BACHELORTHESIS

Guidelines for the development of a GUI-regression-testing-framework for SCADA-applications using the example of the JCOP-Framework as used by the CERN LHC and its experiments

to obtain the degree of

Bachelor of Science

(BSc.)

written at

Research Group for Industrial Software (INSO)

at the Vienna University of Technology

under the supervision of

Thomas Grechenig

by

Stefan Klikovits

Ignaz-Till-Gasse 23, 7011 Siegendorf

stefan.klikovits@gmail.com

Geneva, 21-Nov-11

Purpose of this document

This bachelor thesis was written by Stefan Klikovits during his internship as a „Technical Student“ at CERN [A]. It is reflecting about the experiences made during the stay and helps the reader to get going faster with their own developments.

1.1 Aim of this paper

The aim of this bachelor thesis is to give a theoretical guidance in developing and integrating a GUI-test-framework.

Its content is derived from the experience gained during the development of the „Emu“-framework, which is used by CERN for the testing of the user-interface of the „JCOP-Framework“ [i], which provides „an integrated set of guidelines and software tools which is used by Developers of a Control System to develop their part of a Control System application.“ [1]

The lessons learned from this project are collected in this thesis, so that others can draw conclusions and use them for creation of their own testing frameworks.

1.2 Structure of the paper

This thesis comprises of several main parts.

The first part describes the analysis of the given environment (e.g. already implemented tests), the environment used for these tests and the global environment (other software used by the development-group, e.g. wikis, bug tracking software, etc.)

The next part is the requirements-analysis, using the input of the main stakeholders, which are the test-developers, the software-developers and the management-level (section-leader).

After that section there follows a more practical part, which describes one by one the integration of each requirement, together with the lessons learned and the conclusions drawn.

The last part of this thesis summarizes all of the lessons learned and aims to give a generic guideline for future projects. It should be considered as an introduction for developers, who are new to this topic, as well as a knowledge base for people who already have experience.

1.3 Methodology of the derivation of information

The information provided in this paper is derived from the practical experience gained during the development of the „Emu“-Framework. As the development of a framework with this specific combination of software is not described in any available doc-

uments or papers, most of the decisions were made after analysis and theoretical reflections.

After the requirement analysis there will be a description of the implementation and of the difficulties encountered.

As this paper aims to give the reader insight into the reasoning-process of the developers and into the implementation-phase, it will also give descriptions of dead ends and try to give hints for a better development and prevention of these misdevelopments.

Table of Contents

Table of Images

2 Basics and prerequisites for understanding this paper

The following chapter explains several terms and expressions used in this paper. It includes a part about the basic set of programs used and the background of the software development process.

2.1 Definition of Terms

This chapter explains the major terms used throughout the paper. If a term is not explained here, the reader may find a short explanation in the Glossary at the end of this thesis.

2.1.1 CERN

The „Conseil Européen pour la Recherche Nucléaire“, better known as „CERN“, is the world's largest particle physics laboratory. It has been „home“ for many particle accelerators and nowadays the Large Hadron Collider is run there. More than 50 % of the world's particle physicists are (either directly or indirectly) connected to CERN's research results.

2.1.2 Large Hadron Collider & Experiments

The „Large Hadron Collider“ [B] is the world's largest particle accelerator. LHC's tunnel is 27 kilometers long and about 50 to 150 meters beneath the Swiss-French border. The temperature of it's superconducting magnets, which can create a magnetic field of up to 8.6 tesla, is constantly kept at 1,9 Kelvin (-271,25 °C).

LHC's main purpose is to accelerate particle beams up to 14 TeV[1]. After the acceleration phase, the particle beams reach over 99,9999 % of the speed of light and get collided in one of four detector points of the tunnel. At these points “experiments” are installed, which measure the data when collisions happen.

The six Experiments are „Atlas“, „Alice“, „CMS“, „LHCb“, „LHCf“ and „TOTEM“. Each detector measures different data and has a different purpose.

LHC is run by CERN, but Universities, Research-Companies and other institutions finance the experiments. CERN only pays a part of the Experiments' expenses.

The amount of data acquired each day is enormous and managing the information-stream is a very important conciliation. Atlas for example is producing 3200 Terabyte

[1] „Teraelectronvolt“, physical unit measuring energy, 10^{12} eV

of data per year. This data is the raw data remaining after three levels of filtering have already been applied.

The other important part is the control and monitoring of the system status. The big LHC-Detectors have over one million Input-Output-Channels which all have to be read and set remotely, as there is only a rare possibility to access the tunnel. [3]

2.1.3 PVSS

PVSS [ii] is a SCADA-application [C], which is used by CERN for monitoring and controlling of some LHC subsystems as well as its experiments. It is the toolkit on which the JCOP and UNICOS [iii] frameworks are built.

2.1.4 JCOP and the JCOP-Framework

JCOP stands for „Joint COntrols Project“. It is a grouping of representatives from the four big LHC experiments and aims to reduce the overall manpower cost, required to produce and run the experiment control systems.

The JCOP-framework is a layer of software components, produced in collaboration, where the components are shared. It is made using common tools, such that the components work together [4]. It is based on PVSS and future plans foresee a more intensive use of the framework not only for the experiment, but also for other controls systems at CERN and elsewhere.

Its main purpose is the reduction of development effort by reusing components, by hiding their complexity and by reducing resources for maintenance and the provision of a higher level of abstraction.

2.1.5 UNICOS-Framework

„UNICOS is a CERN framework developed to produce control applications for three-layer industrial control systems. UNICOS provides developers with means to develop full control applications and operators with ways to interact with all items of the process from the most simple (e.g. I/O channels) to the high level compounded objects (e.g. a sub part of the plant). In addition UNICOS offers tools to diagnose the process and the control system.“ [5]

This framework is based on PVSS and built upon the JCOP-framework.

2.1.6 Squish

Squish [iv] is the GUI-testing-tool used at CERN. It allows access to the user interface and permits doing interaction. It is also possible to verify the correctness of data by doing verifications.

2.1.7 Python

Python [v] is a modern high-level programming language, which allows its users to follow different programming paradigms such as object-oriented programming as well as imperative programming. It is also possible to use functional programming, although it is not very well developed in this direction.

It is developed and maintained by the Python Software Foundation and currently available in version 3.2.

2.1.8 Emu-Framework

Emu is the name of a Squish-harness used by the CERN EN-ICE section to develop GUI-tests for verifying the correctness of the JCOP and the UNICOS frameworks.

2.1.9 AUT, PUT

„AUT“ is Squish's abbreviation for „Application Under Test“. „PUT“ is CERN's abbreviation for „Program Under Test“. Both terms have the same meaning and can be exchanged. In our case the AUT/PUT is called PVSS00ui.exe, which is PVSS's GUI-application, and is started with several arguments, which define, which panel (the canvas, containing all the UI-Objects) should be started and some other parameters like menu-bars, etc.

2.1.10 Emu-Library

An „Emu-Library“ (also „Library“ or short „Lib“) is a Python-file containing one or more classes (see object-oriented Python). These libraries are logically divided into two interaction layers. One layer provides low-level interaction routines for accessing UI-widgets like buttons, textfields, tables or (context-) menus. The second layer is providing routines for doing more complex interactions which include several steps. It relies on the functionality of the first layer.

For example the library „guiObjects.py“ contains a class called „QtButton“ which provides methods for clicking, reading the text or checking the enabled-state of pushbuttons.

An example for a second-level-library would be „paLib.py“, containing a class called „ProjectAdministration“. Its methods allow for example the parameterized creation, starting/stopping or removing of projects.

The purpose of this library structure is, that the low-level routines should be implemented by „core“-programmers, who know about the interaction between Squish and Qt. Test developers then use these first-level-libraries for creating reusable second-level-libraries, which allow the rapid development of testcases and testsuites needing only a minimum knowledge of Qt.

2.1.11 Testcase, Testsuite, Verification

As the development of tests should happen in a coordinated and uniform way, it is necessary to define the terms „testcase“ and „testsuite“.

In the Emu-framework a testcase is a script, which defines a procedure for several GUI-interactions. After the execution of each command it should verify that execution succeeded by doing one or more verifications.

Verification can happen through different comparison-routines such as screenshot-comparison, value-comparison (texts, Booleans, numbers) or file-comparison. This comparison, which always results in a Boolean value, (“pass” or “fail” of the verification) is also called a “verification point”.

A testsuite is a bunch of logically related testcases. For example all testcases that verify the correctness of a particular component should be put together into a single testsuite. In this way running this testsuite will allow the test-executor to have a look at the results and check if the functionality of the entire component is still maintained.

2.2 Used (employed) Software (for the test)

The following section will give a short explanation to all the software which is used by the Emu-Framework.

2.2.1 Operating-System-Level

The following operating systems are the targeted platforms for the Emu software. These operating systems are officially supported by the Emu-team and it is planned that in the future the tests will run on all of them.

2.2.1.1 Microsoft Windows

The main operating system at CERN is Windows. The IT-department currently supports the versions Windows XP [vi] and Windows 7 [vii]. Windows XP is usually run with Service Pack 3. Windows 7 is distributed with Service Pack 1.

Windows Vista is not supported at the moment and it is very unlikely that there will be official Windows Vista support in the future.

As mentioned before, the operating systems are chosen by the IT-department, which does the maintenance of CERN computers.

The machines, which are used for the Squish testing are virtual machines (VM). The reason for this is, that VMs get backed up by the Computer Center, which takes the responsibility from test developers and test managers. This method is also cheaper because the VMs use resources only briefly which frees resources for other tasks/machines whereas physical machines would just run idle.

2.2.1.2 Linux (Scientific Linux CERN)

„Scientific Linux CERN 5 is a Linux distribution build within the framework of Scientific Linux which in turn is rebuilt from the freely available Red Hat Enterprise Linux 5 (Server) product sources under terms and conditions of the Red Hat EULA. Scientific Linux CERN is built to integrate into the CERN computing environment but it is not a site-specific product: all CERN site customizations are optional and can be deactivated for external users." [6]

The current version supported by CERN is SLC5 [viii]. SLC4 is - in theory - supported as well, but the plan is to move all machines to SLC5 by the end of December 2011. There is also some preparation for the next version (SLC6).

2.2.1.3 Communication with the OS

The communication with the operating systems is handled by Python (Version 2.7). For executing system-calls, file and file system operations the main usage way are Python-Scripts which are called either from the command line [xiii], shell or Hudson.

The main reason for using Python is, that it is possible to use the same code on any platform supported by Python and it takes responsibility from the Emu and test developers, as normally there are no compatibility-problems when using python code on different platforms.

2.2.2 Software used at testcase level

The following software is used for the tests. All of the following software is either tested or directly used for testing the software.

2.2.2.1 PVSS[2]

PVSS is an abbreviation for „Prozess Visualisierungs- und Steuerungs-System“, a program written by the Austrian company„ETM Professional Control“. It is a SCADA-Application, which supports the control of different projects such as production-sites (including machines), facility management, water-management, etc.

This is done by acquiring data from Input-devices and sending commands to Output-devices. The data which is acquired, is usually displayed on a computer-screen and commands are sent following operation action, which often invokes background functions.

The reason for using software like PVSS is, that it is very simple to abstract the technical part from the management part. The user does not need to know the specific hardware setup, he only needs to know which parameters he wants to be set, rather than how to configure the device itself.

Another reason is that with interactions with simple GUI-Objects, large tasks can be fulfilled. For example it is possible to set thousands of parameters, values or „real“ objects like valves with very few interactions.

In most cases the user interface of these applications aims to recreate the look of the real production site, by displaying screenshots, objects or other information, which helps to know which machine, is controlled by different UI-Objects.

[2] The latest version of the Software is not called “PVSS”. The new name is “WinCC Open Architecture”. As CERN did not migrate to the latest version yet and the software in use is still called ‘PVSS’, the program will be called ‘PVSS’ throughout this thesis.

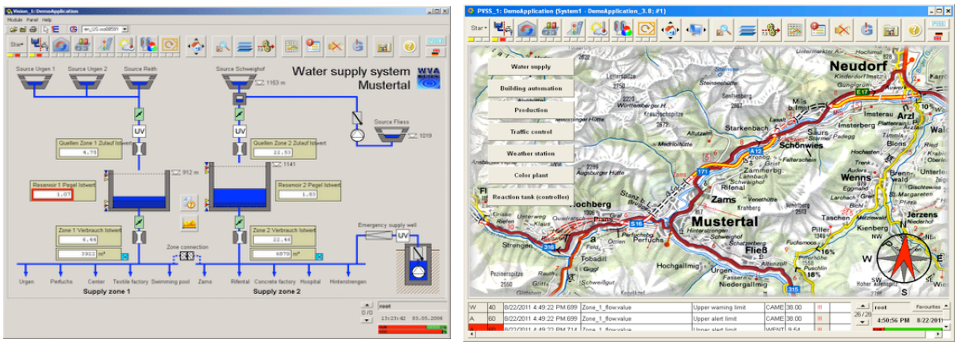

Figure 1 – PVSS user interface examples[3]

2.2.2.2 JCOP & UNICOS

The PUT (program under test) is PVSS. More specific the JCOP and the UNICOS frameworks are tested.

JCOP framework is „*an integrated set of guidelines and software tools which is used by Developers of the Control System to develop their part of the Control System application.*“ [7] The framework consists of 2 major parts.

The first part is a set of guidelines and rules which should be considered when developing software. These guidelines help to develop a homogenous operation safety system and make the software more maintainable.

The second part is a collection of functions, panels and standard elements, which can be used in the development of new control systems. This collection is built and maintained by the EN-ICE group and, to a minor part, in collaboration with the experiments.

UNICOS stands for „UNified Industrial COntrols System“ and is a framework for industrial automation components. The aim of the framework is to provide a homogenous interface for the control of hardware from different producers, which normally have different control systems. This software offers a special kind of abstraction which masks out certain peculiarities of single components. It is based on PVSS and uses some of the JCOP framework components.

[3] Imagesources:
left image (22.08.2011 - 16:35):

http://pc.wednus.com/_/rsrc/1303317197254/Home/kb-1/spm/pvss00ui.png?height=318&width=400

right image: PVSS DemoApplicattion (PVSS 382)

2.2.2.3 Squish

Squish is a program, which allows user interface testing. It was originally made to test Qt user interfaces (UIs), but expanded its possibilities to several different user interfaces like for example webpages, Mac, iOS and Java UIs.

The software allows writing testscripts in four scripting-languages and is available for Windows, Linux and Mac OS X.

These features make it a very good choice for the GUI-tests of PVSS-applications, because it is possible to test PVSS's Qt-UI on Windows and Linux, which are used by CERN and supported by PVSS as well.

At the moment there is no other tool, which provides comparable features for Qt-Testing. Squish provides ObjectMaps (identification of objects by properties), Spying (selection of a widget of the running application) and the recording of testcases.

Squish is manufactured by Froglogic, a company based in Hamburg, Germany, which was founded by former Qt-Developers.

2.2.2.4 Python

On the testing-level python is used in two different ways. First it is the scripting language used to run and control the procedural testscripts of Squish's testcases.

Secondly, we have built the object-oriented libraries providing reusable functionality to the user.

2.2.2.5 Hudson & Jenkins

Hudson [ix] is a free Continuous Integration tool supported by Oracle. It is distributed under the MIT license. In the beginning it was used for Java projects but due to the big community there are lots of plugins and extensions to make the software more configurable.

In February 2011 the Hudson community split into two projects [8]. Whereas the Hudson project is still supported by Oracle and is going to keep the name, there is a second project called "Jenkins" [x]. Jenkins has the same code base and the only obvious differences are the name and the logo.

Until now there are only small differences and the functionality of both projects is considered to be equal. Most of the available plugins work on both systems and therefore it is not possible to choose one project for technical reasons only.

2.2.3 Other tools & software

The following software is not directly involved in the test-process but also has a major role for the development of the framework.

2.2.3.1 SVN

SVN (short for „Apache Subversion“) is a software versioning system. The purpose of this software is to maintain the source code of a program. Developers can „checkin“ to submit changes in the source code and „checkout“ to update the local copy of the source code. It is also possible to checkout a previous version of the software.

2.3 Development of the PUT

One of the biggest frameworks produced in the EN-ICE group is the JCOP framework. Its software development process is based on an adapted waterfall model. [12]

Each component starts with a short requirements document. This document is a simple file containing the requirements of the future component in short form and very informal.

This document is reviewed by the „Framework Working Group“. This group contains developers from each CERN group which uses the JCOP framework.

After this review is over a first implementation proposal is created. This can be either a detailed document or a prototype or something similar. This implementation proposal is reviewed again and changed if necessary.

After having an agreement on the proposal the implementation phase starts. This phase is followed by a testing bit and a documentation phase. The last phase (not in the diagram) is the „Support and maintenance“ phase in which bugs are fixed and change requests are managed

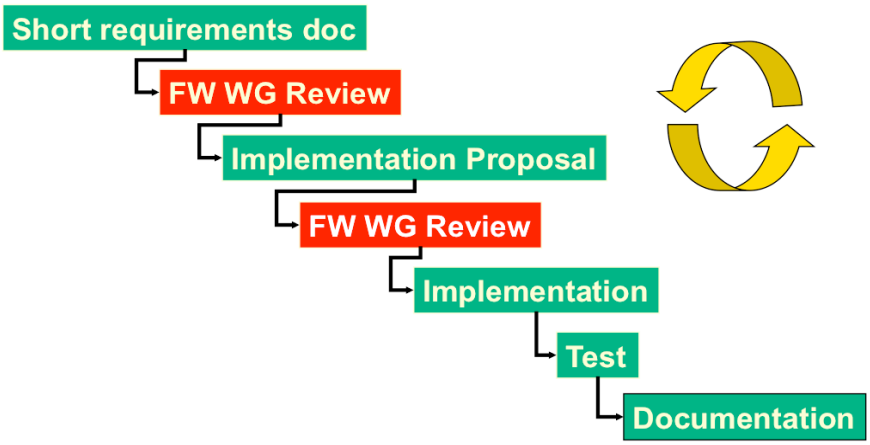

Figure 2 - Software development process

The maintenance and support phase is well structured. Each support call is registered in an internal bug tracking system. Each issue is analyzed by the designated support person, who has the following options:

- In case of a request for information he either answers this issue himself or dispatches the issue to an expert (usually one of the developers) with this topic.

- In case of a problem (bug), he opens a so-called “change request”, which contains the necessary information for reproduction of the problem.
- In case of a feature request, he opens a change request as well, but this request is first discussed with the section leader. If the requested feature is going to require lots of resources then it is also discussed with the Framework Working Group.

3 Requirement analysis of the Emu framework

3.1 Stakeholders' points of view

There are several stakeholders involved in testing the PUT. At the beginning of the Emu-project a small group of people (2-4) cover basically all roles. Therefore it is possible that a test-developer is responsible for developing tests, maintaining the libraries and also covers the role of the test-manager.

3.1.1 Test developer requirements

There are two groups of test-developers. One group is the PVSS developers who also write tests for verifying the correctness of their components. The other group consists of explicit Squish-developers, who are only responsible for writing tests. The specification for their work is normally obtained by asking a PVSS-developer for specification. They are also responsible for building and maintaining the library structure, which is used by the „real" test-developers.

3.1.2 Program developer requirements

The program-developer merely requires verification of his program after he changes code. The tests should be run soon after the check-in so that the program-developer gets results quickly and either has the validation confirmed or information about new problems in his code returned.

If the program-developer does not write the tests on his own, it is necessary for him to write a detailed specification so that another test developer can write the test for him.

3.1.3 Test manager requirements

The test-managers' role is to handle the rollout of Emu on different machines (operating systems, PVSS-versions, framework-versions) and to ensure the correct execution of the test suites. He is responsible for keeping track of the errors.

It is his responsibility that the tests run on a regular basis and that the developers get their test results. He is also in charge of chasing people if there are errors in their tests which don't get fixed for a certain amount of time.

3.1.4 Section leader requirements

The section leader wants to have the possibility to have a look at the current test results, to confirm that everything is alright. In the normal daily routine he only acts in a supervisory role and does not actually take part in the testing process.

3.2 List of requirements

The following list contains the points that are planned for the first version of the Emu-Framework. The requirements were extracted from talks with test developers, the section leader and test managers as well as the EmuCore-developers.

- The framework should give the possibility to implement tests quickly, without knowledge of the detailed Squish-PVSS-interface and the routines, which are used for accessing the Qt user interface.
- The written code should be re-usable. (Therefore a library-structure is going to reduce/eliminate the necessity of duplicate code and make re-use possible.)
- The test managers need to be informed about the test results. They should therefore receive a summary by email. This email should contain the main results (positive, negative), but not directly include all the details. The test manager only dispatches the bug fixing tasks to the test developers.
- A test developer should only receive emails for tests he is responsible for. This email should contain more information regarding the developer's testsuites, so he gets a detailed description of the fails. Other testsuites should be mentioned only in a summary (number of tests, number of positives, number of negatives).
- As JCOP and UNICOS are run on different platforms, the tests have to support both of those platforms. Therefore the tests should be stable enough for running them with different OS-, PVSS- and framework-versions.
- The rollout of the Emu-framework on new target machines should be uncomplicated. Even users, who are not familiar with the Emu, should be able to perform this task given suitable instructions. A document containing the details of the rollout should be provided.
- Emu needs to be controlled from a central point. The best-case scenario is, that a single click triggers the test-run on all defined target machines. If this implementation is not possible, the triggering should be as easy, and the management of the tests as convenient, as possible. The implemented system should preferably have a web user interface for easy access.
- The tests should be stored together with the program being tested, in the same directory. The testing libraries should be managed in a separate repository. Hence the development of the EmuCore and the development of the tests can be handled separately. Also, the test developers have easier access to their suites, as the suites are stored together with the components they are developing.

3.3 Mapping requirements onto development steps

The development until the current state of Emu was divided into several milestones.

1. Change from many technologies to one
2. Development of a library structure
3. Evaluation of Hudson/Bamboo as well as subsequent integration
4. Changing the structure from client server to remote-target test runs (also on different OS)
5. Changing the structure so that the tests are stored together with their components and thus separate off the Emu core

The development of new tests should not be interrupted by these developments. It is necessary to be especially careful not to interfere with the tests which already run daily.

3.3.1 Reducing the number of used technologies

Until the start of this project, there were many programming languages used for running Emu. This led to incompatibilities on different platforms as well as messy code and required knowledge of all these languages to be able to maintain Emu. The aim of this milestone is to change this scenario so that the number of employed technologies is as small as possible, whilst being able to interact with as many platforms as possible. In the best case there should be no OS-dependencies in the code.

3.3.1.1 Requirements

- The number of used technologies should be reduced to a minimum
- The developed code should be portable and run on all (JCOP-) supported platforms and operating systems
- The scripts and code should be „readable“. This means that all the developments should follow the department’s programming guidelines as far as possible and also follow a consistent style where the guidelines are not applicable or defined

3.3.2 Develop a library-structure

The development should have several libraries, which can be loaded individually, so that there are no dependencies. Libraries should make code-reuse possible and maintenance faster. The main purpose of these libraries is to offer code-reusability. They must also be easy to extend. A „nice to have“ feature would be the derivation of classes, so that test developers can derive their own special classes from the general classes, which are implemented in the Emu-core.

3.3.2.1 Requirements

- The libraries should be as independent as possible.
- The written code should be reusable at other points of the software
- In case of unavoidable dependencies, there should be as few of them as possible and no circular dependencies
- The library structure should promote rapid testcase development
- Testsuite maintenance should be simplified by reducing the number of instances of redundant code

3.3.3 Easy management of many tests

Emu should be managed and triggered from a central point. It should be possible to look at the results without physical access or a remote-desktop-connection to the target machine. After the tests have run, e-mails should be sent out, which contain a summary of the latest results.

3.3.3.1 Requirements

- It should be possible to manage Emu through web access
- The software should display results in a clear way
- The tests should be automatically triggered through the software (possibly in a parameterized way)
- The results of test-runs should be sent by e-mail to the developers

3.3.4 Run tests on remote machines

As the number of tests is very likely to increase, it should be possible to outsource portions of a test-runs to other machines. This behavior is compatible with running tests on platforms other than the one from which the tests are triggered. It would be nice if the system could run the tests automatically on a free machine in a group of machines. Results should be independent of which machine the test is run on. That means that two machines, which are running the same OS, PVSS and JCOP, should have the same behavior and both be possible test-nodes.

3.3.4.1 Requirements

- The test-software (Emu) should be runnable on all supported platforms
- The behavior on all machines should be the same
- Nice to have: automatic shifting of test-runs from one target machine to another.

- Nice to have: run a test on one slave in a group of equal machines

3.3.5 Distributed development

After the Emu-development team has implemented the first tests and the test harness, the test-development process should be changed. Every test-developer should write, maintain and fix his tests independently. These tests should be stored together with the developed component. The Emu-nightshift scripts are responsible for checking out the test-suites from SVN and running them on the appropriate target-machines.

The EmuCore-developers will be responsible for creating, maintaining and extending the Emu-library structure, which is used by the test developers to write their tests. The results of a test should be sent to the developer(s) of the concerned component and not to all developers. Test managers should still get all test-results.

3.3.5.1 Requirements

- The Continuous Integration server should look for applicable tests in the SVN-Repository and run them
- The CI should support a trigger-mechanism where it is possible to specify which tests should run on which machines
- The tests should be stored together with their program-components in the repository
- Emu-library-development and test-development should be independent
- Emails should only be sent to the developers concerned and to the test managers. Developers, who are not involved in a certain component, should not get the test results for this component.

4 „Step by Step“ implementation of the requirements

The following part will describe the integration of the development-steps mentioned before.

4.1 Reduce the number of used technologies

Until the start of this project, there were many programming languages used for running Emu. This led to messy code and incompatibilities on different platforms. Furthermore it required knowledge of all of these languages to be able to maintain Emu. The aim of this milestone is to change this scenario so that the number of employed technologies is as small as possible, while being able to interact with as many platforms as possible. In the best case there should be not a single OS dependent sections in the code.

The first analysis of the existing testing-framework („old Emu“) showed, that further development, as well as maintenance had become difficult and messy and the existing code had lots of workarounds because of platform-differences. Due to step-by-step evolution Emu had to cope with a growing number of technologies. The latest version of the old Emu used more than five different types of scripts and tools on two different platforms.

4.1.1 Detailed interaction analysis of the existing Emu-code

Two sides of Squish

First it is important to know that Squish consists of two separated Applications, the Squishrunner and the Squishserver.

The Squishrunner is responsible for parsing a testscript and sending calls to the Squishserver. The Squishserver is responsible for receiving these calls and executing them. After execution it reports back to the Squishrunner. It is not necessary that both programs run on the same computer. The connection between runner and server can also be established over a network. This allows the execution of tests on a remote machine.

Who talked to whom?

Besides the communication between the two Squish-applications and the parsing of the test script, Emu had several other interactions between different scripts.

Figure 3 illustrates the different technologies, which have been interacting with each other.

At first, the tests were triggered on Linux-machines by using a Shell-script, which started Squishserver and Squishrunner. This script was also responsible for parsing some configuration-data which was stored in an external .ini-file.

For easier maintenance, under windows the tests were controlled by the same Shell-script. For making this possible, Emu was triggered by a „Scheduled Task“, which opened a batch-file. In this batch-file there was the call to the Shell-script, which was executed using the Cygwin [xi] compatibility suite.

There was also the need for more specialized functionality, e.g. file or folder operations, which were also implemented by using Linux-programs (Shell-commands, like „grep“, „diff“, etc.) to be able to perform file-comparisons during the test-execution or similar operations.

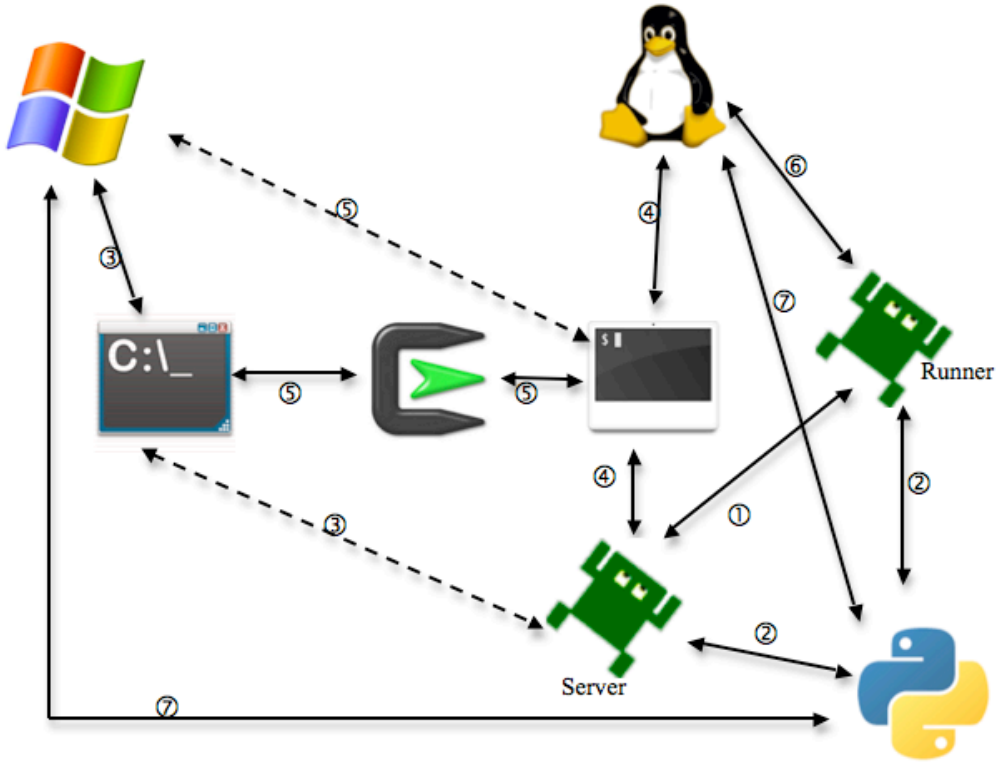

Figure 3 - Old communication diagram

*Diagram explanation (*Figure 3*)*

① Squish consists of Squishrunner and Squishserver which communicate with each other
② Squish executes Python-code
③ Windows: A Scheduled Task triggers the execution of a CLI-Script which calls the Squish-Tests
④ Linux: A cron-Job triggers a bash-job to execute Squish
⑤ For redundancy-elimination Windows calls bash-scripts too. For achieving this, a commandline calls Cygwin, which executes the bash-script.
⑥ Squish can also access the OS (execution of tools, Registry-operations under Windows)
⑦ During the execution of tests, Python-scripts also access the OS (file operations, diffs, etc.)

4.1.2 Problems of this approach

Cygwin != Linux

One of the major problems is that, although lots of the functions are available, Cygwin does not provide all of the functionality, which is available on Linux.

One example is the possibility of performing Perl-grep (grep with the „-P“ argument), which makes it possible to perform the grep command in a much faster way. This led to a very low performance of the tests on Windows-machines, so that the execution of this command was slowed down by a factor of at least 20.

OS-dependencies

There are several differences between Windows and Linux, which make maintenance and development more difficult than necessary. Every interaction with the operating system has to be checked for possible different behavior and very often it is necessary to implement the same routine separately for each OS.

One problem was the path-problem. Due to the fact that Windows uses backslashes („\“) in its paths and Linux uses forward-slashes („/“), it was necessary to perform several checks and write an OS dependent section of code to solve this problem.

Know-how-problem

Every one of these technologies has to be maintained, changed and/or extended. For doing this, it is necessary to have knowledge about the principles of these technologies. The more technologies there are used, the more knowledge there has to be, which leads to bigger expenses in terms of learning this knowledge.

4.1.3 New way of Emu communication

To solve these problems, we did an extensive analysis of the technologies used and looked at other scripting/programing languages. This led to the discovery, that Python was capable of solving most of these problems. Further, it would also be possible to use the same libraries as the tests.

Switching to Python also solves most of the OS dependent code problems, as Python takes care of transforming between the two slashes as needed. Many Python modules for operating system and file system interaction exist and can be used in the same way on all employed operating systems avoiding many potential incompatibilities.

The usage of Python moves the effort put into resolving OS dependent code from the Emu-team to the Python community. This reduces maintenance work and allows the team to put more effort into development of the actual testing-logic. Another advantage is that there is a very large and active Python-community with lots of scripts and tutorials for many different tasks.

New communication ways

The new technology-diagram (Figure 4) is much simpler. It uses Python as central interface between the operating system and Squish. The OS calls Python-scripts directly from a command line shell. There is no need to switch between scripting languages. It is possible to use the same libraries as the tests and access the same data. This reduces redundancy, so that many settings are done only once.

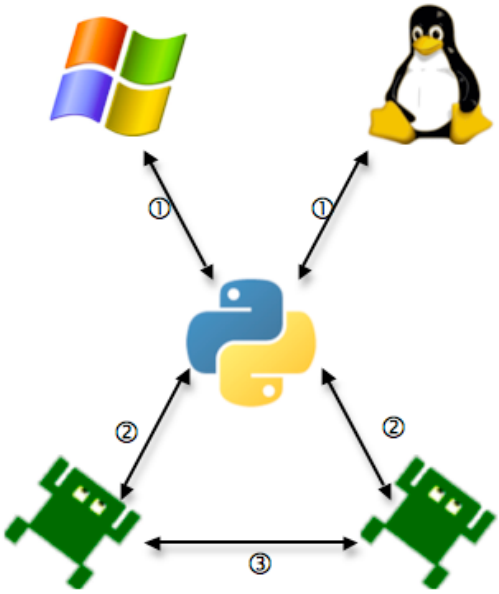

Figure 4 - New communication diagram

*Diagram explanation (*Figure 4*)*

① & ② Python is the interface between OS and Squish. It executes Squish-programs and also interacts with the OS during the tests (file system operations, etc.)

③ Squishrunner and Squishserver communicate during the execution of tests

Still need for OS dependencies

As there are differences in the different operating systems, it is not possible to eliminate all OS dependent code entirely. For example PVSS settings are stored in the registry under Windows, but on Linux-machines they are saved in a file.

Another example where forks are necessary is when PVSS has different behavior on different operating systems. This also requires the tests to be different on each platform. Unfortunately Squish does not provide functionality for running testcases only on specific platforms, so a developer has to check individually, if a test is being executed on a particular system.

4.2 Development of a library structure

A first look at the existing code showed, that it was difficult to read and that there were many repeated sections. In some cases there was ‘simple’ code-reuse (parsing a python-file at runtime), but there was no real code-reuse in form of global functions, libraries or similar.

This led to long test scripts as well as high overhead for maintenance and bug fixing. New developments took a long time, as there was no way of having the advantage of already existing code (except ‘simple’ copy&paste).

In particular the low level routines for UI-widget-handling were repeated very often. One example is the routine of setting the text of a textfield. This routine consists of the following steps:

- Access (find) the textfield
- Empty the textfield
- Type new value into the textfield

Of course there were even longer routines, which were repeated very often. One example for a very long routine which handles a UI-widget is the routine which opens a path in a tree (like a file system-tree). This routine consists of various checks and actions for verifying that the correct treenode gets selected.

The old approach specified the exact clicks which have to be performed every time and implemented the necessary checks each time when there as an interaction with a tree. This lead to a big overhead and also minimized the possibility to make rapid adjustments, when the “Application Under Test” (short: AUT) changed.

4.2.1 Designing a library structure

For solving these problems it was necessary to build libraries which can be used in the tests and can assist the developer in producing an implementation more rapidly.

The first step was to extract the ‚low-level’ routines. Widgets like buttons, textfields, comboboxes, menus or trees are now controlled by routines in this new library. This central collection of functionality works as an interface to the UI. Developers use this library to create their tests, without necessarily having to know how to access properties of the UI.

Besides this basic library there is a need for several other libraries, which allow the user to execute frequently used routines. These libraries use the before mentioned low-level-classes to access the UI.

The aim of the higher level libraries is to provide functionality for accessing several widgets which rarely changes their behavior. Examples of these libraries are the Pro-

jectAdministration-Lib (provides functionality for creating, modifying and deleting projects), the Filesystem-Lib (interface for accessing the file system in a high-level way) and the Configuration-Lib, whose purpose is to parse configuration files („.ini“-files).

4.2.2 Final draft of the Lib-structure

These thoughts lead to a library-structure which looks like this:

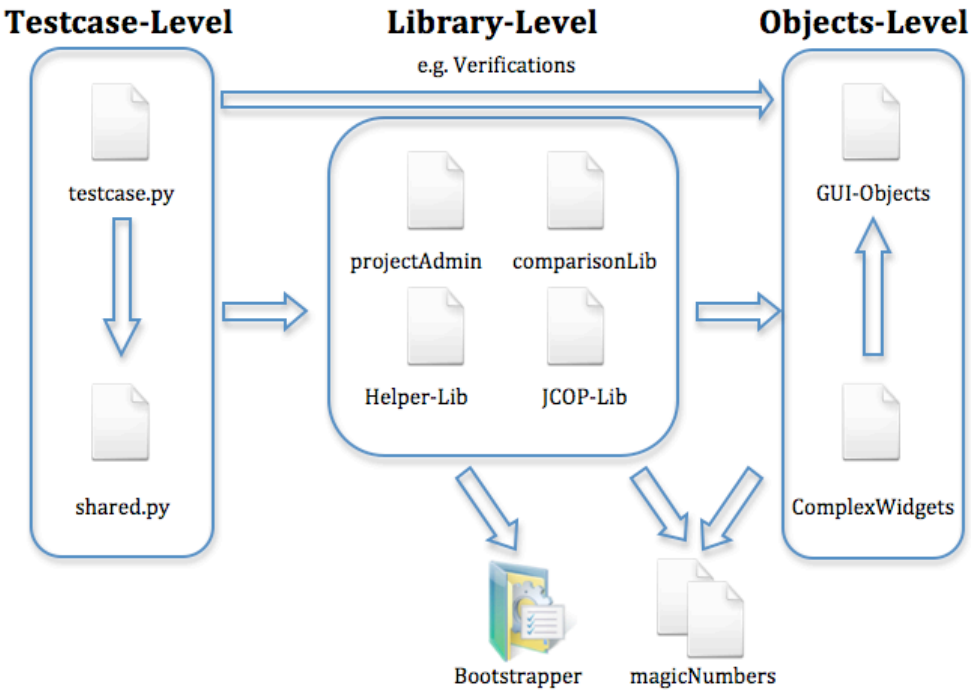

Figure 5 - Library structure (dependency diagram)

Objects-Level

As the diagram shows there are two low-level-libraries. The first one called ‚GUI-Objects’ contains classes which implement functionality for interaction with the UI. This is the most important library and is the only interface between Emu and the Qt user interface.

The other lib called ‚ComplexWidgets’ is also for interaction with widgets. The classes implement functionality for the more advanced widgets.

One example of these widgets is a context menu button, which opens a context menu after clicking on a button. This widget consists of two widgets of the guiObjects-lib.

Another example would be a file dialog. A file dialog consists of multiple buttons, textfields and comboboxes and always has the same properties.

Library-Level

On a higher level („Library-Level“) there are several libraries which use the first-level-libraries to implement more advanced routines. For example the ProjectAdministration-Lib provides functionality for creating a project. This routine clicks all the buttons and sets all the values necessary for project creation.

Another example is the comparisonLib, which is responsible for comparing values and, possibly backing up the files being compared. It is also responsible for creating an entry in the test-log.

The library level also offers a connection to external software-packages. An example would be the PVSSBootstrapper, a tool that can short-circuit the normal project-creation-procedure.

Testcase-Structure

The testcase-structure consists of a testcase-file and a shared-file. The testcase-file is a procedural script which executes routines which eventually determine the quality of the program under test. This can be either by verification of some conditions or simply by the fact that the program does not crash at execution-time

The shared-file is a collection of data used by the whole testsuite. Routines which occur only in the testsuite are also stored in this file together with variables used by the whole testsuite (e.g. project names, file system paths, etc.). This way of working is, in fact, supported directly by Squish.

A testcase can access the Objects-level directly. This happens when a testcase has to access some UI-widget properties for verification.

All three levels use a file called „magicNumbers“ which stores global variables and values whilst also parsing other configuration files.

4.2.3 Code example for the Lib-structure

For an easier understanding of the three library-levels, the reader should have a look at the code sample in Appendix A. Note that this code-sample is only an excerpt from the actual files.

4.3 Easy test-management

It is important that tests can be triggered without physical access to the test computers. This is necessary so that people who check in changes into the source code management system can also trigger the tests independently of where they are.

The test management system should provide the possibility to look at the latest testresults and offer the possibility to actively inform users after tests finish e.g. by email or news feed.

4.3.1 Decision: Make or buy?

At the beginning there was the decision to be made whether to use an already existing system or to develop a set of home-made tools to fulfill the requirements. After a short web research the decision was made to take an existing continuous integration system („CI“) as there is a large community offering valuable support.

A second reason for choosing a CI tool was that a CI tool could be used to control other parts of the program development process in the future.

There were three CI-systems of potential interest for our purposes. On the one hand there was „Bamboo“, which is a commercial tool created by Atlassian, and on the other hand there were Hudson and Jenkins, which are available for free and Open Source. Note that the Jenkins-project is a split-off from the Hudson-project so the codebase is nearly the same and to date there is no major difference between these tools.

4.3.2 CI-candidate: Atlassian Bamboo

As the EN-ICE-group at CERN already uses Atlassian’s Jira and Confluence tools there was a strong desire to also use Bamboo [xii] as it is from the same company. Integration with Atlassian’s other software should therefore be straightforward.

Other departments at CERN also use Bamboo and could provide support during the implementation.

4.3.2.1 Evaluation of Bamboo

During the evaluation-phase we discovered that Bamboo did not fully meet our expectations.

We found several problems. Although there is a large and active community the number of useful plugins available is, compared to other available CI-systems, relatively small. Also there are some problematic aspects where we would have had to do some „dirty“ workarounds.

For example, there is no possibility in Bamboo to run a job without reference to a source code management system. In our case this would mean that we either would

have to write a plugin or create an empty dummy repository just so that Bamboo could checkout something before starting the execution.

4.3.2.2 Thoughts about Bamboo

Bamboo is a good tool for „normal“ CI-projects, but its shortcomings meant that it would be inconvenient.

4.3.3 CI-candidate: Hudson/Jenkins

As the Hudson and Jenkins projects have the same codebase we had to decide on one of them and chose Hudson over Jenkins for no particular reason. Fortunately, we have had no reason to regret this decision.

4.3.3.1 Setup of the CI-system

The installation of Hudson was a more difficult than the setup for Bamboo. Hudson needs to be run inside of an Apache Tomcat Webserver, whereas Bamboo has an executable for installation. Hudson also needs to be run with a certain user, because otherwise some process interactions wouldn’t work. Normally Hudson offers installation as a Windows Service, but in our case this did not work either because the service forced the wrong user to run the processes.

After the installation it was straightforward to setup a job to call the existing Emu routines.

4.3.3.2 Evaluation of Hudson

The evaluation of Hudson was favorable. Not only is this software free and flexible, but also the community is very big and active, which means that there are many solutions already available on the internet for each problem we came across.

4.3.3.2.1 Use of plugins for Hudson

As Hudson has a very big community and is open source there are a lot of useful plugins available. For our purposes the most important plugins are the „Python“ plugin, which makes it possible to enter python code to be executed as well as the „Copy to Slave“ plugin which implements easy transfer of files to a slave-node before a job as well as copying results files back afterwards.

4.3.3.2.2 Usage of the Squish-plugin

Squish offers a plugin for executing tests from Hudson. Unfortunately with this plugin tests cannot be run on a target machine. There is the possibility to have a squishserver running on a different machine than the squishrunner, but due to some performance improvements this option is not possible for Emu. For example the file system routines

are called direct calls to the OS which would be executed on the machine which is running the squishrunner rather than the squishserver.

Instead of the squish-plugin we wrote some python-scripts which do the same things as the squish-plugin but can be run on a different machine. (More on this topic can be found in Chapter 4.4.2)

4.3.3.2.3 Other features of Hudson

Hudson offers everything that the Emu-Team asked for.

The main feature is that Hudson runs on Windows on an Apache Webserver offering a web-based user interface and Hudson can therefore easily be ported to any other machine without substantial changes.

Hudson can parse testresults (as long as they are in JUnit-Format) and permits the automatic generation of emails containing testresults and other details of any failed tests.

Hudson also offers several job trigger possibilities to run the tests after a certain time is reached or automatically after a checkin to the source code management system.

4.4 Remote testing (testing on different machines)

For several reasons CERN needs to execute tests on different machines. This chapter explains these reasons, as well as several problems and solutions which came up during the implementation process.

4.4.1 Why CERN aims to run tests on remote machines

One very important requirement is that the tests should be runnable on different platforms. That means that the tests should support the same platforms as the software is run on.

Normally one would think that this feature comes out of the box, because PVSS as well as Squish support both the Windows and Linux platforms - the employed operating systems at CERN. But in fact the transition from one operating system to the other is non-trivial in terms of effort.

Another motive for remote testing is that the number of tests is very likely to grow quickly and will make it impossible to execute all tests on a single machine. The reason for that is that UI-tests, when compared to unit tests, are very slow, as tests have to wait for the display of panels with their associated typing- and clicking-actions.

4.4.2 How Emu executes tests on a remote machine

For execution on a remote machine, Emu uses a builtin Hudson-feature. Hudson is able to execute a command directly on a different machine. This makes it possible to easily move tests from one machine to another. The only requirement on the target

machine is that the Java Runtime Environment has been installed to execute the „Hudson slave agent“.

Hudson can then execute tasks on the client or target machine.

> **Outline:** For users who only access the web user interface this approach is nearly invisible and they might think that the tests were executed on the Hudson master machine. Hudson even allows them to access files in the workspace which are stored on the slave-machine.

Using a standing connection the Hudson master machine calls a Python-script on the target machine which takes care of the rest of the execution. The target script first starts an instance of the squishserver. Next, it calls a squishrunner, tells it which testsuite to execute and where to store the output-files.

After the execution of the tests the script on the target machine triggers the sending of results via email. Hudson parses the JUnit-formatted output and could also send these details via email, but this feature is not used, because

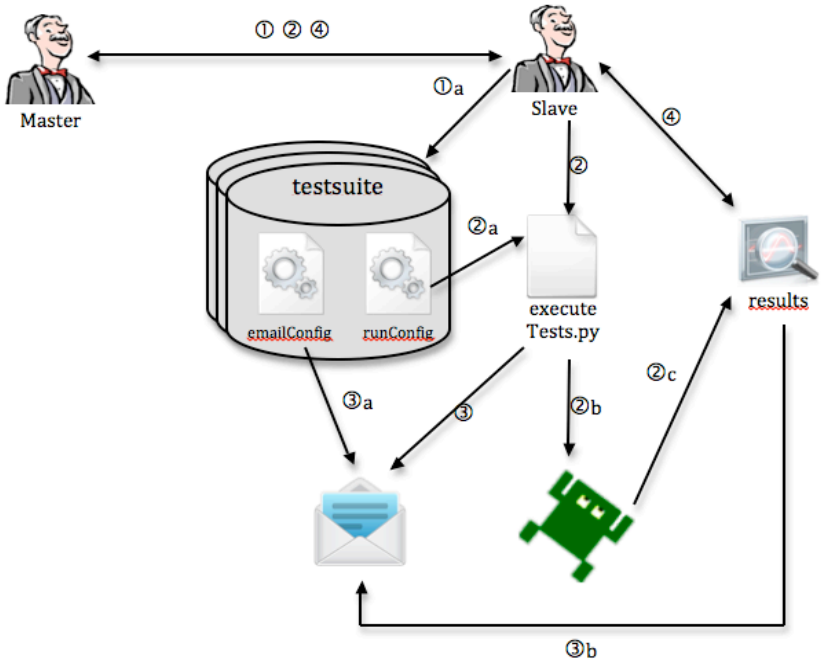

Figure 6 - How Hudson executes jobs

*Diagram explanation (*Figure 6 - How Hudson executes jobs*)*

① Hudson master tells the Hudson slave to get and deploy the EmuPackage
  ①a That causes the testsuites to be updated on the Hudson slave machine
② The Hudson master triggers the execution of applicable tests (The Hudson slave dispatches the call to a script)
  ②a Hudson server gets the information concerning which testsuites to run from a config-file in the testsuite itself
  ②b Run the test with Squish

②c Squish produces testresults
③ After the test execution Hudson server emails the results
③a Hudson server gets recipients from the testsuites
③b the content for the email
④ Hudson parses the results into a human readable form

4.4.3 Why the Squish plugin is not used

The reason for not using the Squish-plugin is, that the current version of the plugin does not fully support running tests on a remote machine. Although the plugin offers the possibility to execute the squishrunner on the Hudson-master node and the squishserver on the target machine, his approach is not applicable for Emu, as it uses several native Python-routines which would then be executed on the Hudson master instead of the target machine.

4.4.4 Problems when executing tests on different targets

There are several problems which may come up during the execution of a test on a new target machine. The following paragraphs explain the major problems encountered during the implementation of this milestone.

4.4.4.1 Different platforms - different behavior

Although on different platforms the general behavior is the same, there might be some differences in detail. This means that possibly there are routines which cannot be used the same way on two different platforms, because due to platform differences the program under test behaves differently.

One example is that PVSS saves data in the Windows Registry, while on Linux the information is stored in a file. So for getting this data the code has to have a fork for accessing the registry on one platform and parsing file operations on the other.

But there is not only the difference between two operating systems. There is also the different behavior of two versions of the same operating system. Stable tests running on WindowsXP can simply fail on Windows 7. This again led to a big effort to stabilize the tests and making them runnable on both platforms.

4.4.4.2 Different hardware - different behavior

As a last point there is also a difference in running tests on machines with the same operating system and version but different hardware specifications. This can lead to timing issues and race conditions, which can be very hard to track down.

The easiest solution for this problem is only to execute the tests on machines with a common specification. Another solution would be not to use hardcoded „snooze“-statements (these tell squish to pause/“sleep“ for a specified time). The problem with this latter approach is that any machine not matching the defined specification is very

likely to fail the tests. Although this solution is quite convenient at the beginning of development, it is better to understand these race conditions and developers should try to write their code in a way, which allows tests to run in the same way on any machine.

4.4.4.3 Maintenance of multiple target machines

The more machines there are, the more difficult the maintenance of these machines gets. Especially the part of keeping all the machines up to date with the current version of the test software, the libraries and the “Program Under Test” (“PUT”). (The reasons why testsuites and libraries are not stored in the same repository are explained in part 4.5)

4.4.4.3.1 Solution of the maintenance problem

To resolve this issue Emu creates a so-called „EmuPackage“ which contains the combination of testsuites, libraries, tools and software versions to be tested. The EmuPackage is created automatically everytime there is a new checkin of any component in the SVN repository.

Each target-machine downloads the EmuPackage and stores these updated files in the correct positions before it executes the tests.

4.4.4.3.2 Advantage of this approach

The fact that the EmuPackage is created and distributed from one central place makes this approach very maintainable and also allows easy testing of other versions.

The only thing the test manager has to do is to provide a tailored EmuPackage and trigger the tests on each platform.

4.5 Distributed development

The development of the JCOP-Framework will be changed, so that every person writing code for JCOP will also create a test which verifies the GUI functionality of the new code.

This requirement introduces a problem because Emu has to collect the testsuites before executing them. Also there is the problem of specifying which suite should run on which platform/target machine. Also there is the necessity for informing the component's developer of the test result.

In contrast to the testsuites, which should always be updated to their head revision from the repository, the libraries need to provide a stable interface for the developers. Therefore it is necessary to introduce a versioned „Emu-Core“, containing libraries, run-scripts and testsetups. The deployed libraries should be at a certain version. This version has to be announced to the developers, so that they can write their code according to this version. The versioning of the library-package is required, so that changes in the libraries don't affect the daily testruns.

There is also the problem that the current version of JCOP needs to be downloaded by every target machine on a daily basis. (This does not exactly belong to the topic of „distributed development“, but it is still discussed here, because the deployment happens together with the libraries and suites.)

4.5.1.1 General principle of the approach

For having an easy deployment process, there will be an Emu-tarball, containing the variable parts of Emu, which will be downloaded and installed on the target machine before every test-run. This so-called “EmuPackage” is created on the Hudson-Master on at least a daily basis to include the latest JCOP “current internal release” for the tests.

4.5.1.2 Solution to the update-problem

For meeting the specific update requirements the test suites are updated everytime there is a checkin to the SVN repository. The Emu-libraries are currently being updated on a manual basis, as there is no versioning done yet. After introducing internal releases of the libraries, these manual checkouts will probably be replaced by a script, which would make it possible to perform parameterized checkouts to update to a specific branch of the repository.

The current internal release is updated through a script, which downloads the new software from an AFS-networked file system.

4.5.1.3 Advantages of a single Emu-Package

The advantage of this approach is that the update on the target machines is easy and robust against changes. Instead of having to tell each machine from where to get the current internal release or which test scripts to update, each target machine simply downloads one compressed file containing all the needed information for executing the tests.

4.5.1.4 Determination of applicable testsuites

The EmuPackage is downloaded into every target machine with all available testsuites. To find out which testsuite should be run on which target, Emu uses a special file called the ‘targetPatforms’-file.

Each testsuite contains one such file. Containing the specification on which operating systems and with which PVSS versions the testsuite should run. If a machine does not fulfill the necessary requirements, it skips the execution of this testsuite and tries the other ones.

This system gives the possibility to easily coordinate and/or restrict the execution of tests on different OS and PVSS setups.

4.5.1.5 Determination of email-recipients

Using this distributed development scenario the number of testsuites is expected to grow quickly and we must consider ho the dispatching of test results is handled Not every developer is interested in all the test results. Usually each developer will only be interested in the results of the testsuites for which he is responsible.

To solve this problem each testmachine contains another file called ‚emailRecipients’. This file contains the email-address of each developer interested in the results of the testsuite.

A given developer may be interested in many different testsuites. To avoid him receiving an avalanche of emails, the email script merges all the results for one developer and sends only one email per machine.

This solution is a tradeoff between a manageable number of emails and the required level of detail from each result.

5 Guidelines for developing a good UI-testing-framework

This section summarizes the experience gained and gives the reader guidelines for creating and/or improving his own testing framework. It does not explain how to choose a good tool for testing a particular user interface.

Once a test tool has been chosen, it is important to choose the right scripting language for it. It is important to choose a scripting language which is supported on all platforms, as otherwise there would be the need for different technologies, lowering the maintainability of the harness and changes need to implemented more than once.

If the testing tool chosen supports more than one scripting language, choose the one with which you will also write the framework. This makes configuration easier, as both tests and framework can access the same config-files. Consider also that emulation is not real support of an operating system. In our example we saw, that Cygwin [xi] is a good program for running bash-scripts, but after getting to the limits of Cygwin we discovered that the difference between the bash emulation and the native implementation was too great to be sustainable. [4]

After creating the first tests, the framework developer should think of a way to reduce code duplication. For the reasons mentioned in the previous section, it is recommended to use object-oriented libraries. This makes new development fast and maintenance quick. [10]

The testscripts should only specify the order of library calls and verify the results. The library-functionality itself should not do any verification. Furthermore there should be a separate testcase for verifying the functionality. Although this approach produces some duplicate code, it does not fake the results by repeatedly calling the same failing functionality. For example, if a simple verification in an often used routine fails, it will produce a failure-message everytime the routine is called. The test system will report a large number of failed tests eventhough there is only one bug.

Another important topic concerns triggering tests on a remote machine and then communication of results to the ‚master' machine. It is also important to consider the issue of updating all the tests on these machines, because the number of machines will grow with the number of operating systems (and their versions) and program versions used. An appropriate solution for these issues is to use a continuous integration tool. Most such tools offer builtin functionality for executing operations on a remote machine.

[4] Cygwin is a software wrapper for porting programs of POSIX to Microsoft Windows. For more information see http://www.cygwin.com/

CI-tools also provide functions for the creation of the AUT and execution of other unit-tests, which may well need to be considered for future developments. [11]

It is important that tests are stable enough to cope with different machine specifications. Whether tests are run on different physical machines or virtual machines, it is very likely that there are going to be timing differences. Hardcoded waiting times should therefore be avoided and dynamic synchronization methods (waiting for certain UI-widgets) should always be used.

Another important question for the framework-developer is the handling of the test results. It is necessary to think of a useful representation of the results for the user. A carefully chosen CI tool can help satisfy this requirement. Modern CI-systems support the display of results as well as their propagation. If, as in our case, this functionality is not sufficient, it will be necessary to develop an extended results-handling module.

Consideration should be given to the development of new tests. Some frameworks save the tests in a single repository, which contains all the tests of all the developers. This approach may be convenient for updating, but it forces the test developer to store the test in a different place from his component. This arrangement is not intuitive and it is better to store the tests and the program together in one repository, making it easy adapt the tests whenever the program changes.

6 Conclusion

This thesis describes the creation of a sophisticated GUI testing framework and how it addresses main steps in testing: development, management and reporting.

We describe how a multi-level library-structure can reduce the effort required for additional development and maintenance. In this way, new tests can use the code from previous tests, or derive functionality from existing libraries. This derivation is achieved by using object-oriented code which is then imported by the testscripts. The testscripts themselves are procedural and call several library-functions to put the PUT into a state where conditions can be verified.

The management part of our GUI testing harness relies on the functionality of Hudson. The usage of this continuous integration tool gives the possibility to trigger the tests as well as view testresults. Hudson provides a web interface and is hence accessible from anywhere on the network, making physical machine access or a remote-desktop-connection unnecessary.

Hudson supports the remote execution on a slave machine. This feature is very important for user interface tests, because it gives the possibility to run tests on different target machines from a single point of control. The main problem with this approach concerns updating of the framework being tested. This thesis describes one solution to this problem using a Hudson plugin to allow the file transfer from the master to the slave-nodes. It describes the collection of the necessary information and the building of a single archive which is then accessed by each target machine to ensure that the framework as well as the PUT are both up-to-date.

On the target machine the same language technology as found in the libraries and testscripts. This approach gives libraries and testscripts the possibility to access the same configuration files as the tests thus avoiding unnecessary duplication of configuration files.

By using a platform independent technology, in our case Python, with minimum effort the same source code is runnable on all of the requested platforms. The only tradeoff that had to be made was at the occurrence of different behavior of the PUT on different operating systems. An example for that would be that on Linux PVSS project configurations are stored in a file, whereas on Windows these configurations are saved in the Windows registry.

One key aspect of this thesis concerns configuration of the testruns. It is shown how a single EmuPackage, which is deployed to every machine can be configured to run or not in a different environment. For this solution we introduced configuration files that are used to compare the machine setup with the information in the testsuite's configuration file so that execution of the testsuite happens in case of a match.

A similar approach was used for the sending of email-reports. The major problem discovered concerned the excess of information being sent to every developer. For this reason a second config-file was introduced, describing which developers are interested in the test. By using a locally developed tool for sending test result summaries, the amount of uninteresting information was reduced to a minimum. In this way a developer gets detailed information, but only for the tests for which he is responsible.

Glossary

AUT, PUT
: The “Application Under Test”, see also *2.1.9* AUT, PUT

CERN
: The European Center for Nuclear Research, see also *2.1.1* CERN

Emu-Library
: A file, which contains a collection of functionality, see also *2.1.10* Emu-Library

Emu-nightshift
: The daily testrun is called “Emu-nightshift” because the tests are triggered during the night.

Emu, Emu-Framework
: Emu is the test framework, which is developed to support developers writing their user interface tests with Squish.

GUI
: see “User interface”

Hudson
: Hudson is an open-source Continuous Integration tool.

Hudson-master
: The master is the machine where the main test triggering logic runs.

Hudson-slave
: Hudson-slaves are connected to the Hudson-master and give the possibility to execute tests on a machine other than the master.

JCOP
: The “Joint Controls Project”, tries to reduce end-user application development effort by creating a framework, see also *2.1.4* JCOP and the JCOP-Framework

LHC
: The “Large Hadron Collider” runs at CERN to research high energy particle physics. See also *2.1.2* Large Hadron Collider & Experiments

Library, Lib
: See “Emu-Library”

Master
: See “Hudson-master”

PVSS
: A SCADA-Application used by CERN to control IO-Devices in the LHC. See also *2.1.3* PVSS

Python
: A platform-independent programming language. See also *2.1.7* Python and *2.2.2.4* Python

Qt
: A framework used by PVSS to create a user interface.

SCADA
: SCADA stands for “Supervisory control and data acquisition” and is a term which is used for defining a certain type of software. PVSS is a SCADA-program and give the possibility to read and set data on multiple IO-devices. See also “PVSS”

Slave
: See “Hudson-slave”

Squish
: A program for testing of user interfaces. See also *2.1.6* Squish

SVN
: A source code management system. SubVersioN. See also *2.2.3.1* SVN

Target machine, target
: Different terms for “Hudson-slave”

Testcase
: A procedural script which combines library routine calls and verifications. See also *2.1.11* Testcase, Testsuite, Verification

Testsuite
: A testsuite is a collection of testcases which normally cover the same component *2.1.11* Testcase, Testsuite, Verification

UI-widget
: “UI-widget” is a term used for objects displayed on a user interface such as buttons, textfields, menus, panels, etc.

UNICOS
: UNICOS is a framework for the production of control applications. It is based on the JCOP-framework. See also *2.1.5* UNICOS-Framework

User interface, UI
: The UI is the part of an application, which displays panels, buttons, textfields, etc.

Widget
: See “UI-widget”

References

Academic references

[1] JCOP Framework Summary, LCG Savannah,
https://savannah.cern.ch/projects/jcopfw/ (last access on 02.05.2011 - 20:22)

[2] ATLAS Factsheet, ATLAS Experiment Group,
http://www.atlas.ch/pdf/atlas_factsheet_4.pdf (last access on 12.08.2011 - 14:00)

[4] Burkimsher, P. C. (2010). "Training Materials for PVSS & The JCOP Framework"
Controls Group, EN Department, CERN

[3] Salter, W. (2003). Industrielles Scada-Paket für Forschungslabor Cern. *ETZ - Eletrotechnik + Automation.* 22/2003, 2-7.
http://www.etm.at/Fachartikel/FA_etz_Cern_DE.pdf (last access on 12.08.2011 - 14:00)

[5] Gayet, P., & Barillère, R. (2005). UNICOS a framework to build industry like control systems: principles & methodology, *CERN*

[6] SLC - Scientific Linux CERN, CERN IT-Department,
http://linux.web.cern.ch/linux/scientific.shtml (last access on 22.08.2011 - 22:40)

[7] JCOP Framework Project, "JCOP Framework Project"
http://j2eeps.cern.ch/wikis/display/EN/JCOP+Framework (last access on 12.09.2011 - 18:15)

[8] Tyler Croy R. (2010). Who's driving this thing.
http://jenkins-ci.org/content/whos-driving-thing (last access on 11.09.2011 - 22:00)

[9] Kanglin Li, K., & Wu, M. (2005). Effective GUI Test Automation: Developing an automated GUI testing tool. *SYBEX.*

[10] Wegner, P. (1990). Concepts and Paradigms of Object-Oriented Programming. *OOPSMessenger, 1, 1 (August), 7-87.*

[11] Fowler, M., (2006). Continuous Integration.
http://martinfowler.com/articles/continuousIntegration.html#BenefitsOfContinuousIntegration (last access on 25.08.2011 - 21:00)

[12] Manuel Gonzalez-Berges, M., (2003). *Proceeding from CHEP'03:* The Joint COntrols Project Framework.

[13] Stapely, N., et. al. (2009). *Proceedings from ICALEPCS 2009:* An Integration Testing Facility for The CERN Accelerator Controls System.

[14] Memon, A. M., (2002). "GUI Testing: Pitfalls and Process", *IEEE Computer, 35(8): 90-91.*

http://www.cs.umd.edu/~atif/papers/MemonIEEEComputer2002.pdf (last access on 14. 09.2011 - 13:25)

[15] Memon, A. M., et. al., (2003). *Proceedings from ICSM'03:* DART: A Framework for Regression Testing 'Nightly/daily Builds' of GUI Applications *http://www.cs.umd.edu/~atif/papers/MemonAIPS2000.pdf (last access on 14.09.2011 - 14:00)*

[16] Memon, A. M., (2001). A Comprehensive Framework for Testing Graphical User Interfaces *http://www.cs.umd.edu/~atif/papers/MemonPHD2001.pdf (last access on 14.09.2011 - 14:00)*

[17] Memon, A.M., & Xie, Q. (2004). *Proceedings from ICSM'04:* Empirical Evaluation of the Fault-Detection Effectiveness of Smoke Regression Test Cases for GUI-Based Software, *pp. 8-17.*

[18] McConnell, S., (1996). Best Practices: Daily Build and Smoke Test, *IEEE Software, vol. 13, no. 4, pp. 143-144.*

[19] Memon, A.M., & Soffa, M. L., & Pollack, M. E. (2001). Coverage Criteria for GUI testing. *ACM SIGSOFT Software Engineering Notes, Volume 26 Issue 5, Sept. 2001*

[20] Memon A. M., & Xie, Q., (2004). *Proceedings of ICSM '04:* Empirical Evaluation of the Fault-Detection Effectiveness of Smoke Regression Test Cases for GUI-Based Software*, pp. 8-17.*

[21] Finsterwalder, M., (2001). *Proceedings of The 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering:* Automating Acceptance Tests for GUI Applications in an Extreme Programming environment. *pp. 20-23, May*.

[22] Lowell, C., & Stell-Smith, J., (2003). Successful Automation of GUI Driven Acceptance Testing. *Lecture Notes in Computer Science, Springer-Verlag Heidelberg, vol. 2675, pp. 331-333.*

[23] Sun, Y., & L. Jones Edward, L., (2004). *Proceedings of the 42nd annual Southeast regional conference:* Specification-Driven Automated Testing. *pp 140-145,*

[24] Zhu, H., (1997). Software Unit Test Coverage and Adequacy. *ACM Computing Surveys, vol. 29, No.4, pp. 366-427.*

[25] Gerrard, P., (1997). Testing GUI Applications. *EuroSTAR'97, pp 24-28.*

[26] Linz, T., & Daigl, M. (2007). How to Automate Testing of Graphical User Interface. *http://www.imbus.de*

[27] Linz, T., & Daigl, M. (2007). GUI Testing Made Painless [Online]. *http://www.imbus.de*

[28] Shehady, R., & Siewiorek, D., (1997). *Proceedings of the 27th Int. Symposium on Fault Tolerant Computing:* A Method to Automate User Interface Testing Using Variable Finite State Machines. *pp80-88*.

[29] Okika, J. C., et. al. (2006). *Proceedings of the 2006 international workshop on Automation of software test*: Developing a TTCN-3 Test Harness for Legacy Software," *pp.104-110*.

[30] Payne, J. E., & Alexander, R.T. , &Hutchinson, C.D., (1997) Design-for-Testability for Object-Oriented Software. *Object Magazine, 7(5): pp. 34-43*

References for software

[i] JCOP framework
http://j2eeps.cern.ch/wikis/display/EN/JCOP+Framework

[ii] PVSS
http://www.pvss.at/index_e.asp?id=&sb1=&sb2=&sb3=&sname=&sid=&seite_id=

[iii] UNICOS framework
http://j2eeps.cern.ch/wikis/display/EN/UNICOS

[iv] Squish
http://www.froglogic.com/products/index.php

[v] Python
http://www.python.org/

[vi] Microsoft Windows XP
http://windows.microsoft.com/en-US/windows/products/windows-xp

[vii] Microsoft Windows 7
http://windows.microsoft.com/en-US/windows7/products/home

[viii] Linux (SLC)
http://linux.web.cern.ch/linux/

[ix] Hudson
http://hudson-ci.org/

[x] Jenkins
http://jenkins-ci.org/

[xi] Cygwin
http://www.cygwin.com/

[xii] Atlassian Bamboo
http://www.atlassian.com/software/bamboo/

[xiii] Microsoft Windows Command Line
http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/cmd.mspx?mfr=true

Other online references

[A] CERN - „Conseil Européen pour la Recherche Nucléaire“
http://www.cern.ch

[B] LHC
http://public.web.cern.ch/public/en/lhc/lhc-en.html

[C] SCADA
http://en.wikipedia.org/wiki/SCADA

[D] ETM
http://www.etm.at

Other used Resources

Icons

http://www.python.org/community/logos/ (last access on 12.08.2011 - 13:15)

http://a1.twimg.com/profile_images/124147150/frog_bigger.png (last access on 12.08.2011 - 13:20)

http://www.pioneerx-estates.co.uk/UserFiles/Image/RCI/linux_logo.png (last access on 12.08.2011 - 13:25)

http://thegadgetsite.com/wp-content/uploads/2011/06/windows-logo.jpg (last access on 12.08.2011 - 13:25)

http://linuxhub.net/wp-content/uploads/2009/07/Terminal-Replacement_256x256.png (last access on 12.08.2011 - 13:30)

http://www.iconshock.com/img_jpg/XMac/security/jpg/128/command_line_interface_icon.jpg (last access on 12.08.2011 - 13:30)

http://3.bp.blogspot.com/_WtC5cg-N9j8/SNubLMRVlHI/AAAAAAAAAIM/GdchsElO_30/s200/cygwin-logo-large.png (last access on 20.08.2011 - 18:48)

http://icons.iconarchive.com/icons/gakuseisean/radium/256/settings-file-icon.png (last access on 21.08.2011 - 0:53)

http://www.iconfinder.com/icondetails/7321/128/development_package_icon (last access on 21.08.2011 - 1:00)

http://www.iconfinder.com/icondetails/3784/128/blank_document_file_page_paper_icon (last access on 21.08.2011 - 1:00)

http://farm4.static.flickr.com/3437/3384877145_c91aacec69.jpg am 24.08.2011 - 11:45)

http://www.iconfinder.com/icondetails/40772/128/email_envelope_mail_newsletter_open_icon (last access on 24.08.2011 - 11:55)

http://www.iconfinder.com/icondetails/46723/128/view_results_icon (last access on 24.08.2011 - 13:05)

Appendix A - Code example of Lib-structure

This code sample has been modified for easier understanding. The real code would be too long and complex to include here.

This code uses Python-syntax. The code explanation follows below.

```
##########################################
# (File: guiObjects.py - objects-level)
class QtButton:
    def __init__(self, objectname):
        self.name = objectname
    def click(self):
        self.waitForEnabled()
        clickButton(self.name) #squish functionality for clicking buttons
    def waitForEnabled(self):
        object = findObject(self.name)
        waitFor(object.enabled, 30) # squish functionality, which waits until
the expression is True

class QtCascadeMenu:
# This class is very complex and its not necessary to understand it for this
code sample.
# Therefore it should be seen as a blackbox.
    def __init__(self, contextMenu):
         # some code gets executed here
    def openPath(self, path):
        # this is the main-routine for working the way through submenus.

class QtTextfield:
    def __init__(self, objectname): #constructor
        self.name = objectname
    def type(self, text):
        type(self.name, text) # call the squish-function to type text

##########################################
# (File: complexWidgets.py - objects-level)
from guiObjects import *
class CascadeButton:
    def __init__(self, openButton, contextMenu):
        self.btnOpenButton = QtButton(openButton)
        self.contextMenu = QtCascadeMenu(contextMenu)
    def openPath(self, path):
        self.btnOpenButton.click()
        self.contextMenu.openPath(path)
##########################################
# (File: projectAdministration.py - library-level)
from guiObjects import *
from complexWidgets import *
class ProjectAdministration:
    btnCreateProject = QtButton("XXXXX") # normally there's an identifier
string instead of XXXXX
    cbtnProjectType = CascadeButton("XXXXX", "YYYYY") # normally there's an
identifier string instead of XXXXX and YYYYY
    txtProjectName = ("XXXXX") # normally there's an identifier string in-
stead of XXXXX

    def createProject(self, projectName, projectType):
        self.btnCreateProject.click()
        self.cbtnProjectType.openPath(projectType)
        self.txtProjectName.type(projectName)
```

```
##########################################
# (File: test.py - testcase-level)
import projectAdministration
def main():
    pa = projectAdministration.ProjectAdministration() # create an instance
of projectAdministration
    pa.createProject("myTestProject", "Standard") # call the function
```

Explanation of the Code:

- **guiObjects** contains 3 Classes (QtButton, QtTextField, QtCascadeMenu)
- All of these Classes have an __init__-function. This method is similar to constructors in other programming-languages
- **complexWidgets** contains the class „CascadeButton“. This class relies on **guiObjects**, because it uses the classes defined in there.
- **projectAdministration** instantiates several widgets; it also contains a function called createProject, which uses these widgets to execute several actions
- **test.py** is the testcase-file which contains a main()-function. In this function there is the instantiation of the ProjectAdministration-class and a parameterized call of the function „createProject()“