VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

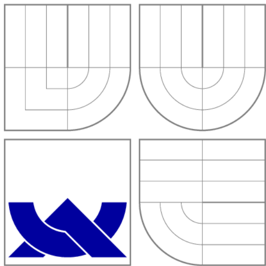
A CONTROL SYSTEM FOR APPLICATION TESTING IN LINUX

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

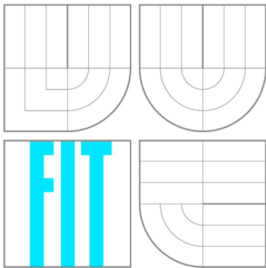
AUTOR PRÁCE
AUTHOR

Bc. EDUARD BENEŠ

BRNO 2009



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

ŘÍDÍCÍ SYSTÉM PRO TESTOVÁNÍ LINUXOVÝCH APLIKACÍ

A CONTROL SYSTEM FOR APPLICATION TESTING IN LINUX

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. EDUARD BENEŠ
AUTHOR

VEDOUCÍ PRÁCE — Ing. ALEŠ SMRČKA
SUPERVISOR

BRNO 2009

Abstrakt

Táto práca sa zaoberá problematikou riadiaceho systému pre testovanie linuxových aplikácií. Práca poukazuje na dôležitosť testovania software a jeho kvality pomocou automatizovaných softwarových nástrojov. Red Hat Test System (RHTS) je jedným z mnohých testovacích nástrojov. Predstavené sú rozdielne prístupy k ich klasifikácii a vyhodnocovaniu. Vybrané nástroje sú vyhodnotené a porovnané so systémom RHTS. V tejto práci je navrhnutý systém pre neinteraktívne testovanie linuxových aplikácií s podporou pre RHTS testy a s dôrazom na budúce rozšírenia. Implementovaný systém je následne otestovaný pomocou navrhnutých testov a popísaných je niekoľko príkladov použitia.

Klíčová slova

test, testování, kvalita, software, nástroj, vyhodnocení, porovnání, CAST, Linux, Python, RHTS

Abstract

This thesis discusses the area of a control system for application testing in Linux. There is a need for testing software and its quality using automated software tools. Huge number of testing tools is available, Red Hat Test System (RHTS) being one of them. Different approaches to classification and evaluation of a testing tools are presented. Selected software testing tools were evaluated and compared with RHTS. The thesis then presents a design of a system for non-interactive application testing in Linux with support for RHTS tests and with focus on future enhancements. Implemented system is finally tested using proposed set of tests and several usage examples are described.

Keywords

test, testing, quality, software, tool, evaluation, comparison, CAST, Linux, Python, RHTS

Citace

Eduard Beneš: A Control System for Application Testing in Linux, diplomová práce, Brno, FIT VUT v Brně, 2009

A Control System for Application Testing in Linux

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Aleše Smrčky. Další informace mi poskytl pan Ondřej Hudlický ze společnosti Red Hat. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

....
Eduard Beneš
May 19, 2009

Poděkování

Děkuji vedoucímu diplomové práce, panu Ing. Aleši Smrčkovi, za cenné rady poskytnuté při vytváření této práce. Rovněž bych chtěl poděkovat panu Ondřeji Hudlickému a všem, co mi poskytli odborné rady. V neposlední řadě chci poděkovat rodině a všem blízkým za podporu.

© Eduard Beneš, 2009.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Contents

Contents 1

1 Introduction 3

2 Software Testing 5
- 2.1 Introduction to Theory of Software Testing . . . 5
  - 2.1.1 Software Testing Techniques and Methods . . . 5
- 2.2 Automated Software Testing and Test Automation . . . 7
- 2.3 Evaluation and Classification of Software Testing Tools . . . 8
  - 2.3.1 Approaches to Evaluation and Classification Criteria . . . 8
  - 2.3.2 Comparison of Red Hat Test System and Selected Test Tools . . . 14

3 Red Hat Test System 16
- 3.1 Overview . . . 16
- 3.2 Architecture and Functionality . . . 16
- 3.3 Red Hat Test System Framework . . . 18
  - 3.3.1 Writing Tests . . . 18
  - 3.3.2 Work Description . . . 19
- 3.4 Related Projects . . . 19
  - 3.4.1 Beaker . . . 20
  - 3.4.2 Table Cloth . . . 20

4 Specification and Requirements 22
- 4.1 Requirements on the System . . . 22
- 4.2 Functionality Requirements . . . 23
  - 4.2.1 Server . . . 23
  - 4.2.2 Client Workers . . . 23
- 4.3 Classification and Requirements on Software Testing . . . 24
  - 4.3.1 Specification of Testing Techniques and Possible Restrictions . . . 24
- 4.4 Tests and Execution Environment . . . 25

5 Design of a Control System for Application Testing 27
- 5.1 Overview of the System . . . 27
- 5.2 Detailed System Design . . . 29
  - 5.2.1 Work Description and Control . . . 29
  - 5.2.2 Watchdog and Heartbeat . . . 31
  - 5.2.3 Managing Available Systems in a Test Laboratory . . . 31
- 5.3 Server . . . 32

5.3.1 Scheduler . . . 34
5.3.2 Database . . . 35
5.3.3 Test Repository . . . 35
5.4 Worker Systems Running Tests . . . 36
5.4.1 Test Environment . . . 37
5.5 User Client Tool . . . 38
5.5.1 Viewing Test Results . . . 39

6 Implementation **44**
6.1 Overview of Used Technologies . . . 44
6.2 Server . . . 46
6.3 Worker . . . 47
6.3.1 Test Environment Plugins . . . 47
6.3.2 Test Environment Isolation . . . 48
6.3.3 Running in a Daemon Mode . . . 48
6.3.4 RHTS Framework in Automated Mode . . . 48
6.4 Client . . . 49
6.5 Proposed Possible Future Enhancements . . . 49

7 Testing and Experiments **51**
7.1 Testing the System . . . 51
7.1.1 Testing the Functionality . . . 52
7.1.2 Load and Long Sequence Testing of the Server . . . 59
7.1.3 Testing Python Source Code . . . 60
7.2 Possible Usage Examples . . . 60
7.2.1 Getting the Community Involved . . . 61
7.2.2 Creating RHTS Tests for Wireshark . . . 61
7.2.3 Evaluating Student Projects . . . 61
7.2.4 Supporting Collaborative Testing in Open Source Project . . . 62

8 Conclusion **64**

Bibliography **66**

A Criteria for Software Testing Tool Evaluation **70**

B Quick Start Instructions **74**
B.1 Server . . . 74
B.2 Client . . . 75
B.3 Worker . . . 76

C Additional and Custom Evaluation Criteria **77**

Chapter 1

Introduction

Goal of this work is to design and implement a control system for application testing in Linux compatible with a Red Hat Test System (RHTS for short) test format and with possible future enhancements. RHTS has many hardware and software requirements in order to be successfully deployed in a target test environment. Therefore one of the goals is to design a system, that does not have such restricting requirements and allows easier installation and deployment for normal users. It is important to get the community involved in a testing process, these days. Allowing it to enhance, customize, and contribute to the project seems to be the most efficient way to accomplish this. Currently there is a huge number of software testing tools from different categories available. Unfortunately none of them supports execution of RHTS tests.

Before doing so, we have to get familiar with software testing which is discussed in the next chapter. The control system is closely related to the Red Hat Test System which has to be evaluated and compared with selected software testing tools. Many different tools are available. An evaluation and classification criteria are needed in order to compare and categorize them. It is not possible to use a single approach, because the tools can be so diverse. We conduct research into this area and selected several different approaches, each having its advantages depending on the application. Namely we present superficial classification proving that it is not sufficient and we can end up with feature rich tools in one category with those having just few key features placing it into that category. Most interesting approach is *Task Oriented View* presented in [23], which we apply on RHTS. Additional criteria should be defined and applied for proper tool evaluation or comparison. These custom criteria will help to select the final tool if we end up with several candidates. Another approach is based on *Testing Maturity Model* and categorizes tools based on supported testing maturity model. Also many commercial evaluations and comparisons are available, targeting large companies looking for the most appropriate solution. On the basis of gained knowledge about evaluation and selection we select software testing tools and subsequently compare them with RHTS.

Desired final system has to be compatible with RHTS in a some way. More specifically it has to be possible to use tests available for RHTS. In order to fulfill this key requirement we have to get familiar with RHTS. It is an internal tool used by Red Hat. We can not use any part that is not publicly available. Red Hat released some parts of the system in open source projects called Table Cloth and Beaker. We will focus on RHTS Framework that provides support for creation of RHTS tests and local execution. Red Hat Test System is

described in Chapter 3.

The next part the work focuses on designing the system. In Chapter 4 and Chapter 5 we present specification, requirements, and design of the lightweight system for non-interactive application testing in Linux.

Final part of the work continues with implementation and testing of the system designed during the term project. The system consists of three parts: server, worker daemon and client CLI application. Server is based on TurboGears Python framework using CherryPy as a web server providing WebUI, XML-RPC server, and yum repository providing test packages. Role of the worker daemon is to get work description and execute tests in specified test environment. A new custom environment can be created easily. Users can submit new test jobs and query for reports in XML format using a CLI client application. Creation of new commands for comfortable customization is also supported. At the end of Chapter 6 many possible future enhancements are proposed. Testing of the system and possible usage examples are described in the Chapter 7. Finally, Chapter 8 concludes the work and several appendixes cover topics like additional evaluation criteria, quick start instructions and breakdown of a software testing.

Chapter 2

Software Testing

Before we start to gather requirements on the system and design it, we will discuss an area of software testing. It is not a goal of this work to provide comprehensive description of software testing in a such short time. Rather than that, we present brief overview of selected topics with focus on giving references to the reader interested in deeper knowledge in software testing.

The first and the most important consideration in software testing is the definition of testing as it can affect the entire testing process. Testing is an activity performed for evaluating product quality, and for improving it, by identifying defects and problems [9].

2.1 Introduction to Theory of Software Testing

2.1.1 Software Testing Techniques and Methods

There are many different possible classifications of software testing techniques and methods. In this section, we will try to present classical approaches. The main problem with the classification is that some kind of tests do not fit exactly into one category. We suggest [31], [24], [18], and [13] for detailed definitions of software testing techniques and methods. List of recommended reference materials related to software testing can be also found in [9].

Classic framework as presented in [29] recognizes four main testing techniques:

Manual This is the oldest software testing technique. The basis is that manual techniques are carried out by people without the help of test automation. Despite the spread of automated testing, manual testing still dominates. Manual tests can be replaced by test automation, but automation can only be justified where repeatable consistent tests can be run over a stable environment [45]. Examples of manual testing techniques are walkthroughs, inspections, desk checking, etc.

Automated Automated software testing technique is implemented by the computer, and often automates established manual process. Using automated technique will increase the reliability of test results, and in a long run saves time and money. Examples of automated techniques are syntax checking, unit testing, integration testing, system testing, etc.

Static Static technique is time independent. Static refers to something that is not running, meaning that the actual software is not used. These techniques check mainly for the sanity of the code, algorithm, or document and can be manual or automated. Examples of static testing are syntax checking, code reviews, inspections, walkthroughs, code analysis, etc. Static testing is widely known as static code analysis

Dynamic Dynamic technique requires execution of the software. It is time depending and focuses on examination of the physical response from the system to variables that are not constant and change with time [46]. Examples are walkthroughs, unit testing, integration testing, system testing and so on.

In addition to the above classification it is important to introduce *test case design strategies*. Testing techniques for test case design can be separated into two categories based on perspective used to create test cases [29].

Black Box Testing Tested software is treated as a black box without any knowledge of internal implementation. Sometimes referred to as functional or behavior testing. The tester is trying to discover under which conditions does the software react differently as it should based on known specification. Test cases are designed only with the knowledge of the inputs and what the expected outcomes should be. Examples of black box testing are equivalence partitioning, boundary values analysis, decision tables, and state transition testing.

White Box Testing In contrast to black box testing test cases are based on the knowledge of internal structure or implementation of the software. The tester has an access to the code and internal logic. Its analysis drives the selection of test data and steps to reveal bugs in the internal logic. Examples of white box testing are decision coverage operation coverage, path coverage, statement testing, decision (branch) testing, and condition testing.

For a complete software examination, both white box and black box tests are required. Some sources (for example [46]) recognize also *grey box testing* [46] internet and web testing.

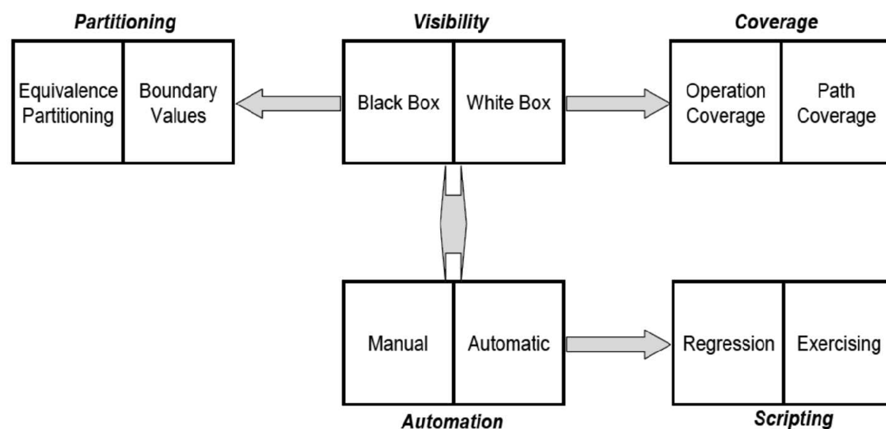


Figure 2.1: Brief overview of testing techniques as presented in [25]

Slightly different approach is presented in [24]. Author presents classification system for testing techniques based on five dimensions. Techniques based on 1) *who* does the testing

(*people-based*), 2) *what* gets tested (*coverage-based*), 3) *how* is it tested (*activity-based*), 4) *evaluation* of whether the test passed or failed (*evaluation-based*), and 5) the risks or *potential problems* you are testing for (*problems-based*).

Examples of testing techniques for each category are listed below. Note that the classification depends on how you look at the category. Some might be placed in different category depending on focus of the expert. Following list provides categorization of software testing techniques from [24]. The book provides also more or less detailed definitions of listed terms. Another useful categorization can be found in [18].

People-based User testing, alpha testing, beta testing, bug bashes, subject-matter expert testing, paired testing, and *eat your own dogfood* testing.

Coverage-based Function testing, feature or function integration testing, manual tour, domain testing, equivalence class analysis, boundary testing, best representative testing, input field test catalogs or matrices, map and test all the ways to edit a field, logic testing, state-based testing, path testing, statement and branch coverage, configuration coverage, specification-based testing, requirements-based testing, and combination testing.

Problems-based Input constrains, output constrains, computation constrains, and storage (or data) constraints.

Activity-based Regression testing, scripted testing, smoke testing, exploratory testing, guerrilla testing, scenario testing, installation testing, load testing, long sequence testing, and performance testing.

Evaluation-based Self-verifying data, comparison with saved results, comparison with a specification or other authoritative document, heuristic consistency, and oracle-based testing.

Software Engineering Body of Knowledge

As a part of theory of software testing one source worth looking at is *The Software Engineering Body of Knowledge* (SWEBOK). SWEBOK is a project of the IEEE Computer Society Professional Practices Committee that establishes a baseline for the body of knowledge for the field of software engineering. Its goal is to serve as a guide, rather than strictly focusing on defining the body of knowledge.

SWEBOK guide provides ten knowledge areas within the field of software engineering: Software Requirements, Software design, Software Construction, Software testing, Software maintenance, Software configuration management, Software Engineering management, Software Engineering Process, Software Engineering Tools And Methods, Software Processes and Product Quality. Figure A.1 shows breakdown of topics discussed in knowledge area dedicated to software testing.

2.2 Automated Software Testing and Test Automation

In this section we provide a brief overview of test tools with the focus on automated software testing. Test tools will be analyzed in more detail later in a chapter dedicated to evaluation

and classification of software testing tools.

What is automated testing? One of many possible explanations is: automated testing is a testing employing software tools, which execute tests without manual intervention.

Automated testing provides many advantages. The primary two are: increased effectiveness and increased efficiency [21]. Today it is recognized as a cost-efficient way to increase application reliability, while reducing the time and cost of software quality programs. Some of the common reasons for automating are reducing testing time, reducing testing costs, replicating testing across different platforms, repeatability and control, and application coverage and results reporting. [24]

Not all tests can and should be automated. Automating without good test design may result in a lot of activity, but little value. Designing tests without a good understanding of automation possibilities may overlook some of the most valuable opportunities for automation [24].

2.3 Evaluation and Classification of Software Testing Tools

2.3.1 Approaches to Evaluation and Classification Criteria

In this section, we present several different approaches to evaluation and classification of software testing tools. Authors of [23] claim that no single approach exists for classifying test tools. We have found it to be right in a sense that it is not possible to use a single approach for all the tools available. Therefore, we present different approaches, each of them having its advantages depending on its application. Consider a scenario where we want to select a tool to improve the testing process. Classification should divide the tools into different classes and thus help to focus on a specific category we need to evaluate. Evaluation of individual tools should provide enough information to select the final candidates.

Superficial Classification

Very superficial categorization presented in [23] divides tools into test planning, test design, test execution, defect tracking and configuration management. More detailed categorization is presented in [27]. At first author mentions that broad categorization divides automation tools into three classes: *unit* test tools, *capture-replay*, and *load* test tools. Then, tools are categorized according to software development phase while they are being used using more specific categories (highlighted using italic):

Designing a system This category contains *requirements capture and analysis tools* and *visual modeling tools*. Formal models created using modeling tools can be in many cases converted into code structures, tests, and data schemes.

Coding a system These tools are working with the code and its internal structure and are often called white box testing tools. Two main types are *static test tools* examining the code without actually running it, and *unit test automation tools* allowing to test isolated components of the system (for example using stubs and drivers).

Testing a system Since this group of tools does not depend on knowing internal structure of the tested system, they are often called black box tools. Typical representative are *capture-replay automation* tools used for GUI testing. *Load test automation tools* are usually used to perform many simultaneous actions while monitoring 'reactions' and responses of the system to this load. Overall properties of the system and environment is measured using *monitoring tools* which become really handy in conjunction with fault injection tools.

Tools used for managing testing Tools in this category are called *test management* tools. They allow various grouping, sorting, prioritizing, and assigning of tests. Supported is also management of test results, histories, monitoring and comparison of metrics. *Test generation* tools are being effectively used in connection with management and automation tools. Problems are managed using *defect tracker* tools (for example Bugzilla). *Data manipulation* tools work with bulk data sets and databases. *Environment management* tools are usually used to manage systems in a test laboratory. They provide comfortable installation and configuration of a test environment and system monitoring.

Evaluation Using Task Oriented View

In [23] authors applied Task and Object-oriented Requirements Engineering (TORE) methodology to identify activities that could be automated or supported by test tool. Using this approach, authors present criteria for classification and tool evaluation. They were systematically derived using the proposed methodology. Detailed evaluation requires installation and use of the test tools. In a real life scenario, evaluation criteria should help to select the final candidates (pre-selection). They would be installed and used for some time before being integrated into the testing process. Presented work focuses on coarse-grained criteria enabling an effective test tool pre-selection.

Criteria for test tools are divided into two classes: quality and functional. As a base for the quality criteria served standard ISO/IEC 9126 extended by criteria related to vendor qualification. Defined quality criteria for testing tools are listed in Appendix A.

Functional criteria were derived using TORE methodology designed to give guidance to the specification of user requirements on different abstraction level. Test process was analyzed from a task oriented view. Tasks and user roles were identified at task level and later redefined by activities at domain level.

Main tasks and corresponding roles involved in a test process as identified in [23]:

ID	Tasks	Role(s)
A	Test planning and monitoring	Test manager
B	Designing Test Cases	Test designer
C	Constructing Test Cases	Test automator, test designer
D	Executing test cases	Tester
E	Capturing and comparing test results	Tester
F	Reporting test results	Tester
G	Tracking Software problem reports/defects	Tester, test manager, developer
H	Managing the test ware	Test configuration manager, test administrator

Evaluation of Red Hat Test System

In this section, we present evaluation of Red Hat Test System (RHTS) using presented task oriented view. Each evaluated category is marked with yes or no value in parentheses, and followed by additional comments to provide more accuracy. We start with evaluation of quality criteria and follow with functional criteria. Evaluation is based on experience working with the system. Detailed list of criteria and numbering can be found in Appendix A.

Q1 Functionality (Yes) Internally developed tool, has positive evaluation in all marks. Security is preserved using user authentication (Kerberos). Interoperability with other tools is not supported by default, but would be possible if necessary. Many third party tools can be incorporated into actual tests.

Q2 Reliability (Yes) System (product) is mature, currently being reimplemented as a new open-source project called Beaker. System is able to detect certain types of failure, it evaluated to be fault tolerant and recoverable. Running test jobs are fault tolerant, systems under test can be recovered automatically.

Q3 Usability (Yes) Easy to learn and understand. Users with basic knowledge in writing bash scripts and/or other programming language should be able to develop new tests within a short time. Knowledge of any other programming language is a plus. System can be used using CLI or Web GUI. Key parts of the system and test skeleton are written in scripting languages.

Q4 Efficiency (Yes) Time and resource behavior is supported by RHTS Scheduler that is coordinating in cooperation with lab managers running test jobs, system installation, setup and more.

Q5 Maintainability (Yes) Maintainability is fully supported. System is implemented using scripting languages (Bash, Python, Perl, ...) and could be divided into several independent parts.

Q6 Portability (No) System uses tests stored in RPM packages and uses many other features specific to Red Hat distributions. Therefore it is not designed to support other distributions by default. We believe it would be possible to accomplish. The system is hard to replace as it provides many special features related to testing Red Hat Linux distributions. Other platforms are not supported.

Q7 General vendor qualifications (Yes) Despite the fact that the system is not available to the public, vendor evaluated to be mature with strong market share and financial stability.

Q8 Vendor support (Yes) This category evaluated positively. New releases are preserving compatibility with previous releases. Training materials, documentation and help from well trained users is available.

Q9 Licensing and pricing (Yes) Current version is for internal use only. New version (Beaker) with many new features should be available as a new open-source project. Many parts of the system are under GPL incense, while some are only for internal use.

Functional criteria were evaluated using three value scale. Possible values were: no support(negative), partial support(positive), full support(positive). Marks that evaluated to be non-negative are listed at the beginning of each category within parentheses. Full list of criteria and numbering can be found in Appendix A.

A: Test planning and monitoring. (1,2,3,4,6)
RHTS allows tests to be created using almost any programming language that can be wrapped/incorporated into RHTS test written in Bash. Supported are Linux operating systems (Red Hat distributions: RHEL, Fedora). Web GUI is supported in all major graphical web browsers (e.g.. Firefox, Opera). System is designed to allow testing 'almost any' application specific characteristics at lower levels for example using third party tools (GUI testing, performance, ...). It is used to test all parts of Red Hat Enterprise Linux distributions, and therefore criteria three and four are supported. There is also basic support for monitoring test activities, for example, by providing dashboard for each test job, duration time, current status and more. Integration with other tools is supported at level that the third party tools can be incorporated into individual tests. Integration with other tools is not required, but would be possible if necessary (Bugzilla, Testopia, ...).

B: Designing Test Cases. (7,8)
Test case designing is supported only partially by providing different workflows for test execution. Workflow describes how will be the selected tests executed. It supports running synchronized tests on different systems (for client—server testing), and creation of custom workflows. This approach allows to use desired testing technique to design the test cases.

C: Constructing Test Cases. (17,18)
Constructing test cases is supported in RHTS by RHTS Framework (in standalone environment). It is described in more detail in Chapter 3. Capturing of executable test cases, generation of (semi)formal models, invalid data generation, stubs, drivers, mock object creation or simulation of faulty system components is not supported.

D: Executing Test Cases. (24,25,26)
Execution of test cases is the most supported category. Setting up and cleaning down of the test environment is fully supported by the RHTS lab. It allows to select desired distribution, architecture, package versions, and many other system and distribution specific parameters. System configuration can be easily set up in the actual test case or as a separated test (and executed before running any other test cases). Tests can be executed on any available bare metal in the test laboratory. This allows testing on desired architecture and hardware configuration. Besides execution of automated test cases, manual testing is also possible. RHTS can setup (install) required testing environment for the tester automatically and inform him that the environment is ready to start manual testing.

E: Capturing and comparing test results. (29)
Test results, log files and other logging information are being stored and can be accessed using Web GUI. Test tool provides good support for logging information on executed test cases and they are available via Web GUI. Comparison facilities are not supported by default (should be possible to incorporate).

F: Reporting test results. (31,32)

Test tool provides full support for both criteria: aggregation of logged test results (and any test results) and customizable (role specific) amount of information. All results are accessible via Web GUI. Available are also e-mail notifications on different event types (e.g.. finished test job).

G: Tracking Software problem reports/defects. (38)

Only regression testing is partially supported. Regression tests can be used to specify Bugzilla bug numbers they are supposed to test. The problem can be described in the test case documentation, but primarily it is kept separately in Bugzilla. New problems are also filed into separate defect tracking tool.

H: Managing the test ware. (39,43)

Test ware management is supported by repository of all available tests. From here they can be installed on any system in the test lab upon request. This makes (re)use of automated tests for regression testing comfortable and efficient. Tracing modifications on tests and maintenance of a test data is done by external (standalone) revision control system.

Red Hat Test System is designed to provide flexibility and allow to create new tests using required testing technique without dictating the test design. This gives the advantage to use the most appropriate tool (or technique) at lower level (test). Rather than having all functionality in one test tool, separate test tools should be used to support other requirements. For example for defect tracking can be used successfully Bugzilla, and for test case management Testopia.

Classification of Testing Tools in Testing Maturity Model

Another possible approach to classification and categorization of testing tools is to use Testing Maturity Model (TMM for short) presented in book [13]. Analyzing this book helped us to classify test tools into categories by supported testing maturity level. We suggest [13] for better understanding of the testing maturity model.

Presented test tool evaluation criteria can help us to select the most appropriate tool from desired category. They are similar to quality criteria presented in [23], but being less formal and requiring more actual experience using evaluated tool. Evaluation criteria are:

- ease of use
- power
- robustness
- functionality
- ease of insertion
- quality of support
- cost
- organizational fit

Detailed explanation of each criterion can be found in [13]. In order to have the greatest benefit from using the selected tools during evaluation, we would suggest to define custom criteria for them. Example of such additional criteria is in Appendix C

Tool categorization in TMM is based on testing maturity level supported by the tool. This is new approach compared to other possible classifications focusing on functionality, testing phase, or testing activity.

List of test tool classification based on supported level in testing maturity model:

TMM Level 1 – debuggers, configuration builders, LOC counters,

TMM Level 2 – test/project planners, run-time error checkers, test preparation tools, coverage analyzers, cross-reference tools,

TMM Level 3 – configuration, management tools, requirements recorder, requirements verifiers, requirements tracer, capture-replay tools, comparator, defect tracker, complexity measure, load generators,

TMM Level 4 – code checkers, auditors, code comprehension tools, test harness generators, performance analyzers, network analyzers, simulators/emulators, web testing tools, test management tools,

TMM Level 5 – process asset library support tools, advanced test scripting tools, assertion checkers, advanced test data generators, advanced test management systems, usability measurement tools.

Other Evaluations of Software Testing Tools

Evaluating tools without experience of working with tools that are being evaluated is hard, inaccurate, and time consuming. Therefore nowadays exist several commercial evaluators of software testing tools that have experience with evaluated tools. They target on companies that are looking for a solution to their software testing but do not have their own resources or expertise to perform the evaluation and comparison of latest solutions.

Bloor Research Group published an evaluation of several commercial tools in [33]. Tools are evaluated in four categories: dynamic testing- client/server, dynamic-testing - character based, dynamic testing - GUI tools, static testing tools. Another example of commercial evaluation of leading testing tools is Ovum. They are evaluating software testing tools from market leaders like Compuware, IBM, Mercury Interactive, Rational Software, Sun Microsystems. We do not have access to any of commercial evaluation.

Major tool vendors represented by Compuware, Emprix/RSW, Mercury, Rational, and Segue (as it was back in year 2001) are evaluated in [17].

There are three automated testing tools selected and compared in [42]: Compuware's QARunn, Mercury Interactive's WinRunner and Rational's Team Test. This work presents real-life situation when company has to evaluate and select the most appropriate tool.

2.3.2 Comparison of Red Hat Test System and Selected Test Tools

In this section we present comparison of selected tools and RHTS. Many tools are available for software testers, both, commercial and open-source. We decided to focus on open-source solutions. Before doing so, we present brief overview of several commercial solutions from various vendors. Rational Testing Products (IBM) can provide complex solution using several products: Rational Test Manager, Rational Robot, Rational Performance Tester, Rational Quality Manager, Rational Functional or Manual Tester, and many plugins and extensions covering all aspects of testing. Another complex solutions are available from Borland (Gauntlet, SilkPerformer, SilkTest, SilkCentral Test Manager). HP offers solutions build using HP Quality Center, HP Functional Testing, HP QuickTest Professional, HP WinRunner and many more tools covering almost all parts of testing are available. Detailed evaluations of there tools are available from several commercial evaluators. These evaluations and comparisons are targeted on large customers looking for testing solutions.

In the rest of this section, informal comparison and discussion of selected tools and Red Hat Test System is provided. No particular evaluation criteria is used. We focus on following subset of supported features based on different methodologies presented above: license, maturity, primary programming language, project community, supported OS platforms and system architectures. Next we compare features related to handling tests and reports, different testing modes (manual, automated, interactive), support for tests written in different programming languages. Support of these features in RHTS is described in section 2.3.1, and by providing evaluation for selected test tools with additional comments comparison is achieved. We focus on open-source solutions. Valuable collection of different open-source testing tools, together with basic categorization and description, is available at `opensourcetesting.org`.

Salome Test Management Framework

Salome Test Management Framework (Salome-TMF for short) is intended to be an open-source solution alternative to QualityCenter [16]. It is feature-rich framework written purely in Java, which makes it multi-OS. Supports creation of tests, automatic or manual test execution, tracking results, managing requirements and defects, producing HTML documentation. Salome-TMF supports several types of test automation: Selenium can be also used for GUI Web testing, Beanshell for Java scripted tests, and other tools like Junit or Abbot. Defect tracking is accomplished by supported interoperability with Bugzilla or Mantis. What makes Salome-TMF unique, is its support for all testing activities (prepare, design, execute, analyze). Furthermore other existing tools can be easily integrated thanks to its plugin architecture, import/export of XML and to QualityCenter.

It is licensed under GNU GPL and primary programming language is Java. Project does not seem to be under active development at the time of writing (since 2007). Ability to run on several platforms is a plus. It is reported to work on Windows, Mac OS X, and Linux. Supports primarily tests written in Java language only. This seems to be really restricting and makes it usable only for projects in Java. Despite rich set of features, it cannot be used as a general tool for testing linux applications as RHTS does.

Software Testing Automation Framework

Software Test Automation Framework (STAF for short) has been developed by IBM [35]. Currently it is an open-source framework designed considering an idea of reusable components (called services in STAF terminology). Similarly as RHTS it tends to remove the necessity of building an automation infrastructure [6]. It provides a pluggable approach supported across a large variety of platforms and languages. STAF operates in a peer-to-peer environment and externalizes its capabilities through already mentioned services, that encapsulate certain set of functionality (logging, process invocation, and so on). Creation of custom services is supported.

One important part of STAF became later STAX, which is an execution engine based on XML language and implemented as STAF service (in Java Language). Its purpose is to allow easy automation of tests and test environments through workflow execution. According to documentation [6], STAX can be used to automate any task. RHTS also uses XML to describe test jobs and other activities, but when compared to STAX it provides less functionality and flexibility. STAX uses Python language to evaluate variables and expressions. The XML document is build of STAX Elements which can represent data, commands or processes, logic and job (test) control, exceptions, signals, functions and so on [6]. This makes it flexible and powerful. GUI monitoring application could be also considered as a plus in some situations.

STAF is licensed under Eclipse Public License (EPL). Some packages distributed with STAF are licensed under their own licenses (Apache Software License, zlib License, Jython License). Primary programming languages for STAF are C/C++ and Java (STAX). It has a good community support and active development. It is reported to work several OS platforms: various Linux distributions, FreeBSD, Mac OS X, and Windows versions, and many system architectures. STAF was designed to be easily usable from a variety of programming languages (e.g. Java, C/C++, Rexx, Perl, Shell, and so on) and extensible. It supports local and distributed streamed results, live test monitoring. Also all three testing modes are possible (manual, automated, interactive). Available services are: Cron, Email, Event, EventManager, FSExt, FTP, HTTP, NamedCounter, Namespace, SXE, Time and more (detailed description can be found in [6] and [22]). Similarly as RHTS, STAF does not provide support for defect tracking, GUI or performance testing and other testing techniques one needs to use tools like SilkTest, WinRunner and so on. Compared to RHTS, STAF has advantage in being multi-platform and providing many additional services (reusable components in STAF terminology).

Other open-source test tools are for example Accerciser [44], Dogtail [14], SAFS [2] for GUI testing Linux applications. As already mentioned for defect tracking we would prefer Bugzilla. For test case management is suitable Bugzilla plugin called Testopia. Sun develops interesting tool for managing performance testing called Faban. Creating a collection of test tools for testing Linux kernel and other features is a primary goal of Linux Test Project [1].

Chapter 3

Red Hat Test System

Red Hat Test System (RHTS) is an automated test system used by Red Hat's quality engineering department for qualifying releases of Red Hat Enterprise Linux (RHEL). This chapter describes its architecture and individual parts that had to be understood in order to successfully design control system for testing linux applications compatible with RHTS tests. RHTS documentation is not available to the public. At first, we present overview of RHTS and its architecture. Later we focus on RHTS tests and jobs through related projects that provide publicly available information about RHTS – Beaker [38] and Table Cloth [3]. Beaker is intended to be an open-source version of RHTS being developed from the ground up.

3.1 Overview

RHTS provides a standardized way to write and run automated tests on packages contained in a Red Hat distribution. Besides that it allows to use almost any testing technique by allowing it to use RHTS only for the control part, environment installation and setup, and following execution of specified test. RHTS focuses on a system level commands and operations [3]. It is written as a mixture of several scripting languages, mostly Bash, Python, Perl.

As we already mentioned it is designed for Red Hat distributions making it highly efficient, but hard to replace or port to different platform. When it comes to testing Red Hat distributions, RHTS provides support for developing automated unit tests, bug reproducers, hardware enablement, regression testing and other types [3]. Users are released from the necessity to install and setup desired system distribution and architecture manually. RHTS does all of this automatically based on user provided parameters.

3.2 Architecture and Functionality

Architecture of Red Hat Test System consists of several components. Namely it is:

- Scheduler
- Repository

- Database
- Test Laboratory Controller
- Systems in a Test Laboratory
- RHTS Framework
- RHTS Tests

The main important parts are the *scheduler* and *individual tests*. RHTS scheduler is responsible for managing activities relates to test execution based on user requirements by determining what, where and how should be launched. Another server part is a test repository holding individual tests and providing them to the systems under test. Database stores test jobs, results and log files. Main purpose of a test laboratory controller is to provide system provisioning. It manages all hardware in a laboratory and provides watchdog functionality for test jobs running on a systems in the test laboratory. Another part, and the only one that is currently publicly available, is RHTS Framework [3], which will be described later. The last part are individual RHTS tests.

There are certain specific requirements on this system, that make it different from commonly known automation tools. Its purpose is to support testing Red Hat Linux distributions. System has to be general enough to allow executing tests designed to use almost any testing technique. Therefore specialized testing techniques as GUI testing, performance and load testing are done at lower levels are using other test tools like Dogtail.

System functionality is in detail presented in section 2.3 using task oriented approach. Here we describe only functionality available from user perspective. Command line interface allows submitting test and test jobs, creating and managing different workflows in a XML files. This functionality is provided by RHTS framework described in next section. Scheduling jobs is also available using Web GUI. Viewing job status, test results, log files, searching available tests, systems in a particular laboratory, old jobs, viewing reports in different formats, monitoring jobs based on different criteria, and other similar functionality are available using RHTS Web GUI. Functionality for installation and setting up of system under test is provided by test laboratory controller. It is a system capable of managing system provisioning according to test job requirements (e.g., distribution, architecture).

Functionality of target hosts executing tests is mainly written as a set of cooperating Bash and Python scripts. This provides an ability to have strong control (over the system) and easy access to shell commands. Close coupling to functionality specific to Red Hat distributions makes it efficient, but unfortunately not portable to different platforms.

For system installation and setup in test laboratory environment, systems are installed and setup up using kickstarts (network installation tool set used by Red Hat distributions). Kickstarts are generated based on XML document generated based on user requirements and describing the test job.

3.3 Red Hat Test System Framework

Red Hat Test System framework (RHTS framework for short) was released as a part of Table Cloth project [3]. Tests developed using RHTS framework can run on developers workstation in a standalone mode, or in a laboratory environment. It defines test API and format of metadata files. By providing this users have everything that is required to create, test, pack and submit new tests. It should be noted that developing and executing more complex or distributed tests might get complicated outside RHTS laboratory environment.

The RHTS framework provides everything necessary to start developing new tests: tools, API libraries, and template files. Propoer test development environemnt can be set up using following RPMs: `rhts-devel`, `rhts-devel-test-env`, `rhts-devel-python`. All packages are available for download from yum repository at following URL:
`http://people.redhat.com/dmalcolm/tablecloth/tools/`.

Package `rhts-devel-test-env` provides components of the test system used when running tests in a standalone mode or within a RHTS laboratory. `rhts-devel` package is for creating and maintaining RHTS tests. It directly provides or pulls in through package dependencies runtime components of the test system for installation on a workstation. Python modules for handling RHTS test metadata are provided by package `rhts-devel-python`.

3.3.1 Writing Tests

RHTS test is a program performing sequence of tasks. Success or failure is determined by the test logic and results are reported using hooks in an API. The responsibility for reporting results is on the test itself. Test can consist of code, data, metadata, and dependencies to other program or test packages.

RHTS framework allows tests to run in two modes: *developer* and *automated*. Developer mode means that the test is running in standalone environment, usually on developers workstation while he is creating the RHTS test. When it is later executed in automated mode, it is able to detect laboratory environment and required variables that are available only in a laboratory environment. We found out this to be especially useful during initial development of new tests locally. It should be noted that RHTS tests by default (and design) run under superuser privileges.

RHTS test has three main components. Two source files (`runtest.sh, Makefile`) and a documentation file (`PURPOSE`). What makes RHTS tests flexible is the ability to add and use other scripts and source files to the test. This can be used to create complex test logic, incorporate third party tools, generate test data specific for different architectures, etc.

runtest.sh A shell script responsible for performing the test case and reporting the results. It is quite common to use it for delegating work to other, more complex, scripts or executables in other languages. The most simple way to create a RHTS test is to use already existing single test executable, in the example below called `my-test`, and execute it from the `runtest.sh`:

```
#!/bin/bash
rhts-run-simple-test $TEST ./my-test
```

Results will be reported together with log file containing captured `stdout` and `stderr` to the server based on the exit code from the executable. Variable `TEST` in the script above reveals that the test environment has to provide certain variables expected by the test script and its helpers. Custom test logic can be created in `runtest.sh` by sourcing `/usr/bin/rhts-environment.sh` file provided by the framework.

Makefile Purpose of this file is to coordinate developing and running RHTS tests. It handles compilation of executables, creating RPM containing all test files, installing test files to expected location, and most of all running the test. It is also responsible for defining mentioned environment variables necessary for running tests and reporting results.

PURPOSE This is a plain text file, intended to be read by human only, providing information about the test such as its description, known issues, and other useful information to help anyone not familiar with the test to understand and run it.

To start creating a new minimal test skeleton one can use RHTS framework tool `rhts-create-new-test`. Running test in developer mode is done by running a `make` tool in the directory containing tests `Makefile`:

```
$ make run
```

Running it in automated mode is more complicated and will not be discussed here. This will be described in the Chapter 5 dedicated to system design as it will be necessary for running RHTS tests in our system. More detailed documentation on RHTS tests can be found in [5]. It describes test components and how to create them, running tests, reporting results, packaging test and tools available in RHTS framework.

3.3.2 Work Description

RHTS jobs are described using XML files. Usually they are created using command line tools designed to generate XML for selected workflow (e.g. for running multihost tests). Figure 3.1 shows simple representation of RHTS test job. Specific example of XML files representing RHTS job and recipe are `recipe.xml` and `nfsvirtual.xml` that can be found in [38].

3.4 Related Projects

There are two open-source projects directly related to RHTS. Around year 2006 Red Hat came with initiative to involve open source community (not only around Fedora Project) in testing with the goal to improve the quality and speed of open source software development [3]. In this section, we give a short overview of the two projects, Beaker and Table Cloth.

3.4.1 Beaker

According to project pages [38] Beaker is an adaptation of Red Hat's RHTS automated test system for Fedora (and the Free Software community in general). At the time of writing this is a new project. It will be entirely automated testing framework. Actual status is available at dedicated Trac page [37]. The project is under slow development. At the beginning these parts of RHTS were made available[38]:

- Documentation and API for tests, test running, test result reporting, etc.
- Automated tests which use those API
- A personal test execution engine (allowing individual testers to run tests on their systems)

Beaker consists of several components. It uses several other Fedora projects to provide required functionality. Here is a list with short description based on information available at [37]:

Logan System scheduler accepting jobs upon request, provides repository of tests and database.

Medusa Laboratory Controller, designed to be build using other Fedora projects to provide specific tasks. They can be used separately. Namely it is:

- *Cobbler* - system provisioning
- *Conserver* - console logging
- *Fenced* - power-cycling machines (PXE installs and recovery)
- *Smolt* - inventory data

It should be noted that the only parts being developed as separate fFedora project are working at the time of writing, but separately. The system as a whole is still under gathering all requirements and development. Server (Logan) is being developed using TurboGears Python web framework. One of the goal is to take more modular approach which is being accomplished for example by using cobbler for provisioning and Smolt for inventory.

3.4.2 Table Cloth

A new testing project that turned into 'Table Cloth Project' was announced at Red Hat Summit in 2006. As part of this project Red Hat contributed several internally developed tests, RHTS testing API, test reviewing policies, test writing guide, and automated GUI test tool called Dogtail [3]. Unfortunately, this project wasn't accepted well by the community is not active anymore.

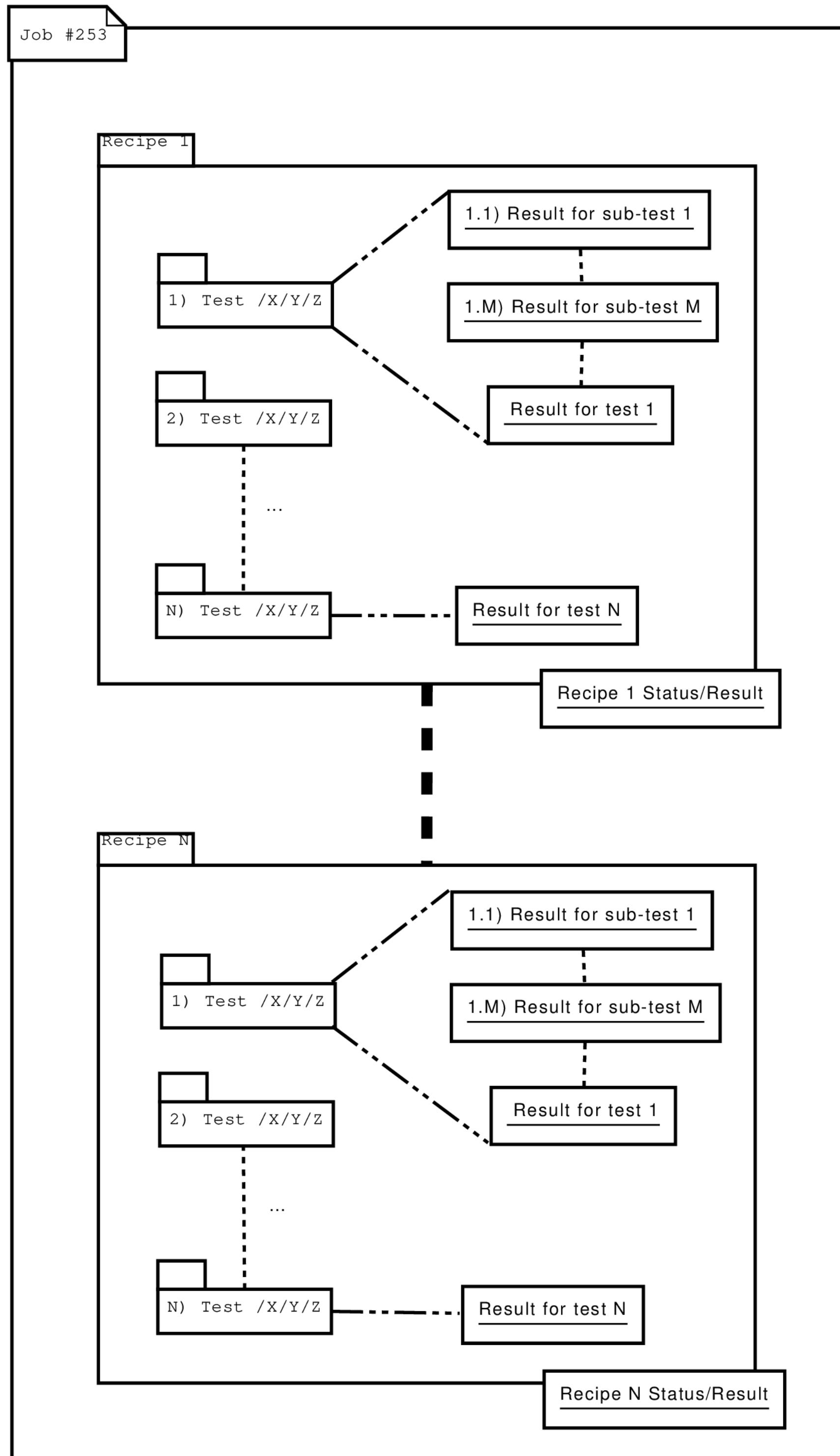


Figure 3.1: An example of possible visual representation of a test job in Red Hat Test System. The test job is identified by ID number. It contains several recipes generated for each system matching requested configuration. Each recipe can consist of one or more tests. Each test can hold results for several sub-tests and together with its own result or current status of execution.

Chapter 4

Specification and Requirements

In this chapter, we discuss specification and requirements on the control system for application testing in Linux. We present requirements on the system, functionality requirements on the system components, system classification, test requirements and ideas on possible future enhancements.

4.1 Requirements on the System

Automated test cases are currently managed and scheduled in Red Hat internal system called Red Hat Test System (RHTS). This system is in more detail described in Chapter 3. It is large and complex system, therefore implemented system should be lightweight and allow straight-forward deployment. This will allow it to be used by arbitrary SW projects on OS Linux. Building shared database of tests with the support from community should have essential effect on software quality. Follows a list of the main requirements on the system.

- Tests compatible with RHTS format, which is in fact a RPM package, and it is referred to as RHTS test in further text.
- For test creation can be used RHTS framework described in Chapter 3.
- It should be possible to execute RHTS tests on local and remote systems. System should allow to manage and schedule automated test cases.
- Each executed test produces and reports some results.
- Test results should be stored and provided upon request.

- Architecture of the system should be based on client-server model. Server is responsible for test job management, test management, client management and preventing possible invalid runs.
- Client (also called worker to distinguish from client side provided to the user) is primarily responsible for provisioning test execution environment, test execution and reporting results.

4.2 Functionality Requirements

4.2.1 Server

This is the part of system users will be interacting with. The primary component, it is composed of, is a test scheduler. Any request from user will process through the server until being served by a client (worker). Main functionality requirements on server part are:

- schedule test job (CLI)
- store tests
- manage test job results
- receive test job results from client (worker)
- dispatch test job to client (worker)
- accept request to run test job
- provide test job results (CLI and WebUI)

Users of the system are creating tests. After placing the tests into test repository on the server, system should allow to request execution of those tests on client system(s). Clients can have different characteristics like architecture, installed OS, etc. Therefore server should be able to in a simple way watch available clients and report if specific request is not possible to satisfy.

Server should be able to pass execution of a test job to client matching characteristics requested by user of the system. Scheduler will be responsible for this. After a client executes the test job, it reports results and log files to the server.

Test results should be stored in local database. It should be possible to access test results using command line interface (CLI) or WebUI. Submitting tests to the server is not defined and is up to the deployment of the system how to submit new tests to the server. Server is only responsible for providing dedicated storage for the test repository. Ideally this could be provided by helper scripts available to the user.

4.2.2 Client Workers

Main responsibility of a client workers is to execute test job and report results to the server. Client systems are installed and configured manually. Running dedicated daemon will show client and its configuration characteristics to the server. Succeeding this will make it available to run test jobs matching configuration requirements. Functionality requirements on client system are:

- test environment provisioning
- test installation/setup
- execute test job

- provide test watchdog
- report test job results
- accept/request new test jobs
- report presence of a client to the server
- environment configuration

Client works on job assigned to it by a server. To achieve this, server needs to know the state and configuration of its clients. This information are reported to the server by each client. Client is responsible for provisioning environment suitable for executing the test job. Provisioning options should be discussed in more detail in Chapter 5 dedicated to system design. Required parts of the test should be installed and also executed by the client.

In RHTS test, results are reported to the server by the test itself and the client is only responsible for providing required resources and environment (packages, libraries etc.). Each test job has defined maximal execution time, if it exceeds it, it should be terminated by watchdog.

Test environment provisioning, test installation/setup, watchdog, accepting new test jobs from server is in RHTS done by dedicated system called laboratory controller responsible for managing all hardware in a test laboratory. The process is in RHTS driven by a generated `kickstart`, and shell script using several helper tools.

Following UML Use-Case diagram 4.1 depicts overview of the functionality that should be provided by the system for three roles: user, server and worker (client).

4.3 Classification and Requirements on Software Testing

Each software testing tool can be characterized using different criteria as we present in section 2.3. Main responsibility of this tool is to server as (lightweight) control system for automated testing. It will provide less functionality but ideally will be more flexible and extensible than RHTS.

The system will provide support for constructing test cases, using the RHTS framework which is described in [5], executing tests, aggregating results and log files. By default there will be no support for defect tracking, capturing, formalized test case design, comparing test results, test revision control, statistics and other advanced features.

4.3.1 Specification of Testing Techniques and Possible Restrictions

Examples of various test types and techniques that should be supported directly by the system or using third party test tools: regression testing, smoke testing, installation testing, configuration coverage, unit tests, endurance testing, manual testing on reserved system (possible future enhancement), long sequenc testing, load testing, performance testing, and any other test type that can be turned into RHTS test

All these testing techniques are possible to use in RHTS. Definitions of listed testing techniques can be found in [18] and [24].

On the other side, there are test types and techniques that are not supported or not possible to implement and execute. Some tests are considered to be destructive and unless executed in a properly isolated environment, they might break the worker system. It would have to be fixed manually, before being able to run another job. For example testing network, rebooting system, or testing other parts that are used by the worker to communicate with server or execute tests has to considered vulnerable to be broken by the test. A use of appropriate test environment should prevent such a vulnerability.

4.4 Tests and Execution Environment

One of the most important requirements is to preserve compatibility with RHTS tests. As presented in [38], Red Hat QE has released automated tests, API they are using and its documentation. Therefore RHTS test format will be used by default. Detailed information about RHTS tests can be found in [3].

It should be possible to use RHTS Framework to develop new RHTS tests. RHTS Framework is already available as a part of Beaker and Table Cloth project. This would also allow to execute (and debug) tests locally in developers mode or automatically in test environment.

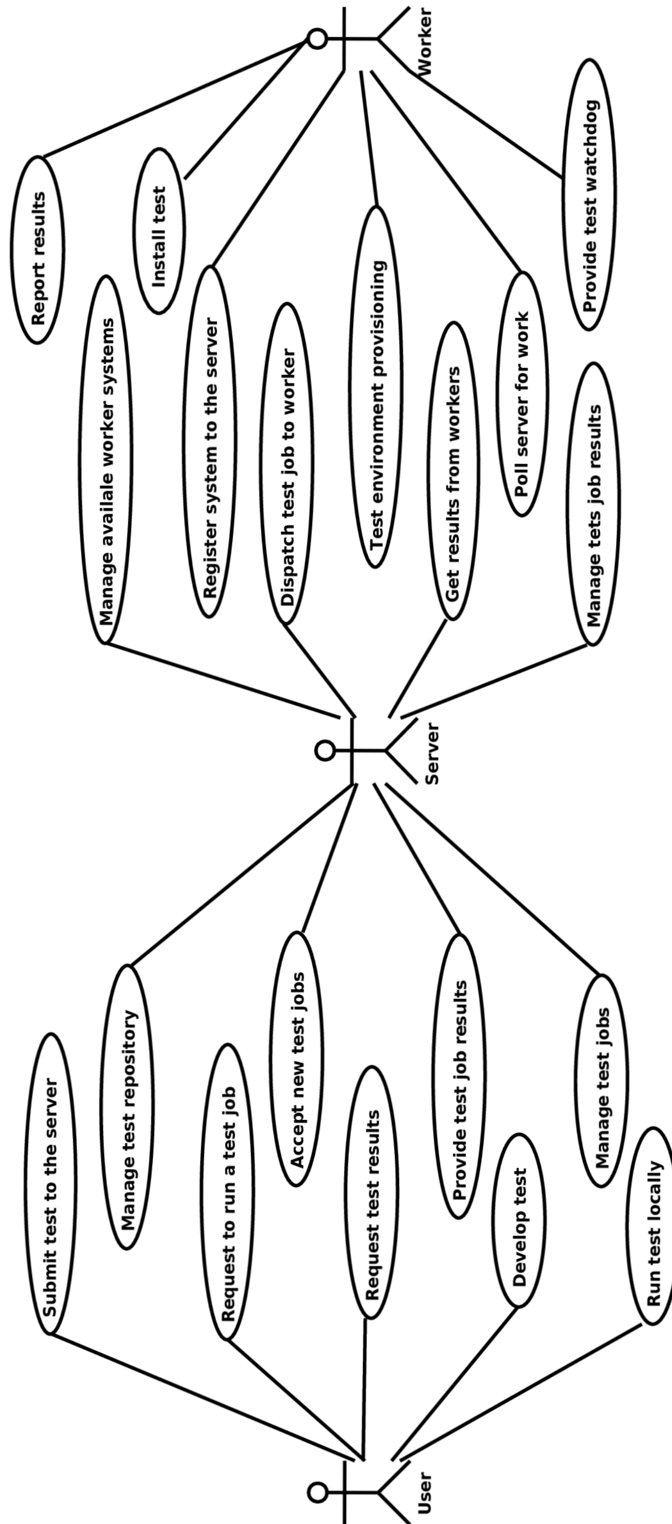


Figure 4.1: Diagram depicting functional decomposition of specified requirements on the system.

Chapter 5

Design of a Control System for Application Testing

In this chapter, we present proposed design of a control system for testing applications in linux. RHTS has many hardware requirements on target hosts, laboratory controller, and server. Therefore one of our goals was to create a system, that would not have such restricting hardware requirements and would allow much more straight forward installation and deployment for normal users. Another goal was to make it flexible and extensible by new features in the future.

5.1 Overview of the System

The system consists of a three main parts: user client (helper) tool, server, and worker. Using command line interface (CLI) client tool, users of the system can scheduler new test jobs (create and submit job XML), access test repository and request for reports. Server accommodates much more functionality: scheduler (stateless), database (for results, jobs, test laboratory, log files), test repository (RPM), job parser and validator, recipe generator, worker system manager (laboratory inventory). Workers are client systems registered at server and polling it for work (test execution). Worker components are daemon, recipe controller and parser, plugins API, plugins, watchdog, and test runner. Here is a list of key points providing overview of the system:

- user side CLI commands to communicate with server implemented in Python
- work (test job) and parameters described by a XML document, derived from Beaker (RHTS), and called test jobs and recipes
- server operates on a test job upon user request
- workers poll server for work, server is stateless
- server dispatches work described by recipe, which is a XML document created by server from workflow
- server build using Python web framework supports adding new features (e.g. WebUI enhancements)

- server designed using Model-View-Controller design pattern and build using Python web framework
- worker receives recipe and parses it to execute requested work
- worker plugins allow adding support for new test execution environments, test formats and test environment isolation
- uses RHTS framework for executing and developing new (RHTS) tests
- test results are accessible via CLI or WebUI
- ideally the implemented system should be extensible, pluginable, configurable
- no user input is currently planned for the WebUI

The system components should be build on top of existing technologies: XML, XML-RPC, Python, RHTS Framework, Python web framework (Django or TurboGears).

Figure 5.1 depicts simplified architecture and chronological overview of the control system for linux application testing. Following ordered list describes possible chronological order of step depicted in the figure:

1. Server provides *Worker Manager* responsible for keeping track of available worker systems that will serve the systems under test. After configuration and start of dedicated daemon/service, worker will try to register/report to configured server. Worker Manager records provided configuration of the worker.
2. User of the system submits test to the test repository.
3. Using command line interface, user can request execution of selected tests on systems matching given criteria. Scheduler evaluates the request, aborts all unrealizable parts and adds new jobs to the queue.
4. Worker systems poll the scheduler for work.
5. Scheduler checks the queue and assigns work to those, matching requested configuration (arch, family, ...) through test job dispatcher.
6. Providing test data to clients. Also provide access to some shared storage.
7. Worker system reports results and log files to the database. Scheduler will need access to the database in order to file results for aborted jobs.
8. User can review current results using CLI. (This would be most likely some simple or batched form).
9. Server provides WWW server for reviewing the results using web browser.

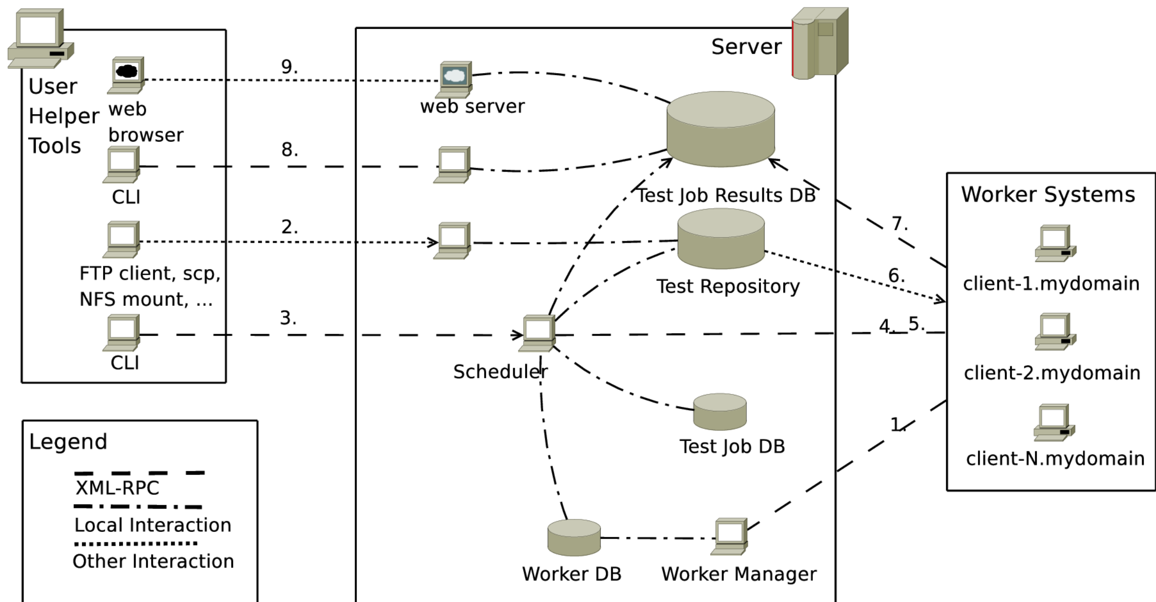


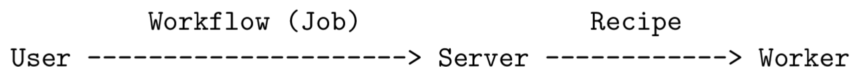
Figure 5.1: Chronological overview of designed system.

5.2 Detailed System Design

5.2.1 Work Description and Control

After evaluating requirements on the system and approaches in different tools (e.g. STAF) we came to conclusion that most appropriate to use the idea of XML document to describe the work and requirements on test job requested by the user, similarly as it is done in RHTS and will be done in Beaker project. This could provide easier compatibility with these systems in the future.

There will be two primary kinds of XML documents passed between user, server and worker: *test job* and *recipe*. Rather than reinventing the wheel we decided to use this concept that has been working in RHTS for many years and customize it for our needs. We preserved naming similar to the one used in RHTS, because structure of both documents will be based on it. First type of the XML document calls *test job*. More precisely it is a test job description document. It is a XML describing the job request from user to server. The second one calls *recipe*, and is generated by the server for each worker system individually from the test job. Its purpose is to serve as a control file for systems under test.



Example of proposed plain test job skeleton without additional attributes and values:

```
<job>
  <workflow> </workflow>
  <owner> </owner>
  <comment> </comment>
```

```
  <recipeSet>
    <recipe>
      <workerProperties>
        <worker name="" value="" />
      </workerProperties>

      <distroProperties>
        <distro name="" value=""/>
      </distroProperties>

      <envProperties>
         <env name="" value=""/>
      </envProperties>

      <test role="" name=""/>
      <test role="" name=""/>
        <params>
          <param name="" value=""/>
          <param name="" value=""/>
        </params>
      </test>
    </recipe>

    <recipe>
     ... second recipe in a job ...
    </recipe>
  </recipeSet>
</job>
```

Example of proposed plain recipe skeleton without additional attributes and values:

```
<recipe id="" job_id="" recipe_set_id="" status="">

    <workerProperties>
        <worker name="" value="" />
    </workerProperties>

    <distroProperties>
        <distro name="" value=""/>
    </distroProperties>

    <envProperties>
        <env name="" value=""/>
    </envProperties>

    <test avg_time="" id="" name="" result="" role="" status="">
    </test>
    <test avg_time="" id="" name="" result="" role="" status="">
    <params>
```

```
        <param name="" value=""/>
        </params>
    </test>
    <test avg_time="" id="" name="" result="" role="" status="">
        <params>
            <param name="" value=""/>
            <param name="" value=""/>
        </params>
    </test>
</recipe>
```

5.2.2 Watchdog and Heartbeat

The system will be equipped with two watchdog mechanisms, one on workers and second on the server. Local guard at the client worker system making sure tests do not exceed assigned execution time. Test run time is specified in the recipe. In case it exceeds assigned time quantum it has to be terminated and the action will be reported to the server. Log files will be collected from the system fro later analysis.

Another watchdog should be at the server side. Its purpose is to watch for broken/dead clients in case some test breaks the worker system in a way that it becomes irresponsive. Ideally this should never happen. Server is stateless. For this purpose we propose to use idea of heartbeats. Each worker will be assigned unique identifier during initial registration phase. Heartbeat holds, besides the identifier, data describing current state of the system.

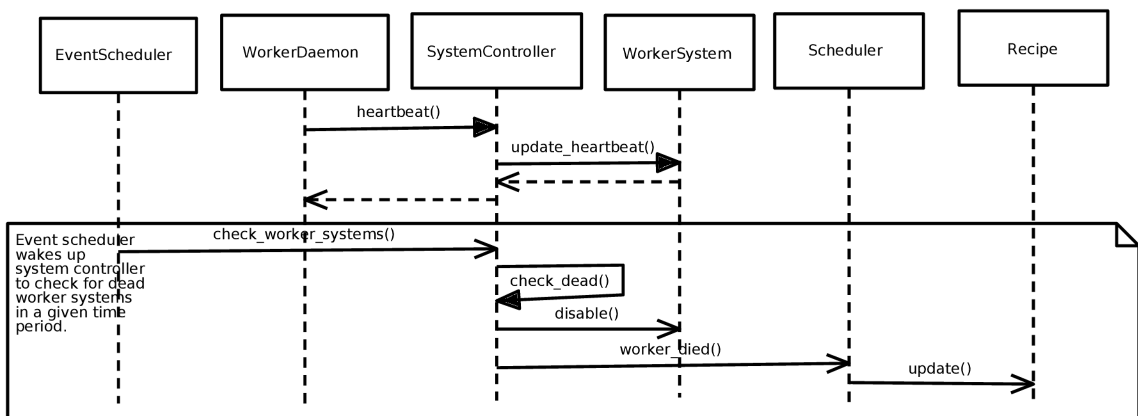


Figure 5.2: UML sequential diagram depicting the heartbeat between worker system (here represented by WorkerDaemon) and server.

5.2.3 Managing Available Systems in a Test Laboratory

Systems in a test laboratory, called worker systems or workers for short have to be installed and configured manually. Than the worker has to be registered on the server. Running a worker daemon on a correctly configured and registered system will trigger periodical

polling of the server for work.

Available systems can be searched using Web GUI provided by the server or listed in the shell console using command line client tool provided to users.

5.3 Server

Analysis of the requirements on the system results in a conclusion that the most appropriate architecture design pattern to use for the server would be Model-View-Controller (MVC). Two Python web frameworks were being considered: Django and TurboGears. Both are quite similar in functionality. TurboGears has better solutions to problems we experienced while exploring features of both frameworks (most important were XML-RPC, scheduler, database models). Greatest downside of TurboGears are too many dependencies and requirements for installation. Django framework comes hand in hand with the requirement on the system to be lightweight from the view-point of installation, application dependencies, and deployment. Figure 5.3 depicts individual parts involved in TurboGears application and matching the MVC design pattern. TurboGears glues together database models in SQLAlchemy (or SQLObject by default), templates or also called view using Genshi (or Kid by default), web handling controller is CherryPy, and JavaScript/Ajax can be handled with MochiKit.

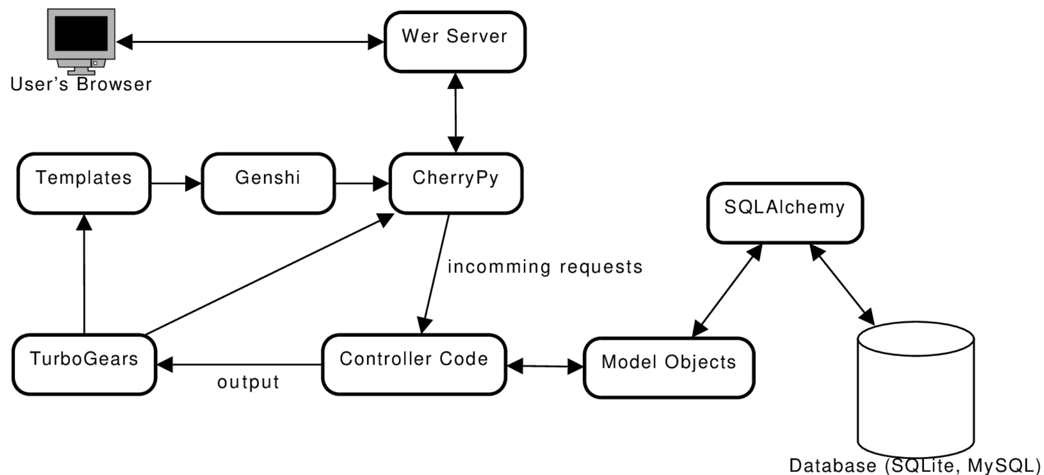


Figure 5.3: Diagram depicting core parts of a TurboGears application following the MVC design pattern. *Model Objects* represent application data, *Controller Code* accesses data in database using model's API and provides output to the templates and template engine (here Genshi). Incoming requests are processed through CherryPy (object-oriented web application framework).

Figure 5.4 depicts designed server classes. Complete database model can be found in Figure 5.5. Containment to Model-View-Controller design pattern is following. Model classes represent the application data that are stored in server's database. Controller classes operate on these data. Follows is a list with short description referring to the Figure 5.5:

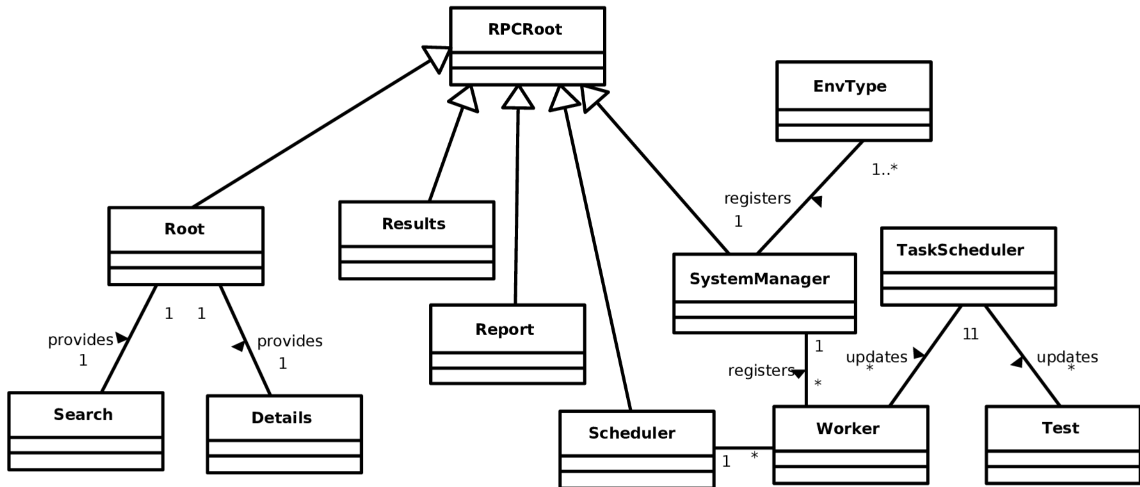


Figure 5.4: UML class diagram depicting designed server classes.

Job Highest test work unit. Consists of a recipe set and some additional information (e.g. job owner).

RecipeSet Serves as a virtual container for several recipes requested in a single test job.

Recipe Unit of work that can be assigned to single worker system at once. Composed of recipe tests, requirements on the test execution environment. Recipe has to have at least one test.

RecipeTest Representation of test on a recipe that should be executed on a worker system. It is composed of reported recipe test results and requested test parameters. It has to be associated with a test package represented by a Test class in the figure.

RecipeTestResult Test can have multiple results. Simplified mechanism for providing several sub-results for a single test run. Each can have associated a log file with it.

RecipeTestParameter Tests can be executed with arbitrary number of custom test parameters.

LogFile Provides some metadata for submitted log file.

Test Represents information about test available in a test repository. For example users can request or view (Web GUI) a list of available tests.

Worker System Information about system available in a test laboratory. Managed by a System Controller.

EnvType Systems can provide different test execution environments requested for a test job (more precisely recipe).

TestEnv For each recipe is created TestEnv object with purpose to enclose data describing test environment.

Result and Status Provide unified result and status values preserving consistency among objects involved in the testing.

Controller classes are:

RPCRoot This is a base class allowing other classes to export their methods via XML-RPC provided by CherryPy by inheriting it.

Scheduler The main part of the server side. Responsible for receiving, updating and providing test results. Polled by worker systems for work. Operates on almost all database model objects. Creates them based on XML description, updates their status and evaluates complex results. Not all database models are depicted in the figure, because of readability.

SystemManager Manages worker systems in the test laboratory. Worker systems register to it, report heartbeats, provide their configuration and so on.

Results Provides custom XML-RPC API expected by RHTS tests running in automated mode and willing to submit test results and log files.

Report Its purpose it to provide access to textual representation of current status and test results for job, recipe, test, worker system. The reports are a XML documents.

TaskScheduler There is a need to register and execute custom periodic tasks in a time. Different tasks can be registered to be scheduled for execution in given interval.

Root The main controller class used by CherryPy, to serve access to other pages, and other objects with registered XML-RPC methods.

Search Provide support for WebUI search functionality.

Details Provide control, logic, for web pages providing details about given model objects.

Viewer In MVC design pattern this would reside in the view. Provides templates presenting output information that is provided by the controller.

5.3.1 Scheduler

One of the most important component of the server is a scheduler. It provides the functionality that manages executing tests on worker systems satisfying given requirements on the architecture, distribution, test environment, and so on. Scheduler is visible to the worker systems which have to be configured to communicate with the desired scheduler.

Worker systems are polling the scheduler for work. The scheduler is most of the time not working and it is state-less. When worker polls it for work, scheduler searches the database for jobs/recipes in queued state waiting to be dispatched for execution. By inspecting worker's configuration it assigns the recipe to it in an atomic operation to prevent possible interaction with some other polling worker. The recipe holds all the information necessary for the testing as it is described in section 5.2.1.

It is important to point out that the scheduler does nothing unless polled for a work. All information needed to manage the scheduling are stored in a persistent form in the database. Unexpected or planned restart of the server does not affect the testing unless some test results were reported during the outage. Results and status changes are reported to the scheduler by calling provided XML-RPC methods.

5.3.2 Database

In the Model-View-Controller design pattern, models represent the data the system will work with. Both Python web frameworks that were evaluated support several database backends (PostreSQL, MySQL, SQLite). Figure 5.5 depicts class diagram for application models. In most cases removing records from the database is not desired. They should be stored for future analysis

Two options are for storing log files: in a database or in a dedicated directory. We would prefer storing log files in a database and providing optional shared storage configured for storing large files or tarballs containing several individual logs gathered by a test.

5.3.3 Test Repository

Test RPM packages will be stored in a Yum repository on the server. This will allow worker systems to download (and install) the test packages using `yum` package manager. Yum is not available for older Red Hat Enterprise Linux releases (older than RHEL 5). We suppose that it is available at the system, or it has to be installed manually. An example configuration of a local yum repository for both sides, client and server (tested on a Fedora):

Server side configuration example:

1. Create destination directory for packages (check your basearch and releasever):

   ```
   $ mkdir -p /var/www/html/yum/tests
   ```

2. Copy desired packages to the created test directory
3. Create repomd (xml-rpm-metadata) repository

   ```
   $ cd /var/www/html/yum/tests
   $ createrepo .
   ```

4. Configure httpd.conf
5. Supposing httpd to be configured for virtual hosts, add following to `httpd.conf` file:

   ```
   <VirtualHost *:80>
     ServerName 192.168.100.100
     <Directory "/var/www/html/yum/base">
       Options Indexes Includes FollowSymLinks
       AllowOverride All
       Order allow,deny
       Allow from all
     </Directory>
   </VirtualHost>
   ```

Client side configuration example:

1. Create new repository configuration file:

```
$ cat << _EOT >> /etc/yum.repos.d/test.repo
[my-local-test-directory]
name=My Test Repository
baseurl=http://192.168.100.100/yum/tests
enabled=1
gpgcheck=0
_EOT
```

2. Smoketest the repo, for example using following command:

```
$ yum repolist --disablerepo='*' --enablerepo=my-local-test-directory
```

As an future enhancement we were investigating other possibilities that could be used instead of Yum and RPM for a test package format. Interesting solution might be using Python `easy_install` and custom web-base repository of Python packages (called eggs in this case) as a replacement for a RPM package format. This might be an interesting future enhancement.

5.4 Worker Systems Running Tests

Tests are being executed in a test environment on a worker systems located in a test laboratory. Worker systems are managed by the server in a sense that it keeps track of available systems, their features and status. Figure 5.6 depicts class diagram of a worker system.

Register Before a new worker system can be used for testing it has to report its presence to the server (test laboratory manager). This step also involves providing system description and configuration to the server allowing him to reject test jobs that are not possible to be completed (requested configuration is not available).

WorkerManager Responsible for managing system status and configuration. Communicates with server's System Controller. Provides functionality for polling server for a new work using provided XML-RPC methods. As a response server can assign work to the worker (in a form of recipe). Status of the worker will change. It will stop polling until assigned work is done. Has Recipe Controller to check the recipe and then hands it over to process it.

Heartbeat Worker system registered on the server reports its presence by sending periodical heart-beats. Time period can be configured. If worker system stops sending heart-beats, it indicates that the system is not working properly most likely due to executing a test.

RecipeController Purpose of a recipe controller is to validate and check received recipes (XML document). Then it parses the recipe and creates corresponding recipe and test objects. Invokes Test Runner to handle the testing.

TestRunner Key component responsible for running tests in the recipe. Based on recipe requirements creates environment and uses it to handle the test in the environment.

Figure 5.7 depicts the interaction with Environment. Before running the test process watchdog is set for the test and if it does not finish within assigned time quantum, kill method provided by Env class is called.

Env This class provides access to the test environment used for test execution. Its purpose is to create and remove the test environment. Individual tests specified in the recipe (this is a XML document) are run using run and kill methods. The idea for such plugins was inspired by the Bazaar plugin architecture, freeIPA plugin architecture, and Pub project. Two environment plugins should be implemented. *Simple* providing no test env. isolation and thus executing the test directly on the host system. *Crutch* plugin should investigate possibilities for using change root environment and LVM snapshots for creation of a desired test environment.

Recipe, Test, Parameters Attributes of database model classes Recipe, Test, TestParam, TestResult are subset of here shown classes which hold also temporary information (e.g. pid).

We will need also to trigger actions in a configured periodic time intervals. This is needed for polling server for a new work, reporting heartbeats, and for watchdog functionality. One possibility is to set a system cron daemon for this purpose, or create separate class to provide required functionality, since test might be messing up with the cron's daemon scheduler. Or provide both approaches. Classes: Watchdog, Poller, HeartBeat would inherit it. The test might be also changing the system time, and requires proper test environment isolation.

5.4.1 Test Environment

The purpose of test environment is to establish and maintain an adequate environment, including test data, in which it is possible to execute the tests ideally in a manageable and repeatable way. Problematic part is proper test environment isolation.

The intention is to have physical systems (under test) without the need to be (re)installed after running each test job. In Beaker this is designed to be performed by cobbler and machines are physically (re)installed. This puts many hardware requirements on the laboratory environment. During specification phase it was decided to create lightweight system released that would be possible to use in shorter time, paying the price for possible vulnerability to be harmed by the running test (unless properly isolated test environment is used). Especially when the RHTS tests are by default running under root (superuser).

Proper isolation of a test environment can be critical for test execution. Executing a test in unpredictable/unknown environment is undesired, as it might be almost impossible to run the test under similar conditions, and do the analysis of results in future. Several possible solutions to the test environment isolation problem were evaluated:

- No change to the worker system (default option)
- Change root environment.
- Linux Logical Volume Management using read-write snapshots.
- Using virtualized guests provided by Xen or KVM.

RHTS test running in an automated environment expects requires several environment variables to be set. Namely they are: `JOBID`, `RECIPESEETID`, `RECIPEID`, `HOSTNAME`, `RESULT_SERVER`, `TESTID`, `TEST`, `TESTPATH`, `OUTPUTFILE`, `TESTVERSION`, `AVC_ERROR`, `RECIPETESTID`.

There should be defined some consistent way for reporting hardware and system profiles for systems under test and report it for each executed recipe. This information will be stored with the recipe for later result analysis. Similar situation applies for collecting system log files (e.g. `/var/log/messages`) from the system under test, after the test was forced to terminate by watchdog. Collected log files and profiles can be copied in a tarball (having unique name) to a shared NFS (Network File System) mount point with a reference to it in server's database.

Change Root Environment Using Logical Volume Management Snapshots

One of possible types of environment for testing is involving change rooted environment and LVM version 2. LVM version 2 supports a creation of snapshot logical volumes which keep the contents of the original logical volume for backup purposes, in our case for test isolation. Follows a list of possible steps how to use such an environment for testing. This could be done by a worker capable of creating it and started based on an attribute in a recipe.

1. Create snapshot (`lvcreate -s -n RECIPE-ID /dev/VolGroup00/desired-test-fs`)
2. Mount snapshot (`/my/snaps/RECIPE-ID`)
3. Bind mount `/proc /dev /sys` and perhaps other pseudo filesystems in snapshot root (`/my/snaps/RECIPE-ID/{proc,dev,sys}`)
4. Change root to the snapshot (`chroot /my/snaps/RECIPE-ID`)
5. Environment is ready to perform the testing (install and execute tests)
6. Leave the change root environment
7. Unmount mounted filesystems
8. Destroy the snapshot (`lvremove /dev/VolGroup00/RECIPE-ID`)

5.5 User Client Tool

User has to interact with the system using command line interface. For this purpose, the system will provide a single user tool providing all the required functionality. The tool will provide commands that are in fact plugins. Advantage of this approach is easy addition of new commands as plugins. Class name will be automatically converted to a command name (e.g. SubmitJob, Results classes in a Figure 5.8).

User can call the command using following scheme (where `late-client` is the provided CLI tool):

```
$ late-client <command> [parameters]
```

Basic helper script commands provided by the user tool:

submit Submitting a test job to given scheduler involves creation of a workflow (XML document describing request work) and sending it over to the server. This command provides the required functionality. Using command parameters user can specify, test, scheduler, environment type, requirements on the worker systems, and so on.

repo Provide support for accessing test repository and querying it using `yum` CLI tool.

report This command provides access to a job, recipe and test results. Using provided command parameters can be requested a list of tests in a recipe and using its unique identification results can be listed in the terminal. Available is also a list of worker systems or description for a worker using its ID. Complete and short reports will be available.

New tests will be developed using RHTS Framework. This framework will allow running tests locally in development mode. Introduction to development of new tests using RHTS framework can be found in Chapter 3 and in [5].

5.5.1 Viewing Test Results

Results will be accessible in two ways: (1) using provided commands in the CLI client tool, and (2) using web browser. Command line version is represented by the `report` command in user client tool `late-client` described in previous section. Requested results are in a textual form sent to terminal's standard output. XML is the most suitable data format for the output. It can be easily transformed into different formats for output or processing the data. Brief list of information provided for different components in both, CLI and Web GUI variant are:

- Job - id, date, owner, status, result
- Recipe - id, run, time,
- Test - id, recipe-id, start-time, finish-time, status, result, fail-logs, list of test's results
- Test results - id, path, result, score, log, log-file

The other way how to access job and test results is using Web GUI provided by the server. Only basic features (compared to RHTS) will be possible: list jobs, view recipes in a job, view test, sub-results for tests, view log files. Purpose of a Web GUI is to provide the same amount of information but in a more user-friendly way than with CLI. It should also allow searching jobs, recipes, tests and workers using ID, status, result or name (similar attribute) if appropriate.

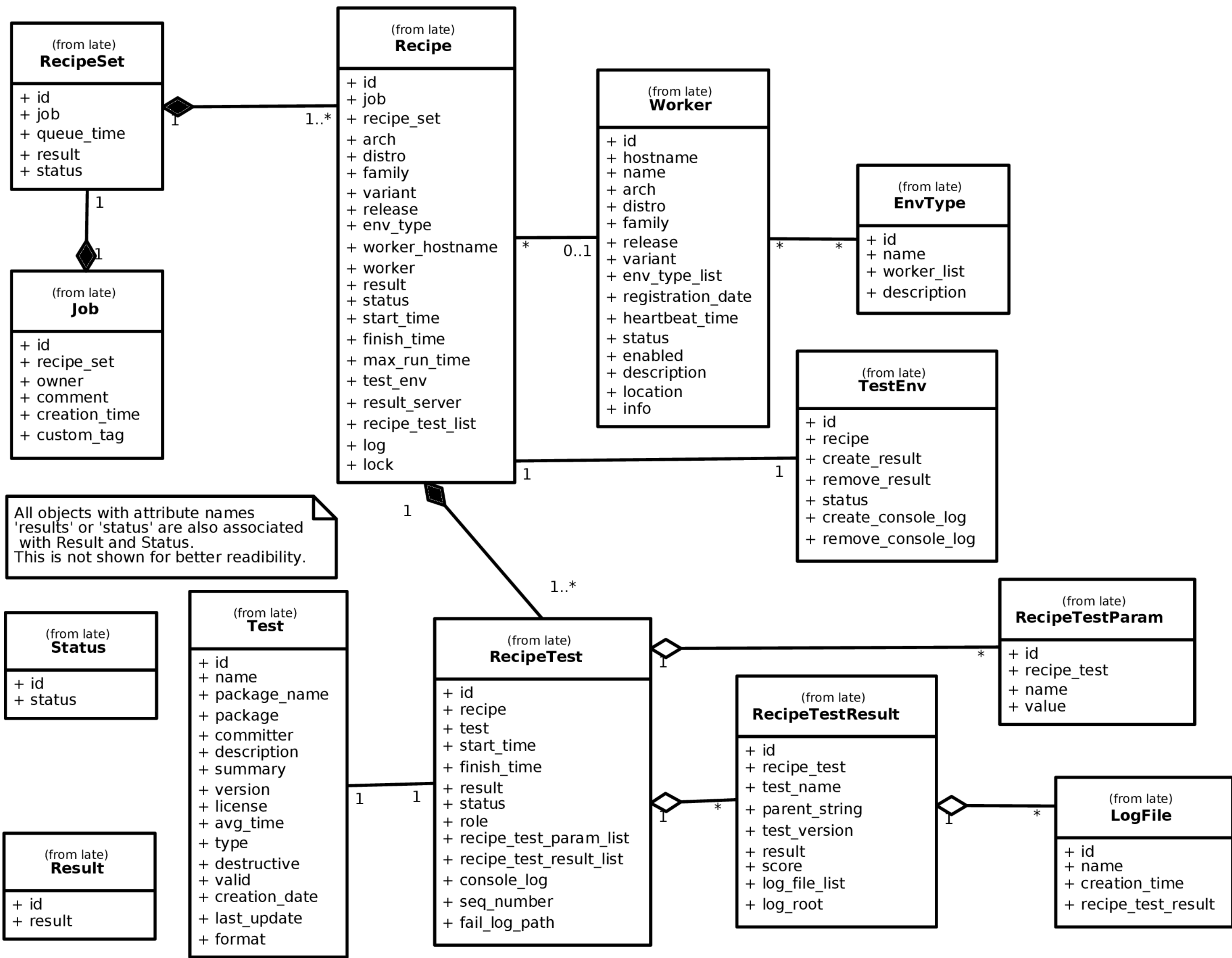


Figure 5.5: UML class diagram depicting database model classes in the proposed MVC design pattern.

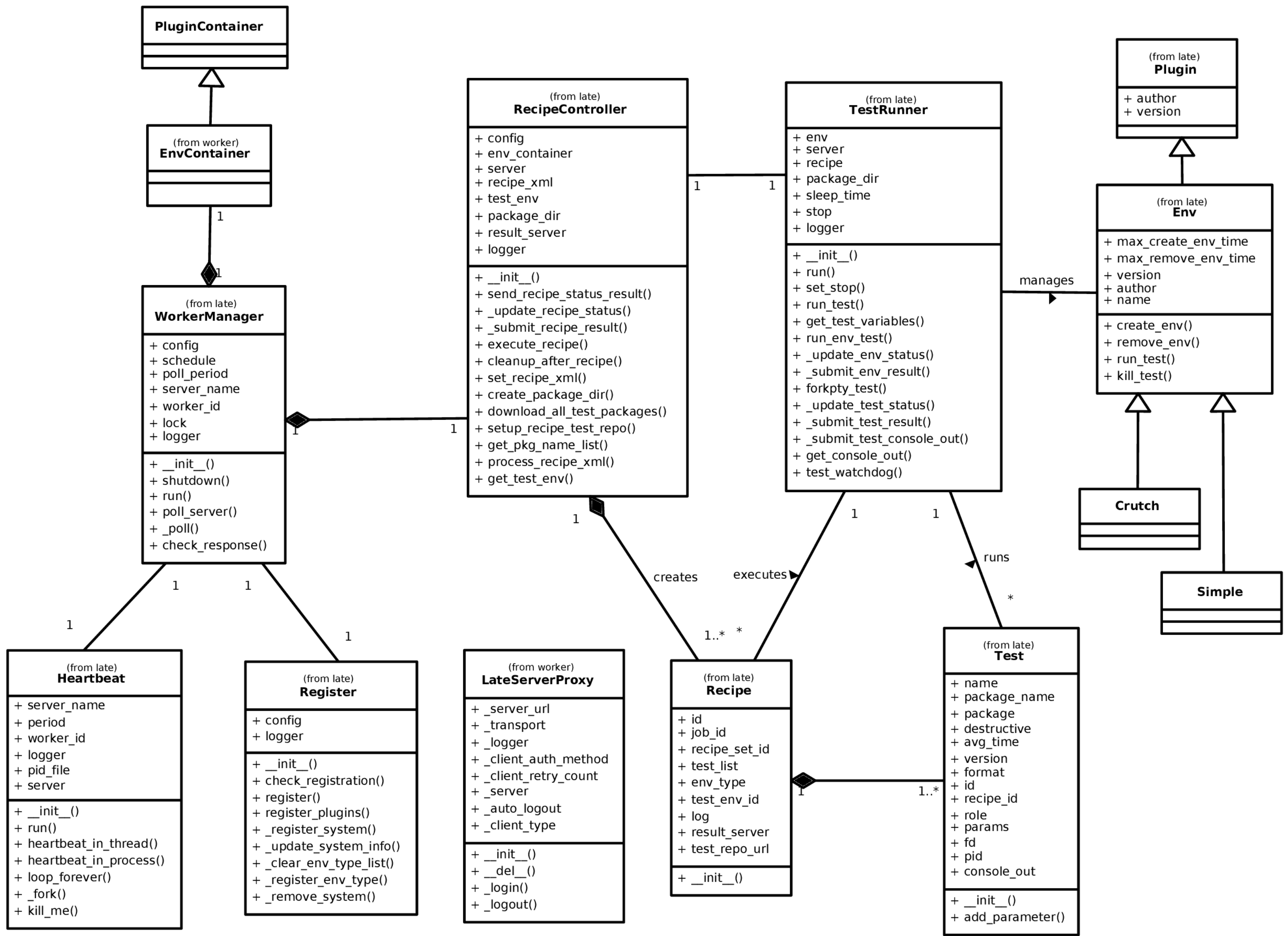


Figure 5.6: UML class diagram depicting the worker classes. Not all methods and attributes are shown.

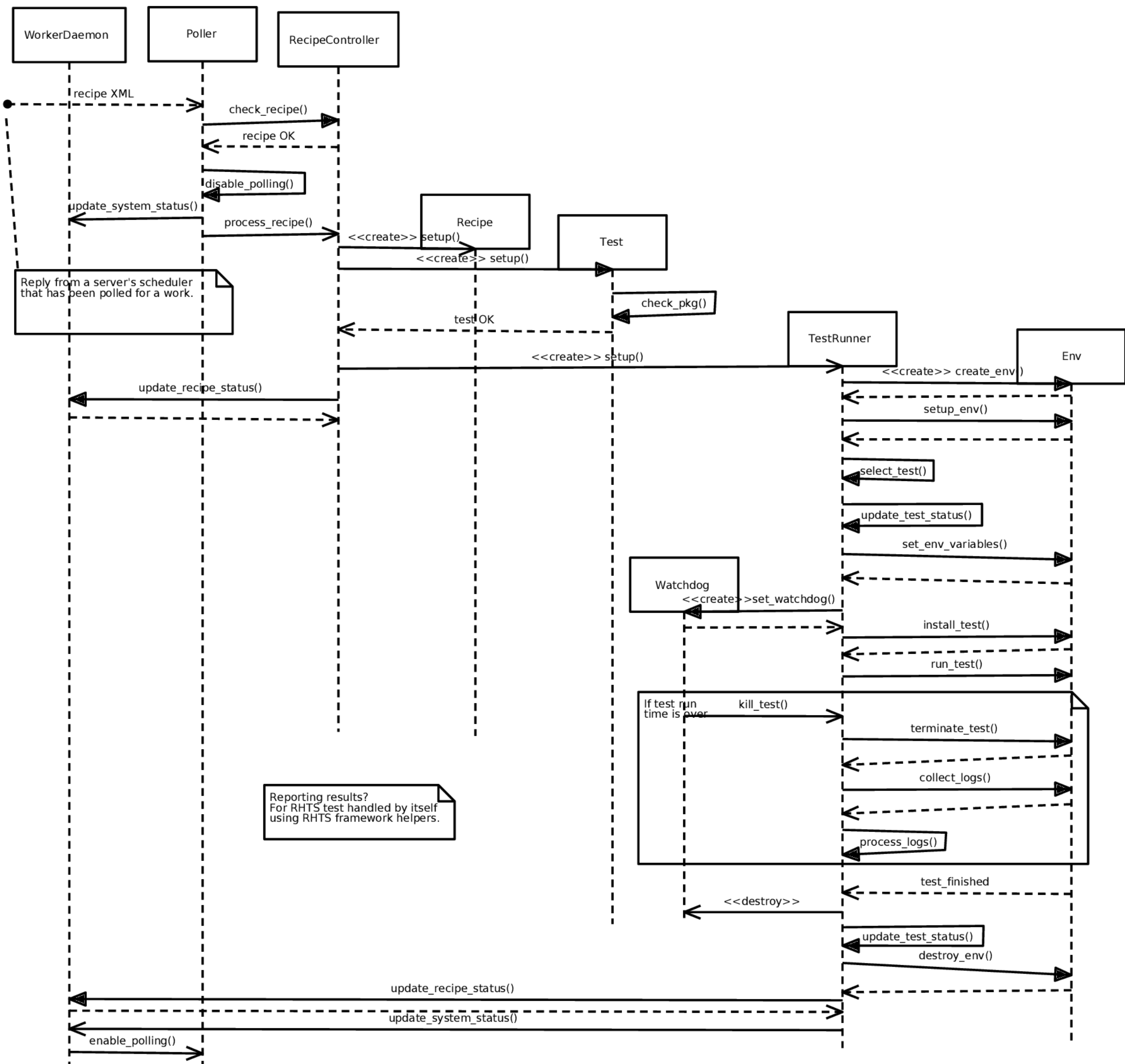


Figure 5.7: UML sequential diagram depicting running a single test described by a recipe (XML document) received in a reply from a server based on a worker's request for a work. Server classes are not shown and the activity starts with the reply. The recipe contains only one test without any test parameters. Diagram also depicts in a frame element possible termination of the test process invoked by the watchdog.

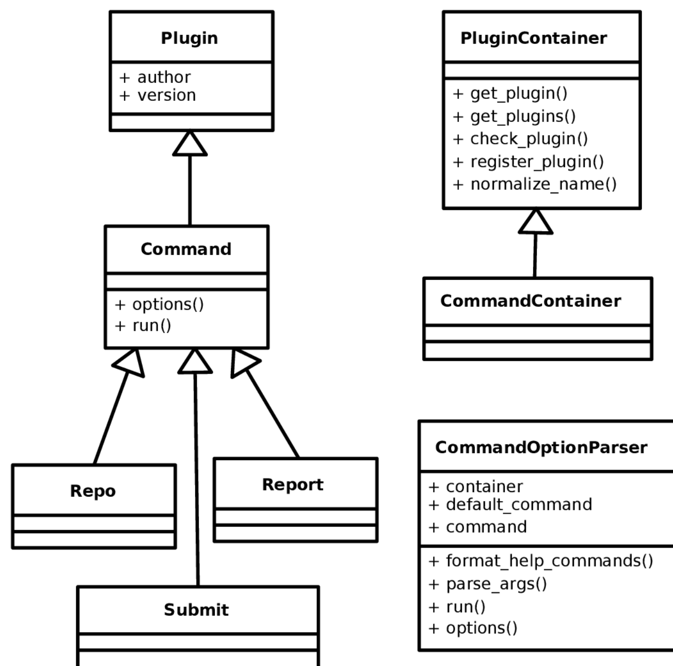


Figure 5.8: UML class diagram depicting design of client tool provided to users. Class names of classes that are inherited from Command are automatically converted to a command name.

Chapter 6

Implementation

This chapter deals with implementation specific problems, possible solutions, decisions that were made and reasoning. The final implementation of the system is quite complex peace of code, despite the fact that it is quite lightweight compared to all possible features, involving many non-trivial technologies. All planned, designed, features were implemented. Besides that, another new features showed to be a "must have" thing, to get a system usable by an average user. Others are left to be implemented as future enhancements. It should be noted that the plan is to submit the implemented system into Fedora distribution.

6.1 Overview of Used Technologies

The system is implemented entirely in a Python programming language. Python evaluated to be the most appropriate to fulfill all the requirements on the system described in Chapter 4. Especially the requirement on extensibility in the future, and allow to build on top of it. The final system is supposed to be customizable and should allow easy addition of a new future enhancements. Following list provides short overview technologies that were used, or considered. Most important representatives from the list are described later in this chapter.

- Cherrypy [12] is a lightweight HTTP framework serving us a web server as a part of the TurboGears framework. Apache is not required, but it's possible to run a CherryPy applications behind it.
- TurboGears (TG for short), rapid development web framework in Python, is used as the main part for the server application. Currently the latest version is 2, but we had to go with version 1 that is currently available in both, Fedora and RHEL. (More precisely in EPEL repositories.) Transition to newer version is possible in the future.
- Kid and Genshi are two templating mechanism that can be used by server application using TurboGears framework. By default we are using Kid, while Genshi can be used just by changing appropriate configuration files. It should be noted, that Genshi has much better performance than Kid, while Kid is the default templating mechanism used by TG.
- Communication among different distributed parts of the system is done using XML-RPC, a remote procedure call protocol using XML to encode its calls and HTTP as a

transport mechanism. This fits well with the server build on top of TurboGears web framework. XML-RPC API provides easy integration with other tools .

- Linux Logical Volume Manager (LVM for short) supports since version 2 (LVM2) read-write snapshots of logical volumes. This technology was used to create an environment plugin we named `crutch`. It is introduced in 5.
- Database and SQL is handled with SQLAlchemy and Elixir. SQLAlchemy [10] is a Python SQL toolkit and Object Relational Mapper. Elixir is a declarative layer on top of the SQLAlchemy library [15].
- As an underlying database we use by default MySQL for production and SQLite in a development mode. This is possible due to SQLAlchemy providing support for SQLite, Postgres, MySQL, Oracle and others.
- There is no restriction on a method that should be used for uploading new test packages to the server. Tests are expected in a dedicated directory, `/mnt/late/test_repo` by default, and can be uploaded there for example over NFS, SSH, HTTP, or any other custom, user specific, configuration.
- Yum package manager [43] together with `createrepo` utility is used to handle management of test packages and provide a test repository.
- Inotify and pyinotify [28] were considered to be used for watching filesystem evens, specifically to watch modifications on a directory holding test packages. Unfortunately this package is not yet available in a standard RHEL repositories or EPEL [19] repositories. Also it requires Linux Kernel feature (merged in kernel 2.6.13) called inotify. Therefore decision to implement own directory watch has been made. Solution using inotify could be cleaner and is left to be one of a future enhancements.
- Extra Packages for Enterprise Linux (EPEL) repositories [19] are required in order to install and run the system on a RHEL 5 system. EPEL provides Fedora packages for RHEL.
- Logging for all components of the system is provided using Python logging facility (so named module). Logs should be available in a files `/var/log/late-<component>.log`.
- Configuration with is handled with ConfigObj, a powerful configuration file reader and writer with ini file format [20].
- For a proper and standard way of handling worker daemon and server application SysV init scripts were written for them.
- Hardware profile of the test environment is obtained with smolt [39].
- Finally an RPM and SRPM packages were created for the application according to Fedora packaging guidelines, to support installation on latest Fedora and RHEL systems.

6.2 Server

Server application is build as a TurboGears application and uses CherryPy as a web server. It can be used with or without Apache. From various possible setups, we prefer and use running it standalone, or using Apache as a reverse proxy for the server application in TurboGears. This offers to take advantage of Apache's HTTPS abilities or have let it to serve custom static files. There are several problems that require periodic execution of some functions at specific intervals. This is accomplished using cron-like scheduler provided by TurboGears [47].

Server application provides access to a XML-RPC methods using the CherryPy. There is no other communication mechanism involved (excluding uploading data files or tests). Using this approach, the XML-RPC API aims to provide easy integration with other tools.

Work to be executed by a worker is described in a XML document, generated by the server and handed over to the worker in a response to calling `poll` method. Follows example of a server response to a worker system, assigning him some work:

```
<?xml version="1.0" ?>
<recipe id="1" job_id="1" recipe_set_id="1"
 test_repo_url="http://tarragon.englab.brq.redhat.com/test_repo/">
  <workerProperties>
    <worker name="ARCH" value="x86_64"/>
    <worker name="ID" value="1"/>
    <worker name="HOSTNAME" value="ibm-e326m.rhts.bos.redhat.com"/>
  </workerProperties>
  <distroProperties>
    <distro name="VARIANT" value="any"/>
  </distroProperties>
  <envProperties test_env_id="1" type="simple"/>
  <test avg_time="120" id="1" name="/examples/late/Sanity/basic-pass"
   package="tmp-examples-late-Sanity-basic-pass-1.1-0.noarch.rpm"
   package_name="tmp-examples-late-Sanity-basic-pass" role="STANDALONE"/>
</recipe>
```

As it is mentioned at the beginning of this chapter, to work with a database the application uses SQLAlchemy and Elixir. The main reason to choose this combination over using SQLObject is that it is going to be the recommended package for TurboGears 2.0. SQLObject was recommended for TG 1.1. SLQAlchemy supports many databases. We decided to go with two relational database management systems to be supported by default in the application: MySQL and SQLite. Support for others, like Postgre is possible to be added in the future.

`Status` and `Result` values were identified as the most critical to preserve integrity of test results. Values for other attributes, like architecture, custom tag, distribution, family, variant, release, could be also implemented as a separate tables. But to allow users of the system to use their own naming conventions, we decided not to apply this restrictions and follow the goal of having a light version of a control system, rather than adding more complexity. Several database model (`model.py`) classes are used to generate XML reports

or other XML documents. Therefore they implement `get_xml_element()` method using lightweight DOM implementation `xml.dom.minidom` to generate XML elements representing the object.

It should be noted that the server application and its scheduler part is meant to be state-less. Meaning that all required data and states are stored in the database and the scheduler operates on a top of it. Using this approach makes it easier and safer to restart it, for example for maintenance, upgrade, etc.

During an execution of a recipe, several textual or binary log files are being generated by the worker system or by RHTS test. These files are uploaded to the server and stored in a dedicated directory.

Currently the server does not provide identity management, but the project is preconfigured to provide easier integration in the future. Details about identity support in TurboGears can be found in [36].

There is also a plan to add support for authentication using Kerberos XML-RPC login methods. Example of a possible implementation can be found in a Koji project [11]. Unfortunately this was not possible to accomplish before the deadline set for this paper.

Monitoring availability and status of the worker systems is done by accepting simple heartbeats sent by individual systems. They are sent by simple agent run on the worker system together with the daemon. Further improvements are planned for future releases, but were not implemented yet due to limited time for this phase of the project.

There are two main approaches towards monitoring availability of worker systems (stations): (1) monitoring from the main station (e.g. ping, ssh, HTTP, and so on), (2) in cooperation with an agent on monitored station. We can monitor availability (currently done only by watching heartbeats from worker systems), direct or indirect information about the load (ping round-trip time, system load, memory usage, number of executed jobs, running processes, and so on). There is also a risk of evaluating the system as not available, while executing a test that actually created a high load and makes the system to act as not available. Therefore, a proper monitoring of not only the worker systems but also the possibly virtualized testing environments in the future will be quite complex task. One of solutions is to integrate support for existing monitoring systems like Nagios, Big Sister, or Zabbix.

6.3 Worker

6.3.1 Test Environment Plugins

To create environment plugins there is an `Env` base class that is meant to be sub-classed by the actual plugin. It defines API methods required by the worker system (more precisely `TestRunner` class) to interact with the actual environment. Namely they are: (1) `create_env` responsible for creation of a test environment, (2) `remove_env` responsible destroying and removing the environment, (3) `run_test` will install and run the test in the

environment, and (4) `kill_test` provides a mechanism to watchdog to kill a misbehaved test, exceeding dedicated time quantum. It should be also noted, that the creation and removal process is also handled as some special kind of test and has result and maximal run time. This can be set in the plugin code.

Currently there are implemented two environment plugins. Each of them provides different level of test isolation from the host system (worker).

Simple plugin provides a simple environment with no isolation of a running test from the worker system. Its purpose is to demonstrate how to implement a plugin. It is suitable for automated execution of tests that do not change configuration of an underlying system in a way that would destroy it, nor does it run or kill any processes or services that would disable the worker. To collect and report information about system hardware profile, smolt [39], hardware reporting tool for GNU/Linux based system, is used to generate the report and include it the log.

Crutch Plugin uses idea of LVM2 read-write snapshotting and `chroot` to provide file system isolation of the running test from a worker. This level of virtualization is still not sufficient.

6.3.2 Test Environment Isolation

As we already noted, there is a real need for proper isolation of a test environemt. There is a high risk of system failure when running destructive tests on the worker. Ideal solution is to use virtualization or direct provisioning of a real HW. Both of them have their pros and cons, and are suitable for different test requirements. Provisioning of HW leads to another requirements in order to deploy the systems successfully. The system would grow in complexity rapidly, not speaking about the time consumed during installation and re-installation after each testing cycle. Another solution might be to use the snapshotting in connection with virtualization. Prepare several original volumes according to our testing needs and create snapshot for each virtualized guest and run it on it.

6.3.3 Running in a Daemon Mode

SysV init script provides standard way for running the worker daemon. Most of the RHTS tests are designed to run with root privileges and modify the configuration of the underlying system. Therefore, we had to preserve this bad habit, and the `late-worker` also runs under with root privileges. This is another sign for the need of proper test environment isolation.

6.3.4 RHTS Framework in Automated Mode

RHTS Framework is together with test wrapper used to create illusion of an automated mode in regular RHTS laboratory for the test using all the helper scripts. RHTS tests executed by a `TestRunner` class behave like in a regular RHTS laboratory. In order to able to access our own server and its XML-RPC interface from RHTS Framework, we had to create simple patch to the `rhts-devel` package after installing it. This is done during the installation of worker package.

6.4 Client

Client application `late-client` provides set of commands to interact with the server. The communication with the server is done via XML-RPC. Client provides simple plugin architecture allowing to easily add new commands. This way users of the system can create new commands executing tasks that match their specific needs, and thus fulfills the requirement on extensibility in the future. Currently `late-client` provides two commands:

submit Provides CLI for submitting test jobs to the server.

report Allows to query server for reports about current status of job, recipe, recipe test, and worker system. We decided to use reports in XML format. This allows to manipulate easily with the data and create other custom reports. Two types of reports are provided: short, and complete.

repo Simple helper command to generate content of a `yum.repo` file for given server. Also provides some examples on how to query the repository.

Creating a new command is quite simple. All that has to be done is to create a new class inherited from `Command` class, implement methods `run()`, `options`, and let the command container know about the plugin by registering it using `register_plugin()` method. The final required step is to place the class file into `commands` directory and it is ready to be used.

6.5 Proposed Possible Future Enhancements

Along with the work on the thesis we come across many interesting projects and ideas, that could improve and enhance the implemented system, or improve the quality of using the system for testing. Here is a brief list of possible future enhancements:

- Web GUI enhancements and new features: scheduling new jobs, statistics, different reports providing results based on a user role (tester, developer, manager), and so on.
- User helper tool enhancements to provide more comfortable interaction with the system by providing most frequently used activities as a new commands.
- Standardized XML report using ATML [4] or TRPI [40] standard.
- Design more general test description based on ATML standard [4].
- Customized reports (for example using XSLT transformations), providing different level of information based on user role or selection.
- Customizable e-mail notifications.
- Allow users to set up a periodical tasks on the scheduler, similar as executing cron jobs. This would be useful for example to execute created test plan on some project during its development and watch progress.
- Port the system to different Linux distributions.
- SSL and Kerberos authentication for both, users and worker systems

- RHTS Framework, test API, and framework helper scripts re-implemented in Python. Many possible, different, approaches dealing with test automation, can be found for example in [6], [8].
- More universal test format based on RHTS but wrapped in Python Eggs.
- Different test environments as worker environment plugins. For example executing tests in a virtualized environment, or providing interaction with different platforms (e.g. Windows).
- Add support for GUI testing by integrating Dogtail [14] or Accerciser [44]. Support for other third party tools could be also added.
- Improve management of worker system. Better monitoring, alarms and notifications on email.
- Integrate support for versioning control system, for example Git. Tests in the test repository would automatically reflect changes on master branch.

Chapter 7

Testing and Experiments

This chapter describes testing of the implemented system and summarizes results obtained during execution of proposed experiments on test the system. At the end of this chapter we present usage examples and possible applications of the system for example by creating a RHTS tests for Wireshark, a popular network analyzer.

7.1 Testing the System

Several black box testing techniques were applied on the system, after implementation and integration testing. Follows an overview of tests that were executed during the development, with simple, short, description. Not all tests were possible to be executed due to time constrains of the project. In the rest of this chapter we used terminology used in [18].

- Functional testing is the main technique we used. It is described later in this chapter, and presents functionality of the system on executed tests.
- Regression testing to verify that found issues, which are supposed to be fixed already, were successfully resolved or do not appear again after fixing another bug.
- Random testing of areas identified during the development of the sample RHTS tests and the system itself. This testing technique was applied throughout all development and testing phases.
- For configuration testing of the system we used RHTS and it was tested on all available architectures for both RHEL and Fedora. Namely the architectures were: i386, x86_64, ia64, ppc, ppc64, and distributions: Red Hat Enterprise Linux 5 server Update 3, Red Hat Enterprise Linux 5 client Update 3, Fedora 9, Fedora 10. This activity focused on verifying installation dependencies and requirements on different distributions and verifying proper operation on machines with different hardware and software configurations.
- Long sequence testing by simulating concurrent user and worker system activity using simple automated scripts.
- Stress testing was performed to test the database behavior under simultaneous access from multiple client systems. Script on a client side had to submit multiple test jobs

in a short time to the server. Testing polling, heartbeat, and scheduler periods being as long as one second long. This would fit into stress testing.

- Source codes in Python were analyzed using following code analyzers: PyFlakes [7], Pylint [41], and PyChecker [30].
- Many other testing techniques and activities could be executed besides those listed above. Due to limited time and people resources not all were possible to fulfill, and we leave it as another possible future work on this project. Unit testing being one of the most desirable. It could be accomplished using PyUnit, standard unit testing framework for Python [34].

7.1.1 Testing the Functionality

Set of RHTS Tests

To test the system, and verify the functionality set of simple tests was created. Running these tests in various combinations exercises behavior of the system. Tests can be found in `examples` in a provided SRPM package.

Basic pass - `/examples/late/Sanity/basic-pass`
: Test to print environment info, sleep for 5 seconds and exit with result PASS and score 0.

Basic fail - `/examples/late/Sanity/basic-fail`
: Test is supposed to print environment info, sleep for 5 seconds and exit with result PASS and score 1.

Watchdog pass - `/examples/late/Sanity/watchdog-pass`
: Test should exceed assigned test run time (15 seconds) and should be killed by watchdog. If the test does not get killed by watchdog, is will report PASS result.

Watchdog fail - `/examples/late/Sanity/watchdog-fail`
: Test should exceed assigned test run time (15 seconds) and should be killed by watchdog. If the test does not get killed by watchdog, is will report FAIL result.

Subtests pass - `/examples/late/Sanity/subtest-basic-pass`
: Test simulates four subtests reporting their own results. All reported results are PASS and the overall test result should be also PASS.

Subtests pass - `/examples/late/Sanity/subtest-basic-fail`
: Test simulates four subtests reporting their own results. All reported results are FAIL and the overall test result should be also FAIL.

Subtests ppff - `/examples/late/Sanity/subtest-basic-ppff`
: Test simulates four subtests reporting their own results. Two subresults are PASS, two subresults are FAIL and the overall test result is expected to be FAIL.

In order to run the tests locally (or by the worker daemon) RHTS Framework is required. RHTS Framework quick start guide can be found in [32]. All listed tests are available from provided SRPM on attached CD disc in directory `examples`.

Testing late-client Application

Client command line application allows user to submit new tests to the server and retrieve reports. Here we present set of functionality tests executed on it.

1. *Submit 1*
   Description: Verify program start-up, and print help message.
   Input: $ late-client submit -h
   $ late-client report -h
   $ late-client repo -h
   $ late-client help
   Expected result: Program prints a help message.
   Actual result: Pass

2. *Submit 2*
   Description: Generate test job XML without committing a job. Exercise the -x option to print a XML that would be sent to the server in order to submit a test job.
   Input: late-client submit -s <your-server> -x -T /aaa/bbb/ccc
   Expected result: Correct structure of the XML document printed into the console.
   Actual result: Pass

3. *Submit 3*
   Description: Submit one test and one recipe to the server.
   Input: $ late-client submit -s <your-server> \
   -T /examples/late/Sanity/basic-pass
   Expected result: Job accepted and its it printed out.
   Actual result: Pass

4. *Submit 4*
   Description: Submit one test multiple times in one recipe to the server.
   Input: $ late-client submit -s <your-server> \
   -T /examples/late/Sanity/basic-pass \
   -T /examples/late/Sanity/basic-fail \
   -T /examples/late/Sanity/basic-pass
   Expected result: Job accepted, its id returned.
   Actual result: Pass

5. *Submit 5*
   Description: Submit multiple tests in one recipe to the server.
   Input: $ late-client submit -s <your-server> \
   -T /examples/late/Sanity/basic-pass \
   -T /examples/late/Sanity/basic-fail \
   -T /examples/late/Sanity/subtest-basic-pass
   Expected result: Job accepted, its id returned.
   Actual result: Pass

6. *Submit 6*
   Description: Submit test with custom test parameters to the server.
   Input: $ late-client submit -s <your-server> \
   -T /examples/late/Sanity/basic-pass -P XLOGIN=xlogin00
   Expected result: Job accepted, its id returned.
   Actual result: Pass

7. *Submit 7*
   Description: Submit one test in more recipes to the server.
   Input: $ late-client submit -s <your-server> -w NAME=worker1 \
   -T /examples/late/Sanity/basic-pass -w NAME=worker2
   Expected result: Job accepted, its id returned.
   Actual result: Pass

8. *Submit 8*
   Description: Submit more tests in more recipes to the server.
   Input: $ late-client submit -s <your-server> \
   -T /examples/late/Sanity/basic-pass \
   -T /examples/late/Sanity/basic-fail \
   -T /examples/late/Sanity/subtest-basic-pass \
   -w NAME=worker1 -w NAME=worker2
   Expected result: Job accepted, its id returned.
   Actual result: Pass

9. *Submit 9*
   Description: Submit a test that is not available in the repository.
   Input: $ late-client submit -s <your-server> -T /aa/bb/cc
   Expected result: Job rejected by the server.
   Actual result: Pass

10. *Submit 10*
    Description: Submit a correct custom XML file using the '-f' option.
    Input: $ late-client submit -s <your-server> -f good-custom-job.xml
    Expected result: Job accepted by the server.
    Actual result: Pass

11. *Submit 11*
    Description: Submit a bad XML file using the '-f' option.
    Input: $ late-client submit -s <your-server> -f bad-custom-job.xml
    Expected result: Get info about document being not well-formed. Nothing gets submitted to the server
    Actual result: Pass

12. *Submit 11*
    Description: Submit a test job to be executed on worker with bad hostname.

Input: $ late-client submit -s <your-server> \
-T /examples/late/Sanity/basic-pass -w HOSTNAME=janedoe -e simple
Expected result: The test recipe should be rejected.
Actual result: Pass

13. *Report 1*
    Description: Retrieve short XML report for a given job, recipe and test ID.
    Input: $ late-client report -s <your-server> \
    -j 10 -j 9 -r 3 -r 4 -t 14 -t 15
    Expected result: Short reports for requested job, recipe, and test IDs is printed into console, and contains sane data.
    Actual result: Pass

14. *Report 2*
    Description: Retrieve complete XML report for a given job, recipe and test ID.
    Input: $ late-client report -s <your-server> -c \
    -j 10 -j 9 -r 3 -r 4 -t 14 -t 15
    Expected result: Complete reports for requested job, recipe, and test IDs is printed into console, and contains sane data.
    Actual result: Pass

15. *Report 3*
    Description: Retrieve list of all available worker systems.
    Input: $ late-client report -s <your-server> -W
    Expected result: List of available worker systems is printed out into the console.
    Actual result: Pass

16. *Report 4*
    Description: Retrieve report for a worker system using its ID.
    Input: $ late-client report -s <your-server> -w 13
    Expected result: Response is printed out into console.
    Actual result: Pass

Testing Worker System

1. *Worker Daemon 1*
   Description: Under root privileges verify starting and stopping the service using SysV init scripts: start, stop, status. All distributions.
   Input: $ sudo /sbin/service start
   $ sudo /sbin/service status
   $ sudo /sbin/service stop
   Expected result: All actions were successful and executed specified action.
   Actual result: Pass

2. *Worker Daemon 2*
Description: Setup the configuration file 'worker.cfg' properly and register the worker to given server. Verify all plugins were registered and the system is listed as available on the server.
Input: `$ late-worker -r`
Expected result: Worker registered and listed on the server.
Actual result: Pass

3. *Worker Daemon 3*
Description: Display worker information while it is started as daemon.
Input: `$ late-worker -i`
Expected result: Information about worker including configuration files and PIDs of daemon, heartbeat, are printed out.
Actual result: Pass

4. *Worker Daemon 4*
Description: Deactivate one plugin from two, change system description and re-register the worker on the same server.
Input: Edit worker configuration file `/etc/late/worker/conf/worker.cfg` and remove one plugin from active_list in plugin section (eg. to list `active_list="simple"`).
Expected result: Only one plugin should be listed for this worker.
Actual result: Pass

5. *Worker Daemon 5*
Description: Submit a test job in a way that will lead to execution on specific worker system. Test case *Submit 8* can be modified for this. Make sure the test is executed correctly.
Input: On worker run `$ late-client submit -s <your-server> \`
`-T /examples/late/Sanity/basic-pass -w HOSTNAME=<worker-hostname>`
Expected result: By observing worker log files execution of the test is confirmed.
Actual result: Pass

6. *Worker Daemon 6*
Description: Submit a test job in a way that will lead to execution using given environment plugin on this worker system.
Input: On worker run `$ late-client submit -s <your-server> \`
`-T /examples/late/Sanity/basic-pass -w HOSTNAME=‘hostname‘ -e simple`
Expected result: By observing worker log files, proper execution of the test is confirmed.
Actual result: Pass

7. *Worker Daemon 7*
Description: Submit a job that should be killed by a watchdog after exceeding assigned time quantum.
Input: `$ late-client submit -s <your-server> \`

-T /examples/late/Sanity/watchdog-pass -w NAME=<worker-name>
Expected result: By observing worker log files termination of the test is confirmed. Test status is WATCHDOG and result WARN.
Actual result: Pass

8. *Worker Daemon 8*
   Description: Verify that all packages specified in a recipe are downloaded correctly into directory set in a configuration file during recipe execution.
   (default is /mnt/late/local_pkgs/)
   Input: During recipe execution $ ll /mnt/late/local_pkgs/
   Expected result: All packages are downloaded successfully.
   Actual result: Pass

9. *Simple Plugin 1*
   Description: Verify that hardware profile and list of installed RPMs is included in the log from environment creation test.
   Input: On server view recipe details and follow 'Create Console Log' link.
   Expected result: Log lists output from smolt and list of packages.
   Actual result: Pass

Testing Server

1. *Server Start 1*
   Description: Setup database, and run the server as non-root and root user from console. All distributions.
   Input: $ sudo start late
   $ sudo start-late
   Expected result: Server is successfully started in both cases.
   Actual result: Pass

2. *Server Start 2*
   Description: With superuser privileges run the server using SysV init scripts: start, stop, status.
   Input: $ sudo /sbin/service late-server start
   $ sudo /sbin/service late-server status
   $ sudo /sbin/service late-server stop
   Expected result: All scripts performed given action successfully.
   Actual result: Pass

3. *Server Repository 1*
   Description: On a remote system add a new repository configuration file and verify it. All packages in a repository directory should be available. Install them locally.
   Input: Add new repository on remote system, could be generated using late-client and its repo command. Execute examples printed by it into the console.
   $ late-client repo -s <your-server>

```
$ yum install <list-of-test-packages>
```

Expected result: All test packages are listed, and were installed locally.
Actual result: Pass

4. *Server Repository 2*
   Description: Add new test to the repository directory. Verify it is made available.
   Input: Add new test
   ```
   $ cp <new-test-rpm> /mnt/late/test_repo
   ```
   Wait for few seconds and query the repository for all available tests as in previous test case. *Expected result*: New test is available.
   Actual result: Pass

5. *Server Repository 3*
   Description: Upload new version of a test into the repository. The test info/metadata should be updated.
   Input: Copy new version of a test that is already present in the directory, but for example has changed "Test Time".
   ```
   $ cp <new-test-version-package> /mnt/late/test_repo
   ```
   Observe messages in a log file /var/log/late-server.log about test being updated, and check changes in webUI tab "Test Repository" by searching for given test name. Wait for a test repository refresh. *Expected result*: Test metadata are updated.
   Actual result: Pass

6. *Server Heartbeat 4*
   Description: Verify that server processes heartbeats from worker systems correctly. Register a worker system and start its daemon. Observe status change of the worker from NO_HB ->READY.
   Input: Setup the worker using test case *Worker Daemon 2* and run it
   ```
   $ sudo late-worker
   ```
   Expected result: Short time after starting worker daemon status change should happen.
   Actual result: Pass

7. *Server WebUI 1*
   Description: Exercise functionality of all tabs in a upper menu bar. Namely they are: Home, Test Jobs, Recipes, Test Runs, Test Repository, Worker Systems.
   Input:
   Expected result: Individual pages for each tab should be displayed.
   Actual result: Pass

8. *Server WebUI 2*
   Description: Exercise functionality of search forms on pages access following tabs: Test Jobs, Recipes, Test Runs, Test Repository, Worker Systems. Try to open all links available from each table displayed on individual pages.
   Input:

Expected result: Only records matching given criteria should be listed, all links should be correct.
Actual result: Pass

9. *Server WebUI 3*
   Description: Display detailed information about Test Job, Recipe, Test Run, Test in a repository, and worker system.
   Input: Enter ID of the object to display into "Goto ID" field from the search form and click "Search" button. Verify functionality of all links in detailed page.
   Expected result: Correct detailed page for given ID should be display, with all links.
   Actual result: Pass

10. *Server WebUI 4*
    Description: Exercise navigation of tables holding list of records (Test Jobs, Recipes, Test Runs, Test Repository, Worker Systems) using simple navigation menu placed above each table. Go to first, last, previous, next, specific page.
    Input: None
    Expected result: Works as expected from executed action.
    Actual result: Pass, but there was an request improve usability by reversing order of displayed records in the tables, newest records first.

11. *Server WebUI 5*
    Description: Disable and enable a registered worker system and change its info in detailed worker page.
    Input: Click "Worker Systems" tab from main menu, click on worker ID in the table. Detailed worker page will be displayed. Set "Enabled" radio button to "No" and save using "Apply changes". Next set Enabled to "Yes" and save again.
    Expected result: Worker system is enabled and disabled successfully, notes to info are stored.
    Actual result: Pass

7.1.2 Load and Long Sequence Testing of the Server

During load testing the server, it was put under simulated work load from three different client locations. At the same time simple Bash scripts were executed on client systems to simulate the work. During the test each client submitted 1000 test jobs and retrieved complete job report for each submitted job, with a sleep time 0.01 sec, in first run, and 0.001 sec, in second run, between individual actions on each client.

The configuration of a test environment was following. Host system with CPU Intel(R) Core(TM)2 CPU T7200 @ 2.00GHz, with 2GB RAM, running 2 virtualized KVM quests, each with 1 CPU and 512MB of RAM assigned. One running server on Fedora 9, and other running worker on Fedora 10. Base system was running Fedora 9. All systems were updated to latest package versions available at the time of writing (April 2009). Polling period set for worker system was 10 seconds, server heartbeat checking period for 15 seconds, and test

database update interval 15 seconds. As a database MySQL was used.

All test jobs were successfully submitted and all reports were retrieved. Worker system was still polling and executing assigned recipes. After the test, the webUI of system (now with several thousands of new records in the database) performed notably slower, but still acceptably, when displaying list of jobs, recipes, and test runs. But not while displaying list of test in the repository, where we had just seven examples of RHTS tests. Short investigation led to conclusion, that the slow-down is a known performance drawback of Kid templating engine. It is the default templating engine in TurboGears version 1.0. Solution would be to convert from Kid to Genshi templating engine. Genshi is designed to address this performance issues users experienced with Kid. This is left as another future enhancement to the webUI part of the server in later releases of the project.

Next we installed the worker and server on a workstation and run it for several days under low load. Evaluation of the log files revealed unexpected Tracebacks generated by TurboGears scheduler executing periodic tasks and most likely related to handling Yum repositories during test database update. More specifically the error message says: `Error accessing file for config file:///etc/yum.conf`, what leads us to conclusion that it was caused by accessing the repository during system update. The most important issue was when the system is using SQLite database. This was expected and gathered log files proved that there is a problem with file locking mechanism and concurrency in SQLite v3. Therefore it is recommended only for development or minimal database load.

7.1.3 Testing Python Source Code

In order to help delivering a good quality of code we decided to run serveral Python source code analysing tools: Pylint, Pychecker, and PyFlakes.

PyChecker system provides support for catching a large set of common errors. It finds problems that are typically caught by a compiler for less dynamic languages. Running the source code through it prior to testing or delivery can catch any lurking potential problems. It is also used for Python standard library. [26]

Pylint is similar to PyChecker in functionality as you can perform the same test scenarios with it as with Pychecker. But it is more feature rich, for example it provides checking line-code's length, checking if variable names are well-formed, provides better customization and configuration options and more.

PyFlakes is the third source code analaysis tool we used. It is also a lint-like tool for Python, similar to PyChecker. Its main main advantage against PyChecker is that it is faster.

7.2 Possible Usage Examples

The system is primarily designed for automated test execution. But it can be used also for other tasks. Modifications and enhancements can make it ready for many usage scenarious not only those described in this section.

7.2.1 Getting the Community Involved

Developers of some software project create a set of RHTS tests for their project and make them available throught the project internet pages to the community. Users can then install our control system for application testing on Fedora system and run these RHTS tests in their own testing environment. The structure of results, reports, log files and test logic have format created by the test developer. Users can submit obtained results and log files, back to the developer. In addition to providing one's own test packages, developers can create their own environment plugins. This allows them to modify or setup testing environment in user conditions in a common way, but that is not suitable to be done in an actual test. These days it is imporant to get the community involved in a testing process. Allowing it to enhance, customize, and contribute to some project seems to be the most efficient way.

7.2.2 Creating RHTS Tests for Wireshark

To support ideas presented in previous paragraph discussing creation of a RHTS test for particular application, we developed two RHTS tests for a real-life software application called wireshark. Wireshark (formerly known as ethereal) is a popular packet sniffer and network analyzer. By downloading the source tarball from project pages we obtained two tests used by upstream to test the application: *dfilter test*, and *fuzz test*. Both can be found in `tools` directory after extracting the source tarball. For detailed description of these two tests, please refer to the sources. After minor modifications and wrapping some logic into `rhntest.sh` new RHTS tests are ready to be used. Fuzz tests first generates packets for different protocols, creates capture files and then fuzzes them before running `tshark` on all of them (for example until an error is found). Dfilter test simply wraps provided test-suite to test wireshark's dfilter mechanism into RHTS test. Tests are placed under `/examples/wireshark/Regression/<test-name>` namespace. As the name suggests both can be used as a regression tests to test new releases of Wireshark. Final RHTS tests can be found on the CD in `dfilter-test` and `fuzz-test` directory under examples. By running these two tests, we can verify new releases of wireshark or disclose new bugs.

7.2.3 Evaluating Student Projects

Another possible usage example offered by the universality of implemented system allows it to be used for evaluation of student project in a batch. Suppose we have hundred students in a course, and all of them have the same assignement. The final hand over files is a commad line application in a tarball and can be build using provided Makefile. For the evaluation purpose instructor creates a RHTS test with the evaluation logic. Limited part of this test can be provided to students allowing them to verify partial correctness of their solution locally on their desktops. Additionally intructor can create a custom command for `late-client` tool that will process all gathered results and create for him the most appropriate report from the XML.

First we need to create a RHTS test with the test logic. There are several possible approaches to accomplish this. We can have a test that will be run once for all projects, and report all results as subresults. Or we can run a test individually for each project. This can be accomplished by creating a test job with a single recipe, that will run the same test several times, but with different parameters (e.g. login of the student). Running individual

tests has some advantages. Each project will be run individually, and in case it exceeds assigned run time, it can be terminated by the test watchdog. The former approach, allows to set watchdog only for the test as a whole. Thus one misbehaved test in the middle of evaluation, can waste the remaining time quantum.

Here is an example of a simple test structure. We submit results of individual testing phases as subresults using following naming schema `/xlogin00/<phase>[/<subtest>]` having a result (PASS, FAIL) and score (integer value), that can represent points gained for the task tested.

1. Download a project tarball from a shared location. Report subresult for download phase, e.g. `/xlogin00/download`. Finish if failed.
2. Extract the tarball. Report subresult for extraction, e.g. `/xlogin00/extract`. Finish if failed.
3. Build the project using provided Makefile. Report success, e.g. `/xlogin/build`. Finish if failed.
4. Run set of subtests to verify correctness of the solution. Report subresult in form of `/xlogin00/correctness/subtest` with result and score for accomplishing it.
5. Report final result and score and perform final cleanup after running another test (e.g. `/xlogin00`).

Along submitting an individual results using `report-result` function, the log file specified by `$OUTPUTFILE` gets submitted to the server. This allows further analysis of the test run.

After creation of the test and testing it locally on some sample data, we can setup a server with the RHTS tests in its repository, setup worker systems, register them, and submit the test job. Or use existing infrastructure.

Results and scores can be examined manually using the WebUI, but with many students in the course, this would be really incomfortable. Therefore, we would suggest to automate this step using the report command (`late-client report` and parse the results from a XML reports. Obtaining the log files can be also automatized by composing its URL from attributes of the result element, and download it locally.

7.2.4 Supporting Collaborative Testing in Open Source Project

Open Source projects involving several developers and tester could benefit from using implemeted system as an ultimate collaboration tool for testing their project. It is supposed to be easily customizable and thus satisfy specific project requirements. Tests in a repository can be executed at predefined phases (dates) or periodically to track current and progress of the project.

Results are available to all project members and can be stored for further result analysis together with log files. Here are several usage examples in a development (or testing) lab:

- Create a test that will download latest project source codes from different branches and will try to build them internally. This can be done every day, and results can be presented on public pages to inform whether it is suitable for download.
- Collect metrics of the project.
- Run set of regression, sanity, functionality, performance, analysis, and other types of testing techniques.
- Perform any other tasks that are suitable for automation.

By default the system does not support GUI testing, but as already noted in possible future enhancements this would be accomplishable by adding other tools (e.g. Dogtail). Supporting it by addition of a defect tracking tool (e.g. Bugzilla), version control system (e.g. Git, Svn), test management (e.g. Testopia), could lead to creation of a solid software development infrastructure.

Chapter 8

Conclusion

In this work, we addressed the need for testing software and its quality using automated software tools and test automation. Implemented system allows controlled execution of tests (in RHTS format) in distributed test laboratory. Another contribution of this work is in familiarization with software testing. We gained knowledge of different approaches to evaluating and comparing software testing tools. This knowledge was used to evaluate Red Hat Test System (RHTS) and to compare it with other tools. According to our evaluation most comparable tool to RHTS from available open-source projects is Software Testing Automation Framework (STAF) though it provides less functionality. RHTS is quite specific tool, because it was designed to help Red Hat's quality engineering department to qualify releases of Red Hat Enterprise Linux. Therefore, its functionality is hard to replace by some other tool. Another important contribution of this work is the evaluation and comparison of RHTS.

Test automation is quite popular today, but during the work on this project we came to a conclusion that before making the decision to automate a test, the value of automating it should be confronted with the value and effort to develop the automated version.

Gathering requirements of the system was more complicated than we expected. Especially functionality requirements. Addition of all useful features would lead to an extremely complex system. It is quite difficult to define the boundary for the designed system to be usable in an efficient way, lightweight, easy to use, and extensible in the future.

The project is implemented entirely in Python programming language. Work description and control is done using XML which seems to be today a standard way as it can be seen in [4] and [6]. We decided to use the concept that has been working in RHTS for many years and customize it for our needs. Red Hat Test Framework providing RHTS test API and some helper tools for test development, is publicly available. Analyzing it revealed that it can help to run RHTS tests in our automated test environment, and provide documented way to create new tests. Another key step was the decision to use Python web framework for server side. It provides good database backend and many other useful features. This will support addition of many future enhancements to the project.

Another area we addressed is a need for proper test environment isolation. For this purpose, worker systems provide plugins capable of creation of a such environment. Implementation of a plugin capable of using Kernel-based Virtual Machine to provide virtualized

guests or other virtualization technology for isolated test execution is left as a future enhancement.

There should be just one tool provided to the user instead of a bunch of different scripts doing similar tasks. We decided to use single user helper tool which provides set of commands and supports easy creation of new ones.

The system consists of a set of components, which can be easily used, modified, and extended. Many possible future enhancements were proposed. They prove the importance and value of this project. List of possible future enhancements can be found at the end of Chapter 6. After final, testing we proposed several possible usage examples. The project was packaged into RPM packages and is planned for inclusion in Fedora project.

Finally, several tests and experiments were executed on the system. For example creation of a RHTS tests for upstream tests used for Wireshark, or an example of possible application for automated evaluation of student projects. Implemented system is not an absolute tool for testing. It should be universal enough to be successfully integrated into the whole testing life cycle together with other tools that support test management, defect tracking, test plan support, and other third party tools.

Bibliography

[1] Linux Test Project. URL: `http://ltp.sourceforge.net`. (Last visited: December 2008).

[2] SAFS Software Automation Framework Support. URL: `http://safsdev.sourceforge.net`. (Last visited: December 2008).

[3] Table Cloth. URL: `https://testing.108.redhat.com`. (Last visited: November 2008).

[4] ATML. URL: `http://grouper.ieee.org/groups/scc20/tii/index.htm`, 2007. (Last visited: December 2008).

[5] RHTS Test Writing. URL: `https://testing.108.redhat.com/wiki/index.php/Rhts/Docs/TestWriting`, 2007. (Last visited: December 2008).

[6] Software Testing Automation Framework (STAF). URL: `http://staf.sourceforge.net`, 2008. (Last visited: December 2008).

[7] PyFlakes. URL: `http://freshmeat.net/projects/pyflakes/`, 2009. (Last visited: April 2009).

[8] Robot Framework. URL: `http://code.google.com/p/robotframework/`, 2009. (Last visited: April 2009).

[9] A. Abran, J. W. Moore, P. Bourque, R. Dupuis, and L. Tripp. *Guide to the Software Engineering Body of Knowledge (SWEBOK)*. IEEE, 2004. ISO Technical Report ISO/IEC TR 19759.

[10] M. Bayer. The Python SQL Toolkit and Object Relational Mapper. URL: `http://www.sqlalchemy.org/`, 2009. (Last visited: April 2009).

[11] M. Bonnet. Koji - RPM building and tracking system. URL: `https://fedorahosted.org/koji/`, 2009. (Last visited: April 2009).

[12] R. Brewer. CherryPy. URL: `http://www.cherrypy.org/`, 2009. (Last visited: April 2009).

[13] I. Burnstein. *Practical Software Testing: A Process-oriented Approach*. Springer Inc., New York, NY, USA, 2003.

[14] Z. Cerza. Dogtail. URL: `http://people.redhat.com/zcerza/dogtail/`. (Last visited: December 2008).

[15] G. de Menten, J. LaCour, and D. Haus. Elixir. URL: http://elixir.ematia.de/trac/wiki, 2009. (Last visited: April 2009).

[16] B. Dillenseger. Test management and load testing with Salome-TMF and CLIF is a Load Injection. URL: http://www.ow2.org/xwiki/bin/download/Events/OW2QuarterlyMeetingGrenobleFrance/SalomeCLIF.pdf, May 2008. (Last visited: December 2008).

[17] E. Dustin. Automated Testing Tool Evaluation Matrix. URL: http://www.stickyminds.com/s.asp?F=S3100_ART_2, 2001. (Last visited: November 2008).

[18] E. Dustin, J. Rashka, and J. Paul. *Automated software testing: introduction, management, and performance*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[19] FedoraProject. EPEL. URL: http://fedoraproject.org/wiki/EPEL, 2009. (Last visited: April 2009).

[20] M. Foord and N. Larosa. Reading and Writing Config Files. URL: http://www.voidspace.org.uk/python/configobj.html, 2009. (Last visited: April 2009).

[21] E. Hoxworth and M. Khan. Best Practices for Automated Enterprise Testing . URL: www.dell.com/downloads/global/power/ps1q07-20060364-Hoxworth.pdf, 2007. (Last visited: December 2008).

[22] IBM Software Group. Practical Approaches to End-to-End Automation with STAF and STAX. URL: http://staf.sourceforge.net/educ2x/practical/Practical.ppt. (Last visited: December 2008).

[23] Illes, Herrmann, Paech, and Ruckert. Criteria for Software Testing Tool Evaluation. A Task Oriented View. In *3rd World Congress for Software Quality 2005*, Munich, Germany, September 2005.

[24] C. Kaner, J. Bach, and B. Pettichord. *Lessons Learned in Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[25] Š. Květoňová. Lecture: Plánování a sledování softwarového projektu. URL: https://wis.fit.vutbr.cz/FIT/st/course-files-st.php/course/AIS-IT/lectures/2_R izeni_SW_procesu.pdf, 2008.

[26] M. Lutz. *Learning Python, 3rd edition*. O'Reilly, 2007.

[27] J. Lyndsay. A Guide to Software Test Tools. URL: http://www.ism.co.at/analyses/Program_Testing/Test_Tools.html, July 2004. (Last visited: November 2008).

[28] S. Martini. Pyinotify: monitor filesystem events with Python under Linux. URL: http://pyinotify.sourceforge.net/, 2009. (Last visited: April 2009).

[29] D. J. Mosley. *The Handbook of Testing MIS Application Software: Methods, Techniques, and Tools for Assuring Quality Through Testing.* Prentice-Hall/Yourdon Press Computing Series, 1993.

[30] N. Norwitz. PyChecker, a python source code checking tool. URL: `http://pychecker.sourceforge.net/`, 2009. (Last visited: April 2009).

[31] R. Patton. *Software Testing.* Sams, Indianapolis, IN, USA, 2000.

[32] J. Poelstra. RHTS Quick Start. URL: `https://fedorahosted.org/beaker/wiki/QuickRhts`, 2009. (Last visited: April 2009).

[33] C. Pooter and T. Cory. CAST Tools: An Evaluation and Comparison. URL: `http://www.dpu.se/blott_e.html`. (Last visited: December 2008).

[34] S. Purcell. PyUnit - the standard unit testing framework for Python. URL: `http://pyunit.sourceforge.net/`, 2005. (Last visited: April 2009).

[35] C. Rankin. The Software Testing Automation Framework. URL: `http://www.research.ibm.com/journal/sj/411/rankin.pdf`, 2002. (Last visited: December 2008).

[36] F. Schwarz. Identity Management. URL: `http://docs.turbogears.org/1.0/Identity`, 2009. (Last visited: April 2009).

[37] Open source project. Beaker. URL: `https://fedorahosted.org/beaker/`. (Last visited: November 2008).

[38] Open source project. Beaker - Fedora Project. URL: `http://fedoraproject.org/wiki/QA/Beaker`. (Last visited: November 2008).

[39] Open source project. Smolt. URL: `http://smolts.org/`, 2009. (Last visited: April 2009).

[40] SpikeSource. Test Results Publication Interface. URL: `http://developer.spikesource.com/wiki/index.php/Test_Results_Publication_Interface`, 2006. (Last visited: December 2008).

[41] S. Thenault. Pylint, Python code static checker. URL: `http://www.logilab.org/857`, 2009. (Last visited: April 2009).

[42] C. Vail. Automated Testing Tool Evaluation. URL: `www.vcaa.com/tools/wsipc-automatedtestingtoolevaluation.pdf`, 2002. (Last visited: December 2008).

[43] S. Vidal. Yum Package Manager. URL: `http://yum.baseurl.org/`, 2009. (Last visited: April 2009).

[44] R. Wang. Accerciser. URL: `http://live.gnome.org/Accerciser`, 2008. (Last visited: December 2008).

[45] Wikipedia. Manual Testing. URL: `http://en.wikipedia.org/wiki/Manual_testing`. (Last visited: November 2008).

[46] Wikipedia. Software Testing. URL:
http://en.wikipedia.org/wiki/Software_testing. (Last visited: November 2008).

[47] C. Zwerschke. Scheduling Tasks with TurboGears. URL:
http://docs.turbogears.org/1.0/Scheduler, 2009. (Last visited: April 2009).

Appendix A

Criteria for Software Testing Tool Evaluation

Main tasks and corresponding roles involved in a test process [23]:

ID	Tasks	Role(s)
A	Test planning and monitoring	Test manager
B	Designing Test Cases	Test designer
C	Constructing Test Cases	Test automator, test designer
D	Executing test cases	Tester
E	Capturing and comparing test results	Tester
F	Reporting test results	Tester
G	Tracking Software problem reports/defects	Tester, test manager, developer
H	Managing the test ware	Test configuration manager, test administrator

Quality criteria [23]

Q1 Functionality suitability, accurateness, interoperability, compliance, security (1-5)

Q2 Reliability maturity, fault tolerance, recoverability (6-8)

Q3 Usability understandability, learnability, operability (9-11)

Q4 Efficiency time behavior, resource behavior (12-13)

Q5 Maintainability analyzability, changeability, stability, testability (14-17)

Q6 Portability adaptability, installability, conformance, replaceability (18-21)

Q7 General vendor qualifications 22 maturity of the vendor, market share, financial stability

Q8 Vendor support 23 warranty, maintenance and upgrade policy; 24 regularity of upgrades, defect list with each release; 25 compatibility of upgrades with previous releases; 26 e-mail support, phone support, user groups; 27 availability of training, recommended training time, price

Q9 Licensing and pricing 28 open source or commercial; 29 licensing used, rigidity (floating node-locking license); 30 price consistent with estimated price range; 31 price consistent with comparable vendor products

Functional criteria [23]

A: Test planning and monitoring. Test tool provides support for:

1. customization of the organizational test process
2. particular programming paradigms and/or languages, operating systems, browser, network configuration
3. application specific characteristics, which require specific testing techniques
4. testing special application domain (e.g. avionics, automotive, etc.)
5. planning of the test process (scheduling, project tracking, risk management)
6. monitoring test activities

- by tracking of the estimated and actual time/test case
- by providing coverage metrics to measure the progress of testing activities
- by providing metrics from different sources (e.g. requirements, test cases)

6. integration with other tools

B: Designing Test Cases. Test tool provides support for:

7. designing test cases for the required test level (unit, integration, system)
8. selecting the test techniques
9. defining test conditions derived from the defined test techniques
10. defining templates for structuring the information specifying test cases
11. generation of logical test cases from semi-formal models
12. generation of logical test cases from formal specifications (e.g. Z)
13. generation/derivation of test data layout
14. optimizing the test case set
15. designing test cases to test quality criteria of the application
16. restricting the test case set (e.g. ranking by prioritization of the test cases, risks assigned to test cases) in case of deadline constraints

C: Constructing Test Cases. Test tool provides support for:

17. editing test scripts
18. developing of test code conforming with accepted software engineering practices
19. capturing of executable test cases
20. generation of concrete test cases from (semi-)formal models
21. generation of (in)valid test data
22. generation of stubs, test drivers, mock objects,
23. simulating missing faulty system components

D: Executing Test Cases. Test tool provides support for:

24. setting-up and clearing-down of the test environment/pre condition and respectively the post conditions for a set of test cases
25. roll-back to initial in case of unexpected errors
26. execution of captured, captured & edited or manually implemented test cases for

functional testing.

27. execution of captured test cases for testing quality criteria
28. stopping and continuation of the execution of a suspended test case

E: Capturing and comparing test results. Test tool provides support for:

29. logging information on executed test cases
30. comparison facilities between specified and actual outcomes

F: Reporting test results. Test tool provides support for:

31. aggregation of logged test results
32. customizable, role specific amount of information

G: Tracking Software problem reports/defects. Test tool provides support for:

33. specifying problem reports/defects by using predefined templates
34. generating entries for recorded defects
35. prioritizing defects
36. tracking change requests/defects and their current status
37. generating statistical information
38. for regression testing

H: Managing the test ware. Test tool provides support for:

39. management of the test ware
40. traceability between the elements of the test ware
41. by tracing modifications on a test object and communicating changes
42. the maintenance of the test data, of the test cases
43. for automated tests to be (re)used for regression testing/in other projects
44. snapshot facilities (by freeze a special state of the test ware)

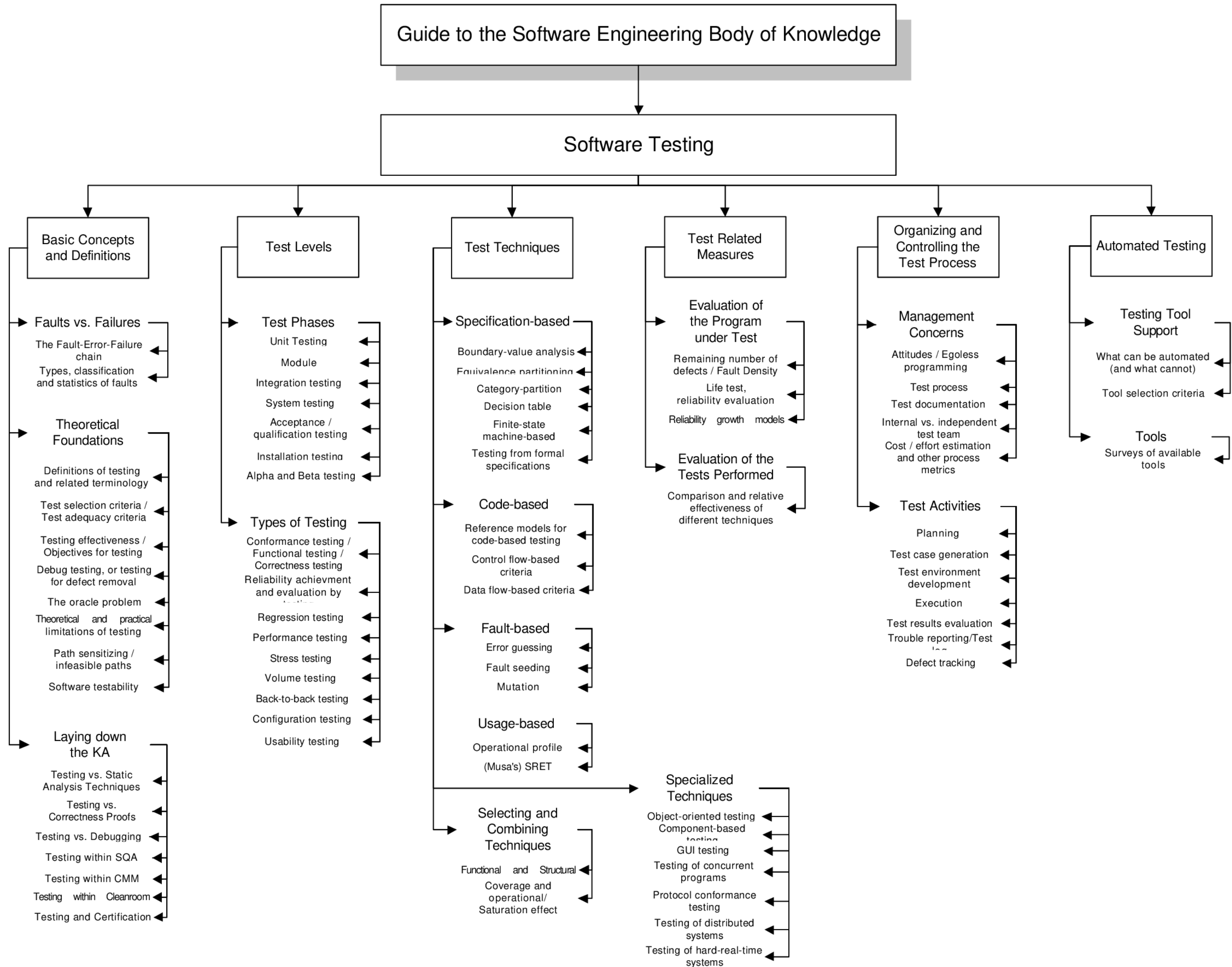


Figure A.1: Breakdown of a software testing as presented in [9]

Appendix B

Quick Start Instructions

Attached CD contains SRPM and RPM packages. Using SRPM package custom RPM packages can be build for Fedora 9, Fedora 10, and RHEL 5 distributions. They are already prebuild and available on the CD.

B.1 Server

In this section are presented simple installation instructions on how to start the server. By default the configuration file for server is set to be used standalone running CherryPy without the need to use Apache (httpd). As a default database MySQL has to be configured. Involved configuration files:
`/etc/late/server/late.cfg`
`/etc/late/server/app.cfg`

1. Install the server RPM package:
   `$ sudo rpm -Uvh late-server-0.1.0-1.fc9.noarch.rpm`
2. Setup MySQL server.
3. Create database and add user:

   ```
   mysql> create database latedb;
   mysql> grant all privileges on latedb.* to late@"localhost" \
   identified by 'pass4late';
   ```

4. Create tables:

   ```
   $ mysql -u late -p latedb < \
   /usr/share/late/server/latedb_create_mysql.sql
   ```

5. Modify SELinux into Permissive mode: `$ sudo /usr/sbin/setenforce 0`
6. Start the server using provided SysV init script:

   ```
   $ sudo /sbin/service late-server start
   ```

7. Open web browser and go to URL `http://localhost:8080`
8. Place some RHTS tests into the repository directory. This is `/mnt/late/test_repo` by default. They should appear in the test repository available via WebUI after few seconds. Examples of RHTS tests are available from the source RPM package or on the attached CD.
9. For troubleshooting start with server log file: `/var/log/late-server.log`

B.2 Client

No configuration files are involved in using `late-client` tool.

1. On the same (or remote) system as server, install the client RPM package:
   `$ sudo rpm -Uvh late-client-0.1.0-1.fc9.noarch.rpm`
2. Create yum repository config file using `repo` command, and follow instruction:

```
$ late-client repo -s localhost:8080

# Save this into file /etc/yum.repos.d/late-test.repo
[late-test-repo]
name=LATE RHTS test repository
baseurl=http://localhost:8080/yum/
enabled=0
gpgcheck=0


Usage examples on how to query the repository with yum
-------------------------------------------------------

List all tests in the repository:
  $ yum list available --disablerepo=* --enablerepo=late-test-repo

Search for test (containing) basic-pass:
  $ yum search --disablerepo=* --enablerepo=late-test-repo example

Display additional info about the tets package:
  $ yum info --disablerepo=* --enablerepo=late-test-repo   <your-test>
```

3. You should be able to submit test jobs to the server using `submit` command.
4. Here is an example of response listing all available tests in the repository:

```
$ yum list available --disablerepo=* --enablerepo=test-repo
Loaded plugins: refresh-packagekit
late-test-repo                                       |  951 B     00:00
late-test-repo/primary                               | 1.4 kB     00:00
```

```
late-test-repo                                                     7/7
Available Packages
tmp-examples-late-Sanity-basic-fail.noarch           1.1-0  test-repo
tmp-examples-late-Sanity-basic-pass.noarch           1.1-0  test-repo
tmp-examples-late-Sanity-subtest-basic-fail.noarch   1.1-0  test-repo
tmp-examples-late-Sanity-subtest-basic-pass.noarch   1.1-0  test-repo
tmp-examples-late-Sanity-subtest-basic-ppff.noarch   1.1-0  test-repo
tmp-examples-late-Sanity-watchdog-fail.noarch        1.1-0  test-repo
tmp-examples-late-Sanity-watchdog-pass.noarch        1.1-0  test-repo
```

B.3 Worker

By default the worker daemon is (pre) configured to be started quickly locally at the system providing server. Involved configuration files:
`/etc/late/worker/conf/worker.cfg`
`/etc/late/worker/conf/log.cfg`

1. Install the worker RPM package:
   `$ sudo rpm -Uvh late-worker-0.1.0-1.fc9.noarch.rpm`
2. Edit worker configuration file `/etc/late/worker/conf/worker.cfg` and set all required values (marked with FIXME in a comment above them). Pay proper attention to server names and URLs. By default it is set to `localhost:8080` and should be ready to use with local server.
3. Check everything is set properly by registering the worker system to the server (localhost:8080): `$ sudo late-worker -r`
4. Try to run the daemon in foreground using command line tool. Issues can be investigated using log files located in files matching pattern `/var/log/late-worker*`.
   `$ sudo late-worker`
5. Finally start the daemon using SysV init script, it should start sending heartbeats to the server and status should change to READY after few seconds.
   `$ sudo /sbin/service late-worker start`

Appendix C

Additional and Custom Evaluation Criteria

Supplementary and custom evaluation criteria specific to desired test tool application domain should be specified, after pre-selection of the test tools. These will help to narrow down the final list and select the most appropriate tool no matter which evaluation strategy has been used for pre-selection. This topic is too specific to custom requirements and therefore is not discussed here in detail.

Example of possible additional criteria:

- Support for tests written in particular programming languages (e.g., C, Bash, Python, Perl, Java).
- Installation automation.
- Support for variety of different system architectures (e.g., x86_64, IA64, PPC, S390).
- Support for variety of different OS and platforms (e.g., RHEL 5, FreeBSD, Solaris, HP-UX, AIX, Windows, Mac OS X, etc).