Design and implementation of a new architecture for remote testing of mobile phones

Relatori:
ing. Giovanni Squillero
prof. Matteo Sonza Reorda

Company Sponsor
Motorola Electronics S.p.a.
ing. Walter Ruzzarin

Anno accademico 2007-2008
To my mother Mariangela
and my father Ezio
Summary

The goal of this work is allowing the execution of tests in Italy, and in general everywhere, in an automatic way. Italy is an attractive place particularly for testing of multimedia services available with mobile phones of third generation; in fact Italy is one of the countries with more networks supporting this technology worldwide.

In the hosting company this type of tests are often required from foreign countries, such India or USA, and since these countries belong to different time zones respect Italy it’s often necessary to wait the next day to obtain the results.

The goal of this thesis is therefore the creation of a system allowing automatic remote phone flashing and testing. Testing operator will choose through a web interface software and configuration to be loaded on each phone such as test cases to be executed; the system will perform automatically all necessary operations and notifies user of the test cases results by email.

With the developed system it will be possible to obtain results very quickly and to use differences in time zones to launch tests round the clock.
I’m grateful to Ernesto Sanchez for the support and the suggestions provided.

I thank the Turin Automatic Tool team at Motorola that helpful and friendly allows me to develop the project, particularly Stefano Gandini, Walter Ruzzarin and Marco Fiorentu’.

I’m grateful to all my teachers at Politecnico for the strong preparation provided especially regarding software development.
# Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Summary</td>
<td>II</td>
</tr>
<tr>
<td></td>
<td>Acknowledgements</td>
<td>III</td>
</tr>
<tr>
<td>1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1</td>
<td>State of the Art</td>
<td>2</td>
</tr>
<tr>
<td>1.1.1</td>
<td>3G networks and multimedia features testing</td>
<td>2</td>
</tr>
<tr>
<td>1.1.2</td>
<td>Motorola Turin Development Center</td>
<td>2</td>
</tr>
<tr>
<td>1.1.3</td>
<td>Current test execution approach</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Thesis goal</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Proposed System Architecture</td>
<td>4</td>
</tr>
<tr>
<td>2.1</td>
<td>Initial system architecture</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Interaction between system components</td>
<td>5</td>
</tr>
<tr>
<td>2.3</td>
<td>Test Controller - Test Station interaction</td>
<td>6</td>
</tr>
<tr>
<td>2.4</td>
<td>Test Controller - Web Server interaction</td>
<td>8</td>
</tr>
<tr>
<td>2.5</td>
<td>System architecture after interactions analysis</td>
<td>9</td>
</tr>
<tr>
<td>2.6</td>
<td>Exceptions handling and logging</td>
<td>11</td>
</tr>
<tr>
<td>2.7</td>
<td>Caching feature</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>Components Architecture</td>
<td>12</td>
</tr>
<tr>
<td>3.1</td>
<td>Mobile phones architecture</td>
<td>13</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Mobile phones hardware</td>
<td>13</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Mobile phones operations terminology</td>
<td>13</td>
</tr>
<tr>
<td>3.1.3</td>
<td>Seem elements</td>
<td>14</td>
</tr>
<tr>
<td>3.1.4</td>
<td>Phone preparation sequence</td>
<td>15</td>
</tr>
<tr>
<td>3.1.5</td>
<td>Phones connection</td>
<td>15</td>
</tr>
<tr>
<td>3.2</td>
<td>Test Stations architecture</td>
<td>15</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Current PCL libraries approach</td>
<td>15</td>
</tr>
<tr>
<td>3.2.2</td>
<td>The new approach used</td>
<td>17</td>
</tr>
<tr>
<td>3.3</td>
<td>Test controller architecture</td>
<td>19</td>
</tr>
</tbody>
</table>
3.3.1 Test stations and phones connection .................................. 19
3.3.2 Test lists assignment ....................................................... 20
3.3.3 Test lists loading ............................................................ 25
3.3.4 Test stations, test lists and test cases status tracking .................. 27
3.4 Database architecture ......................................................... 28
   3.4.1 Files stored into database ............................................. 29
   3.4.2 Phones and test stations management .................................. 29
   3.4.3 Test cases management .................................................. 32
   3.4.4 Test lists and users management ...................................... 32
   3.4.5 Files management .......................................................... 34
3.5 Web interface architecture ................................................... 35
   3.5.1 Folder structure on web server ....................................... 35
   3.5.2 User authentication ....................................................... 36
3.6 Exceptions management ....................................................... 36
   3.6.1 Exceptions types ........................................................... 36
   3.6.2 Exceptions hierarchy ..................................................... 38
4 Experimental evaluation ......................................................... 40
   4.1 Web interface implementation ............................................ 41
      4.1.1 Login page .............................................................. 41
      4.1.2 Main menu ............................................................... 41
      4.1.3 Test lists menu ........................................................ 41
      4.1.4 Test list editing ........................................................ 42
      4.1.5 Custom files management .......................................... 44
      4.1.6 Test list execution interface ....................................... 45
      4.1.7 Logs download interface ............................................ 46
   4.2 Description of the exploited technologies .................................. 47
   4.3 Java RMI technology ........................................................ 47
      4.3.1 Introduction and goals ................................................ 47
      4.3.2 Architecture ............................................................. 48
      4.3.3 Skeletons and stubs ..................................................... 48
      4.3.4 Serialization ............................................................. 49
      4.3.5 Transport Layer .......................................................... 49
      4.3.6 Remote objects binding and naming ................................ 49
      4.3.7 Distributed garbage collector ...................................... 49
      4.3.8 Java RMI use in the developed system ............................... 50
   4.4 AJAX technology ............................................................. 51
      4.4.1 AJAX technology usage in the developed project ................. 51
   4.5 Session variables ........................................................... 54
      4.5.1 PHP session variables implementation ................................ 54
      4.5.2 Security related aspects ............................................. 54
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>User requirements</td>
<td>22</td>
</tr>
<tr>
<td>3.2</td>
<td>Available phones on test station number 2 of the example</td>
<td>22</td>
</tr>
<tr>
<td>3.3</td>
<td>Cost of operations on a single phone</td>
<td>23</td>
</tr>
<tr>
<td>3.4</td>
<td>Evaluation of possible associations between available phones and user</td>
<td>24</td>
</tr>
<tr>
<td>3.5</td>
<td>Coefficients used in the example</td>
<td>25</td>
</tr>
<tr>
<td>3.6</td>
<td>Evaluation of the better test station in the provided example</td>
<td>25</td>
</tr>
<tr>
<td>3.7</td>
<td>Logic choosing which operations are needed</td>
<td>27</td>
</tr>
<tr>
<td>3.8</td>
<td>Exceptions table</td>
<td>37</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Initial system outline</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>System outline with test controller</td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>Final system outline</td>
<td>9</td>
</tr>
<tr>
<td>3.1</td>
<td>Seems elements structure</td>
<td>14</td>
</tr>
<tr>
<td>3.2</td>
<td>Phone connection outline</td>
<td>16</td>
</tr>
<tr>
<td>3.3</td>
<td>Test list assignment example</td>
<td>21</td>
</tr>
<tr>
<td>3.4</td>
<td>Test list loading example</td>
<td>26</td>
</tr>
<tr>
<td>3.5</td>
<td>Database outline</td>
<td>30</td>
</tr>
<tr>
<td>3.6</td>
<td>Exceptions hierarchy</td>
<td>39</td>
</tr>
<tr>
<td>4.1</td>
<td>Test list editing interface: test list details and phones configuration</td>
<td>42</td>
</tr>
<tr>
<td>4.2</td>
<td>Test list editing interface: test cases in current test list</td>
<td>44</td>
</tr>
<tr>
<td>4.3</td>
<td>Custom files management interface</td>
<td>45</td>
</tr>
<tr>
<td>4.4</td>
<td>Test list execution interface</td>
<td>46</td>
</tr>
<tr>
<td>4.5</td>
<td>Logs download interface</td>
<td>47</td>
</tr>
<tr>
<td>4.6</td>
<td>Java RMI Architecture Outline</td>
<td>48</td>
</tr>
<tr>
<td>4.7</td>
<td>Java RMI interactions in the developed system</td>
<td>50</td>
</tr>
<tr>
<td>4.8</td>
<td>Ajax technology timeline example</td>
<td>51</td>
</tr>
<tr>
<td>4.9</td>
<td>Runtime generated XML example generated during test list execution interface loading (see section 4.1.6)</td>
<td>52</td>
</tr>
<tr>
<td>4.10</td>
<td>Java RMI interactions in the developed project</td>
<td>53</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction
1.1 State of the Art

Following paragraphs describe the motivations advising the project and the approach used by company where this thesis has been developed.

1.1.1 3G networks and multimedia features testing

“3G networks are wide area cellular telephone networks which evolved to incorporate high-speed internet access and video telephony.” [5]

This type of network implements the third generation of mobile phone standard and adds to phone new features, such as doing video call, watching video streams and high speed connection to Internet. All these features must be developed on 3G phones and tested on a large number of networks.

Italy is a very attractive place for testing all these features since it’s one of the few countries providing four 3G networks. [6]

In the hosting company the testing of new phone software releases on Italian networks takes a lot of time due to the different time zone respect areas where the tests are developed. Usually a phone test request issued by a developer in a foreign country, for instance India or United States of America, takes a full day from the submission to the results; this is due principally to the difference of time zone and to the absence of an automatic procedure allowing remote tests execution.

1.1.2 Motorola Turin Development Center

This thesis has been developed within the Automation Team in Mobile Device Business department of the Motorola Development Center of Turin.

This center started his activity in 1999 and it has become one of the most important Motorola Excellent Center in the EMEA (Europe, Middle East, Africa) area. [1]

1.1.3 Current test execution approach

Mobile phones and the softwares managing them have to be tested before they are delivered to the customers. In this way it’s possible to avoid bugged products release requiring high substitution costs. These tests are usually made for the different services provided by simulating keys pression via software and checking reaction on display.

Hosting company has developed libraries, written mainly in Java language, allowing two levels test execution: the first one allows direct phone key press (e.g. press OK key, then LEFT key, etc.) whereas the second one gives an abstraction for a large number of phone models (e.g. take a photo, etc.).
Each phone feature is checked through tests, called test cases, already developed. Before test list execution phones software and configuration must be prepared; moreover a file containing the names of test cases to be run must be created in order to launch test list.

These operations are done by the user through different applications requiring a lot of manual work; particularly the preparation of devices is quite complicated due to a lot of parameters to be configured and dangerous operations to be done that could prevent devices from working.

1.2 Thesis goal

The goal of this thesis is the creation of a system allowing automatic remote phone flashing and testing. Testing operator will choose software and configuration to be loaded on each phone such as test cases to be executed; the system will perform automatically all necessary operations and notify user of the test cases results by email.

Testing operator will access to the system through a friendly web interface and test preparation will be followed as much as possible in order to avoid human errors. Moreover the service must be usable by a lot of users at the same time: for this purpose requests needing the same resources will be queued.

In addition system should be designed to be used in a large environment where not only users require tests from all over the world, but also the testing places are located far each other.

This thesis covers the whole of the project: from the design of its architecture to the coding of it, from the interfacing with the phones control libraries to the web site implementation; from the database design and implementation to the logic behind the algorithm controlling all the system.

Moreover some preexisting libraries have been extended in order to improve their functionalities and make better the integration of the project with the preexisting code.

In the next chapters developed system will be described in details.
Chapter 2

Proposed System Architecture
2.1 Initial system architecture

Initially the testing system was based on two components: user interface and test station.

The former one is a web server allowing the user to dialog with the system through a web interface without the need of additional software. Through this component user will be able to:

- choose phone models to be used;
- select or upload softwares and configurations to be set up on phones;
- choose test cases to be run;
- launch test lists and download the results.

The latter component manages connection with phones and implements the high level web interface features through low level methods, such as open connection with phone, start test list, etc.

2.2 Interaction between system components

As depicted in figure 2.1 the interaction between users and web server is done through the HTTP protocol.

The data exchange between the test station and the phones is represented on the right side of the outline; this interaction is possible using libraries developed by hosting company in order to control phones, called Phone Control Libraries (PCL).

About the remaining interaction between the web server and test station it isn’t possible to use a direct connection as depicted in the figure due to the following reasons:

- the web server is controlled by PHP scripts executed during the loading of the page; due to the limited execution time of these scripts, less than a second, it isn’t possible to manage the test station directly;
- it’s hard to allow more users to use at the same time the system and queue tests;
- it’s not easy to manage more test stations in the system.

Therefore it’s necessary to introduce another component that is responsible for making a connection between user interface and test station and allows to join to the system other test stations. This new component will be always on and will have the following tasks:
• connection management between web server and test stations;
• test lists assignment to test stations with load balancing and queues management;
• notification of results to the user at the end of each test list execution;

The new component has been called *test controller* and is depicted in figure 2.2.

### 2.3 Test Controller - Test Station interaction

A series of possible solutions for this interaction has been evaluated; the main ones are reported in the following paragraphs.

**ASCII interface**  All methods to be called by remote are codified as an ASCII command sent on a TCP/IP communication channel.
2.3 – Test Controller - Test Station interaction

Figure 2.2. System outline with test controller

- **PRO:** it allows using of different programming languages on the two sides of TCP/IP channel;
- **PRO:** it partially simplifies debug operations because almost all communications are in ASCII format;
- **CON:** for each method remotely invokable a corresponding command with parameters must be defined;
- **CON:** it’s necessary to implement a protocol to allow binary data transfer;
- **CON:** it introduces an overhead due to ASCII encoding of commands and values.

**Low level writing of data structure over communication channel** A communication class containing all information about command to be executed is defined; for each invocation of a remote command an object of this class is created and written on the communication channel.

- **PRO:** minimum overhead;
• CON: debug not easy;
• CON: use of different programming languages on the two sides of the TCP/IP channel is quite hard;
• CON: necessary to define a mapping between received objects and methods to be called.

Java Remote Method Invocation (RMI)  This technology allows the definition of objects with methods remotely invokable like Remote Procedure Calls (RPC) in C language.

• PRO: low level network communication methods are already implemented;
• PRO: easy to add new methods remotely invokable;
• PRO: each object implementing Serializable interface can be transferred;
• CON: overhead due to the high flexibility of this technology;
• CON: need to use Java language on both sides of TCP/IP channel.

After an evaluation of positive and negative aspects of each solution the last one has been chosen; therefore Java language must be adopted on both test station and test controller.

2.4 Test Controller - Web Server interaction

In the interaction between test controller and web server there are the following issues:

• different programming languages: it’s necessary to create a communication channel between the test controller written in Java language and the web server controlled by PHP scripts;
• data persistence: in case that it’s necessary to reboot the test controller no critical data should be lost.

In order to satisfy these requirements a relational database has been inserted between test controller and web server; in this way it’s possible to store on it all critical data of the system, as well as easily connect web server and test controller.

About the type of relational database to be used there was no particular requirement; for convenience MySql database server has been chosen.
2.5 System architecture after interactions analysis

After all these considerations the outline in figure 2.3 has been defined.

In the following paragraphs system functions are summarized by role.

Users

- access from the entire hosting company network to the website;
- create customized test lists choosing between available test cases;
- upload on the website customized software versions or choose between ones already available;
- choose phone models to use during test list execution as well as software versions and configurations;
- retrieve results once tests have been executed.
**Web server**

- provides a web interface allowing definition of test lists as well as of phones configuration;
- stores on database test lists to be executed and related data;
- provides results obtained during test lists execution to the user.

**Database**

- contains all critical information used by system;
- provides storage for all shared files.

**Test Controller**

- retrieves from database software versions and configurations to be set up on phones;
- retrieves from database test lists to be executed;
- assigns test lists to test stations and performs load balancing if possible;
- notifies user of results by email once test list execution ends.

**Test Stations**

- manage connection with devices connected by USB ports;
- set up phones and launch test lists driven by test controller;
- notify results to test controller.

**Phones**

- devices where the tests are executed through libraries provided by hosting company;
- their software versions and configurations are changed before tests execution;
- during tests execution, their keyboards are virtually pressed and their displays are captured.
2.6 Exceptions handling and logging

The designed system has to be run continuously for a long time: for this reason exceptions must be managed and the system should be able as much as possible to continue its execution normally. Moreover to simplify bug fixing all troubles are stored into the database together with a detailed log of the involved components.

2.7 Caching feature

In order to avoid unnecessary network traffic a caching system has been developed between each test station and test controller. In this way it’s possible to avoid network transfer if the requested file has been already downloaded and the original one has not been changed.

This feature works both with the files stored into the database and with the ones saved into disk storage.
Chapter 3

Components Architecture
3.1 Mobile phones architecture

Next paragraphs will discuss mobile phones architecture, especially related to Motorola ones.

3.1.1 Mobile phones hardware

Phone memory is a flash memory and it can be subdivided into five parts by functionality.

**Boot loader**  This part of memory is read at phone boot and it’s rarely updated; it’s responsible for phone software setting up and loading.

**Software**  It allows use of phone functionalities such as doing calls, reading messages, etc. Since it’s often updated, a series of tests has been developed in order to ensure that phone functionalities continue working properly after software changes.

**Runtime configuration**  It defines software behavior only partially changeable through phone graphical interface. It allows definition of a lot of configurations, such as user interface language, if battery should be recharged, etc.

**Filesystem**  This area of memory contains resources used by phone software such as localized texts, icons, skins, ringtones, etc. It can be customized by customers that want to resell phones with a particular user interface.

**Phasing data**  This data is set on each phone during production and depends on each single device, such as the antenna calibration. It’s important to store this information before software updates because they could be deleted during this process.

3.1.2 Mobile phones operations terminology

This section describes terminology used in the next chapters.

**Flex file flexing**  This operation involves a file, called flex file, describing changes to phone configuration as well as resources to be stored into filesystem. Memory containing configuration data is entirely rewritten during this process while filesystem is only partially changed.
**Mini-flex flexing**  *Mini-flex* file content is smaller if compared with a flex file: it changes only a few configurations and resources.

**Flashing**  This operation is usually done using a particular file called *1FF file*. This file contains:

- a *software* to be stored into flash memory;
- a *flex file* to be flexed as in previous paragraph.

### 3.1.3 Seem elements

Runtime configuration data are organized into several seem elements; each one is arranged as matrix. This memory can be imaged as a three dimensional array as shown in figure 3.1.

Each seem element is addressed through four numbers:

a) element id: it chooses the matrix in the first set;

b) record number: inside the matrix it selects the row number;

c) offset: inside the row it chooses the element start position;

d) length: it defines the number of bytes constituting the element.

Therefore seem element highlighted in the figure is addressed through these four hexadecimal numbers: 00F5, FFA8, 42 and 2.
3.1.4 Phone preparation sequence

Usually this sequence is followed during phone preparation:

- each phone is flashed with a 1FF file containing the software baseline that is a stable build version;
- each phone can be flexed with a flex file or mini flex file to adapt it to a particular customer;
- each phone can be flashed again with a reflash software version with a few changes compared with the baseline.

3.1.5 Phones connection

Power supply is provided to phones through a battery eliminator, that is a particular battery having the external contacts not connected with the internal elements as usual but with an external power supply. Power supply and battery eliminator are connected each other through a relay board, a device controlled by personal computer capable of close or open each couple of contacts. In this way it’s possible to temporarily switch off power supply in order to force phones reboot.

Data connection between phones and test station is provided by USB cables. The Vcc pin is connected to another couple of contacts on relay board in order to disable USB connection with phones.

Figure 3.2 depicts the way phones are connected to test station.

3.2 Test Stations architecture

Each test station uses phone control libraries developed by hosting company that provide an interface for flashing and testing phones.

3.2.1 Current PCL libraries approach

Following paragraphs describe the way test lists are normally executed in order to highlight improvements given by the new approach.

Phones preparation Phones are connected to personal computer one by one in order to set up software and configurations. This operation is done through a graphical interface and takes about 20 minutes for each phone.
Configuration file  User must provide a configuration file having the following syntax:

OBJECT1 = pcl.io.connection
OBJECT1_CONNECTION = USB 4.1.0

OBJECT2 = pcl.commands.advanced
OBJECT2_CONNECTION = OBJECT1

OBJECT3 = pcl.commands.base
OBJECT3_CONNECTION = OBJECT1

OBJECT4 = pcl.logging
OBJECT4_CONNECTION = OBJECT1
OBJECT4_COMMANDS_OBJECT = OBJECT2

LOG_LEVEL = 1
This configuration file defines configuration parameters for one phone. Lines with syntax `OBJECTn = ...` define new objects of the type on the right of equal sign, while the ones with syntax `OBJECTn_X = ...` define property X for object `OBJECTn`.

For each phone it’s required:

- an object of type `pcl.io.connection` for managing connection with phone, with `CONNECTION` property declaring hardware address where phone can be found, or `AUTO` to select the first phone available;

- an object of type `pcl.commands.advanced` that allows sending of commands to phone and defines, through the `CONNECTION` property, the name of `pcl.io.connection` object to be used for physical connection to the phone;

- an object of type `pcl.commands.base` allowing keys pression and display capture, requiring the same properties definition as the previous object;

- an optional object of type `pcl.logging` allowing to save log messages provided by phone, requiring besides the `CONNECTION` property like previous object also the `COMMANDS_OBJECT` one, defining the name of `pcl.commands.advanced` object to be used.

Lines not starting with keyword `OBJECT` are stored as global properties of the Java virtual machine and are used by the libraries for particular configurations, for instance defining log level.

For each object it’s possible to define some optional properties left out of the description.

**Test list preparation and execution**  
User must write into a file the class names of test cases he wants to execute. Afterwards the library is invoked through command line with the names of configuration file and test list as parameters.

Test list execution takes about five minutes each test case; at the end the libraries generate a spreadsheet file wherein user can found useful information to fix problems happened during test list execution.

### 3.2.2 The new approach used

Following paragraphs describe troubles emerged from the current approach and solutions adopted in the new one.

**Need to automatically set up phones software and configuration**  
PCL libraries provide some methods useful to set up phones software and configuration: in order to automatically execute test lists, these methods must be used instead of using external graphical tools.
Troubles in using available testing libraries Approach usually adopted has some troubles in integration with the system to be developed.

Actually the class Test List Runner (TLR), contained into PCL libraries, allows only one test list execution with a single configuration and implements a Singleton pattern\(^1\) therefore the execution of more test lists as well as the change of phones configuration requires the creation of different processes using different Java virtual machines with the following disadvantages:

- need for inter process communication;
- slowing down in performances.

In order to avoid the creation of a process for each test list execution another solution has been adopted; an extension of the TLR class has been created with the following features:

- implements Singleton pattern as the extended class;
- the first time it’s accessed, some low level protected methods of extended class are called in order to start the test list runner with a given properties configuration file and an empty test list;
- afterwards the phones can be added each one with its own configuration;
- a new method called launchTestList allows the execution of a test lists through low level protected methods invocation.

Although it implements a singleton pattern, thanks to the method launchTestList it’s possible to execute more consequential test lists.

Low level phone loading and related objects creation As in the previous paragraph at TLR start no phone configuration is given. To give more flexibility to phone creation a parser has been created with some additional features than the TLR one.

First of all it allows macros definition using special characters \($\) and \(\)\). The content between the two parenthesis is searched into a runtime defined hash map containing a list of substitutions. So a standard configuration file is defined and when a phone is added its content is customized. For instance a substitution hash map contains a key \(\text{CONN}\) with associated value \(\text{USB 1.1.0}\) is provided; when parser

\(^1\)Singleton pattern is used to avoid the class to be instantiated over and over: to obtain this feature the constructor of the class is protected and the only way to get the unique object instance is through a provided static method. \(^7\)
reads \texttt{OBJECT1\_CONNECTION}=$\{\text{CONN}\}$, it substitutes \$\{\text{CONN}\}$ with \texttt{USB 1.1.0} that is the current phone hardware address.

Moreover if substitution is related to a file its content is downloaded from test controller. In this way it’s possible to define phone related files at runtime.

After the required substitutions are done, a customized parser creates needed objects using PCL libraries and keeps trace of them. In this way it’s possible to disable and enable again phones allowing phone disconnection and reconnection.

\textbf{Phones Reordering} Usually test cases access to phone related objects through an array sorted by phone loading order.

To increase the flexibility of the system it’s required that phones order could be changed before each test list execution; to allow this it isn’t enough to enable phones in a different order or keep disabled phones not involved. Rather it’s necessary to enable phones all together and reorder the array to move involved phones at the beginning.

\section{3.3 Test controller architecture}

Test controller is composed by a main loop scheduling the following tasks one by one:

- test stations and phones connection;
- test lists assignment;
- test lists loading.

For each task a refresh time defining the period after that the task will be scheduled again is associated: when the scheduled time came on, task is executed or postponed if another one is still running.

The following paragraphs will describe all these tasks in details.

\subsection*{3.3.1 Test stations and phones connection}

An object called \textit{Remote Station} has been created with the following characteristics:

- it manages connection with each test station;
- it’s executed in a separated thread avoiding to slow down the test controller during slow operations execution;
• it stores locally information about associated test station in order to avoid useless network traffic.

In order to collect remote stations objects an hash map has been created: this hash map is called remote station pool and it’s addressed by the test station id.

This task, executed one time per minute, retrieves from database enabled test stations and searches them into remote stations pool: for each one, if remote station already exists, no action is performed. Otherwise a new remote station object is created with the configuration data retrieved from database. Moreover the configuration of phones associated to this test station is retrieved and stored into the remote station.

Once remote station configuration has been completed a new thread is created: this new thread is constituted by an infinite loop performing continuously the following operations:

• try\(^2\) to connect to the assigned test station if not yet connected;
• go in a synchronized area controlled by a semaphore;
• if a test list has been loaded:
  – prepare phones on the test station;
  – execute assigned test list on the test station;
  – send results to user by email.
• invoke the wait method on semaphore awaiting for a new test lists assignment.

### 3.3.2 Test lists assignment

This task is executed twice in a minute in order to assign waiting test lists to test stations with minor cost. This cost is the sum of the following components:

• relative preparation costs: proportional to the required time to adapt phones to user configuration requirements;
• usage of test station cost: in order to balance load, more exploited test stations cost more;
• link cost: if a test station is geographically far it costs more because data transfer takes more time.

In figure 3.3 and in the following paragraphs detailed assignment algorithm such as an example will be proposed in order to clear up concepts.

\(^2\)if connection fails remote station tries to reconnect again after 30 seconds.
3.3 – Test controller architecture

**Pick an unassigned test list from database**

Algorithm builds a list of connected test stations and picks from database a not assigned test list with a scheduled execution date elapsed or not specified, beginning from the one with greatest priority. Also user phones configuration requirements, phone model and customer are retrieved from database beginning from the ones requiring a particular customer.

**List available test stations and count phones matching with requirements**

A list of phones connected to the system is build and user phone configuration requirements are analyzed; for each requirement algorithm selects test stations having

---

**Evaluate the better association between available phones and user requirements**

Test station #1
- no. of matching phones: 1
- test station cost: 175
- best solution: 0
- best solution cost: 175

Test station #2
- no. of matching phones: 2
- test station cost: 150
- best solution: 0
- best solution cost: 175

Test station #3
- no. of matching phones: 0
- test station cost: 250
- best solution: 1
- best solution cost: 250

Test station #4
- no. of matching phones: 2
- test station cost: 150
- best solution: 0
- best solution cost: 175

---

**Assign test list to test station with minimum cost**

Test station #2
- no. of phones: 4
- test station cost: 150
- best solution: 0
- best solution cost: 150

Test station #4
- no. of phones: 3
- test station cost: 250
- best solution: 1
- best solution cost: 250
at least one matching phone, increments counter and removes the first matching phone from the list of the available ones.

**Exclude test stations that don’t satisfy requirements** Leave out the test stations having the counter less than the number of phones required by test list: so test stations not satisfying user requirements are ignored. If all the test stations in the system are left out, the test list is marked as *unavailable* and algorithm execution is interrupted.

**Evaluate the better association between available phones and user requirements** In this paragraph the attention is focused on test station number 2 of the provided example: user requirements are reported in table 3.1 whereas the phones connected to current test station are displayed in table 3.2.

<table>
<thead>
<tr>
<th>phone config. #</th>
<th>model #</th>
<th>baseline #</th>
<th>flex #</th>
<th>reflash #</th>
<th>customer #</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 3.1. User requirements

<table>
<thead>
<tr>
<th>phone #</th>
<th>model #</th>
<th>baseline #</th>
<th>flex #</th>
<th>reflash #</th>
<th>customer #</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 3.2. Available phones on test station number 2 of the example

For each possible association between user requirements and available phones the following algorithm is executed:

- if available phone model is not the same as the required one reject this association;
- if user required a particular customer but the one available on phone is not the same reject this association;

\[^3\] A phone matches requirements if both models and customers are the same; customer correspondence is ignored if no particular one is required by user.
• if available phone baseline, flex and reflash are the same as the required ones accept this association without any cost;

• if available phone has not been flexed and both baseline and reflash are the same as the required ones, accept this association with cost equal to flex one;

• if available phone has not been reflashed and both baseline and flex are the same as the required ones, accept this association with cost equal to reflash one;

• if available phone has not been neither flexed nor reflashed and baseline is the same as the required one accept this association with cost equal to the sum of reflash and flex ones;

• otherwise since neither software nor flex of available phone can be reused, this association is accepted with cost equal to the sum of baseline flash, flex and reflash ones.

This algorithm has been applied to given example with the costs in table 3.3; results are reported in table 3.4.

<table>
<thead>
<tr>
<th>operation</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>baseline flashing (1FF file)</td>
<td>100</td>
</tr>
<tr>
<td>flexing (flex file)</td>
<td>50</td>
</tr>
<tr>
<td>reflashing</td>
<td>25</td>
</tr>
</tbody>
</table>

Table 3.3. Cost of operations on a single phone

For each configuration into table 3.4 unused phones are marked with a · sign whereas the not applicable label is put on rows rejected because phone model or customer differs from the required one.

The unit of measurement of costs is not specified because it isn’t a real monetary cost. In fact it’s only a value proportional to the time needed to perform operations useful to find the best solution through minimization of costs.

In the example the best configuration for the current test station is the number 3, using the phones #1 and #3 respectively as the first and the second one in the user test list, with a cost of 175 units.

**Assign test list to test station with minimum cost** Once the best solution has been found on every suitable test station, each test station cost, $TSC$, is evaluated using the following equation:
Table 3.4. Evaluation of possible associations between available phones and user requirements

<table>
<thead>
<tr>
<th>Attempt number</th>
<th>Phone - User req.</th>
<th>Operations required</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#0    #1    #2    #3</td>
<td>Baseline Flex Reflash</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>·  ·  0  1</td>
<td>2  2  2</td>
<td>350</td>
</tr>
<tr>
<td>2</td>
<td>·  ·  1  0</td>
<td>not applicable</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>·  0  ·  1</td>
<td>1  1  1</td>
<td>175</td>
</tr>
<tr>
<td>4</td>
<td>·  0  1  ·</td>
<td>not applicable</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>·  1  ·  0</td>
<td>not applicable</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>·  1  0  ·</td>
<td>1  1  2</td>
<td>200</td>
</tr>
<tr>
<td>7</td>
<td>0  ·  ·  1</td>
<td>2  2  1</td>
<td>325</td>
</tr>
<tr>
<td>8</td>
<td>0  ·  1  ·</td>
<td>not applicable</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>0  1  ·  ·</td>
<td>1  1  2</td>
<td>200</td>
</tr>
<tr>
<td>10</td>
<td>1  ·  ·  0</td>
<td>not applicable</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1  ·  0  ·</td>
<td>not applicable</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>1  0  ·  ·</td>
<td>not applicable</td>
<td></td>
</tr>
</tbody>
</table>

where

\[ TS_c = \frac{PC_c}{PC_n} \cdot PC_{coeff} + \frac{U_t}{U_a} \cdot U_{coeff} + \frac{L_c}{L_a} \cdot L_{coeff} \]

- \(PC_c\) is the lowest phones preparation cost, equals to 175 in the example;
- \(PC_n\) is the number of phones involved into test list;
- \(U_t\) is the number of test list assigned to this test station since system reset;
- \(U_a\) is the average of \(U_t\) calculated on active test stations;
- \(L_c\) is the link cost with the test station retrieved from database;
- \(L_a\) is the average link cost calculated on active test stations;
- \(PC_{coeff}, U_{coeff} \) and \(L_{coeff}\) are respectively the phone configuration, test station usage and link cost coefficients.

Changing coefficients in this equation it’s possible to give more relevance to the costs to adapt phones, to level off the work distribution on test stations or to reduce the link costs. In the example coefficients in table 3.5 have been used.

Applying the formula to the example the results in table 3.6 are obtained and the test station number 2 is chosen because it’s the one with the lowest cost.
### 3.3 – Test controller architecture

<table>
<thead>
<tr>
<th>symbol</th>
<th>name</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>$PC_{coeff}$</td>
<td>phone configuration</td>
<td>1</td>
</tr>
<tr>
<td>$U_{coeff}$</td>
<td>test station usage</td>
<td>20</td>
</tr>
<tr>
<td>$L_{coeff}$</td>
<td>link cost</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 3.5. Coefficients used in the example

<table>
<thead>
<tr>
<th>Test Station #</th>
<th>$PC_c$</th>
<th>$PC_n$</th>
<th>$U_t$</th>
<th>$U_a$</th>
<th>$L_c$</th>
<th>$L_a$</th>
<th>$TS_c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>175</td>
<td>2</td>
<td>10</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>143</td>
</tr>
<tr>
<td>4</td>
<td>325</td>
<td>2</td>
<td>8</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>205</td>
</tr>
</tbody>
</table>

Table 3.6. Evaluation of the better test station in the provided example

Once test station with the lower cost is defined its id is written in the database row corresponding to the current test list and its status is changed to assigned. Moreover user requirements related to this test list are updated with the phone ids corresponding to the associated phones on the assigned test station.

#### 3.3.3 Test lists loading

Once every five seconds assigned test lists are retrieved from database and loaded as depicted in figure [3.4](#).

Pick from database an assigned test list waiting to be loaded Assigned test lists not yet loaded are extracted by priority; test list is skipped if assigned test station is not connected or busy.

Select from database related users requirements and test cases The name and parameters of user chosen test cases are extracted from the database together with user requirements.

Through a synchronized area test cases and user requirements are communicated to thread managing the connection with the assigned test station.

Prepare phone involved in the assigned test list The thread managing the connection with the assigned test station checks involved phones configuration in order to determine which operations should be performed, as reported in table [3.7](#) with the following legend:

- = available phone has the same configuration as required by user;
3 – Components Architecture

Figure 3.4. Test list loading example

- ≠ available phone configuration differs from user requirements;
- − don’t care;
- △ flex (or reflash) on phone has not been flexed (or flashed) yet;
- ⋆ operation must be skipped if no particular flex (or reflash) is required by user.

Load test list together with required files on test station and launch test list

Generated test list file is transferred to test station and its execution starts. For each executed test case, status is reported to test controller that notifies user of it through the web interface.
At the end of test list execution retrieve results from test station Once test list has been completed test controller retrieves from the test station a spreadsheet file called \textit{MMLogFile} and containing detailed information about each test case.

Notify test results to user through the email service Test controller sends a detailed email to user containing test cases names and status. Test cases results are published on the web interface together with the generated \textit{MMLogFile} file downloadable by user at any time.

3.3.4 Test stations, test lists and test cases status tracking

In order to keep user informed about his test lists, status is stored into database for all these elements.

Test stations status tracking This status is stored for each test station in the system; acceptable values are:

- Offline: test station is not connected to the system;
- Loading: test station is connected to the system but it isn’t completely loaded yet;
- Idle: test station is loaded but has no work assigned;
- Flashing: test station is preparing devices for test list;
- Running: test station is running a test list.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
baseline & flex & reflash & action \\
\hline
= & = & = & do nothing \\
\hline
= & = & $\Delta$ & reflash \\
\hline
$\Delta$ & = & = & flex \\
\hline
$\Delta$ & $\Delta$ & = & both flex and reflash \\
\hline
$\neq$ & - & - & baseline flash + flex $\star$ + reflash $\star$ \\
\hline
\end{tabular}
\caption{Logic choosing which operations are needed}
\end{table}
Test lists status tracking  This status is stored for each test list inserted into database; acceptable values are:

- Editing: user is still editing the test list;
- Waiting: test list is ready to be assigned to a test station;
- Assigned: test list has been assigned to a test station but it’s waiting to be loaded;
- Flashing: assigned test station is preparing phones in order to adapt them to the required configuration;
- Running: assigned test station is running the test list;
- Done: test list execution has been completed;
- Unavailable: the user requirements are not satisfiable by test stations available in the system.

Test cases status tracking  This status is stored for each test case inserted in the current test list; acceptable values are:

- Waiting: test case is waiting to be executed;
- Running: test case is running;
- Passed: test case result is passed;
- Failed: test case result is failed.

3.4 Database architecture

The Mysql database has been chosen in order to store data and virtually connect the web server with the test controller; this database server is used together with the InnoDB engine that provides the following features:

- locking statements: it’s possible to obtain writing or reading locks on one or more tables to keep database consistent; these statements are used to prevent other threads from reading tables that current thread is updating and vice versa;
• transactional statements: these statements are needed by algorithms requiring more update queries to be executed, such as the test list assignment one; since these algorithms might fail wherever, it should be possible to revoke changes and roll back the database at the initial status.

Locking statements are often used in the developed code while transactional ones only a few.

The following sections will describe the approach used for file storage and will examine database structure depicted in figure 3.5. Almost all the tables contain an enabled field, that defines if the row should be considered during the queries: usually rows with enabled field set to false are ignored. Moreover some tables have the created field storing the row insertion date.

As usual primary keys are defined in all the tables with the name id and are written in the outline in boldface.

### 3.4.1 Files stored into database

All files needed by system to work properly are stored into database or on a storage drive in order to concentrate files as much as possible.

Small files, with size less than 20 megabytes, are conventionally called included files since they are directly stored into database as a blob field; in this case an other field with the same name but followed by _changed suffix is created if it’s necessary to save the file last change date; in this way caching feature described into paragraph 2.7 can work properly.

Vice versa larger files can’t be stored directly into database, because the driver doesn’t provide methods allowing to split binary data inserts in parts; due to this limitation it isn’t possible to perform inserts or updates of data having large size without exceeding system memory available to the Java virtual machine. For this reason, when large files have to be stored, a folder corresponding to the table field is created on database server hard disk; in these directories each file name is the same as the corresponding row id. This type of file will be conventionally called linked file.

### 3.4.2 Phones and test stations management

These tables contain information about the existing test stations and the connected phones. The following paragraphs will describe tables in details.

**Phones**  Each phone in the system must contain a row in this table. The description of table fields follows:
Figure 3.5. Database outline

- model id: it’s the id corresponding to the model of this phone;
- device address: it stores USB port address where phone is connected;
- baseline, flex and reflash ids: they’re the ids corresponding to software and
flex versions installed on the device;

- language file: it’s the file name containing the translations for the installed software;

- imei: it stores the *International Mobile Equipment Identity* associated to this phone, that is a unique number assigned at every GSM or UMTS mobile phone [8];

- imsi: it stores the *International Mobile Subscriber Identity* associated to the sim in the phone, that is a unique number assigned at every GSM or UMTS network user [9];

- mobile phone number: it’s the number to dial to call the phone;

- customer id: the id corresponding to the customer selling the sim;

- power and connection relays: the number of relay to close in order to supply power to phone or enable connection with PC respectively.

**Test stations** Each test station connected to the system must be listed in this table. The contained fields are listed below:

- IP address: the IP address where test station is bound;

- status: the status of test station, for more details see paragraph 3.3.4;

- link cost: it’s used during test list assignment algorithm.

**Customers** This table lists available customers in the system. The description of table fields follows:

- name: it’s the name of the customer, for instance *Vodafone*;

- img: it’s the customer logo, stored in a blob field and used in the web interface.

**Phone models** It contains the models of available phones; the contained fields are described below:

- platform id: it’s the id of the platform corresponding to the current model;

- name: it’s the name retrieved from the device, for instance *RAZRV3xx*;

- img: it’s the picture representing the phone, stored in a blob field and used in the web interface.
3.4.3 Test cases management

These tables contain data about predefined and customized test cases such as phone platforms.

Predefined test cases  This table contains information about customized test cases available in the system; the fields contained in the table follow:

- test central id: it’s an unique textual identifier assigned at each test case;
- filename: it’s the classname implementing the test case, comprehensive of the package name;
- description: it provides information about actions performed by test case;
- stress: if test tries to do a lot of times the same actions;
- phones number: the number of phones required;
- time: the average time its execution takes.

Custom test cases  This table stores data about user defined test cases:

- user id: the id of user owner of this test case;
- name: it’s the name chosen by the user;
- type: indicates if test case is binary or ASCII, that is respectively a Java class or a sequence of keys to be pressed on phone;
- data: test case provided by user.

Test case groups  It contains information about groups test cases are subdivided in.

Platforms  It contains available software platforms.

Platforms - test cases  It provides a many-to-many relationship between each test case and compatible platforms.

3.4.4 Test lists and users management

The tables of this section stores data about user and defined test lists. The section is composed by four tables.
3.4 – Database architecture

Users  It provides information about users of the system. The contained fields are:

- login: the user name used to access to the system;
- email address: the address retrieved through directory services;
- last login date: the date of last login of the user in the system.

Test lists  This test list contains both waiting and executed test list:

- user id: the id of user owner of this test list;
- name: the name assigned by user;
- priority: a number expressing the priority assigned;
- executed on: if equal to null test list will be executed as soon as possible; otherwise it will be launched on the specified date;
- status: the status of current test list, for more details see paragraph 3.3.4;
- test station id: it’s the id of the test station to whom the test list has been assigned, if available;
- MMLogFile: once test list execution ends, contains the generated spreadsheet with results;
- started: the date when the test list execution started;
- flashtime: the time taken by flashing and flexing operations.

Test cases - testLists  It creates a many-to-many relationship between test lists and test cases, both predefined and customized; moreover it stores the status of test case\(^4\) in test list and the ordering.

Phone configurations  It contains user requirements about phone features:

- test list id: it’s the test list to whom this phone config belongs;
- loading order: it defines the loading order of this phone used by test cases;
- baseline, flex and reflash ids: they’re the ids corresponding to software and flex versions required on the device;

\(^4\)for more details about test case status see paragraph 3.3.4
• customer id: the id corresponding to the required sim customer, null if no particular customer is required;

• test station id: it’s the id of the phone to whom the phone configuration has been assigned, if available.

3.4.5 Files management

The following paragraphs describe tables dedicated to file storage:

Baseline, flexes and reflashes tables These tables are very similar and contain flash and flex files available in the system; and they are linked with platforms and users tables. Contained fields are:

• platform id: it’s the platform for whom this file is suitable, null if it isn’t platform specific;

• compass file id: it’s the corresponding Compass id, if available;

• user id: the id of user owner of this file, null if this is a predefined file available for all the users;

• type: it can be local, uploaded or compass;

• name: the name associated to this file;

• status: current status of the file, it can be to be downloaded, downloading or available.

Compass files This table contains available files on Compass that is the system used by hosting company to share files all over the world. The available fields are listed below:

• platform id: it’s the platform for whom this file is suitable, null if all the platforms;

• name: the name of the folder containing the file;

• root folder id: it’s the id associated to the folder containing the file;

• release notes file id: it’s the id associated to the related release notes file;

• release notes directory id: it’s the id associated to the folder containing the release notes file.

5compass is the system used by hosting company to share files
Generic files  This table contains list of generic files that cannot be saved in other tables. In this way it’s possible to concentrate these files in the same place. Each file can be addressed by a numerical id.

3.5 Web interface architecture

A web interface has been designed in order to allow testing operators to access to the testing system.

The interface uses profusely AJAX technology: in this way it’s possible to reduce response time and improve the interaction of user with the system.

A detailed description of the way AJAX technology works is available in the section 4.4.

The following sections will describe web interface architecture.

3.5.1 Folder structure on web server

Webserver folders are structured as shown in the following list:

- **conf/**: configuration data about web interface such as database access information;
- **files/**: files of the single pages, for instance login page, test list one, etc.;
- **imgs/**: graphics used into website;
- **jscript/**: javascript downloaded by browser;
- **libs/**: shared libraries common to all the developed pages;
- **stylesheets/**: style sheets used by browser to change the appearance of the pages.

Requests for both HTML and XML files are done through *index.php* file, that’s stored into server root. This file loads all the necessary libraries, opens a connection with the database and check user is correctly authenticated. On success the script tries to load the page specified through the *page* parameter from *files/* subfolder. If request doesn’t provide the name of the page to be loaded, script loads the main menu. Rather if a not valid page name has been provided an error message is displayed on user browser.

To increase security, script checks that provided page name only contains alphanumerical chars, limiting the user to access exclusively to files contained into directory *files/*.
3.5.2 User authentication

When an user accesses the website the HTTP protocol is switched for HTTPS one providing more security through authentication of server and encryption of the communication channel.

In order to allow user authentication web site has been interfaced with LDAP internal service, allowing to check username and password provided by user. Moreover in this way it’s possible to know user details, such as family name and email address.

The first time user logs into the web interface a new row in table users is created with user details. In this way it’s possible to reference the user by a numerical id instead of using the provided alphanumerical username.

On user login an array is stored into a session variable called login and containing all useful information about the user: in this way it’s possible to filter provided information and display only data related to the connected user.

Moreover Users table also defines which accounts are administrative: these accounts are able to see detailed information about system, such as logs and test stations status.

3.6 Exceptions management

Following paragraphs will describe developed exceptions types and hierarchy. In this way exceptions thrown both on test controller and on test stations are automatically logged into database and when possible the program execution continues normally.

3.6.1 Exceptions types

As shown in table different kind of exceptions have been defined: they differ on severity level and on the system component that throws it.

Recoverable Test Station Exception This type of exception could be due to:

- wrong configuration: a phone is not correctly configured;
- phone key not found: request phone key is not available on selected phone;
- storage error: an error happens during results storage;
- PCL libraries error: a not severe exception has been thrown by phone control libraries.

\footnote{For detailed description about session variables functioning, see 4.5}
3.6 – Exceptions management

<table>
<thead>
<tr>
<th>Type</th>
<th>Unrecoverable</th>
<th>Recoverable</th>
<th>Remote</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thrown by</td>
<td>Local</td>
<td>Local</td>
<td>Remote</td>
</tr>
<tr>
<td>Test Station</td>
<td>Unrecoverable</td>
<td>Recoverable</td>
<td>Remote</td>
</tr>
<tr>
<td>TestStation Exception</td>
<td>TestStation</td>
<td>TestStation</td>
<td>TestStation</td>
</tr>
<tr>
<td>Test Controller</td>
<td>Unrecoverable</td>
<td>Recoverable</td>
<td>Remote</td>
</tr>
<tr>
<td>TestController Exception</td>
<td>TestController</td>
<td>TestController</td>
<td>TestController</td>
</tr>
</tbody>
</table>

Table 3.8. Exceptions table

This exception is logged and doesn’t prevent system from continuing the execution normally.

**Unrecoverable Test Station Exception**  This type of exception is thrown when test station goes in an undefined state preventing from continuing.

It can be generated by the following errors:

- test station configuration or environment error: phone configuration is wrong or it’s not possible to allocate hard disk resources;
- phone connection lost: at least one phone is no more available;
- wrong methods call sequence: test controller doesn’t follow the correct order in methods call (e.g. enable request of a phone already enabled, etc.);
- PCL libraries error: a severe exception has been thrown by phone control libraries.

When test controller receives this type of exception it requires an immediate reboot to test station. If exception has been thrown during a test list execution the test list is suspended.

**Recoverable Test Controller Exception**  This exception is due to not severe errors during test controller execution. They can be thrown in the following circumstances:

- an Unrecoverable Test Station Exception has been thrown;
- wrong methods call sequence: if a thread in test controller doesn’t follow the correct order in methods call (e.g. launch of a test list on a disconnected test station, etc.);
• user notification error: an error happens during email report sending.

This exception is logged but allows test controller to continue normally.

**Unrecoverable Test Controller Exception**  This exception is the most critical for system and prevents test controller from continuing its execution. This could be due to:

• lost connection with database;

• severe error during access to database (e.g. query errors, etc.);

• wrong test controller configuration or inconsistent data.

Test controller tries to force a reboot on all test stations leaving them into the initial state. After that, it tries to store exception and stops its execution.

**Remote Test Controller and Test Station Exceptions**  As described communication between test controller and test stations is done through Java RMI. All methods invokable through this technology throw a particular exception called *RemoteException*. This type of exception is thrown automatically by Java RMI library when:

• there is a network error (e.g. connection lost, etc.);

• there is a marshalling error (e.g. called method not found, wrong parameters, etc.).

When this type of exception is caught a new exception of type *RemoteTestControllerException* or *RemoteTestStationException* is thrown in order to allow an higher level management of it. The type of thrown exception depends on where the *RemoteException* is caught.

### 3.6.2 Exceptions hierarchy

Described exception types implement common exception classes as depicted in figure 3.6.
Figure 3.6. Exceptions hierarchy
Chapter 4

Experimental evaluation
4.1 Web interface implementation

In the following paragraphs will be described some screenshots captured on web interface that is the most visible result of this work for the end user.

4.1.1 Login page

When a user connects to the website a form requiring username and password is displayed; if the user correctly logs in into the interface a welcome message is displayed above the main menu together with the last access date.

If the user pushes the button log-out or simply closes the browser window the authentication is lost and username and password will be required again.

4.1.2 Main menu

It provides a list of the pages accessible through the web interface:

- Create a new test list;
- Show test lists and retrieve test results;
- Browse available phone models;
- Browse available test stations configuration;
- Browse test stations status;
- Download available logs.

User can access to the requested page clicking on the corresponding name in the menu. The last two functionalities of the list are displayed and accessible only if administrative rights have been provided to the connected user.

4.1.3 Test lists menu

This menu displays the name and the status of the test lists created by the user.

Near each test list status a link allows to access to test list details: if test list status is editing the editing interface is displayed, otherwise the test list execution one is provided.
4.1.4 Test list editing

This page is fully implemented through the AJAX technology, that allows both loading and storing data; so there is no need to press any button after changes, since data is automatically stored immediately after insertion.

User can change status from editing to running through the two buttons placed one on the top and the other one on the bottom of the page having the label End test list definition.

This interface allows to change test list parameters, phones configuration and test cases: the following paragraphs will describe each part in details. Each part might be collapsed or made read only by clicking on the toggle buttons with minus sign and view only label respectively.

Test list details  As depicted in the higher part of figure 4.1 this interface allows to change test list name and scheduled execution date: user might insert an execution date or activate the As soon as possible radio button.

![Edit test list interface](image)

Figure 4.1. Test list editing interface: test list details and phones configuration
4.1 – Web interface implementation

**Phones configuration**  Allows to change the configuration of phones to be used in the current test list, as represented in the bottom of figure 4.1.

To add a new phone configuration the button with *Add phone* label must be pressed; when a new test list is created a phone configuration is automatically added. Buttons with the *delete* label on the right allow to remove the related phone configuration from the test list.

Each phone configuration allows to select the model between the available ones displayed in the list on the left side of the table. Clicking on a phone model, photo and platform will be updated according to the selection; moreover the available baselines, flexes, reflashes and customers are retrieved filtered by selected model and loaded into the corresponding drop-down menu.

The baseline, flex and reflash drop-down menus display the available files for each operation divided following four categories:

- predefined general: globally defined files available for all the users and working with all the platforms;
- user defined general: customized file uploaded by connected user and working with all the platforms;
- predefined platform specific: globally defined files available for all the users and working with the selected platform;
- user defined platform specific: customized file uploaded by connected user and working with the selected platform.

Drop-down menus show only categories with at least one file and allow to manage uploaded files clicking on the button *Browse custom files...* on the right.

**Test cases**  As depicted in figure 4.2 the list in the left side shows available test cases while the one in the right side the test cases to be executed in the current test list. At the beginning both lists are empty because no test case group has been selected yet and no test case has been added to the current test list.

First user must select a test case group from the drop-down menu: the left list will be automatically filled with the available test cases in the selected group compatible with the first phone configuration. User can add test cases to the current test list selecting them in the left list and pressing the button with the arrow right oriented: selected test cases are automatically added at the bottom of the right list.

User can change order to test cases in the current test list selecting test cases in the right list and clicking the button with label *move before* or *move after*; then user must select a test case in the right list in order to choose the point where to move the selected test cases.
4.1.5 Custom files management

Clicking on one of the buttons with label *Browse custom files* user can access to the interface allowing the management of uploaded files depicted in figure 4.3. Uploaded files are divided by platform, corresponding to one or more phone models, and by file type.

Using the first drop-down menu user might select a platform or choose the item *all platforms* for browsing not platform specific files. In the second drop-down it’s possible to select the file type to browse: *baseline*, *flex* or *reflash*.
4.1 Web interface implementation

Manage your custom files

![Custom files management interface](image)

The list shows user uploaded files for the selected platform and file type. Using the buttons with labels Delete and Rename it’s possible to remove items from the list and change their names.

Finally the input box with the label Add a new file allows to upload new files: user must choose a file from computer through the button Browse... and then press on the button Upload; once file upload is completed, an input box is displayed allowing user to change the name of the new file.

4.1.6 Test list execution interface

This interface allows to obtain information about a test list not in editing status.

As depicted in figure 4.4, the structure of the interface is similar to the test list editing one but through this one user cannot change data.

The first two parts summarize test list and phones configuration provided by user; moreover in the first part it’s possible know the status of the current test list, for instance if it’s waiting or running.

In the third part there is a table containing description and status of the test cases in the current test list. Moreover near test cases in running status a bar is displayed informing user about the progress. Both status and progress information
are updated every five seconds through AJAX technology.

At the end of test list execution test list status is changed into done and a button appears in the first part of the page, allowing to download the generated MMILogFile.

### 4.1.7 Logs download interface

Through this interface administrative account can know information about happened errors, such as the type of caught exception and the error string, and download the following files:

- **stack trace**: this file contains the stack trace when the exception has been thrown; in this way administrative users can known the exact point where error has happened;

- **test station initialization and running logs**: these files contain logs written respectively during test station loading and execution;

![Figure 4.4. Test list execution interface](image-url)
4.2 Description of the exploited technologies

- test controller initialization and running logs: these files contain logs written respectively during test controller loading and execution.

An example of this interface is depicted in figure 4.5.

<table>
<thead>
<tr>
<th>LogId</th>
<th>TestId</th>
<th>Type</th>
<th>Message</th>
<th>Stack Trace</th>
<th>TestStation</th>
<th>TestController</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>2514</td>
<td></td>
<td>PHONE_CONNECTION_LOST</td>
<td>IOException, enable failed, phone #3 is not connected or part is being tested</td>
<td>download (4 KB)</td>
<td>not available</td>
<td>download (4 KB)</td>
<td>2007-10-26 10:10:55</td>
</tr>
<tr>
<td>2515</td>
<td></td>
<td>TESTSTATION_CONNECTION_ERROR</td>
<td>not available</td>
<td>download (4 KB)</td>
<td>not available</td>
<td>download (4 KB)</td>
<td>2007-10-26 11:07:40</td>
</tr>
<tr>
<td>2516</td>
<td>04</td>
<td>PHONE_CONNECTION_LOST</td>
<td>IOException, enable failed, phone #3 is not connected or part is being tested</td>
<td>download (4 KB)</td>
<td>not available</td>
<td>download (4 KB)</td>
<td>2007-10-26 11:56:38</td>
</tr>
</tbody>
</table>

Figure 4.5. Logs download interface

4.2 Description of the exploited technologies

4.3 Java RMI technology

4.3.1 Introduction and goals

Java RMI technology provides to Java developer a new paradigm for network applications also referenced as the world of distributed object computing [2]. The RMI name is an acronym of Remote Method Invocation.

The goal of this technology is allowing the programmer to use remote objects like local ones. This allows to easily develop network applications involving several Java virtual machines executed on several hosts of the network. In this way developer can leave out the implementation of network communication and focus his attention on functionality of the application itself.

The only differences between normal Java objects and remotely invocable ones are:

- each remotely invocable object require to implements an own interface defining all remotely invocable methods;
• each remotely invocable method throws a `RemoteException` in the event that network troubles are encountered;

• object creation requires beside `new` operator invocation, `Naming.rebind` call on server side as well as `Naming.lookup` call on client side.

4.3.2 Architecture

As show in figure 4.6, Java RMI uses a traditional architecture wherein there are two actors:

• a server: offering a service;

• a client: requiring a service.

Each remotely callable class on the server side must implement a shared interface with the client. On client side a class implementing the same interface is automatically build by Java compiler.

4.3.3 Skeletons and stubs

These classes are automatically created by Java compiler in order to make easy the use of this technology.

A `Stub` class is a particular proxy class running on the server and listening to the network for requests, forwarding them to the real implemented methods and giving back the return value to the caller.
4.3 – Java RMI technology

The counterpart on the client side is called Skeleton, that is a Java class mapping remote class methods. When a client application calls a method of this class the call is transferred on the communication channel to the stub class; at the end of stub execution, return value is given back from this class.

4.3.4 Serialization

Serialization is “the process of converting a set of object instances that contain references to each other into a linear stream of bytes” [3]. It’s used by Java RMI technology in order to transfer data between virtual machines.

Both parameters and return values of remotely invokable methods must be serializable; for this purpose they must implement Serializable interface. Only native types, like int or boolean, don’t need to implement this interface.

Objects implementing Serializable interface must contain all native or serializable data types and define a serialVersionUID, that is a version identifier allowing to check if both client and server use the same class version.

Otherwise they must implement two methods, writeObject and readObject, in order to define how object serialization must be done.

4.3.5 Transport Layer

Connection between virtual machines is done through Java Remote Method Protocol (JRMP) on the top of TCP/IP network protocol.

JRMP protocol is a proprietary stream based protocol but some alternate implementations are also available.

4.3.6 Remote objects binding and naming

The list of remotely invocable objects available on a host is maintained by a Java thread called RMI registry.

When a new instance of a class is created it must be bound using the static method rebind of the class Naming; moreover a service name string must be provided in order to allow to access this object.

When client wants to access to a remote object, static method lookup of the class Naming is invoked with host address and service name.

4.3.7 Distributed garbage collector

As usually in Java a garbage collector is available, responsible of deallocation of no more referenced objects. In Java RMI technology garbage collector has been extended in order to support remote referenced objects.
For each remotely invocable object, server keeps track of clients accessing it and when all the references have been dropped related memory is released.

Since this system works on a network and connection to clients could be dropped, every reference has a lease time set by default to 10 minutes; if the client doesn’t refresh the connection before this time expires the reference is dropped.

4.3.8 Java RMI use in the developed system

![Diagram of Java RMI interactions in the developed system]

In the developed system, test controller and test station are both Java RMI clients and servers, as shown in figure 4.7.

Beginning from the top of the figure the first arrow represents the main connection between test controller and test station. This connection is created by test controller when test station is connected to the system. Afterwards test station creates a feedback connection to the test controller.

The second arrow represents the connection between test controller files storage and test station one. This connection is created on test station initialization and it’s responsible of files transfer between test station and test controller.

The last arrows in the figure going from test controller host to test station one, represent the connection between remote station object and phones available on the test station. These connections are created on phone loading and are used in order to access directly to phones, e.g. to enable or disable phone, to reboot it, etc.

The remaining two arrows represent a virtual connection between FileStorageProxy objects on phone and FileStorage one on remote station. Really FileStorageProxy is only a proxy object to FileStorage one contained inside the test station one; in this way it’s possible to easily use a different work directory for each phone.
4.4 AJAX technology

“The intent is to make web pages feel more responsive by exchanging small amounts of data with the server behind the scenes, so that the entire web page does not have to be reloaded each time the user requests a change.” [4]

Ajax, that is an acronym of *Asynchronous JavaScript and XML*, has been used on web interface in order to improve usability. As shown in figure 4.8, Ajax starts playing when normal HTTP page request ends.

Ajax technology is implemented through a series of Javascript methods, allowing to open background connection with server in order to upload and download data from it. This method is invoked on user request or when a timer elapses.

![Ajax technology timeline example](image)

Figure 4.8. Ajax technology timeline example

4.4.1 AJAX technology usage in the developed project

The developed project uses Ajax technology in the following way:

- on page loading JavaScript code is executed and a Javascript object responsible for communication with server is declared;

- when an update or data storing is required a new instance of Ajax communication object is created and a request to server is done with the following parameters:

  - namespace: a string defining the context of the request;
- namespace id: a string defining the referenced id inside the namespace (e.g. for testlist namespace, test list id is used);
- change: a string defining which type of change is required, if any;
- change parameters: depending on type of change required, allows communication to server of data retrieved from user to be stored;
- requests: an array of strings used for requiring data to server;

- when server receives a request to a particular page, called _ajax.php_, checks user authentication data\(^1\); on success, it updates the database according to given parameters and provides the required data to the user through a runtime generated XML document, like one in figure 4.9.

```xml
<ajaxreply sleep="5000" namespace="testlist"
request="testlist.detailed.full,phoneconfig.detailed.full,testcase.detailed.full">
  <request name="testlist.detailed.full">
    <testlist id="28" name="Test" priority="0" executeOn="" created="2007-10-26 17:00:13" status="running"
mmlogfile="0" started="2007-10-26 17:30:12"
flashtime="0"/>
  </request>
  <request name="phoneconfig.detailed.full">
    <phoneconfig id="6" testlist_id="28" phonebundle_id="9" phonemodel_id="1"
base_line_id="9" flex_id="3" reflash_id="0" customer_id="1"
phonemodelname="RAZRV3xx" customername="TIM"
baseLineName="Q651190_96.80.006R.sbf" flexName="TIM flex v3xx.hs"
flashname=""/>
  </request>
  <request name="testcase.detailed.full">
    <testcase id="1091" testcentralid="App.Multimedia.Video Telephony:009-012"
description="Outgoing Video Telephony calls." status="passed"
progress="1.00"/>
    <testcase id="1092" testcentralid="feature.Multimedia.2P Video Telephony:009-026"
description="Remote terminal releases the call during the Video Call
Connecting State." status="failed" progress="1.00"/>
    <testcase id="1093" testcentralid="feature.Multimedia.2P Video Telephony:009-027"
description="Remote terminal switches off the phone during Video Call
Connecting State." status="passed" progress="1.00"/>
    <testcase id="1094" testcentralid="feature.Multimedia.2P Video Telephony:009-028"
description="Selecting the Camera OFF in Video Call Connecting State." status="waiting" progress="0.00"/>
    <testcase id="1095" testcentralid="feature.Multimedia.2P Video Telephony:009-029"
description="Selecting the Mute Audio in Video Call Connecting State." status="waiting" progress="0.00"/>
  </request>
</ajaxreply>
```

Figure 4.9. Runtime generated XML example generated during test list execution interface loading (see section 4.1.6).

---

\(^1\)for further details on authentication, see paragraph 3.5.2
when browser receives XML structure a JavaScript read contained data and updates web page without need to reload it.

This technology allows the project website to:

- edit test list details without need to reload page (name, scheduled date);
- add phone configurations, browse available customers, baselines, flexes and reflashes by model and store updates;
- show available test cases by test case group and edit test list entries;
- update test cases status during test list execution;
- manage custom files uploaded by users.

In figure 4.10 a typical update and data storing outline is shown.

Figure 4.10. Java RMI interactions in the developed project
4.5 Session variables

Session variables are used to store information across subsequent page executions. In this way it’s possible to add state information to HTTP protocol that is stateless.

4.5.1 PHP session variables implementation

Usually each variable assigned in a PHP script is accessible for less than a second and is dropped when script execution ends. This is a problem when it’s requested to store user related information between subsequent executions.

Rather when a variable is registered as a session one, PHP automatically creates a file into webserver containing variable data and assigns to them an unique alphanumeric code of 32 digits. This code is stored as a cookie named PHPSESSID that browser automatically provides to web server in the following requests. Server when receives a valid PHPSESSID cookie, creates a variable reading the content from related file.

This type of cookies are usually stored with an expiration date in the past; so when user quits the browser PHP session id is lost and session data are unrecoverable. In this way if session variables are used to provide user authentication, at the next access to website login data will be required again.

4.5.2 Security related aspects

With session variables it’s possible to recognize user without storing any critical information on his computer: in fact malicious users cannot change session data because they are stored into server. They can only change the PHPSESSID cookie but it’s quite hard to match another valid session id, preventing user from replace himself with another one.
Chapter 5

Conclusions
5.1 Achieved goals

The testing system has been designed and implemented entirely in all the described parts by the student proposed as candidate.

Some work should be still done in order to allow the system to be exploited in the everyday working experience: the main job is testing the system with a lot of test stations and users at the same time and adding additional improvements to the web interface.

Moreover test station process, developed as a Java application, should be enclosed into a service allowing the start at the operating system boot and the automatic restart when application exits.

Finally Java RMI should be secured through the authentication of the remote part.

5.2 Next Steps

Next steps for the project are listed in the following paragraphs.

Customized test cases execution With this new feature it will be possible to upload and execute test cases. Customized test cases are subdivided into three kinds, depending on the format of the provided test cases:

- ASCII test case: a list of phone keys to be pressed;
- binary test case: a Java testing class;
- jar test cases: an archive containing Java testing class.

Moreover an interface allowing the definition of ASCII test cases should be added to the web interface.

Actually tables needed by this feature have been already designed and implemented into the database.

Developing of libraries allowing external processes to create new test lists Through this new feature, together with the previous one, it will be possible to add new test lists containing binary test cases generated through $\mu$GP$^3$.

Automatic download of software releases Software releases should be downloaded automatically from hosting company web site on user request; in this way it is no longer necessary that the software releases are downloaded by hand and then uploaded on the project web site.
Actually tables needed by this feature have been already designed and implemented into the database.

**Test controller proxy**  When some test stations are far from the test controller a new component should be introduced providing caching features in order to reduce large files transfer time. In this way all download requests will be addressed to the test controller proxy that retrieves the requested file from local cache if available or forwards the request to the test controller otherwise.

**Integration with the system managing CRs**  Interfacing the developed project with the system used by hosting company in order to manage change requests to mobile phone software. In this way it will be possible to learn the change request related to the failed test cases.
Bibliography