Continuous Integration

In software engineering, **continuous integration** (**CI**) implements *continuous* processes of applying quality control — small pieces of effort, applied frequently. Continuous integration aims to improve the quality of software, and to reduce the time taken to deliver it, by replacing the traditional practice of applying quality control *after* completing all development.

Theory

When embarking on a change, a developer takes a copy of the current code base on which to work. As other developers submit changed code to the source code repository, this copy gradually ceases to reflect the repository code. Not only can the existing code base change, but new code can be added as well as new libraries, and other resources that create dependencies, and potential conflicts.

The longer a branch of code remains checked out, the greater the risk of multiple integration conflicts and failures becomes when it is reintegrated into the main line. When developers submit code to the repository they must first update their code to reflect the changes in the repository since they took their copy. The more changes the repository contains, the more work developers must do before submitting their own changes. Eventually, the repository may become so different from the developers' baselines that they enter what is sometimes called "integration hell",[1] where the time it takes to integrate exceeds the time it took to make their original changes. In a worst-case scenario, developers may have to discard their changes and completely redo the work.

Continuous integration involves integrating early and often, so as to avoid the pitfalls of "integration hell". The practice aims to reduce rework and thus reduce cost and time.

A complementary practice to CI is that before submitting work, each programmer must do a complete build and run (and pass) all unit tests. Integration tests are usually run automatically on a CI server when it detects a new commit. All programmers should start the day by updating the project from the repository. That way, they will all stay up-to-date.

The rest of this article discusses best practice in how to achieve continuous integration, and how to automate this practice. Automation is a best practice itself.[2][3]

Principles of continuous integration

Continuous integration – the practice of frequently integrating one's new or changed code with the existing code repository – should occur frequently enough that no intervening window remains between commit and build, and such that no errors can arise without developers noticing them and correcting them immediately.[4]

Normal practice is to trigger these builds by every commit to a repository, rather than a periodically scheduled build. The practicalities of doing this in a multi-developer environment of rapid commits are such that it's usual to trigger a short timer after each commit, then to start a build when either this timer expires, or after a rather longer interval since the last build. Automated tools such as CruiseControl, Jenkins, Hudson, Bamboo, BuildMaster, AnthillPro or Teamcity offer this scheduling automatically.

Another factor is the need for a version control system that supports atomic commits, i.e. all of a developer's changes may be seen as a single commit operation. There is no point in trying to build from only half of the changed files.

To achieve these objectives, continuous integration relies on the following principles.

Maintain a code repository

Main article: Revision control

This practice advocates the use of a revision control system for the project's source code. All artifacts required to build the project should be placed in the repository. In this practice and in the revision control community, the convention is that the system should be buildable from a fresh checkout and not require additional dependencies. Extreme Programming advocate Martin Fowler also mentions that where branching is supported by tools, its use should be minimized.[4] Instead, it is preferred for changes to be integrated rather than for multiple versions of the software to be maintained simultaneously. The mainline (or trunk) should be the place for the working version of the software.

Automate the build

Main article: Build automation

A single command should have the capability of building the system. Many build-tools, such as make, have existed for many years. Other more recent tools like Ant, Maven, MSBuild, OpenMake Meister or IBM Rational Build Forge are frequently used in continuous integration environments. Automation of the build should include automating the integration, which often includes deployment into a production-like environment. In many cases, the build script not only compiles binaries, but also generates documentation, website pages, statistics and distribution media (such as Debian DEB, Red Hat RPM or Windows MSI files).

Make the build self-testing

Once the code is built, all tests should run to confirm that it behaves as the developers expect it to behave.

Everyone commits to the baseline every day

By committing regularly, every committer can reduce the number of conflicting changes. Checking in a week's worth of work runs the risk of conflicting with other features and can be very difficult to resolve. Early, small conflicts in an area of the system cause team members to communicate about the change they are making. Committing all changes at least once a day (once per feature built) is generally considered part of the definition of Continuous Integration. In addition performing a nightly build is generally recommended.[*citation needed*] These are lower bounds, the typical frequency is expected to be much higher.

Every commit (to baseline) should be built

The system should build commits to the current working version in order to verify that they integrate correctly. A common practice is to use Automated Continuous Integration, although this may be done manually. For many, continuous integration is synonymous with using Automated Continuous Integration where a continuous integration server or daemon monitors the version control system for changes, then automatically runs the build process.

Keep the build fast

The build needs to complete rapidly, so that if there is a problem with integration, it is quickly identified.

Test in a clone of the production environment

Having a test environment can lead to failures in tested systems when they deploy in the production environment, because the production environment may differ from the test environment in a significant way. However, building a replica of a production environment is cost prohibitive. Instead, the pre-production environment should be built to be a scalable version of the actual production environment to both alleviate costs while maintaining technology stack composition and nuances.

Make it easy to get the latest deliverables

Making builds readily available to stakeholders and testers can reduce the amount of rework necessary when

rebuilding a feature that doesn't meet requirements. Additionally, early testing reduces the chances that defects survive until deployment. Finding errors earlier also, in some cases, reduces the amount of work necessary to resolve them.

Everyone can see the results of the latest build

It should be easy to find out whether the build breaks and, if so, who made the relevant change.

Automate deployment

Most CI systems allow the running of scripts after a build finishes. In most situations, it is possible to write a script to deploy the application to a live test server that everyone can look at. A further advance in this way of thinking is Continuous Deployment, which calls for the software to be deployed directly into production, often with additional automation to prevent defects or regressions.[5][6]

History

Continuous Integration emerged in the Extreme Programming (XP) community, and XP advocates Martin Fowler and Kent Beck first wrote about continuous integration circa 1999. Fowler's paper[7] is a popular source of information on the subject. Beck's book *Extreme Programming Explained*,[8] the original reference for Extreme Programming, also describes the term.

Advantages and disadvantages

Advantages

Continuous integration has many advantages:

- when unit tests fail or a bug emerges, developers might revert the codebase to a bug-free state, without wasting time debugging
- developers detect and fix integration problems continuously - avoiding last-minute chaos at release dates, (when everyone tries to check in their slightly incompatible versions).
- early warning of broken/incompatible code
- early warning of conflicting changes
- immediate unit testing of all changes
- constant availability of a "current" build for testing, demo, or release purposes
- immediate feedback to developers on the quality, functionality, or system-wide impact of code they are writing
- frequent code check-in pushes developers to create modular, less complex code[*citation needed*]
- metrics generated from automated testing and CI (such as metrics for code coverage, code complexity, and features complete) focus developers on developing functional, quality code, and help develop momentum in a team[*citation needed*]

Disadvantages

- initial setup time required
- well-developed test-suite required to achieve automated testing advantages
- hardware costs for build machines can be significant[*citation needed*]

Many teams using CI report that the advantages of CI well outweigh the disadvantages.[9] The effect of finding and fixing integration bugs early in the development process saves both time and money over the lifespan of a project.

Software

Main article: comparison of continuous integration software

Software tools for continuous integration include:

- AnthillPro: continuous integration server by Urbancode
- Apache Continuum: continuous integration server supporting Apache Maven and Apache Ant
- Apache Gump: continuous integration tool by Apache
- Automated Build Studio: proprietary automated build, continuous integration and release management system by AutomatedQA
- Bamboo: proprietary continuous integration server by Atlassian Software Systems
- Bitten Automated Build: continuous build plugin for Trac
- BuildBot: Python / Twisted-based continuous build system
- BuildHive: free cloud-hosted continuous integration service for GitHub projects, based on Jenkins.
- BuildMaster: proprietary application lifecycle management and continuous integration tool by Inedo
- CABIE: Continuous Automated Build and Integration Environment: open source, written in Perl
- Cascade: proprietary continuous integration tool
- CDash: open source, web-based software testing server associated mostly with CMake but also being used for CI of the PHP-based Midas platform
- Circle CI: Hosted continuous integration for web applications
- codeBeamer: proprietary collaboration software with built-in continuous integration features
- CruiseControl: Java-based framework for a continuous build process
- CruiseControl.NET: .NET-based automated continuous integration server
- CruiseControl.rb: Lightweight, Ruby-based continuous integration server that can build any codebase, not only Ruby, released under Apache Licence 2.0
- ElectricCommander: proprietary continuous integration and release management solution from Electric Cloud
- FinalBuilder Server: proprietary automated build and continuous integration server by VSoft Technologies
- Go: proprietary agile build and release management software by Thoughtworks
- Hudson; MIT-licensed, written in Java
- Jenkins: (split from Hudson); MIT-licensed, written in Java
- Rational Team Concert: proprietary software development collaboration platform with built-in build engine by IBM
- SCLM: software configuration management system for z/OS by IBM Rational Software
- Tddium: Cloud-hosted elastic continuous integration service for Ruby and Javascript applications
- TeamCity: proprietary continuous-integration server by JetBrains with free professional edition
- Team Foundation Server: proprietary continuous integration server and source code repository by Microsoft
- Tinderbox: Mozilla-based product written in Perl
- Travis CI: A distributed build system for the open source community.
- VectorCAST/Manage: A proprietary automated regression testing and continuous integration tool by Vector Software

See also

- Build light indicator
- Build automation
- Continuous design
- Test-driven development
- Multi-stage continuous integration

- Comparisons of continuous-integration software

Further reading

- Duvall, Paul M. (2007). *Continuous Integration. Improving Software Quality and Reducing Risk*. Addison-Wesley. ISBN 0-321-33638-0.

References

1. ^ Cunningham, Ward (05 Aug 2009). "Integration Hell". *WikiWikiWeb*. Retrieved 19 Sept 2009.
2. ^ Brauneis, David (01 January 2010). "[OSLC Possible new Working Group - Automation"]. *open-services.net Community mailing list*. Retrieved 16 February 2010.
3. ^ Taylor, Bradley. "Rails Deployment and Automation with ShadowPuppet and Capistrano".
4. ^ ***a*** ***b*** Fowler, Martin. "Continuous Integration". Retrieved 2009-11-11.
5. ^ Continuous deployment in 5 easy steps
6. ^ Continuous Deployment at IMVU: Doing the impossible fifty times a day.
7. ^ Fowler, Martin. "Continuous Integration".
8. ^ Beck, Kent (1999). *Extreme Programming Explained*. ISBN 0-201-61641-6.
9. ^ Richardson, Jared (September 2008). "Agile Testing Strategies at No Fluff Just Stuff Conference". Boston, Massachusetts.

External links

- Continuous integration by Martin Fowler (an introduction)
- Continuous Integration at the Portland Pattern Repository (a collegial discussion)
- Cross platform testing at the Portland Pattern Repository
- Continuous Integration Server Feature Matrix (a guide to tools)
- Continuous Integration: The Cornerstone of a Great Shop by Jared Richardson (an introduction)
- A Recipe for Build Maintainability and Reusability by Jay Flowers
- Continuous Integration anti-patterns by Paul Duvall
- Extreme programming
- Version lifecycle MediaWiki