Masarykova univerzita
Fakulta informatiky

Integration of the native Android application testing into Arquillian framework

Diploma thesis

Bc. Štefan Miklošovič

Brno, spring 2013

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Advisor: RNDr. Adam Rambousek

Acknowledgement

I would like to thank Karel Piwko, my leader in Red Hat Inc., for his valuable advises and insights during writing this thesis which helped me to understand the internal working of Arquillian framework a lot better. My thanks are going to Aslak Knutsen from Red Hat Inc. too for the similar reasons. Last but not least, I want to thank Chelsea Cavlovic for her patience, who, as a native speaker, always helpfully answered my English language-related questions.

Abstract

The aim of this thesis is to design and implement an Arquillian container for Android devices, which is used during functional tests of web applications running at enterprise application servers as well as during functional tests of native Android applications which are running locally at virtual or physical Android device.

Keywords

Arquillian, Android, ShrinkWrap, testing, functional, container, extension, JBoss, enterprise, Drone, JavaEE

Identical units tested under identical conditions will not be identical on the final test after being buried under other components and wiring.
Murphy's law

Contents

Chapter 1

Introduction

The result of one's activity in the manner of a tool or an application, which is to be used by him or by others, in order to be helpful and to fulfill the specified aim and before its introduction to production, has to be tested.

In context of the Murphy's law at the beginning of this thesis, we could ironically say that it holds in software engineering very strongly. Software engineering and development of software as such strives for quality in sense of absence of bugs and smooth integration into existing solutions or functionality of the product.

Quality engineering and testing methodologies in Java enterprise environment have been the subject of very vivid discussions. Couple of techniques as unit testing by testing frameworks as JUnit, TestNG and others have emerged and whole concept of programming like test driven development or eXtreme Programming have been propagating widely.

Besides the unit testing as the very basic principle of testing, integration and functional testing is as important as the former one. In the zoo of frameworks and plethora of tools for JavaEE platform, it is important to be sure not only components of the software solution work as expected but also their interconnection, communication and integration with application containers and other enterprise layers is functioning as well.

While trying to doing so, we realize very early that it is not easy task and in lot of cases not even possible. The reason is that we are are trying to bring the whole runtime into our test so we are orchestrating all layers trying to fake the original target environment in which our product will be running. The orchestrating and faking all complex enterprise services as dependency injection, messaging or database performance

and simulating system errors and communication between multiple entities in our software solution is heroic work and it is not possible to simulate exactly the same scenario multiple times and repeatedly.

This thesis aims to improve functionality and integration testing of web and native Android applications on JBoss platform cooperating with Android platform. The crucial role in the functional and integration testing plays Arquillian testing framework.

Arquillian is a tool, which gets rid of aforementioned impossibilities of testing cases. The core principle which stands behind Arquillian is the philosophy of bringing the tests into target runtime and not otherwise - the typical way of dealing with tests so far. This simple switching of point of view makes whole enterprise testing in JavaEE revolutionary.

It is important to realize the essential component without which testing by Arquillian would be meaningless is so called application container or application server. The application container is a place where, roughly speaking, application is deployed and made accessible for users. Arquillian is container agnostic, which means it does not matter against what target environment the application is tested.

Arquillian is not only a tool for rigid testing of enterprise services and technologies but it also provides a way how to make testing via automating web browser in case we are trying to test web application. There are already solutions how to automate web browser such as Selenium - the pioneer of automated testing frameworks. JBoss community developed a tool which is based on Selenium of name Arquillian Graphene. Arquillian Graphene is designed as set of extensions for Selenium WebDriver.

The functional testing by automating web browsers has its counterparts in Android platform, to name a few, Solo or Robotium. This thesis tries to focus on integrating these tools into Arquillian environment and to make testing of Android application as similar as possible so we can gain from advantages of Arquillian in this direction too.

Chapter 2

On software development processes

In this chapter, we are discussing two most widely-known approaches to the development of the software - serial (or sequential) development model and iterative development model. Both models will be described and their pro et contra will be discussed. We will discuss both models in sense of their attitude to testing process or methodology too.

2.1 Serial development process

Serial development model is a traditional software development process. This process is commonly used in the most development projects. The main description of the process is that it is divided into set of activities. When one set is finished, the other starts. When all sets of activities are done, the whole development process is considered to be about its end. The development model of such pattern is also called waterfall process since there is no easy way how to go back in the development and return to some previous stage. There is basically no way how to switch to the older stage or at least it is used to be challenging and more resource, time and finance demanding.

The process has typically five phases as it is shown in the next figure. To name them, they are called: requirement analysis, design, implementation, testing, installation and maintenance.

Requirements analysis: This step is the most important phase of the waterfall model. In this phase, all customer requirements are gathered, involving clear definition of the customer goals and expectations. Requirements analysis is also the place where the understanding of the customer's context and constrains are

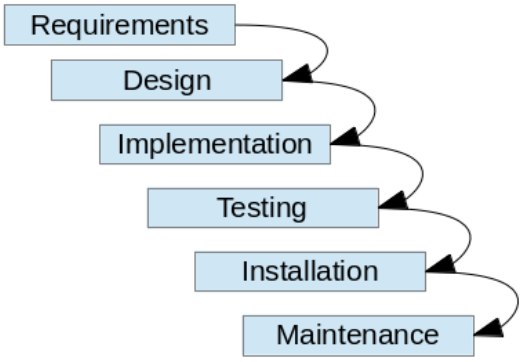

Figure 2.1: Waterfall development process

specified. To obtain all these information, customer is involved into this phase very actively since he represents the only source of information which developers have access to. The outcomes of the analysis are persisted into so called software requirement specification (SRS) of the application. The SRS is describing the behavior of the system entirely and in the great depth and detail. The SRS in used in the next phases actively.

Design: After SRS is available, software architects and designers are about to start the designing the overall look of the software solution. The design phase involves defining the hardware and software architecture, specifying performance and security parameters, designing data storage containers and constraints, choosing the IDE and programming language, and indicating strategies to deal with issues such as exception handling, resource management and interface connectivity. The output of this stage is called SDD - software design description.

Implementation: After the final product is fully designed and all requirements of the customer were incorporated, the implementation of the product takes its place. The input to the implementation phase is the SDD of the system from the design phase. The implementation phase is the logical continuation of the design phase where the product is built.

Testing: Testing phase, which follows the implementation phase, ver-

ifies that the application or the software solution is meeting requirements put on the product in the first phase and additionally that the application is bug free. The test cases are written in order to evaluate the implementation. There are various categories of testing - unit testing, system testing or acceptance testing. If some errors, bug or malfunction is observed, it is up to the development team to refactor the code, incorporate changes and do tests once again until the product is valid and ready to be installed.

Installation: Installation phase actually delivers the developed and tested product to the customer. It can be installed by customer itself, development team or by other third party.

Maintenance: Waterfall model is finished by maintenance phase which follows installation phase at the client site. There is not any functionality added. This phase is used to tweak the product in a such way that its performance is improved and attributes of the product are modified to better reflect clients execution environment.

To sum the sequential (serial) development models, the main idea behind this kind of models is to design and project every single aspect of the final product in very deep details. Only after very careful specification of what is going to be done, further steps are taken, ignoring the client or customer from the development process, when all possible information are extracted. The reason of doing so is to reduce wasted time by avoiding the need to go back and make corrections which are used to be costly.

The main disadvantage of this approach is that the never-go-back mentality of a serial process makes it very hard to maintain and build adaptable and modifiable code in the future. The sequential model tries to avoid changes while the product is being developed.

On the other hand, these types of development processes are suitable for projects where it is known keeping the room for changes in the future is not necessary or not required. In addition, having a precise idea of what application will consist of and what requirements are put on it can uncover notable flaws that might occur in later stages.

2.2 Iterative development process

Iterative development processes can be view as direct opposite of sequential ones. Functionality is added by iterations, from very basic to more advanced, from simple to more complex. In this section, we will discuss two approaches which are very popular among iterative development processes. Tests coders are preparing while they are developing the actual application is the activity which is these days not considered to be only the side effect and the evil which has to be eventually done, but it is a way and strategy how to develop as such. This approach to the developing has of course its name - test driven development (TDD). Test driven development plays very nicely with another very popular term which symbolizes whole principle of coding - eXtreme Programming (XP).

2.2.1 Test driven development

Test driven development is a way of programming when you are writing your tests first and before any single line of code of the target application is written. The first step is to quickly add a test which has to fail upon first run. Only when test is written, you write the application code and run that test against what is written so far to get rid of failing test and have the test which passes. If we run the code and that code is still failing, we have to return back to coding, repair the code and we have to run tests once again. Only when tests are passing, we can write other tests which test different functionality and do the same process again.

The process just described is called TFD (test-first development). Test driven development is understood as a TFD with refactoring. This approach to coding process is completely different from what majority of programmers are used to. The traditional view is to code some subset of the functionality and then to write tests to see if it fail or passes and react accordingly. The importance of getting tests written first is in the way how we think about the design of the application. We are trying to make our design the best possible in advance thinking about how some particular design is going to be helpful for us. TDD is the opposite. We are focusing on new functions so every other functionality added constantly improves our design and just fits our needs.

There are two levels of TDD

- **Acceptance TDD (ATDD):** With this approach, we are writing single acceptance test and just enough production code to fulfill that test. The goal of ATDD is to specify detailed, executable requirements for our solution on a just in time (JIT) basis. ATDD is also called Behavior Driven Development (BDD).
- **Developer TDD:** With developer TDD we write a single developer test, sometimes inaccurately referred to as a unit test, and then just enough production code to fulfill that test. The goal of developer TDD is to specify a detailed, executable design for our solution on a JIT basis. Developer TDD is often simply called TDD.

The result of programming TDD way can be summarized in these points:

- Our development environment must provide rapid response to small changes.
- Unit tests has to run fast (they have short setups, run times, and break downs).
- Unit tests run in isolation, we should be able to reorder them.
- Tests are using real data

2.2.2 Extreme Programming

Extreme programming is one of the several popular agile processes. The extreme programming style of the software development is hard to describe in it's entirety but in order to complete the image of how the programming can be and is done these days, we have to evaluate it.

What is one doing if he develops in the extreme way? To describe his attitude superficially, the first of all, so-called extreme developer starts to code as soon as possible and does not think about the overall design of the application in advance. At the first look, this basic strategy can appear to be dangerous or wrong at all from the academic point of view where sequential development strategies or waterfall-like

strategies are used rather often, but this approach has some undeniable advantages and desirable consequences which are highly valuable and will be discussed.

When developer team starts to work on some project, the very typical beginning is to plan everything into details and the significant amount of time is spent with the customer describing and specifying what output he wants to eventually see. The big problem of this approach is the uncertainty of the output. With the best intentions to catch users needs and requirements, nobody is able to be completely precise about how the final product will look like. It is not the fault of the development team, meaning its disability to understand what the customer wants. Typically, a customer or a consumer of an application even does not know what should be done. The customer is used to have only vague idea what kind of problem the application should solve and he is willing to change the goal, design and requirements as the application is under the construction.

For these reasons, the typical scenario mentioned in the sequential models, which is to consult everything in advance, is flawed and not sufficient. It is often the case that after the customer declares what he wants and the consulting is done, the second time we face the customer is upon final delivery of the product. In the extreme programming paradigm, the customer is the essential part of the developing process and he is participating all the time application is under construction. Of course, one has to design and plan at least something, there has to be clear borders e.g. the problem domain the application will be dealing with, but every further details of how it should be done later are meaningless.

The extreme way of programming is also about doing plans, for sure. We are planning, indeed, but the strongest attitude can be sum up in the slogan "Do the Simplest Thing that Could Possibly Work" and the abbreviation YAGNI which stands for "You Are not Going to Need It".

While the meaning of the first phrase is clear, the mentioned abbreviation has to be clarified. When we are developing the application, we are very often ending up with the design which is over-engineered and after completion of some problem, we have coded infrastructure for solving problems which do not even exist yet and we provided functionality which is not necessary. We likely think that by coding a lot in advance, we will be free from implementing it once there will be the

need to do so. We are keeping doors widely open to hook and modify almost everything or at least with the smallest pain. But the question is - what if that situation will never happen? By implementing the functionality we do not need, we will face two very probable outcomes. The first is the need of taking care of the unused code and pulling it all the development way with us which adds additional complexity to the existing code and by this way it becomes more error prone. The second is the fact that our imagination of what user wants is completely out of range so we are wasting the time.

The key concepts of the extreme programming can be sum in the following points:

- make frequent small releases
- the project is divided into iterations
- simplicity
- no functionality is added early
- refactor whenever possible
- code the unit test first
- integrate often
- all code must have unit tests
- when a bug is found, tests are created

Extreme programming suffer from some drawbacks too. Extreme programming is more code-centered then design-centered so it is more suitable for smaller projects or in projects which do not span over more development teams. Using extreme programming in such scenario would be more counterproductive then useful. Since extreme programming lacks design documentation because it varies constantly, we are short of reuse opportunities. Extreme programming does not maintain quality plan. Having such plan reduces time reserved for testing. Extreme programming spends huge amount of time on testing process.

In spite of mentioned drawbacks, extreme programming can be successful because it focuses on the customer's satisfaction in every moment. We could know that some feature will be needed in the future,

but we are not going to implement it right now, even it is very easy to introduce and implement, because it is not necessary in the present moment. When the time comes, the code can be refactored and missing functions added and it does not matter if we are in the late period of the life cycle of the product. The architecture and design of the project or its part is kept as simple as possible so further modifications and additions can be done over the skeletal design - enabling continuous refactoring.

2.3 Necessity of continuous integration

Both models of software developing we discussed earlier - sequential development process which representative is the waterfall model and iterative development models as TDD and XP, have to integrate the functionality and code into the already existing code base.

While in the sequential development process the integration phase is standalone, long term and time consuming event, it is often only the bad practice and habit why it is so. The key to the successful and painless development scenario is to integrate part by part, code by code and day by day as iterative approach suggests. This approach is rarely taken in case of sequential development. On the other hand, TDD and XP approach is very suitable for so called continuous integration. We will describe what continuous integration (CI) is and how can be helpful in both development models mentioned in the previous sections.

We can explain continuous integration process on the example in context of XP as well as waterfall on the next figure:

The very first necessity in order to do CI is to have a central place where the all source code and resources necessary to build and run the project are stored. It is important to realize that a source code repository should contain everything in order to build the project successfully that is not only source code itself but whole tools, scripts and resources. In the ideal state, we should check out the code from the repository on the clean (developer) machine and run tests, build application and run it from scratch. The most popular distributed source code repository nowadays is Git, so a reader can bear in mind this kind of repository while reading further.

When a developer clone the repository or when he update his local

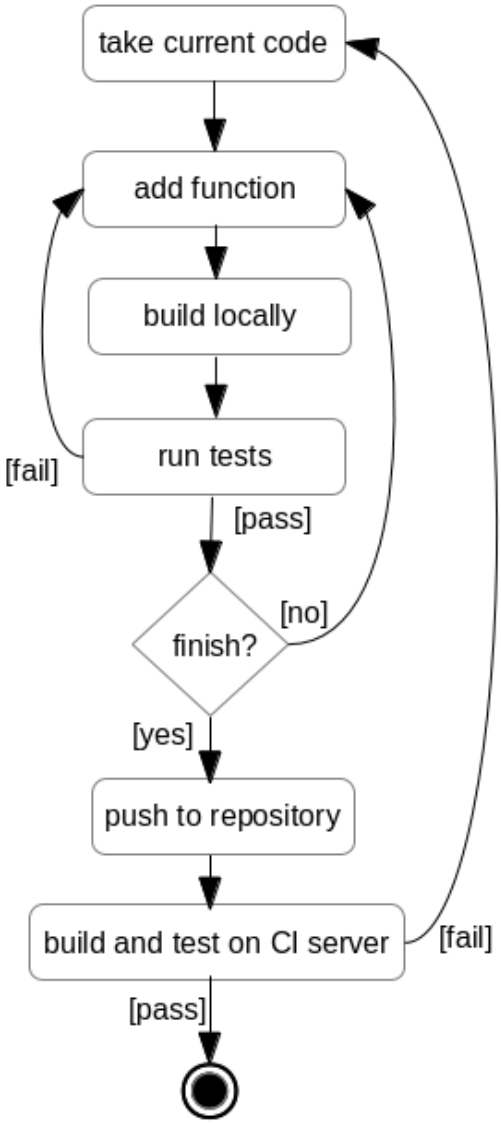

Figure 2.2: Continuous integration

repository to the newest commit to synchronize his local repository with the main repository, he starts to add a functionality into the code. It does not matter what kind of functionality or change it is really, but let's say, for the sake of argument, he is trying to add some business functionality or fulfill some user's story card.

While doing so, he writes the tests, as the TDD approach is advising, builds the project locally, ensures himself that tests which verifies the new functionality are failing and starts to implement what is necessary.

He builds the whole project or module of the application as he develops, checking his progress. He repeats this process while tests are failing. When they stop to do so, it is a sign that the functionality is done and he can commit and push to the remote repository where everybody checks the source code from.

By committing to the remote repository, his work is done only partly. It is so because there is a continuous integration server which fetches the whole repository, build the project and runs all tests again.

The necessity of continuous integration server in determined by distributed nature of the source code repository. More then one developer can check the source code so more then one developer can work on the project simultaneously which means that after both developers commit their work, after resolving possible clashes which originate from the work of both of them, we have to build and test once again since there can be still possibility that some artifacts or resources are not pushed to the repository so build which is run externally from development machine can fail.

Activities related to the continuous integration can be handled automatically or manually. Automatic builds can be executed on some preferred interval or when changes in the main repository are detected.

After getting the initial image of the CI, when one wants to use CI, he needs to build and maintain an infrastructure which: **ref na fowllera**

- contains a single source repository
- automates builds
- makes builds self-testing
- builds the newest code on the integration machine

- keeps builds fast
- tests in production environment

As mentioned earlier, source code can be tracked in Git but other source code repository can be used as SVN or Mercurial.

The automation of the builds can be provided by Java building tool called Maven. The role of the continuous server can be handled by the Java software component called Jenkins.

It is clear that the main stress will be put on the fact that the builds and tests have to be done fast because they are done frequently. If our builds, functional and integration tests took tens of minutes or hours, it would be useless to use the TDD and XP in the JavaEE development. Tests are driving the development. Testing *is* development. We are doing incremental changes, we are adding functionality frequently and piece by piece and we also integrate on-the-fly so we know better every moment how is our project doing which is only hardly doable in the sequential development model when the integration exhibits at the end of the implementation and refactoring of the code is very hard. Continuous integration process encourages to do smaller steps and checking how we are doing, keeping the overview of the project up to date.

We are also trying to do all the tests in the target environment so we need take our tests to the runtime and not otherwise as it was discussed in the previous sections.

Chapter 3

Testing in JavaEE environment

3.1 Traditional testing techniques

3.1.1 JUnit

3.1.2 Concept of mocking

3.2 Functional testing of Java web applications

3.2.1 Selenium

3.2.2 WebDriver

Chapter 4

Arquillian overview

4.1 Introduction

4.2 Core principles

4.3 Principles of the in-container testing

4.4 Anatomy of an Arquillian test

4.4.1 Arquillian descriptor

4.4.2 Arquillian test

4.4.3 Deployment

ShrinkWrap

4.4.4 Test enrichment

4.5 Extensibility of Arquillian

4.5.1 Arquillian container API

4.5.2 Arquillian SPI

Chapter 5

Arquillian extension for Android

5.1 Anatomy of the Arquillian extension

Chapter 6

Design of Arquillian container

Chapter 7

Merging event driven extension into container context

Chapter 8

Implementation of Arquillian Android container

Chapter 9

Testing of JavaEE application with Android container

Chapter 10

Native testing of Android application