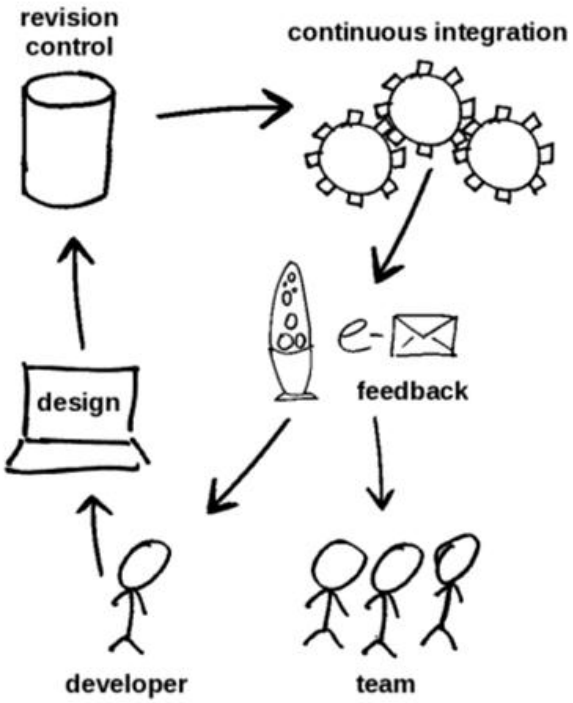
Continuous Integration (CI)

A survey of about 2000 researchers showed how coding has become important part of researcher's toolkit:

- 38 % spend at least one fifth of their time developing software
- But only 47 % have a good understanding of software testing

Source: Information taken from: Nature 467, 775-777 (2010) | doi:10.1038/467775a

CI is a development practice which requires developers to integrate code into a common repository multiple times a day. Each check-in is then verified by an automated build, allowing developers to detect problems early. By integrating in a regular basis, we can discover errors swiftly, and trace them effortlessly. Continuous Integration systems wrap version control, compilation, and testing into a single repeatable process. We create/debug code as usual. We check the code and then CI system builds code, tests it, and reports back to us.



Continuous Integration can refer the following:

- A practice of always building a working version of your system each day
- A practice of always running the test cases each day
- A practice of committing changes to the version control system each day
- An automated system that monitors changes to the version control system such that when it detects the commit:
    - it checks out that version of your system
    - builds it, runs all tests, and verifies that it passes
    - if it does, it creates a new official release
    - if it doesn't, it notifies the appropriate developers

Making CI Work

- To make continuous integration possible, we need
  i. a source code repository
  ii. a check-in process
  iii. an automated build process
  iv. a willingness to work incrementally
- The first two are enabled by version control
  - The most popular version control systems (git, mercurial) are known as distributed version control systems. This basically means that all developers have a "master copy" of the repository; developer has everything we need to do development on our local machine.
- Check-In Process
  - By a check-in process, the book means that each of our developers has a similar way in which they develop code
  - Grab the latest version of the system (however that's accomplished)
  - Write tests and make changes until we are ready to check in our work
  - Run all tests and ensure that all of them pass
  - Check for any updates that occurred while we were working
  - Run all tests again and ensure that all of them pass
  - Check in your changes

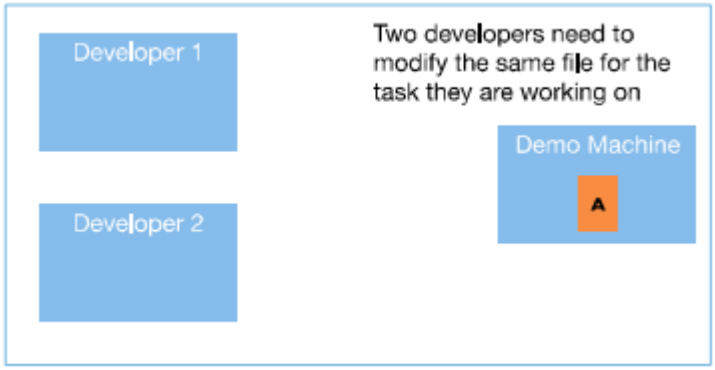
  At the end of this process, if the team is using an automated system, it will kick in, check out your changes, test them, and then update the official release (assuming everything passed)

- Automated Builds
  - Most programming languages are associated with automated build frameworks
  - C and Make
  - Java and Ant / Maven
  - All of these depend on a specification of some sort and a command you invoke to run the build
  - Most integrated development environments have automated builds as one of their core capabilities. We create a project, add/modify files, click "Run", and the project is built for you automatically

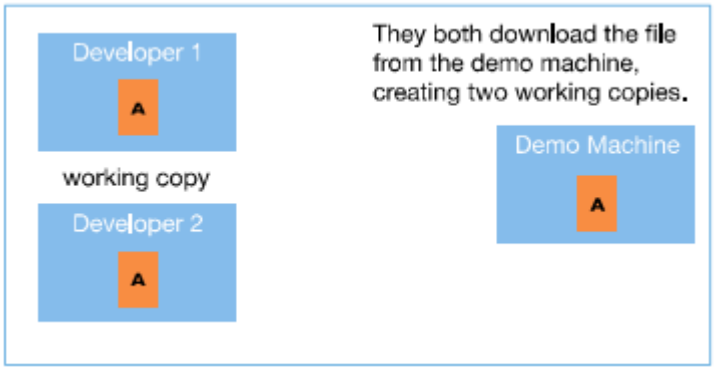
- Work Incrementally
  The key is to integrate new code into the system in small chunks. We want to be integrating new code or changes into our system and building our system several times per day. This ensures that we are finding incompatible changes quickly
  If we fail to integrate new/changed code multiple times per day, we will set ourselves up for pain in the near future. Perhaps right at the end of an iteration, where everything is supposed to come together, but instead fails, leading to a missed demo and extra work trying to integrate large sets of changes all at once.

Example

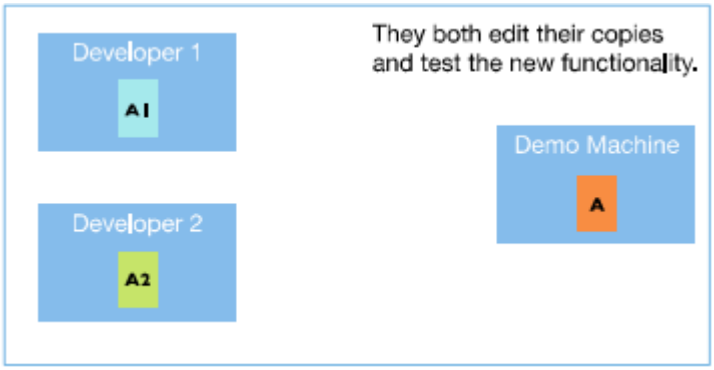
1. Without SCM- no tracking of changes in the code



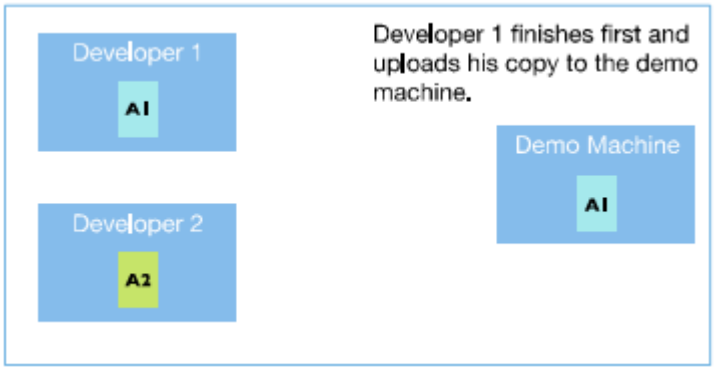
1



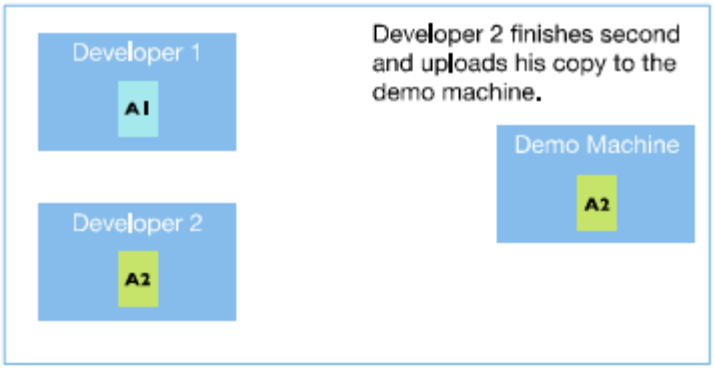
2



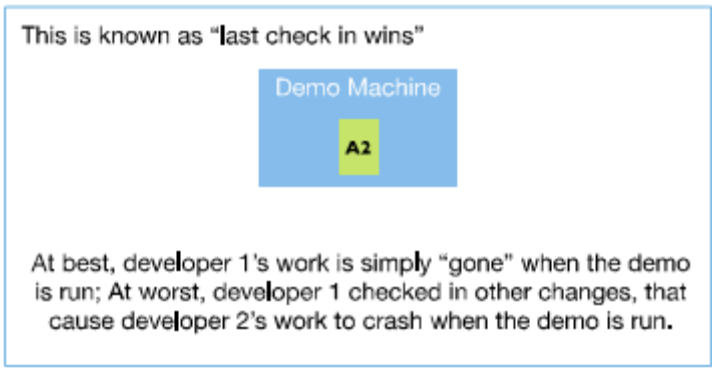
3



4

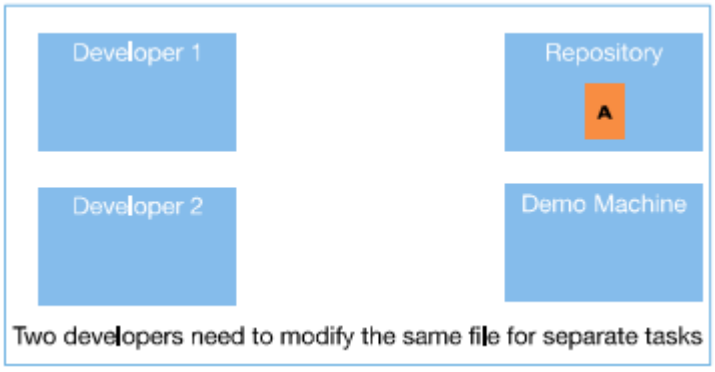


5

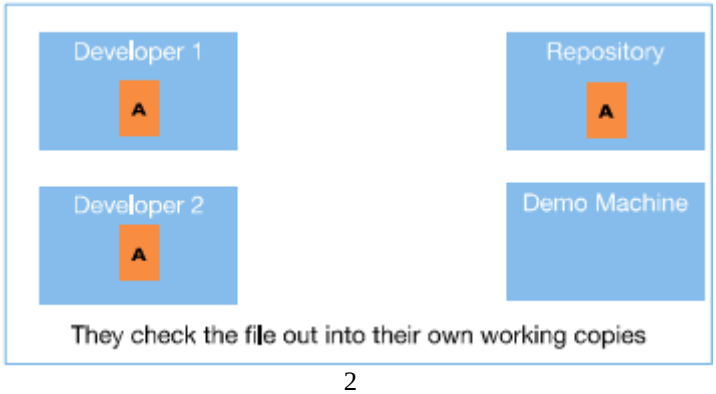


6

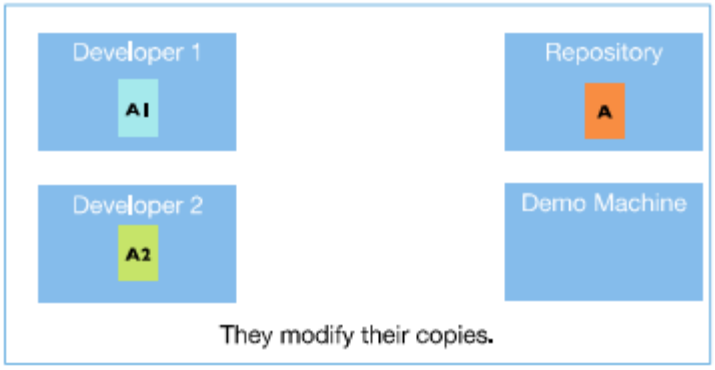
2. With SCM- tracking of changes in the code(to avoid the pitfall of above scenario 1)



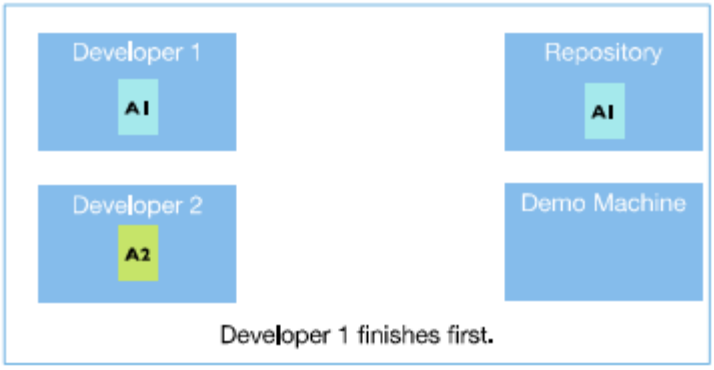
1



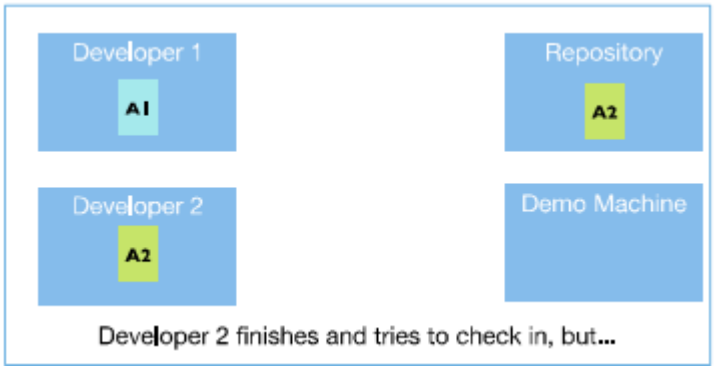
2


Developer 1
A1
Developer 2
A2
Repository
A
Demo Machine
They modify their copies.

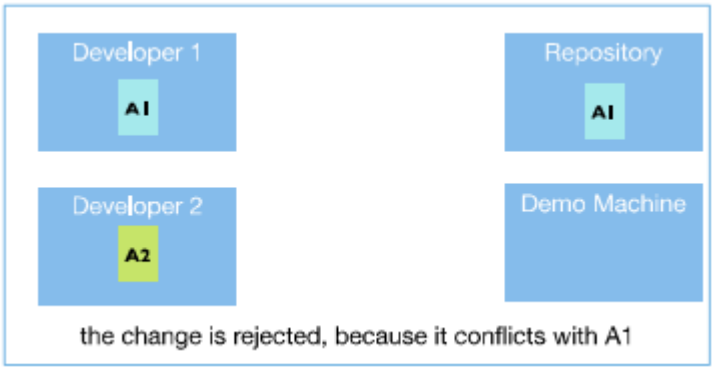
3


Developer 1
A1
Developer 2
A2
Repository
A1
Demo Machine
Developer 1 finishes first.

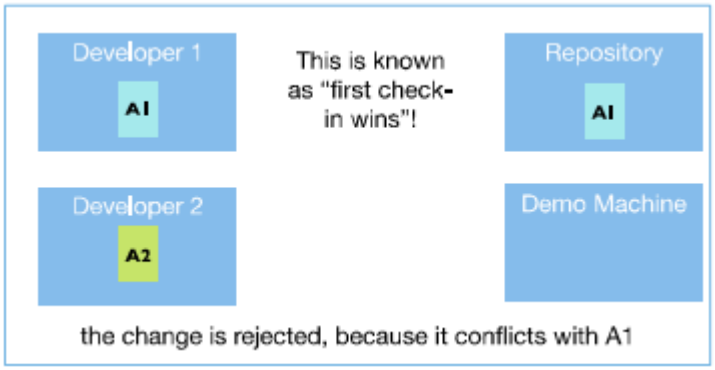
4


Developer 1
A1
Developer 2
A2
Repository
A2
Demo Machine
Developer 2 finishes and tries to check in, but...

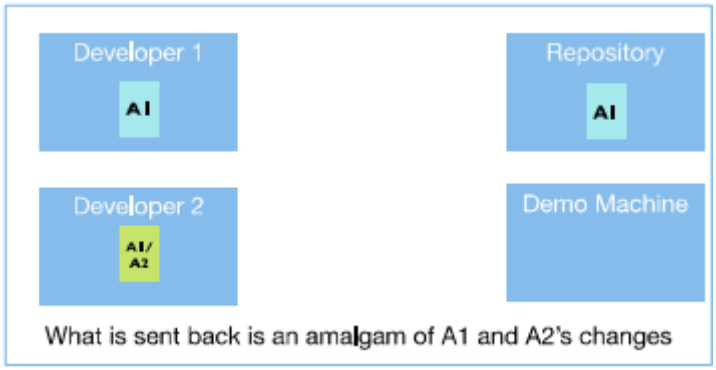
5


Developer 1
A1
Developer 2
A2
Repository
A1
Demo Machine
the change is rejected, because it conflicts with A1

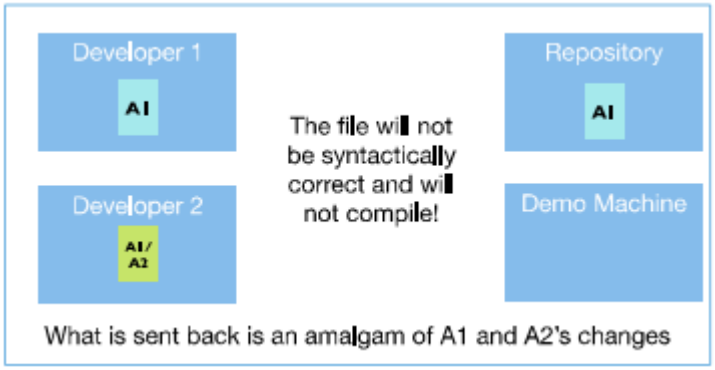
6


Developer 1
A1
Developer 2
A2
This is known as "first check-in wins"!
Repository
A1
Demo Machine
the change is rejected, because it conflicts with A1

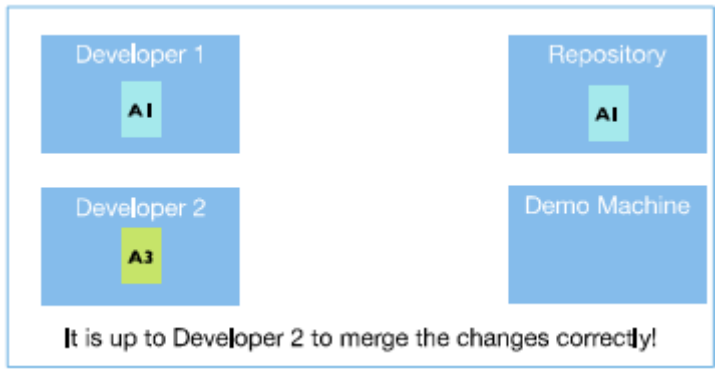
7


Developer 1
A1
Developer 2
A1/ A2
Repository
A1
Demo Machine
What is sent back is an amalgam of A1 and A2's changes

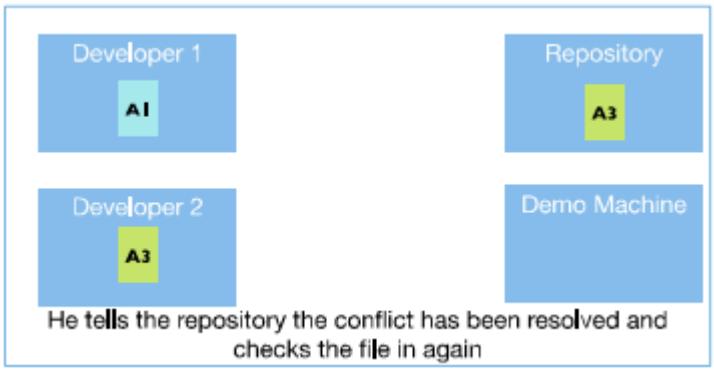
8


Developer 1
A1
Developer 2
A1/ A2
The file will not be syntactically correct and will not compile!
Repository
A1
Demo Machine
What is sent back is an amalgam of A1 and A2's changes

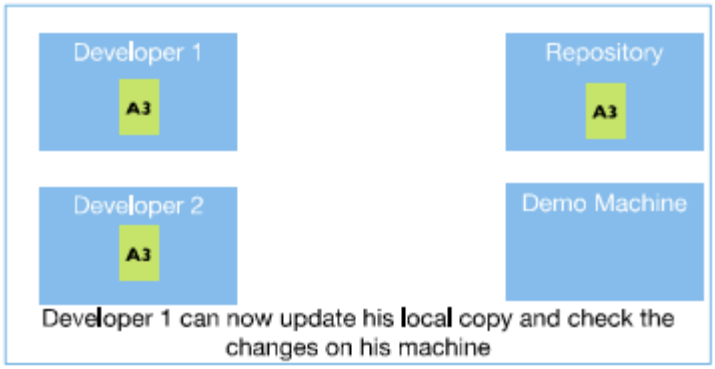
9


Developer 1
A1
Developer 2
A3
Repository
A1
Demo Machine
It is up to Developer 2 to merge the changes correctly!

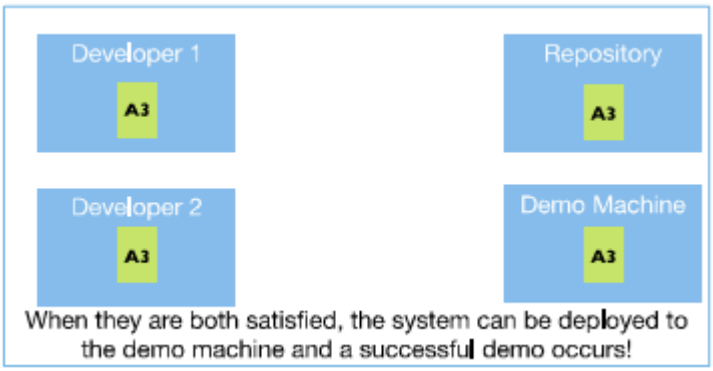
10



11



12



13

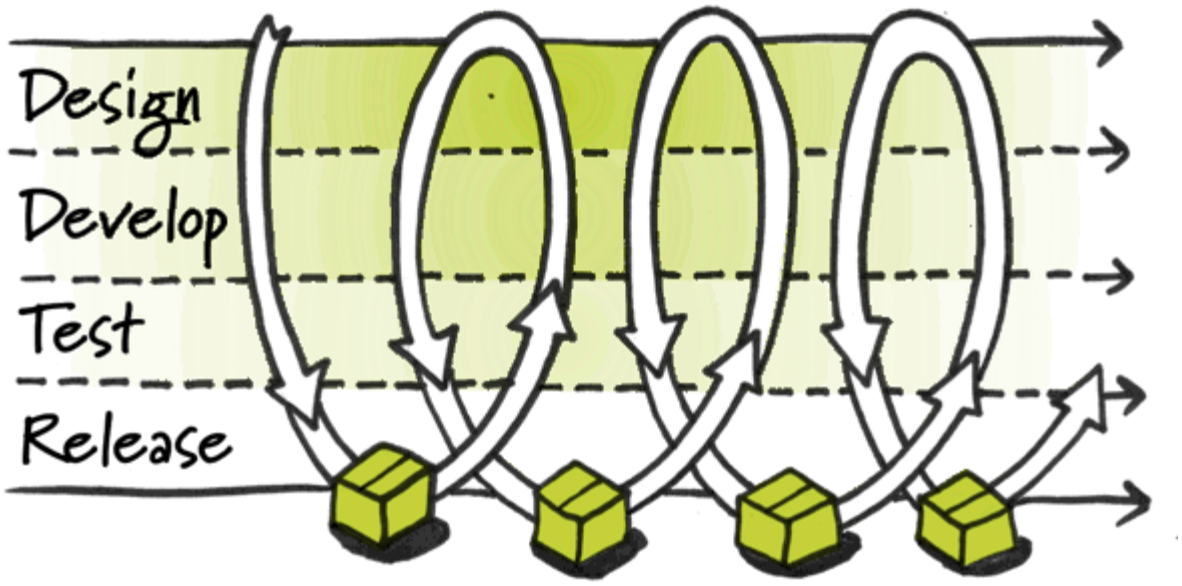


Fig: Showing a typical CI process

Continuous integration is a software engineering concept practiced mainly in extreme programming (XP). It advocates the merging of all developer working copies with a shared mainline in a continuous manner, for e.g. several times a day. Its main aim is to prevent integration problems, called as i*ntegration/merge hell*. Integration Hell is the situation where the time taken to integrate exceeds the time taken to make their original changes. In worst-case, developers may have to discard their updates and redo the whole work. CI was originally intended to be used in combination with automated unit tests written through the practices of test-driven development. Initially this was conceived of as running

all unit tests and verifying they all passed before committing to the mainline. This helps avoid one developer's work in progress breaking another developer's copy. If necessary, partially complete features can be disabled before committing using feature toggles. In addition to automated unit tests, projects exploiting CI normally use a build server to implement continuous processes of applying quality control (in general: small pieces of effort, applied frequently.)

Developers should be integrating and committing code into the code repository as frequently as possible (probably every few hours). It is not advisable to hold onto changes for more than one day. Continuous integration often avoids diverging or fragmented development efforts, where developers are not communicating with each other about what can be re-used, or what could be shared. Everyone needs to work with the latest version. Changes should not be made to obsolete code causing integration overheads. Developers should go on integrating code, normally, after successful (100%) unit tests or after the accomplishment of some smaller portion of the planned functionality. Only one pair integrates at any given moment and after only a few hours of coding to reduce the potential problem location to almost nothing. CI ensures to avoid/detect compatibility problems early. Integration is a "*pay me now or pay me more later*" kind of activity. Hence, if you integrate throughout the project in small amounts we do not need to integrate the system for weeks at the project's end while the deadline slips by. We have to always work in the context of the latest version of the system.

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. CI ensures that integrating is done early and often, and hence avoids the downsides of *integration hell*. CI aims to reduce rework and improve in terms of cost and schedule.

Continuous Integration doesn't get rid of bugs, but it does make them dramatically easier to find and remove." - Martin Fowler, Chief Scientist, ThoughtWorks

Since we integrate frequently, there is significantly less back-tracking to discover about the things which went wrong. This allows more time building core software features rather than spending time for back-tracking and debugging. Continuous Integration is cheap but not following the principal of CI may lead to a more costly situation. This is because if we are not following a continuous approach, then there will be longer periods between integrations. This makes it exponentially more difficult to discover, trace and solve problems. Such integration problems may force the project off-schedule, or even fail.

According to Martin Fowler, CI is comprised of 10 key practices

1. **Maintain a Single Source Repository**
   - *store all code required for a build in source control*
2. **Automate the Build**
   - *have an IDE-independent build server that can build and launch a system from a single command*
3. **Make Your Build Self-Testing**
   - *write a suite of automated tests that can check a large part of the code base for bugs (not necessarily TDD)*
4. **Everyone Commits to the Mainline Every Day**
   - *integration is about communication, by committing frequently developers can identify conflicts quickly*
5. **Every Commit Should Build the Mainline on an Integration Machine**
   - *maintain a continuous integration server that monitors the repository for changes to trigger builds and send notifications*
6. **Keep the Build Fast**
   - *keep builds slim, if necessary create a two-stage build pipeline for isolated compilation and testing*
7. **Test in a Clone of the Production Environment**
   - *have at least one environment that duplicates the production environment*
8. **Make it Easy for Anyone to Get the Latest Executable**
   - *anyone involved with a software project should be able to get the latest binary*
9. **Everyone can see what's happening**
   - *extremely important to convey state of the mainline build; can be done with website, desktop alerts, lava lamps*
10. **Automate Deployment**
    - *not necessarily to Production, have scripts to deploy applications quickly. also consider tools for rollback approach.*

Benefits of CI:

- Easy and quicker integration i.e. no long and tense integrations
- Increase visibility which enables greater communication
- Catch issues early and solve them early
- Spend less time debugging and more time adding features
- Proceed in the confidence by building on a solid foundation
- Avoids Confusions: Stop waiting to find out if the code works
- Reduce integration problems allowing the software delivery within schedule

CI is just more than a mere process. It is backed by quite a few essential principles and practices.

The Practices

- Maintain a single source repository
- Automate the build
- Make your build self-testing
- Every commit should build on an integration machine
- Keep the build fast
- Test in a clone of the production environment
- Make it easy for anyone to get the latest executable
- Everyone can see what's happening

- Automate deployment

How to do it

- Developers check out code into their private workspaces.
- When done, the commit changes to the repository.
- The CI server monitors the repository and checks out changes when they occur.
- The CI server builds the system and runs unit and integration tests.
- The CI server releases deployable artifacts for testing.
- The CI server assigns a build label to the version of the code it just built.
- The CI server informs the team of the successful build.
- If the build or tests fail, the CI server alerts the team.
- The team fixes the issue at the earliest opportunity.
- Continue to continually integrate and test throughout the project.

Team Responsibilities

- Check in frequently
- Don't check in broken code
- Don't check in untested code
- Don't check in when the build is broken
- Don't go home after checking in until the system builds
- Many teams develop rituals around these policies, meaning the teams effectively manage themselves, removing the need to enforce policies from on high.

Continuous Deployment

- Continuous Deployment is closely related to Continuous Integration and refers to the release into production of software that passes the automated tests.
- Essentially, "*it is the practice of releasing every good build to users*," explains Jez Humble, author of Continuous Delivery.
- By adopting both Continuous Integration and Continuous Deployment, we not only reduce risks and catch bugs quickly, but also move rapidly to working software.
- With low-risk releases, we can quickly adapt to business requirements and user needs. This allows for greater collaboration between outputs and delivery, fuelling real change in our organization, and turning our release process into a business advantage.

References:

http://www.extremeprogramming.org/rules/integrateoften.html
http://en.wikipedia.org/wiki/Continuous_integration
http://www.thoughtworks.com/continuous-integration
http://www.martinfowler.com/articles/continuousIntegration.html
http://www.cs.northwestern.edu/academics/courses/394/slides/fall12/Bose%20CI%20Northwestern%202012-11-05.pdf
http://www.cs.colorado.edu/~kena/classes/5828/s12/lectures/27-continuousintegration.pdf