Wolfgang Staudacher, BSc

Development of Test Strategies for Integrated Test Automation of Web Applications for Datacenter Monitoring

MASTER'S THESIS

to achieve the university degree of

Master of Science

Master's degree programme: Telematics

submitted to

Graz University of Technology

Supervisor

Ass. -Prof. Dipl.-Ing. Dr. techn. Christian Steger

Institute for Technical Informatics

Graz, February 2015

AFFIDAVIT

Kurzfassung

Die Entwicklung von komplexen Software-Systemen für mobile Anwendungen und Web-Plattformen erfordert für die Qualitätssicherung eine Anwendungs- und Plattform übergreifende Durchführung von funktionalen Tests. Die manuelle Durchführung von funktionalen Tests für verteilte Systeme ist nicht nur zeitaufwändig, sondern aufgrund von fehlenden Methoden und Werkzeugen auch wenig automatisierbar. Eine Vielzahl von Testmöglichkeiten wird sowohl für mobile Anwendungen als auch für Web-Applikationen angeboten. Einzelne Software-Module werden verschiedenen Arten von Tests unterzogen. Diese Masterarbeit beschäftigt sich mit dem Design und der Implementierung von Teststrategien, die es ermöglichen funktionale, verteilte Tests automatisiert im Zusammenspiel von verschiedenen Anwendungen auf unterschiedlichen Plattformen durchzuführen.

Abstract

The development of complex software systems for mobile applications and web platforms requires for quality assurance as well as the implementation of application- and cross-platform functional tests. The manual execution of functional tests for mobile or web applications as well as for distributed systems is not only time consuming, but due to lack of methods and tools little automated. A variety of test options is offered both for mobile applications as well as web applications. Individual software modules are to be tested through various types of tests. This project aims to develop a concept for test strategies, to enable functional, distributed and automated testing in the interaction of different applications performing on different platforms. This thesis deals with the design and implementation of test strategies, enabling functional, distributed automated testing the interaction of various applications perform on different platforms.

Acknowledgement

This master thesis would not have been possible without the guidance and the help of several individuals who in one way or another contributed and extended their valuable assistance in the preparation and completion of this study. First and foremost I would like to thank my family, especially my parents and my sister for the support throughout all my studies. They have supported me since my birth in all situations and gave me the financial means and the assistance to finish this study. I also express my special gratitude to all my friends and fellow students who helped me throughout the years. I also have to thank the whole Institute for Technical Informatics for the opportunity of this master thesis.

Contents

List of Figures

List of Tables

List of Abbreviations

XML	Extensible Markup Language
REST	Representational state transfer
RESTful	Representational state transfer
SQL	Structured Query Language
REST	REpresentational State Transfer
SOAP	Simple Object Access Protocol
PUT	Parameterized Unit Test
GUI	Graphic User Interface
JML	Java Modeling Language
BPEL	Business Process Execution Language
BERT	Behavioral Regression Testing
IMEI	International Mobile Station Equipment Identity
URI	Uniform Resource Identifiers
JPF	Java Pathfinder
UML	Unified Modeling Language
MMS	Multimedia Messaging Service
IDE	Integrated Development Environment
ADT	Android Developer Tools
OS	Operating System
JSON	JavaScript Object Notation
AJAX	Asynchronous JavaScript + XML
PAPILLON	Profiling and Auditing of Power Information in Large and Local Organizational Networks
UNITE	Understanding Non-functional Intentions via Testing and Experimentation

XMPP	Extensible Messaging and Presence Protocol
ZATS	ZK Automated Testing Suite
ZTL	ZK Testing Language
ARC	Android Runtime for Chrome
APK	Android Application Package

Chapter 1

Introduction

In the last couple of years the usage and data transfer of people using the internet has increased rapidly. This massive and growing amount of data needs to be stored somewhere. For fail safe systems as well as for performance improvement, data is often stored reluctant. To cope with this massive amount of data, datacenters are built all around the planet. A datacenter is a facility to house computer systems and associated components like storage systems and telecommunication systems. The amount of energy, used for these systems as well as for their cooling is increasing as well. By managing the cooling and processing power better, lots of energy, as well as the connected costs could be saved. To manage such a system, the visualization of this consumed energy is needed. The development of complex software systems for mobile applications and web platforms requires for quality assurance as well as the implementation of application- and cross-platform functional tests. The manual execution of functional tests for mobile or web applications as well as for distributed systems is not only time consuming, but due to lack of methods and tools little automated. A variety of test options is offered both for mobile applications as well as web applications.

Individual software modules are to be tested through various types of tests. This project aims to develop a concept for test strategies, to enable functional, distributed and automated testing in the interaction of different applications performing on different platforms.

1.1 Motivation

By mastering the test complexity of the system, the functional reliability should be improved with a reasonable temporal and organizational effort. An important aspect is the temporal synchronization in the simultaneous test execution on different devices. By building an integrated test environment (with the use of freely available test tools), the concept

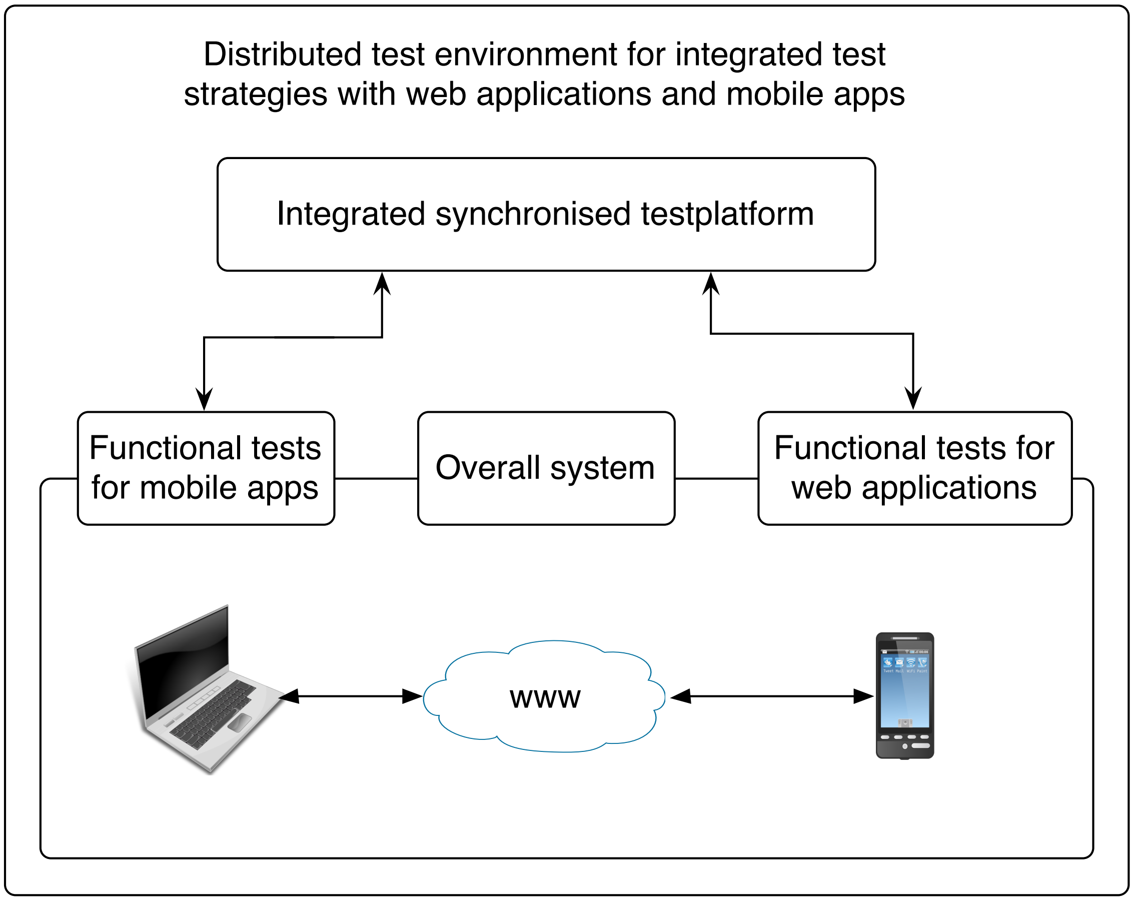

Distributed test environment for integrated test strategies with web applications and mobile apps
Integrated synchronised testplatform
Functional tests for mobile apps
Overall system
Functional tests for web applications
www

Figure 1.1: System overview

is to be evaluated and the possible potentials for a subsequent product development should be considered.

1.2 Outline

This master thesis is about the design and the implementation of visualization tools for energy monitoring of data centers. These visualization tools consist of a mobile app for handheld devices and a web application for a more detailed visualization and configuration. Furthermore the design and the implementation of a test environment for this system is necessary. A software that exceeds a certain size, can never provide 100 % covering. The goal is, to test as much as possible of the code with as much as possible data. The following requirements should be met from the developed visualization software:

- Intuitive usage of the web application
- Visualization of the consumed energy of the components in near-real time

- Visualization of different configuration menus depending on the user
- Notification services for user and superuser
- Automatic updates and dynamic handling

The mobile app will be an app for Android OS. For a near-real time alerting notifications, a cloud-based push technology was integrated. For the data transfer between the database and the Android app, a web server service was implemented as well. The Android app should meet the following requirements:

- Intuitive usage of the Android app
- Energy efficient
- Near-real time alerting notifications
- Visualization of the data in a derived way of the web application

To ensure a proper and lasting functionality of the system, an automated test suite will be implemented. This test suite should check the software for as much as possible errors, and achieve a high code coverage. For an easy to use test suite, it should be automated as much as possible and should not exceed a reasonable execution time. To ensure that, special strategies will be developed. The test suite should meet the following requirements:

- Testing of the single components
- Automated testing of the whole system
- Reasonable execution time
- A proper visualization of the coverage

1.3 Structure

This work is partitioned in the following chapters. In **Chapter 2**, the related work is listed. First, some important techniques for testing web applications, like unit testing, integration testing and regression testing are described. Then some existing work on mobile applications testing is discussed, followed by a detailed description of related work on testing distributed systems.

The design of the system is described in **Chapter 3**, starting with a detailed description of the original PAPILLON technology. This section is followed by a description of useful development tools and techniques. Afterwards, a system overview with its requirements is given.

Chapter 4 shows the implementation of the system, the development tools and technologies used for the implementation and the testing strategies for single components and the whole system.

In **Chapter 5**, the results of the master thesis are discussed. In detail the different testing methods with their results as well as the fulfilled requirements to the Android app and the web application are shown.

Chapter 6 gives a conclusion as well as an outlook to possible future development and makes the completion oft the thesis.

Additional information to the system and on how to execute everything can be found in the Appendix.

Chapter 2

Related Work

Testing software is a absolute necessity when it comes to developing software. Without it, there is no way of making sure the software works as the user expects to. It helps discovering defects, helps learning about the reliability of the software and showing the it works within the requirements. Testing of software should be done from the beginning of the development process. It is even quite common, to think about how to test a method whilst writing it and adapting it that way, so it would be easier to test. This chapter deals with existing work on the testing of Software. Since there are already a large number of test strategies for individual components in software systems, as well as overall applications, distributed systems and cross-platform systems, the following sections and subsections list basic strategies, the possible combinations and potential of existing systems and their achieved results. The structure of the chapter will be divided in the testing for the desktop and web applications, mobile devices and for distributed systems.

2.1 Desktop and Web Application Testing

The most important tests in both the traditional as well as the standardized testing, with respect for desktop or web applications are hereby unit tests, integration tests and regression tests. These tests are quite distinctive and can be made at the beginning of the development of software. One approach for developing software is the division into stages. The V-model, which is shown in figure 2.1 is a popular way to carry this out.

A unit test is used to check the functional items (modules) of programs for correct functionality. Since algorithms at the module level usually have a limited complexity and are activated via clearly defined interfaces, they can largely be fully tested with relatively few test cases. This is a prerequisite for the following test level called integration tests. In order to be able to align the test cases on the integrated cooperation of various functional

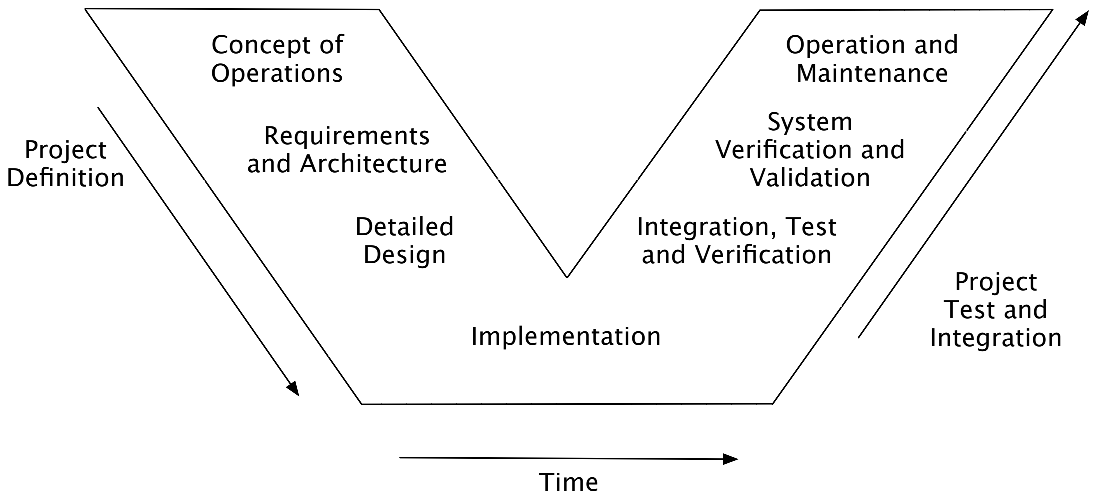

Concept of Operations
Requirements and Architecture
Detailed Design
Implementation
Integration, Test and Verification
System Verification and Validation
Operation and Maintenance
Project Definition
Project Test and Integration
Time

Figure 2.1: V-model

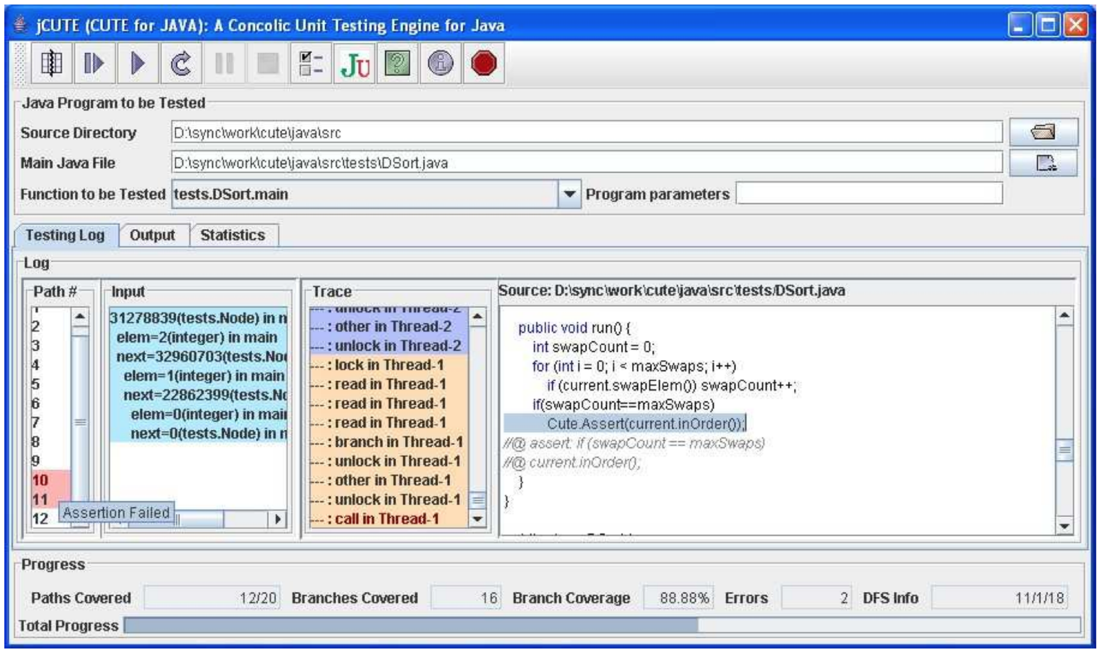
parts or the entire application. Furthermore, unit tests are usually made at the beginning or used relatively early in the development process in order to eliminate possible errors and their sources in an early stage, to minimize the cost and time consumption.

Integration tests indicate a coordinated series of individual tests, which serve to test various interdependent components of a complex system in interaction with each other. For each dependency between two components of a system, a test scenario is defined which is to ensure the correct operation of the exchange of data on the common interface.

A regression test refers to the repetition of test cases for modifications in already tested parts of the software, to ensure that no new errors occur(regression). Such modifications occur regularly e.g. due to the maintenance, modification and correction of software. Regression testing is a dynamic test technique. Due to the nature of repetition and frequency of this, an automation would be useful as well as time- and cost-saving. The test cases must be specified and provided with a desired result, which is compared with the actual result of a test case. A direct reference to the results of previous test runs is usually not made.

2.1.1 Unit Testing

In 2002, unit testing was first attempted to be automated by having used so-called unit-test-oracles. This was achieved by using formal specifications (for example: conditions prior to performing and thereafter) instead of test code. Through this abstraction it was easier to read and adapt everything[11]. Furthermore, it was easier to find specification errors and extend these formal specifications to other programs. In 2006, a tool for systematic and automated execution of tests in sequential C programs and parallel Java programs was presented in [1]. Figure 2.2 shows the GUI of jCute.


jCUTE (CUTE for JAVA): A Concolic Unit Testing Engine for Java
Testing Log
Output
Statistics
Assertion Failed

Java Program to be Tested
Source Directory D:\sync\work\cute\java\src
Main Java File D:\sync\work\cute\java\src\tests\DSort.java
Function to be Tested tests.DSort.main
Program parameters

Log

Path #: 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12

Input
```
31278839(tests.Node) in n
  elem=2(integer) in main
  next=32960703(tests.No
    elem=1(integer) in main
    next=22862399(tests.N
      elem=0(integer) in mai
      next=0(tests.Node) in n
```

Trace
```
--- : other in Thread-2
--- : unlock in Thread-2
--- : lock in Thread-1
--- : read in Thread-1
--- : read in Thread-1
--- : read in Thread-1
--- : branch in Thread-1
--- : unlock in Thread-1
--- : other in Thread-1
--- : unlock in Thread-1
--- : call in Thread-1
```

Source: D:\sync\work\cute\java\src\tests/DSort.java
```java
  public void run() {
    int swapCount = 0;
    for (int i = 0; i < maxSwaps; i++)
      if (current.swapElem()) swapCount++;
    if(swapCount==maxSwaps)
      Cute.Assert(current.inOrder());
//@ assert: if (swapCount == maxSwaps)
//@ current.inOrder();
  }
}
```

Progress
Paths Covered 12/20 Branches Covered 16 Branch Coverage 88.88% Errors 2 DFS Info 11/1/18
Total Progress

Figure 2.2: GUI of jCute from [1]

An extension for unit testing was introduced in 2011, which allowed the enclosure of the parameters. These so-called PUTs (Parameterized Unit Tests) help the behavior of the methods perform for all test arguments and to simplify and speed, the error detection [12].

Once systems exceed a certain size, the execution times and the composite costs of unit testing become a problem. By testing with randomized values for the parameters in unit testing, the speed of such tests can be increased and the execution time can be minimized without affecting the results widely.

One such tool is called Nighthawk [13], which used randomized parameters for the test cases using a genetic algorithm and minimizing the execution time of the tests below 10% of the original execution time.

Unit testing in distributed systems represent a more complex matter, since certain problems and errors such as end-to-end response time, security and reliability occur only when the system is tested as a whole.

To carry out these tests anyway, a system called UNITE was introduced[14]. It was developed for the purpose of performing unit testing of non-functional properties such as compatibility, performance and security in a distributed system. UNITE shows that by means of a distributed system with three components, so that unit tests can be simplified and can be used by distributed systems in early stages of software development.

Another useful tool to perform automated unit tests for Java programs is the JUnit testing framework, which is already integrated in Eclipse, Ant and Maven. It offers several practical features and provides the results in either green, which means the test has been completed successfully, or red what either means, that the test failed, or the result to be calculated is not the expected result. Eclipse also offers a third(blue) option for test cases that are supposed to fail. JUnit is based on concepts originally developed under SUnit for Smalltalk. It has been ported to many other programming languages such as C, C ++, C #, Fortran, Microsoft .NET, PHP and Python and is usually referred to with the generic term xUnit.

A good way of using the JUnit framework is shown in [15]. This paper shows a possible and good way of using it together with the JML.

2.1.2 Integration Testing

A model-based approach for the generation of integration test cases is presented in [16] . Modeling languages and model-driven software development aim to raise the abstraction of the code already in the early stages of development and create similarly as in [11] a model-based testing environment.

Integration testing of object- and aspect-orientated programming is carried out in [17] using a configurable integration depth. A graphical representation of the control flow and the data flow between units is made to identify possible sources of errors.

Software products from a software family usually have similar features and similar problems and sources of errors. In order to extract such features and to reduce the testing effort for the entire development process, [18] presents a so-called "compositional symbolic execution" technique, which analyzes the data flow between the features using a dependency graph and setting up a tree for the possible interactions. Using this tree to minimize the number of interactions which are really needed during execution, and thus time and effort is saved without affecting the results.

Another approach, that does the computing of the interprocedural data dependencies before testing is introduced in[15]. This way the testing tool can use existing path-selection techniques which are based on the data flow for interprocedural testing and thus reducing the necessary data for it.

Integration testing for distributed systems show similar problems as unit testing in [14]. To get this under control a detailed description of the problems that can occur with the usage of protocols for communication within a distributed system and how to act against them is presented in [19]. Furthermore, the paper shows problems with missing codes and a lack of test automation of a project that is constantly evolving. For this purpose, a test environment called SymNet is presented, which was designed for exactly such simulations. The main challenge here is the synchronization of the distributed system, the detection of "false positives" and avoiding redundant states.

For the generation of automated test cases for embedded systems with real-time interactions, [2] makes usage of global variables, which are used for the synchronization of memory accesses, interrupt service routines and monitoring the system. The access patterns in the execution of the software is recorded and used for analyzing the errors. Depending on the usage patterns, different test sequences are applied, which are supposed to control the dependancies and data flow between the modules. Figure 2.3 shows the different steps of the generation of the test model.

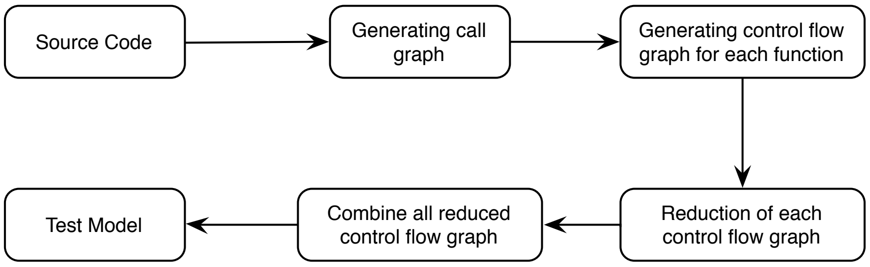

Source Code
Generating call graph
Generating control flow graph for each function
Reduction of each control flow graph
Combine all reduced control flow graph
Test Model

Figure 2.3: Different steps of test model generation reconstructed from [2]

Evolutionary approaches are used in [20] for an automated test data generation for coupling based integration testing using a genetic algorithm to use the coupling path as an input with different sub paths.

The combination of architecture-driven, model-based testing principles and regression-based testing strategies is presented and discussed in [21]. To do this efficiently and comprehensively the authors use a Delta-oriented approach to parallelize the tests of the individual components in the various stages of the development. This is furthermore automated for reusing the tests for different systems. Figure 2.4 shows how this has been achieved and that it was limited to integration testing and unit testing.

2.1.3 Regression Testing

Since regression testing is used when changes in software have been carried out, it can often occur, that in a modified or longer existing software testing frameworks are too large or too expensive. [22] suggests how to take care of that by either minimizing particular tests, selecting only a part of them or prioritize the order in that the tests are performed. This saves time and money without the error detection rate to change much. Figure 2.5 shows how test case selection for regression testing usually is carried out.

An automatic selection of test cases for regression testing is also described in [23]. This is shown using two different versions of a BPEL (Business Process Execution Language) services. The approach is to analyze the paths of the associated BPEL flowchart and use the differences for the test case generation.

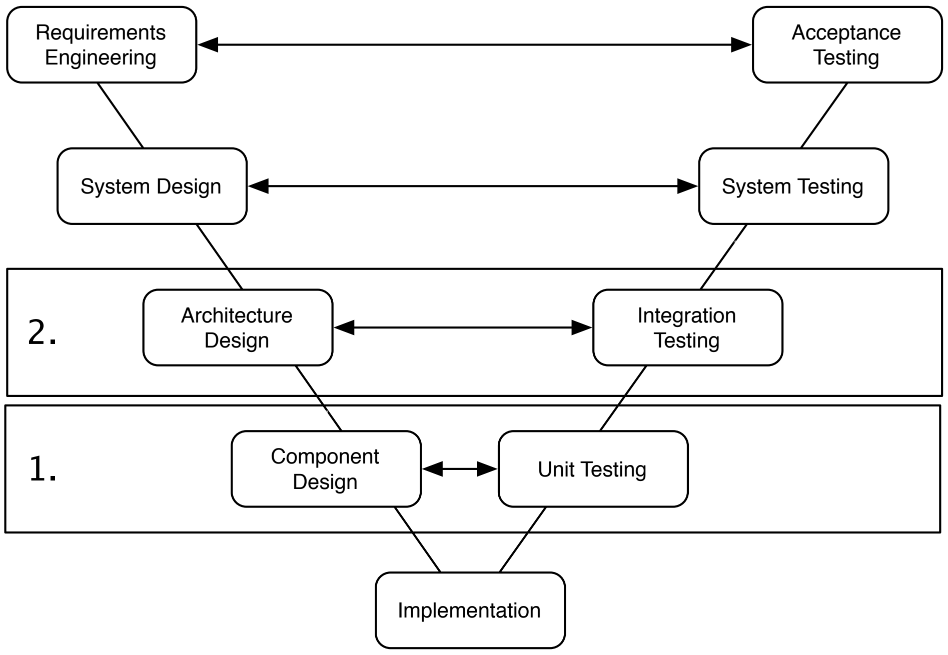


Figure 2.4: Parallelized V-model reconstructed from [3]

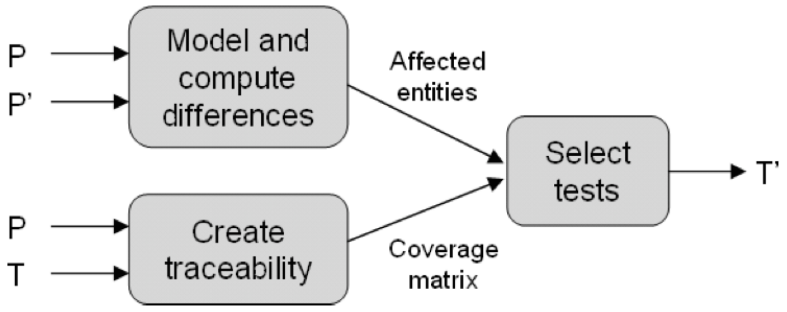


Figure 2.5: Usual regression testing from [3]

Prioritization of test cases and their targets is presented in [3]. Changes in software are often of varying extend. A risk analysis to determine to what extent the change will affect the entire system can be used to reduce the number of tests and test cases, ignoring the changes with low risk or moved backwards in the test order.

Another approach called Behavioral Regression Testing or short BERT is presented in [24]. BERT analyzes two versions of a program, finds the differences in the versions by producing a large number of test inputs, sends them through both versions and lists the differences in the results and more detailed information for the developer. By this approach, only a fraction of traditional regression test considerations are left for the developer.

Most automated testing frameworks for regression tests are limited to testing changes in the code. But what happens in case of changes that are not in the code? This is discussed in more detail and a technique is presented, that deals with changes in software that aren't in the code, but rather changes such as a corresponding database schema or a configuration file in [25]. This technique computes the dependencies between test cases and the external data and uses this to automate the testing. Figure 2.6 shows how this is carried out.

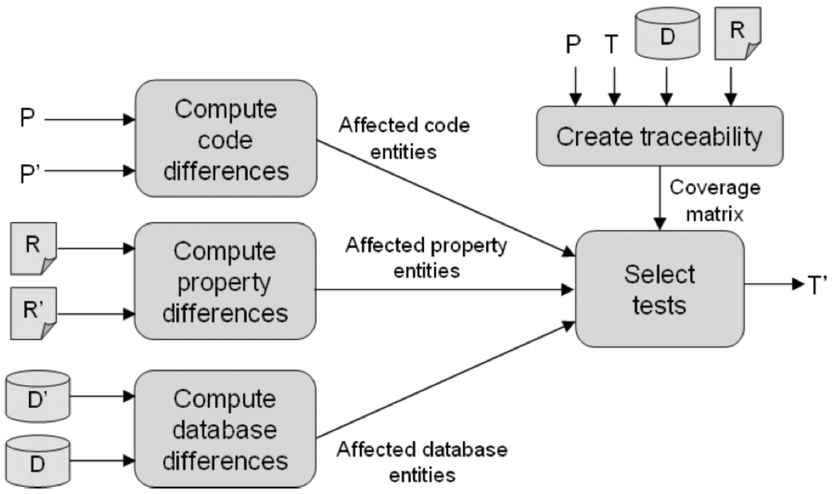

P
T
D
R
Create traceability
Coverage matrix
P
P'
Compute code differences
Affected code entities
R
R'
Compute property differences
Affected property entities
Select tests
T'
D'
D
Compute database differences
Affected database entities

Figure 2.6: Regression testing in the presence of non-code-changes from [25]

A good strategy and practical recommendations on test selection for integration testing and regression testing to reduce the testing expenses is described in [26]. This approach was compared to the retest-all-strategy and the paper showed, that the strategy was able to discover all errors by executing only 35 % of the total number of test cases.

2.2 Mobile Application Testing

In recent years the use of mobile devices as well as the performance of these devices and the complexity of their software has grown rapidly. This growth is offset by an increased need of safety and error prevention, as well as the desire to implement that at low cost.

As manual testing is not only very time consuming and expensive, the aim here is of course to automate testing. Various surveys and published statistics[27] indicate, that mobile applications have not been tested sufficiently. Many of the techniques described in the following chapters to find errors and problems took for testing finished apps and hardly any could withstand the testing. Of course it is an almost impossible task to test apps on all possible errors, minimizing the number of these failure possibilities is tried instead.

Automated testing of mobile applications provides an additional challenge, since there is a high number of different mobile devices, which in turn have different hardware-specific characteristics. There are also many different mobile operating systems such as Android, iOS, BlackBerry, Windows Mobile and Symbian to name a few. Furthermore, there are several versions of these operating systems, which in turn have different limitations and capabilities. The three main test groups for mobile applications include testing for functionality, usability and consistency.

2.2.1 Usability

Because testing of usability is performed to promote the acceptance of apps and thus is very dependent on human interactions and reactions, it can be difficult to automate it. The author of [28] shows a lot of testing-methods for the usability of several android apps and their results with several different test users. To begin with, the evaluation of usability testing is automated in [29], which saves at least some of the time and cost. Methods for recording, analysis and presentation of such Usability tests are shown in [30], [31] and [32]. These are mainly for web applications, that can be transferred to mobile applications, but they are less conclusive then.

A toolkit for the same tasks on mobile basis is introduced in [4], specifically for Android, but it refers (as well as the aforementioned) on statistical information that can be used to find usability problems, but can not provide 100 % results. Figure 2.7 shows how the tester or developer specifies a baseline for the usage of the app, the sequence in which a test user does it and the differences in it.

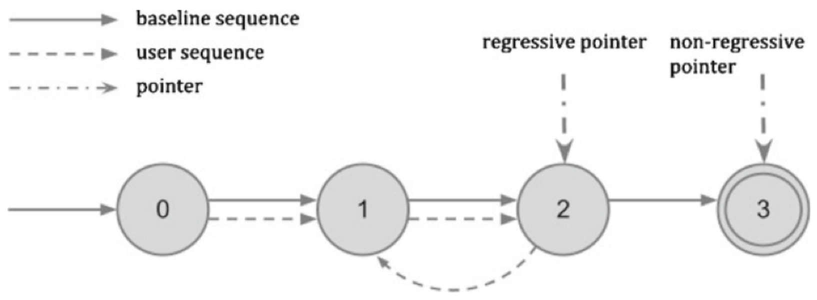


Figure 2.7: Usability model from [4]

Some mobile applications have problems with the design, specifically with the identification of controls or similar. These so called ”low discoverability issues”, which present a major problem for User Interfaces with ”touch” -based controls are discussed in [33]. Here the authors demonstrate a working concept for the automated detection of such problems.

2.2.2 Functionality and Consistency

For functionality and consistency there are more approaches and solutions including already functioning ones. There are some free available tools to test Android apps.

Monkey Runner [34], for example, is a tool that provides the ability to perform automated black-box testing for Android applications. This is achieved through different Python scripts that simulate user interactions, take screenshots of the user interface and stores them for later inspection or comparisons. Although the name and function are quite similar, the U/I Application Exerciser Monkey [35], or short just Monkey is a different tool for testing Android applications, running on a different level of the Android OS and offers less interaction for the tester.

Monkey creates a stream of pseudo-random user events such as clicking, typing, swiping and various events at the system level to test the app. It can be used both on an Android simulator and on a real device.

Robotium [36] is another open source project for testing of Android applications. It is based on a gray-box principle, which means only little knowledge of the application code is required. Stable test cases can be developed with little cost, fast execution times can be achieved, and various external libraries can be integrated without affecting the test results.

Java Pathfinder [37] is an automated system to verify the executable Java bytecode. It was developed by NASA and in 2005 it was made to an open source project. It can use a model of the Java bytecode to compare the individual states of the program with the current state of the program to find race conditions and deadlocks. The Java Pathfinder is extended to test Android applications in [38]. This was accomplished by having the Android application framework modeled on Java Pathfinder using different extensions, so that Android application can run on the Java Virtual Machine.

This approach falls into the group of white-box testing. Something that is also made in [39] in a similar form. Here a technique is presented which analyzes the code of Android-specific apps and is able to produce a large number of test cases from it. This test cases are then executed parallel on different emulated Android systems on a cloud to find errors and optimize the execution time.

The opposite of white-box testing is black-box testing, which means the tester does not know what is going on inside the system, or has no access to the code and it is therefore even more difficult to carry out an automation. Something that has been proposed and implemented in [40]. This is achieved by the following steps. First, user interactions in a mobile application are recorded and are stored as events. For these events are then various test scripts generated to test this and other mobile applications. A similar approach is called GUI ripping, in which user inputs are fully simulated and the system checks for errors and inconsistencies. There are a lot of these event-based testing methods for Android apps[41],[42],[5]. They usually have a workflow shown in Figure 2.8 or similar to it.

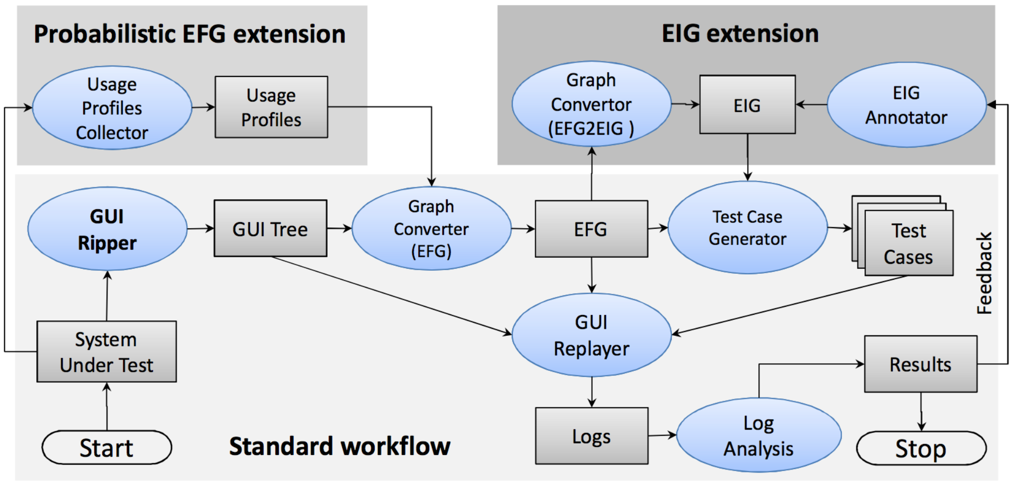


Figure 2.8: Standard workflow of GUI ripping from [5]

A method is presented in [27], which automatically scans the complete GUI software, by opening all possible activities and their widgets, properties and values are extracted. This extracted information is then analyzed by the test designer and used to automatically generate test cases for it. To transfer the whole thing on an Android device, of course, brings new challenges with it. A GUI Ripper is presented in [43], which tests Android apps on crashes where minimal tester interactions are necessary. Only the configuration of the test process and the creation of an Android Virtual devices are carried out manually. The GUI ripping and testing themselves, which are both time and cost consuming operations are performed automatically. The same authors develop this work even further in [6]. The newer version, called AndroidRipper, was compared with the free available test software Monkey [35] and it was shown that AndroidRipper found more errors in about the same time. AndroidRipper works in three stages which can be seen in more detail in Figure 2.9. The first stage is the only one in which manual interactions are required, and the rest is automated.

iCrawler [44] is a similar program for iOS applications. It identifies ”unique states” to generate a directed graph of the states. These ”unique states” are used to avoid duplicate conditions and thus to minimize the testing effort. However iCrawler is still associated with some manual settings and afford.

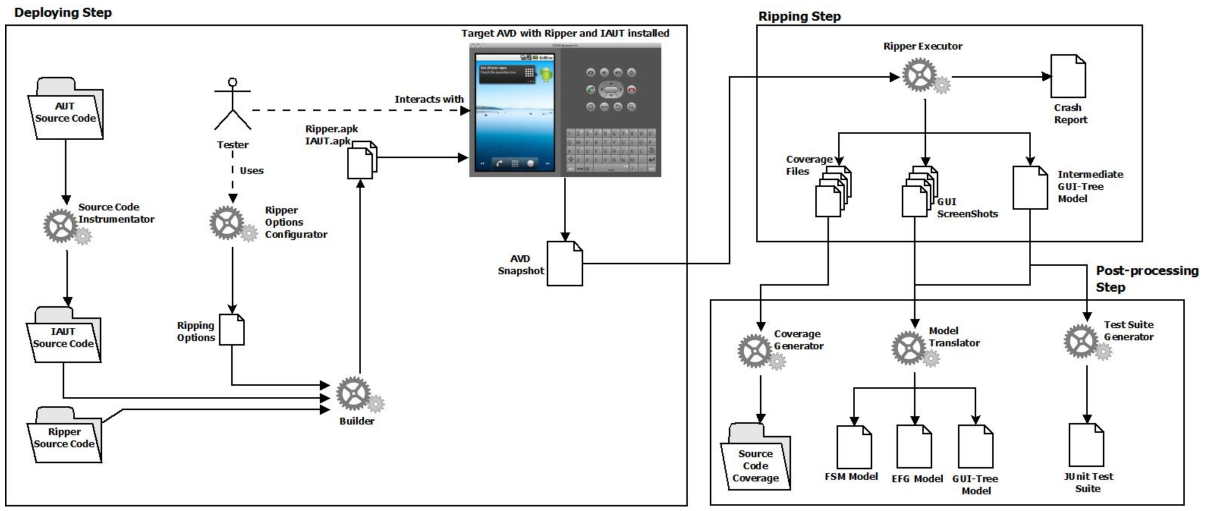


Figure 2.9: Different stages of AndroidRipper from [6]

2.2.3 Resource Management

A primary problem with mobile devices are the limited resources that are available for an application. Wrong managing of these resources can lead to slowing down of the application, to crash and to an unpleasant user experience. A way to visualize memory leaks in Java is shown in [45]. This was achieved by recording the average duration of temporary objects and filter out those that exist for a longer time and where it is referenced from.

Leakbot, for example, is an automated tool to find memory leaks[7]. It achieves this through the combination of three techniques: First, classify the data structure for possibilities of a memory leak as small as possible. After that the expected changes to these regions at risk are compared to the actual changes. In this way endangered storage areas are monitored more closely and errors are found faster. Figure 2.10 shows the fringe. It is the boundary between older and recently created objects and is used to find memory leaks.

LeakChaser is another tool for finding memory leaks[46]. However, it is possible for the designer to intervene in the test procedure to achieve better results. LeakChaser operates in 3 different levels, with little manual interactions of the developer on the highest and advanced settings to improve the search in the lowest.

A new method is proposed in [47] to find memory leaks in Android applications. The principle is to classify GUI events into various ”neutral” cycles. Each cycle as soon as it is completed, will have no influence on the total use of resources of the application at the end. Such a ”neutral” cycle could be the repeated rotation of the screen or the minimizing and reopening of the app.

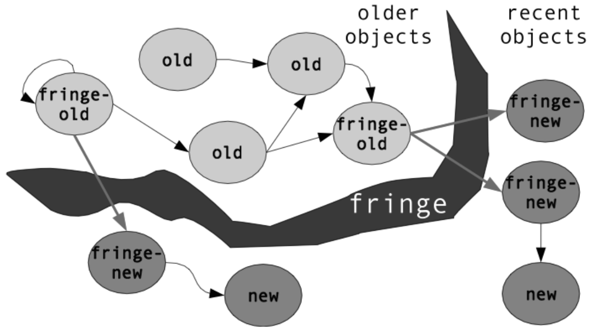
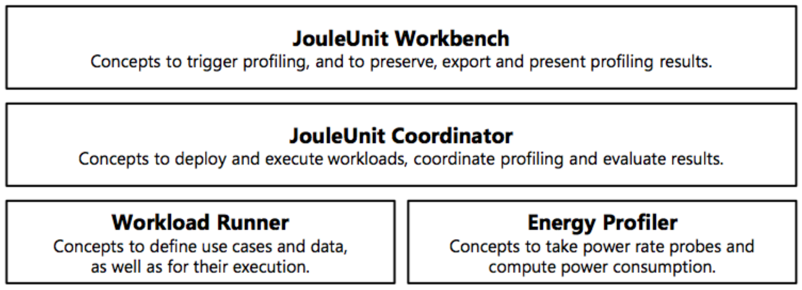


Figure 2.10: The fringe: a boundary between older and recently created objects from [7]

Although testing software through the energy consumption is a well-known approach[48], it has been little used for mobile devices so far. An app is tested based on the energy it consumes in [49]. Starting with an already known power model a specific one for a mobile device was developed. It monitors the kernel and various hardware components and thus the current energy consumption can be compared with an expected value to find errors during the execution of the app.

The JouleUnit framework is an energy profiling framework for Android and NAO robot applications[8]. It consists of 4 components as shown in Figure 2.11. For the workload runner the JUnit framework is used to provide test data definition and capabilities to setup and tear down individual workload runs. The energy profiler provides capabilities for sampling the energy consumption. The coordination layer is used for synchronization of the workload runner and the energy profiler. The JouleUnit Workbench integrates the framework into the Eclipse IDE supporting editors, wizards and views.



JouleUnit Workbench
Concepts to trigger profiling, and to preserve, export and present profiling results.

JouleUnit Coordinator
Concepts to deploy and execute workloads, coordinate profiling and evaluate results.

Workload Runner
Concepts to define use cases and data, as well as for their execution.

Energy Profiler
Concepts to take power rate probes and compute power consumption.

Figure 2.11: The 4 different components of JouleUnit from [8]

2.2.4 Non-automated and Beta-testing

As an alternative to automated testing of mobile devices there is, e.g. the opportunity to register on the Google Play Developer Console [50] apps for Alpha or Beta testing and have it tested by a group of test users and getting feedback for it. This should give insight in design and usability issues as well as functionality and consistency. For iOS, there is also the opportunity to let Apps be Beta tested. One of these ways is called iBetatest [51]. While only a Google+ account is required for Google Play Developer Console, there is a fee for iBetatest. The benefits of this form of testing is the fact, that user get beta versions of software before they are available on the market and developers get advice, error messages and suggestions for improvement.

Similar to that is Applause. Applause is a company, which perform a so-called "in-the-wild testing" principle used for Android, iOS, desktop- and Web applications[52]. Main difference from beta testing is, that there are specially trained testers who know what to look for, know the target group the application is directed to and they work under conditions outside the laboratory.

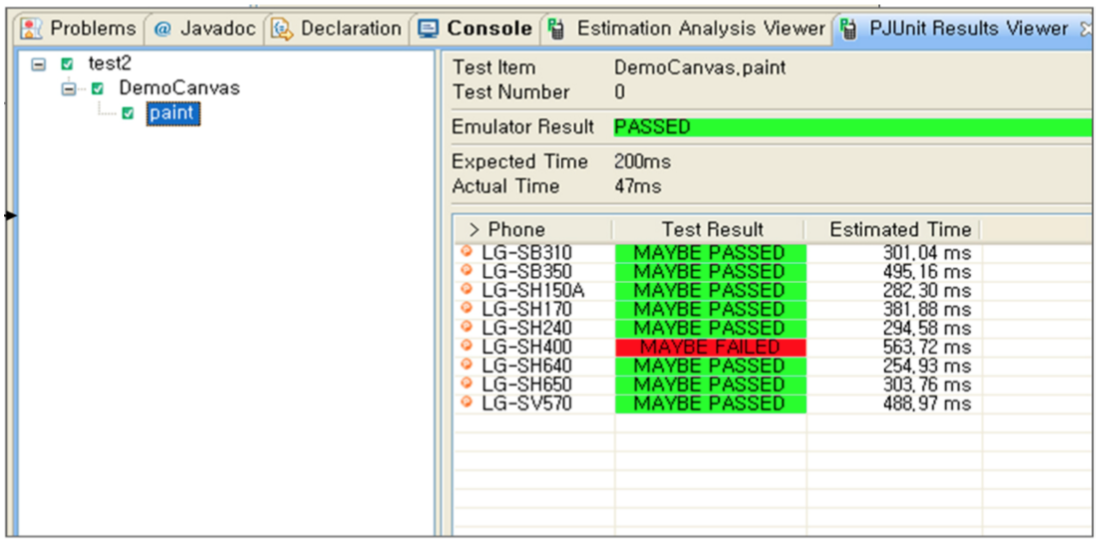
Furthermore for iOS there is, e.g. Test Flight [53] and for Blackberry Fledge, a Black-Berry Smartphone Simulator [54]. Test Flight is a free test environment for iOS applications that will be used to test beta versions. Until recently, Android applications could also be beta tested. However, Burstly purchased the makers of Flight Test and Flight Test will also be integrated in iOS 8 to test self-written apps to simplify this for developers. Fledge is less complex than Flight Test and able to perform simple test scenarios in the form of interrupts. As an alternative to them Test Fairy can be used for Android[55].

2.2.5 Unit Testing

Unit test have proven themselves for desktop systems and can of course also be used for mobile applications. In[Towards Unit Testing of User Interface Code for Android Mobile Applications] is a special focus on unit testing, GUI testing and GUI testing through unit testing. Also included is a description of several problems with unit testing the GUI and a list possible solutions to them.

An evaluation of unit testing techniques on mobile platforms as well as white-box testing and black-box testing is shown in [56]. Furthermore the importance of unit testing and advanced tools for automated testing and automated test case generation is stressed out. How to unit test a controller of an application with commonly used techniques for unit testing is presented in [57]. The paper also shows how this can be achieved by using a standard Eclipse environment and the convenience of an unit testing framework.

A method for performance testing called MObilePBDB, an abbreviation of Mobile Performance Benchmark Database is introduced in [9]. Furthermore, a performance testing tool called PJUnit is introduced, which has been developed as an Eclipse plugin and is shown in Figure 2.12. The main characteristics of this tool are the creation of automated test cases, the execution of these automated test cases, the estimation of performance on an emulator and a performance prediction on real devices.



Test Item	DemoCanvas.paint
Test Number	0
Emulator Result	PASSED
Expected Time	200ms
Actual Time	47ms

Phone	Test Result	Estimated Time
LG-SB310	MAYBE PASSED	301.04 ms
LG-SB350	MAYBE PASSED	495.16 ms
LG-SH150A	MAYBE PASSED	282.30 ms
LG-SH170	MAYBE PASSED	381.88 ms
LG-SH240	MAYBE PASSED	294.58 ms
LG-SH400	MAYBE FAILED	563.72 ms
LG-SH640	MAYBE PASSED	254.93 ms
LG-SH650	MAYBE PASSED	303.76 ms
LG-SV570	MAYBE PASSED	488.97 ms

Figure 2.12: PJUnit plugin for Eclipse from [9]

2.3 Distributed Systems

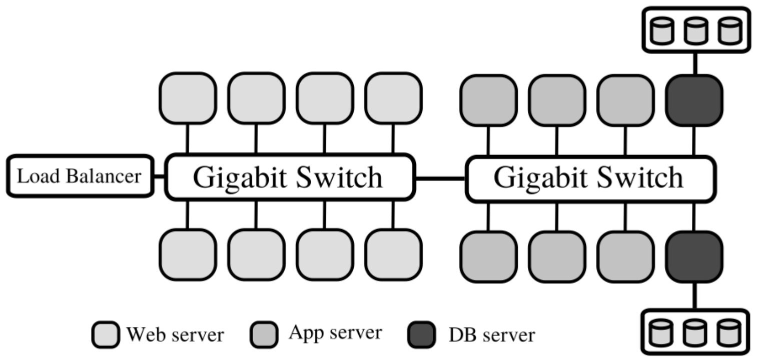
Testing distributed systems is usually a tricky business, because one of the most important things is to ensure that a connection exists, and that this does not affect the testing process. In a distributed system with a wireless connection that is even more difficult than for a wired connection, because there might be more fault possibilities.

Other issues that must be considered when testing distributed systems is the complexity of system itself. If it is a client / server system it could perform the test on the server and test a large part of the system based on the communication between server and clients.

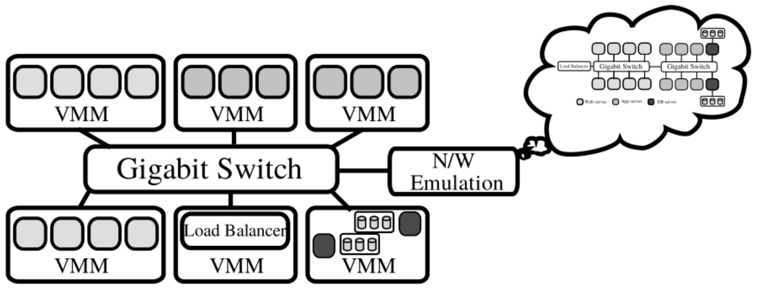
A more complex system, with several different independent components that share a common communication network, can lead to problems such as non-deterministic states by race conditions between the components or parallel actions performed in the system. Another problem is the test case generation to ensure that the entire system is covered. A test and monitoring tool (TMT) that covers that problems and the authors implement a version of it in C++ for UNIX or Windows NT platforms and in Java for distributed systems with Java is presented in [58].

A further problem with testing distributed systems is, that most testing is performed only on small parts or smaller numbers of it, but there are also distributed systems, which connect to thousands of machines. To test such a system DieCast was developed[10].

DieCast emulates the complete system with virtual machines with different parameters and tries to simulate all the problems that can happen in such a system and repeat these test runs several times to find inconsistencies in the system.



(a) Original System.



(b) Test System.

Figure 2.13: Scaling a network service to the DieCast infrastructure from [10]

A stress test for a distributed system is presented in [59]. This stress test uses the UML model of the distributed system, extended with timing informations and an analysis of the control flow in the sequence diagram in order to find problems or errors in communication of the system.

Ranorex [60] is a company offering many different options for testing. These range from the generation of automated tests for desktop, web and mobile applications as well as for recording various test scenarios and to create use of bug reports. It supports a large number of development systems and has been ported to many programming languages.

An object-oriented framework to test distributed systems is proposed in [61]. This system should be able to test distributed systems rapidly and adaptively by using scenario specification techniques and verification patterns.

A survey of testing web services shows a lot of problems and possible solutions for testing web services like SOAP or REST, data generation of test cases and several approaches for specific problems[62].

2.4 Summary

For this Chapter, a lot of related work has been analyzed. It was shown, that unit testing, integration testing and regression testing have proven themselves for desktop systems and web applications. Unit-testing is working on mobile devices as well and has proven itself for that purpose, too. Furthermore is GUI-Ripping a popular and well-working approach for testing mobile applications. Several test environments and testing techniques for distributed systems are shown to be satisfying as well.

Chapter 3

Design

This Section is about the design of the different components for the distributed system as well as the resulting challenges. Furthermore it will give insight to design of the test suite for the single components as well as the whole distributed system together. The first section introduces the company Stratergia and their energy monitoring technology PAPILLON followed by a short description of Green IT. Moreover it gives a detailed overview of the architecture and the data flow of the original system works and which benefits arise from the use of PAPILLON. It will also include a description of the data, that the original PAPILLON generates and uses for visualization.

Afterwards, section 3.2 will give an outlook of what the extended PAPILLON will look like and how it will work. In the following section the design of the web application is introduced as well as the necessary requirements. In section 3.4 will be shown, how the watchdog server is supposed to work, which is responsible for calculating the additional information and alerting users in case of a certain threshold being reached. Following is the description of the planned testing of the proposed watchdog routine. Section 3.5 will show the design of the proposed web service which will be used for transferring the requested data to the Android device. Afterwards, section 3.6 describes the function of the proposed Android app in detail. The design of the testing suite of the web application is discussed in section 3.7. Following sections are about the testing of the watchdog service, the testing of the server for the app data and the testing of the Android app. In section 3.11 different methods and approaches of testing parts of the system together as well as the whole system are discussed. The final chapter will give a short summary about the proposed design.

3.1 Stratergia

Stratergia is an Irish company based in Dublin. It is the global leader in non-intrusive, meter-less, datacenter power measurement and management systems. The main goal of this is to reduce the consuming energy of a datacenter as well as to reduce the carbon footprint of it. The Stratergia PAPILLON technology allows energy savings and costs averaging 38% in the daily operation and management of datacenters.

3.1.1 Green IT

Green IT (or Green ICT) refers to efforts to use information and communication technology (ICT) throughout their life cycle to make the environment and resources. This includes the optimization of resource consumption during manufacturing, operation and disposal of equipment. The carbon footprint is a measure of the total amount of carbon dioxide emissions (measured in CO_2) and methane (CH_4), which is caused directly and indirectly by an activity or arises about the life stages of a product.

Datacenters are facilities with a very high energy consumption, which is estimated to be around 100 to 200 times higher than normal facilities. Furthermore it is estimated, that datacenters account for about 1.1 % to 1.5 % of the worlds total energy use. Since the number of datacenters is increasing all the time and the energy consumption for the components is increasing as well, it is a good start for reducing the carbon footprint.

3.1.2 PAPILLON

PAPILLON (Profiling and Auditing of Power Information in Large and Local Organizational Networks) is a Master-Client-System that estimates the consumed power of a server(=client) through a virtual power meter with an accuracy of 2%, so it doesn't need any additional hardware.

3.1.3 PAPILLON Architecture and Dataflow

The clients(or agents) read the three stats(CPU, network IO and hard disk IO) from the operating system and send them to the master. According to the power model of the component the master calculates the actual energy consumption and stores in a database for further reference. The clients then sleep for 60 seconds, wakes up and repeats this process. Figure 3.1 and figure 3.2 show the architecture of the PAPILLON system as well as the dataflow of it.

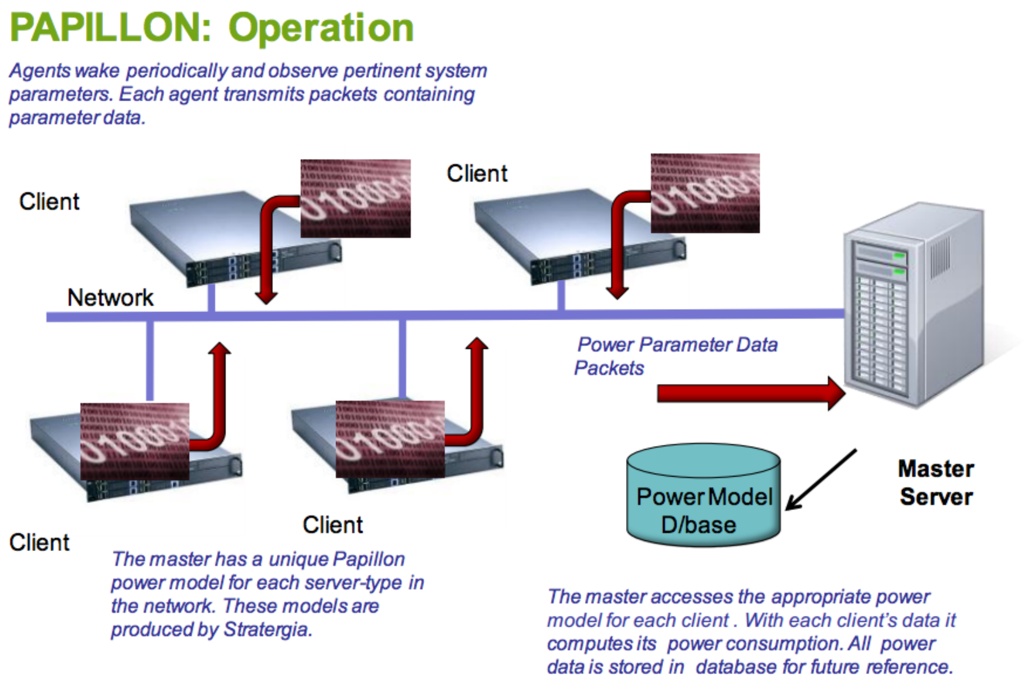

PAPILLON: Operation
Agents wake periodically and observe pertinent system parameters. Each agent transmits packets containing parameter data.
Client
Client
Network
Power Parameter Data Packets
Client
Client
Power Model D/base
Master Server
The master has a unique Papillon power model for each server-type in the network. These models are produced by Stratergia.
The master accesses the appropriate power model for each client . With each client's data it computes its power consumption. All power data is stored in database for future reference.

Figure 3.1: PAPILLON Architecture[1]

Benefits of using PAPILLON

PAPILLON opens a choice of many energy saving actions in a datacenter:

- A virtual power meter on every server
- Power measurements/metrics for effective management
- Optimized rack space
- TCO analysis
- Accurate power provisioning
- Cost center budgeting

[1]Source: `http://www.stratergia.com`
[2]Source: `http://www.stratergia.com/pappilon`

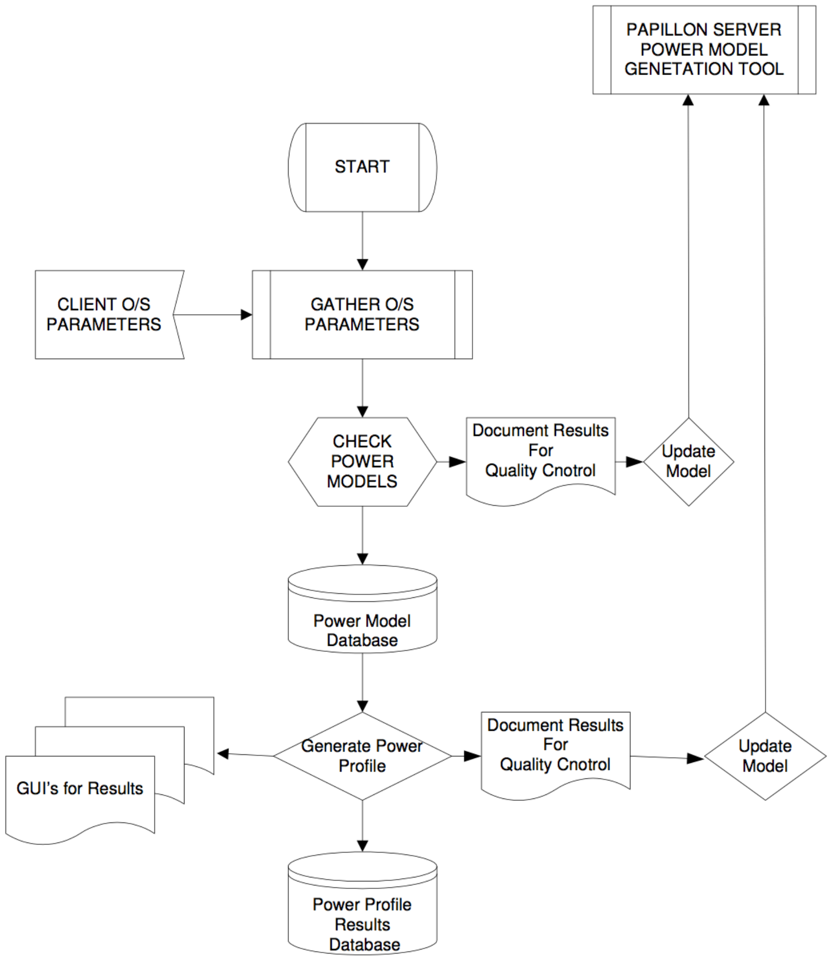


Figure 3.2: PAPILLON Dataflow[2]

- Directed Cooling
- Greener operations and Strategy

3.1.4 PAPILLON Database Schema

There are clients installed on each server. These clients send the estimated power consumption to the PAPILLON server who stores them for further use in a database. PAPILLON uses the hardware layout to group them together, e.g.: server(=hosts). The database schema of that hardware layout is shown in figure 3.3. The original version of PAPILLON is only able to display the hosts and three apps of each host which used the highest amount of energy. For better visualization and comparison this will be extended, so that each component can be watched separately.

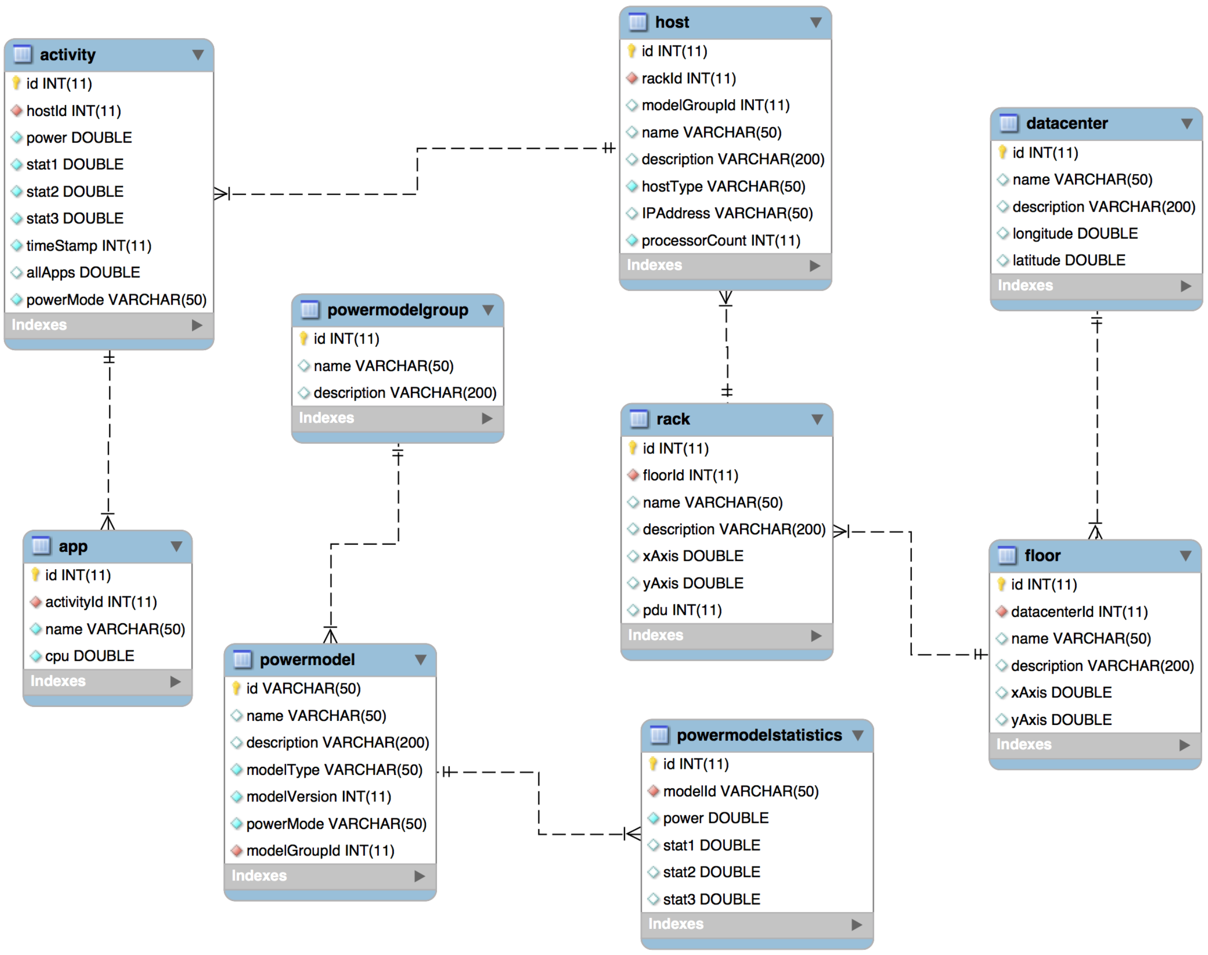


Figure 3.3: PAPILLON database schema

As shown in figure 3.3, to work out a proper working visualization and virtualization of a datacenter, PAPILLON split up and grouped the components in a meaningful way. So a datacenter is divided into different stories, which are labeled floors. Each floor is filled with racks and each rack consists of several servers which are labeled as hosts. The energy consumption of a host is given by an activity and for each activity are the three highest energy consuming apps stored. The following lists the important parts of this database schema.

Datacenter

The table 3.1 holds information for the datacenters. The name of the datacenter, the description as well as the longitude and the latitude will be shown in the visualization of datacenters.

No.	Field	Type	Unit	Description
1	id	integer	-	primary key, auto increment
2	name	varchar	-	name of the datacenter
3	description	varchar	-	description of the datacenter
4	longitude	double	degree	longitude of the datacenter
5	latitude	double	degree	latitude of the datacenter

Table 3.1: Description of database entity datacenter

Floor

The name of the floor, the name of the corresponding datacenter, the description of the floor and the xAxis and the yAxis will be shown in the visualization of the floors.

No.	Field	Type	Unit	Description
1	id	integer	-	primary key, auto increment
2	datacenterId	integer	-	datacenter id
3	name	varchar	-	name of the floor
4	description	varchar	-	description of the floor
5	xAxis	double	-	location of the floor
6	yAxis	double	-	location of the floor

Table 3.2: Description of database entity floor

Rack

The name of the rack, the name of the corresponding floor, the description of the rack and the xAxis and the yAxis will be shown in the visualization of the racks.

Host

The name of the host, the name of the corresponding rack, the type and the description of the host and its ip address will be shown in the visualization of the hosts. The modelGroupId and the processorCount are used for the calculation of the carbon footprint.

No.	Field	Type	Unit	Description
1	id	integer	-	primary key, auto increment
2	floorId	integer	-	floor id
3	name	varchar	-	name of the rack
4	description	varchar	-	description of the rack
5	xAxis	double	-	location of the rack
6	yAxis	double	-	location of the rack
7	pdu	integer	-	power distribution unit

Table 3.3: Description of database entity rack

No.	Field	Type	Unit	Description
1	id	integer	-	primary key, auto increment
2	rackId	integer	-	rack id
3	modelGroupId	int	-	power model group Id
4	name	varchar	-	name of the host
5	description	varchar	-	description of the host
6	hostType	varchar	-	type of the host
7	IPAdress	varchar	-	IP address of the host
8	processorCount	integer	-	number of processors

Table 3.4: Description of database entity host

Activity

The power of an activity will be used for the visualization of the different hosts for the given timeStamp. Stat1, stat2 and stat3 are the three stats from the client, which are used to calculate the consumed power at that time. The three stats from the OS are the CPU, the network IO and the hard disk IO.

No.	Field	Type	Unit	Description
1	id	integer	-	primary key, auto increment
2	hostId	integer	-	host Id
3	power	double	W	consumed Power of the host
4	stat1	double	-	CPU stat
5	stat2	double	-	network IO stat
6	stat3	double	-	disk IO stat
7	timeStamp	integer	S	timestamp of the measure
8	allApps	double	-	stat for all apps
9	powerMode	varchar	-	mode of powerconsumption

Table 3.5: Description of database entity activity

App

Each activity is basically split into three apps. These three apps are the three highest energy consuming apps and their names will be in a pie chart in the visualization.

No.	Field	Type	Unit	Description
1	id	integer	-	primary key, auto increment
2	activityId	integer	-	activity Id
3	name	varchar	-	name of the app
4	cpu	double	-	CPU Value

Table 3.6: Description of database entity app

Powermodel

A power model is a model of a specific hardware type used to calculate the real energy consumption from the stats retrieved from the OS.

No.	Field	Type	Unit	Description
1	id	integer	-	primary key, auto increment
2	name	varchar	-	name of the powermodel
3	description	varchar	-	description of the powermodel
4	modelType	varchar	-	type of the powermodel
5	modelVersion	integer	-	version of the powermodel
6	powerMode	varchar	-	mode of powerconsumption
7	modelGroupId	integer	-	group Id of the powermodel

Table 3.7: Description of database entity powermodel

Powermodelgroup

This table is used to group power models for further use.

No.	Field	Type	Unit	Description
1	id	integer	-	primary key, auto increment
2	name	varchar	-	name of the powermodelgroup
3	description	varchar	-	description of the powermodelgroup

Table 3.8: Description of database entity powermodelgroup

Powermodelstatistics

The power model statistics are used for generating newer and more precise power models and updating older ones.

No.	Field	Type	Unit	Description
1	id	integer	-	primary key, auto increment
2	modelId	varchar	-	id of the powermodel
3	power	double	W	power consumption
4	stat1	double	-	CPU stat
5	stat2	double	-	network IO stat
6	stat3	double	-	disk IO stat

Table 3.9: Description of database entity powermodelstatistics

3.2 System Overview

The original PAPILLON system is able to gather enough information about the energy consumption and the carbon footprint for datacenters. Since it only allowed visualization of single hosts, which isn't sufficient in bigger datacenters or in a system with more components. To achieve that, the PAPILLON system needs to be extended. This will include an improved visualization, an user and threshold management and a visualization on a mobile device. Figure 3.4 shows the original and the extended PAPILLON and the components that need to be added.

The most important parts will be the web application, which will allow the visualization of the data as well as the management of everything and the watchdog service, which is in charge of calculating and storing the new values and notifying the alerting service.

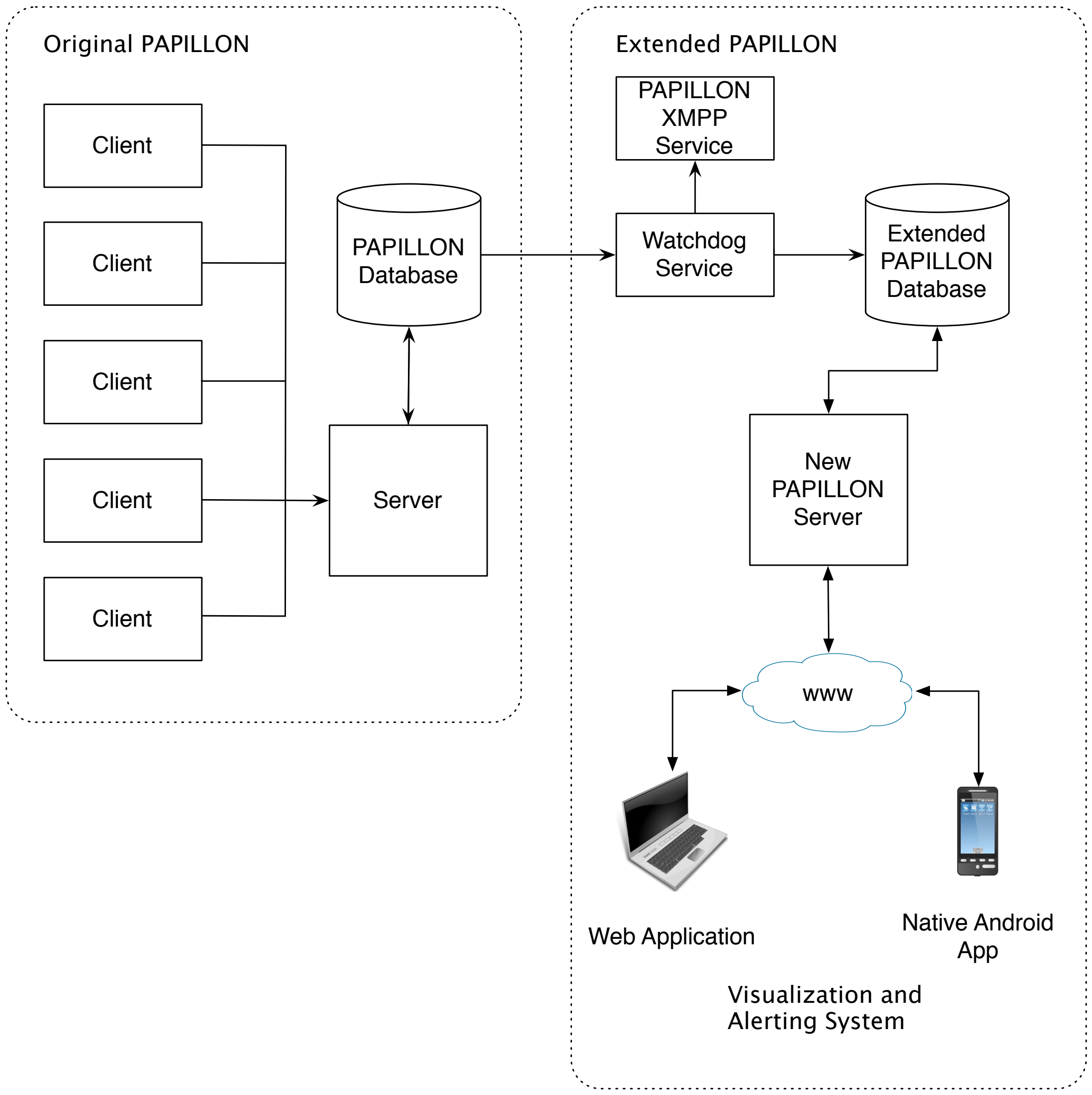


Figure 3.4: System Architecture

3.3 Web Application

This section describes the design of the web application. This includes the requirements of it and a mock up for the visualization page and the setting page. The basic layout of the visualization page will be the same as in the original PAPILLON, but it will include more visualization options like displaying of higher components and will be user specific for his assigned components.

3.3.1 Requirements for the Web Application

The web application must fulfill several requirements to ensure proper behavior of the web application and a pleasant workflow for users and superusers.

Usability

The usability of the web application needs to be easy to handle and self explaining for the end-user.

Reliability

The representation of the data needs to be accurate. And it should always display the correct data and options for the logged in user.

Near-real time

The visualization of the data needs to be with as fast as possible. Since there is an update of data in the database every 60 seconds, the update of the visualization should be synchronized with it.

Dynamic

Since the visualization is depending on the user who is logged in, the pages must be dynamic to ensure proper representation.

Security

In the settings part of the web application, changes to users and alert notifications can be made. The ensure only superusers can do that, preparations must be made.

3.3.2 Visualization

This section describes the design of the visualization page. The user should be able to navigate through the components of the PAPILLON system and select which ones he wants to display. The data should be continuously updated, so that the user can see the changes in energy consumption of the component. Moreover for the energy consumption of the individual hosts, the user should be able to access a pie chart, that can visualize the apps which consumes the most energy. Figure 3.5 shows a mock up of the visualization page which was generated using a tool called Mockup Builder[3].

[3]`http://www.mockupbuilder.com`

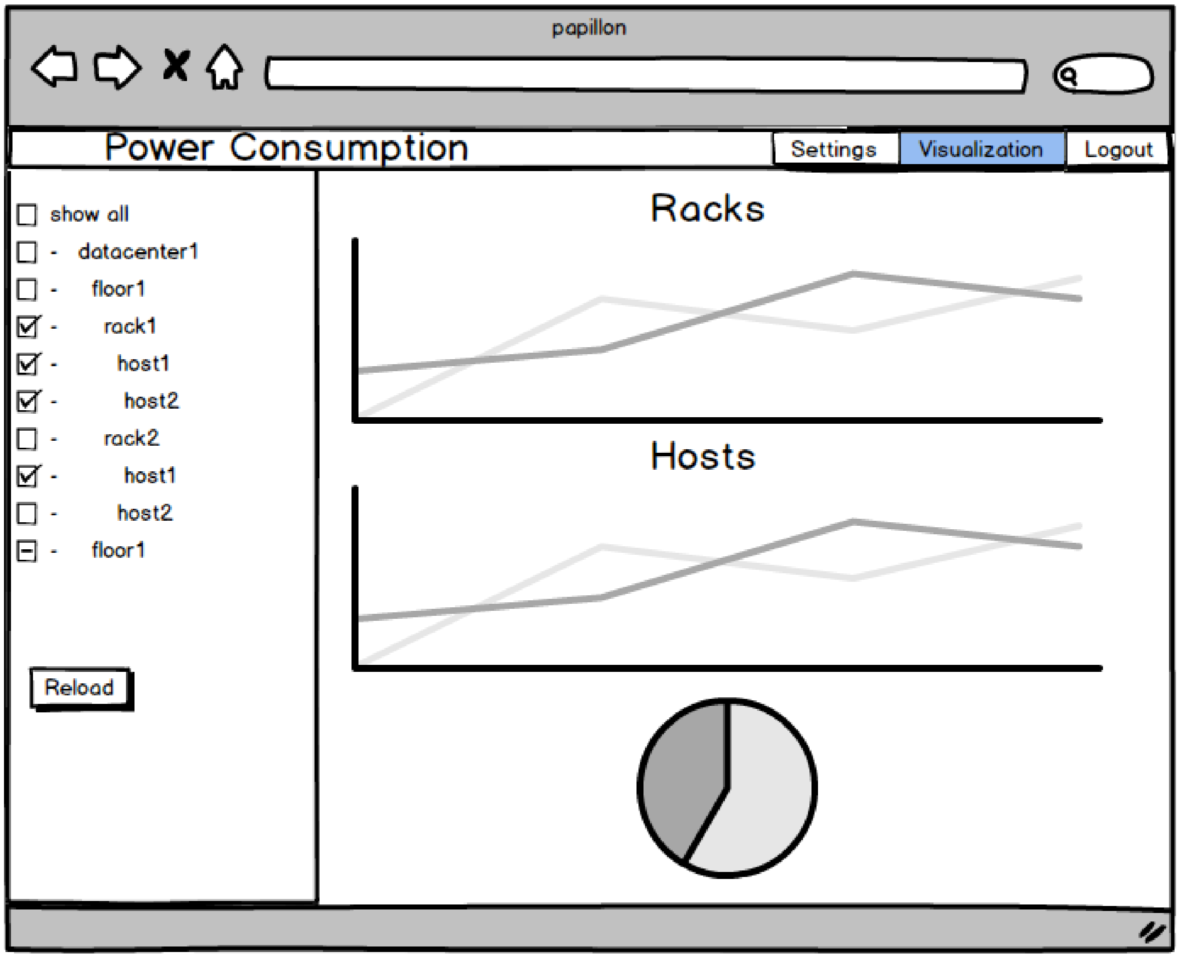


Figure 3.5: Visualization page mock up

3.3.3 Settings

This section describes the design of the settings part. Users should be able to make changes to their own account, like username or password and adding an Android device for visualization on it. In addition to that, each user can access the threshold values used for his components so that he can manage the alert notification service. Figure 3.6 shows a mock up of the settings page. This was also generated with the assistance of Mockup Builder[4].

3.4 Watchdog

The watchdog is one of the most important parts of the extended PAPILLON. In the first step it will take activities from the original PAPILLON database, calculate the power

[4]`http://www.mockupbuilder.com`

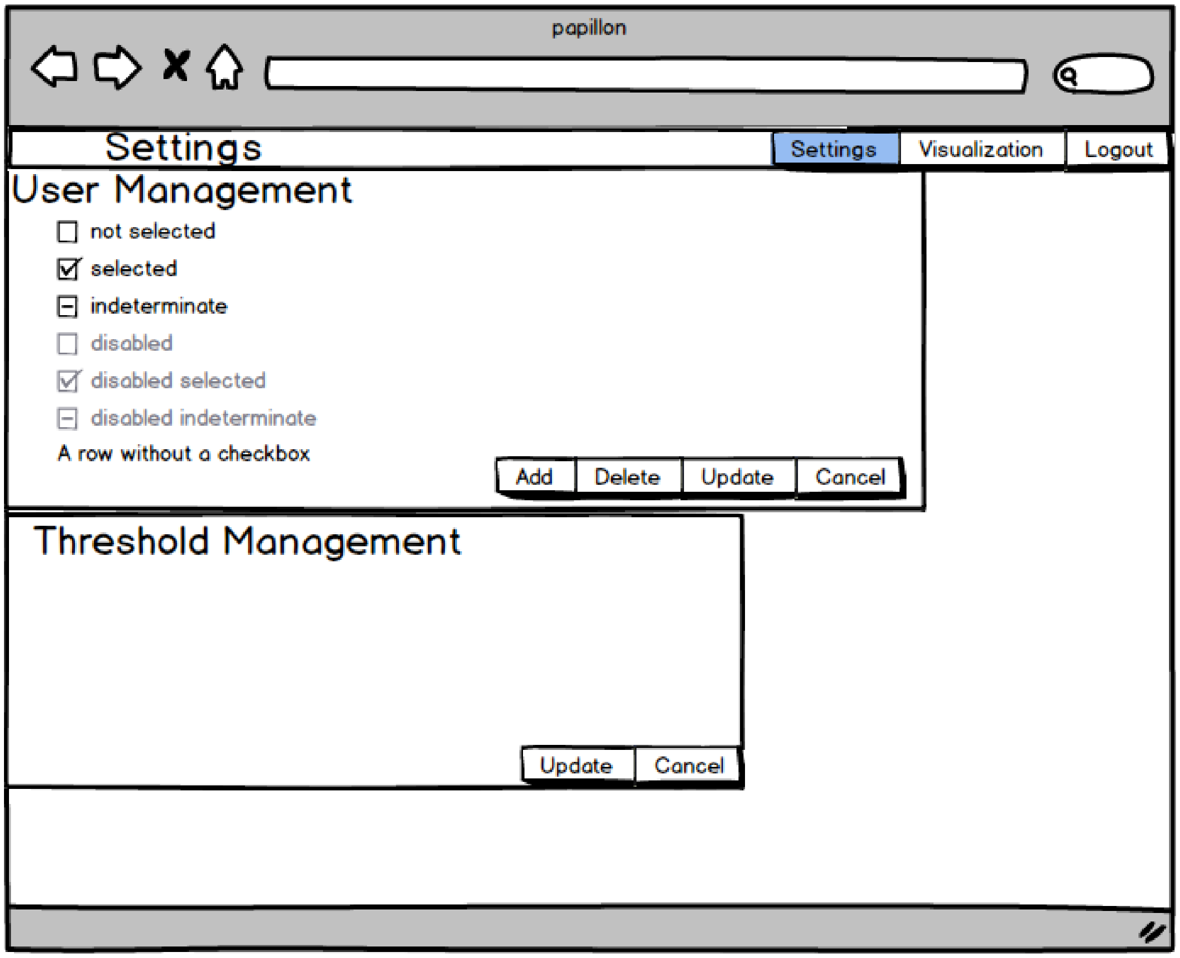


Figure 3.6: Visualisation page mock up

consumption of its parent component, and place this as a new extended activity in the database of the extended PAPILLON. Moreover it will compare the power consumption of the activities from the original PAPILLON with the corresponding threshold values and trigger an alert in case of one reaching it. In the second step it will use these calculated power consumption values to calculate the ones of its parent, place them in the database of the extended again and compares it to its thresholds. This will repeat till it is at the highest component. If there was one or more values that exceeded their corresponding alerts, the notification service will start and send each alert to its user. Since the original PAPILLON gathers and stores values for activities every 60 seconds, the watchdog wakes up every 60 seconds, does its tasks and goes back to sleep. This enables a synchronized system and near-real time notifications. Also, since calculation times change with the amount of Servers attached to the system, the watchdog can individually adapt its sleeping time. Because it is also possible that the activities of the original system can be out of sync, it will always compare its time with the activity with the lowest hostId. For the possibility of a missing database entry of an activity, the watchdog takes the last existing one to estimate the power consumption of higher components. However, if more than 3 entries are missing, it checks if the host is still running or the configuration of the system has changed. In case of an error here, like the missing host, a database error or some other error, the watchdog will send a notification to the specified administrator. Figure 3.7 shows the simplified program flow of the extended PAPILLON watchdog task.

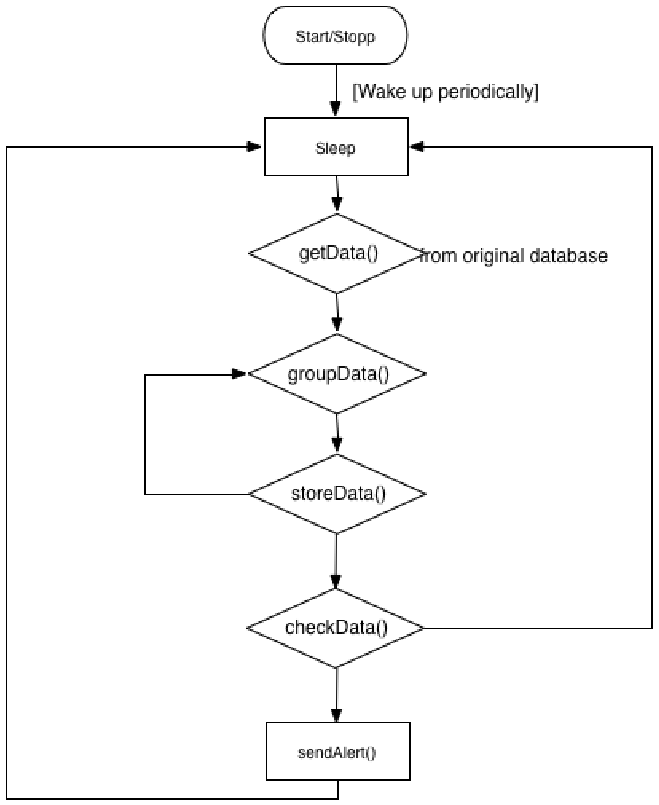


Figure 3.7: Simplified program flow of the watchdog task

3.5 Server for App Data

Although it is possible to send larger amounts of data via Push-Notifications, Push-Notifications are usually used to tell the device that something happened and that it needs to do something. In this system they will be used to tell the app to load specified data from the Server. There is already a working API from the original PAPILLON which delivers the data in either XML or JSON. However, since this system will need more data and some other information as well, this will be extended to fit the requirements.

3.6 Android App

The App should be able to visualize the energy consumption of a datacenter or the different components of the datacenter on a handheld device. Furthermore should it include Push-Notifications via cloud services for components of the datacenters which reach a certain threshold of energy consumption. This threshold can be adjusted by an administrator of the corresponding user administration-tab of the web application. This web application will include the specification of which user can access the visualization data of which components of the datacenters as well as which user gets which Push-Notifications. Figure 3.8 and Figure 3.9 show the mock up of the Android app. These were also generated with Mockup Builder[5].

3.6.1 Requirements

Several requirements must be fulfilled to ensure a correct functionality as well as a pleasant user experience.

Usability

The app should be able to visualize the data in a simple form of the web application and usable in a logical similar way.

Near-real time

Since the Android app should be able the alert the user of certain events, the alerting system should work in near-real time.

Version and Hardware Independent

The Android app should run smoothly on different devices and different versions of the Android OS.

Energy efficient

Mobile devices have a limited energy supply, so the application needs to use power saving strategies like push by alert or refresh on commit.

[5]`http://www.mockupbuilder.com`

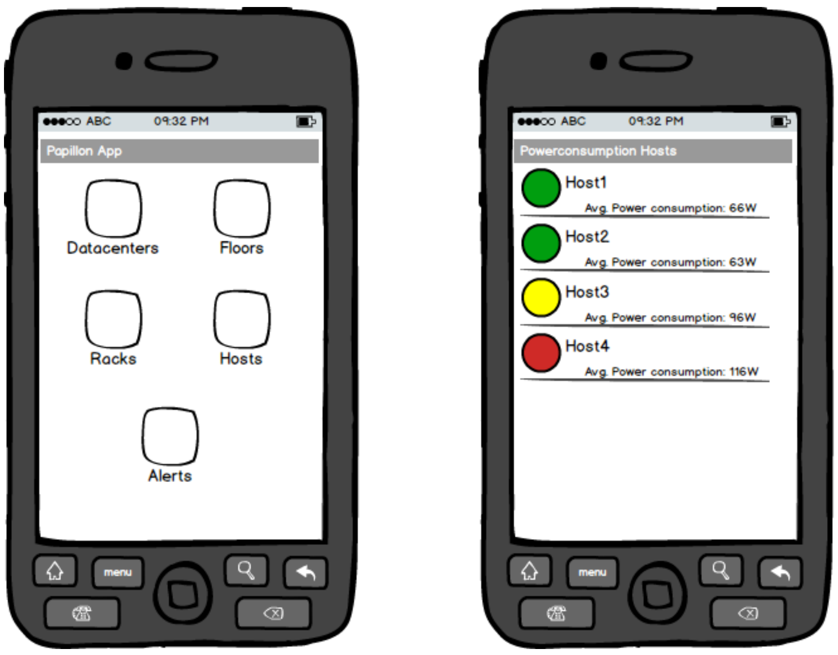


Figure 3.8: Android mock up: Start view and list view

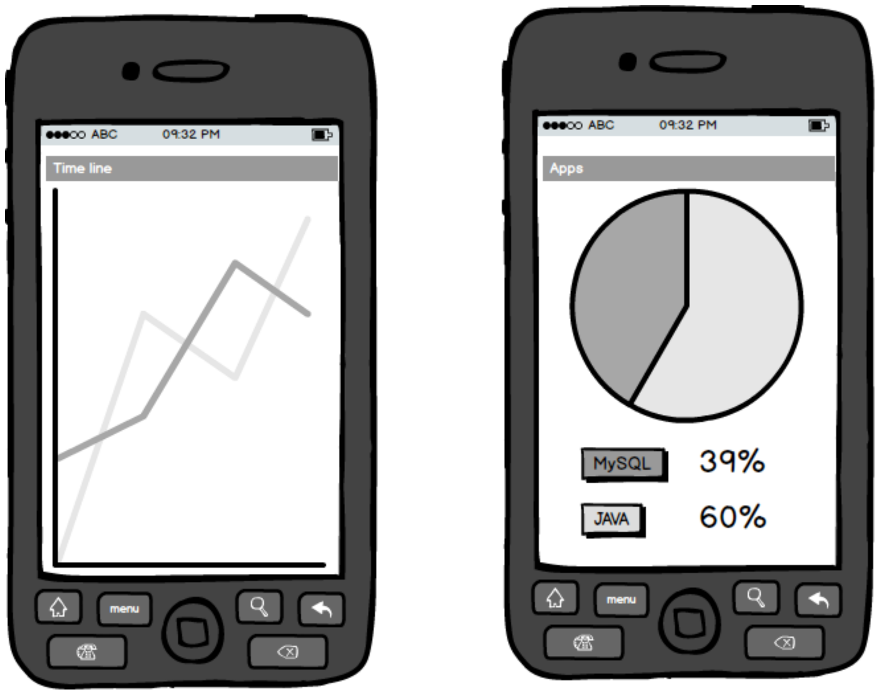


Figure 3.9: Android mock up: Powerstatistics

3.7 Web Application Testing

For the web application to work proper, it must be tested proper. This section deals with the requirements for the web application testing as well as a possible approach with unit testing.

3.7.1 Requirements for the Web Application Testing

To ensure, that the web application is tested to a sufficient amount, following requirements for the testing are to be met.

Automated

Since the project is quite big with a lot of different components, as much as possible must be automated. A test suite that executes all the test cases and delivers an easy to read output about the results would be best.

Coverage

The test suite should cover as much as possible of the written code of the project.

Execution Time

The larger the project, the larger the execution time of the test suite. That execution time shouldn't exceed a reasonable time.

3.7.2 Unit Testing

A unit test is a method of a class, which is used for testing only. Several such test classes are called a test suite and an execution of the test suite should be performed in the specified order. To make the system sufficiently to test a particular test method should be written for all methods. As this is, however, very time and cost consuming, can be seen on methods where it makes little sense to testing, such as getters and setters do without. Furthermore, all test methods should be called multiple times with different values to offer more security. There is a form of unit testing for the ZK-framework, especially designed for it. But since there is a lot of dynamic in this part as well as several asynchronous tasks, testing this part is quite complicated. A combination of regular unit testing with special Unit and Integration testing is required.

Integration tests identify errors and problems that occur during assembly of individual modules. To perform this, various unit tests should be combined and performed again repeated with different values. Furthermore, as many different combinations should be achieved.

3.8 Watchdog Testing

Since the watchdog is one of the most important parts of the whole system, it needs to be tested thoroughly. There will be lots of calculations, database interactions and possible notifications for the app. Also, the timing is very important for the watchdog, since an out of sync error might cause missing values, wrong calculations or a missed notification. The first step of testing the watchdog is by unit testing all methods of it. This will be done by creating mock up data in the form the watchdog would usually get from the database and test its basic functionality without the actual database. The next step is testing one whole calculation step of the watchdog routine. Again, the created mock up data will be used for that instead of the actual database. To test the timing of the watchdog, the synchronization, the database connection and the notification service, more than one step of the routine is needed. More components working together makes it even more complicated to do so. To ensure a smooth and correct functionality this will be tested in the combined testing3.11.

3.9 Server Testing

To test the server for the app data, similar to the watchdog more components will be needed to work together again. The core functionality will be tested with help of unit tests and the rest will be tested in the combined testing3.11.

3.10 Android App Testing

In this section the testing of the android app will be discussed. Starting with the requirements, which are quite similar to the testing of the web application, To test the Android app two different approaches have been taken into closer consideration. GUI–Ripping and unit and integration testing with help of the JUnit framework.

3.10.1 Requirements for the Android App Testing

For the web application to work proper, it must be tested proper. To ensure this, following requirements for the testing are to be met.

Automated, Coverage and Execution Time

Similar to the testing of the web application, it should be as much as possible automated, have a high code coverage but still a reasonable execution time.

Version and Hardware Independent, Energy Efficient

Similar to the requirements for the Android app, its testing should be version and hardware independent and energy efficient.

3.10.2 Unit Testing

Unit tests have proven themselves for desktop systems and can of course also be used for mobile applications. There are specific libraries for using the JUnit framework for Android apps.

3.10.3 GUI-Ripping

GUI ripping has been proven to be a reliable test method for mobile devices and is also due to the fact that it uses the graphical interface for testing, easier to port to other platforms and quite hardware and OS version independent. In addition, to find faulty managing resources, the combination of GUI ripping and the use of the separation into neutral cycles would (like in [31]) make sense, since a similar approach is used for the two of them.

They both start out with the generation of the tree model of the app and use this tree model for their testing purpose. Figure 3.10 shows a possible tree of the proposed app.

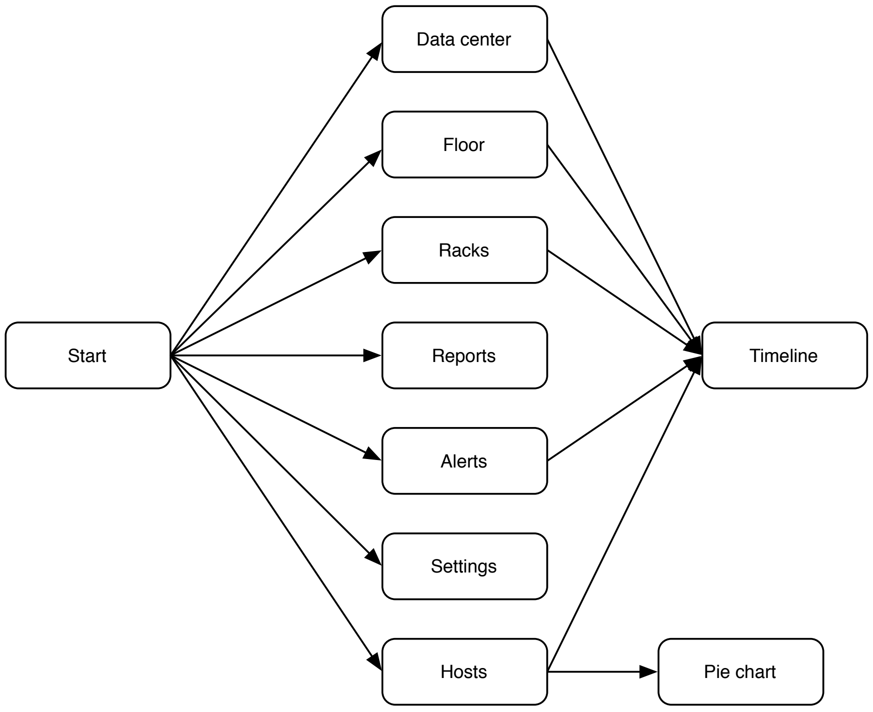


Figure 3.10: Possible tree of the app

Graphical interfaces of Android applications consist of two primary components: Activities and views. Activities are the part of the application with which the user can interact, thus ensuring the compilation of the content. This content, along with the view or viewGroup (more views) form the front-end of apps. This can be considered as states to establish a tree model of the application with conditions. For the states to store all the graphic elements, attributes, values, buttons and links them in the tree model with states that can be reached from this point. About the properties and values neutral cycles can be calculated, which are then called when testing repeated.

Once the app is analyzed and the tree model was established, testing can begin. It will go through all the states of the apps and neutral cycles are repeated several times. In addition to that, interrupts, such as minimizing the Apps, an incoming call or the likes can be simulated.

By expanding the neutral cycles on larger areas of the program and the monitoring of the memory in use, a large part of the memory can be checked, and a large proportion of the possible problems can be found.

Figure 3.11 shows the usual activity lifecycle. Since handling resources in the wrong stages of the lifecycle will result in errors, special treatment has to be made for these and special care must be taken for the different stages onCreate(), onStart() and onResume() and the corresponding onPause(), onStop() and onDestroy().

3.11 Combined Testing

Since the extended PAPILLON will be a system consisting of a database, a watchdog routine, a web server and an Android app, a lot of errors are possible between the components. At this point, the single components should be tested thoroughly and work within their requirements. To assure components working together, their communication between them and their timing needs to be tested. Starting with just two of them, like the original PAPILLON and the watchdog routine, would make sense, since those are the first ones needed during the normal usage of the system as well. Making sure of the correct visualization of the data in the web application would require testing the 2 databases to work with the web front end. For the app to display the correct data it would require the 2 databases, the transfer and the necessary calculations and the app itself to work together. The notification service would additionally need the watchdog routine as well.

[6]Source: `http://developer.android.com`

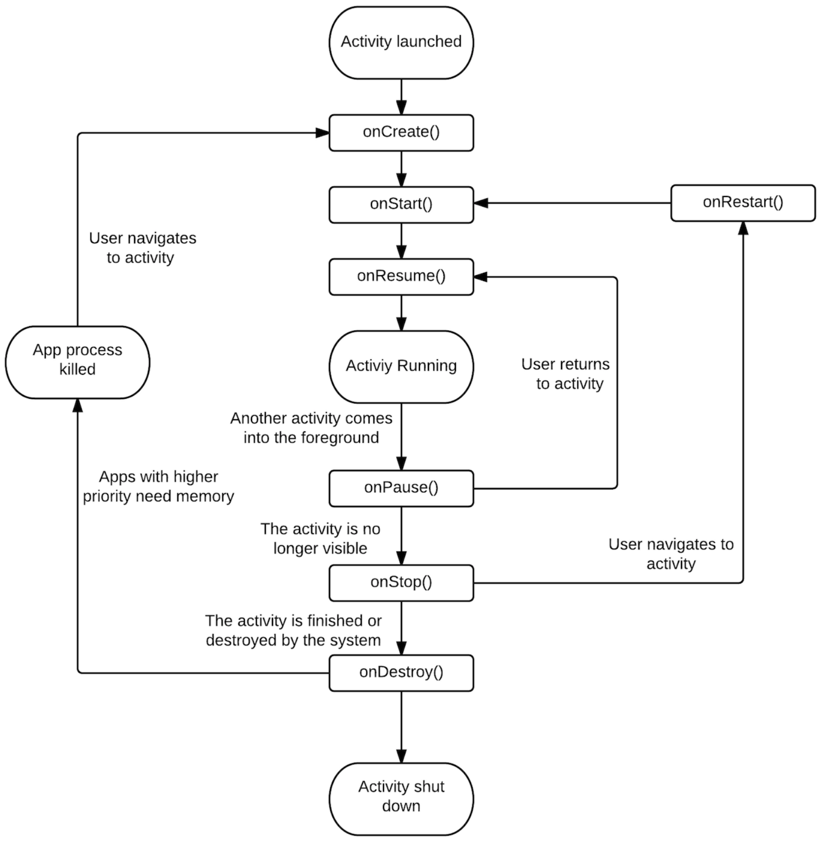

Activity launched
onCreate()
onStart()
onRestart()
User navigates to activity
onResume()
App process killed
Activiy Running
User returns to activity
Another activity comes into the foreground
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to activity
onStop()
The activity is finished or destroyed by the system
onDestroy()
Activity shut down

Figure 3.11: Activity Lifecycle of an Android App[6]

3.12 Summary

Several designing issues were discussed in this chapter. First, a detail description of the original PAPILLON system was given. Followed by the required changes to implement the extended PAPILLON system. A description of the design of the proposed web application and the Android app including a mock up was given afterwards. Finally several possible testing strategies have been discussed in detail.

Chapter 4

Implementation

This chapter is about the implementation of Android App, the corresponding Web application and the testing in detail of those two components. The first section features the used development environment as well as the used technologies in detail. The second section is describing the extension to the database, which was necessary for the new system. The following section will be about the Web application. Starting with the visualization part of the Web application, the administration of the whole system and the watchdog for the monitoring of the energy consumption. In the fourth section the RESTful web service for the Android app is explained. The fifth section will show the Android app in detail. Section 6 will be about testing the Web application. And the seventh section will be about testing the Android app with the help of JUnit and GUI-Ripping. Furthermore there is an extended notification service for superusers, which is configured to notify them in an event of an error of the system. These notifications are described in the eighth section in detail.

4.1 Development Environment

This section will describe the tools and techniques used for the implementation of the software. Section 3 gave already an outlook of the design of the web application, the Android app and the testing of each part as well as the testing of the system as a whole. Following is the description of the development tools and techniques which are used to achieve that.

4.1.1 Eclipse

Eclipse is an integrated development environment (IDE) mostly written in Java for Java developing. Through various plug-ins it can be used to develop applications in many programming languages like C, C++, PERL and PHP to name only few of them. Furthermore it has a version of JUnit already included.

4.1.2 ZK Framework

ZK is an open-source Ajax Web application framework, written in Java, that enables creation of graphical user interfaces for Web applications.

The core of ZK is an event driven mechanism based on AJAX, which can be described by either a XUL component set or a XHTML component set, and a ZUML markup language for designing user interfaces.

ZK uses the approach called focused-on-the-server synchronization components and pipelining between clients and servers is done automatically by the engine, and Ajax code completely transparent to developers of web applications. Therefore, end users get an interaction and similar to those of a desktop application response, whereas the complexity of development is similar to that which would encoding desktop applications. Figure 4.1 shows the architecture of the ZK framework.

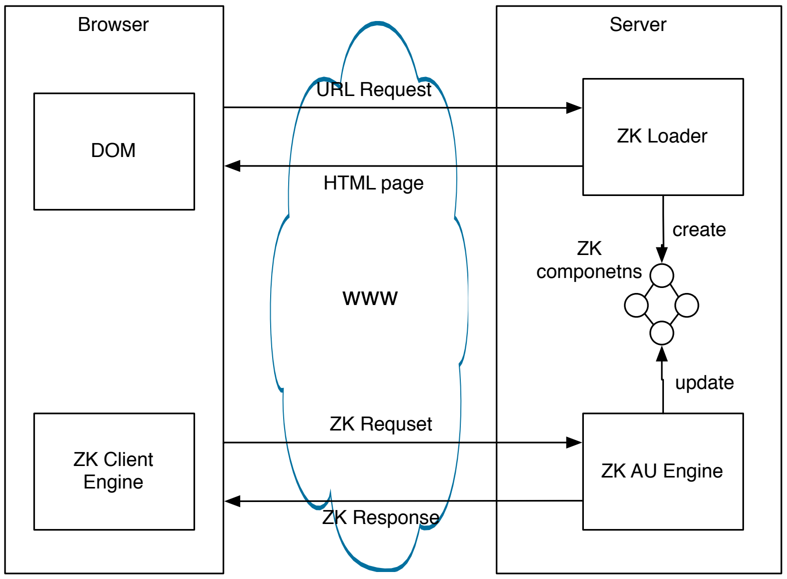


Figure 4.1: ZK framework architecture

4.1.3 Hibernate ORM

Hibernates main task is the Object–Relational Mapping (OR mapping, short ORM). This allows ordinary objects with attributes and methods to be stored in relational databases and generate from corresponding records in turn objects. Relationships between objects are mapped to the corresponding database relations. Hibernate solves object-relational impedance mismatch problems by replacing direct persistence–related database accesses with high–level object handling functions.

4.1.4 MySQL

MySQL is one of the world's most widely used relational database management systems. It is available as open source software as well as a commercial enterprise version for different operating systems and forms the basis for many dynamic websites.

4.1.5 MySQL Workbench

MySQL Workbench, the successor to the now-defunct DBDesigner 4 of FabForce, a database modeling tool, that allows database design, modeling, creation and editing (maintenance) of MySQL databases into a single, integrated environment. MySQL Workbench can import the XML models of DBDesigner and should run on any X Window System and MS Windows.

4.1.6 Apache Tomcat

Apache Tomcat was chosen as a web server for the project. It is open-source, implements several different specifications of Java and provides a pure Java HTTP web server environment for running Java code.

4.1.7 REST

Representational state transfer (REST) is an abstraction of the architecture of the World Wide Web. This architectual style can be applied to the development of web services, in which they are referred to as RESTful web services.

The architecture of REST is defined by a configuration of components, connectors, and data constrained. The most important ones are listed below:

- Client – server: Clients aren't concerned with data storage or similar things and servers aren't concerned about the user state or the user interface. Furthermore new clients can be added or replaced in an easy way.
- Stateless: No client-data is stored on the server between requests. Every new request holds all the information that is necessary.
- Cacheable: Clients can cache responses. This helps improving scalability and performance.

- Uniform interface: The uniform interface simplifies and decouples the architecture, which means the server can identify individual requests and send the response in HTML, XML or JSON for example.

4.1.8 SOAP

SOAP is a network protocol specification that allows data exchange between systems and remote procedure calls can be made. SOAP is an industrial standard of the World Wide Web Consortium (W3C). It is based on XML for data representation and on Internet protocols of the transport and application layer for transmission of messages.

The structure of a SOAP message is shown in figure 4.2. It is usually a XML document with the following elements: An envelope, which identifies it as a SOAP message, a header, a body with the response information and maybe a fault for error information.

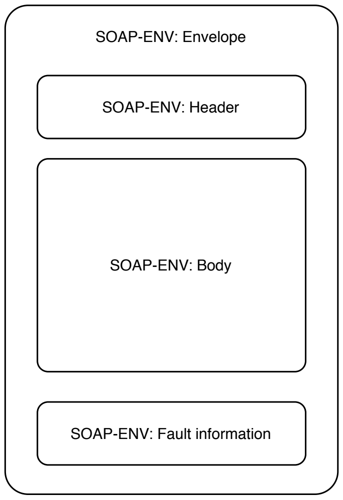


Figure 4.2: Structure of a SOAP message

4.1.9 XML

XML is a set of rules for representing hierarchically structured data in the form of text files. XML is used among other things to be platform– and implementation–independent exchange of data between computer systems, particularly via the Internet.

4.1.10 JSON

JavaScript Object Notation, is an open standard format that uses human-readable text to transmit data objects consisting of attribute–value pairs. It started with being just for JAVAScript (hence the name), but is now language independent and is primarily used for transmitting data from a server to a web application. The syntax of JSON is simpler and therefore it is often easier to read and write. In general, JSON also reduces the overhead compared to XML.

In XML, many values and properties could potentially be described as both attributes as well as child nodes, which can lead to problems if this is not prevented by very strict specification. JSON can not experience this problem. One of the strengths of JSON is the fact that it is valid JavaScript in defining itself there are some limitations. This allows a JSON definition in JavaScript directly with the eval () - convert function in a JavaScript object. With data from potentially unsafe sources but a parser should definitely be used as eval and possibly execute malicious program instructions.

4.1.11 Android Developer Tools

The Android Developer Tools (ADT) is a plugin for Eclipse to develop Android applications. It provides an integrated environment to build the Applications, debug the Applications as well as exporting the signed or unsigned application package for further use.

The ADT also includes an Emulator for Android handheld devices and an Android Virtual Device Manager for creating and managing different types of handheld devices or different Android OS versions.

4.1.12 XMPP

XMPP and its extensions support for messaging, conferencing with multiple users, view the online status, file transfers, dispatch of certificates and many other services. The network architecture is reminiscent of the Simple Mail Transfer Protocol (SMTP). Each tethered to the Internet XMPP server can exchange messages with other servers. Thus, compounds on the seller's borders are possible. Messages are then forwarded to your own server, then to the remote server and the receiver by the user.

4.1.13 Metis Mobile Cloud

XMPP interacts via the Metis Mobile Cloud. The MMC consists of two components. The Android service (Xmppd) and the Android library (XAFLib). The library is integrated in the Android service. The service is a daemon which provides the functionality for the XMPP communication.

4.1.14 AChartEngine

AChartEngine is a library for Android applications that enables charting. The supported chart types can contain multiple series, can be displayed with the X axis horizontally, which is the default, or vertically and support many other custom features. The charts can be built as a view that can be added to a view group or as an intent, such as it can be used to start an activity. The time chart as well as the pie chart are implemented using the AChartEngine library.

4.1.15 JUnit

JUnit is a framework for testing Java programs, especially for the automated unit testing of individual units (classes or methods). A JUnit test has only two possible outcomes: either the test succeeds (then he is "green") or he fails (then it is "red"). The failure may be caused by an error or a false result (failure) have both signaled by exception. The difference between the two terms is that failures are expected, while errors occur rather unexpectedly. Technically failures are signaled by a special exception named "AssertionFailedError" while all other exceptions are interpreted by the JUnit framework as error.

4.1.16 ZATS Mimic

ZK Application Test Suite is a collection of tools which can help users test their ZK-based application. Currently this suite has one module, ZATS Mimic, but it is still being further developed. ZATS Mimic is a unit test library that can be used with any well-known unit test framework (e.g. JUnit and TestNG) to test ZUL-pages without an application server or a browser.

4.1.17 ZTL

ZTL (Zul Testing Language) ZTL affords developers the ability to create testing cases for their application to verify functionality and features. ZTL is build upon standard testing libraries JUnit and Selenium, combining together to provide seamless integration with ZK.

Additionally ZTL takes testing to a new level with the introduction of ICT (image comparison testing) providing an easy way to test component positioning in your application. For the purposes of this work, the ZATS Mimic library seemed more fitting than the ZTL, so it was decided to use this one.

4.1.18 eclEmma

EclEmma is a free tool for Eclipse to show the code coverage of the software. In Eclipse it highlights lines in either green, yellow or red to show if the line is fully tested, partly tested or not tested at all, to ease the search for missing ones. Furthermore it can display a coverage summary in form of a list or export a report in either HTML, zipped HTML, XML, CSV or as a native JaCoCo execution data format.

4.1.19 MonkeyRunner

MonkeyRunner is a tool from the ADK which lets you control the connected device or emulator from outside of Android code. It uses Python scripts with which it can install, launch and manipulate apps, as well as taking screenshots. For manipulating the app it offers to send keystrokes, simulate touch events or send a drag gesture (press, move, and release) to the device. If functions provided by MonkeyRunner and python are not sufficient, the developer can make a custom extension. Furthermore it enables multiple device control, which means a test script could run on more devices or emulators at the same time to increase the amount of data that is tested.

4.1.20 ARC Welder

Android Runtime for Chrome (ARC) is a compatibility layer and sandboxing technology for running Android applications. ARC Welder Chrome app allows you to run Android apps on Chrome OS or on the Chrome web browser. The amount of apps running in the Welder is limited and since its beta version was released quite recently, it is probably still quite buggy. For developers who want to create Android apps that also run in Chrome OS and the Chrome browser, its useful for testing.

4.2 Extended Database Schema

Since the original PAPILLON database didn't include several tables used for the watchdog or any sort of user management, the database schema is extended. In Figure 4.3 is shown, which tables and what relations are added.

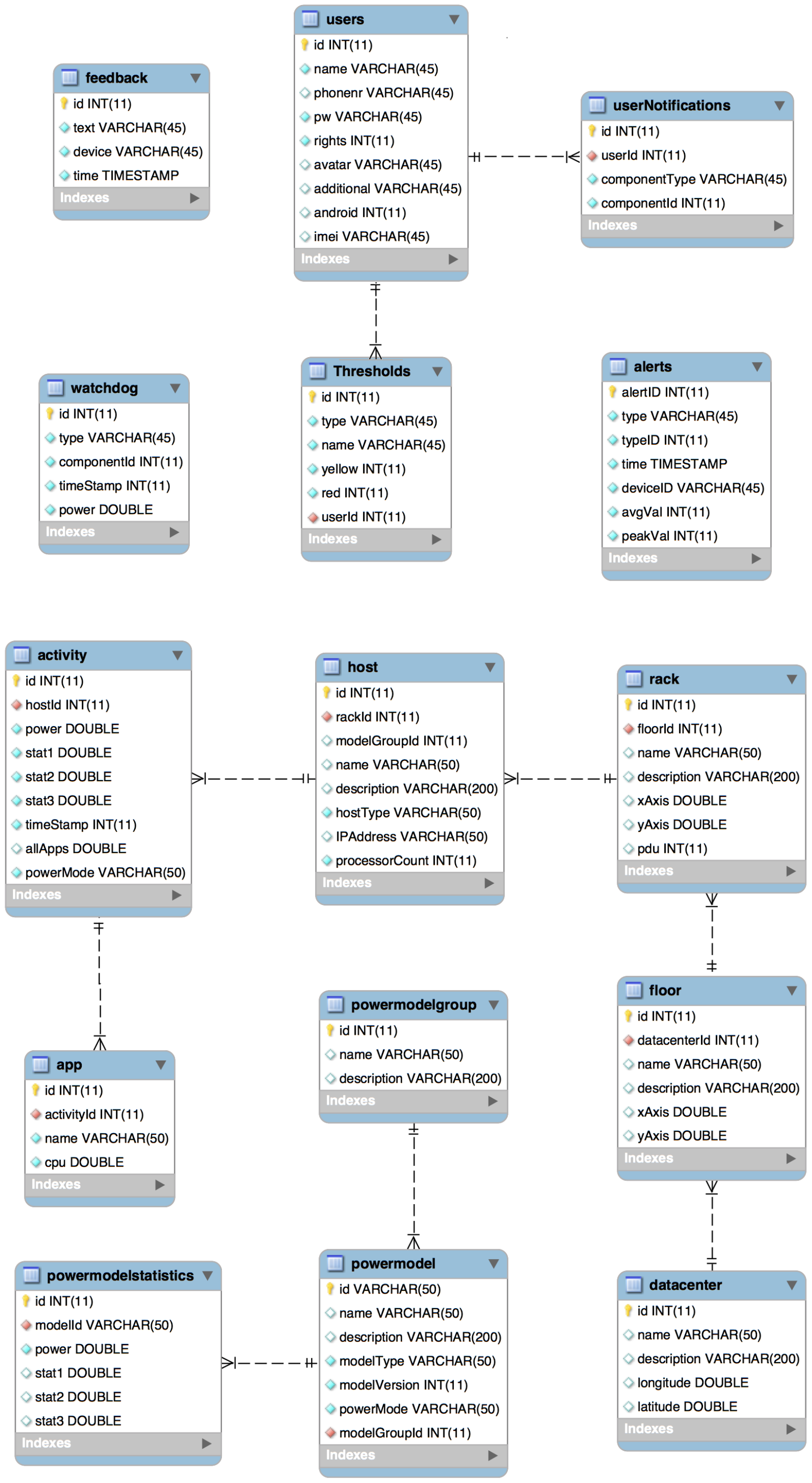


Figure 4.3: Extended database scheme

4.2.1 Entity Description

The following section describes each entity of the extended database in more detail and information about the further usage is given. Also the source to the data for the described entity is mentioned.

Thresholds

This table holds the thresholds for yellow and red alerts of the different component types. The user with the corresponding userId can access the data in this table in the settings page of the web application.

No.	Field	Type	Unit	Description
1	id	integer	-	primary key, auto increment
2	type	varchar	-	type of the threshold
3	name	varchar	-	name of the name of the threshold
4	yellow	integer	W	value for yellow alert
5	red	integer	W	value for red alert
6	userId	integer	-	id of the corresponding user

Table 4.1: Description of database entity thresholds

Users

The user table consists of the name, the phone number, the password, the rights for the user administration as well as the threshold administration, his avatar and if he assigns an Android handheld device to his profile the phone or tablet and the IMEI for the handheld. This data is accessable by a superuser in his settings page.

No.	Field	Type	Unit	Description
1	id	integer	-	primary key, auto increment
2	name	varchar	-	name of the user
3	phonenr	varchar	-	phone number of the corresponding user
4	pw	varchar	-	password of the user
5	rights	integer	-	rights of the user
6	avatar	varchar	-	location of the avatar
7	additional	varchar	-	addition information to the user
8	android	varchar	-	Request successful
9	imei	varchar	-	IMEI of the corresponding android phone

Table 4.2: Description of database entity users

UserNotifications

This table is used for the cloud notifications. If the watchdog recognizes an alert, it looks up the userId associated with the componentType and the componentId of the alert. Then it uses the userId to find the corresponding user in the users table and notifies that handheld device.

No.	Field	Type	Unit	Description
1	id	integer	-	primary key, auto increment
2	userId	integer	-	id of the corresponding user
3	componentType	varchar	-	type of the component that triggered the alert
4	componentId	integer	-	id of the component that triggered the alert

Table 4.3: Description of database entity userNotifications

Watchdog

The purpose of this table is to store the grouped power values of the racks, floors and data centers which are calculated during the watchdog routine in 4.4. These grouped power values are then used for displaying in the visualization part of the web application and for the time chart of the Android app. Since the calculation time of these values increases the more components there are, storing and reusing them is way faster then calculating them for every request.

No.	Field	Type	Unit	Description
1	id	integer	-	primary key, auto increment
2	type	varchar	-	type of the component
3	componentId	integer	-	id of the component
4	timeStamp	integer	S	timestamp of the measure
5	power	double	W	consumed power of the component

Table 4.4: Description of database entity watchdog

Alerts

If there is an alert, the watchdog server in addition to notifying the handheld device via cloud, stores the alert with important information like the time, the component, the average power consumption and the peak power consumption that set the alert of in this table. From the Android app the user can access the older alerts as well, but only for new ones will he get notifications. If a notification is not acknowledged by Android app, the

XMPP server resends it two more times. This data is part of the user management for user with an Android hendheld and can only be accessed by superuser.

No.	Field	Type	Unit	Description
1	alertId	integer	-	primary key, auto increment
2	type	varchar	-	type of the component
3	typeId	integer	-	id of the component
4	time	timestamp	S	time of the alert
5	deviceId	varchar	-	IMEI of the corresponding android phone
6	avgVal	integer	W	average power consumption of the component
7	ack	boolean	-	Acknowledgment of the notification
8	peakVal	integer	W	peak power consumption of the component

Table 4.5: Description of database entity alerts

Feedback

This table is used for storing feedback-messages sent from the Android app users to the server. The Information can be viewed by superusers on the user administration page of the web application.

No.	Field	Type	Unit	Description
1	id	integer	-	primary key, auto increment
2	text	varchar	-	feedback text from the app
3	device	varchar	-	IMEI of the device
4	time	integer	S	timestamp of the message

Table 4.6: Description of database entity feedback

XMPPPapillon

This table is used by the XMPP server for sending alerts to the corresponding device. This is the second part of necessary settings for user with an Android device. Can only be done by superusers in their corresponding settings page.

4.3 PAPILLON Web Application

The web application provides the visualization of the energy consumption of the different components. Moreover it offers the management of user and threshold settings. The visualization offers the possibility to illustrate the different components in singular or

No.	Field	Type	Description
1	id	integer	primary key, auto increment
2	userId	integer	id of the user
3	number	inter	Phone number of the corresponding Android app
4	message	varchar	command for loading the data via RESTful

Table 4.7: Description of database entity feedback

grouped views. The settings offers the management of the Android device, the thresholds for the alerts and profile settings for the corresponding user. For superusers it offers all the settings for users, datacenters and the extended notification service.

4.3.1 Class Description

Figure 4.4 shows the class diagram of the developed software. The NavigationViewModel and the NavigationbarController present the core of the navigation through the web application and decides on the further visualization of the application. Depending if the user wants to look at energy consumption data from components with an installed PAPILLON client or make changes in the settings. The SidebarController displays the possible components for the corresponding user and the user can select which components he wants to view.

Basic users can adjust settings of thresholds and of himself like adding an Android device or changing the notificationtypes. Superuser can make adjustments to other users as well. The ThresholdViewModel and the UserViewModel are responsible for that.

LoginController

The LoginController is used for managing the login and logoff to the homepage. It basically consists of the two text values for the user name and the password as well as an label for error or welcome messages.

AuthenticationService

The AuthenticationService is used to set and get the user rights from the user database and is necessary to distinguish the possible features which the user is allowed to access.

NavigationViewModel

The NavigationViewModel handles the displaying of the visualization and the settings. For the visualization of the power statistics it uses the chartBox objects obtained from

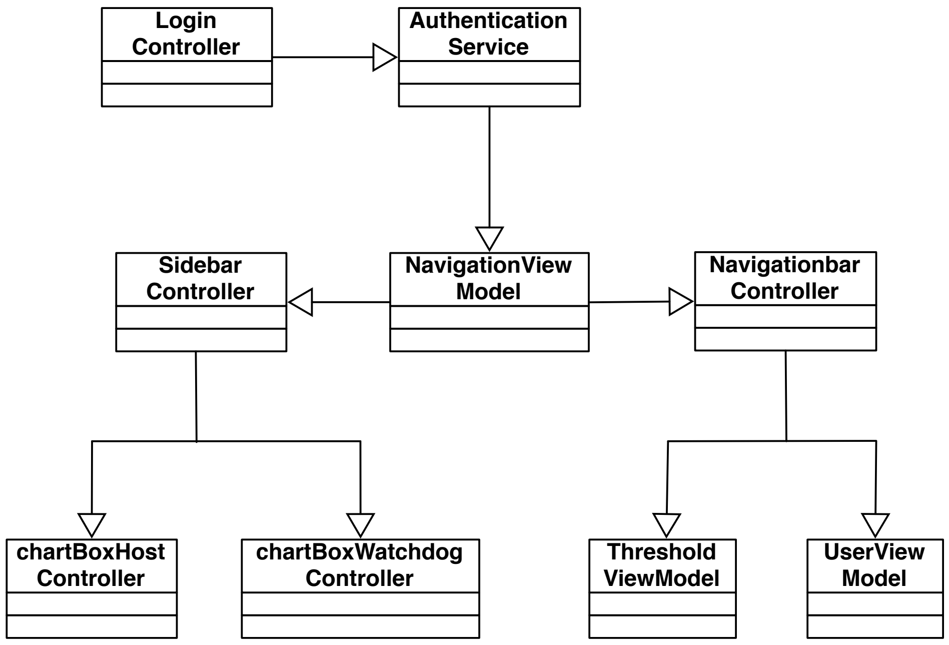


Figure 4.4: Web application class diagram

the ChartBoxHostController and the ChartBoxWatchdogController. In case of the first it will display an empty page.

NavigationBarController

This class contains the controls to navigate between the visualization and the settings as well as the logging off.

SideBarController

This class contains the controls to select, add and remove components from the visualization.

ChartBoxHostController

This class handles the visualization of the hosts and the apps. Since the power consumption of the hosts is from the original PAPILLON, which means it's data is in a separated table. Furthermore a user can click on a point of the visualization to see the power consumption of the different apps for this point of time in form of a pie chart.

ChartBoxWatchdogController

The ChartBoxWatchdogController handles the visualization of the racks, floors and data centers.

ThresholdViewModel

This class contains the controls for the thresholds. This includes the visualization and the updating of the thresholds for each user.

UserViewModel

This class contains the controls for the users. This includes the visualization and the updating of each user for himself as well as the visualization and the updating for all users for superusers.

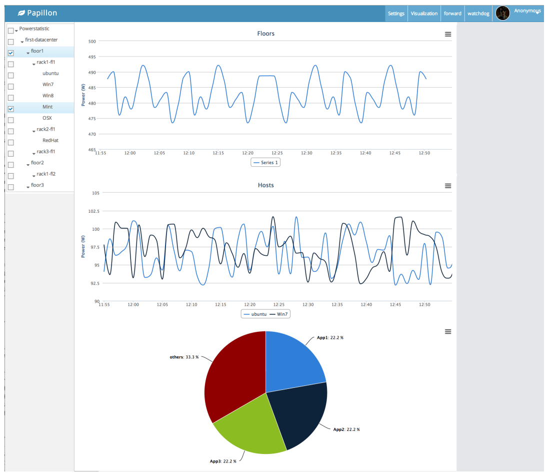
4.3.2 User Interface and Interaction

The web application will consist of three major parts. These are a visualization tab, a settings tab and a thread running in the background, which is the watchdog service. The visualization is divided into the sidebar on the left side of the page, the navigation bar on the top and the presentation of the data. With the sidebar the user can select, which data to display in the presentation area as well as the possibility to change the visualization time. The user can select different components and add or remove them on different levels of abstraction. The navigation bar is used for switching between visualization and settings as well as for logging off. A screenshot of the visualization part is shown in Figure 4.5a. The user administration page allows to manage different settings like the thresholds of the components. A screenshot of it is shown in Figure 4.5b. Since there are different rights associated with the users, the view might be reduced to some users.

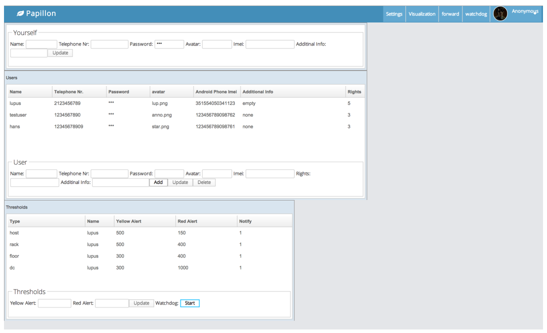
The user interface of the PAPILLON web application is illustrated in a more detailed way in figure B.1 and B.2 of the Appendix of this thesis.

4.3.3 API Communication

The communication flow of the PAPILLON web application is shown in figure 4.6. First, a login statement with username and password information is sent to the authentication service. This queries the user database if a tuple with the transmitted username and password exists. In case of non-matching elements an error message is sent back to the web application. Otherwise the NavigationViewModel is called to generate and display the corresponding data for the user. The generated page is displayed by the web application



(a) Visualization



Yourself

Name: Telephone Nr: Password: *** Avatar: Imei: Additinal Info: Update

Users

Name	Telephone Nr.	Password	avatar	Android Phone Imei	Additional Info	Rights
lupus	2123456789	***	lup.png	351554050341123	empty	5
testuser	1234567890	***	anno.png	123456789098762	none	3
hans	12345678909	***	star.png	123456789098761	none	3

User

Name: Telephone Nr: Password: Avatar: Imei: Rights: Additinal Info: Add Update Delete

Thresholds

Type	Name	Yellow Alert	Red Alert	Notify
host	lupus	500	150	1
rack	lupus	500	400	1
floor	lupus	300	400	1
dc	lupus	300	1000	1

Thresholds

Yellow Alert: Red Alert: Update Watchdog: Start

(b) Settings

Figure 4.5: Visualization and Settings of the new PAPILLON

in the browser of the user. The configuration and settings of the web page as well as the Android app a corresponding settings page is accessed in the same way.

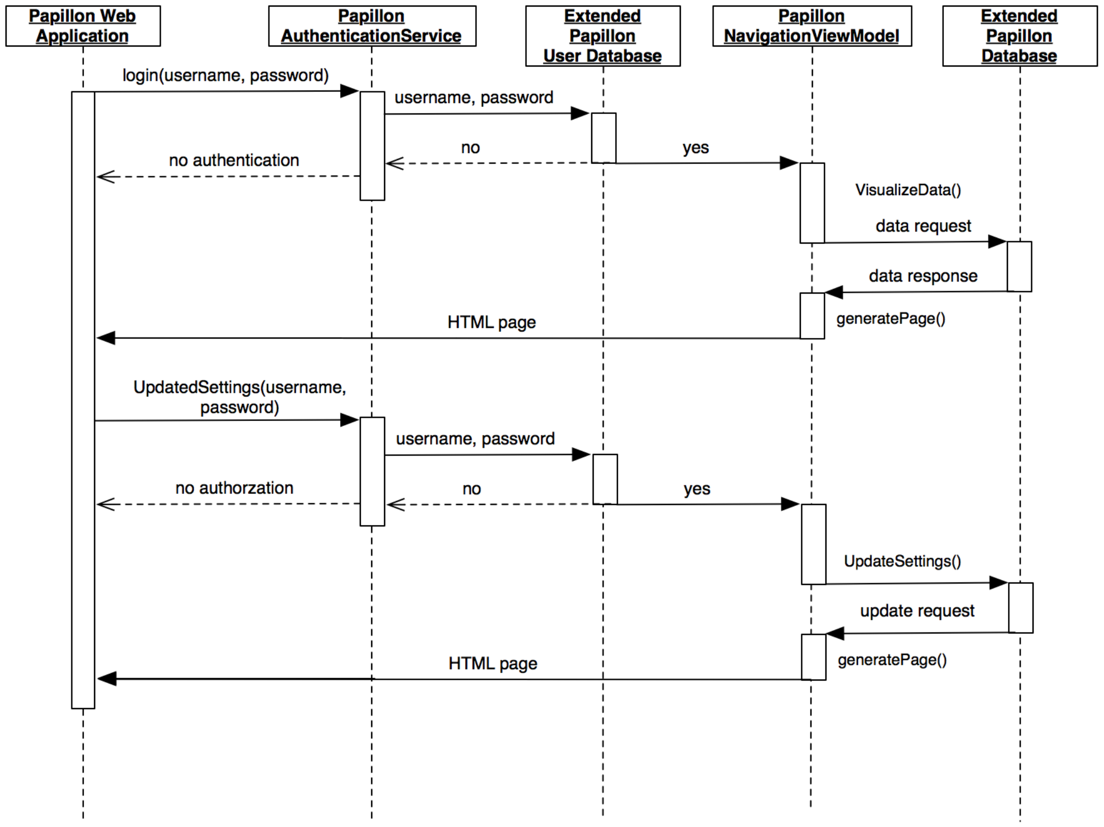


Figure 4.6: Web application communication flow

4.4 Watchdog

The original database of PAPILLON only covered the energy consumption of the hosts. To be able to find energy consumption values of higher sections, some calculations are needed. So the watchdog starts with reading the last entries for every active host and checks, if they reached their given thresholds. If there are hosts that reached one, the type, the id and the kind of alert are added to an alert-list.

Then the watchdog groups the values from the database to the corresponding racks and puts them in a separated table of the database. Then it checks again, if the values of the racks reached their given thresholds. If there are, the list is updated and the same (grouping, writing to database and checking) is done for the floors and the data centers as well. The new values from the database will then be used for displaying the data for data centers, floors and racks on the Android app as well. The alerts are now in different lists consisting of the type of alert (red/yellow), the type of the component and the id of this component.

4.4.1 API Communication

Figure 4.7 shows the communication flow of the watchdog routine. It repeats itself every 60 seconds to update the database with new values and notify the XMPP service if necessary. It starts with loading data from the original database, grouping it to the higher components and storing them in the extended database.

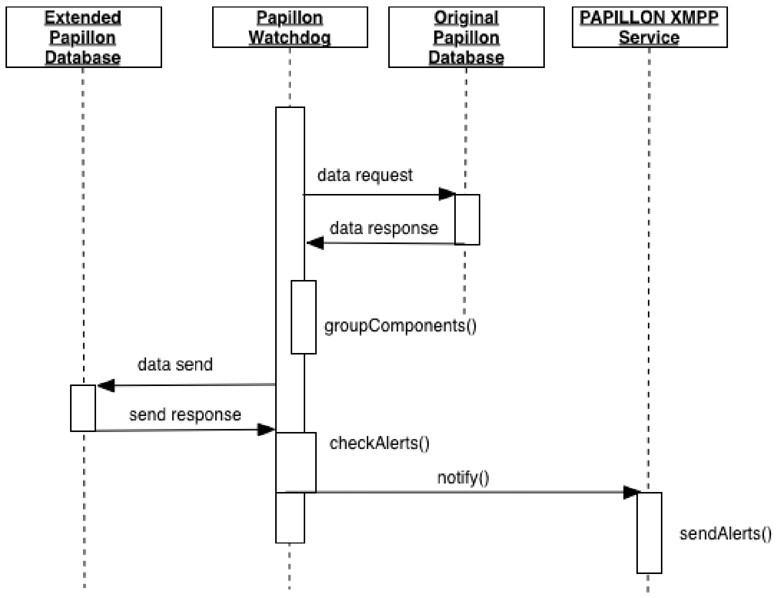


Figure 4.7: Watchdog service communication flow

4.5 RESTful Web Service

In the original PAPILLON, there was already an API which offered all of the information and data in form of JSON or XML. Since the extension of the project, more components than just the hosts are being visualized. Which means the consumed power of the higher components is not that API. The app could calculate these values the same way the watchdog does, and visualize them after that. However, being of smaller computing power as well as having lesser memory this would put a lot of stress on the mobile device. Furthermore the amount of data that would have been transferred to make this calculations as well as unneeded data would exceed a reasonable amount. So a new service to offer this data needed to be implemented. The RESTful web service is based on a client-server model. The clients, in our case the Android app, is sending a http request to the server. Depending on the request, the server will retrieve the requested data from the database, if necessary calculate further values to transfer, bring them in proper form and transfers them in JSON format.

4.5.1 Http Requests

The http requests coming from the Android app will consist of the type of data they request and the deviceId of the Android device. The server then looks up the deviceId in the user-table and retrieves according to the type and components assigned to the user the required data. To save time and data, this payload is minimized before transferring.

4.5.2 Http Response

The http responses will be the requested data in JSON-format, which will be discussed in detail in Section 4.6.3. In the event of an error, or an unknown request, specified error-codes will be transmitted which are listed in table 4.8. Some of them are standard HTTP status codes and some have been implemented to fit the project.

Code	Meaning	Description
200	OK	Request successful
400	Bad Request	The request could not be understood by the server
401	Unauthorized	The request requires user authentication
404	NOT FOUND	The server has not found anything matching the Request-URI
408	TIMEOUT	The calculations of the data exceeded the specified time
701	NO DATA	No data from the database
702	NO DATABASE	No connection to the database
703	PAPILLON STOPPED	Papillon has stopped

Table 4.8: Status-codes and their meaning

4.6 Android App

The Android app should be able to visualize the energy consumption of the different components on a handheld device. This visualization should be derived in an appropriate way of the web application. Furthermore should it include Push-Notifications via cloud services for the components which reach a certain threshold of energy consumption. The detailed data will be transferred via a RESTful web service.

4.6.1 Class Description

The Android app consists of seven activities. Figure 4.8 shows the activity class diagram of the developed Android app. The start view activity is center of the navigation and the first one to start. Each activity usually works with one or more layouts and some of them use listadapter to individualize the representation of the data. Through proper management of the layouts, apps can be used on devices with different screen sizes as well as in portrait mode and landscape mode.

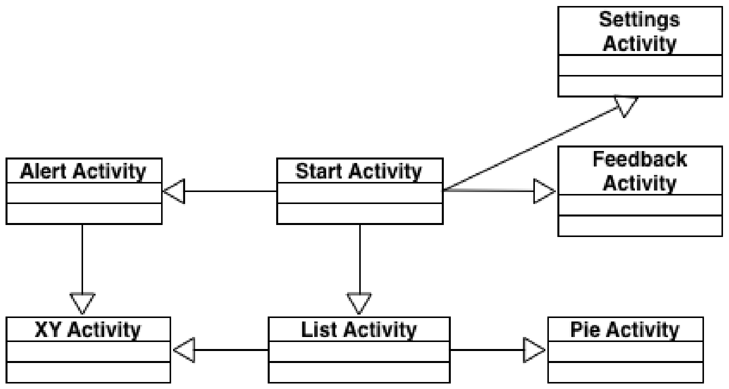


Figure 4.8: Android app class diagram

Start Activity

This is the first activity at the start up. For testing purposes, there is a flag to set, which makes the app start up in the alert activity. The start activity loads the settings and checks if the device has a working wifi connection. If not, there will be a prompt to activate the wifi, because the app can't work without.

Alert Activity and Alert Adapter

This is the activity which is started when an alert notification from XMPP service arrives at the phone. It uses the AlertAdapter to sort and split the alerts by color (red/yellow).

By clicking on one of the alerts, it will display the power consumption of the last hour for that component.

List Activity and ActivityAdapter

This activity is started from the start activity by either clicking on Datacenter, Floors, Racks or Hosts. This activity then requests the necessary data via the web service and the ActivityAdapter will then display the chosen components with their status color and the average power consumption. The layout to this combination is based on a swipe to refresh layout.

Feedback Activity

The feedback activity reads the imei from the phone and sends it via the web service together with the feedback message to the RESTful web server.

Usersettings Activity

Usersettings covers the managing of the different thinks like the start up options, the background color or the visualization time.

XY Chart Activity

This activity is used to display a graph of the power consumption over the preset time. Updating of this one happens automatically every 60 seconds.

Pie Chart Activity

The pie chart activity is used to display the 3 most power consuming apps of a host at a certain time. That power is loaded via the stats in the list view for the hosts and passed on to this activity, so no data needs to be loaded.

Web Service

The web service is a asynchronous task used to transfer data. The different activities call that web service task with the mode and the specified service url to request or send data. In case of requests, the web service loads the data in background and uses an OnCompleteListener() to send a broadcast message. This message is received by the activity who then starts to display that data. If no data was transferred, it displays an appropriate error message. If the connection is not used for some time, the web service terminates it. In case of a send it awaits the terminating OK from the receiving part, otherwise it will display an error message as well.

VisualObjects

To minimize the payload of the data transfers, the Android app as well as the RESTful web service changes the original Activity-objects and the additional information to visualObjects, which consists just of the necessary information for displaying it in a list form. The energy consumption of higher components is also transformed into these visualObjects. VisualObjects consist of the following member variables:

- id: the id of the component
- name: the name of the component
- avgPower: the average power it consumed in the last hour
- color: the state of the component in form of a color

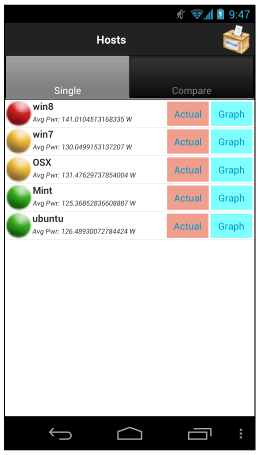
AlertObjects

AlertObjects are used to display alerts and like visualObjects minimized for the payload. Since more information is needed to display an alert, they have more member-variables. They occur less than visualObjects, so using 2 different types for it minimizes the payload of the overall usage.

- id: the id of the component who set the alarm off
- name: the name of the component
- type: the type of the component
- avgPower: the average power it consumed in the last hour
- peakPower: the peak in power consumption, that set the alarm off
- alrmType: the type of the alert (yellow/red)
- dateTime: the time of the alert in form of the date and time

4.6.2 User Interface and Interaction

The app has two different start views: a simple start view and an extended start view. The simple view is for users with less than 20 components (this value can be adjusted by the administrator). As the name suggests, the simple view is to make the navigation to the components easier, in case there are not enough to use navigation through higher components. The start view can be changed via the settings of the app. The simple view is shown in figure 4.9.


9:47

Hosts

Single | Compare

win8
Avg Pwr: 141.0104513168335 W
Actual Graph

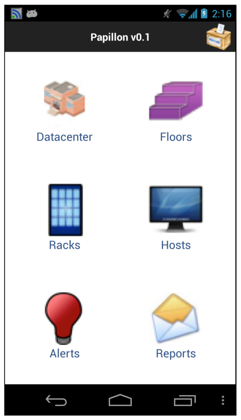
win7
Avg Pwr: 130.0499153137207 W
Actual Graph

OSX
Avg Pwr: 131.47629737854004 W
Actual Graph

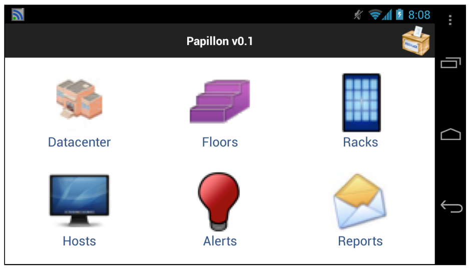
Mint
Avg Pwr: 125.36852836608887 W
Actual Graph

ubuntu
Avg Pwr: 126.48930072784424 W
Actual Graph

Figure 4.9: Simple-View of the App



(a) Normal view of the App



(b) Landscape view of the App

Figure 4.10: Extented-Start-View of the App

The extended view will allow more navigation and will be for users with more components or superusers (which can access all the data). The extended view is shown in figure 4.10 in both normal and landscape view.

Each list will include a signal color for energy consumption: green, red and yellow. For each component there will be two more views: a pie chart with the top 3 consuming components of the lower architecture and a graph for the consumption of the past hour. Furthermore the App is able to display the energy consumption in similar form to the web-Application. The Alerts-View will pop up and will be a list of components which reached the preset defaults, if one or more components reach it. To keep the amount of data for transferring at minimum, reloading of data is done with a pull and release gesture and is shown in figure 4.11.

There is also a feedback-tab, where users can give feedback of the app or report errors.

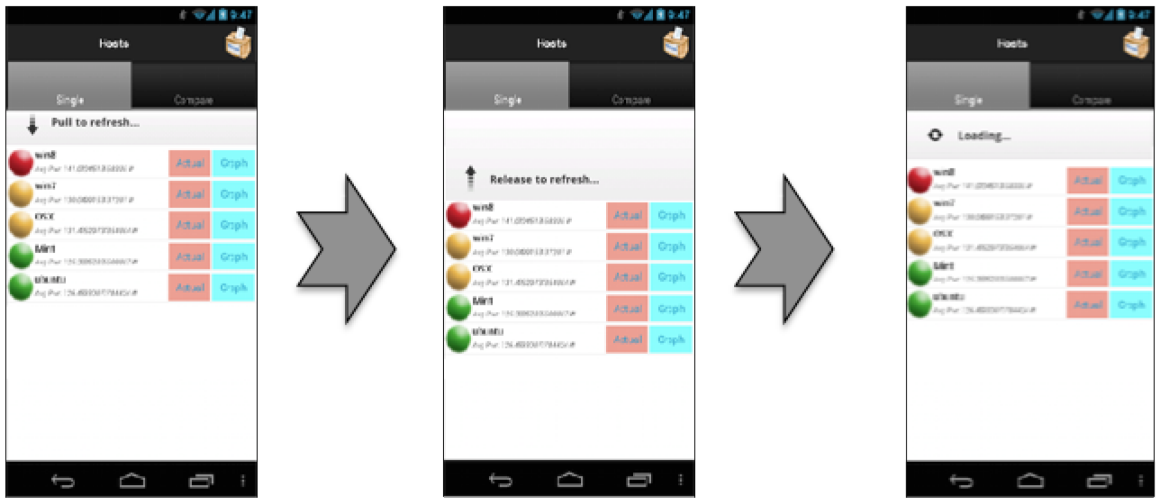


Figure 4.11: Pull-and-release-gesture for reloading

These feedbacks will be sent to the RESTful server and stored in a database. Furthermore they will be visible for superusers in the web application.

4.6.3 API Communication

The communication flow between the PAPILLON Android App and the back-end of the system is shown in figure 4.12. The transferred data (except the notifications) is in JSON format. The app sends a request to the RESTful web service URL, adding the requested component type (e.g. da for datacenter) and the IMEI of the phone. Via the IMEI and the component type the web service gathers the corresponding components, calculates the average power consumption of the last hour, sets the signal color and sends the data in form of a JSON array to the app. As soon as the transfer is complete, the app extracts each JSON string from the array and puts it into the list for displaying. The watchdog runs independent and wakes up periodically to do his tasks including to check for alerts. As soon as an alert happens, it notifies the XMPP service which sends the alert to the corresponding phone.

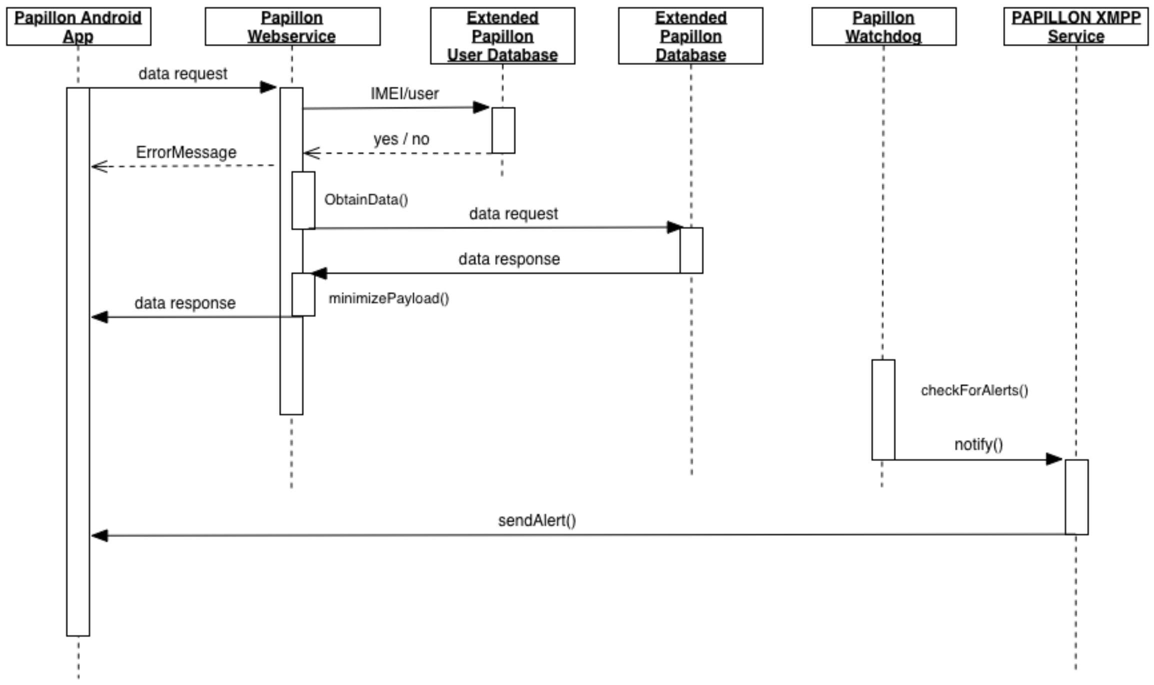


Figure 4.12: Android app communication flow

4.7 Web Application Testing

This section is about the testing of the web application. Since there is a java-based part of the code and a zul–based part, the testing was approached separately. Unit testing was mainly used for the java part, since it is easier to separate. For the zul–based part, more components were needed to work together, and the ZATS Mimic library was used to achieve that.

4.7.1 Unit Testing

For unit testing the JUnit framework which is already included in the current Eclipse versions is used. For testing the single units of the system, a huge amount of mock data needed to be created. This mock data was mostly created instead of data coming from the database. The tests were all fed with correct data as well as incorrect data to test different behavior. Usual unit tests have the statements Following is a description of some unit tests used during the developing.

View Models

The view models of the framework are core part of the navigation through the application. These view models return the proper ZUL–Pages of it. These return values are tested with different mock data.

Sidebar

The two major parts of the sidebar are the list of components and the chartBoxObj that regulates the charts that are to be shown. Several test cases have been written to test the proper generation of that list and the corresponding chartBoxObj.

4.7.2 Integration Testing

For integration testing several unit tests are packed together to a bigger test. Furthermore all the tests from the regular unit testing were executed again, without the mock data, but with real data from the database instead. Following is the description of tests which grouped several functional components together. Most of these test cases are with support of the ZATS Mimic library.

Login Tests

The process of login and logout is accompanied by the authentication service. This is tested by creating a new client that accesses the login page and tries different combinations of usernames and passwords and verifying the responses. Since an account is needed to access the page, the authentication service is included in all the following test cases as well.

User Management Tests

The User management page can only be accessed by superusers, hence tests different user rights are executed. Furthermore tests for adding, deleting and updating users are executed as well.

Threshold Management Tests

The thresholds are individual for each user, so again different tests with different users are executed. Further tests are mainly for selecting and updating data.

Sidebar Tests

The sidebar is again individual for each user, so different tests with different users are executed again.

Navigationbar Tests

The tests for the navigation bar are limited to a logoff process and navigation of the side. Furthermore there are tests for correct displaying of profile picture and the profile name.

Chartbox Tests

There are two different chart boxes. One for the activities of one or more hosts and another one for displaying the racks, floors and data centers (based on watchdog elements).

4.7.3 Regression Testing

These tests were carried out and adapted during the whole development process. Since the execution time of the whole tests never exceeded an unreasonable limit, no special regression testing strategy was developed or used. Since there were changes during the development process made throughout the whole project, it would probably have been more time consuming than what the decreased testing would have saved.

4.8 Watchdog Server Test

Since the Watchdog routine runs in its own thread and repeats its function, the testing of the complete routine was carried out in the combined testing (Chapter 4.11). However, the core function of it was derived into a separate function with one single run of the routine. This was carried out with mock data generated for the energy consumption of the activities. Test cases with different possible errors have been written as well as test cases for correct behaving. In table 4.9 examples for used values and errors are listed.

No.	Behavior	Description
1	ok	values ok
2	ok	Reaching thresholds different colors (yellow/red)
3	ok	Reaching thresholds different components
4	error	Client down
5	error	Host not found
6	error	Out of time
7	error	Zero power

Table 4.9: Examples of used values and errors

4.9 RESTful Web Service Test

Similar to the watchdog, testing the complete RESTful web service is done in the combined testing (Chapter 4.11). The core functionality, however like retrieving the data and calculating correct values of this data and mock data is carried out within the test suite.

4.10 Android App Testing

In this section the testing of the Android app is shown in detail. Starting with the unit testing for mobile applications in the first section and followed by GUI-ripping in the second. Figure 4.12 shows the order in which the tests will be automated.

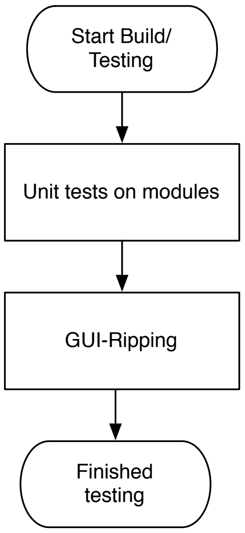


Figure 4.13: Test order of the Android app

4.10.1 JUnit

For unit testing the Android app several unit tests have been written using the JUnit framework included in Eclipse. The Android SDK offers libraries specific for testing activities and layouts. A separate project was generated and since a virtual device is much slower than a real device, this tests have been developed with the assistance of a real device again.

The testing was split into activity testing, which basically use the graphical front end of the app, the testing of the different adapters and corresponding lists and the testing of the asynchronous tasks like the loading and reloading of the data. Since the app is depending on data loaded via the web service, a huge amount of mock data has been generated to ensure testing the logic of the app is carried out first. For further testing these tests have been carried out again with real data, to ensure proper testing of the connection as well.

Since all activities are part of the navigation within the app, for all activities is tested which activities can be accessed and if they load correctly.

Start Activity Tests

Since the app starts with a different screen, if there is a certain amount of data, which is again depending on the user, different tests with different users have been carried out.

Feedback Activity Tests

Basically just the navigation is tested here and the appearance of the keyboard and if keystrokes appear on the screen. Sending and saving of feedbacks is tested during the GUI–Ripping (Chapter 4.10.2) and the combined testing (Chapter 4.11).

ActivityList Tests

For this activity several tests with different types of components of the visualization have been written.

AlertList Tests

This activity was tested for proper visualization of the alerts. Since this activity is automatically launched in case of an alert, which can't be tested without outside intervention, this is be tested during the GUI–Ripping (Chapter 4.10.2) and the combined testing (Chapter 4.11). There is an additional flag which can be set to start the app already into the alert list view. However, this is already tested in the home activity tests.

PieChart Tests

The pie chart activity was just tested for proper visualization of the data and errors that could occur with wrong or missing data.

XY Chart Tests

The time chart activity was tested for proper visualization of the data, updating of this data and of course, errors that could occur with wrong or missing data.

Adapter Tests

The Adapters are used for displaying lists of different types of components or alerts. So these adapters are tested for the different data they have to process and tested for wrong or missing data.

WebService Tests

Testing the web service without the connection doesn't make much sense, so testing the web service is limited to testing the behavior of the web service in case of errors. This errors are listed below.

- No connection
- No data
- Wrong data

4.10.2 GUI-Ripping with MonkeyRunner

For additional testing of the GUI, MonkeyRunner scripts were generated to test the app in different situations. This was carried out with Python scripts for three devices and one emulator. This should test the app for memory leaks or unexpected faults. The script takes screenshots at defined times and records the console output of MonkeyRunner.

The tree model of the GUI, which was generated manually is shown in Figure 4.14. This tree was used for navigating through the app. The tree model was generated by hand because it is quite limited in size. And since not many layout changes were carried out during the development process, not many changes have were necessary to adapt this tree. Each node consists of the activity with all the possible buttons as child nodes.

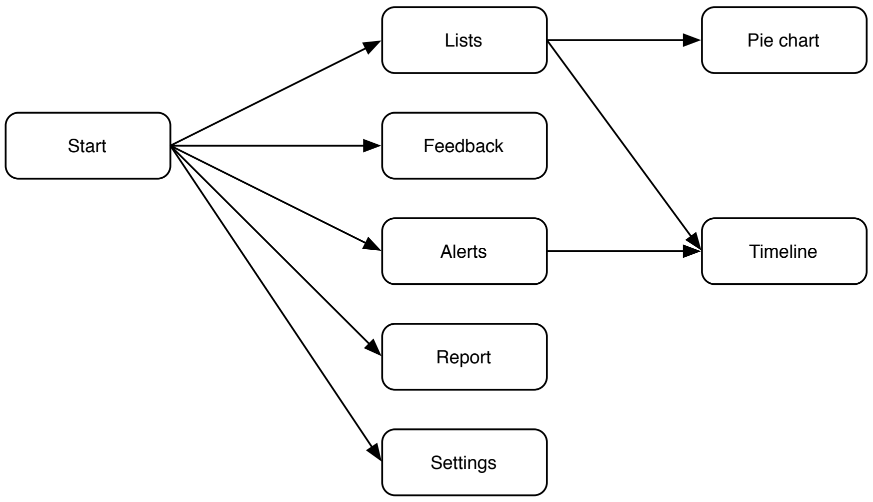


Figure 4.14: Actual tree of the Android app

The Python scripts were generated the following way: A separate Java project was generated. This project allows the tester to adjust the settings like delay time during action, the different test cases, file names and repetitions of the test. It has the necessary Python code connected to the different actions (like pressing a button or taking a screenshot) and then uses the tree model of the app, including the activities and possible buttons of an activity to generate the the whole script for a test case. This allows to adapt, change and make new test cases quite easy. A detailed description of how to run the different test cases is given in section A.2.3.

The following shows the detailed descriptions of the test cases that are carried out by the MonkeyRunner.

Stress Test different Stages

This tests the behavior of the app when starting, pausing, resuming, stopping, restarting and recreating the app as well as the behavior when tilting or turning the device. Since data is loaded on demand and not much is stored on the device, the use of the neutral cycles was reduced to smaller ones and is basically covered in the stress tests. Since this test does not show much of the app itself and uses the output of the console to record the process, it does not include any screenshots.

Navigation Test

This test case navigates through all possibilities of the app with help of the tree model and repeats all possible combinations to ensure no unexpected behavior occurs. To control the results of this test, the recorded console output as well as screenshots, which are taken in every location of the tree model can be used.

Stress Test Navigation and different Stages

This test case combines the navigation test with the stress test of different stages. The script navigates through the tree model and simulates the behavior of the different stages during each location of the tree again. This test should produce the exact same screenshots than the navigation test. The console output should include the switching of the different stages.

Long Running Time Test

Several different tests were generated to test the behavior of the app when running for a longer period of time. The test cases differ in the delay times between the actions, the number of actions and the type of the action to achieve a broad variety of test scripts.

Navigation and Send Feedback Test

This tests the navigation to the feedback activity and the sending a sample feedback. The feedback message is generated via defined clicks. Just a simple test to test the input handling of the app, since it is the only location of the app, where text can be entered. It is also part of the combined testing later.

4.11 Combined Testing

Figure 4.15 shows all the components and how they communicate with each other.

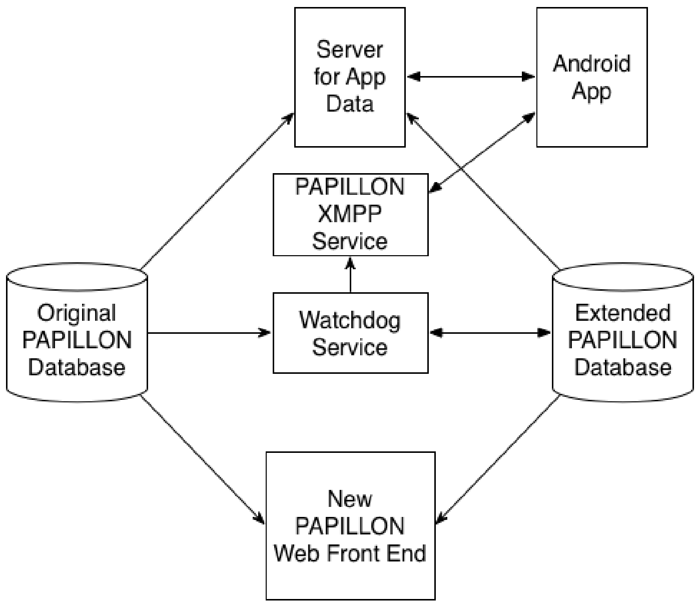

Server for App Data
Android App
PAPILLON XMPP Service
Original PAPILLON Database
Watchdog Service
Extended PAPILLON Database
New PAPILLON Web Front End

Figure 4.15: Communication between the components

A combined testing strategy has been developed to test the distributed system of the server and the Android app. The output of the RESTful web service and the Watchdog server can be accessed within the console of Eclipse or the Tomcat web server it is running on. In addition, the output of them can also be redirected into a file. The output of the Android app can be accessed with Eclipse or MonkeyRunner, so comparing that output to an estimated one, calculated from the predefined energy consumption data should give insight of errors and proper behavior. To test the whole system in action, two separate steps are necessary.

4.11.1 Server Side

For the whole system to run, it needs a constant flow of data. For that, a script has been generated to simulate the energy consumption data for activities and places them in the monitored database of the original PAPILLON. The script places the values every 60 seconds (similar to the original PAPILLON). The used values for this are quite similar to the one shown in 4.9, which means for errors as well as correct behavior.

4.11.2 Client Side

To cover as much as possible, the interaction with the Android app is necessary as well. Quite similar to the navigation test, a MonkeyRunner script was generated, that navigates through all possible views of the app and, at the feedback tab send a feedback to the database. Since the order is predefined, the generated output of the console of the RESTful web service, in addition with the output of the console and the generated screenshots of MonkeyRunner can be used to detect errors in the navigation between the views or wrong visualization of data. Furthermore it should test the correct behavior of the notification service.

4.12 Extended Notification Service

In addition to the given requirements, an extended notification service was implemented. Which means, whenever there is an error within the system, e.g.: no connection to the database during a http-request from the RESTful web service, a notification is send to a specified superuser with the description of the error. This way superuser can respond quickly to such errors. The following lists these errors as well as their meaning and where they occur. During the developing and running of test scenarios this feature was used for notifying the success or error of test runs to shorten their execution time. To assure proper working of the watchdog service, the RESTful web service and the notification service itself, a periodic notification about running faultless and some additional information can be send as well.

4.12.1 RESTful Web Service

The RESTful web service is a good starting point to search for faults. Since all the transfer from data (except the alerts) goes through it, it can detect errors with the device as well as errors from the database.

RESTful Running

A superuser can in the settings part of his web application set a periodic notification about the status of the RESTful web service. Starting with either OK or the number of errors, the notification lists in short the history since the last notification. Table 4.10 lists the explanations to the notifications.

No.	Shortcut	Description
1	OK	No errors to report
2	1-99	Number of errors
3	da:32	Number of queries for datacenters
4	fl:81	Number of queries for floors
5	ra:302	Number of queries for racks
6	ho:872	Number of queries for hosts
7	ti:435	Number of queries for timeline
8	ap:15	Number of queries for apps

Table 4.10: Shortcuts for extended notification service for the RESTful web service

No database connection at RESTful

This happens when the RESTful web service tries to retrieve data from the database and there is no connection to it. This can be caused by a missing network connection or the database server being down.

Unknown IMEI

This error occurs when a handheld device with an unknown IMEI tries to access data. This usually means a new device or an wrong configuration of the device. It can also mean a wrong configuration of the user database.

Wrong IMEI

This error occurs when an IMEI, that exists in the database, tries to access data, that isn't associated with it.

4.12.2 Watchdog Service

The watchdog service is another good source for searching for faults. Since it reads the values from the database and can detect errors with values or faults from the clients.

Watchdog Running

The watchdog can send a similar status notification concerning the status of it as the RESTful web service. If they are both activated, the notification service merges the 2 messages together and sends them as one. Table 4.11 lists the explanations to the notifications.

No.	Shortcut	Description
1	OK	No errors to report
2	1-99	Number of errors
3	da:32	Number of calculations for datacenters
4	fl:81	Number of calculations for floors
5	ra:302	Number of calculations for racks
6	ar:0	Number of red alerts
7	ay:3	Number of yellow alerts

Table 4.11: Shortcuts for extended notification service for the watchdog

No database connection at Watchdog

Basically it has the same meaning like the one in the RESTful web service. But if the error isn't easy to find, it helps locating it. And since the watchdog runs continuously, the error notification will be faster.

No data

This is accompanied by the type and the id of the component that triggers that error. It means there is no data in the database for the component at the specified time. This happens usually when a PAPILLON client is down.

No notificationtresponse

This happens, when a notification is sent to an user and the notification isn't acknowledged after 2 hours.

4.12.3 Web Application

In case of errors in the web application, the extended notification service will provide a superuser with a notification about it too. And again with the possibility of a periodic status notification.

Web Application Running

If the web application status notification is also activated, the notification service merges the messages together and sends them as one. Table 4.12 lists the explanations to the notifications.

Wrong Password

This happens, when an user tries to log on and enters a wrong password.

No.	Shortcut	Description
1	OK	No errors to report
2	1-99	Number of errors
3	lo:32	Number of logins
4	cs:81	Number of changes in the settings

Table 4.12: Shortcuts for extended notification service for the web application

Unknown User

This happens, when an unknown user tries to log on.

Exception

If something unexpected happens, and the server throws an exception, the superuser will receive a notification with the location of it in the source code.

Chapter 5

Results

This chapter will present the features of the implementation and the results obtained during testing the implementation.

5.1 Features

The following sections show the features of the web application and the Android app and gives a short overview of the whole system.

5.1.1 Web Application

The web application is able to visualize the different components of the PAPILLON system in various ways and offers a proper way to navigate through it. It offers different visualization time spans and can group components together. Moreover it includes an authentication service to give only permitted users access to the application. Depending on that permission of the user different settings to choose. This includes the user administration itself for superusers and administration of an Android device and the threshold of the alerts as well as the account settings for normal users.

5.1.2 Android Application

The Android app is able to visualize the different components of the PAPILLON system in a reasonable derived way of the web application. Furthermore it is able to receive alert notifications in case the given thresholds are reached. The app also includes a feedback option, where user are able to send feedback or report problems with the app. The navigation of the app is kept intuitive and straight forward and close to the one of the web application. Additional there is a simpler view of the app if all components can be shown in the start screen.

5.2 Web Application Testing

This section will describe the results of testing the web application with the JUnit framework. Moreover the code coverage of the system is shown.

5.2.1 JUnit-Testing

The output of the JUnit framework is shown in Figure B.5. It shows the elapsed time, the number of test cases and the number of errors and failures.

The code coverage in Figure B.6 shows how much of the source code is tested by the written test suite. The test suite achieved a coverage above 82 %. Since testing methods like getters and setters of the classes is quite an useless task, the code coverage of the part which handles the classes and the objects is only around 45 %, but the code coverage of the logic behind the application is above 90 %. A lot of missed instructions were caused by exception handling for exceptions that never occurred during testing.

5.3 Android App Testing

This section will describe the different test cases and the results of testing the Android app with GUI–Ripping and JUnit testing. For a detailed explanation of how to run these test cases refer to section A.2.2 and A.2.3.

5.3.1 GUI-Ripping

The output of the console and the screenshots of the app needed to be evaluated by hand. Table 5.2 shows the carried out test cases with the corresponding results. The test cases were executed on three different devices and one emulator. The devices and the emulator as well as their Android versions are listed below.

Device Nr	Device/Emulator	Android Version
Device 1	Samsung Galaxy Nexus	4.4.2
Device 2	Motorola Moto G	4.1.1
Device 3	HTC Evo	2.3.1
Emulator	Nexus S	4.3.5

Table 5.1: Used Devices and Versions

5.3.2 JUnit Testing

Testing the Android app with help of the JUnit framework gave the opportunity to test single modules and combined modules with Eclipse. Sadly, there are no proper working,

Testrun description	Reruns	Passed
Stresstest different stages	8	x
Navigation test	8	x
Stresstest navigation and different stages	8	x
Stresstest long time	8	x
Navigation and send feedback test	8	x

Table 5.2: Test cases carried out for GUI–Ripping

open source plugins for Eclipse available to show the code coverage of the source code. Figure B.7 shows the output of the JUnit framework in Eclipse. Furthermore, the cloud services did not work with the libraries offered by the Android SDK. For that purpose a different version of the app(without the notification service) was built.

5.4 Combined Testing

The script written for generating energy consumption values for the activities was used to test the complete system as well. The output of the console of the Android app as well as the output of the console of the web application and the RESTful web service had to be compared by hand again. Table 5.3 shows the carried out test cases with the corresponding results.

Testrun description	Reruns	Passed
Proper behavior for long period of time	8	x
Client down	8	x
Wrong data	8	x
Cloud boot up	8	x
Alert and cloud notification	8	x

Table 5.3: Test cases carried out for combined testing

Chapter 6

Conclusion and Future Work

This chapter summarizes and concludes the results gathered in this thesis and listed in the last chapter and with screenshots in the Appendix. An outlook gives an overview of implementation opportunities for the future.

6.1 Results

The distributed system for the representation of the power consumption as well as the management of everything works within the proposed requirements. In some areas it has more features than suggested, like the extended notification service for superuser. The test suite achieved quite good results and coverage. Over 90 % of the logic of the source code is tested and there is additional testing for the Android app and the whole system. Different versions of the app were tested on different devices to make sure they work as well. The reason for slightly different versions lays in the different versions of the operating system and the connected differences in support libraries. So is for example the notification service not working on Android OS 2.3.5. Since it did not work with the unit testing anyway, such a version was already existent.

It is really important to start testing from the beginning. In the areas where testing was not done exactly as planned, mistakes carried on till late in the development. The more advanced the developing already is, the more difficult and time consuming will it be the correct these mistakes.

6.2 Future Work

However good a software is already tested, testing is basically never done, and improving the test suite is always a good idea. Also, the coverage of a software is just as good as the unit tests used for testing. So there is always room for improvement. The suite used normal unit tests. Parameterized unit tests would cover more input variations and test more possibilities. Furthermore the System started to get quite big already, if an extension for it where planned, regression testing strategies will start to pay off in execution time of the suite. For the GUI-Ripping as well as the combined testing, the console output of the different parts of the system could be optimized and automatically compared by another script to just show differences in the outputs. This would reduce the amount of work of comparing the outputs by hand. An example for further improvement is that the whole system could be synchronized. Since the original PAPILLON is not synchronized, it would be necessary to start there. The watchdog service already synchronizes with the values in the database, so building on that, a complete synchronizing of the system should be possible too. The transmitted data is not encrypted at the moment. For security reasons, this could be adapted as well.

Appendix A

How to

This chapter will explain how to install, build, run and test the implementation for users not familiar with the source code or the development environment. To get most parts of the system running, the original PAPILLON needs to be running first. For installing the original refer to the accompanied installation and usage guide. Then the database needs to be extended. This can be done by running the sql skript extendDB.sql from the main folder.

A.1 Front ends

The source code was developed using Eclipse so for a first build it is the easiest way to import the source code in your Eclipse workspace. It was developed and tested and should therefor work with J2SE1-5. It was tested and should work with all newer versions as well. Mars and Luna where the two versions of Eclipse used for developing and testing the software.

A.1.1 Web Application

Just to get the web application running, the generated war(papillonzk.war, also in the main folder) file can be deployed on a web server like Apache Tomcat. For adjustments of the source code, the project, including all libraries should be imported into Eclipse. This project includes the source code of the JUnit tests as well. For running the web application on a local machine, Jetty for Eclipse was used. To configure the extended PAPILLON system and insert/edit user and component entries, use the settings page of the web application and the user admin with the password zk7322 for logging in. To have a simulation of the system, the serverside part of the combined testing can be used. This can be started at the watchdog section of the settings page of a superuser.

A.1.2 RESTful Web Service

Just to get the RESTful web service running, the generated war file(papillonRESTservice.war, also in the main folder) can be deployed on a web server. For adjustments of the source code, the project including all libraries should be imported into Eclipse. This project includes the source code of the JUnit tests as well. For the RESTful web service to run proper, it needs a running extended database.

A.1.3 Android App

To get the Android app running, there are several steps necessary. For proper visualization of data, proper data needs to be in the database. This means either the extended PAPILLON is running(Server and client(s)), or simulated data can be generated and stored in the database. Furthermore the PAPILLON RESTful web service must be running. A proper configuration of the app in the user settings is necessary as well. There exist several ways to install the app. It can be download from the web to the device. You can also push the apk(Android Application Package) file from a desktop and install it, which is also needed for running the MonkeyRunner scripts. Or it can be installed by executing the source code in Eclipse and selecting the device as a target.

APK

To push the apk from a desktop, the Android Developer Tools are needed.

Eclipse

To execute the source code of the Android app, the Android Developer Tools needs to be installed. The app was developed using the revision 19.0.3 of the Android SDK Build-tools. Since some of the functions are limited in other revisions the first build should be with this revision and Android 4.4.2(API 19). There are two versions of the app. One with cloud support and one without. Both versions are in the main folder of the project. The one without cloud support runs on Android 2.3 and up and the one with cloud support from Android 4.1 and up.

A.2 Testing

To test the system, Eclipse is necessary. Additional steps needed for JUnit testing is descriped followed.

A.2.1 Web Application

The project for the web application includes the source code of the JUnit tests as well. The test code is organized in the same packages like the source code in src/test/java. To execute this tests the user can run the project as JUnit test. To visualize the code coverage a tool called eclEmma is used. Available in the Eclipse Market Place. Eclipse then enables to run the coverage of a project.

A.2.2 Android JUnit

The Android JUnit test source code is in a different project. It needs to be imported and the necessary extensions need to be installed. Then the user can run the source code as JUnit test.

A.2.3 GUI-Ripping with MonkeyRunner

For the GUI-Ripping an Eclipse Project is used to generate the MonkeyRunner scripts. The project called MonkeyGenerator is in the main directory as well. To execute the scripts, the ADT are needed. For execution use the command monkeyrunner -plugin <plugin_jar><program _filename ><program_options >.

A.2.4 Combined Testing

To run the combined testing, the simulation script needs to be running, which can be started in the watchdog section of the settings page of a superuser. Furthermore, the MonkeyRunner script for combined testing must be executed.

Appendix B

Front end applications

B.1 Web Application

B.2 Android App

This section shows the different views of the Android app.

B.3 JUnit and Code Coverage output

The following section shows the code coverage of the web application and the Android app and gives a short description of the results.

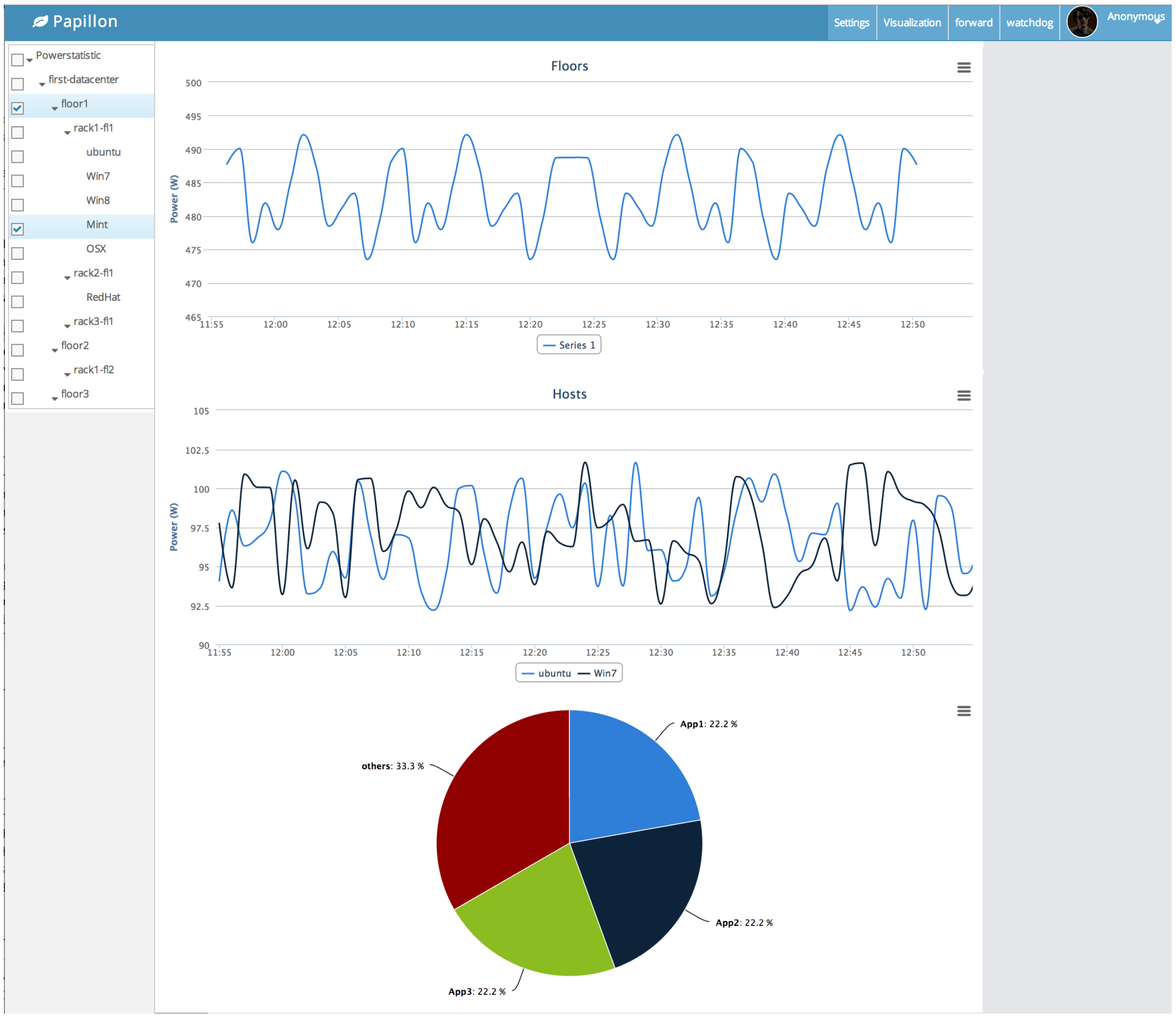
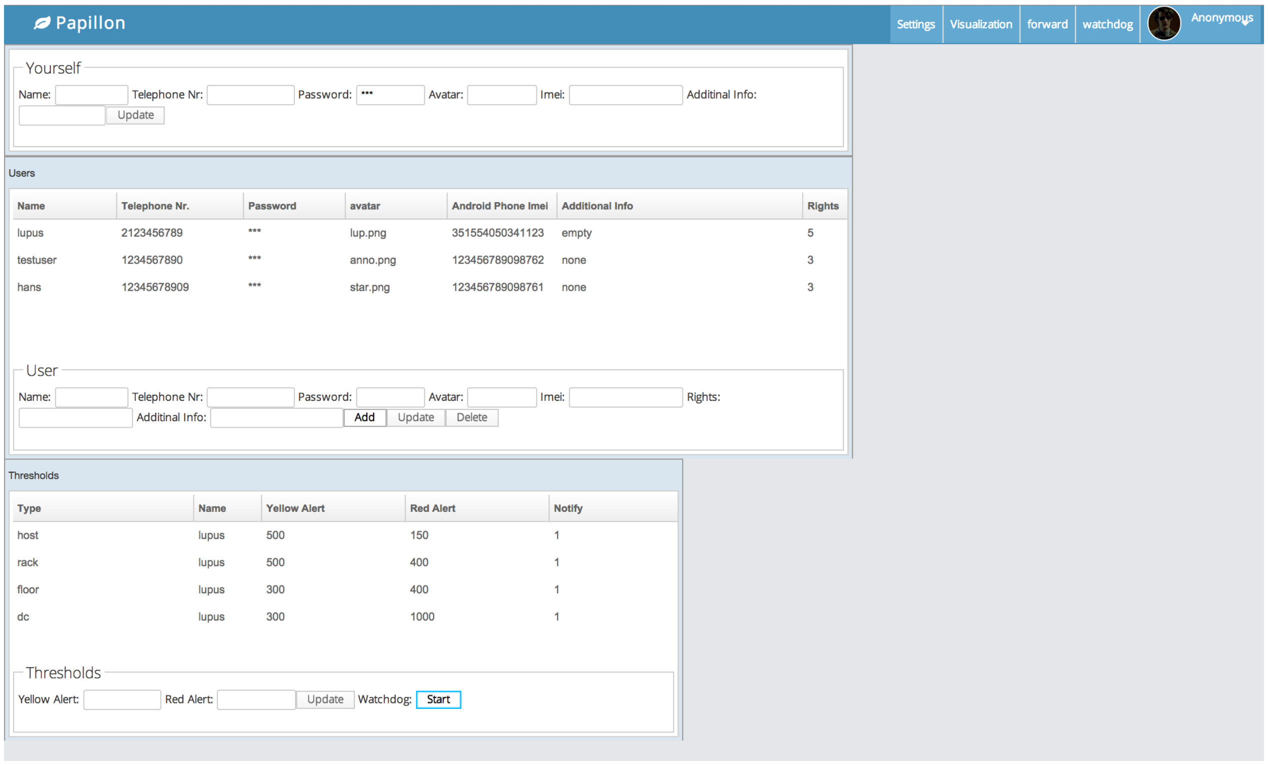


Figure B.1: Visualization of the data



Yourself

Name: Telephone Nr: Password: *** Avatar: Imei: Additinal Info: Update

Users

Name	Telephone Nr.	Password	avatar	Android Phone Imei	Additional Info	Rights
lupus	2123456789	***	lup.png	351554050341123	empty	5
testuser	1234567890	***	anno.png	123456789098762	none	3
hans	12345678909	***	star.png	123456789098761	none	3

User

Name: Telephone Nr: Password: Avatar: Imei: Rights: Additinal Info: Add Update Delete

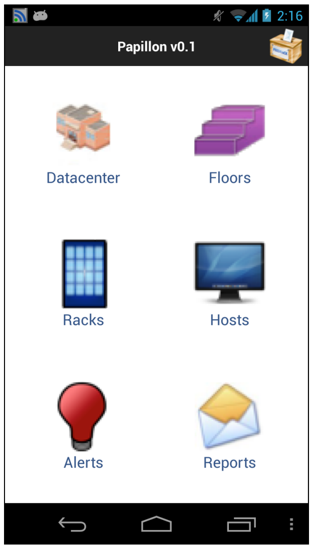
Thresholds

Type	Name	Yellow Alert	Red Alert	Notify
host	lupus	500	150	1
rack	lupus	500	400	1
floor	lupus	300	400	1
dc	lupus	300	1000	1

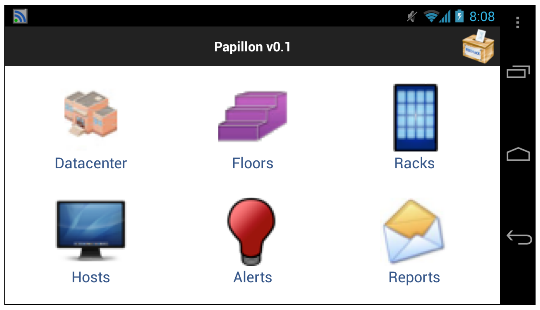
Thresholds

Yellow Alert: Red Alert: Update Watchdog: Start

Figure B.2: Settings

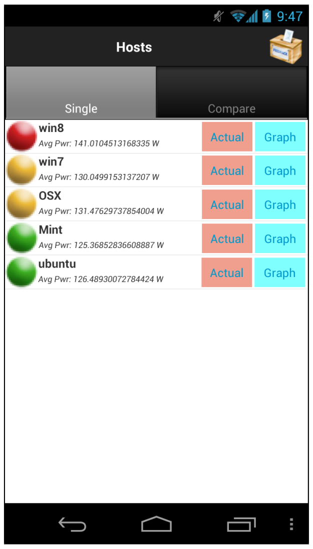


(a) Start view



(b) Start view in landscape view

Figure B.3: Start view in portrait or landscape view



win8
Avg Pwr: 141.0104513168335 W
Actual Graph

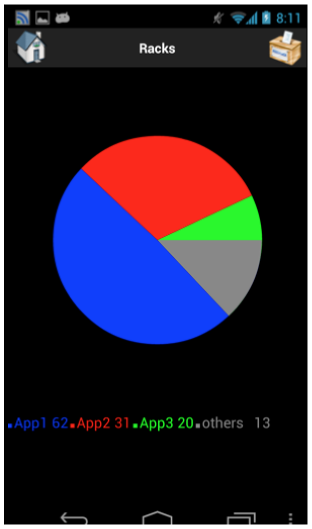
win7
Avg Pwr: 130.0499153137207 W
Actual Graph

OSX
Avg Pwr: 131.47629737854004 W
Actual Graph

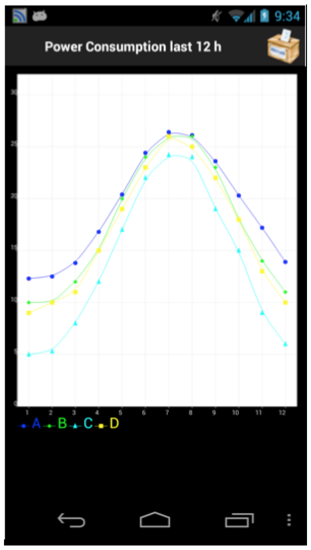
Mint
Avg Pwr: 125.36852836608887 W
Actual Graph

ubuntu
Avg Pwr: 126.48930072784424 W
Actual Graph

(a) List view

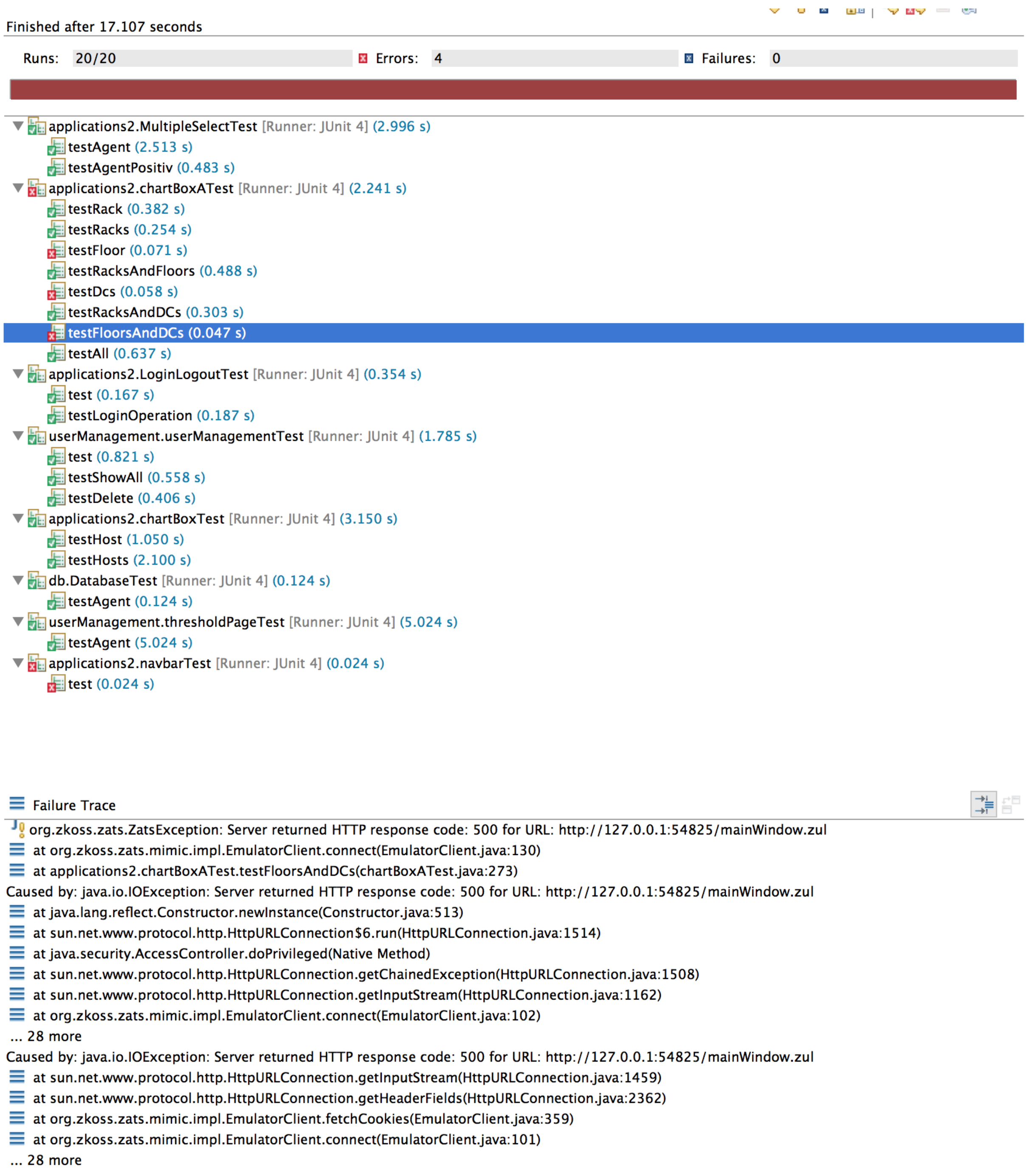


(b) Graphical representation



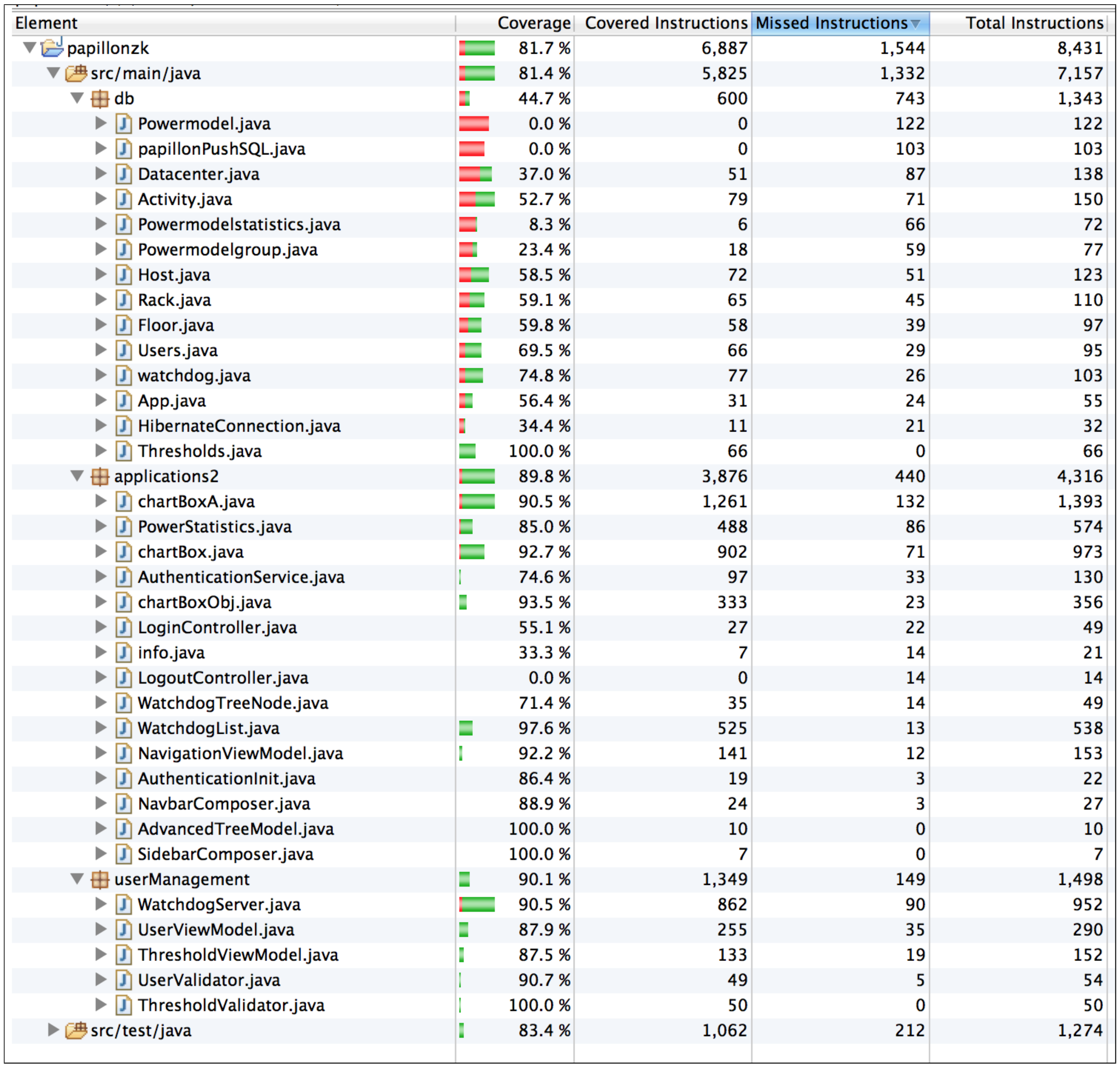
(c) Graphical representation

Figure B.4: List view of the components and graphical representation of the power consumption



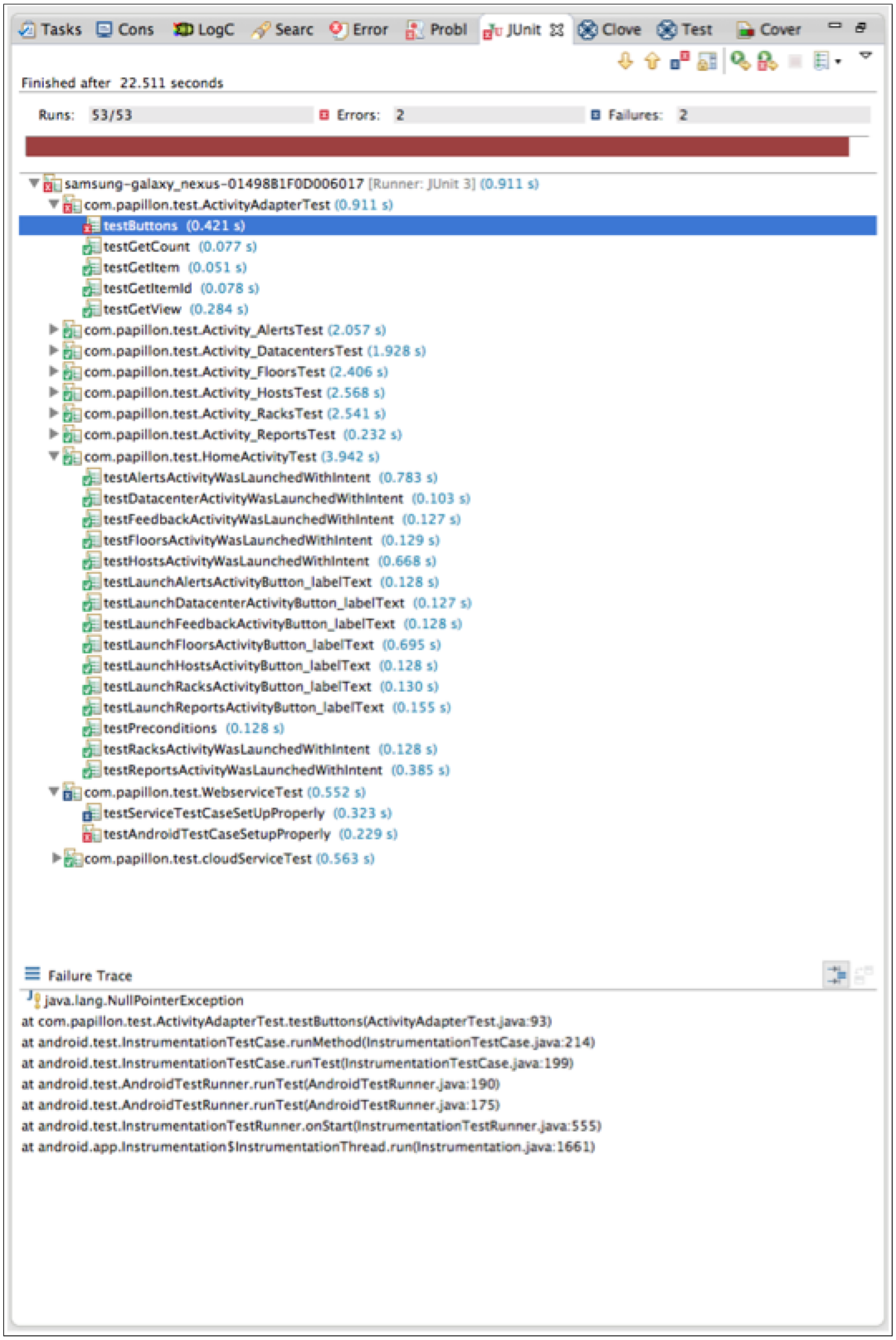
```
Finished after 17.107 seconds

Runs:  20/20          Errors:  4          Failures:  0

applications2.MultipleSelectTest [Runner: JUnit 4] (2.996 s)
    testAgent (2.513 s)
    testAgentPositiv (0.483 s)
applications2.chartBoxATest [Runner: JUnit 4] (2.241 s)
    testRack (0.382 s)
    testRacks (0.254 s)
    testFloor (0.071 s)
    testRacksAndFloors (0.488 s)
    testDcs (0.058 s)
    testRacksAndDCs (0.303 s)
    testFloorsAndDCs (0.047 s)
    testAll (0.637 s)
applications2.LoginLogoutTest [Runner: JUnit 4] (0.354 s)
    test (0.167 s)
    testLoginOperation (0.187 s)
userManagement.userManagementTest [Runner: JUnit 4] (1.785 s)
    test (0.821 s)
    testShowAll (0.558 s)
    testDelete (0.406 s)
applications2.chartBoxTest [Runner: JUnit 4] (3.150 s)
    testHost (1.050 s)
    testHosts (2.100 s)
db.DatabaseTest [Runner: JUnit 4] (0.124 s)
    testAgent (0.124 s)
userManagement.thresholdPageTest [Runner: JUnit 4] (5.024 s)
    testAgent (5.024 s)
applications2.navbarTest [Runner: JUnit 4] (0.024 s)
    test (0.024 s)

Failure Trace
org.zkoss.zats.ZatsException: Server returned HTTP response code: 500 for URL: http://127.0.0.1:54825/mainWindow.zul
at org.zkoss.zats.mimic.impl.EmulatorClient.connect(EmulatorClient.java:130)
at applications2.chartBoxATest.testFloorsAndDCs(chartBoxATest.java:273)
Caused by: java.io.IOException: Server returned HTTP response code: 500 for URL: http://127.0.0.1:54825/mainWindow.zul
at java.lang.reflect.Constructor.newInstance(Constructor.java:513)
at sun.net.www.protocol.http.HttpURLConnection$6.run(HttpURLConnection.java:1514)
at java.security.AccessController.doPrivileged(Native Method)
at sun.net.www.protocol.http.HttpURLConnection.getChainedException(HttpURLConnection.java:1508)
at sun.net.www.protocol.http.HttpURLConnection.getInputStream(HttpURLConnection.java:1162)
at org.zkoss.zats.mimic.impl.EmulatorClient.connect(EmulatorClient.java:102)
... 28 more
Caused by: java.io.IOException: Server returned HTTP response code: 500 for URL: http://127.0.0.1:54825/mainWindow.zul
at sun.net.www.protocol.http.HttpURLConnection.getInputStream(HttpURLConnection.java:1459)
at sun.net.www.protocol.http.HttpURLConnection.getHeaderFields(HttpURLConnection.java:2362)
at org.zkoss.zats.mimic.impl.EmulatorClient.fetchCookies(EmulatorClient.java:359)
at org.zkoss.zats.mimic.impl.EmulatorClient.connect(EmulatorClient.java:101)
... 28 more
```

Figure B.5: JUnit output of the web application



Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
papillonzk	81.7 %	6,887	1,544	8,431
src/main/java	81.4 %	5,825	1,332	7,157
db	44.7 %	600	743	1,343
Powermodel.java	0.0 %	0	122	122
papillonPushSQL.java	0.0 %	0	103	103
Datacenter.java	37.0 %	51	87	138
Activity.java	52.7 %	79	71	150
Powermodelstatistics.java	8.3 %	6	66	72
Powermodelgroup.java	23.4 %	18	59	77
Host.java	58.5 %	72	51	123
Rack.java	59.1 %	65	45	110
Floor.java	59.8 %	58	39	97
Users.java	69.5 %	66	29	95
watchdog.java	74.8 %	77	26	103
App.java	56.4 %	31	24	55
HibernateConnection.java	34.4 %	11	21	32
Thresholds.java	100.0 %	66	0	66
applications2	89.8 %	3,876	440	4,316
chartBoxA.java	90.5 %	1,261	132	1,393
PowerStatistics.java	85.0 %	488	86	574
chartBox.java	92.7 %	902	71	973
AuthenticationService.java	74.6 %	97	33	130
chartBoxObj.java	93.5 %	333	23	356
LoginController.java	55.1 %	27	22	49
info.java	33.3 %	7	14	21
LogoutController.java	0.0 %	0	14	14
WatchdogTreeNode.java	71.4 %	35	14	49
WatchdogList.java	97.6 %	525	13	538
NavigationViewModel.java	92.2 %	141	12	153
AuthenticationInit.java	86.4 %	19	3	22
NavbarComposer.java	88.9 %	24	3	27
AdvancedTreeModel.java	100.0 %	10	0	10
SidebarComposer.java	100.0 %	7	0	7
userManagement	90.1 %	1,349	149	1,498
WatchdogServer.java	90.5 %	862	90	952
UserViewModel.java	87.9 %	255	35	290
ThresholdViewModel.java	87.5 %	133	19	152
UserValidator.java	90.7 %	49	5	54
ThresholdValidator.java	100.0 %	50	0	50
src/test/java	83.4 %	1,062	212	1,274

Figure B.6: Code coverage of the web app



```
Finished after 22.511 seconds

Runs: 53/53        Errors: 2        Failures: 2

▼ samsung-galaxy_nexus-0149881F0D006017 [Runner: JUnit 3] (0.911 s)
  ▼ com.papillon.test.ActivityAdapterTest (0.911 s)
      testButtons (0.421 s)
      testGetCount (0.077 s)
      testGetItem (0.051 s)
      testGetItemId (0.078 s)
      testGetView (0.284 s)
  ▶ com.papillon.test.Activity_AlertsTest (2.057 s)
  ▶ com.papillon.test.Activity_DatacentersTest (1.928 s)
  ▶ com.papillon.test.Activity_FloorsTest (2.406 s)
  ▶ com.papillon.test.Activity_HostsTest (2.568 s)
  ▶ com.papillon.test.Activity_RacksTest (2.541 s)
  ▶ com.papillon.test.Activity_ReportsTest (0.232 s)
  ▼ com.papillon.test.HomeActivityTest (3.942 s)
      testAlertsActivityWasLaunchedWithIntent (0.783 s)
      testDatacenterActivityWasLaunchedWithIntent (0.103 s)
      testFeedbackActivityWasLaunchedWithIntent (0.127 s)
      testFloorsActivityWasLaunchedWithIntent (0.129 s)
      testHostsActivityWasLaunchedWithIntent (0.668 s)
      testLaunchAlertsActivityButton_labelText (0.128 s)
      testLaunchDatacenterActivityButton_labelText (0.127 s)
      testLaunchFeedbackActivityButton_labelText (0.128 s)
      testLaunchFloorsActivityButton_labelText (0.695 s)
      testLaunchHostsActivityButton_labelText (0.128 s)
      testLaunchRacksActivityButton_labelText (0.130 s)
      testLaunchReportsActivityButton_labelText (0.155 s)
      testPreconditions (0.128 s)
      testRacksActivityWasLaunchedWithIntent (0.128 s)
      testReportsActivityWasLaunchedWithIntent (0.385 s)
  ▼ com.papillon.test.WebserviceTest (0.552 s)
      testServiceTestCaseSetUpProperly (0.323 s)
      testAndroidTestCaseSetupProperly (0.229 s)
  ▶ com.papillon.test.cloudServiceTest (0.563 s)
```

```
java.lang.NullPointerException
at com.papillon.test.ActivityAdapterTest.testButtons(ActivityAdapterTest.java:93)
at android.test.InstrumentationTestCase.runMethod(InstrumentationTestCase.java:214)
at android.test.InstrumentationTestCase.runTest(InstrumentationTestCase.java:199)
at android.test.AndroidTestRunner.runTest(AndroidTestRunner.java:190)
at android.test.AndroidTestRunner.runTest(AndroidTestRunner.java:175)
at android.test.InstrumentationTestRunner.onStart(InstrumentationTestRunner.java:555)
at android.app.Instrumentation$InstrumentationThread.run(Instrumentation.java:1661)
```

Figure B.7: JUnit output of the Android app

Bibliography

[1] K. Sen and G. Agha, "CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-checking Tools," in *Computer Aided Verification*, pp. 419–423, Springer, 2006.

[2] M. I. Hossain and W. J. Lee, "A Scalable Integration Testing Approach Considering Independent Usage Pattern of Global Variables," International Journal of Software Engineering and Its Applications, 2013.

[3] P. Huang, X. Ma, D. Shen, and Y. Zhou, "Performance Regression Testing Target Prioritization via Performance Risk Analysis," ICSE, 2014.

[4] X. Ma, B. Yan, G. Chen, C. Zhang, K. Huang, J. Drury, and L. Wang, "Design and Implementation of a Toolkit for Usability Testing of Mobile Apps," Springer, 2012.

[5] C. H. Liu, C. Y. Lu, S. J. Cheng, K. Y. Chang, Y. C. Hsiao, and W. M. Chu, "Capture-Replay Testing for Android Applications," in *Computer, Consumer and Control (IS3C), 2014 International Symposium on*, pp. 1129–1132, IEEE, 2014.

[6] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using GUI Ripping for Automated Testing of Android Applications," IEEE, 2012.

[7] N. Mitchell and G. Sevitsky, "LeakBot: An Automated and Lightweight Tool for Diagnosing Memory Leaks in Large Java Applications," IEEE, 2012.

[8] C. Wilke, S. Götz, and S. Richly, "JouleUnit: A Generic Framework for Software Energy Profiling and Testing," in *Proceedings of the 2013 workshop on Green in/by software engineering*, pp. 9–14, ACM, 2013.

[9] H. Kim, B. Choi, and W. E. Wong, "Performance Testing of Mobile Applications at the Unit Test Level," in *Secure Software Integration and Reliability Improvement, 2009. SSIRI 2009. Third IEEE International Conference on*, pp. 171–180, IEEE, 2009.

[10] D. Gupta, K. V. Vishwanath, M. McNett, A. Vahdat, K. Yocum, A. Snoeren, and G. M. Voelker, "DIECAST: Testing Distributed Systems with an Accurate Scale Model," ACM, 2011.

[11] Y. Cheon and G. T. Leavens, "A Simple and Practical Approach to Unit Testing: The JML and JUnit way," in *ECOOP 2002Object-Oriented Programming*, pp. 231–255, Springer, 2002.

[12] S. Thummalapenta, M. R. Marri, T. Xie, N. Tillmann, and J. de Halleux, "Retrofitting Unit Tests for Parameterized Unit Testing," pp. 294–309, 2011.

[13] J. H. Andrews, T. Menzies, and F. C. Li, "Genetic Algorithms for Randomized Unit Testing," *Software Engineering, IEEE Transactions on*, vol. 37, no. 1, pp. 80–94, 2011.

[14] J. H. Hill, H. A. Turner, J. R. Edmondson, and D. C. Schmidt, "Unit Testing Non-Functional Concerns of Component-based Distributed Systems," in *Software Testing Verification and Validation, 2009. ICST'09. International Conference on*, pp. 406–415, IEEE, 2009.

[15] M. J. Harrold and M. L. Soffa, "Selecting and Using Data for Integration Testing," *Software, IEEE*, vol. 8, no. 2, pp. 58–65, 1991.

[16] M. Mussa and F. Khendek, "Towards a Model Based Approach for Integration Testing," Springer, 2011.

[17] B. Barbieri de Pontes Cafeo and P. C. Masiero, "Contextual Integration Testing of Object-Oriented and Aspect-Oriented Programs: A structural Approach for Java and AspectJ," IEEE, 2011.

[18] J. Shi, M. B. Cohen, and M. B. Dwyer, "Integration Testing of Software Product Lines Using Compositional Symbolic Execution," Springer, 2012.

[19] R. Sasnauskas, P. Kaiser, R. L. Jukic , and K. Wehrle, "Integration testing of protocol implementations using symbolic distributed execution," IEEE, 2012.

[20] S. A. Khan and A. Nadeem, "Automated Test Data Generation for Coupling Based Integration Testing of Object Oriented Programs Using Evolutionary Approaches," in *Information Technology: New Generations (ITNG), 2013 Tenth International Conference on*, pp. 369–374, IEEE, 2013.

[21] M. Lochau, S. Lity, R. Lachmann, I. Schaefer, and U. Goltz, "Delta-oriented Model-based Integration Testing of Large-scale Systems," elsevier, 2014.

[22] S. Yoo and H. M., "Regression Testing Minimization, Selection and Prioritization: A Survey," Wiley Online Library, 2010.

[23] B. Li, D. Qiu, H. Leung, and D. Wang, "Automatic Test Case Selection for Regression Testing of Composite Service Based on Wxtensible BPEL Flow Graph," elsevier, 2012.

[24] W. Jin, A. Orso, and T. Xie, "Automated Behavioral Regression Testing," IEEE, 2010.

[25] A. Nanda, S. Mani, S. Sinha, M. J. Harrold, and A. Orso, "Regression Testing in the Presence of Non-code Changes," IEEE, 2011.

[26] H. K. Leung and L. White, "A Study of Integration Testing and Software Regression at the IntegrationLevel," in *Software Maintenance, 1990, Proceedings., Conference on*, pp. 290–301, IEEE, 1990.

[27] A. Memon, I. Banerjee, and A. Nagarajan, "GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing," ACM, 2012.

[28] F. Horváth, B. Mészáros, and T. Gergely, "Usability Testing of Android Applications," in *9TH CONFERENCE OF PHD STUDENTS IN COMPUTER SCIENCE*, p. 24, 2014.

[29] M. Y. IVORY and M. A. HEARST, "The state of the art in automating usability evaluation of user interfaces," in *ACM Computing Surveys*, pp. 470–516, ACM, 2001.

[30] J. C. Bastien, "Usability Testing: A Review of Some Methodological and Technical Aspects of the Method," elsevier, 2008.

[31] J. Harty, "Finding usability Bugs with Automated Tests," ACM, 2011.

[32] F. T. W. Au, S. Baker, I. Warren, and G. Dobbie, "Automated Usability Testing Framework," pp. 294–309, CRPIT, 2008.

[33] D. Bader and D. Pagano, "Towards Automated Detection of Mobile Usability Issues," emis, 2013.

[34] n. . a. Android Developers, Url = http://http://developer.android.com/tools/help/monkeyrunnerconcepts.h "MonkeyRunner."

[35] n. . a. Android Developers, Url = http://developer.android.com/tools/help/monkey.html/, "UI/Application Exerciser Monkey."

[36] "Robotium." accessed at: July 2014, `https://code.google.com/p/robotium/`.

[37] “Java Path Finder.” accessed at: February 2016, `http://babelfish.arc.nasa.gov/trac/jpf/`.

[38] H. van der Merwe, B. van der Merwe, and W. Visser, “Verifying Android Applications Using Java PathFinder,” in *ACM SIGSOFT Software Engineering Notes*, ACM, 2012.

[39] R. Mahmood, N. Esfahani, T. Kacem, N. Mirzaei, S. Malek, and A. Stavrou, “A Whitebox Approach for Automated Security Testing of Android Applications on the Cloud,” IEEE, 2012.

[40] J. Bo, L. Xiang, and G. Xiaopeng, “MobileTest: A Tool Supporting Automatic Black Box Test for Software on Smart Mobile Devices,” IEEE, 2007.

[41] R. Chandra, B. F. Karlsson, N. Lane, C.-J. M. Liang, S. Nath, J. Padhye, L. Ravindranath, and F. Zhao, “Towards Scalable Automated Mobile App Testing,” tech. rep., Technical Report MSR-TR-2014-44, 2014.

[42] A. Memon, I. Banerjee, B. N. Nguyen, and B. Robbins, “The Frst Decade of GUI Ripping: Extensions, Applications and Broader Impacts,” in *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pp. 11–20, IEEE, 2013.

[43] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and G. Imparato, “A Toolset for GUI Testing of Android Applications,” IEEE, 2012.

[44] M. E. Joorabchi and A. Mesbah, “Reverse Engineering iOS Mobile Applications,” in *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pp. 177–186, IEEE, 2012.

[45] W. De Pauw and G. Sevitsky, “Visualizing Reference Patterns for Solving Memory Leaks in Java,” Springer, 1999.

[46] G. Xu, M. D. Bond, F. Qin, and A. Rountev, “LeakChaser: Helping Programmers Narrow Down Causes of Memory Leaks,” ACM, 2011.

[47] D. Yan, S. Yang, and A. Rountev, “Systematic Testing for Resource Leaks in Android Applications,” IEEE, 2013.

[48] E. O’Driscoll and G. E. O’Donnell, “Industrial Power and Energy Metering a State-of-the-Art Review,” elsevier, 2012.

[49] C. Yoon, D. Kim, W. Jung, C. Kang, and H. Cha, “Appscope: Application energy metering framework for android smartphones using kernel activity monitoring,” USENIX ATC, 2012.

[50] "Google Play Developer Console." accessed at: February 2016, `https://developer.android.com/distribute/googleplay/developer-console.html`.

[51] "iBetaTest." accessed at: February 2016, `http://ibetatest.com`.

[52] "Applause." accessed at: February 2016, `http://www.Applause.com`.

[53] "Testflight." accessed at: February 2016, `https://developer.apple.com/testflight/`.

[54] "Fledge." accessed at: February 2016, `http://www.bianor.com/blog/using-fledge-for-Testing-blackberry-Applications/`.

[55] "TestFairy." accessed at: November 2014, `http://www.Testfairy.com`.

[56] S. Benli, A. Habash, A. Herrmann, T. Loftis, and D. Simmonds, "A Comparative Evaluation of Unit Testing Techniques on a Mobile Platform," in *Information Technology: New Generations (ITNG), 2012 Ninth International Conference on*, pp. 263–268, IEEE, 2012.

[57] B. Sadeh and S. Gopalakrishnan, "A Study on the Evaluation of Unit Testing for Android Systems," *International Journal of New Computer Architectures and their Applications (IJNCAA)*, vol. 1, no. 4, pp. 926–941, 2011.

[58] A. W. Ulrich, P. Zimmerer, and G. Chrobok-Diening, "Test Architectures for Testing Distributed Systems," Proc. of Quality Week, 1999.

[59] V. Garousi, L. C. Briand, and Y. Labiche, "Traffic-aware Stress Testing of Distributed Systems Based on UML Models," ACM, 2011.

[60] "Ranorex." accessed at: February 2016, `http://www.ranorex.com/`.

[61] W.-T. Tsai, L. Yu, A. Saimi, and R. Paul, "Scenario-Based Object-Oriented Test Frameworks for Testing Distributed Systems," in *Distributed Computing Systems, 2003. FTDCS 2003. Proceedings. The Ninth IEEE Workshop on Future Trends of*, pp. 288–294, IEEE, 2003.

[62] M. Bozkurt, M. Harman, and Y. Hassoun, "Testing Web Services: A Survey," *Department of Computer Science, Kings College London, Tech. Rep. TR-10-01*, 2010.