Automated testing: A look at Architecture

By:
Paul Batty

Supervisor:
Manfred Lau

March 2017

The dissertation is submitted to
Lancaster University
As partial fulfilment of the requirements for the degree of
Integrated Masters of Science in Computer Science

I certify that the material contained in this dissertation is my own work and does not contain unreferenced or unacknowledged material. I also warrant that the above statement applies to the implementation of the project and all associated documentation. Regarding the electronically submitted version of this submitted work, I consent to this being stored electronically and copied for assessment purposes, including the Schools use of plagiarism detection systems in order to check the integrity of assessed work. I agree to my dissertation being placed in the public domain, with my name explicitly included as the author of the work.

Date:

Signed:

To get the code and other documents visit the website at:
http://www.lancaster.ac.uk/ug/battyp/autotesting

Abstract

This paper...

Keywords:

Introduction

Automated testing in no new idea as the field of computer science strives to automated all that it can. However, the idea and implementation of automating the development pipeline, going under the name of continuous integration and continuous deployment is a ever changing field. This has lead to the rise of devops also known as development operations dedicated to maintaining and building such systems with more of a focus on the collaboration of the different team involved in the pipeline such as quality assurance, development and management.

During a recent project tasked with the aim to create the pipeline from the ground up there were many question that had to be answered. Such as what is the difference between continuous integration and continuous deployment. How does each part of the system fit into the big picture. What is the end game of the system. This is covered in the first section of this paper.

Besides just understanding the reasoning and terms used behind the names, it was time to design and implement a system for the project. In addition to looking at the project that this idea spawned from, the paper will look at other attempts and experience from both industry and academic to understand the ideal architecture. This is covered in the second and third chapters.

From here the paper will look at the how this architecture will work in practise, the patterns and the direction that this field is heading towards making sure that it is on the right track. This is covered in the final sections four and five.

During the creation of the project there seemed to be no coherent or common meanings behind some of the terms. There also was a lack of architectural designs and implementations with most arguing about the systems cost-benefits.

Therefore, the main aim of this paper is help define and understand the systems and terms used in this field. In addition to looking at the common themes used in practise to provide a guide for the creation of other automated systems, and those looking to understand this area more in depth.

1 Background

The following chapter will cover two sections. Firstly, looking at where the idea for continuous development came from and the how the field ended up where it is today, going over the terms used. The second part looking at some of the tools used.

1.1 History of Continuous deployment

Continuous deployment is in a group of methodologies under the name of extreme programming (XP) which in turn is part of the Agile process Wells D. (1999). The core principles of extreme programming is to be adaptive to change and quick feedback for everyone involved. Developers get feedback on the code, bugs and features. Clients get the features they need and Managers can make decisions about the direction of the project without bringing the whole system down. TODO:cite

This movement started in March 1996 by Ken Back TODO:cite with continuous integration going further back to 1991 by Grady Booch TODO:cite. The main change between that of Booch's design and extreme programming, is that Booch placed a one integration a day limit, whereas extreme programming favours much more. TODO:cite

The core idea behind Booch's idea is to avoid problems when a new release is integrated into an old system. It could achieve this goal via automated unit tests. Each test would run through a single public method and make sure that it is performing as it should. For example if a method takes two numbers and return the sum of the numbers. A unit test would test that *1+1* will return *2*, trying edge cases such as using letters and so on. In total there would be a group of tests for every public function.

After the developer has made a change to the code base they would run the tests if they all passed then the code was OK to be check in and used in the next release. This was enhanced with the idea of test driven development, where the test are written first then the change.

This all started to kick off around 1997 with the continuous integration being place inside of the extreme programming movement. This continued until 1999 through various books and publications by the movement, namely Kent Beck.

Up to this point continuous integration just consisted of developers writing unit

tests and running them locally to make sure that everything passes. When all the test pass the developer would then check in the changes in to the version control system (VCS). Other developer then working on the same code base will be able to get the latest code and know that it works.

This started to change around 2001 with the release of CruiseControl, because in the previous system what if a developer did not run the unit tests, or forgot or check in some files, so it would work fine on their local set-up but nowhere else. Therefore rather then leaving it up to the developer it could be automated. This introduced the idea of build servers.

A build server would sit there and depending on the particular set up and work-flow of the project, would take the changes run the tests against them and then send out a report to the developer, or anyone who was interested. Now the if the developer forgot something it would be caught before anyone else started working on top of the changes.

So far most of the work was performed by developers for developers, in order to assure that the current state of the code base was in a always working condition. This continues until 2008 when Patrick Debois and Andrew Shafer meet up and discuss bridging the gap between development, system administrators and other roles within the agile infrastructure. For example the developer environment is different to the test environment witch in turn is different to QA and production environments.

This then sparked the next stage in the movement, the creation of devops. This in turn created a whole host of new tools such as Jenkins (Hudson), Puppet and Chef just to name a few. These new tools made continuous integration easier then ever, and as they gained maturity started to see a lot of use in industry.

As these tools started to gain popularity and with the internet being widespread, there was a shift to not only able to test, but as as the code is in a always working condition push out to the customers so they can always have the latest version, features and so on. This goes under the name of continuous deployment. This allows bugs to be fixed almost as quickly as they are found due to the reproduction of the customers environment back over in the developers workstation.

Today, the transition over to continuous deployment is still being made, with more tools arriving. The idea of serverless severs and tools such as Docker in order to increase the reproducibility of the environments faster and with better accuracy.

In short the entire pipeline is beneficial to everyone and comes in three different levels, automated builds, continuous integration and continuous deployment. Each being a step up from the one before it.

1.2 Tools

In the previous part this paper named various tools that are used in the pipeline, they fall into three large categories, version control systems, testing and infrastructure. In this next part the paper will look at each of the categories in turn to understand where they fit into the big picture and how they are used.

1.2.1 Version control systems

A version control system (VCS) or version control goes under two other names, revision control and source control. A VCS has a simple premise to manage the changes performed on a document. For example every time a document is edited, the changes are stored alongside the timestamp, user and other metadata. Then if at some point down the line a user want to see who made the changes or undo the changes as it broke the system they should be able to easily.

In short it is a system to manage a documents version, over long periods of time even when the system in closed, restarted or moved. This lays the foundations for more complex operations. As the changes are stored comparisons of different versions can be made, allowing them to be merged together.

If user A sees an error in a document that the owner has not fixed, they can show the changes to the owner, if the owner agrees and likes the changes they can merge them in. This applies the changes from user A into the main document. This is also known as a pull request.

If the owner makes a change to the document they do not need to send a pull request as they own the original, therefore they can merge right away. This merging is also called checking in, as the person is checking in there changes to the VCS.

The VCS however, does not just record changes to just a single document but everything inside of a folder. Therefore it will track adding new files, deleting files and renaming files. Each VCS handles it slightly differently, however this is irrelevant to this paper.The folder will the in case of software development be the project root.

If a VCS is tracking all files, then when checking in changes it would be a waste to do this per file, instead changes are grouped together under a commit. This commit is then checked in one go. Rather than a single file at a time, this allows relevant changes to be grouped together creating a nicer history for the project.

VCS has one more main function to cover in respects to this paper, called branching. Similar to that in a tree VCS have a main or master branch called the trunk. A branch is similar to a workspace, and represents the changes on that workspace that got it to its current point.

When branching it creates a copy of the current branch allowing it to take off in a different direction. Therefore development can go into two different directions without issues of people working on the same file. Then if needed can be merged back. This is better represented graphically as seen below in figure 1.1:

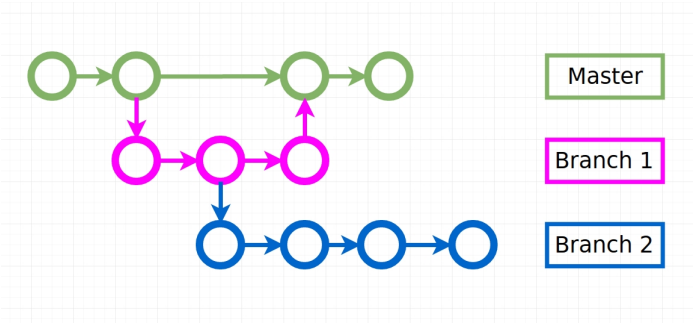

Figure 1.1: VCS Branching

Each circle on the figure represents a commit (a group of changes). The figure also shows that branch one splits off from master, using it as its base then merged back in. While branch two uses branch one as its base, however has not yet been merged. There are different work-flows around this feature covered in more depth in section TODO:link the section.

So far the paper has gone on about a folder that exists, this folder goes under the name of a repository. The next part will look at where the repository is located, such that multiple people or a single person can work on the project taking advantage of VCS, and how they differ. There are two main way that this is achieved, both use a client sever architecture.

The first has the server contain the repository, then the developer will create a local copy of the repository according to the branch they are on. They then work on the local copy editing files, however anything else such as creating a branch,

merging or checking out files is performed by the server therefore a connection is required.

The second way is distributed and has the developer create a local repository that mimic the server one then everything can be performed locally. When a connection to the server is available they can push their changes on the the server repository. This can be seen figure 1.2:

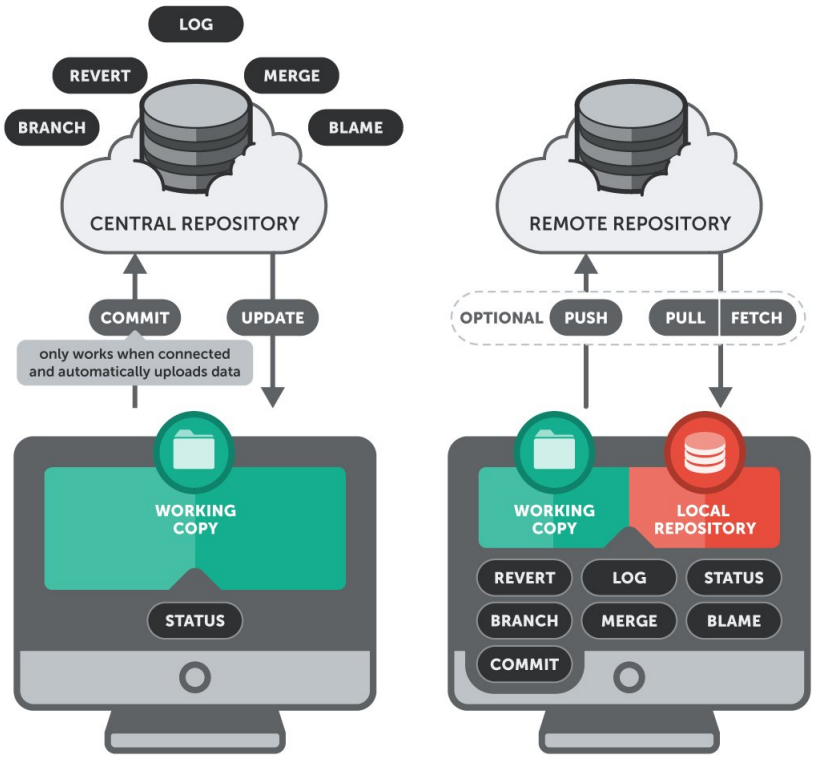

Figure 1.2: VCS systems modified from Fournova Software (2013)

There are a lot VCS out there, however the most popular is GIT, TFS and Subversion (Stackoverflow (2017)). Git follows the distributed system whereas TFS and Subversion uses the repository server system.

There are a lot more intricate details to VCS however they will not be covered this paper.

1.2.2 Testing

1.2.3 Infrastructure

2 Evaluation

eval

3 Conclusion

conc

References

Wells D. (1999), On line publication, Extreme programming: A gentle introduction, http://www.extremeprogramming.org/, Last Accessed 20th April 2017

Fournova Software. (2013), On line publication, Switching from Subversion to Git, https://www.git-tower.com/learn/git/ebook/en/command-line/appendix/from-subversion-to-git, Last Accessed 25th April 2017

Stackoverflow. (2017), On line publication, Developer survey results, http://stackoverflow.com/insights/survey/2017#work-version-control, Last Accessed 25th April 2017

4 Appendix