure 6:8: Test file header.

Having this header enables the program to ensure that the test files and command files entered as parameter values are entered in the right order. If they are not, one of the following texts is displayed:

```
File [ testFile ] not a compatible testFile
Check inputted parameters carefully

File [ cmdFile ] is a testFile
Check inputted parameters carefully
```

6.2.3 File Access

Inputting a file as a parameter value forces the program to access files within the system. This accessing can produce file errors. Any possible operating system error that can arise when trying to access files that actually exists is handled in a fail-safe fashion. Putting aside errors of this kind, the user is only left with two types of problems.

Problem number one is trying to access files that do not exist. The tool then displays the following error:

```
File [ fileName ] does not exist
```

Problem two is the problem with overwriting already existing files. The program always checks whether or not a file already exists before writing to it. If the file does exist the tool displays a prompt with three choices:

```
regTUX> File exists: (o)verwrite, (a)ppend, (e)xit ?
```

The user can then decide whether to overwrite the existing file, to append to it or to simply exit the program.

6.3 Parameter Error Handling

The tool requires that the user inputs the parameters and their values in accordance to the usage text. The class `InputHandler` contains error handling to ensure this behavior. The error messages are of two types; one for the parameters that take an exact number of values, and one for the parameters that take an unspecific number of parameters.

The first type, which is used in for instance Section 6.1.4, is formatted in one of the two following ways:

```
ERROR: Required exactly X parameters
ERROR: Required at least X parameters
```

In this example, X denotes the number of parameters required for the specific task.

The second type, which is used in, for instance, Section 6.1.5 is formatted in the following way:

```
ERROR: Incorrect number of parameters
```

7 Conclusions

During this project we have realized that the choice of programming language we made served our purpose well. The chosen programming language, namely Java, was developed with dedication to platform independency, which has made the language very popular. This popularity is arisen from the fact that developers do not need to adapt their software for the platforms it is supposed to run on. However, the platform independency makes it hard when developers need to interface with the operating system in order to carry out their specific job.

We encountered one major problem that could be derived from the platform independency issue. This occurred when we tried to terminate commands in the UNIX environment. The problem arose from the fact that the Java runtime execution command did not have sufficient buffer space for input and output streams[7]. The solution to this problem was to run commands in a separate thread. Doing so enabled the communication between Java and UNIX to act fine.

By using Java with all its benefits, the implementation process has gone smoothly. But our choice of programming language has also presented us with some challenges. For example, process handling in UNIX was hard to perform due to Java's implementation in this particular area. There where also some known bugs which we had to handle concerning process handling and runtime execution[13]. Nevertheless, if we were to implement the tool all over again, we would have chosen the same programming language. We feel that the advantages of Java greatly outweigh the disadvantages, which is why we would follow the same path.

Since our supervisor at Ericsson did not work in the same city as we did, most of the communication took place via e-mails. With the frame of our work we did not encounter any problems with this kind of communicative approach. We would even like to argue that the communication might have worked even better, because our supervisor and we had to be totally clear from the beginning to avoid excessive e-mail correspondence.

The developed test tool fulfills all requirements Ericsson stated. However, there are some points that we would have done differently had we started over. Firstly, our design was a little imprecise when we started the implementation process. While the program was growing, we made our already constructed classes responsible for more tasks, instead of rewriting our

predefined model. New demands from Ericsson that increased the functionality also made our code increase in complexity. There was especially one class, named `InputHandler` (see 5.3.2.), which was too dependent in the system. Later we refined the class, but we could have saved much time by doing that in an earlier phase. Our experience from this occurrence is to examine and review new tasks more carefully and leave more space for new functionality.

Secondly, we should have put all our printouts in a separate class for easier maintenance earlier. The most important thing, after functionality, is how the program reacts on user input. In this text based test tool we could only inform the user via printouts. We had to rectify our printouts more than once and thereafter we realized that a separate class was to prefer.

Finally, we decided, in conjunction with Ericsson, that our tool's classes should be bundled into an executable JAR-file[4]. This agreement was reached because we felt that such a delivery should vastly improve the usage of the tool. Having all classes nicely bundled would mean that the risk of accidentally removing one or more of the classes would significantly diminish. This idea proved to be the last piece of the puzzle towards making the tool what Ericsson requested, thus completing the thesis work and making our delivery possible.

References

[1] Concurrent Programming in ERLANG, Second Edition - *http://www.erlang.org/download/erlang-book-part1.pdf* - February 2004.

[2] Ericsson AB (Internal Document) - *Functional Specification for WPP Performance Monitoring Support*. -2004.

[3] Ericsson AB (Internal Document) - *Test Specification for Performance Monitoring*. - Revision C, 2004.

[4] Executable Java - *http://homepage.eircom.net/~pugsleypaul/java/jars.htm* - May 2004.

[5] Extensible Markup Language (XML) 1.0 (Second Edition) - *http://www.w3.org/TR/2000/REC-xml-20001006.pdf* - March 2004

[6] General Packet Radio Service - *http://www.gsmworld.com/technology/gprs/intro.shtml* - February 2004.

[7] Java Forum, Process Handling - *http://java.sun.com/developer/qow/archive/135/index.jsp* - March 2004.

[8] Java Plug-in Technology - *http://java.sun.com/products/plugin/* - February 2004.

[9] Java™ 2 Platform, Standard Edition, v 1.4.2 API Specification - *http://java.sun.com/j2se/1.4.2/docs/api/* - February 2004.

[10] NetBeans IDE 3.6 - *http://www.netbeans.org/kb/platform_use.html* - March 2004.

[11] Rational Rose Enterprise Edition -

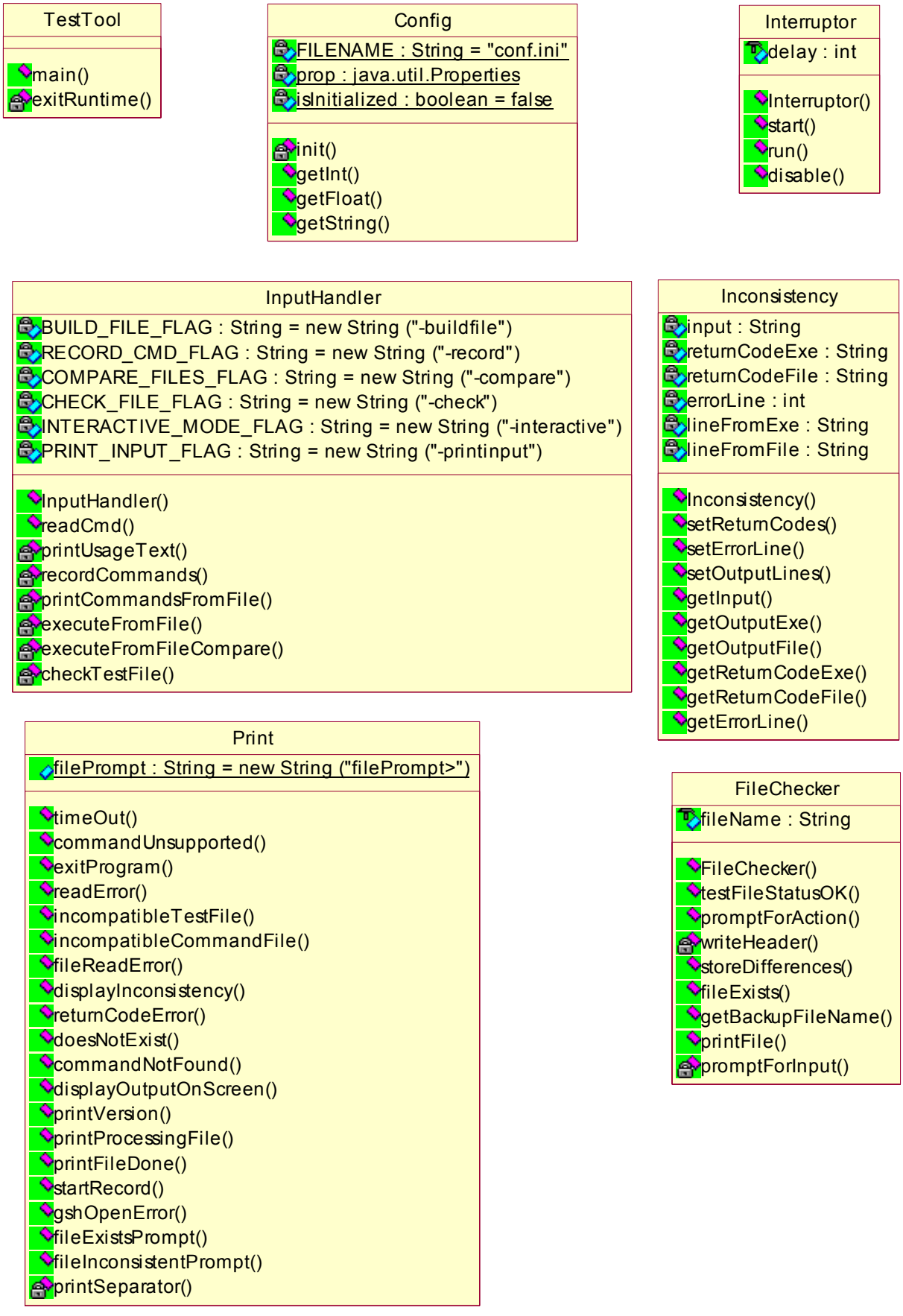
http://www-106.ibm.com/developerworks/java/library/j-jmod0508/ - February 2004.

[12] Universal Mobile Telecommunications System - *http://www.umtsworld.com/technology/overview.htm* - February 2004.

[13] When Runtime.exec() won't - *http://www.javaworld.com/javaworld/jw-12-2000/jw-1229-traps.html* - March 2004.

[14] Xemacs 21.4.15 - *http://www.xemacs.org/Documentation/index.html* - March 2004.

A Class Diagram




CmdExecute
returnCode : int
input : String
commandTimedOut : boolean
CmdExecute()
execute()
getCmd()
getReturnCode()
isSuccessfulExecution()
getStandardOut()
getStandardErr()
commandTimedOut()
storeResult()
setReturnCode()
CmdListen
CmdListen()
startPromptingForInput()
inputIsNewGshShell()
doExecute()
writeTestFile()
promptForInput()
-xCmd
-out
-err
-xCmd
-fileWriter
OutputStreamHandler
output : java.util.Vector
OutputStreamHandler()
run()
readHandlerOutput()
TestFileWriter
fileName : String
TestFileWriter()
writeRootElement()
writeResults()
doNotKeepFile()
getFileName()
writeStartTag()
writeEndTag()
writeInput()
writeReturnCode()
writeOutput()
-fileWriter
InputFileReader
InputFileReader()
getCommand()
getResults()
doExecute()
checkCmd()
writeTestFile()
readFile()
promptForInput()
CompareContainer
input : String
returnCode : String
output : java.util.Vector
CompareContainer()
CompareContainer()
inputEquals()
returnCodeEquals()
outputEquals()
makeInconsistencyContainer()
isConsistent()
partMatches()
removeIgnoreFromLines()
getInput()
getReturnCode()
getOutput()
-currentCommand
-fileReadCommand
TestFileReader
allFileInput : java.util.Vector
testFileName : String
TestFileReader()
setCurrentCommand()
inputFoundInFile()
setCommandFromFile()
compare()
getInputFromFile()
removeInputTags()
findPositionInFile()
getReturnCodeFromFile()
getOutputFromFile()

B Usage Text

```
NAME
       tt - a tool for Command Line Interface (CLI)
       regression test

SYNOPSIS
       tt parameter [option] files

DESCRIPTION
       This program tests the CLI according to the following:
       A command is executed by the program. Upon receiving
       return code 0, the program stores the output from stdout.
       If the return code is greater than 0, stderr is used.
       There are two sorts of files connected with input,
       commandFiles and testFiles. commandFile denotes a file
       containing commands used as input when creating a testFile,
       while testFile is the file in XML-format created by the
       program itself. The testFile is what is used when running
       tests, for instance regression tests.
       A specific file, called conf.ini exists, containing
       configuration settings for the program. It also contains a
       timeout value. This value, entered in seconds, specifies the
       allocated time for the execution of a command. If the command
       has not been executed within this timeframe, it is
       interrupted by the program. The default value is 30 seconds.
       (FILETYPE)* indicates one or more files of this type.
       (FILETYPE) indicates exactly one file of specified type.

FILETYPES
       commandFile
       Denotes a file containing CLI commands that are used in
       order to build a testFile. These commands should conform
       to the commands specified in the CLI documentation.
       testFile
       A file built by the program. Contains CLI input, return
       code, output, parameter name and parameter value. This
       file uses XML style tags in order to facilitate reading.

OPTIONS
       -interactive
              Used as a modifier together with parameter -check
              or parameter -compare. This modifier provides the
              user with a choice to save any difference in the
              comparison to a separate file. The new output is
              saved to a file with the name (testFile)_versionX
              where X is the lowest integer that does not
              denote an already existing file.
              Using this mode causes the program to halt its
              execution at any inconsistency, thus waiting for a
              decision from the user regarding how to continue.
              +Usage: -check -interactive (testFile)*
```

```
            +Usage: -compare -interactive (testFile) (commandFile)

PARAMETERS
      Parameters instruct the program to perform a specific task.
      The parameters below are used together with options and files
      according to the synopsis above. This makes the program
      perform the required tasks.

      -buildfile
            Used as a single parameter.
            Constructs a testFile from the given input in the
            commandFile.
            +Usage: -buildfile (testFile) (commandFile)*

      -check
            Used as a single parameter, followed by the names
            of the testFiles that are to be checked for
            consistency. Runs a check for consistency on all
            input in the testFiles. Should any file be found
            not to be consistent, the program prints by default
            out any differences on the screen.
            +Usage: -check (testFile)*

      -compare
            Used as a single parameter, followed by the name
            of a testFile and the name of a commandFile.
            Compares the execution of commands from one
            commandFile with results already stored in a
            testFile. Any differences will by default be
            printed out on the screen
           +Usage: -compare (testFile) (commandFile)

      -printinput
            Used as a single parameter, followed by the names
            of the testFiles that are to be traversed.
            Prints all input from testFiles to the screen.
            +Usage: -printinput (testFile)*

      -record
            Used as a single parameter, followed by the name
            of a testFile to which the results are to be
            written.
            This command opens a shell prompt where the user
            can enter commands one at the time. This command
            is the executed, and the result is stored in the
            testFile. Recording mode is exited by using the
            command 'exit' at the prompt.
            +Usage: -record (testFile)
```