302

Development Practice and Quality Assurance

Dr Robert Chatley - rbc@doc.ic.ac.uk

@rchatley **#doc302**

In this section we will talk about and demonstrate technical practices that modern development teams commonly undertake in order to deliver software smoothly and reliably as a team, and quality assurance practices that they follow to ensure that their software is always in a working state.

Version Control

"The first thing you should be told when you start a new job…" - Steve Freeman

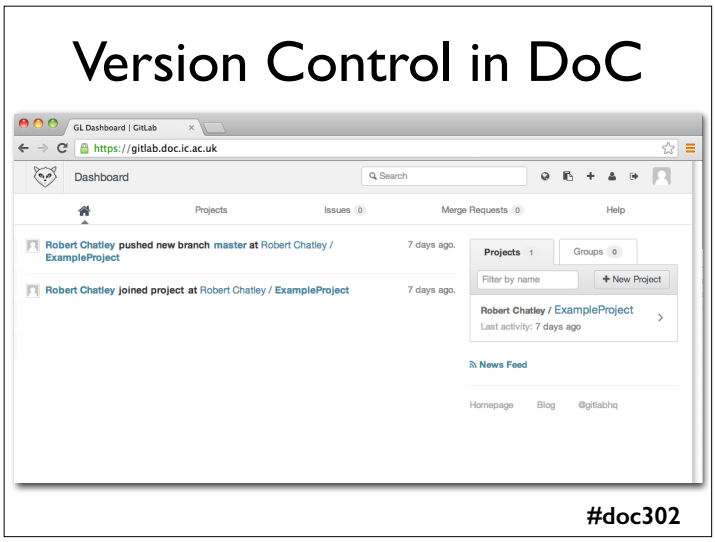
#doc302

Version control (sometimes referred to as "Software Configuration Management - SCM") is fundamental to team development. It allows teams of developers to work together on a project, to track everything that got changed at every point, to revert changes, go back to a known good version, or search through the project's history to find out how things used to be, and who changed them.

- CVS
- Subversion (svn)
- ClearCase
- Perforce
- Mercurial
- Darcs
- Bazaar
- Git

#doc302

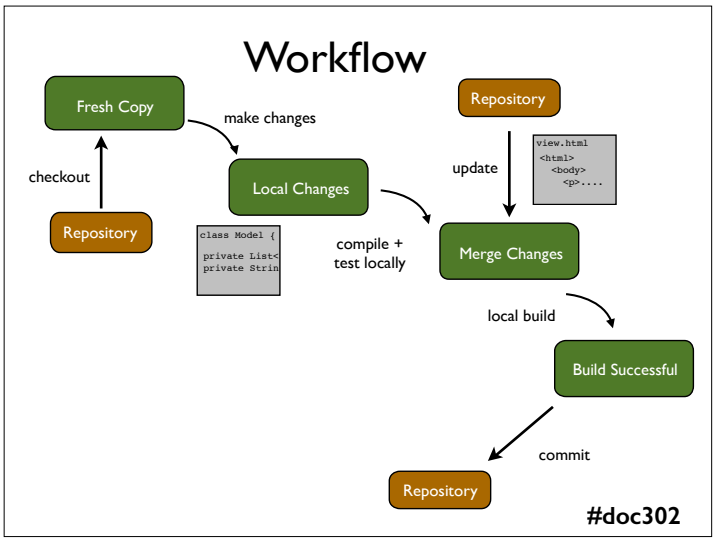
There are many different version control tools available, some commercial, some free. They typically fall into two categories: centralised (those listed in the left column above), or distributed (those listed in the right hand column).



In DoC the favoured version control system is Git. In order to aid collaboration, you are encouraged to use the department's GitLab service (which is like an internal version of GitHub), or indeed GitHub itself. Set up groups or shared projects to allow you to work collaboratively on your codebase.

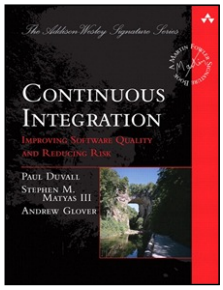
Some teams like to have different branches for working on different features, to keep them separate. If too many different features are worked on at once, different people can end up with different versions of the software on their machines (or SCM branches). Merging back together is often difficult, time consuming and painful. Particularly if we are releasing often, it causes a lot of problems if we have different parallel versions of the code in development.

When things are painful, we try to do them more often. Continuously integrating your changes into the master copy reduces merge problems and gives us one true version of the latest code.



For successful team development using version control as a collaboration mechanism, the normal workflow is as follows: a developer checks out the latest version of the code from the repository, then makes the necessary changes to the code to implement the feature they are working on. They build and test the system locally on their machine. Then they update from the repository to see if their colleagues have made any changes whilst they have been working. If there are changes to be merged in you should build and test the code again once these have been applied. Once you have a successful build, commit your changes back to the repository so that they are available to everyone else.

Continuous Integration



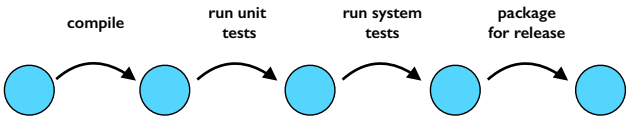
#doc302

"Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible." - Martin Fowler - http://www.martinfowler.com/articles/continuousIntegration.html . It is one of the practices that make up Extreme Programming. - http://www.extremeprogramming.org/rules/integrateoften.html

Continuous Integration is a practice, not a tool, although there are tools that can help. In this article, James Shore explains how to do Continuous Integration with an old workstation, a rubber chicken, and a bell. - http://www.jamesshore.com/Blog/Continuous-Integration-on-a-Dollar-a-Day.html

---

Automated Build



Write a script to perform all of these steps from one command - "one button build"

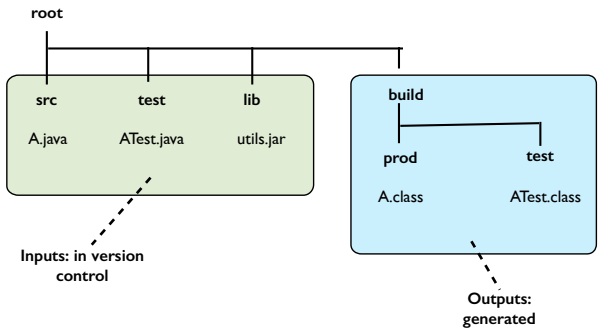
Stop as soon as any step fails

Build tools help: Ant, NAnt, Rake, MSBuild etc

#doc302

Key to being able to build and test your code easily and reliably is to automate this process. There are many tools available for automating build processes for different languages. Another benefit of having an automated build is that everyone builds the project in the same way, and the build files can be checked in to version control with the source code, so that whenever a new developer checks out the code, they can build it easily in one step.

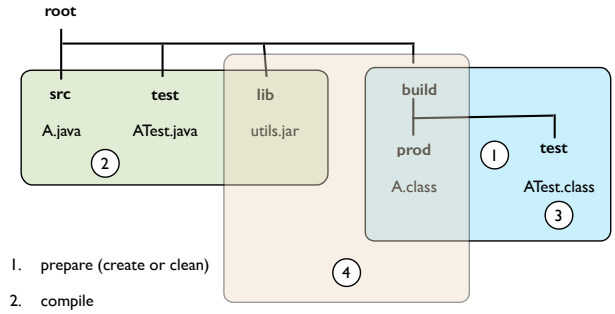
Automated Build: Project Structure



#doc302

A typical project structure is to separate production code from test code and library code using separate directories. Built code is (in a compiled language) separate from source code, and normally source code and tests are not built into artifacts that are to be deployed.

Automated Build: Stages

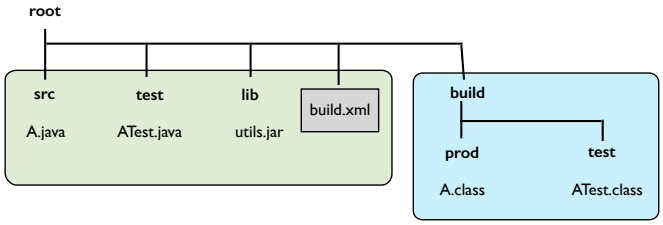


1. prepare (create or clean)
2. compile
3. test
4. package

#doc302

A typical build process will: a) create the relevant directory structure, and clean out any generated files remaining from the last build. b) compile the latest source code and tests c) run the tests against the generated binaries d) if all goes well, package the binaries up into a form to be deployed or distributed.

Automated Build: Using Ant




```
$ ls
build.xml lib src test
$ ant compile
```

#doc302

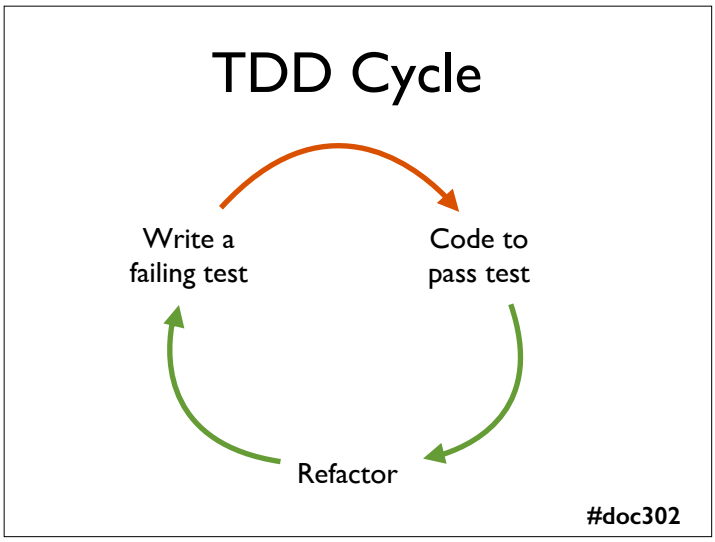
Ant is an example of a build tool for Java projects that is quite commonly used, but many other tools are available. The instructions for the build are encoded in an ant file, in XML format, which is conventionally called build.xml. In the build file you define different targets for different stages of the build, and these can be triggered individually using the command line ant tool.

Using Ant: build.xml

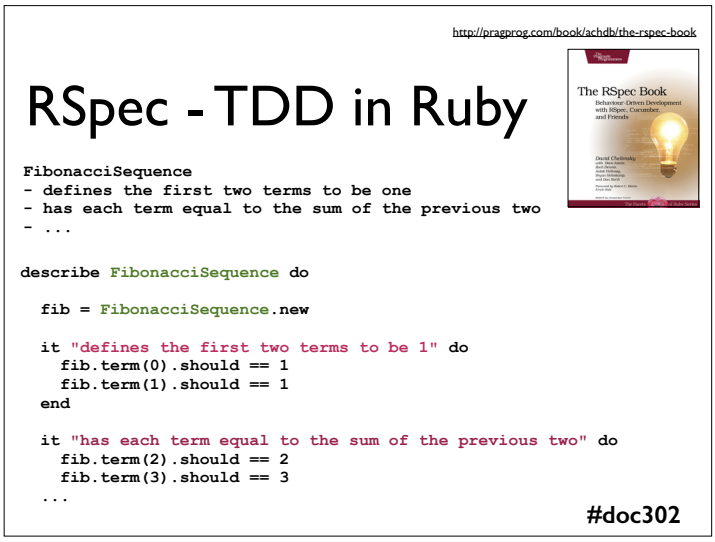
```
<project name="MyProject">

  <property name="src.dir" value="src" />
  <property name="build.dir" value="build/classes" />
  <property name="lib.dir" value="lib" />

  <path id="project.classpath">
     <pathelement location="${build.dir}" />
     <fileset dir="${lib.dir}" includes="*.jar"/>
  </path>

  <target name="prepare">
    <mkdir dir="${build.dir}" />
  </target>

  <target name="compile" depends="prepare">
    <javac srcdir="${src.dir}" destdir="${build.dir}">
      <classpath refid="project.classpath" />
    </javac>
  </target>

</project>
```

#doc302

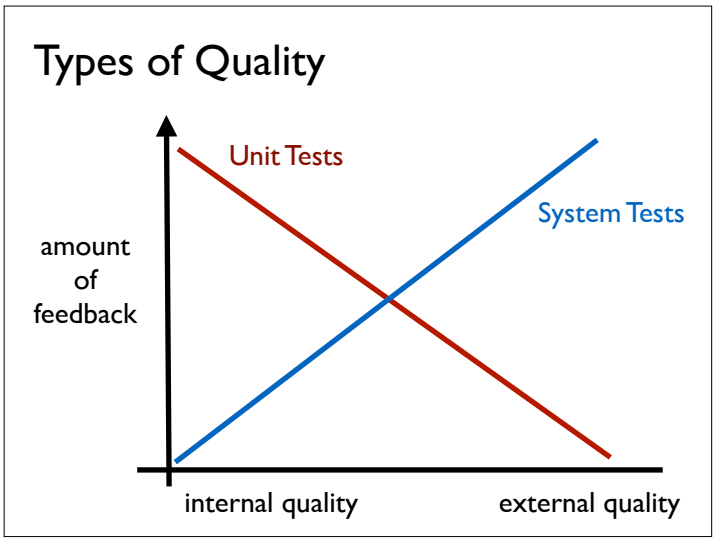
The above shows an example of an Ant build file. Different stages of the build are defined as targets. Targets can be specified as depending on other targets, so that running a later target in the chain will cause the earlier ones to be triggered first, so running the tests will first compile the code.



When following the Test-Driven Development (TDD) practice, we work in a cycle. We start by writing a test. We write the test first, before writing the implementation. This helps us to specify what we want the code we are about to write to do. How do we expect it will behave when it works properly? Once we have written a test, we watch it fail. This is expected, as we haven't implemented the feature yet. Then we write the simplest possible piece of code we can to make the test pass. After this we *refactor* our design to clean up, remove any duplication, improve clarity etc etc. Then we being the cycle again. When working with unit tests this cycle should be short, and provide us very rapid feedback about our code.

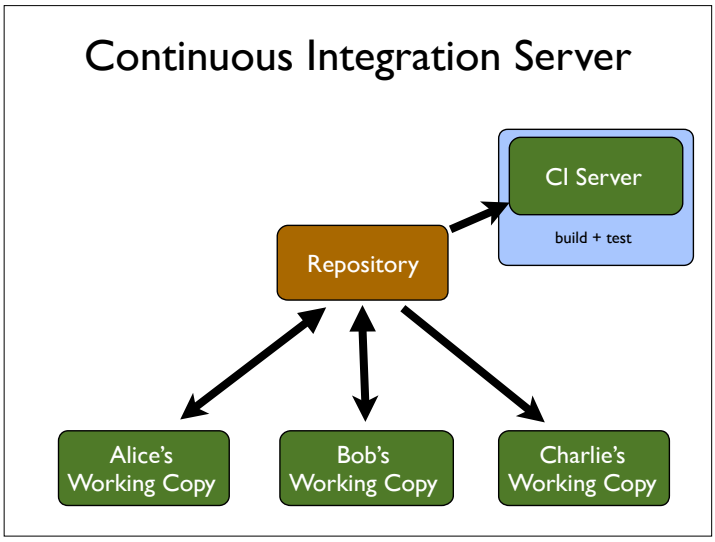


TDD is practised widely in current software development. There are unit testing tools for almost every language, and we can apply the principles of TDD to help ensure quality regardless of the language we are working in. Here we show an example in Ruby using RSpec.

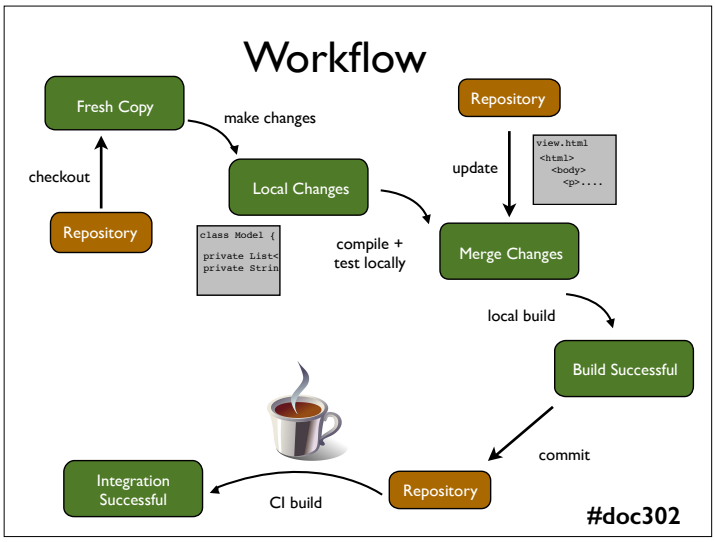


Two different types of test, unit and system, give us two different types of information about the quality of our system. Unit tests give us a lot of feedback about the quality of our code - they help us gauge the internal quality of our system, but they don't really tell us much about whether the system as a whole does anything useful for the user. System level tests give much more feedback on the external quality - does the system behave correctly as the user sees it, but don't tell us much about the quality of the code inside.

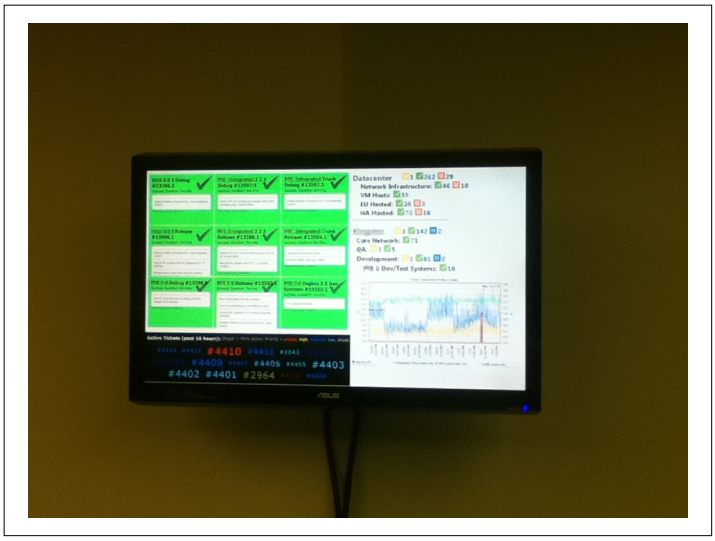
If you set up the build and the project structure carefully at the beginning of the project, then normally it doesn't need to be changed too much during day to day development. But, it is important to take care of the build. As more and more code, and especially tests, are added to the project the build can slow down. If it takes more than 10 minutes to run then people become reluctant to run the build, and then do not benefit from the feedback it provides. The quality of feedback is also reduced by builds that fail for spurious reasons - people begin to expect, or ignore, failures.



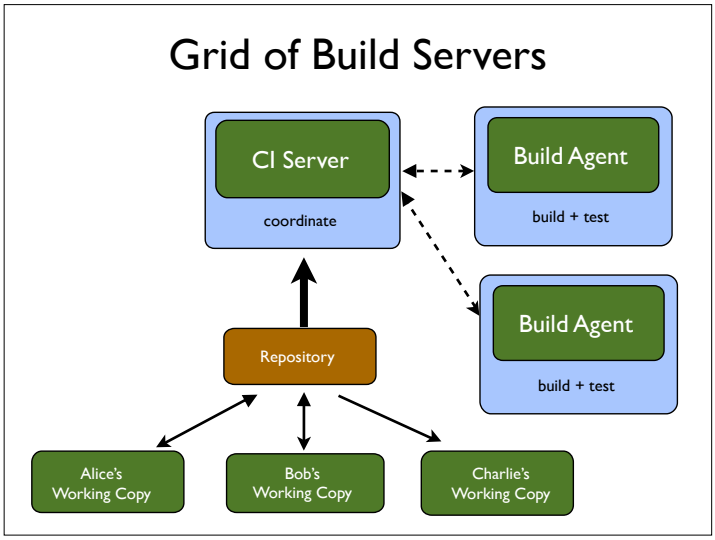
Sometimes, on larger projects with many frequent commits, it is difficult to run a full build locally on every commit. A secondary check, and perhaps a larger test suite, can be run by a Continuous Integration (CI) server. The CI server can also gather data and statistics on changes that have been made, test failures, fixes etc etc. The CI server can be set up to build on a regular schedule, or to watch the repository and run when it detects a change.



When we have a CI server, our workflow is extended so that the CI build runs after we commit our changes. If the CI build takes around 10 minutes or less to run, then we can take a short break, get a cup of coffee, and reflect on our work while it runs. Then, when we have confirmed that the CI build was successful, we can move on to the next feature.

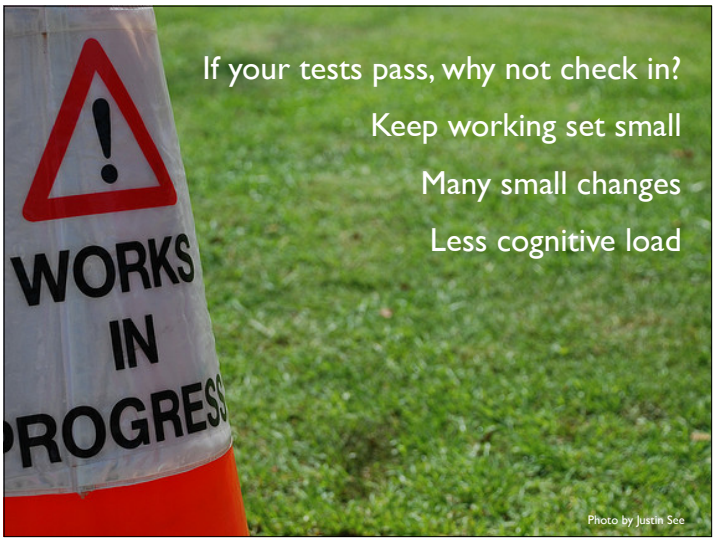


Teams often use a visual signal, such as a display screen, to show the status of their builds, and maybe other builds that they depend on. Then if they notice that a certain build fails - normally shown boldly in red - then they can turn their attention to fixing it before continue to work on new features.



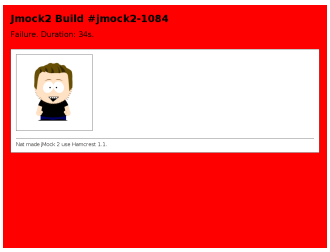
For teams that have a lot of CI builds to run - perhaps testing on different platforms - a grid of CI servers or agents can be useful. CI servers such as TeamCity or Jenkins support this without too much work. Parallelising the builds and farming them out to different agents can speed up the rate at which feedback is obtained a lot.

If building and testing the code for your project is painful - perhaps it has some manual steps, or tests often fail - then trying to do it more often highlights the pain points, and encourages us to fix them using automation and improving the tests.



If we make large changes, and try to check in once a day, or less often, we often have to fix large merges, and cause lots of tests to fail which have to be fixed before we can commit. If we have a quick build, and work in small change sets then it is much easier to keep development flowing. You don't have to wait until you have a complete feature before checking in, you can work in small steps, committing after each, as long as your build passes.

CI Culture



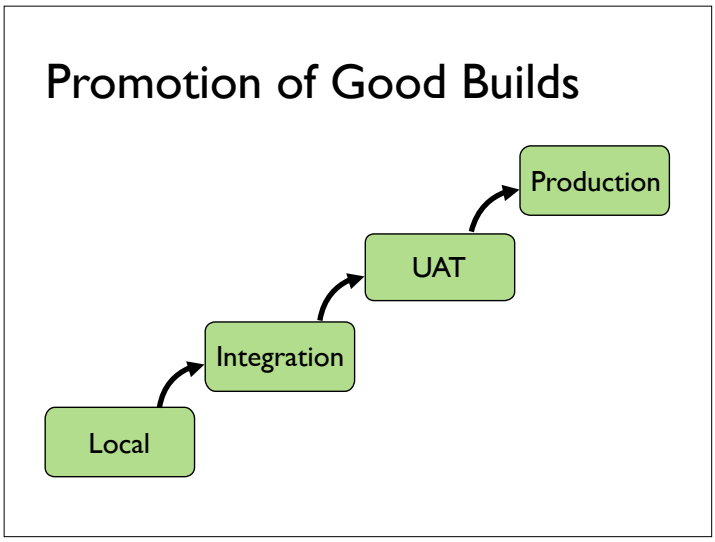
Don't break the build

If the build is broken
- fix it
- don't commit unless you are fixing it

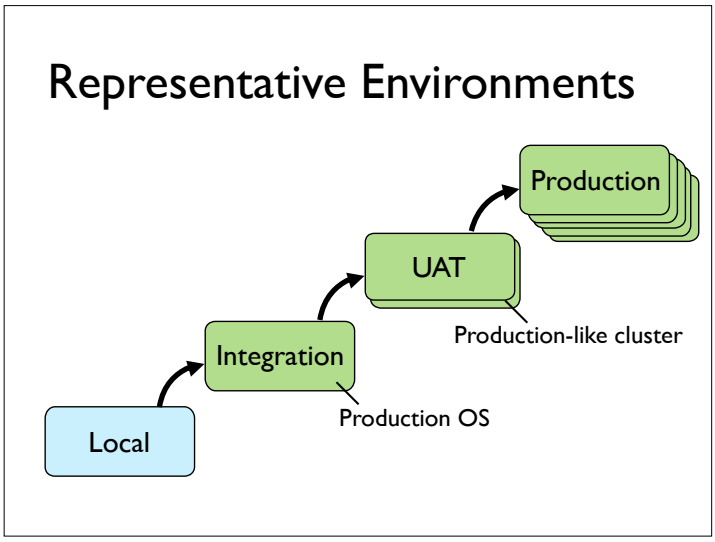
#doc302

The main rule for development teams using CI is "don't break the build". Breaking the build affects other team members and prevents them from getting their work done, so run tests locally before you check in to make sure that everything works. It is inevitable that the build will break once in a while. When this happens, the team should concentrate on fixing it before committing more changes. If the build stays red for a long time then people start to ignore it and its value is lost.

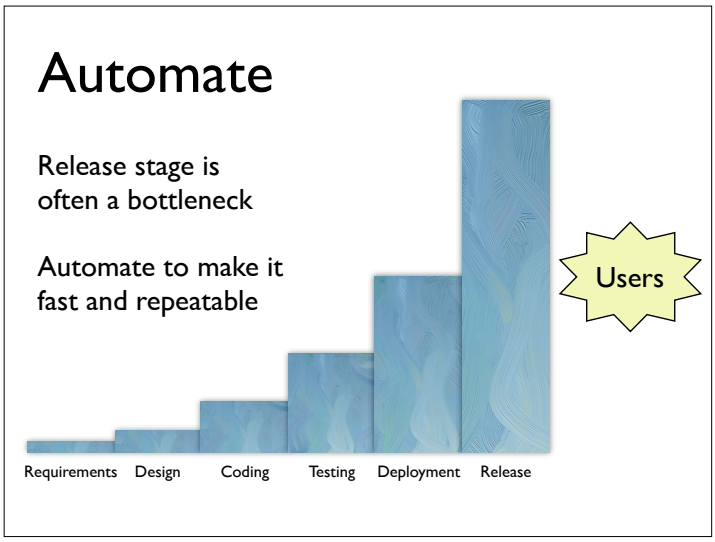
Release is often one of the points of a project that causes most stress. Pushing software into production to be used by real users is unfortunately often the point where problems come up and fixes have to be made.



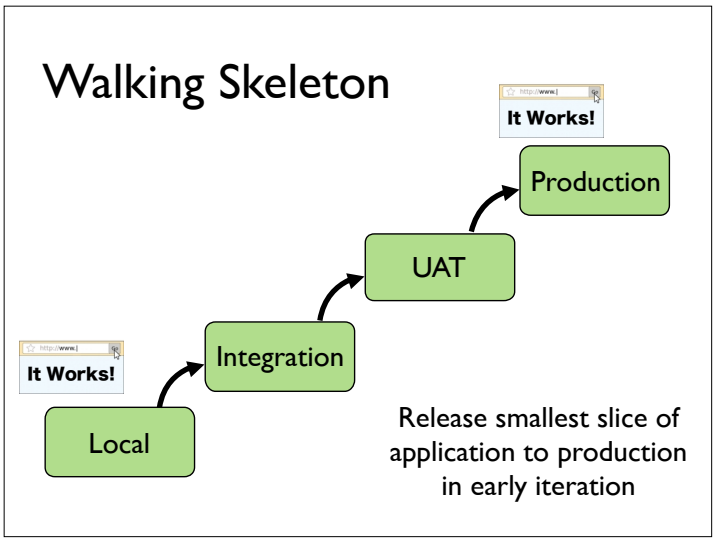
We don't want to put code straight into production - we want to test it first in various pre-production environments, and promote builds through this chain when we are confident that they are of good quality.



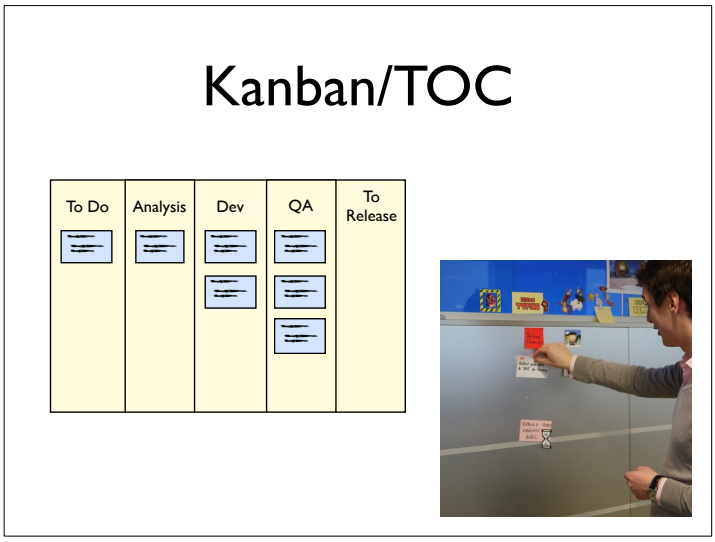
Often developers' local environments differ substantially from production environments. They may differ in architecture, OS, network infrastructure etc etc. We aim to test in a production-like environment as early as possible.



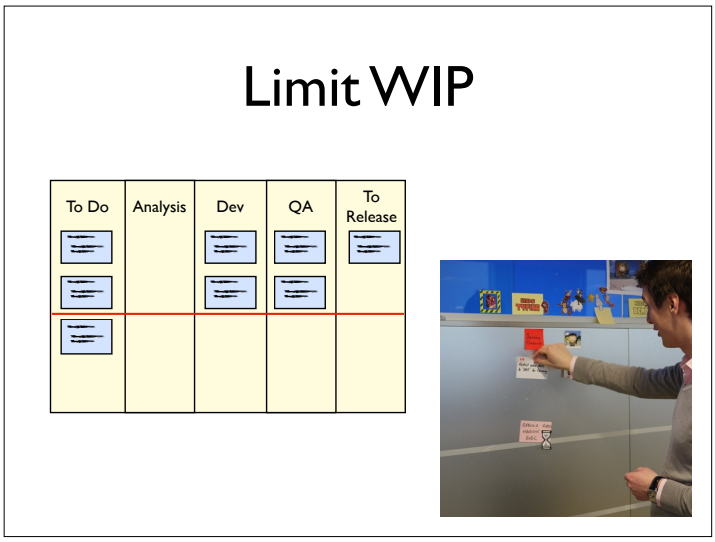
It doesn't matter how much code you have written and how many features you have implemented it if you can't get it into production. If no-one can use it then it isn't complete. Release processes, especially manual ones, are often a bottleneck.



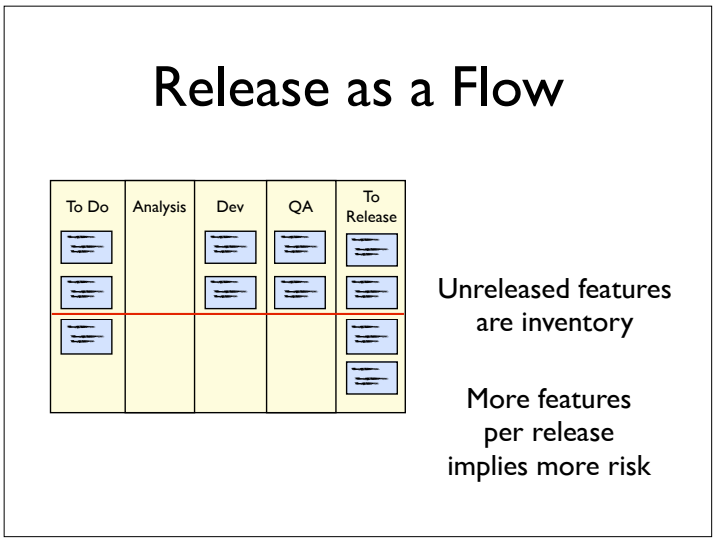
Trying to release to production often reveals problems and incurs delays. If you leave this right to the end of your project then it can have a big effect on whether the project is delivered on time. We aim to iron out these difficulties as early as possible in the project by deploying a walking skeleton in the first iteration.



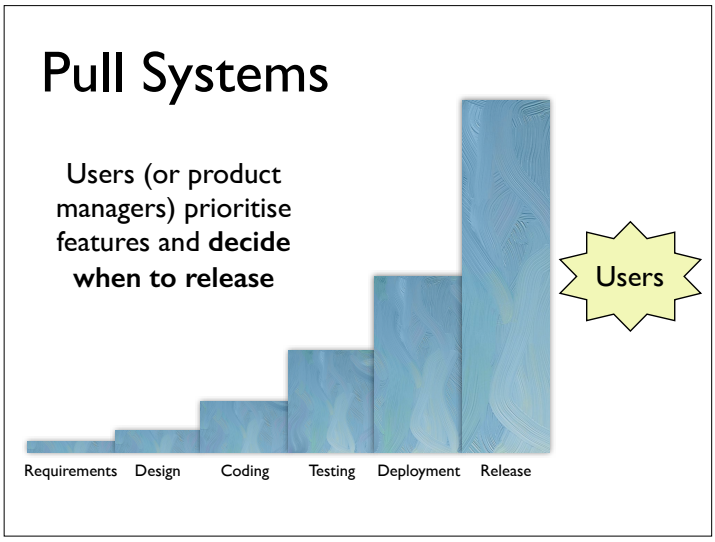
This kanban board shows that a number of stories have been developed, but they are backing up in QA - none have been released. The team should focus on getting these features through QA and into production before doing more dev or analysis work.



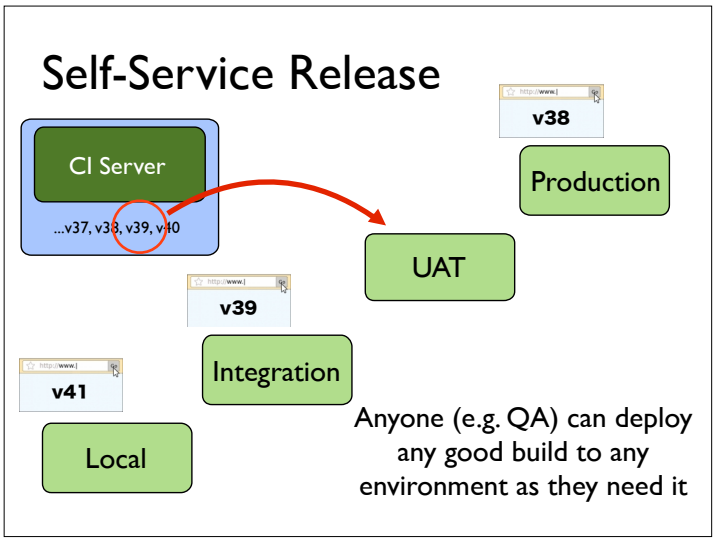
One way to help highlight these problems is to enforce a work-in-progress (WIP) limit on each activity. You can only pull a story into the next phase when there is a space to do so.



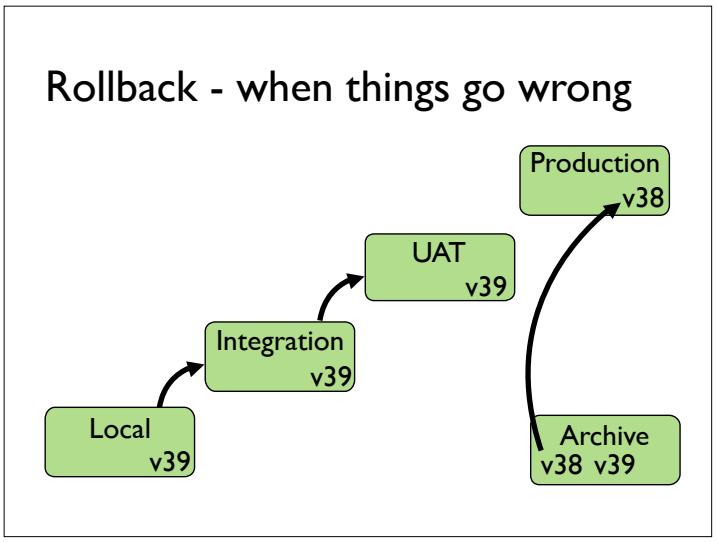
Keeping the work in progress low and releasing often, a few features at a time helps to keep customers happy as they see a continuous flow of delivery. It also means that we have less half-developed features to maintain, and smaller changes in each release.



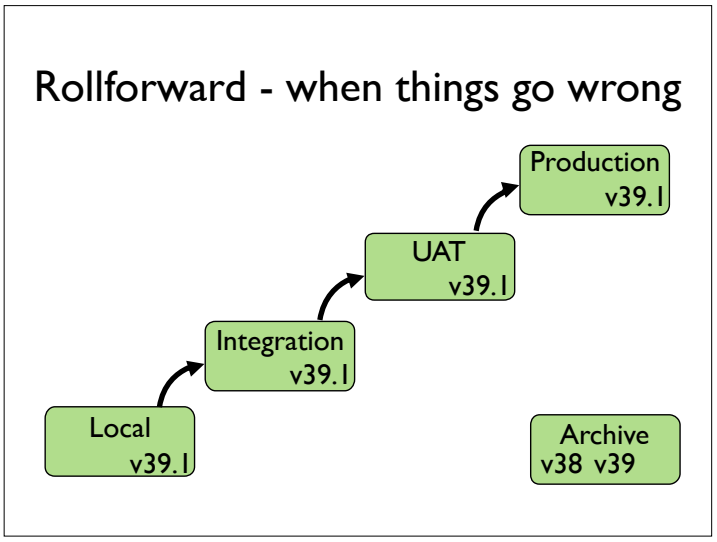
In an agile process, prioritisation of features for the team to work on is done by the customer. If we integrate continuously and always have a releasable (tested) build, then the business can decide when they want to release - when they are happy with the functionality.



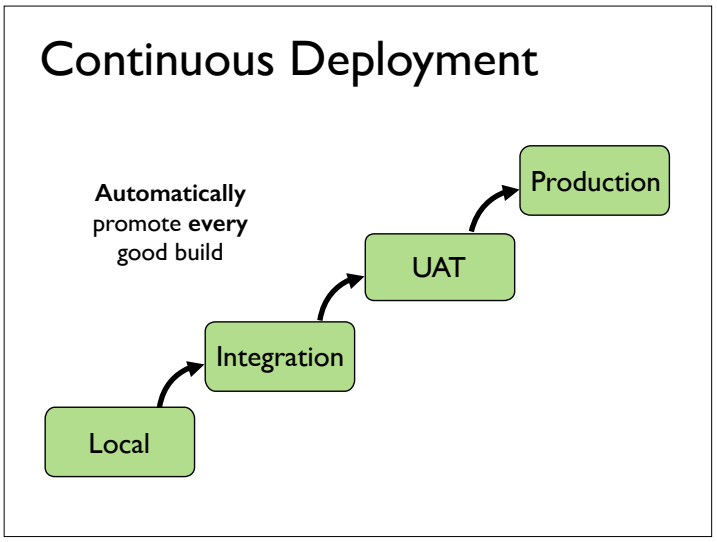
If we automate the release fully, it can be a self-service operation. If the QA team wants to release a particular version to a UAT server, they just go ahead and do that. Ideally if the business want to promote a version to production, they should be able to do that whenever they want. IT should not be a bottleneck.



If we quickly notice that things are not working correctly with the new version, we can roll back to the previously released version. It is a good idea to make sure that this is easy to do as the times that you need to do this are often very stressful.



Sometimes it is not appropriate or possible to rollback. In these cases it is sometimes tempting to make changes on production. This is very risky. If your release procedure is smooth and automated you are better to create a new version with the fix and roll forward through the normal procedure.



Under the continuous deployment model, we release very often, perhaps many times per day, each time a feature is ready. If all of the automated tests pass, the system automatically deploys the build to production. This gives us very rapid feedback.

This model has gained some popularity, particularly in the startup world. You might have good reasons why you wouldn't want to deploy all the way to production automatically following a commit, but it is a useful thought experiment to try and decide why not.