220

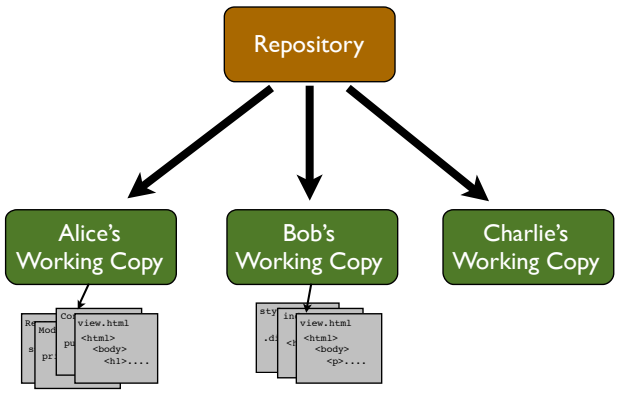
Continuous Integration

Dr Robert Chatley - rbc@imperial.ac.uk

@rchatley #doc220

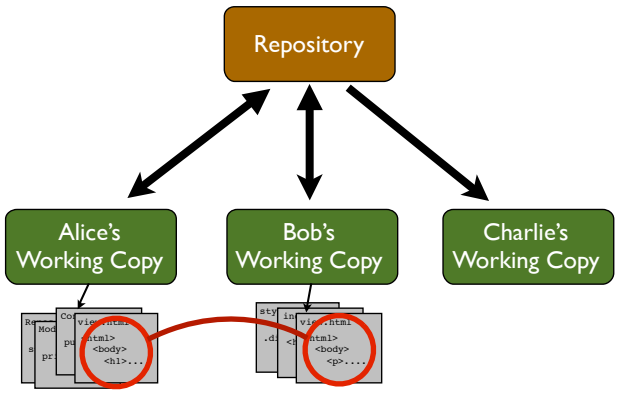
In this section we look at practical tools and techniques for developing applications collaboratively in a team environment, supporting the frequent delivery of features to customers. We discuss how using tools to support your development can help to ensure that you maintain good quality throughout your project.

Concurrent Development



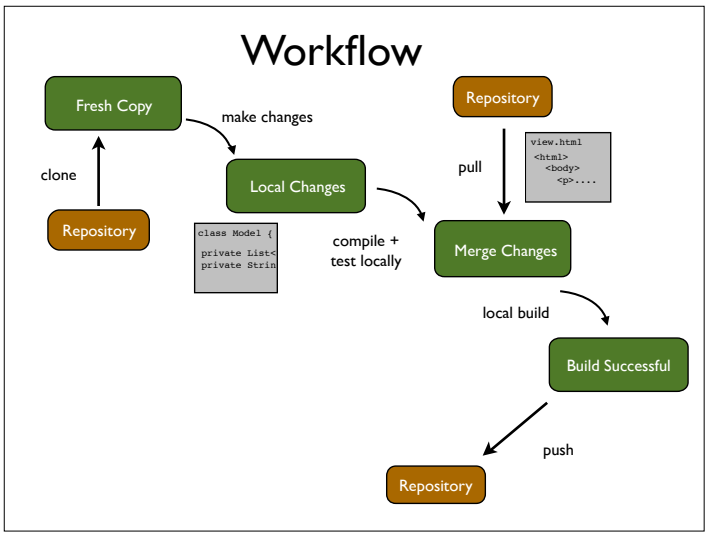
When working in a team environment, different developers will make changes to the same codebase in parallel. They will each checkout a working copy of the code onto their local machine, and develop and test locally there. When they have completed a change, and are confident that their code works, they will commit it back to the shared repository.

Conflicting Changes

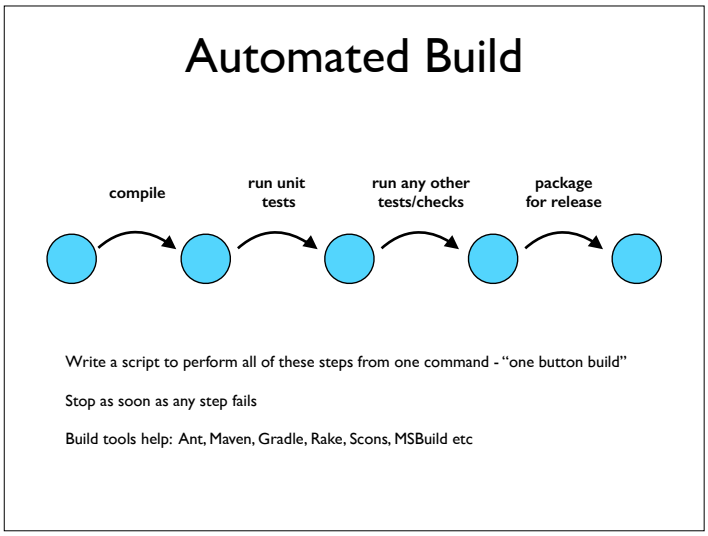


While sometimes different developers work in different areas of the code, some files tend to be touched often and so it may be that two or more developers make changes in the same area of code at the same time. This creates conflicts which need to be resolved before the code can be committed.

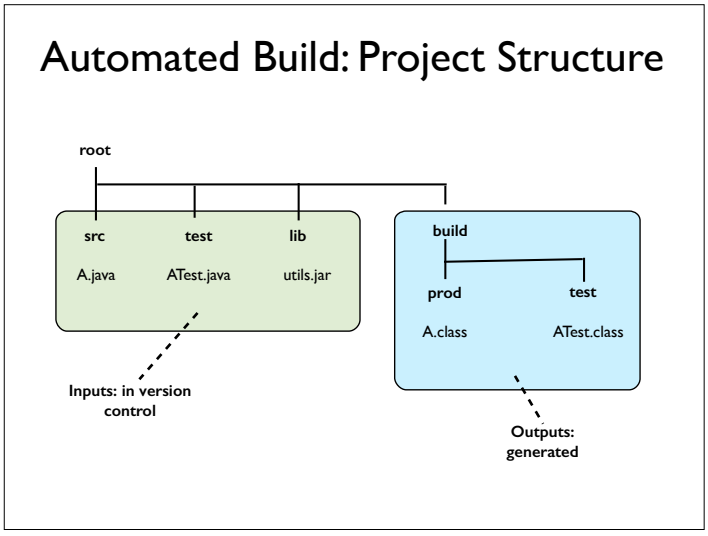
Merging conflicting versions of code can be very painful, especially if change sets are large. You may come across tricky cases such as where while you are making changes to a particular class, someone else has refactored the code and deleted that class from the system. To combat this sort of conflict it pays to frequently and continually commit small changes back to the repository, and keep synchronised with the master version.



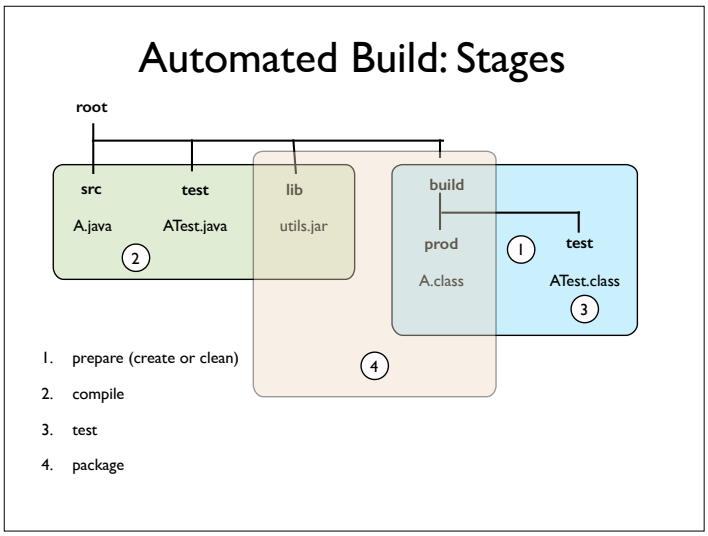
For successful team development using version control as a collaboration mechanism, the normal workflow is as follows: a developer checks out the latest version of the code from the repository, then makes the necessary changes to the code to implement the feature they are working on. They build and test 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.



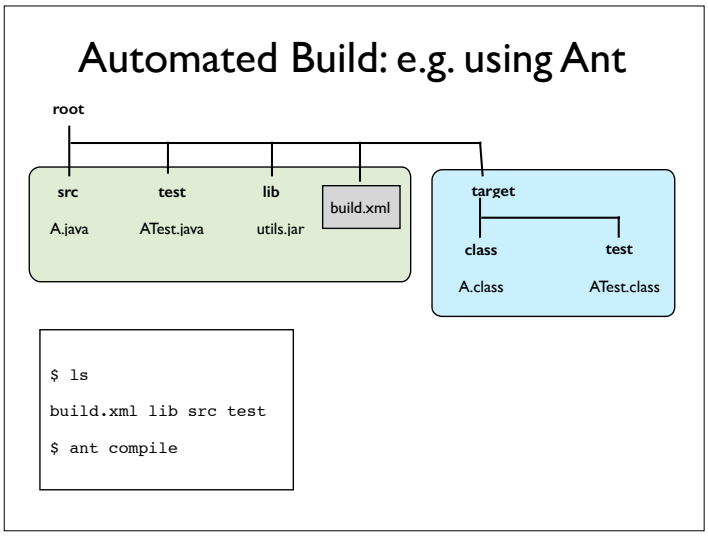
Key to being able to build and test your code easily and reliably is to automate this process. There are many tools available for automating build processes for different languages. Another benefit of having an automated build is that everyone builds the project in the same way, and the build files can 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.



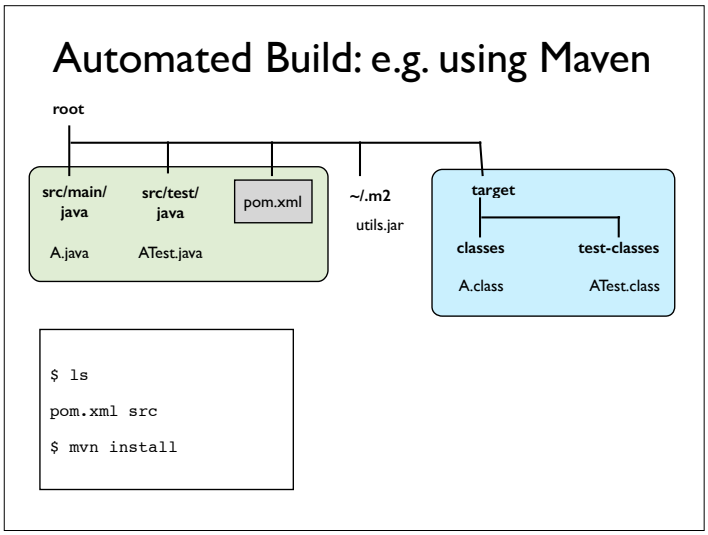
A typical project structure is to separate production code from test code and library code using separate directories. Built code is (in a compiled language) separate from source code, and normally source code and tests are not built into artifacts that are to be deployed.



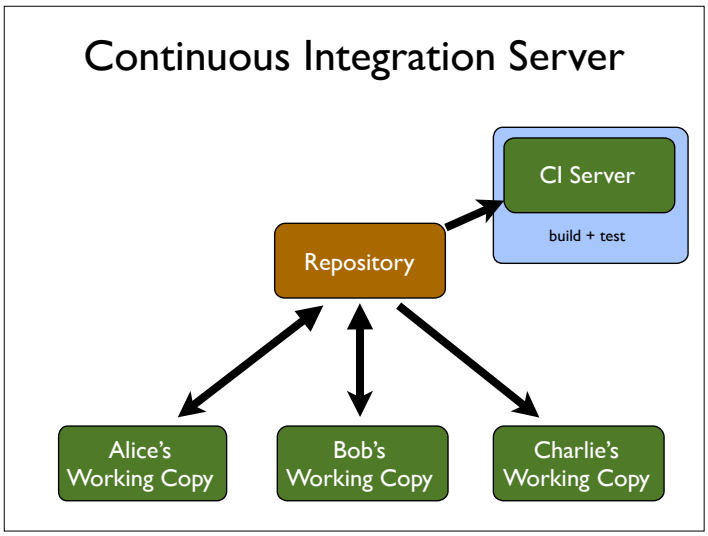
A typical build process will: a) create the relevant directory structure, and clean out any generated files remaining from the last build. b) compile the latest source code and tests c) run the tests against the generated binaries d) if all goes well, package the binaries up into a form to be deployed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



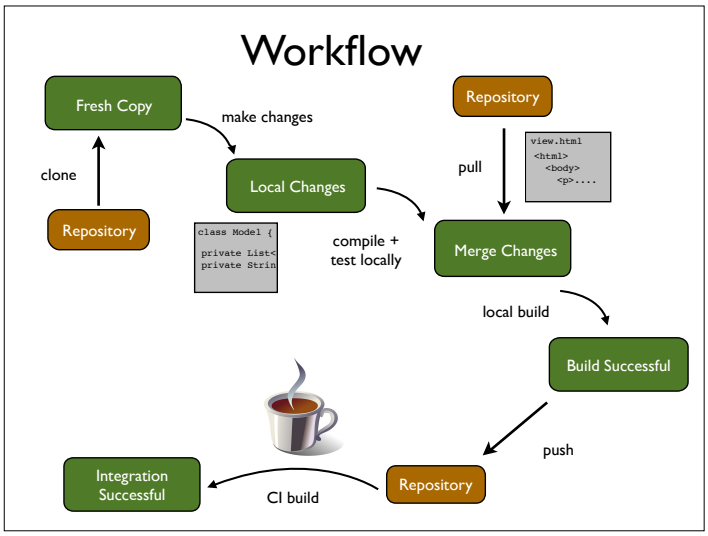
Ant is an example of a simple build tool for Java projects that is quite commonly used, although it is becoming a bit out of date now. The instructions for the build are encoded in an ant file, in XML format, which is conventionally called build.xml. In the build file you define different targets for different stages of the build, and these can be triggered individually using the command line ant tool.



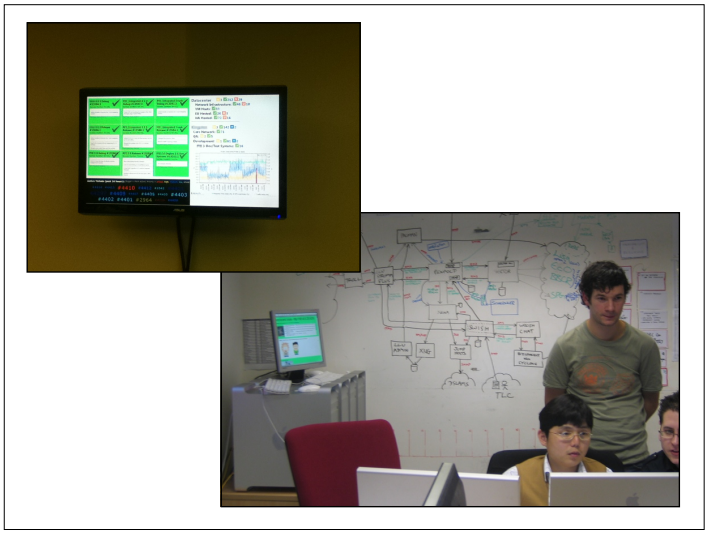
Maven is a more common tool nowadays (although other tools are overtaking it, like Gradle) which works in a very similar way. One notable difference is that with Maven builds, we do not check the libraries we need into version control, instead, the pom.xml includes a section saying what dependencies we rely on, and when the build runs, they are downloaded from a repository on the internet onto the local machine where the build is being run.



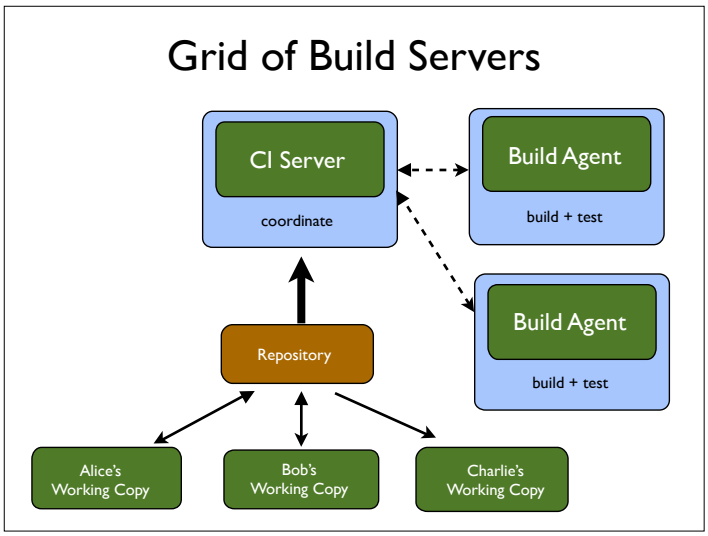
Sometimes, on larger projects with many frequent commits, it is difficult to run a full build locally on every commit. A secondary check, and perhaps a larger test suite, can be run by a Continuous Integration (CI) server. The CI server can also gather data and statistics on changes that have been made, 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.



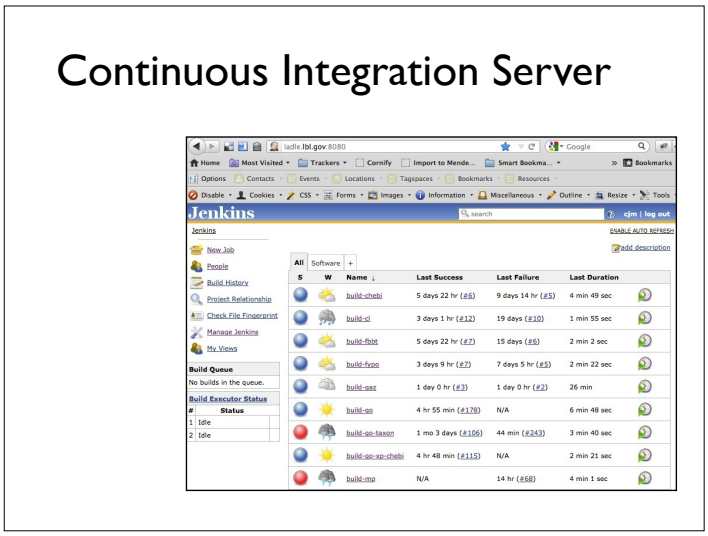
When we add a Continuous Integration (CI) server, our workflow is extended so that the CI build runs after we commit our changes. If the CI build takes around 10 minutes or less to run, then we can take a short break, get a cup of coffee, and reflect on our work while it runs. Then, when we have confirmed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



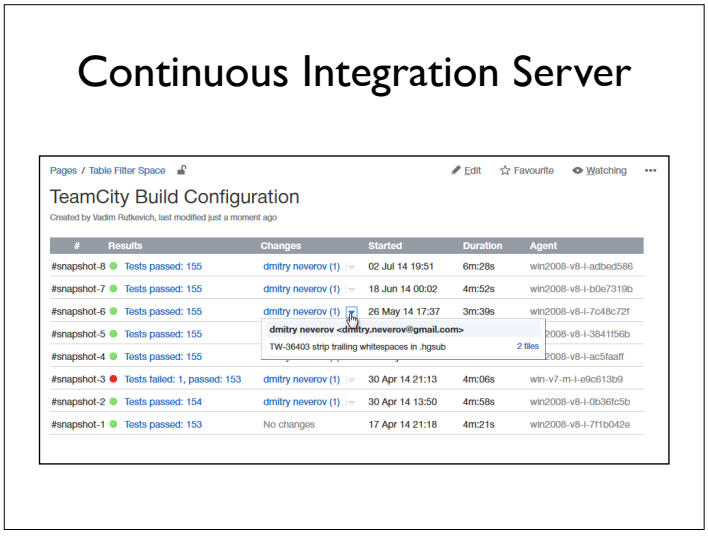
Teams often use a visual signal, such as a display screen, to show the status of their builds, and maybe other builds that they depend on. Then if they notice that a certain build fails - normally shown boldly in red - then they can turn their attention to fixing it before continue to work on new features.



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.



Here is a screenshot from Jenkins - a popular open source CI server.



Here is a screenshot from TeamCity - a popular CI server by JetBrains (who also make IntelliJ).