fastlane - Continuous Delivery for iOS Apps

Student: Felix Krause

Supervisor: Christopher Casey

A project report submitted in partial fulfillment of the Degree of BSc (Hons.) Software Engineering

17th April 2015

Abstract

Releasing updates for iOS Applications to the App Store is still a manual task, which involves hours of work. This includes creating code-signing certificates, building and signing the app, taking new screenshots, uploading the app metadata to Apple's back-end *iTunes Connect* and submitting the update. All these steps need to be done on a local machine and could not previously be automated.

While web development projects already make use of *Continuous Delivery*, app updates of iOS apps are still released manually, which results in a slower release cycle and more work for developers.

Until now, all tasks require manual user interaction with *Xcode*, the *Apple Developer Portal* or *iTunes Connect*.

This document describes the development of a set of developer tools to automate all the above tasks and make them accessible using the command line. All programs can be run standalone or executed as part of *fastlane*, an application that connects all deployment-related tools into a workflow.

The goal of the *fastlane* tools was to provide a set of command line tools, which can be run on any Mac computer or Mac based server.

It is important to store the information for the app submission on the file system. This way, it is possible to release new app updates just by running a command. When doing a release manually, the developer has to have a lot of background information about the steps required to do a release. The *fastlane* tools support configuration files that contain all required information for the submission process.

The goal of *fastlane*, the tool itself, was to connect all build tools into a structured workflow. It supports different deployment environments (e.g. *App Store Distribution*, or *Beta Distribution*) for various release methods. *fastlane* integrates with all *fastlane* tools and commonly used third party applications.

Table of Contents

Chapter 1: Introduction

Releasing an update as an iOS developer involves a lot of manual tasks. There is no official API from Apple to release new app versions and no existing way to automate this process.

The steps required to release a new update:

- Run unit tests
- Increase the version number
- Create new screenshots for each device and every language
- Upload newly generated screenshots to *iTunes Connect*
- Create a new iOS developer signing certificate (if necessary)
- Create or download the provisioning profile for this app
- Create a new push certificate for the app and upload it to the push notification server
- Build the app and bundle it to an ipa-file
- Sign the *ipa*-file
- Upload the *ipa*-file to Apple *iTunes Connect*
- Update the app description and change log on *iTunes Connect*
- Submit the new app update for review on *iTunes Connect*

All those steps require a user interaction with different applications and websites. The above listed steps are usually completed by an iOS developer, since technical knowledge is necessary.

Since the process is so complex, companies decide to release updates less often. New developers cannot release app updates, since usually only one iOS developer of the team knows about the exact steps of the submission process.

The goal is to have all information required to submit an app update stored on the file system in a version control system, to easily deploy from any machine by any team member.

1.1 Overview

Chapter 2 describes the context of the project and existing work in the area of *Continuous Integration* and *Continuous Delivery*.

Chapter 3 contains the requirements of this project, which also includes the interface of the tools.

Chapter 4 describes the overall architecture of how the tools are set up and how they interact with external services.

Chapter 5 contains a detailed description of how the individual tools work and what alternative approaches are available.

Chapter 6 describes some of the key implementation problems, how they were solved and what different approaches are available.

Chapter 7 goes into detail about how *fastlane* was tested and how it is already used in production systems.

Chapter 8 is an evaluation about the project as a whole.

Chapter 9 contains the plans for future development of *fastlane*.

Chapter 2: Literature Review

2.1 Context

Web services can be updated multiple times a day without causing any downtime. Established companies like (GitHub, 2012) deploy new versions to their production web services up to 100 times a day. This is possible using *Continuous Delivery*: The release process is completely automated and does not require any interaction from the developer.

This project should adapt the concept of *Continuous Delivery* to work with iOS app projects. There are some noticeable differences between a web development project and an iOS app:

Updates for iOS applications need to get approved by Apple, which takes about 7 to 14 days per update. That is why the Continuous Delivery process needs to be adapted to work with Apple's ecosystem. Instead of deploying multiple times a day, it is only possible to release a new update every one or two weeks, because of the Apple review time.

Another difference between web and iOS projects is the deployment itself: on web projects, the developer usually has to deliver the latest version to the production server, migrate the database and clear all caches. iOS projects on the other hand require more steps. For example: taking new screenshots for the App Store, uploading the app metadata to the *iTunes Connect* back-end and submitting a new build on the *iTunes Connect* front-end.

iOS development teams usually already make use of a *Continuous Integration* system, which builds and tests the app after each commit. This is usually implemented using *Jenkins* (self hosted) or services like *Travis-CI*.

The goal of this project is to not only cover *Continuous Integration*, but to offer a *Continuous Delivery* solution. This means it must be possible to distribute new app updates to the App Store.

2.2 Overview of Continuous Delivery

Continuous Integration and *Continuous Delivery* can be achieved using automation: tests and deployments should run completely automatic. The more often a process is executed the more stable and streamlined it becomes.

(Humble and David, 2010):

> In software, when something is painful, the way to reduce the pain is to do it more frequently, not less.

When a user releases an iOS app for the first time, it is a complicated and unclear process. There are many things to consider, like signing certificates, testing and the submission of the *ipa* file. The more often the user releases an update, the easier it becomes. This does not mean the developers should release more updates manually. Instead, programmers should automate the submission process to enable updates every day or even multiple times a day.

2.2.1 Levels of Automation

Often, companies start with a *Continuous Integration* setup, which builds and tests the software when new code was committed. (Minick, 2014) describes the "Continuous Delivery Maturity Model", which defines the different levels of automation. The author goes into detail of the different levels of automation in software companies. For example, developers build the application using their IDE locally, later they build using scripts that are run on its own *Continuous Integration* server. This concept applies to iOS projects as well, since developers usually manually build the iOS app on their local machine and deploy from there, instead of doing so from its own server. The most important information of the paper by (Minick, 2014) is the different levels of automation. An iOS developer team will go through the same phases as any other software project: The developers start automating repeating tasks one by one, until a fully automatic release of the app is possible.

2.2.2 Time required

When the development team detected a bug and wants to release a new version to the production system, there are usually some release steps involved. For web projects this might require updating multiple web servers, restarting them and re-connecting a database server. As the release might be time critical and urgent, it is important to have a stream-lined release process to only require the minimal work from the developer and release as fast as possible.

> How long would it take your organization to deploy a change that involves just one single line of code?

(Humble and David, 2010)

If a bug fix needs to be released, how long does it take to deploy the change to the end user? How long will it take, until all users of the app are using the latest version? This is more difficult for iOS app projects than for web services, since users might not update their apps. Additionally, iOS apps have to wait for a review by Apple when submitting an app update.

2.3 Development Process using Continuous Integration

Some companies have one "release day" each month, where all servers are turned off and a new update is deployed over night (Humble and David, 2010). The authors claim this approach is risky and can cause many errors, such as failures during the deployment and increased workload for servers. They believe the deployment should be fully automatic and only done by scripts to not make any mistakes during the deployment and save developer's time.

The same kind of companies exist for iOS projects as well: They work on a new version about 3-6 months and release a big update, which might end up with a lot of bugs, since there were so many changes introduces since the last version.

While the level of automation of testing and building for iOS projects is similar to regular software projects, *Continuous Delivery* for iOS apps much different, as described by (ship.io, 2012). The author explains how building and signing of mobile apps is different from regular web applications and what problems developers have to solve. The article only goes into detail for beta app distribution, which is not the same as App Store distribution regarding app upload and code signing.

Continuous Integration and *Continuous Delivery* is a key part of an iterative development approach. (Rational Software , 1998) explains the Rational Unified Process for software development projects. During the implementation phase, developers need to make sure to not cause any regressions (breaking a part of the software that already worked before).

A good approach to avoid regressions is to introduce automatic testing and run it after every change in the version control system (e.g. *git*). This way it is clear which developer introduced the regression. Additionally, automated tests help catching newly introduced errors early.

2.4 Continuous Delivery and Version Control

There is a lot of information required during the release process. Not only the information how the software gets deployed, but also the data needed during the deployment (e.g. login information, change logs, etc.). It is good practice to store as much information as possible on the file system in version control, so every employee and computer has access to the required information.

(Humble and David, 2010) explain the two most important principles for reliability in the field of *Continuous Delivery*:

> The repeatability and reliability derive from two principles: automate almost everything, and keep everything you need to build, test, and release your application in version control.

If a company releases updates by hand, no information is stored in version control. This is because a developer deploys manually, not following a certain script or specific order.

(Humble and David, 2010) claim to store everything in version control to be able to roll back to older versions of the software and still be able to deploy a build. Additionally, this enables every team member and server to access the latest information and scripts.

It should be possible for every team member to release a new build of the software without asking how to do so. Instead, the developer should only trigger a script that does the deployment automatically. That way, the company does not depend on one employee who is responsible for a release. This is relevant, since the developer might be sick, on vacation or unavailable for the business.

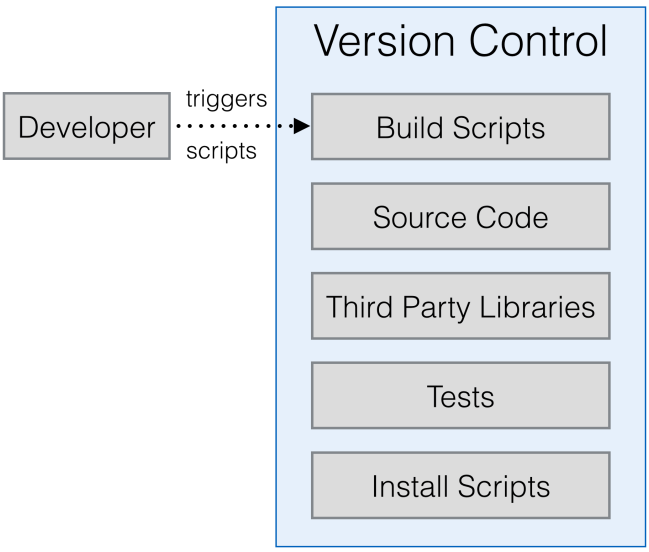

Figure 1: A user only has to trigger the build scripts to deploy a new version of the software project. All information needed for the release is stored in version control.

When all scripts for building, testing and releasing the app are stored in version control, they are accessible for every developer and allos execution on an external *Continuous Integration* server. There is no need to configure the *Continuous Integration* server, since changes in the deployment scripts are automatically pulled from version control.

Storing everything in version control also is a big advantage when introducing new employees to an existing software project. Instead of explaining how to build and how to deploy new versions, the new team member only has to trigger the build or deploy script.

(Humble and David, 2010) explain the ideal workflow for a new employee:

> It should be possible for a new team member to sit down at a new workstation, check out the project's revision control repository, and run a single command to build and deploy the application to any accessible environment, including the local development workstation.

Once the deployment is fully automated, the developer just needs to trigger a script. This should be the same for existing employees and for new ones.

(AccuRev, 2008) confirms that everything needed to deploy a new version, should be stored in version control. This includes third party libraries, property files, database schemas, test scripts and install scripts.

2.5 Continuous Integration for Open Source projects

The *fastlane* project itself is open source and therefore received pull requests from its users. As changes might introduce regressions in other parts of the software, it is important to run automated tests for each change to catch problems early. The *fastlane* project was developed using the "Test Driven Development" approach to ensure a good test coverage from the beginning. The tests are executed after each commit by a *Continuous Integration* service called *Travis CI*. (Deshpande and Riehle, 2008) state the importance of using a *Continuous Integration* system during the development. The authors claim that open source projects which make use of *Continuous Integration* tend to have smaller commits. This enables a faster detection of regressions using automated testing.

The big advantage of hosting source code on GitHub is the social factor of it: Other developers can contribute to the code base and propose improvements and new features. (Vasilescu et al., 2014) evaluates the development approach of using GitHub and a Continuous Integration service like *Travis-CI*. *fastlane* makes use of both GitHub and *Travis-CI*. All tests are executed on a remote virtual machine to guarantee a clean state of the computer.

Every pull request and code change is tested. When an external developer provides a pull request, which breaks some other part of the software, *Travis-CI* will immediately mark the commit as "failed".

It is important to catch newly introduced errors early. This makes fixing them much easier as the developer knows the commit causing the problem.

> This continuous application of quality control checks aims to speed up the development process and to ultimately improve software quality, by reducing the integration problems occurring between team members that develop software collaboratively

(Humble and David, 2010)

That is the gist of *Continuous Integration*: detect regressions early to fix them quickly. For open source projects, this not only applies to team members, but also to external developers contributing to the project using pull requests.

2.6 Conclusion

Some iOS developer teams already make use of *Continuous Integration* using *Travis-CI* or *Jenkins*. The goal with *fastlane* is to upgrade the setup to also deploy and distribute builds to different services, like the App Store. *Continuous Delivery* is established for web development projects. The concept must be adapted to work with iOS projects. There are other constraints to consider, like the Apple review times and the absence of an API to release new app updates.

Chapter 3: Requirement Analysis

To enable a fully working *Continuous Delivery* setup for iOS app, the following requirements must be met.

3.1 Functional Requirements

The functional requirements describe the features that need to be implemented:

- **Communication with Apple's servers:** The configuration and execution of *fastlane* should happen on a local computer. *fastlane* has to transfer all user inputs and configuration to Apple's back-end system (*iTunes Connect* and *Apple Developer Portal*).
- **Controlling the local developer tools:** fastlane should use the local *Xcode* installation and iOS simulator to generate new screenshots, build the app and prepare everything for the upload to the App Store.
- **Configuration of the tools:** Additionally to letting user provide command line parameters, it must be possible to use configuration files. The configuration files should be stored on the file system to store them in version control. More information about the configuration can be found in section 6.4.
- **Interface:** *fastlane* should offer a command line interface to let the developer interact with it. Additionally it should be possible to use the exposed classes directly for more control.
- **Error handling:** When *fastlane* runs into errors (e.g. server unreachable, invalid user input), it should provide a standard Unix exit code. Additionally *fastlane* should provide an error message for the user and display the full stack trace for easier debugging.

3.2 Non-Functional Requirements

The non-functional requirements describes how *fastlane* should behave and work:

- **Sensible Defaults:** Getting started with *fastlane* should be easy and fast for the user. The tools should provide a setup wizard that helps developers getting started. Any information that can be fetched from somewhere automatically should be fetched by *fastlane*. This includes things like automatic detection of the project if it is in the current folder. More information about sensible defaults and configuration can be found in section 6.4.
- **Easy Installation:** Installing the *fastlane* tools should be easy for the user, ideally using a package manager that resolves dependencies automatically.
- **Mac compatibility:** Since iOS developers usually develop on a Mac OS X computer, the tools need to work on Mac OS X.
- **Extensibility:** Developers should be able to extend the functionality of *fastlane* without having to modify the core source code. Instead *fastlane* should provide a plugin-like interface to allow the execution of custom user actions as build steps.
- **Security:** The user's credentials must never leave the local computer. Instead *fastlane* should make use of the built in Mac OS Keychain to securely store and retrieve the credentials. The Keychain is provided by the operating systems and is responsible for storing sensible data.

- **Documentation:** Each tool should have its own documentation describing how to use it. For further documentation, the tools should automatically generate documentation files for all classes and methods based on the source code.
- **License:** The *fastlane* project should be open source under the MIT license. More information about the open source approach can be found in section 8.6.
- **Testability:** The classes should be structured to enable automatic testing. More information about testing can be found in section 5.1.2.

3.3 Alternative Requirements

These requirements did not end up in the final implementation:

- **User Interface:** Instead of providing a command line interface, *fastlane* could also offer graphical user interface to interact. While this would make the initial setup of *fastlane* easier, the time required for implementing this is not worth the effort, as many developers prefer command line tools to graphical applications.
- **Installation:** Instead of using a package manager to install *fastlane* (like *Ruby gems*), *fastlane* could also be installed using a Mac app. The advantage from the user's perspective is the easy installation without having to use a package manager. While this seems like a great solution, other *Ruby gems* could not use *fastlane* since it would not be available via a dependency manager.
- **License:** *fastlane* could also be offered as a hosted service for other developers to use. Developers could register and grant access to the *git* repository of their apps. The main problems with this approach are the additional hosting efforts and security issues. Providing a *fastlane* as a service could earn a monthly revenue by charging subscription from the users.

3.4 General philosophy of the tools

Even for non-commercial projects, usability is important for the success of a software project. The following requirements regarding usability were defined:

3.4.1 Easy configuration

This section explains the goals of the configuration of the *fastlane* tools. More information about the individual tools can be found in Chapter 5:

3.4.1.1 Finding of information

The tools should automatically fetch as much information as possible without requiring any interaction with the user.

Some examples:

- *deliver:* When the user provides an *ipa* file, *deliver* should fetch the bundle identifier and app version number from it. If the user provided this information anyway, *deliver* will compare them to verify they match, since it is very important to use the correct app identifier and version number.
- *snapshot:* The user usually does not have to pass a path to an *Xcode* project. *snapshot* looks for an *Xcode* project in the current directory. If there are multiple projects, it will ask the user to specify which one to use.

3.4.1.2 Default values

All tools provide sensible default values, so users do not have to set every option by themselves. For example:

- *snapshot* needs a list of simulator types to use. By default, *snapshot* uses the standard iPhone simulators. Only if the users want to set their own simulators, they provide this information.
- *sigh* creates App Store profiles by default, since developers usually need provisioning profiles for App Store submissions.

3.4.2 Similar API

Some of the tools provide a configuration file (e.g. *Deliverfile*, *Fastfile*). Additionally to the file-based configuration, the users can pass parameters when executing the tools. More information about the ways to configure the tools can be found in section 6.1.

3.4.3 Easy to install

The tools should be distributed using a package manager to allow the user to easily install it without manually installing the dependencies.

3.4.4 Exposed classes

Especially with the more complex tools like *deliver* is beneficial to expose useful *Ruby* classes to the public. Other developers can then add *deliver* as dependency to access the public classes for interacting with iTunes Connect.

3.4.5 Useful error messages

If something goes wrong, it is more convenient for the user to immediately see why an error occurred and how to fix it than looking up the error code in the documentation. This way, the documentation does not have to include a list of all error codes and messages with information how to fix it.

An example error that requires the user to fix the problem:

```
Beta Testing is not enabled for this app. Open '#{current_url}' and enable
TestFlight Beta Testing.
```

Another example:

If the error is caused by a wrong user input, the tools tell the users about the available options to choose from:

```
Could not find Team with name '#{name}'. Available Teams: #{available}.
```

3.4.6 Transparency

Since most tools are dealing with sensitive data, they should be as transparent as possible:

- The executed terminal commands are shown in the log output
- Logging of what the tools are currently doing, for both local commands and web front-end scripting
- The configuration is stored in text files
- User credentials are stored in the Keychain
- *deliver*: Generation of a PDF report to confirm the updated metadata

Chapter 4: Design

This chapter explains the general approach of how the problem was solved. This includes the choice of programming language, the software architecture and the way the tools communicate with Apple's servers.

4.1 Choice of programming language

Ruby is an open source, dynamic scripting language. *fastlane* is implemented in *Ruby* for the following reasons:

- The resulting project can be distributed as *gem*, which is a package of *Ruby* source code. To install *fastlane*, the user only has to run `gem install fastlane` and it will install *fastlane* with all its external dependencies. The resulting *gem* is hosted on RubyGems, which is the standard way to distribute *Ruby* packages.
- There are a great variety of third party *gems* available to speed up development, like XML parsers, image processing tools and networking libraries.
- *poltergeist gem* to control a headless web browser for accessing the *iTunes Connect* back-end.
- Many other developer tools for iOS apps are based on *Ruby* (e.g. *shenzhen*), which makes it easy to integrate them into *fastlane*.
- Support for *Test Driven Development* using *RSpec, webmock* and *pry*.
- Interactive debugging of code using pry.
- Other libraries can add *fastlane* as dependency and use the exposed classes and methods.
- It is pre-installed on Mac OS.

4.1.1 Alternative Programming Languages

The language itself is not as important as the available third party libraries to interact with Apple websites.

Some alternative programming languages and their advantages and disadvantages:

4.1.1.1 Python

Python is a scripting language with similar features to *Ruby*. It offers a package manager (*pip*) to distribute projects and supports *Test Driven Development*. There are open source libraries available to support XML parsing and front-end scripting. While *python* is pre-installed on Mac OS X, the package manager "*pip*" is not.

The main reason for deciding to use *Ruby* instead of *Python* was the existing iOS developer tools, which are mostly based on *Ruby*.

4.1.1.2 C#

C# cannot be executed on a Mac OS X system. Since every iOS app developer has to use a Mac, Mac OS X has to be supported.

A C# application would be distributed as binary. The user does not use a dependency manager to install the application, which makes updates of the application more tedious.

4.1.1.3 Java

Java is a programming language that supports different operating systems without having to write platform specific code. In contrast to *Ruby*, the source code is compiled into an intermediate language, which is then interpreted by the *Java Virtual Machine*. This allows catching errors on compile time, rather than on run time. While *Java* fulfills most requirements, there are some reasons why this project was not implemented in *Java*:

- *Java* is not pre-installed on any operating system and requires the user to install it
- *Java* is generally slower to develop compared to scripting languages like *Ruby*, mostly because the developer does not have to re-compile *Ruby* code after each change.
- *Java* is more verbose than *Ruby* regarding class and method names

4.2 Architecture

Each tool solves one task (as seen from the user's perspective). Instead of providing one big tool, capable of doing everything, *fastlane* makes use of individual *Ruby gems*. This enables very flexible releases of new gem versions and allows a clear separation between the different code bases.

This results in shared code, like the login on the Apple front-end or accessing the iTunes Search API. This code was moved into its own *Ruby gem* named *fastlane_core*. Each tools has the *fastlane_core gem* as a dependency. This makes future maintaining easier, since the shared code only needs to be changed once in case something needs to be updated.

fastlane itself is its own *Ruby* gem, which launches the other tools and third party integrations. It has a dependency to all *fastlane* tools (*deliver*, *snapshot*, etc.) and some third party tools, like *shenzhen* (Thompson, 2014a).

Additionally to the *fastlane_core*, *fastlane* uses a second shared *Ruby gem*, named *CredentialsManager*. It is responsible for returning the user's Apple ID and password from the Keychain. In case the login information does not exist yet, it will ask the user for it and display information about how it is stored and what it is used for.

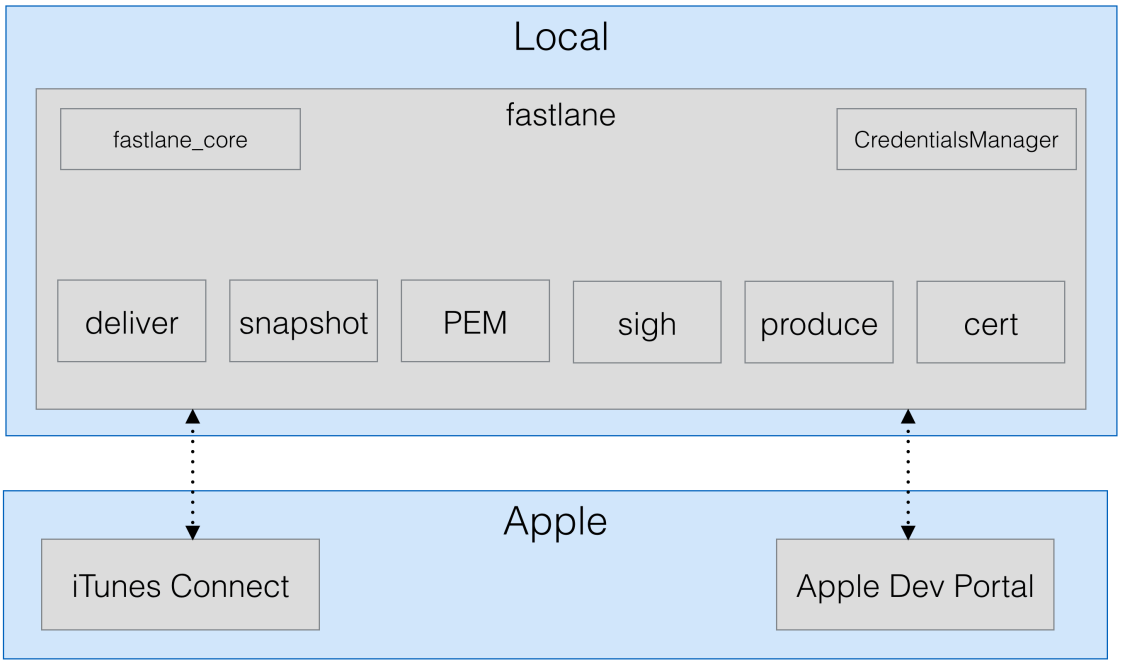

Figure 2: The graph shows the overview of the used Ruby Gems. *fastlane* itself can be seen as a parent *gem*, triggering the other tools and making sure they succeed. The individual tools usually communicate with Apple's servers shown on the bottom of the graph.

More information about *fastlane* can be found in section 5.9.

4.2.1 deliver

The main problem to solve is interacting with the Apple developer tools (e.g. `xcodebuild`, *instruments*) and Apple's servers (e.g. *iTunes Connect*, *Apple Developer Portal*), since there is no official public API available.

For *deliver* it was necessary to upload a large number of screenshots to *iTunes Connect*. Apple provides a tool to upload app metadata, which can also be used to upload new screenshots: *iTMSTransporter*. The documentation of this tool is outdated and only available for registered Apple developers.

deliver makes use of this tool to upload the screenshots and app metadata information (e.g. app title, description and change log). *deliver* provides an easy to use interface to modify the app metadata and upload it to *iTunes Connect*. Under the hood *deliver* stores the provided information in an `XML` file, which matches the undocumented *iTMSTransporter* requirements.

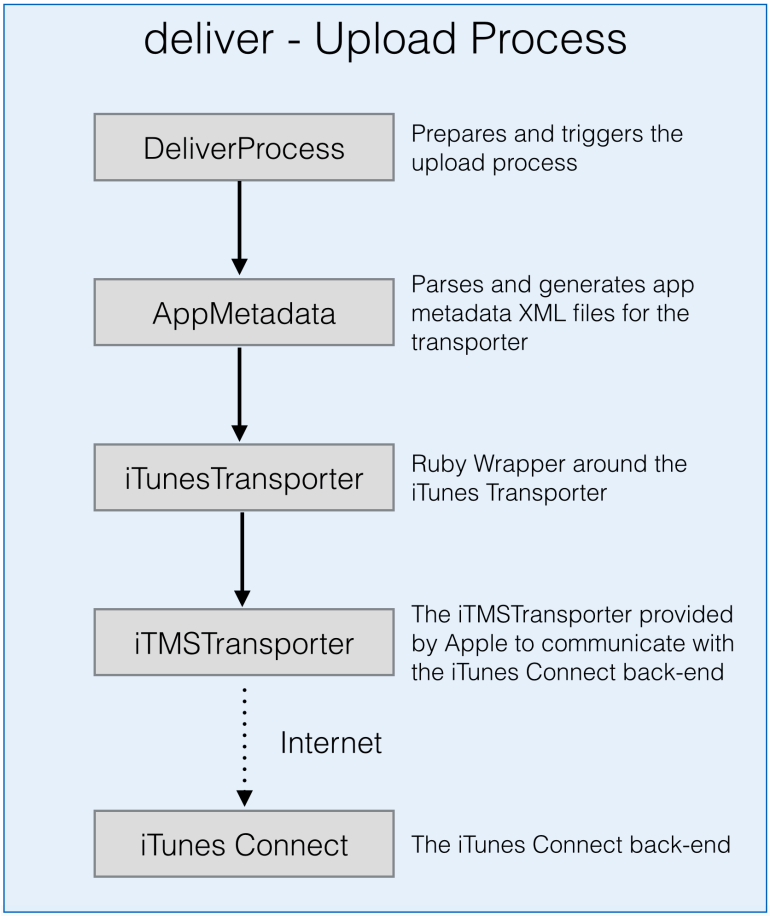

Figure 3: The graph shows the classes used to upload new app metadata to iTunes Connect.

Since not all information can be uploaded this way, *iTunes Connect* scripting was also necessary. *deliver* makes use of *poltergeist* to update the missing information (app icon, app review information, etc.).

It was necessary to use a JavaScript enabled browser, since *iTunes Connect* makes use of *AngularJS*, which modifies data before it gets submitted to Apple.

deliver uses a so called *Deliverfile*, which is a standard *Ruby* file that contains all information needed to deploy a new version to the App Store.

Example *Deliverfile*:

```
screenshots_path "./screenshots"

title({
  "en-US" => "Your App Name"
})

changelog({
  "en-US" => "iPhone 6 (Plus) Support"
})

copyright "#{Time.now.year} Felix Krause"

automatic_release false

primary_category "Business"
secondary_category "Books"

price_tier 5

ipa do
  system("cd ..; ipa build") # build your project
  "../fastlane.ipa"
end

success do
  system("say 'Successfully submitted a new version.'")
end
```

The *copyright* line contains a dynamic string, which will always use the current year. The *ipa* part does not contain a simple value, but a whole *Ruby* block, which gets executed, as soon as *deliver* needs the path to the *ipa* file to use.

This makes the *Deliverfile* really flexible and customizable. More information about the *Deliverfile* can be found in section 6.1.

4.2.2 Other tools

The general architecture of the remaining tools is quite similar to *deliver*. More information about the other tools can be found in Chapter 5: .

4.2.3 fastlane

The goal of *fastlane* was to connect all other tools into one workflow. The user should be able to define which actions to run in which order and *fastlane* will execute them.

It is common to build the iOS app for different environments (e.g. App Store distribution, Beta distribution, Testing only, etc.). *fastlane* supports any number of environments defined as *lanes*. The user can declare a *lane* and the actions that should be executed in the particular environment.

fastlane itself contains over 40 built-in actions (including third party tools). Using a command provided by *fastlane*, users can easily add their own actions, which they can later share with the community.

Similar to *deliver*, *fastlane* stores all information in *Ruby* files, to easily deploy from any computer. (More information about the configuration files can be found in section 6.1)

The main use case of *fastlane* is to run on a *Continuous Integration* server. *fastlane* can be used for both *Continuous Integration* and *Continuous Delivery*.

4.2.4 Fastfile

To configure *fastlane* the developer creates a *Fastfile*. Below is an example configuration file:

```
before_all do
  cocoapods
  xctool
end

lane :test do
  snapshot
end

lane :beta do
  version_bump
  sigh
  ipa
  deliver :beta
end

lane :appstore do
  version_bump
  snapshot
  sigh
  ipa
  deliver
end

after_all do
  slack
end

error do
  slack
end
```

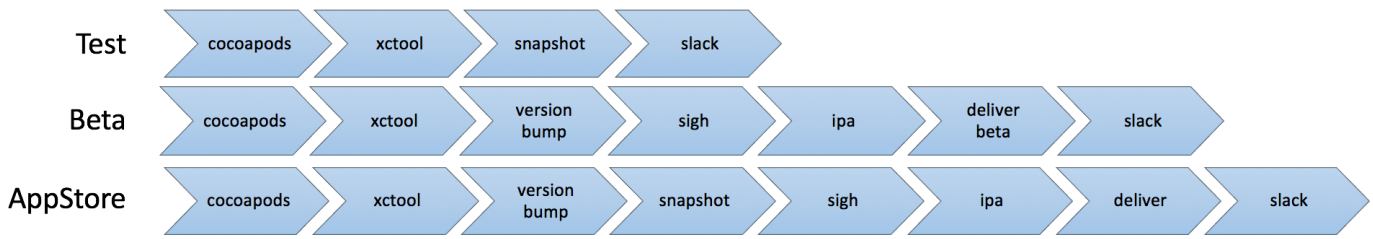

Figure 4: The different lanes with their actions in the given order, configured using the *Fastfile*

As seen in the example code and graph, different actions are executed when running fastlane in different environments. The user can choose which *lane* to run. *fastlane* will take care of running the defined actions in the correct order. If one build step fails, the whole *lane* gets aborted and the *error* block is called.

fastlane will automatically pass values from one action to the next ones by providing a shared space to store values. For example, after creating new screenshots using *snapshot*, they are passed on to *deliver* to upload them to the App Store.

The user only has to define the order of the build steps and not details about the implementation about each of the tools.

fastlane provides a *before_all* block, which is executed before starting any *lane*. This can be used to set up the project or run automated tests.

After a *lane* was successfully executed, the *after_all* block can be used to send a success message to communication services (like Slack) or run any other actions (e.g. custom scripts)

If one build step failed, no further actions gets triggered. Instead the error block is run to send or display an error message.

The above shown example *Fastfile* was simplified. There are more options that can be set for each action:

```
lane :test do
  sigh :adhoc
  slack({
     message: "Successfully released update",
     success: true
  })
end
```

The *Fastfile* makes use of *Ruby's* dynamic parameter options: the user can simple pass a *symbol* (the `:adhoc`) or a hash with any number of key values. The parameter will be passed to the called action.

Chapter 5: The tools

fastlane consists of a set of tools, each described below.

The first line of each tool is the "tag-line" which is used as a one-sentence description on GitHub.

5.1 General

A general description of problems and approaches that apply to all implemented tools.

5.1.1 Transfer App Metadata

Almost all *fastlane* tools need to access or modify data stored on either *iTunes Connect* or the *Apple Developer Portal*.

There are multiple relevant approaches to interact with the Apple services.

5.1.1.1 Update data using the `iTMSTransporter`

The *iTMSTransporter* is bundled with *Xcode* and allows the upload of general app metadata and screenshots to *iTunes Connect*. This approach is very clean and well supported by Apple. It supports features like automatic re-connects after the Internet connection was interrupted and resuming of uploads.

Unfortunately, the *iTMSTransporter* was not updated for the latest changes of *iTunes Connect*, which means, not all metadata can be uploaded using this tool. Additionally, it is not possible to create a new version or to submit a new build using this tool.

While this approach is used by *deliver*, it is not possible to use it for all required actions. *deliver* has to combine this solution with other approaches.

5.1.1.2 Update data by controlling a web browser

A headless browser is a web browser controlled by a script or program, accessing normal web pages. In contrast to a non-headless browser, a headless browser does not render the web pages to the display.

With a tool like *poltergeist*, *deliver* is able to control a headless web browser. It is very natural to use this approach, since the controlled browser triggers the same actions a user would. If something goes wrong during the script (e.g. an element is not accessible) *deliver* will automatically generate and show a screenshot of the current state of the web browser to make debugging easier.

The main problem with this approach is the front-end scripting itself: When the design or layout of the website changes, the script will eventually fail.

Additionally, there is much traffic and performance overhead involved when using the web front-end to transfer app data. This includes transferring the HTML, CSS and JavaScript code for each page.

Error handling is less flexible when using this solution, since the error messages are shown somewhere on the website, and not easily accessible for the script.

5.1.1.3 Update data using a HTTP client

Using a standard HTTP client in *Ruby* it is possible to interact with the Apple front-end. This would be easy if Apple provided an API to access and modify data. Unfortunately that is not the case.

iTunes Connect provides an unofficial JSON API used by the *AngularJS* based front-end. During this project, (Krause, 2015a) released an unofficial documentation of the iTunes Connect JSON API.

The *Apple Developer Portal* does not feature a JSON API. It is based on simple web forms with standard *GET* and *POST* actions.

5.1.1.4 Result

deliver makes use of the *iTMSTransporter* approach to upload the large amount of data (screenshots and *ipa* file).

For all other actions, *fastlane* makes use of the headless web browser approach. This allows faster development and easier debugging, as the interaction happens naturally. This approach also allows generating of screenshots of the web site if something goes wrong.

In the future it makes sense to make use of the HTTP client approach. Using a HTTP client improves testability and makes the script's execution time faster.

5.1.2 Automated Tests

deliver and *fastlane* were developed using the Test-Driven-Development approach, which means, the unit tests were written before actually implementing a new method.

5.1.2.1 Testing framework

The tests were written with *rspec*. It supports an easy to read syntax and provides useful helper methods.

Below an example test:

```
it "returns true when building in Travis CI" do
    stub_const('ENV', { 'TRAVIS' => true })
    expect(FastlaneCore::Helper.is_ci?).to be(true)
end
```

The above shown example test verifies the behavior of the *fastlane_core* when running on Travis CI: When the environment variable *TRAVIS* is set to true, the method `Helper.is_ci?` should return true.

5.1.2.2 Internet Connection

fastlane makes use of external web services, including the iTunes Search API, the iTunes Connect API and the *iTMSTransporter*. The unit tests should not require an Internet connection, but at the same time, the web related code should still be tested. *fastlane* makes use of HTTP stubbing to simulate server responses when running tests.

Automated tests should not depend on a working Internet connection for various reasons:

- A test should not fail because of a slow or unstable Internet connection, as there was no error in the code base, but in the connection.
- The tests should not fail if an external web service is offline.
- The tests are much faster when emulating the server responses locally.
- Some web services require an API key, which is used to limit the number of requests. This can cause problems when running tests multiple times in a short period of time.
- *fastlane* accesses account specific data on iTunes Connect and other Apple web services. Almost all API calls require a valid Apple account, which cannot be stored publicly in the *git* repository.

To simulate the server responses, *fastlane* uses *WebMock*, a *Ruby* gem to easily stub HTTP requests.

As a result, all tests work completely without an Internet connection and do not depend on any external web services.

5.1.3 Documentation

Other *Ruby* Gems can add a dependency to a *fastlane* tool. This is useful for similar tools that want to access existing functionality of *fastlane*. The projects provide an automatically generated documentation of the available classes and methods to make it easier for third party developer to integrate the *fastlane* tools.

fastlane uses *yard*, an open source tool to generate documentation based on the source code. It uses a *JavaDoc*-like syntax to add more information to a module, class or method in the source code.

An example documentation for a method used in *deliver*:

```
# Adds a new locale (language) to the given app
# @param language (FastlaneCore::Languages::ALL_LANGUAGES) the language you want to add
# @raise (AppMetadataParameterError) is raised when you don't pass a correct hash with language codes.
# @return (Bool) Is true, if the language was created. False, when the language already existed

def add_new_locale(language)
...
```

The documentation can be generated by running *yard* using the command line. The configuration of the tool is stored in the *.yardopts* file in the root of the project:

```
yardoc --no-private lib/**/*.rb
```

The configuration file contains the following information:

- Only generate a documentation for public classes and public methods
- A path to the *Ruby* files, that should be used

yardoc provides a service, that automatically generates and updates the documentation after each change in the code base. It is then hosted on *RubyDoc.info*.

5.1.4 Logging

All tools communicate via the terminal with the user. To provide a good user experience, there has to be a way to show different messages depending on the use case. Often the user wants to only see the most important messages to see the progress of the tools.

When there is an issue with the tools, the user must be able to show more messages to get details about what the tool is doing and what part is not working.

By providing different logging levels it is possible to show a colored output to the user, making it easier for the user to distinguish non-important log messages and errors.

The possible solution for the *fastlane* project are:

5.1.4.1 Write a logger class

fastlane_core could provide a logger class used across all tools that supports different outputs like terminal display or file stream. When logging a message it displays the message and adds additional information to it, like the time it was generated.

5.1.4.2 Use the built-in Ruby Logger class

Ruby provides a logger class that supports the most important features a developer needs for a logger. It has 5 log levels integrated (*Debug, Info, Warn, Error* and *Fatal*) and supports logging directly to the terminal output and to files.

While writing a logger class solves this problem, the project makes use of the built-in logger as it is provided by *Ruby* without any extra work.

To support a colored output depending on the log level, *fastlane_core* provides a block configuring the output to display the current time and the log level.

5.2 deliver

Upload screenshots, metadata and your app to the App Store using a single command

deliver is responsible for uploading app metadata, screenshots and `ipa` files to *iTunes Connect*. It is configured using a *Ruby* based configuration file, called *Deliverfile* (An example *Deliverfile* is available in section 4.2.1). Once configured, the user only runs *deliver* using the command line without specifying any additional parameters to upload new metadata or a new app update.

deliver is able to upload all relevant app metadata, like app title, description, keywords, app icon, price tier, app category and app review information. A full list of available options can be found on the official documentation on GitHub by (Krause, 2015b).

deliver uses the *iTMSTransporter* to upload the most basic app metadata (this includes app screenshots, description and keywords). Some information cannot be uploaded using the *iTMSTransporter* for example the app icon, creation of new versions and submission of apps. This is why also a headless browser is used for the missing actions.

5.2.1 Alternative approaches

The app metadata could also be submitted using the unofficial JSON API (Krause, 2015a) provided by iTunes Connect, which is used by the *AngularJS* based web front-end.

The binary still needs to be uploaded using the *iTMSTransporter*.

5.3 snapshot

Automate taking localised screenshots of your iOS app on every device

Apple requires different screenshots for each device type and each language. Per language/device combination, Apple recommends uploading 5 screenshots. The goal is to have screenshots showing the same content for each language and device type, just on a different devices and languages. Up until now, screenshots were created manually, either by using the iOS Simulator on the Mac or a real iPhone or iPad.

snapshot makes use of the built in technology *UI Automation*, which is offered by *Instruments*. *snapshot* runs through all enabled languages and device types and triggers a *UI Automation* script for each run. The *UI Automation* script is provided by the developer. After running the script, the generated screenshots are copied over to the correct folder, renamed to contain the device type and language. When all runs are completed *snapshot* generates a HTML based summary to get an overview of all screenshots.

The big advantage of this approach is a combination of being able to interact with the app the same way a user does (tapping on elements) and testing the app while creating the screenshots. In the scripts, the developer can add his own *asserts* to make sure, the user interface looks and behaves as expected.

More information about the technical challenges of *snapshot* can be found in 6.3.

5.3.1 Alternative approaches

5.3.1.1 Using iOS code only

Instead of interacting with the app from the user's perspective, the developer could add interaction code right inside the app, which is only executed in test builds. This approach has the disadvantage of being tightly coupled with the app's source code and the general architecture. It is more time consuming to create those kinds of tests compared to *UI Automation* JavaScript code, since it requires setting up the individual views. On the other hand it allows easy pre-filling of the app's content using *Objective C* or *Swift* code.

There is an open source library by (Sutherland, 2014) available that helps with this approach.

5.3.1.2 Using the same screenshots for different languages and device types

Some companies decide to create one screenshot for all device sizes and languages. They change the size of the image to pass Apple's image validation.

While this is possible and can be used, it is a bad practice. Users expect screenshots to match their native language. When using the same screenshots for different device types, they end up looking blurry and the text looks too small or too large.

5.4 frameit

Quickly put your screenshots into the right device frames

The screenshots generated by *snapshot* (see section 5.3) are usually used for the App Store. To use the screenshots on websites or print media, it is common to add device frames around the screenshots. This is usually done by hand using an image editing software.

The goal of *frameit* is to provide a simple command line tool that automatically detects all screenshots in the current directory and adds device frames around them. The tool makes use of the official device images provided by Apple. Those images cannot be bundled with *frameit* because of Apple's licensing.

To work around this issue, *frameit* will ask the user to follow a quick interactive setup to download and extract the stock images from the Apple website. It helps the user by opening the browser window and by assisting the setup process.

frameit makes use of the *imagemagick* library to convert the *Photoshop* images provided by Apple to *png* and to insert the actual screenshots into the device frame.

5.4.1 Alternative approaches

5.4.1.1 Provide device images

As it is still some work for the user to set up the initial device screens on the first launch of *frameit*, it would make sense to provide the images with the library.

This would require a designer to create custom device frame images, which need to be updated after every release of new Apple devices. Also, providing the images with the library would drastically increase the size of the library, resulting in much longer download and installation times for the user.

5.5 PEM

Automatically generate and renew your push notification profiles

Push notification profiles are needed for a web server to send push notifications to the users of an iOS app. *PEM* creates a new signing request, which is uploaded to the *Apple Developer Portal* to create a push ID. The generated file is then used with the private key to generate a *.pem* file, which can be uploaded to the web server.

5.5.1 Alternative approaches

There is no other known way to automate this process using *Xcode*, the Keychain or a different technique. When doing this procedure manually, the user has to first export the certificates from the Keychain and then run terminal commands to generate a *PEM* file.

5.6 sigh

Automatically create, maintain and repair provisioning profiles

To sign and distribute an iOS app, it is necessary to use a valid provisioning profile. One provisioning profile is used for one App ID (can be created by *produce*, see section 5.8) for one code signing identity (can be created by *cert*, see section 5.7).

The process of creating a provisioning profile is all based on the *Apple Developer Portal* web front-end:

- Login and navigate to the provisioning profiles
- Select the type of the profile (Development, Ad Hoc or App Store profile)
- Select the App ID and the code signing identity
- Name and download the profile

The most complicated part of implementing this tool was the downloading of the resulting profiles. More information about file downloads from the *Apple Developer Portal* can be found in section 6.2.

5.6.1 Alternative approaches

Besides the used technique, there is only one way to generate a provisioning profile. This feature is built into *Xcode* and can create, download and maintain the user's provisioning profiles in just one click. While this sounds promising, it will revoke all existing certificates, making it unusable for production setups. To trigger the *Xcode* sync, it is required to interact with *Xcode* using the mouse, which is not something that can easily be automated.

5.7 cert

Automatically create and maintain iOS code signing certificates.

cert is used in combination with *sigh* to prepare everything required to sign an iOS app. *cert* is responsible for creating and maintaining the iOS code signing identity.

The key challenge was to compare the identities available online with the identities available in the Keychain of the computer.

To list all available code signing identities, the Keychain offers a command:

```
security find-identity -v -p codesigning
```

which returns the *UDID*s and names of the available signing identities:

```
1) 07460002E9BEDA84C6129BD47... "iOS Development: Felix Krause"
2) 7D8F558757A4FAC855356744B... "iPhone Distribution: SunApps GmbH”
```

Using the JSON API used by the *Apple Developer Portal* it is possible to receive a list of available code signing certificates with its *UDID* (the hex value shown before the identity name). This way, *cert* compares if any of the profiles match. If so, there is a valid identity available on both Apple's server and locally, which can be used to generate a provisioning profile with (more information about *sigh* in section 5.6). If that's not the case, cert will create a new identity by first creating a signing request to Apple's server and use that to generate a new *.cer* file, which will be imported to the Keychain.

It is important to not revoke or disable any existing certificates, as this will invalidate all provisioning profiles created with this signing identity.

5.7.1 Alternative approaches

The only alternative way to create a new signing is using the graphical user interface. This can either be *Xcode* or the Keychain. When using the Keychain, the signing request needs to be uploaded to the *Apple Developer Portal*.

Xcode doesn't require any interaction with the *Apple Developer Portal*, but will revoke all existing identities. This is a problem when using it on a new computer, since no profiles are available there yet.

5.8 produce

Create new iOS apps on iTunes Connect and Dev Portal using your command line

produce is very similar to existing tools. There were no big technical challenges. *produce* uses the *Apple Developer Portal* and *iTunes Connect* to create a new App ID on both web sites, ready to be filled in by *deliver*.

5.9 fastlane

Connect all iOS deployment tools into one streamlined workflow

All the above listed tools are independent from each other. With *fastlane* it is possible to connect them.

5.9.1 Lanes

With *fastlane*, the user defines different build environments (e.g. “Testing”, “Beta Distribution” and “App Store”) that execute a set of build actions in a given order. The user defined which build steps to run in which order. *fastlane* will execute each build action and as soon as one build action fails, the whole build fails. More information about the concept can be found in section 4.2.3.

The configuration is defined using a *Fastfile* (more information about the configuration file available in section 6.1):

```
lane :test do
  sigh :adhoc
  slack({
     message: "Message"
  })
end
```

The *Fastfile* is a normal *Ruby* file, which follows the naming convention of other similar systems (like *Makefile*, *Rakefile*, *Podfile*). When *fastlane* is started, it will use the *Fastfile* from the current directory and execute it. It is being run in its own context to receive all method calls. Each build step is a method call to a build action.

As seen on the graph, all lanes might share some of the build steps. *fastlane* provides a `before_all` and `after_all` code block to execute shared code.

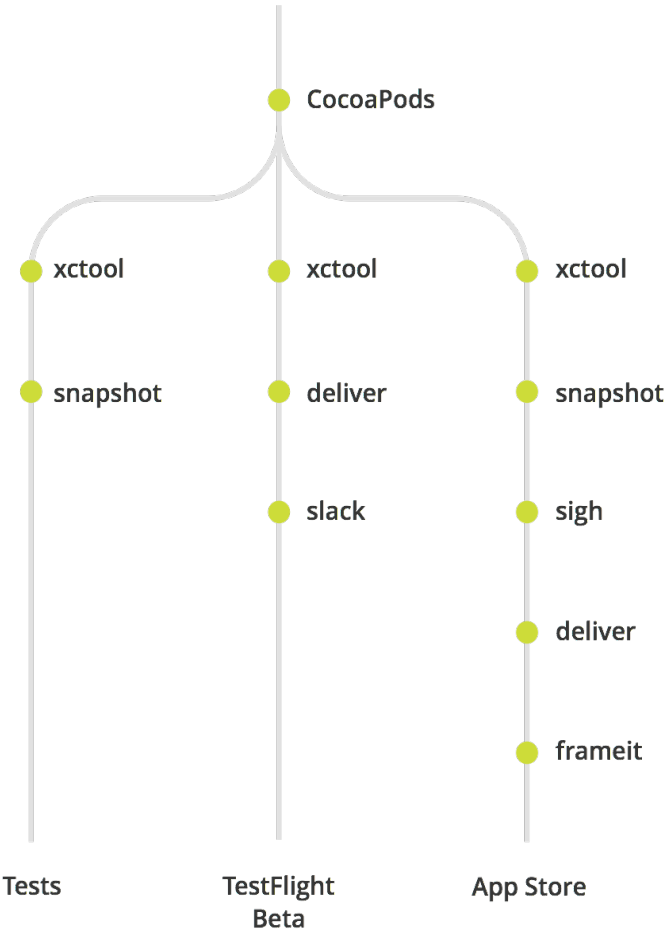

Figure 5: A graphical overview of the multiple lanes and their actions

5.9.2 Actions

fastlane is distributed with over 40 built in integrations and also offers a way for users to add their own private action. Each build step is its own *Ruby* file with its own class.

Example action: "Say", which reads out a given text using the computer's speakers:

```
module Fastlane
  module Actions
    class SayAction
      def self.run(params)
        text = params.join(' ')
        Actions.sh("say '#{text}'")
      end
    end
  end
end
```

fastlane automatically maps the method-call `say("Hi there")` to make use of the class with the name "SayAction" by capitalizing the word and appending "Action" to it. The reason for appending "Action" is potential naming conflicts with other *Ruby* based tools, like *CocoaPods* or *Shenzhen*.

fastlane automatically iterates through all source files inside the *actions* folder and imports the *Ruby* files. This way, there is no need to maintain a list of available actions. The user only adds a new class to the local actions folder and it is available immediately.

5.9.3 Lane environment

The result of one build step usually is relevant to the next actions. For example, after creating new screenshots, *deliver* needs the path to the screenshots to upload them.

The lane environment is a shared key-value store provided *fastlane*, which can be accessed by every build action. Each build action can define a list of keys it might set during its execution.

```
module Fastlane
  module Actions
    module SharedValues
      BUILD_NUMBER = :BUILD_NUMBER
    end

    class IncrementBuildNumberAction
       def self.run(params)
        ...
        Actions.lane_context[SharedValues::BUILD_NUMBER] = new_build_number
        ...
```

As seen on the above example, this build action will store a build number in the *fastlane* environment.

For the user of *fastlane* this means not having to think about manually passing values between actions or setting paths manually. The *fastlane* context is an important concept, which enables developers of actions to implement powerful new features.

5.9.4 Quick Start

To provide a better boarding experience for the user, *fastlane* offers a quick start by running `fastlane init`. The setup will ask for the user's app identifier and Apple ID, since they are needed for almost all tools. They are stored in a configuration file named *Appfile*.

fastlane automatically detects which build tools are enabled for this project, by checking if certain files are stored in the project directory. For example, all projects using *CocoaPods* have to have a *Podfile*. Additionally, the setup asks the user if a certain feature should be enabled, if that is not the case already. Once this is finished, *fastlane* creates a *Fastfile* for the user with all the features and options enabled which were determined before.

fastlane also moves all existing configuration files (like the *Deliverfile*) into a new folder called *fastlane* to have all *fastlane* related files in one location.

The setup process not only creates new files and folders, but also moves existing ones. If something goes wrong during that process (like permission errors), *fastlane* automatically reverts the changes which have already been applied.

5.9.5 Extensions

fastlane makes it easy for developers to add their own actions. `fastlane new_action` starts a setup that creates an action template for the user. By default, the generated action is stored inside the local "fastlane" folder.

5.9.6 Alternative approaches

5.9.6.1 Use existing technologies instead of the Fastfile

Instead of implementing its own system, *fastlane* could make use of existing concepts, like a *Rakefile* or *Makefile*. They already support various features, like task dependencies and are already available. The main problem is not having full control over the behavior of those tools. It would not be possible to implement all features in such a well-integrated and automated way.

5.9.6.2 Use environment variables instead of the lane context

fastlane makes use of a *Ruby* hash (key-value store) to enable the communication between build steps. It allows storing and retrieving of not only string and numeric values, but also complex *Ruby* objects.

Instead of having its own hash to store the values, *fastlane* could make use of environment variables. The problem with this solution is possible name conflicts with external values and the lack of storing complex objects. For example, right now it is possible for an action to store *Ruby* objects or even code blocks in the lane context, which allows even more advanced actions.

5.9.6.3 Use a static configuration

The initial idea for the *Fastfile* was to use a *YAML* or *JSON* file to configure the available actions and lanes. Example:

```
{
  "beta": [
    "sigh",
    "deliver",
    ...
  ],
  ...
}
```

As seen in the example, there would not be a lot of ways to set options for the individual build steps. The main problem with this approach is the lack or accessing and modifying the lane environment. Additionally it is not possible to dynamically read in files or run custom scripts on run time.

Chapter 6: Key Implementation Problems

Below are some of the key problems that needed to be solved. Each item describes the problem, different approaches and the solution that was used in *fastlane*.

6.1 Configuration

The configuration of the tools should be fully flexible to easily fit in into different kinds of setups.

The following techniques could be used:

6.1.1 Static configuration files, like a JSON or XML file.

JSON and XML files are easy to use and widely known. They can easily be generated by other applications.

The problem is the missing flexibility: There is no way to execute a script to fetch or generate a specific value.

For example, *deliver* requires a path to a signed *ipa* file. When running *deliver*, the file should be generated and the path to the newly built file should be returned. That would not be possible with a static configuration.

Another use case is to fetch the latest app description or version number from a remote server, which can only be solved using scripts.

6.1.2 Parameters and environment variables

The tools could be configured by passing values via command line arguments or environment variables. While this would work, it does not solve one of the main goals of this project: It does not store the information required for a release in version control.

One of the main goals of *fastlane* was to store all information required for a release on the file system, to easily deploy from any computer.

6.1.3 Solution: Ruby file

CocoaPods uses a so-called *Podfile* to store all project dependencies. The *Podfile* itself is a *Ruby* script without the file extension *.rb*. *CocoaPods* makes use of the function `eval`, which executes a string in the context of the current application.

deliver uses the same technique: The configuration file is called *Deliverfile*. The *Ruby* script gets executed when the user runs *deliver*.

A code snippet of an example *Deliverfile* (more information about the *Deliverfile* can be found section 4.2.1):

```
screenshots_path "./screenshots"

success do
  system("./my_script.sh")
end
```

The first part of the *Deliverfile* is actually a method call, passing one parameter of the type *String*.

The `success` code makes use of a *Ruby* block, which gets executed by *deliver*. *Ruby* blocks are anonymous functions, which can be called anytime. The last line of the block is the return value of the method.

All available options can be either set as parameter or as a block.

deliver now needs to "collect" the values to start with the submission process. *deliver* uses a class (`Deliverer`), which has the task of collecting and storing all the values to make them easily accessible.

The same approach was used for *snapshot* (*Snapfile*) and *fastlane* (Fastfile).

6.1.3.1 Disadvantages

Since the configuration file is based on *Ruby*, it is more difficult to use for non-*Ruby* developers. If there is a syntax error, the exception message might not be clear to them. Most developers are already familiar with standard formats like *JSON*, which would make debugging for them easier.

6.2 Download files from the Apple Developer Portal

sigh creates a new provisioning profile on the *Apple Developer Portal*, which it needs to download after finishing. *poltergeist* does not support downloading of files from a website.

6.2.1 AJAX

Usually this can be solved by injecting JavaScript code into the current *poltergeist* session to download a file. In this case, *sigh* would inject a simple AJAX request, which returns the content of the file to download.

This approach did not work for *sigh*, since the file to download is a binary file, which is not correctly transferred from the web session to the *Ruby* script.

6.2.2 Plain HTTP Request

It is easily possible to access the download URL of the provisioning profile, but Apple properly secured the download by verifying the active session. Using *wget*, *curl* or a similar tool to send a basic HTML request ended up with a redirect to the login page of the *Apple Developer Portal.*

6.2.3 Fetching the cookies from the session

The tools access the cookies of the current browser session, which contain all information needed and uses this information to send a simple HTTP request in *Ruby*. This includes the session ID and the selected developer team:

```
path = "./example.cer"
downloadbutton = first(".button.small.blue")
url = downloadbutton["href"]
cookieString = ""

page.driver.cookies.each do |key, cookie|
  cookieString << "#{cookie.name}=#{cookie.value};"
end

data = open(url, {"Cookie" => cookieString}).read
File.write(path, data)
```

This code works for both text data and binary data, as it uses *Ruby* code for the actual HTTP request.

6.3 Reinstall the app

snapshot is used to generate new screenshots using *UI Automation*. More information about *snapshot* is available in section 5.3.

snapshot compiles the app before starting the *UI Automation* tests. It does so using `xcodebuild`:

```
xcodebuild
    -sdk iphonesimulator
    CONFIGURATION_BUILD_DIR='/tmp/snapshot/build'
    -project '/Users/felixkrause/.../example/Example.xcodeproj'
    -scheme 'Example'
    DSTROOT='/tmp/snapshot'
    OBJROOT='/tmp/snapshot'
    SYMROOT='/tmp/snapshot'
    clean
    build
```

This command does a clean build of the project to make sure, no outdated code or assets are inside the resulting package. Notice how all output paths are set to */tmp/snapshot*. This means, all build files and the resulting *.app* file will be stored in the given folder.

When using the *Instruments* command line tools to launch the *UI Automation* script, *snapshot* passes a path to an *.app* file for which *Instruments* should run the script. A generated command looks like this:

```
instruments
  -w 'iPhone 6 (8.2 Simulator)'
  -D '/tmp/snapshot_traces/trace'
  -t 'Automation'
  /tmp/snapshot/build/Example.app
  -e UIARESULTSPATH '/tmp/snapshot_traces'
  -e UIASCRIPT './snapshot.js'
  -AppleLanguages '(en-US)'
  -AppleLocale 'en-US'
```

According to the documentation, the app to the given path is used for running the script. Unfortunately, *Instruments* uses the version of the app that is already installed quite often instead, which makes the tool less useful. When the developer changes something in the source code and re-compiles, *Instruments* would then still use the outdated application.

To work around this problem, *snapshot* completely uninstalls and installs the app on the simulator. Using the built-in tool `xcrun simctl` it is possible to interact with the simulator using the command line. The documentation of this tool is incomplete and not up to date. When running `xcrun simctl help`, there are four relevant commands:

```
...
boot                Boot a device.
shutdown            Shutdown a device.
install             Install an app on a device.
uninstall           Uninstall an app from a device.
...
```

It seems straight forward to uninstall and install the app from the simulator using the above listed commands, but there were problems using it: The `xcrun simctl` runs the simulators headless. When running `xcrun simctl boot [device]`, there is no simulator shown on the graphical user interface.

The resulting source code used by `snapshot`:

```
com("killall 'iOS Simulator'")
com("xcrun simctl boot '#{udid}'")
com("xcrun simctl uninstall booted '#{app_identifier}'")
com("xcrun simctl install booted '#{@app_path.shellescape}'")
com("xcrun simctl shutdown booted”)
```

As this solution will delete the app before installing it again, it leaves the `Instruments` tool no choice but to use the latest version. Since some users might prefer not to uninstall the app before running *snapshot*, there is a command line option to skip the forced re-install.

6.4 Configuration

While adding more features to all *fastlane* tools, there were more and more ways to configure the available options:

- **Environment variables**: Especially for build servers it is important to set specific options using environment variables. This makes it easy to change values, without changing the actual code base. Additionally environment variables allow different configurations for different computers.
- **Command line parameters**: A user should be able to pass options via command line parameters. These options are usually features, which are used infrequently.

  Examples are:
  - Show verbose output
  - Skip the generation of a PDF summary
  - Force create a new provisioning profile, even if one exists already

- **Configuration file**: Some of the *fastlane* tools provide a configuration file (e.g. *Deliverfile*, *Fastfile*, *Snapfile*). Those files are stored in version control and contain the required information for a tool to solve its task. If a user provides a command line parameter or environment variable, the values provided by the configuration file will be overwritten
- **Try to fetch the value**: Sometimes, the *fastlane* tools can figure out a value, if the user did not provide it.

  Examples are:
  - The user did not provide a project path, but there is only one project in the current directory.
  - The user did not provide simulator types to use, so *snapshot* can list all available simulators and use those.

- **Default value**: Some values are optional and usually do not require a configuration. The tool can use a default value, if the user did not provide it.

  Examples are:
  - The file name of resulting files is static
  - If no output path is defined, the current directory is used

- **Ask the user:** If the user did not provide a value for a certain option, but the tool requires this value, the user is asked to provide the missing information.

  Examples are:
  - Running *sigh*, without providing an app identifier
  - Running *deliver*, without providing user credentials

After the initial release, the code handling the configuration was in each tool. With the release of a new version of *fastlane_core*, the configuration system is handled centrally across all tools.

Each tool using this new feature, now contains a definition of the available options and *fastlane_core* takes care of handling user inputs, validating inputs and generating the `[tool_name] --help` output. The *description* value is shown using the *help* command.

An example entry:

```
FastlaneCore::ConfigItem.new(
  key: :app_identifier,
  short_option: "-a",
  env_name: "SIGH_APP_IDENTIFIER",
  description: "The bundle identifier of your app"
)
```

By providing the available options in one central place, it was possible to better integrate the tools into fastlane. (to provide configuration inside the *Fastfile*)

For example: A user running *sigh* in the command line might use the following input:

```
sigh --app_identifier "com.krausefx.app " --development -force
```

In *fastlane*, the same command looks like this:

```
sigh(
  app_identifier: "com.krausefx.app ",
  development: true,
  force: true
)
```

The centralized configuration system allows a much better unit testing coverage and easier maintainability.

Chapter 7: Testing

This chapter contains details about how the project is used already and how it was tested.

7.1 Example project

To test all *fastlane* features and tools, an example project was set up. The example project is available on GitHub.
The example project contains a simple *Xcode* project. The main focus of the repository is the *fastlane* integration.

The example *Fastfile* demonstrates a working setup for *deliver*, *snapshot*, *sigh*, *xctool*, *produce*, *Crashlytics Beta*, *frameit* and *Slack*.

To use *snapshot*, the example project shows a working snapshot configuration, which takes screenshots of the app.

While the project is available for everyone to download and use, it cannot be run without modifications, since it requires a valid Apple account. The Apple account is stored in the *Appfile*. When a user wants to deploy the example project, only the *Appfile* must be updated to use a different Apple ID.

This project is used to test the different integrations before a new release of any of the tools is published. Below an example output when running *fastlane* for the example project:

```
INFO [2015-04-05 11:22:04.82]: -----------------------------------------------
INFO [2015-04-05 11:22:04.82]: --- Step: Verifying required fastlane version ---
INFO [2015-04-05 11:22:04.82]: -----------------------------------------------
INFO [2015-04-05 11:22:04.82]: fastlane version valid
INFO [2015-04-05 11:22:04.82]: --------------------------
INFO [2015-04-05 11:22:04.82]: --- Step: xcode_select ---
INFO [2015-04-05 11:22:04.82]: --------------------------
INFO [2015-04-05 11:22:04.82]: Setting Xcode version to /Applications/Xcode-beta.app for all build
steps
INFO [2015-04-05 11:22:04.82]: Driving the lane 'test'
INFO [2015-04-05 11:22:08.16]: ------------------
INFO [2015-04-05 11:22:08.16]: --- Step: cert ---
INFO [2015-04-05 11:22:08.16]: ------------------
INFO [2015-04-05 11:22:13.18]: Login into iOS Developer Center
INFO [2015-04-05 11:22:24.93]: Login successful
INFO [2015-04-05 11:22:27.25]: Downloading URL:
'https://developer.apple.com/account/ios/certificate/certificateContentDownload.action?displayId=LHNT9
C2DPQ&type=R58UK2EWSO'
INFO [2015-04-05 11:22:29.37]: Found the certificate LHNT9C2DPQ-R58UK2EW which is installed on the
local machine. Using this one.
INFO [2015-04-05 11:22:30.50]: Use signing certificate 'LHNT9C2DPQ' from now on!
INFO [2015-04-05 11:22:30.50]: ------------------
INFO [2015-04-05 11:22:30.50]: --- Step: sigh ---
INFO [2015-04-05 11:22:30.50]: ------------------
INFO [2015-04-05 11:22:30.89]: Login into iOS Developer Center
INFO [2015-04-05 11:22:32.39]: Fetching all available provisioning profiles...
INFO [2015-04-05 11:22:33.19]: Checking if profile is available. (76 profiles found)
INFO [2015-04-05 11:23:08.69]: Downloading profile...
INFO [2015-04-05 11:23:09.55]: Successfully downloaded provisioning profile
INFO [2015-04-05 11:23:09.73]: Provisioning profile of app 'Gut Altentann' with the name '1 Gut
Altentann AppStore' successfully generated and analysed.
INFO [2015-04-05 11:23:09.73]: Installing provisioning profile...
INFO [2015-04-05 11:23:09.74]: Profile installed at …/Provisioning Profiles/662c2320-a097-4b04-82b3-
c23ba4330493.mobileprovision"
./Distribution_net.sunapps.1.mobileprovision
INFO [2015-04-05 11:23:09.74]: -----------------
INFO [2015-04-05 11:23:09.74]: --- Step: ipa ---
INFO [2015-04-05 11:23:09.74]: -----------------
DEBUG [2015-04-05 11:23:09.74]: ipa build -m
"/Users/felixkrause/Developer/fastlane/example/Distribution_net.sunapps.1.mobileprovision"
```

```
INFO [2015-04-05 11:23:09.74]: [SHELL COMMAND]: ipa build -m
"/Users/felixkrause/Developer/fastlane/example/Distribution_net.sunapps.1.mobileprovision"
INFO [2015-04-05 11:23:13.44]: [SHELL OUTPUT]: Configuration was not passed, defaulting to Debug
INFO [2015-04-05 11:23:14.21]: [SHELL OUTPUT]: xcodebuild  fastlane.xcodeproj
INFO [2015-04-05 11:23:14.22]: [SHELL OUTPUT]: xcodebuild -sdk iphoneos -project "fastlane.xcodeproj"
-scheme
INFO [2015-04-05 11:23:14.22]: [SHELL OUTPUT]: "Example" -configuration 'Release' clean build archive
1> /dev/null
INFO [2015-04-05 11:23:22.02]: [SHELL OUTPUT]: xcrun  PackageApplication
INFO [2015-04-05 11:23:22.20]: [SHELL OUTPUT]: zip
INFO [2015-04-05 11:23:22.20]: [SHELL OUTPUT]:
/Users/felixkrause/Developer/fastlane/example/fastlane.app.dSYM
INFO [2015-04-05 11:23:22.24]: [SHELL OUTPUT]: Successfully built:
INFO [2015-04-05 11:23:22.24]: [SHELL OUTPUT]:
/Users/felixkrause/Developer/fastlane/example/fastlane.ipa
INFO [2015-04-05 11:23:22.25]: --------------------
INFO [2015-04-05 11:23:22.25]: --- Step: slack ---
INFO [2015-04-05 11:23:22.25]: --------------------
INFO [2015-04-05 11:23:23.24]: Successfully sent Slack notification
INFO [2015-04-05 11:23:23.24]: fastlane.tools finished successfully
```

The interesting parts of the output are:

- **cert** found a valid code signing identity and stored its ID in the lane context for future actions.
- **sigh** found and downloaded an existing provisioning profile, that can be used.
- **ipa** made use of the newly downloaded provisioning profile using the `-m` option when building and signing the app.
- Since all actions were successful, a new message gets posted on *Slack*.

7.2 Companies using fastlane

fastlane is already used in production by Artsy, ProductHunt, Philips, Bosch, MindNode, SunApps, Forepoint, AppInstitute, Panic Inc, Venmo and Xing.

The number of submitted issues on GitHub says a lot about the popularity of a project: Issues are not always bugs with the software, but often questions or feature requests. The more issues are being submitted, the more developers are actively using the library. There were over 300 submitted GitHub issues after the release of *fastlane* and its tools.

Another metric is the number of pull requests on GitHub: Pull requests are submitted by developers already using the tool in the production environment, but want new features or improvements. After the release of fastlane and its tools there were around 100 submitted pull requests. Most of them were merged into the repository. *fastlane* in particular received many pull requests for new build steps like different beta testing services and testing frameworks.

7.3 Usage numbers

With a new release in March 2015, *fastlane* is able to track the number of times each tool is launched. This results in valuable information about the popularity of each tool and how they are used together (see figure 6 below):

- Many developers use *fastlane* without any of the other tools. This means some developers use fastlane to connect third party tools without using the individual *fastlane* tools (like *deliver*)
- *fastlane* is by far the most used tool, followed by *sigh*, *deliver* and *snapshot*.
- Some developers use the individual tools (e.g. *deliver*) without using *fastlane*.
- Within 3 weeks, *fastlane* and its tools were launched over 20,000 times.

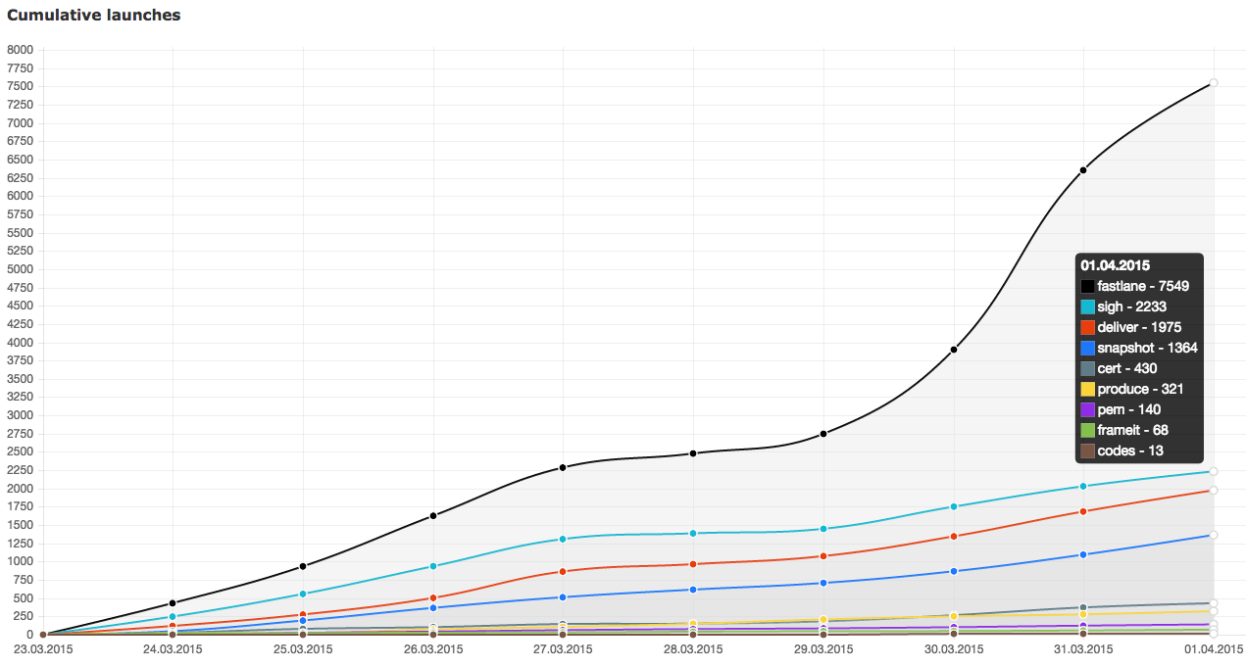

Figure 6: The cumulative number of launches of the various tools within 9 days

7.4 Commercial Aspects

Companies that already integrated *fastlane* save hours of releasing app updates manually. MindNode for example supports 15 languages, resulting in 375 screenshots to generate and upload.

fastlane saves expensive developer time, as many tasks need to be done by a developer and cannot be delegated to a non technical employee, but can be automated by scripts.

MindNode is now able to build and distribute new beta builds to their beta testers by running *fastlane*. This drastically increases the quality of the app, since users can already try and use the beta version before being submitted to the App Store.

7.5 Press Coverage and Articles

With more developers using the tools, there were also articles and podcasts covering *fastlane*. Some examples:

- Dave Verwer linked four of the tools in the best-known iOS Developer Newsletter "iOSDevWeekly". This resulted in about 1,000 visitors each time.
- *deliver* and *fastlane* were featured on *ProductHunt*, a website that showcases new products and services.
- The app developer podcast "GoneMobile" by (Shackles and Dick, 2015) published a 40-minute interview with Felix Krause about *fastlane*.
- (Sergio , 2015) published an article on InfoQ with a short interview about *fastlane*.
- Various tools were linked in the "iOS Goodies" Newsletter.
- A case study about the *fastlane* project on the UCLAN website by (Jones, 2015).

Chapter 8: Evaluation and Critical Review

8.1 Scope of the project

The initial goal of this project was to only provide a way to upload app metadata and app updates to the App Store. With the public release of each of the tools, I noticed more and more things missing that need to be automated. This resulted in five more tools being developed and released. Users were writing their own build scripts to run the individual tools. I wanted to provide better way to connect them. The planning and development of *fastlane* took longer than the other tools, as it is more complicated and had to support a wide variety of integrations for the initial launch. Even after the release of *fastlane*, two more tools were released and the number *fastlane* integrations increased from 13 to over 40 thanks to the great adaption rate and user contributions.

8.2 Development

There were no serious incidents during the development. Thanks to the feasibility report, I tried to get the key parts running to test if the project is possible. I already had some *Ruby* experience with the web framework *Ruby on Rails*, but have never before worked on my own *Ruby Gem*. While developing a *gem* was new to me, it got significantly easier after the first tool was released, as the project structure stayed the same.

For developing *deliver* I used the "Test Driven Development" approach by writing the tests first. This was very useful, since the user interface was the last thing I implemented. The tests are automatically executed for each commit by "Travis-CI".

Since the initial release of the tools, the communication with the users is a key part when developing on the *fastlane* tools. Most feedback was received on GitHub via "Issues" and on Twitter.

8.3 Releases

The initial release of *deliver* was on the 5th November 2014. Since then, *deliver* alone had 35 releases (Krause, 2015c).

Each release involves uploading the *pkg* file to *RubyGems*, adding a *git* tag and providing a change log on GitHub. I received very positive feedback about the detailed release notes and documentation.

8.4 Support and Pull Requests

With over 300 issues (250 being resolved), responding to every incoming query, feedback, bug report or feature request takes up a lot of time.

Each pull request submitted by users must be carefully reviewed before they are merged into the repository. I not only have to make sure the feature is implemented correctly, but also verify the coding style, documentation and tests.

Feedback received from users is very valuable. (Thompson, 2014b) wrote about that topic in the context of open source projects:

> Each question is a data point for what could be clarified or improved within your own software and documentation

This is a really relevant point: If a user needs to ask for help, something can be improved in the documentation or setup process. *fastlane* provides meaningful and helpful error messages that help the user get the issue resolved without having to look at the documentation.

I think carefully about the ways to improve documentation and the tools for every submitted GitHub issue. Not every user takes the time to submit an issue, which means, some developers just give up instead of writing me.

It is important to discuss issues publicly so that other users experiencing the same problem can find the discussion.

> For each person who asks a question, there are dozens of others who don't and get frustrated and give up.
>
> …
>
> Answering one question on a mailing list or developer forum helps many more people than just the asker.

(Thompson, 2014b)

This turned out to be correct for GitHub issues: users running into a problem usually use the GitHub search to look for existing issues and reply to the issue if they are also affected or have something to share.

8.5 Commercial considerations

All tools are available open source under the MIT license. Alternatively, *fastlane* could be provided as a hosted service:

A user pays a monthly fee to get access to *fastlane*. The servers are hosted and maintained by *fastlane*. While this would probably generate some revenue, it is clear, that it does not only involve the development of *fastlane* itself, but also building a server infrastructure of Mac OS X computers. There are more things to consider, like user's privacy and how to securely store user credentials.

Since I wanted to focus on providing a great *Continuous Delivery* solution, I did not want to invest time in setting up a server infrastructure. This approach would also mean I would have to actively reach out to potential customers to generate sales.

8.6 Publishing the project as open source code

It was not clear from the beginning if *fastlane* should be open source or provided as a service. I decided to publish it as open source for the following reasons:

- It allows developers to modify the project to fit their needs and to extend the features provided by *fastlane*. As a developer, it is great to know you can fix something if you reach the limit of a project.
- It is easier to gain trust from developers, as they can verify the project securely stores their user credentials.
- Many larger companies have a policy that their source code must not run on external services.
- Developers can write their own plugins and extensions.
- Since the project is open source and free to use it was widely adopted in a short period of time. The more users and companies use this tool, the better it gets, as every pull request adds more features and improvements for all users.
- If for some reason Apple changes something in the release process, there will quickly be a fix available, provided by one of the developers using *fastlane*.

I decided to be fully open with this system and provided all *fastlane* related code under the MIT license.

8.7 What was learnt from the project?

8.7.1 Launching an open source project

Most open source projects are launched silently and do not gain lots of public interest. Since no money is involved, the developers often do not care about how to design the project page and where to publish links about the project.

I tried to make *fastlane* and its tools easy to get started and present them, as they were products. Some examples are:

- Each tool has its own logo with a different color.
- I structured the project page to make it immediately clear what the tool does and what it is used for. Additionally I highlight the differences to exiting tools.
- The project page contains a list all features of the tool.
- Each tool's project page contains links to all other tools.
- For some tools, I added a gif screen recording to the project page to quickly visualize what the tool does and how fast it is.
- During development I worked using a private GitHub repository and launched the project publicly once the initial version was finished.
- Due to the traffic on the launch day, almost all tools became the number one trending *Ruby* repository of the week or the month.
- I made sure, the most important iOS developer news report about the tools, including the *iOS Dev Weekly* Newsletter.
- All tools link to each other on the project pages and documentation. If a developer only uses one tool, he will know about the other tools as well.

8.7.2 Tests are important

The tools were developed using the TDD-approach. While this was really useful during the development, it also helped to develop new features faster without breaking other parts of the software. Additionally, it made reviewing pull requests submitted by users simpler, since the tests results were displayed right on GitHub.

Most pull requests actually contained new tests for the new features submitted. This was quite surprising to me. The other developers took the time to look into the existing test suite and implemented new tests to cover their newly implemented features.

8.7.3 Testing and updating the project

Quite often users send in support requests with a problem they encounter. It is often difficult to fix problems other users experience, as they might be in a completely different environment. There are many external factors involved, like the Internet connection, the computer, the project and the configuration of the *fastlane* tools.

This requires an easy way to best emulate the environment of the users. During the development I ended up using the following techniques:

- Use *rvm* by (Seguin, 2015) to simulate different *Ruby* versions when using and testing the tools
- Use the Network Link Conditioner to simulate bad network connections. This application is provided by Apple and allows setting a custom Internet speed and package loss to test your code in different environments.
- Write tests first when fixing an issue: When an issue needs to be fixed, I start with writing a failing test. After the test is finished, I work on fixing the bug until the test succeeds.

8.8 Crash Reporting

Since all tools are executed locally on the developer's computer, there is no way to tell what *fastlane* actions the users have integrated and what integrations are causing the most problems. With an update in April 2015 *fastlane* reports the number of launches per action and the number of failures. *fastlane* tracks the data during the tool's execution and uploads it after the run is finished. The developer is asked and can easily opt out.

The results are collected on a server and visualized in a simple table to get a quick overview of actions that fail more often than others.

xctool	0.227	132	30
git_status_check	0.2	5	1
ensure_git_status_clean	0.187	91	17
reset_git_repo	0.171	35	6
metrix	0.095	21	2
carthage	0.077	13	1
crashlytics	0.071	98	7
produce	0.07	43	3
increment_build_number	0.065	368	24
hockey	0.064	235	15
xcode_select	0.056	18	1
sigh	0.042	427	18
cert	0.022	91	2
cocoapods	0.019	310	6
slack	0.015	199	3
fetch_build_number	0.0	4	0

Figure 7: The generated crash statistics
First column: action name,
Second Column: ratio of launches and errors,
Third Column: Number of launches,
Fourth Columns Number of errors

Chapter 9: Future Work

Since this project is already widely used and has a lot of potential, there are lots of plans for future development:

- **Mac Support**: Currently all tools are now focused on iOS apps. Some of the scripts must be adapted to work with Mac apps as well (*deliver*, *cert*, …), while others would need a complete re-write, like *snapshot*. *UI Automation* is only available for iOS apps. Additionally, building and signing Mac apps works differently and would need some changes to make it work.
- **fastlane for Android**: This is a really important step, since it would broaden the target group from iOS developers, to most mobile app developers.
  The main issue when developing this is the support of other platforms, probably Linux and Windows. *fastlane,* as it is now, can only run on Mac OS X computers.
  What needs to be adapted:
    - **deliver**: There is a *Google Play Publishing API* provided by (Google, 2015), which makes implementing uploading of new builds much easier than it was for *iTunes Connect*.
    - **snapshot**: *spoon* by (Square, 2015) is an open source project for creating screenshots for Android apps. It supports quite similar features as *snapshot*. The only thing missing is a *Ruby* wrapper around it.
    - Certificates for Android apps are generally easy to use and maintain. The only action required from *fastlane* is making sure a valid key is available.
- **Xcode plugin**: *Xcode* supports the development of plugins to show user interface elements and output logs right in IDE. By placing a new button on the menu bar, the developer would be able to choose a *lane* to run and see the output right inside *Xcode* without having to switch to the terminal.
- **Switch to HTTP requests** for interaction with *iTunes Connect*: It makes sense to use a standard HTTP client to send GET and POST requests to the unofficial JSON API used in *iTunes Connect*. This would result in much faster and more stable execution of *deliver* and the other tools. There is an unofficial documentation of the JSON API available by (Krause, 2015a).
- **More fastlane integrations**: At the time of writing, there are already 40 integrations of *fastlane* available, for example: AWS S3 distribution, Slack, Hipchat, Crashlytics and DeployGate. Many of the available integrations were provided by *fastlane* users.
- **Easier installation of fastlane:** Some people are not familiar with the *gem* system and might run into issues during the installation. Providing *fastlane* as a package to download from a website additionally to providing it via gem installation would be a great solution.

Chapter 10: Conclusion

The *fastlane* project was about creating a variety of developer tools to cover all steps necessary to automate the submission process of iOS apps. This includes creating all required profiles, generating new screenshots, uploading metadata to *iTunes Connect* and submitting the app for review. The *fastlane* project needs to use both the local Apple developer tools (like *xcodebuild*) and the Apple web services (like *iTunes Connect*).

Instead of providing one big tool capable of solving all kinds of tasks, the project makes use of multiple individual tools to clearly separate different features and use-cases.

The software was distributed as gems via *RubyGems* and runs locally on the developer's computer. This allows the developer to have full control over the environment and installed dependencies.

After the release of the individual tools *deliver*, *snapshot*, *frameit*, *PEM* and *sigh,* there was no really good way to connect those tools. Every developer using the *fastlane* tools had to build their own scripts to trigger the individual tools.

fastlane was developed to enable developers to connect all *fastlane* tools. This includes passing on information from one build step to the next ones. *fastlane* also integrates with commonly used third party tools. As it turned out, a lot of teams use *fastlane* with third party tools only.

This document includes usage statistics and user feedback, since the release of the software happened during the project period. Thanks to the early release of the software, it was easy to gather user feedback and further optimize the documentation and setup experience.

fastlane was an unexpected success and became much bigger than originally planned. Is is actively being used around 1,000 times a day and has a huge influence in the iOS developer community.

Almost all tools became the number one trending *Ruby* repository of the month on GitHub. Various tools got featured on well-known developer blogs and newsletters.

All source code is fully open source under the MIT license. Other developers can contribute to the project by submitting Pull Requests with new features or improvements.

Chapter 11: Appendices

11.1 Glossary

- **Apple Developer Portal**: A website provided by Apple. It is used by iOS developers to create, maintain and download code signing certificates, provisioning profiles, App IDs and push notification profiles.
- **App Store**: The App Store is used to distribute iOS apps to the end users. It is provided and maintained by Apple.
- **Bundle Identifier**: A bundle identifiers are usually written in reverse DNS notation (e.g. *com.krausefx.myappname*). It is unique and cannot be used by multiple apps. It is used for various actions, like launching the app using an URL scheme (e.g. from the Safari web browser).
- **CocoaPods**: A dependency manager for iOS libraries. It is widely used, not only for third party libraries, but also for company internal projects.
- **Commit:** A commit in the context of *git* version control is composing and storing of a snapshot of the current code. It usually contains only the differences to the last snapshot.
- **Continuous Delivery**: *Continuous Delivery* is a technique, which uses automated testing, and continuous integration to speed up the development and release of software. This does not mean the developer should release a new public version after each commit but it states that every commit can easily be deployed.
- **Fork**: A fork copies the code base from the original repository to a new one. It is usually used in combination with pull requests. (Quaranto, 2009)
- **Instruments**: A set of developer tools provided with *Xcode* to debug an iOS app (e.g. memory leaks, performance) but also to run automation scripts using *UI Automation*.
- **ipa**: The `ipa` stores the iOS app. It is a compressed format and can be extracted using a standard ZIP application. The usual process of creating an *ipa* file is: Compile the app, sign it and then compress the resulting folder into an `ipa` file. The resulting file is then used to submit the app to the App Store.
- **iTMSTransporter**: This is a tool, which is integrated into *Xcode*. It was originally used to upload new music files to the iTunes store. With the recent updates it is possible to use it to upload general app metadata, screenshots and new *ipa* files to *iTunes Connect*. The tool is based on Java. The documentation is only accessible to registered iOS developers.
- **iTunes Connect**: a website provided by Apple. It is used by iOS developers maintain the app's metadata, upload screenshots and submit new versions for review. It is based on AngularJS and uses a WebObject back-end.
- **Jenkins**: A Continuous Integration system, which is often used to execute jobs after each commit. Jenkins is often used by iOS developers, since it is also compatible with Mac OS X. It is plugin based and offers a variety of helpers to speed up setting up a *Continuous Integration* system.
- **poltergeist**: a headless web browser for *Ruby* based on *PhantomJS* by (Leighton, 2011). It is used to control the *iTunes Connect* and *Apple Developer Portal* websites.
- **Provisioning Profile**: To properly sign and submit an iOS app, the developer needs to use a provisioning profile. It can be created on the *Apple Developer Portal*. It is unique for each app identifier and code signing identity.
- **pry**: An open source *Ruby* tool to launch an interactive *Ruby* shell in the context of any *Ruby* script.

- **Pull Request**: When developers want to suggest a code change on GitHub, they can fork the repository, implement the changes in their code base and submit a pull request. The pull request shows a notification on the original repository about the code change, which is ready to be merged.
- **Push Notification**: Push notifications are the messages shown on the user's iPhone/iPad when the app is not running but wants to display a message. This is used for informing the user about important updates or announcements. To send push notifications, the developer needs to create a push notification profile, which then is uploaded to a web server. The web server communicates with the *Apple Push Service* using the push certificate to send new messages.
- **Regression**: A regression is used in the context of automated tests. The document talks about a regression when the source code was changed and broke some part of the software system. The goal of automated tests is to catch those regressions.
- **Ruby Gem**: a *gem* is a package containing *Ruby* source code. It is used to distribute *Ruby* tools on *RubyGems* by (Quaranto, 2009). *RubyGems* hosts the uploaded gems and let's users install them with only one line of code: `gem install [gem_name]`.
- **Travis-CI**: A hosted *Continuous Integration* solution, which is free to use for open source projects. It also supports Mac OS X builds, which makes it interesting for iOS developers as well. The job configuration is done using a *.travis* configuration file in the repository.
- **UI Automation**: A framework provided by Apple `Instruments`, which is based on `JavaScript` and used to simulate user interaction of the app. Commands like tap on this button, or swipe from x to y can be used to click through the app's screen. It is primarily used for integration tests. Since the scripts can also create screenshots, this technique is used by `snapshot` to create new screenshots.
- **WebObjects**: a Java based server application framework developed and maintained by Apple. It is used for most back-end software by Apple, including iTunes Connect.
- **Xcode**: An IDE provided by Apple for iOS and Mac App developers. It features a code editor, integrated version control and supports building, signing and distributing of iOS/Mac apps.
- **xctool**: An open source library provided and maintained by Facebook to easily run unit tests from the command line.

11.2 Project Proposal

Department of Computing Degree Project Proposal

Name: Felix Krause **Course:** Software Engineering **Size:** double
Discussed with (lecturer): Christopher Casey
Current Modules (and previous modules if computing or direct entrant)
Optional: Computer Graphics
The Project Title
Continuous Delivery for iOS applications
Project Context
Releasing an update for an iOS application is time consuming and risky. The developers have to take and upload screenshots of the app for every screen size and every supported language for each update.
Creating and maintaining the code signing profiles and push notifications certificate is a manual process with a lot of sources of errors.
About six months ago I open sourced a tool to automate **taking** screenshots (https://github.com/KrauseFx/rScreenshooter).
The bachelor project should cover the whole process of trigger unit- and integration tests, providing an easy way to create screenshots, uploading the app metadata to iTunesConnect, building and signing the application, uploading the resulting package onto iTunesConnect.

Specific Objectives
Updating app metadata, taking and uploading app screenshots, building and signing the application, uploading the resulting package to Apple

References

Ship.io (2012) **What Makes Continuous Delivery For Mobile Different?** Available at: http://blog.ship.io/2012/11/24/what-makes-continuous-delivery-for-mobile-different/ [Accessed 18. September 2014]

Humble J. and Farley D. (2010) **Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation**. Boston, Ma: Addison-Wesley

Kammah, N. (2014) **Etsy's Journey to Continuous Integration for Mobile Apps.** Available at: http://codeascraft.com/2014/02/28/etsys-journey-to-continuous-integration-for-mobile-apps/ [Accessed 18. September 2014]

Singleton, A. (2014): **Continuous Delivery: Benefits and Costs.** Available at: http://www.continuousagile.com/unblock/cd_costs_benefits.html [Accessed 18. September 2014]

Abtrantes, B. (2014): **Continuous Integration for iOS Apps.** Available at: https://medium.com/@brunoabrantes/continuous-integration-of-ios-apps-6f6ca8a9804c [Accessed 18. September 2014]

Annema K. (2014): **Ship your app weekly.** Available at: https://blog.yourkarma.com/ship-your-app-weekly [Accessed 18. September 2014]

Potential Ethical or Legal Issues
none

Resources
Taking Screenshots:

- Subliminal (https://github.com/inkling/Subliminal)
- rScreenshooter (https://github.com/KrauseFx/rScreenshooter)
- iOS-Screenshot-Automator (https://github.com/toursprung/iOS-Screenshot-Automator)

Uploading app metadata onto iTunesConnect

- Using the frontend with Capybara
- Using iTMSTransporter

Uploading the app itself to iTunesConnect

- Using iTMSTransporter
- Scripting Xcode directly
- Scripting the *Application Loader* directly
- xcrun validate command (doesn't work since the latest Apple update)

Potential Commercial Considerations - Estimated costs and benefits
Recently Apple announced the new iPhones with other screen sizes. For every app, the developer has to manually take and upload five screenshots per screen size per language, which results in 100 images when the app supports four languages.
Continuous Delivery is already used in most server side applications, where deployments are made dozens of times each day.
On iOS, unit tests just start getting established, since Apple started officially supporting them in Xcode. Unit- and integration tests are a requirement to start with continuous delivery to not ship broken apps.

There are no external costs involved. The resulting library should be open sourced and available to every iOS developer.

Proposed Approach
Tasks:

- Interact with the iTunesConnect web application
- Create and maintain code signing profiles and push notification certificates using the frontend of the Apple Developer Console
- Provide an easy way to create screenshots and prepare them for upload
- Build and sign the application
- Find a way to upload the resulting package to the iTunesConnect back-end
- Specify a *Deployfile*, which contains all required information to deploy an update
- Develop a Ruby gem for easy install and setup
- Validating and correcting the inputs given from the "Deployfile"

Stages:

1. 2 Weeks: Control the iTunesConnect frontend to create and update apps
2. 2 Weeks: Find a way to upload a signed ipa-file to the iTunesConnect back-end
3. 2 Weeks: Define class structure, which expose useful methods to help with the deployment process
4. 2 Weeks: Specify the *Deployfile* and document it
5. 1 Week: Implement the flow of the whole process by calling the different callback methods from the *Deployfile*
6. 1 Week: Provide a default implementation for all available callbacks
7. 1 Week: Easy way to integrate into Jenkins with valid JUnit reports

11.3 References

AccuRev (2008) *Software Configuration Management Best Practices for Continuous Integration*, [Online], Available: http://www.elegosoft.com/files/Downloads/Paper/best-practices_continuous-integration-in-scm_en.pdf [14 March 2015].

Deshpande , A. and Riehle, D. (2008) *Continuous Integration in Open Source Software Development*, [Online], Available: http://dirkriehle.com/wp-content/uploads/2008/03/oss-2008-continuous-integration-final-web.pdf [14 March 2015].

GitHub (2012) *Deploying at GitHub*, 29 August, [Online], Available: https://github.com/blog/1241-deploying-at-github [14 March 2015].

Google (2015) *Google Play Developer API*, 14 Feb, [Online], Available: https://developers.google.com/android-publisher/ [11 Apr 2015].

Humble, J. and David, F. (2010) *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*, 1st edition, Addison Wesley.

Jones, S. (2015) *University of Central Lancashire Blog*, 23 Jan, [Online], Available: http://www.uclan.ac.uk/about_us/case_studies/students_fastlane_software_in_demand.php [07 Apr 2015].

Krause, F. (2015a) *iTunes Connect JSON API Docs*, 20 Mar, [Online], Available: https://github.com/fastlane/itc-api-docs [07 Apr 2015].

Krause, F. (2015b) *deliver Deliverfile documentation GitHub*, 23 Mar, [Online], Available: https://github.com/KrauseFx/deliver/blob/master/Deliverfile.md [10 Apr 2015].

Krause, F. (2015c) *deliver Releases GitHub*, 01 Apr, [Online], Available: https://github.com/KrauseFx/deliver/releases [11 Apr 2015].

Leighton, J. (2011) *poltergeist*, 28 Oct, [Online], Available: https://github.com/teampoltergeist/poltergeist [06 Apr 2015].

Minick, E. (2014) *Continuous Delivery Maturity Model*, 10 February, [Online], Available: https://developer.ibm.com/urbancode/docs/continuous-delivery-maturity-model/ [14 March 2015].

Quaranto, N. (2009) *RubyGems*, 06 Apr, [Online], Available: https://rubygems.org/ [06 Apr 2015].

Rational Software (1998) *Rational Unified Process - Best Practices for Software Development Teams* , [Online], Available: https://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf [14 March 2015].

Seguin, W.E. (2015) *Ruby Version Manager*, [Online], Available: https://rvm.io/ [06 Apr 2015].

Sergio , S.D. (2015) *infoq*, 23 Jan, [Online], Available: http://www.infoq.com/news/2015/01/fastlane-ios-continuous-deploy [07 Apr 2015].

Shackles, G. and Dick, J. (2015) *GoneMobile*, 01 Mar, [Online], Available: http://gonemobile.io/blog/e0023-Continuous-Delivery-for-iOS-Apps/ [07 Apr 2015].

ship.io (2012) *What Makes Continuous Delivery For Mobile Different?*, 24 November, [Online], Available: http://blog.ship.io/2012/11/24/what-makes-continuous-delivery-for-mobile-different/ [14 March 2015].

Square (2015) *Spoon*, 02 Apr, [Online], Available: https://github.com/square/spoon [11 Apr 2015].

Sutherland, K. (2014) *KSScreenshotManager*, 06 Oct, [Online], Available: https://github.com/ksuther/KSScreenshotManager [11 Apr 2015].

Thompson, M. (2014a) *shenzhen*, [Online], Available: https://github.com/nomad/shenzhen.

Thompson, M. (2014b) *NSHipster - Stewardship*, 27 January, [Online], Available: http://nshipster.com/stewardship/ [14 March 2015].

Vasilescu, B., Schuylenburg, Wulms, J. and Serebrenik, A. (2014) *Continuous integration in a social-coding world: Empirical evidence from GitHub*, [Online], Available: http://www.win.tue.nl/~aserebre/ICSME2014ERA.pdf [14 March 2015].