Continuous Integration and Testing for Android

Michael Geiss (315702)
Berlin Institute of Technology (TU-Berlin)
June 22, 2012

***Abstract*—Developing software projects gets more complex every day due to an increase of tools, libraries and techniques used in current software development processes. An example is the creation of an Android application that is based on Java development with several extensions in terms of mobile optimization and security. In addition to the scope a project is being implemented by several developers to compensate the knowledge needed for this complexity. However, this causes an increase of problems when integrating the software in the roll-out phase, since all the different artifacts and source code modules need to be combined and merged. In order to improve this process Martin Fowler coined the term Continuous Integration in 1999 and provided a set of best practices to reduce upcoming problems and decrease the time spend for finding bugs and errors in software projects. To seriously apply Continuous Integration there are tools to support this process, i.e. Jenkins or CruiseControl. This paper deals with the presentation of the core artifacts and further features given by these tools and describes in what way they can be applied in general and in particular to the Android build process.**

I. Introduction

A lot of software projects are too complex to be developed by one or two developers but they require at least 5 to 10 of them. This is due to, inter alia, the scope and variety of technologies applied. Although there are several techniques to support the development of team projects (i.e. modularization of software), the integration process of the software gets more complex with each programmer. Possible reasons are the developers using different operating systems, IDEs or another setup of the project environment. Furthermore their source code might make use of different versions of dependent libraries or software, resulting in alternating behavior of the software project on each developer's machine. The same problem is likely to occur in case a developer uses an obsolete version of certain parts of the project. That makes tracking down errors very complicated especially when the all the source code parts are integrated almost at the end of the project. By then the origin of failure is very difficult to be spotted.

Testing software is an important issue within a software development process as well. A lot of developers write their own tests to only verify the functionality of their own modules. However, a lot of problems occur on the border to other modules or when trying to run the project as a whole. Therefor using these personal tests for the whole project is insufficient. Another disadvantage of manual testing is it requiring the user to execute tests by hand. This takes a lot of time and can become very annoying.

Another troublesome aspect is the build process of a software project itself. This process can turn out to be quite complex and its manual execution will lead to a lot of inconvenient actions that need to be performed in order to build project as a whole.

The difficulties mentioned so far can be prevented by following the Continuous Integration paradigm, whose practices are described in *section II*. To support this process there are several tools available. They are covered in *section III* and their generic components are being abstracted. An example that applies to the first three paragraphs in terms of complexity and testing is an android application. Its build process has quite a large scope and requires several steps from its compilation to deploying this application. Due to the use of Java it is also common to divide the project into several modules that need to be tested both individually and as a whole. Therefor this paper deals with the application of CI tools for Android projects in general in *section IV*. This also includes describing the Android build process in *section IV-A*. The last section concludes this paper and gives an outlook on current progresses in terms of Continuous Integration.

II. Continuous Integration

The Problems mentioned in *section I* exist since the beginnings of software development. Even though there were some approaches to deal with these difficulties according to [1] it was an article written by Martin Fowler[1] that publicly discussed this problem. According to [2] *Continuous Integration* (CI) is

> *"[...] a software development practice where members of a team integrate their work frequently [...] Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible [...]"*

In his blog, Martin Fowler illustrates several of these best practices that deal with the integration problems described in *section I*. The following subsections II-A to II-K provide an overview of these approaches according to [2] and [3] that formerly introduced this topic.

A. Maintain a Single Source Repository

Working on larger software development projects includes dealing with a lot of files i.e. source code, pictures, configurations and libraries. Since they change over time and might be altered by different developers it becomes difficult to always

[1]Continuous Integration: http://www.martinfowler.com/articles/continuousIntegration.html, last accessed: June 2, 2012

work on the current version of the project. *Version control systems* i.e. *SVN*, *Git* or *CVS* simplify this process by creating a single source repository that contains the current version of all parts of the project. This way the developers do not have to move the updated files around via obsolete technologies (i.e. email, USB-Stick, or network shared folders) but they only need to update/commit the sources from and to this repository. All commits are and resulting versions are logged and can be reverted. Every single version (so called revisions) of a software project can be tested. Thus, finding errors in a project is far more easy and comfortable by checking since which revision or change an error started occurring.

B. Automate the Build

Building a software development project often is a complex process involving i.e. compilation, downloading dependencies, creating and moving elements in the directory structure or loading schemes into a database. To not risk making mistakes performing these actions the build process can be automated by using build-tools like *make*, *ant*, *maven* or *MSBuild*. They enable a system or user to build and launch a system using a single command to start the corresponding build-tool script. This method also saves time in comparison to the manual execution of a build process.

C. Make your Build Self-Testing

A common approach in software development is *Test Driven Development* [4]. It proposes the principle to write test scenarios for software before implementing the software itself. That way the developer is more aware of the final functions of the product. The test scenarios can also be used as a task list. In terms of Continuous Integration these tests should be automated and executed every time the project is build. This enables the developer to early see whether his latest code produces errors and in what degree the current version meets the requirements specified by the test cases.

D. Everyone Commits To the Mainline Every Day

For the integration process it is important to grasp the dependencies between the different project modules. Often there are dependencies that create errors due to a wrong defined specification or interface. To detect problems between the codes of different users as soon as possible it is essential for the developers to commit their implementations regularly, at least once a day. This way, all developers can use the current code of the others which reduces errors. Furthermore, the system is able to test the project as a whole and thus reveals problems more quickly.

E. Every Commit Should Build the Mainline on an Integration Machine

To prevent developers from working on a faulty version of the project, each commit to the repository should be built and tested on an integration machine. If a commit produces errors the developer is informed to fix this problem. Once the new code builds without any failures the commit is marked as successful and can be checked out by others.

F. Keep the Build Fast

To get a quick feedback on the state of the current build the results from each new commit should the developer as soon as possible. According to [2]

> *"[. . .] a ten minute build is perfectly within reason."*

In general the build process consists of downloading dependencies, compiling the code, and testing it. The optimization of the former is quite difficult. Moreover, running the tests to the software requires most of the build time and that is where an optimization is more useful. Therefor the test scenarios have to be split. The first test unit consists of local unit tests that run quite fast. The second test unit includes larger tests i.e. scaled tests or working with external databases. When building the software the first test unit will be launched right after building on the same machine. However, the second unit needs to be tested on an alternative machine, so that the main server is still available for other user's tests. The developers commit might be approved after a successful run of the first test unit even though the second test could produce errors. This trade off in terms of a correct commit and the notification of the user is worth to be considered.

G. Test in a Clone of the Production Environment

In general, rolling out the software to the production environment results in several problems. They originate from the differences between test environment and production environment. To reduce these problems test environment should be created that is as equal as possible to the production environment. Every aspect that is different could result in an error while rolling out the software project, and the more errors need to be fixed later on. Fixing these might take a lot of time that could be saved. Another problem is that due to differences between test environment and production environment, corresponding automated tests might not work in the latter one which results in manual verification that consumes more time.

H. Make it Easy for Anyone to Get the Latest Executable

Often the executables are not only needed by developers, but by colleagues that want to present the current state to customers or investors. It could also be used for presenting certain features to get a feedback on them. Therefor it is important that the latest executable is stored at a well-known location. This will prevent time wasting search actions and result in faster feedback.

I. Everyone can see what's happening

Communicating the state of the main build line is one of the most important things. That could include displaying the current condition of the build process with corresponding error logs but also statistics on the error rate in commits by a certain developer. The information on the state is essential for tracking problems or keeping track of the progress made over a certain period.

J. Automate Deployment

Continuous Integration involves working with different environments. As described in *section II-F* two or more test environments are reasonable. Therefor the executable needs to be moved to different locations in order to be tested. When having developers that are very eager to commit every hour a day, each time a new test suite is to be run which makes it reasonable to automate the deployment.

K. Benefits of Continuous Integration

In [2] Fowler states, that the *"[. . .] most wide ranging benefit of Continuous Integration is reduced risk"*. That aspect is true for Continuous Integration providing practices that help to detect integration problems very fast. This prevents from *"last-minute hiatus before release dates"* [5]. It is not only the the reduced risk, but also automating several steps in the build process including the test suite that make these best practices attractive to developers and time saving. Developers also profit from early warnings concerning broken code and especially conflicting changes. In case of errors it is now more easy to *" [. . .] revert the code-base back to a bug-free state without wasting time debugging"* [5]. For project managers the CI techniques provide a comfortable way to keep track of the project and detect bottlenecks in the development process.

III. Continuous Integration Tools

To support the Continuous Integration process there are several tools that try to internalize the different practices in it. To demonstrate the elements needed *Figure 1* summarizes the CI practices in a process and depicts the essential artifacts that can be adopted from *section II*. The *Developer* commits the newest version of his source-code to a *Version Control System*. In parallel, the *CI-Server* regularly checks for updates on the Version Control System. If there is a newer version of the source-code available it is checked out by the *CI-Server*. Now the project is being built by a *Build-Tool* that is executed by the server returning its results back to the *CI-Server* where they are evaluated. This information is now put on the *CI-Servers Webpage* to inform the *Developer* about the current condition of the project and whether his commit was successful or included errors.

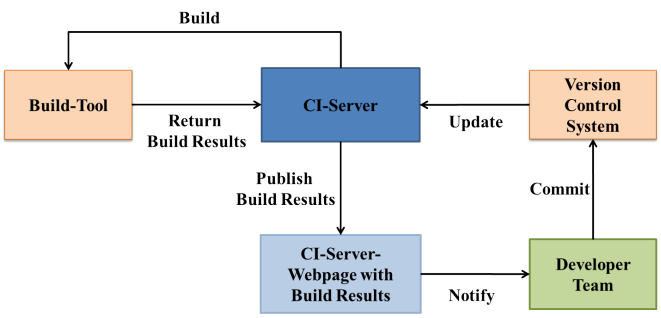

Figure 1. Continuous Integration Tool - Generic Structure based on [1]

In summary, the most important artifacts that CI-Tools need to have in order to enforce the CI practices are given in the following enumeration [6], [5], [1]:

- Version Control System (i.e. CVS, Subversion, or Git)
- Build Tool (i.e. Ant, Maven, MSBuild)
- Support for programming languages or frameworks (i.e. Java, .Net, Android SDK)
- UI for build information
- Notification System (i.e. UI or email)

According to [6], [5], [1] the most common tools are *Hudson/Jenkins*, *CruiseControl*, and *Apache's Continuum*. They all couple the requirements mentioned above and are suitable for working on software projects. Despite their support for these features, some of their differences and characteristics will be described in the following paragraphs:

CruiseControl[2] – CruiseControl is one of the oldest tools around. Still it only offers configuring it via a XML-file. The language support focuses on *Java*, but can be extended to i.e. *.NET* via plug-in.

Apache's Continuum[3] – Continuum is a tool that is mainly for Java developers. In relation to CruiseControl it has quite a short learn curve [5] and it is the only tool offering a role based security. This feature is for setting up which users can have access to certain parts of the project.

Jenkins/Hudson[4] – Jenkins is a fork of Hudson [1], so they use a common base. A comparison of these tools is given in [7]. Both are quite new in the Continuous Integration area. However, they provide a comfortable UI that allows configuring the tools and setting up projects. This makes them both more attractive to new users, as well as the fact that its learn curve is like Continuum's quite short [5]. Jenkins' and Hudson's architecture is plug-in based. Therefor both are easy to extend i.e. in matter of supported languages, build tools and version control systems.

There are further CI tools that can be compared i.e. in [8].

IV. Apply Jenkins for Android Projects

This section is to demonstrate in what way CI tools support the Continuous Integration process. Specifically, Continuous Integration is to be applied to the Android build process which is first going to be introduced. The tool of choice for supporting Android is Jenkins.

A. Android Build Process

The Android build process is depicted in *Figure 2* and involves several steps of compilation and generation to finally create the *.apk* file. This one then can be run under an android device or emulator.

According to [9] the build process is based on a Java build process. Its extension includes compiling the application resource files and generating an *R.java* component to be able to access the resources in Java. This process is done by the

[2]http://cruisecontrol.sourceforge.net/, last accessed: June 08, 2012
[3]http://continuum.apache.org/, last accessed: June 08, 2012
[4]https://hudson.dev.java.net/, last accessed: June 08, 2012

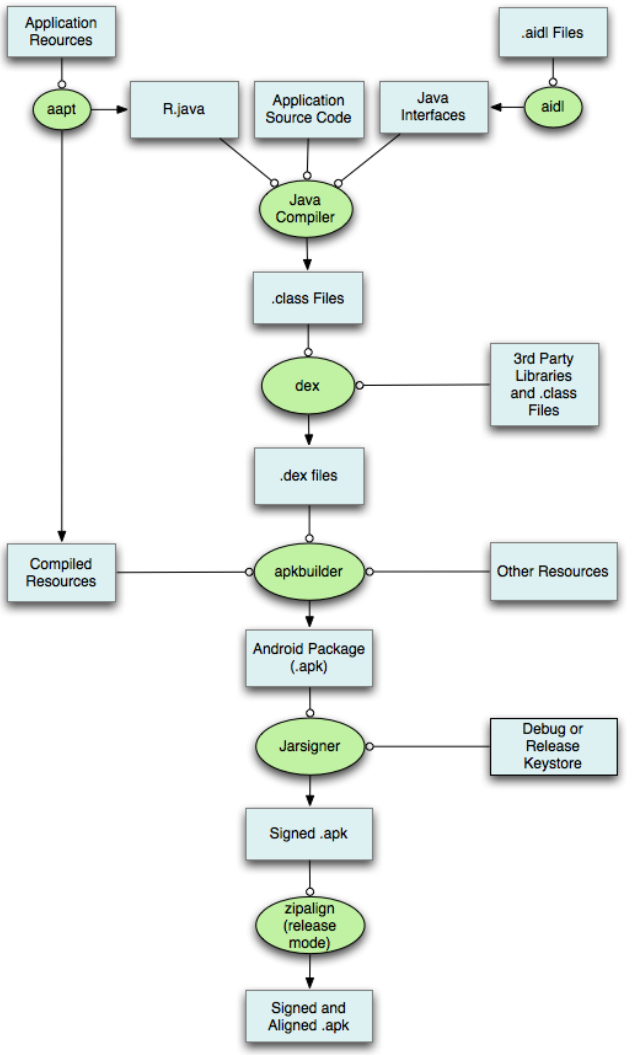

Figure 2. Android Build Process [9]

Android Asset Packaging Tool (aapt). Another tool that was added to Java's build process is the conversion of Android specific interfaces (created with Android Interface Definition Language – AIDL) into Java interfaces. The two new artifacts are now combined with the sources and the Java compiler generates the binaries. In a further step these *.class* files are now being optimized for mobile devices by converting into Dalvic code (*.dex* files). This step involves integrating 3rd party libraries that were used creating the application. The next step in this build process is to create the *.apk* combining the *.dex* files and all non-compiled resources i.e. configuration files or images. Afterwards the *.apk* has to be signed to be able installing it on a device. For that either a debug key or a release key can be used, where the debug key is only for test purposes the release key for rolling the application out on Google's market. The effect of this altered process is being discussed in the following subsection when describing the build management tools.

B. Jenkins Supports the Android Build Process

To support the Android build process with a Continuous Integration tool, this work focused on Jenkins. It offers an easy to use interface and several plug-ins that contribute to the Android build process and its integration. Therefor the path to the Android SDK is required which can be entered via the UI. Alternatively, Jenkins can be set-up to download and use the specific SDK version needed by the project.

For getting the source code Jenkins provides *Version Control System* plug-ins for i.e. Git[5], Subversion[6] or CVS[7] and can be configured to check the repository for update in certain intervals. The build process can be timed in certain intervals as well, and Jenkins provides the possibility to build the code each time a developer commits to the repository.

In terms of *Build Management* the focus lies on Ant[8] or Maven[9]. The UI even offers to download the latest version of each tool which reduces the costs of configuring it. However, when preparing the production environment with a certain version Jenkins allows the user to set the latter one up as well. *Section IV-A* pointed out, that there are several ways a build process can be extended. Therefor it is important that the Continuous Integration tool is also able to handle these adaptations. In general this support can be managed by including certain plug-ins into the build management tool. This also applies to both Maven and Ant. Using one of these build tools the whole process depicted in *Figure 2* can be automated. One feature that needs to be added via plug-in, for example, is the signing of *.apk* files [10].

To support *Automated Tests* Jenkins can be extended via plug-ins. JUnit[10] is available for default Android tests according to [11] but also XUnit tests [12] in case the Android Application was written in NDK. Jenkins can provide a graphical presentation of the test results as depicted in *Figure 3* illustrating the failed test cases and the total number of test cases for each build. Jenkins also informs the user which tests exactly caused problems. JUnit for Android differs a little bit from the JUnit for Java. Thus, it is also possible to perform tests on UI elements with the Android version of this test suite. On the other hand this requires the user to include an Android emulator so the UI tests can be properly performed. This is probably one of Jenkins most advanced features, integrating such a plug-in. Without using an emulator it is not possible to access any Android classes [13]. Next to JUnit there are further test suites that can be integrated in this Continuous Integration tool to serve different user's needs. Monkey is a command line tool that is used for performing stress tests on an Android application by

[5]http://git-scm.com/, last accessed: June 08, 2012
[6]http://subversion.tigris.org/, last accessed: June 08, 2012
[7]http://savannah.nongnu.org/projects/cvs/, last accessed: June08, 2012
[8]http://ant.apache.org/, last accessed: June 08, 2012
[9]http://maven.apache.org/, last accessed: June 08, 2012
[10]http://www.junit.org/, last accessed: June 08, 2012

sending random requests to a device [13] [14]. Jenkins not only integrates this tool but also publishes its results. Another tool that can be integrated in this Continuous Integration tool is Robotium[11] which is an adaptation of Selenium[12] for Android. It provides black box tests that let the user verify UI elements and its actions. Its results can be graphically displayed in the Jenkins UI as well. In addition to these test suites Jenkins allows using shell scripts and installing or uninstalling certain Android packages for test purposes which makes the tool very powerful and neat in terms of testing Android applications.

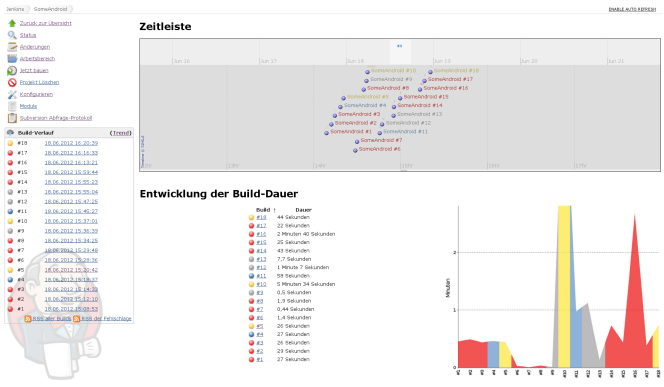

Figure 3. Jenkins UI - Project Trends

As a Continuous Integration tool Jenkins also provides different methods to *visualize the current state* of a project and its build process. First of all it depicts in what degree the last couple builds were successful. Additionally the corresponding console output can be displayed to evaluate and track occurring errors. Another feature offered by Jenkins is showing all the previous builds in a calendar as shown in *Figure 3* as well as some further information. This gives the user an overall view about the whole project. Information on builds can be obtained via mail or rss-feed as well.

V. Conclusion and Outlook

In summary Continuous Integration provides several techniques that help improving and optimizing the development of software. They can make the development more rapid by automating a lot of processes and reduce the need for debugging. Since there are already established best practices the developers should be encouraged to make use of Continuous Integration.

This topic also contributes to current research topics. One approach is *Continuous Delivery* [15] [16] that makes use of the Continuous Integration practices. Its idea is to rapidly develop small features which are then presented to a customer. That way the customer's feedback can fast be integrated in the software project which allows shorter development cycles. This approach reminds of practices performed in *Extreme Programming* [17] projects.

Another current issue to Continuous Integration is its extension with *Cloud Computing* or *Virtualization* [1]. This approach allows parallelizing different steps in the build process. Especially automated tests could benefit from this when applying the practices in *section II-F* where the second unit tests could be performed in parallel to the first unit tests. That way the build process would be executed more quickly and the developers could be notified about the state of their commit even timelier.

[11] http://code.google.com/p/robotium/, last accessed: June 08, 2012
[12] http://seleniumhq.org/, last accessed: June 08, 2012

References

[1] B. Feustel and S. Schluff, "Continuous integration in zeiten agiler programmierung," 02 2012, http://www.heise.de/developer/artikel/Continuous-Integration-in-Zeiten-agiler-Programmierung-1427092.html, last accessed: June 19, 2012.

[2] M. Fowler and M. Foemmel, "Continuous integration, http://martinfowler.com/articles/continuousIntegration.html," 2005.

[3] P. Duvall, S. Matyas, and A. Glover, *Continuous integration: improving software quality and reducing risk*, 1st ed. Addison-Wesley Professional, 2007.

[4] D. Astels, *Test Driven development: A Practical Guide*. Prentice Hall Professional Technical Reference, 2003.

[5] V. Kofman, "The best continuous integration tools," 01 2009, http://www.developer.com/open/article.php/3803646/The-Best-Continuous-Integration-Tools.htm, last accessed: June 19, 2012.

[6] J. F. Smart, "Which open source ci tool is best suited for your application's environment?" JavaWorld.com, 11 2006, http://www.javaworld.com/javaworld/jw-11-2006/jw-1101-ci.html, last accessed: June 19, 2012.

[7] B. Bickel, "Jenkins vs. hudson - time to upgrade," bob-bickel.blogspot.de, 03 2011, http://bobbickel.blogspot.de/2011/03/jenkins-vs-hudson-time-to-upgrade.html, last accessed: June 19, 2012.

[8] M. D. Laudato, "Comparing continuous integration tools," technistas.com, 06 2010, http://technistas.com/2010/06/07/comparing-continuous-integration-tools-part-1/, last accessed: June 19, 2012.

[9] "Building and running," http://developer.android.com/, 06 2012, http://developer.android.com/guide/developing/building/index.html, last accessed: June 19, 2012.

[10] "Signing your applications," http://developer.android.com/, 06 2012, http://developer.android.com/guide/publishing/app-signing.html, last accessed: June 19, 2012.

[11] "Testing fundamentals," http://developer.android.com/, 06 2012, http://developer.android.com/guide/topics/testing/testing_android.html, last accessed: June 19, 2012.

[12] G. Meszaros, *XUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007.

[13] L. Vogel, "Android testing with the android test framework, robotium, monkey and robolectric," http://www.vogella.com/, 03 2012, http://www.vogella.com/articles/AndroidTesting/article.html, last accessed: June 21, 2012.

[14] "Ui/application exerciser monkey," http://developer.android.com/, 06 2012, http://developer.android.com/guide/developing/tools/monkey.html, last accessed: June 19, 2012.

[15] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*, ser. The Addison-Wesley Signature Series. Addison-Wesley, 2010, http://www.informit.com/articles/article.aspx?p=1829417, last accessed: June 19, 2012. [Online]. Available: http://books.google.de/books?id=6ADDuzere-YC

[16] J. Humble, "Continuous delivery vs continuous deployment," http://continuousdelivery.com, 08 2010, http://continuousdelivery.com/2010/08/continuous-delivery-vs-continuous-deployment/, last accessed: June 19, 2012.

[17] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.