Continuous Integration Training

Christophe Demarey
Rev 0.1
March 2013

I. Requirements

Programming experience and knowledge of developer common tools (version control, build, etc.)

II. Objectives

The goal of this training is to give to participants key points to be able to build a working software in an automated way, set up a build pipeline and quickly detect problems. All these abilities will free developers from boring and repetitive tasks and they will have more time to focus on added value tasks. It will also points out and describes the development process.

Continuous Integration: Principles and practices

- Identify key concepts of continuous integration
- Reduce risks using continuous integration

Setting up a Continuous Integration system

- Build software at every change
- Test continuously
- Inspect the code continuously
- Deploy continuously
- Get a continuous feedback

III. Introduction

« It's hard enough for software developers to write code that works on their machine. But even when that's done, there's a long journey from there to software that's producing value - since software only produces value when it's in production. »
Martin Fowler

We will see how continuous integration helps to:

- produce deployable software at every step in your development lifecycle,
- reduce the time between a defect introduction and its discovery, thereby lowering the cost to fix and
- increase the code quality of your software by building software often rather than waiting to the latter stages of development.

Exercise

In your opinion, what is Continuous Integration?

IV. Continuous Integration: Principles and practices

A. Background

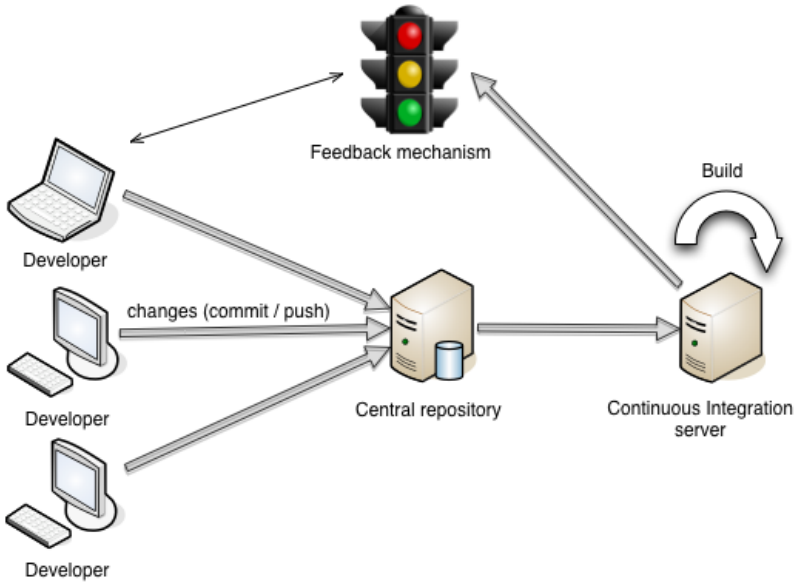
Increasing user requirements
Software development nowadays is more like handicraft (manual work).

B. Identify key concepts of continuous integration

a) Big picture

Continuous integration is, first and foremost, **a process** backed by a set of tools.

It is best to think of continuous integration as a mindset that allows you to reduce risk by frequently integrating incremental software development changes. Basically, it represents the realization and refinement of a common software development best practice: the daily build and smoke test (preliminary testing to reveal simple failures, severe enough to reject a prospective software release).



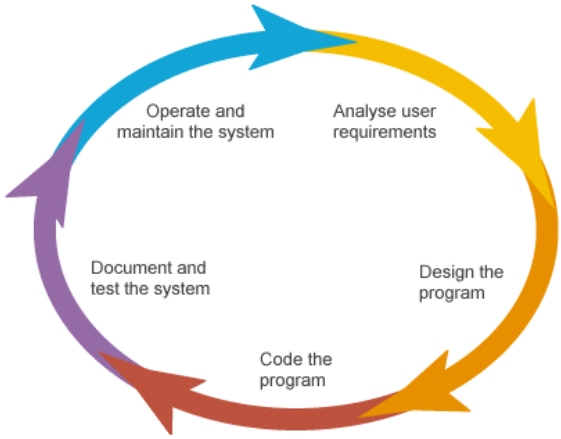
1. A developer commits changes to the version control system (central repository).
2. The repository, via a post-commit hook, tells the CI server a change has occurred.
3. The CI server retrieves the latest sources from the central repository and then builds the software (build script is shipped with sources). This step is called integration.
4. The CI server aggregate build results and generates a feedback. This feedback is often materialized by emails but it can also be instant messaging, tweets, a visual feedback on a screen or a red light in the dev room, etc.

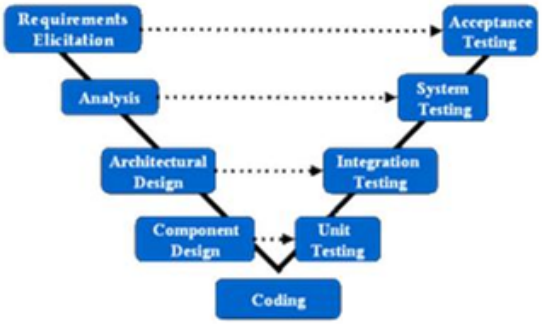
Continuous integration is not

- a tool,
- nightly build.

b) Incremental software development …

The concept of incremental software development is closely coupled to the concept of iterative software development.





Iterative and incremental software development is the basis of agile software development practices (XP, Scrum, Kanban).

c) What is integration? Why is it difficult?

Definition from Wikipedia:
"integration: the engineering practices and procedures for assembling large and complicated systems from less-complicated units, especially subsystems"

We can add that integration is not only assembling pieces but getting a new **working** system. So, you need to check the behaviour of the resulting system!

Why is it difficult?

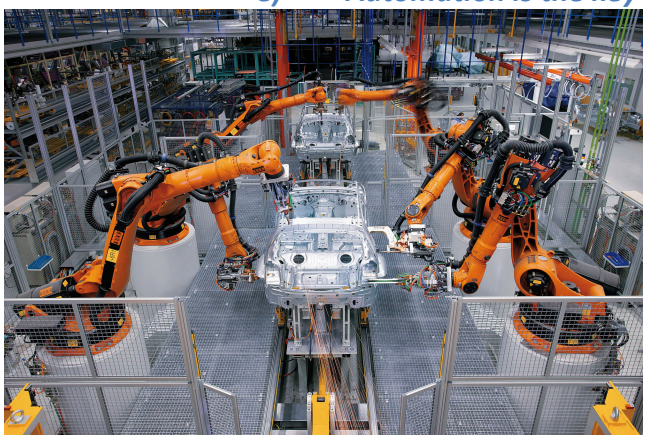
- Blind mode: Given a specification, you develop a component in blind mode. You only know if the specification is right and enough detailed once you try to integrate. If you defer the integration phase (acceptance testing) at the end of the project, it may leads to serious problems.
- Need to agree an a common API.
- The world change in the same time you are developing.

"If it's painful, don't put it off, do it more often!"

d) What is a build?

Build is not just a compile. A build is the process that takes source files as input, performs a set of actions (compilation, testing, inspection, deployment, …) to get a cohesive unit (the software) that works.

e) Automation is the key



=> define and explicit the process
=> no human error

f) Small increments

Iterative and incremental software development implies to explode your requirements in a lot of small and coherent parts that will be part of iterations / increments.
More details: http://essentials.xebia.com/small-increments

Software decomposition

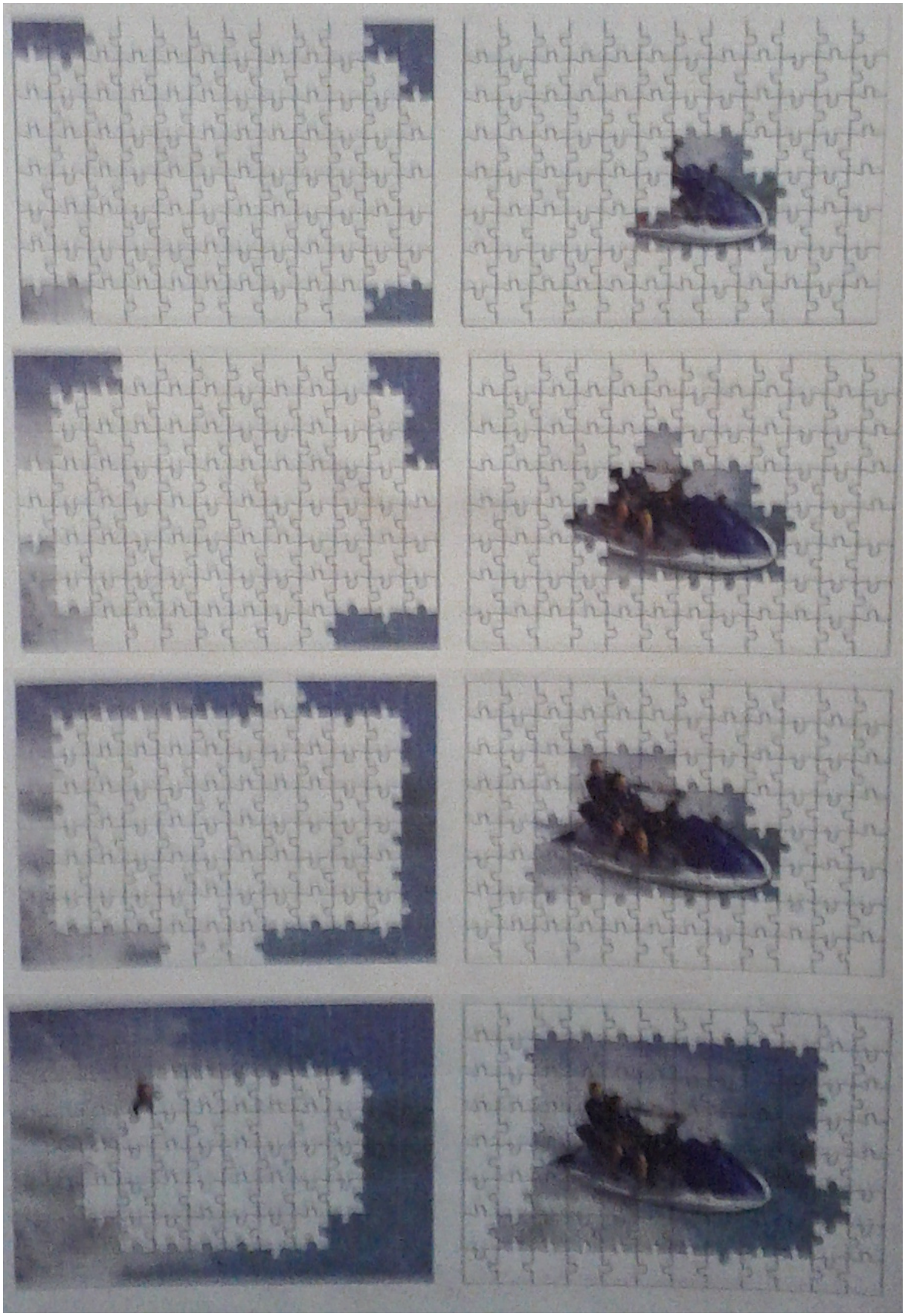
Valuable elements first

Development of software pieces must be driven by the value.
Example:

1. a box with wheels
2. with a motor
3. with seats

and not

1. air conditioning
2. mp3 player
3. GPS



Incremental development: classic and value driven

Developing in small increments is necessary but useless if changes are not pushed to the central source code repository! You need to commit code very frequently: "Frequently is more often than you think".

If we try to resume: we need to **commit very often, small changes and a change/commit should only contain a feature at a time** (only one reason to include or revert this change).
More details: http://essentials.xebia.com/one-change-at-a-time

g) CI practices

- Commit often to the central source code repository
- Commit one change at a time (small increments)
- Avoid getting broken code
- Run private builds
- Don't commit broken code
- Fix broken builds immediately
- Write automated developer tests
- All tests and inspections must pass

Exercise

Questions to ask to yourself

- Are you using a Version Control System (VCS, i.e. svn, git, etc.)?
- Is your VCS shared by all developers?
- Is your project's build process fully automated? Is it repeatable?
- Are you writing tests?
- Are you running automated tests?
- Is tests execution part of your build process?
- How do you check coding / design rules?
- Do you have an automated feedback?
- Are you using a dedicated integration machine to build software?

C. Reduce risks using continuous integration

Replace « big and long » integration phases (especially at the end of the project) with « small and frequent » ones (think of continuous compilation in eclipse).

Pros : time saving, eliminate human errors
Cons : initial set up time required, well-developped test suite, hardware costs

By effectively practicing continuous integration, you find out what goes wrong at every step rather than late in the development cycle. CI helps you identify and manage risks when they occur, making it easier to evaluate the health of your project and then take adapted decisions. Problems are often caused by non-managed risks.

We will focus on the key risks you can reduce by using CI.

Exercise

In your opinion, what are the risks of a development project?

1. Lack of deployable software

The deadline is coming and you still don't have a software you can build or deploy.
Sample use cases:

- "It works on my machine": you forgot to commit some files or the test environment is not exactly the same as the one on the developer machine.
- "I cannot test on the database": database scripts are not available for tests
- "Manual deployment": even id the process is simple, it takes time and you can miss some steps.

Solution

- Avoid tight coupling between your IDE and your build process.
- Use a separate machine for integrating / testing your software.
- Ensure that everything you need to build the software is in the source code repository (including database artifacts such as database creation scripts).
- Use a Ci server (ex: Jenkins) with an automated build
- Run the build when a change occurs in the source code repository.
- Automate the deployment process by adding it to your build (eliminate mistakes and avoid to waiting the deployment guy)

2. Late discovery of defects

Having tests is good but not enough. Manual testing offers no warranty that tests are run at each change.
Sample use case "regression testing": a developer often run tests related to the source code he is working on. But, it's quite rare he will run the whole test suite of the software because he find that useless and it takes too much time to run. The result is the introduction of side effect bugs, difficult to catch.

Solution

Automate tests run at each change of your software and check the test coverage!

3. Lack of project visibility

Here the topic is about **getting the right information at the right time** => communication basics. The information needs to be shared among all developers, available at any time in a known place and up to date.
The best way to achieve this task is to use a Continuous Integration server (the known place):

- to publish latest build artifacts,
- to publish and notify about latest tests build reports,
- to generate an up to date documentation (ex: you can use doxygen to generate the whole API documentation, generate design diagrams, etc.). All the documentation should not be generated but only the part closed to the source code, to ensure up to date.

4. Low quality software

You can have potential defects when your software is not well designed, is not following project standards, or is complex to maintain. This is also known as **code or design smells**.
"Code smell is any symptom in the source code of a program that possibly indicates a deeper problem.", http://en.wikipedia.org/wiki/Code_smell.
Overly complex code, code that does not follow the architecture and duplicated code all usually leads to defects in the software.
To ensure coding standards adherence, use a short document (one or 2 pages) and use an automated inspection tool as a part of the build. You can also use an automated inspection tool to check design rules. You can also analyse software dependencies.
The wonderful copy/paste mechanism leads to a lot of duplicated code in the software, making it more difficult to understand and, moreover, to maintenance problems. You will always forgot to update one duplicate and you will have defects. To manage duplicate code, use automated source code analysis tools and refactor to reduce duplicate code.
To resume, use software static analysis tool as part of your build combined with the integration server to share reports.
Tools example: checkstyle, jdepend, PMD

Exercise

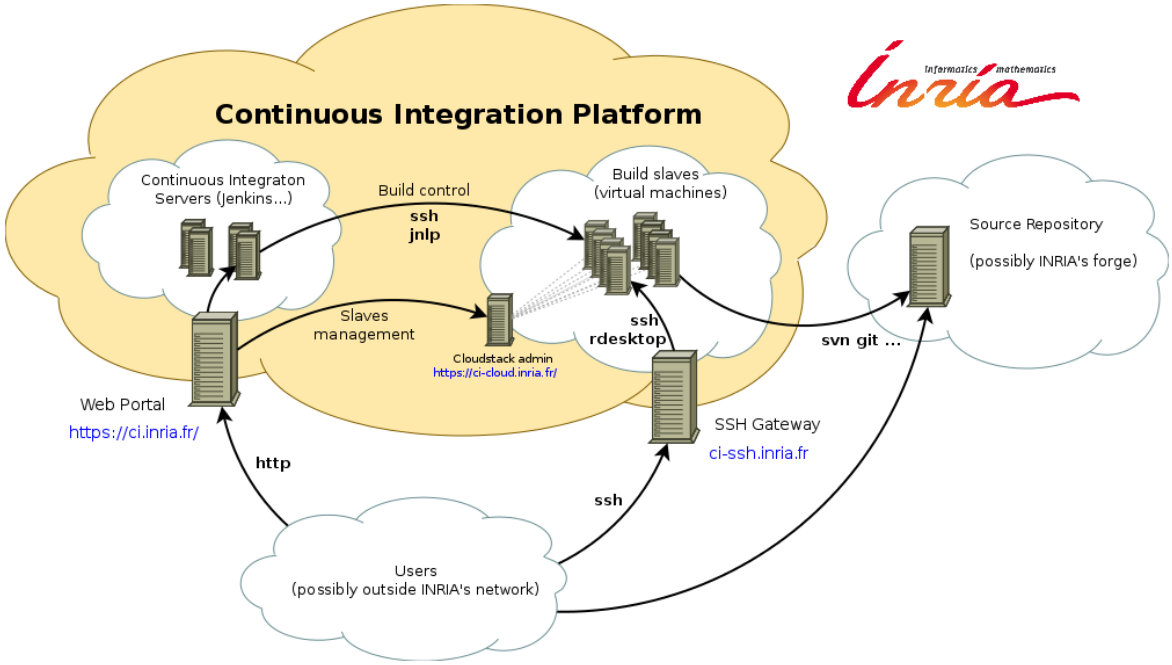
Rapid feedback

V. Setting up a Continuous Integration system

This session is mainly about practicing Continuous Integration.
Requirements: a computer with internet access.
You will find practice instructions on the dedicated document.

Jenkins dashboard :

Inria Continuous Integration platform architecture:



A. Build software at every change

Automate builds: create build scripts that are decoupled from IDEs. They will be executed by a CI server at every repository change.

Perform **single command builds**: you should be able to type one command to execute a build from your build script.

Separate build scripts from your IDE: you should be able to run your automated build without needing an IDE.

Centralize software assets: to decrease the number of broken dependencies, centralize all software assets

Define a **project layout**: create a consistent, logical directory structure, which makes it easy to build the software. For example, you can use the maven layout for a Java project.

Fail build fast: the faster the feedback occurs, the faster the problem can be fixed. Execute build activities in the order of what is most likely to fail first.

Build for any environment: run the same automated build for any environment. Don't duplicate build scripts. Use configuration files.

Use a **dedicated integration build machine**: ensure that this machine is free of old build artifacts.

Use a CI server to automatically poll for version control changes and run an integration build on a separate machine.

Run fast builds: Try to get your integration builds down to 10 minutes by increasing computer resources, offloading slower tests, offloading or reducing inspections, and running staged builds.

Staged builds: Run lightweight "commit" builds that perform compile, unit test exection, and deployment followed by heavyweight "secondary" builds that include other (slower) tests and inspections. You can also use distributed builds.

B. Test continuously

"Our acceptance tests validate that we built the right thing, while our unit and functional tests verify that we built the thing right."

1. Unit test

Unit tests verify the behaviour of small elements in the software system, which are more often a single class.

```
public class TestAdder {
    public void testSum() {
        Adder adder = new AdderImpl();
        // can it add positive numbers?
        assert(adder.add(1, 1) == 2);
        assert(adder.add(1, 2) == 3);
        assert(adder.add(2, 2) == 4);
        // is zero neutral?
        assert(adder.add(0, 0) == 0);
        // can it add negative numbers?
        assert(adder.add(-1, -2) == -3);
        // can it add a positive and a negative?
        assert(adder.add(-1, 1) == 0);
        // how about larger numbers?
        assert(adder.add(1234, 988) == 2222);
    }
}
```

2. Component test

Component tests verify that components interact to produce the expected aggregate behaviour. They use more dependencies than unit tests. They may require a fully installed system. They are longer to run than unit tests.

3. System test

System tests exercise a complete software system (the System Under Test) and require a fully installed system. These tests verify that external services like web services. They are longer to run than components tests.
They are different than functional tests, which test a system much like a client would use the system.
Examples: regression test, smoke testing

4. Functional test

Functional tests are also known as acceptance tests. They test the functionalities of an application from the viewpoint of a client. This is a Quality assurance (QA) process and a type of black box testing.
Selenium: live demo with ci.inria.fr and firefox.

5. Test doubles

When you're doing testing, you're focusing on one element of the software at a time - hence the common term unit testing. The problem is that to make a single unit work, you often need other units.

In Automated unit testing, it may be necessary to use objects or procedures that look and behave like their release-intended counterparts, but are actually simplified versions that reduce the complexity and facilitate testing. A test double is a generic (meta) term used for these objects or procedures. There are different kinds of test doubles:

- **Dummy** objects are passed around but never actually used. Usually they are just used to fill parameter lists. They have no influence on tests.
- **Fake** objects simply implement the same interface as the object that they represent and return pre-arranged responses. They usually take some shortcut, which makes them not suitable for production (an in memory database is a good example).
- **Stubs** provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test. Stubs may also record information about calls. You often use stubs inside unit tests when you're testing that a class or method derives the expected output for a known input. It allows testing the state of an object.
- **Mocks** are objects pre-programmed with expectations, which form a specification of the calls they are expected to receive. They are useful to test side effects, protocols and interactions between objects. They are used a lot in TDD.

More details on http://martinfowler.com/articles/mocksArentStubs.html

Examples:

a) Fake

A Data Access Object exemple :

```
class MemberDAO {
  private Connection connection;

  public Member find(String id) {
    return connection.createQuery("..").findOne();
  }
}
```

and its fake:

```
class FakeMemberDAO {
 private Map<Long, Member> members;

 public Member find(String id) {
  return this.members.get(id);
 }
}
```

b) Stub

The interface of the component to test :

```
public interface MailService {
  public void send (Message msg);
}
```

A stub exemple implementing the service interface :

```
public class MailServiceStub implements MailService {
 private List<Message> messages = new ArrayList<Message>();

 public void send(Message msg) {
   messages.add(msg);
 }

 public int numberSent() {
   return messages.size();
 }
}
```

A test using the stub :

```
public void testMemberMailSentWhenSubscribed() {
 // Création d'un membre
 Member member = new Member("login", "password");

 // On insère le stub à la place du mailer par défaut
 MyApp.setMailer(new MailServiceStub());

 // On enregistre le membre
 MyApp.register(member);

 // Un mail doit être envoyé
 assertEquals(1, mailer.numberSent());
}
```

c) Mock

```
public class OrderInteractionTester extends MockObjectTestCase {
  private static String TALISKER = "Talisker";

  public void testFillingRemovesInventoryIfInStock() {
    //setup - data
    Order order = new Order(TALISKER, 50);
    Mock warehouseMock = new Mock(Warehouse.class);

    //setup - expectations
    warehouseMock.expects(once()).method("hasInventory")
      .with(eq(TALISKER),eq(50))
      .will(returnValue(true));
    warehouseMock.expects(once()).method("remove")
      .with(eq(TALISKER), eq(50))
      .after("hasInventory");

    //exercise
    order.fill((Warehouse) warehouseMock.proxy());

    //verify
    warehouseMock.verify();
    assertTrue(order.isFilled());
  }

  public void testFillingDoesNotRemoveIfNotEnoughInStock() {
    Order order = new Order(TALISKER, 51);
    Mock warehouse = mock(Warehouse.class);

    warehouse.expects(once()).method("hasInventory")
      .withAnyArguments()
      .will(returnValue(false));

    order.fill((Warehouse) warehouse.proxy());

    assertFalse(order.isFilled());
  }
```

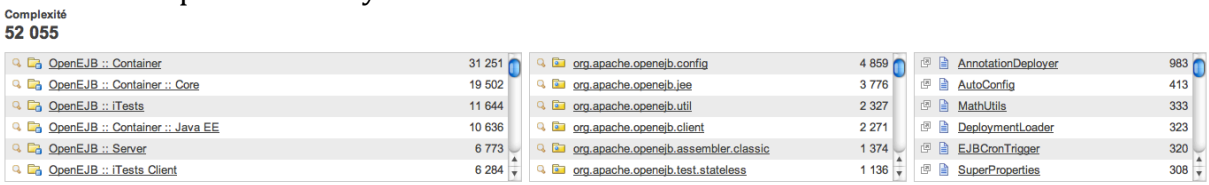
6. CI and tests

- Automate tests
- Categorize tests to be able to run slower tests at different intervals than faster tests.
- Write tests for defects : regression tests to ensure that the defect will not surface again. Use a bug tracker.
- Run faster tests first
- Make component tests repeatable : ensure that the data is in a “known state” (ex;: use a database test framework)
- Limit the number of asserts in a test case to spend less time tracking down the cause of a test failure.

C. Inspect the code continuously

1. Reduce code complexity

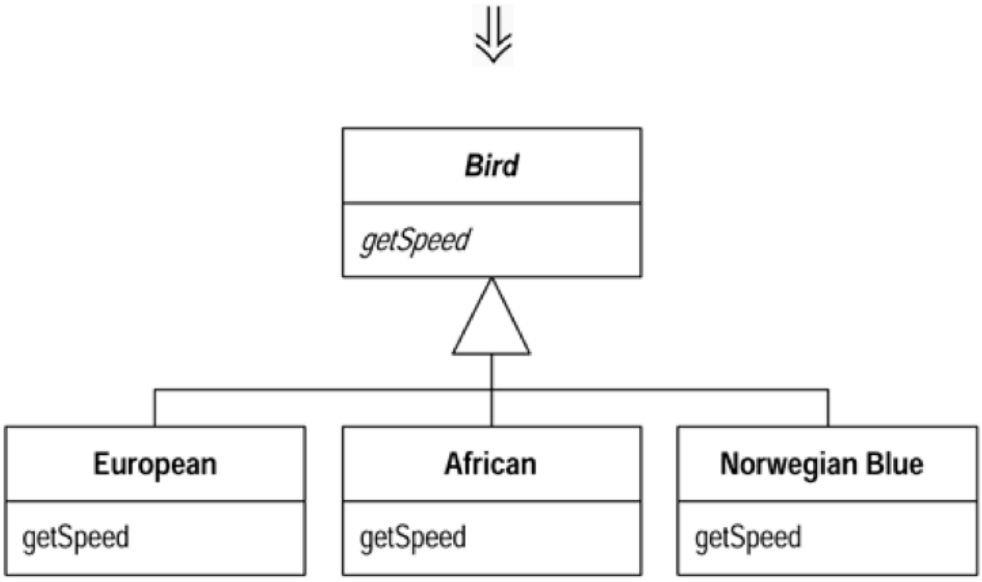
Cyclomatic complexity is a software metric to measure the number of linearly independent paths through a program's source code.
Reduce cyclomatic complexity in your code base by leveraging automated inspector tools such as Sonar or JavaNCSS to identify areas of your code with higher complexity. Run these inspectors from your automated build.



Some ways to reduce cyclomatic complexity :

- Split the code in small methods describing a consistent logic,
- Take advantage of an OO language: replace conditionnals with polymorphism,
- Perform input validations first and either return an error output or throw an exception if the validation fails.

```
double getSpeed() {
    switch (_type) {
        case EUROPEAN:
            return getBaseSpeed();
        case AFRICAN:
            return getBaseSpeed() - getLoadFactor() * _numberOfCoconuts;
        case NORWEGIAN_BLUE:
            return (_isNailed) ? 0 : getBaseSpeed(_voltage);
    }
    throw new RuntimeException ("Should be unreachable");
}
```



2. Perform design reviews

Incorporate Tools that can help determine packages/assemblies that are highly dependent on other packages and may lead to brittle architecture.
Some Tools : maven-dependency-* , ndepend

JavaNCSS Metric Results

[package] [object] [function] [explanation]

The following document contains the results of a JavaNCSS metric analysis.
JavaNCSS web site.

Packages

[package] [object] [function] [explanation]

Packages sorted by NCSS.

Package	Classes	Functions	NCSS	Javadocs	Javadoc lines	Single lines comment	Multi lines comment
org.springunit.framework	4	24	131	28	311	1	60
org.springunit.framework.junit4	4	11	56	2	16	0	84

Classes total	Functions total	NCSS total	Javadocs	Javadoc lines	Single lines comment	Multi lines comment
8	35	187	30	327	1	144

Objects

[package] [object] [function] [explanation]

TOP 30 classes containing the most NCSS.

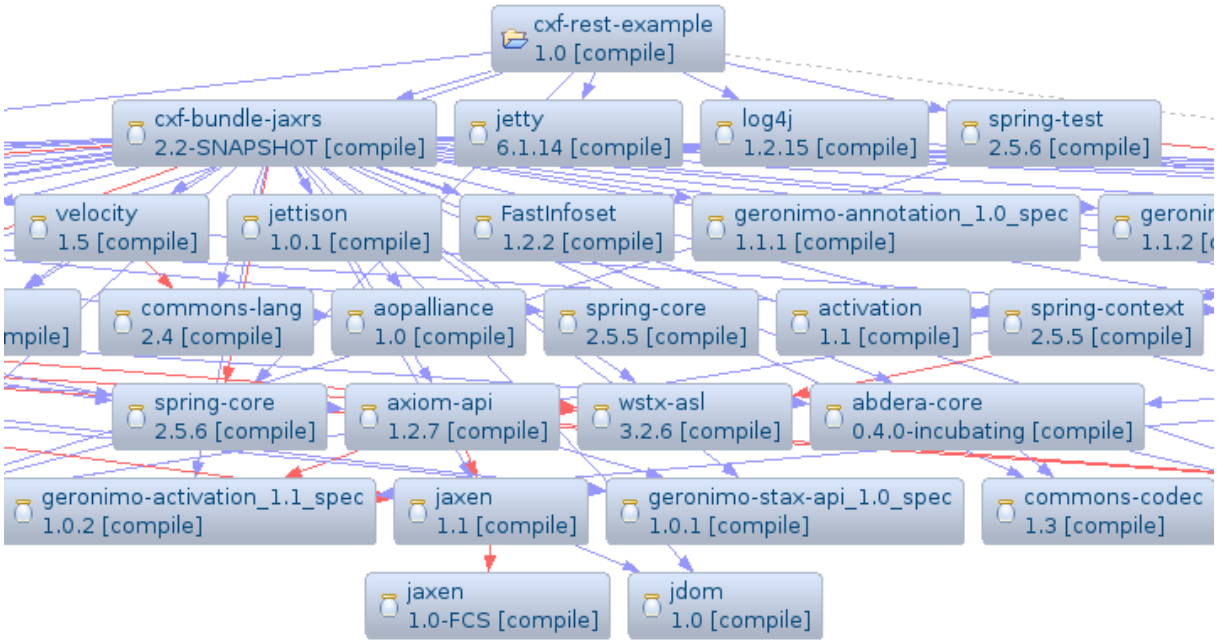
Object	NCSS	Functions	Classes	Javadocs
org.springunit.framework.HierarchicalSpringUnitContext	66	8	0	9
org.springunit.framework.SpringUnitTransactionalTest	20	8	0	9
org.springunit.framework.junit4.NameRunner	18	5	0	1
org.springunit.framework.SpringUnitContext	16	4	0	5
org.springunit.framework.SpringUnitTest	12	4	0	5
org.springunit.framework.junit4.NameListener	9	2	0	1
org.springunit.framework.junit4.SpringUnit4Test	6	2	0	0
org.springunit.framework.junit4.SpringUnit4TransactionalTest	6	2	0	0

TOP 30 classes containing the most functions.

Object	NCSS	Functions	Classes	Javadocs
org.springunit.framework.HierarchicalSpringUnitContext	66	8	0	9
org.springunit.framework.SpringUnitTransactionalTest	20	8	0	9
org.springunit.framework.junit4.NameRunner	18	5	0	1
org.springunit.framework.SpringUnitContext	16	4	0	5
org.springunit.framework.SpringUnitTest	12	4	0	5
org.springunit.framework.junit4.NameListener	9	2	0	1
org.springunit.framework.junit4.SpringUnit4Test	6	2	0	0
org.springunit.framework.junit4.SpringUnit4TransactionalTest	6	2	0	0

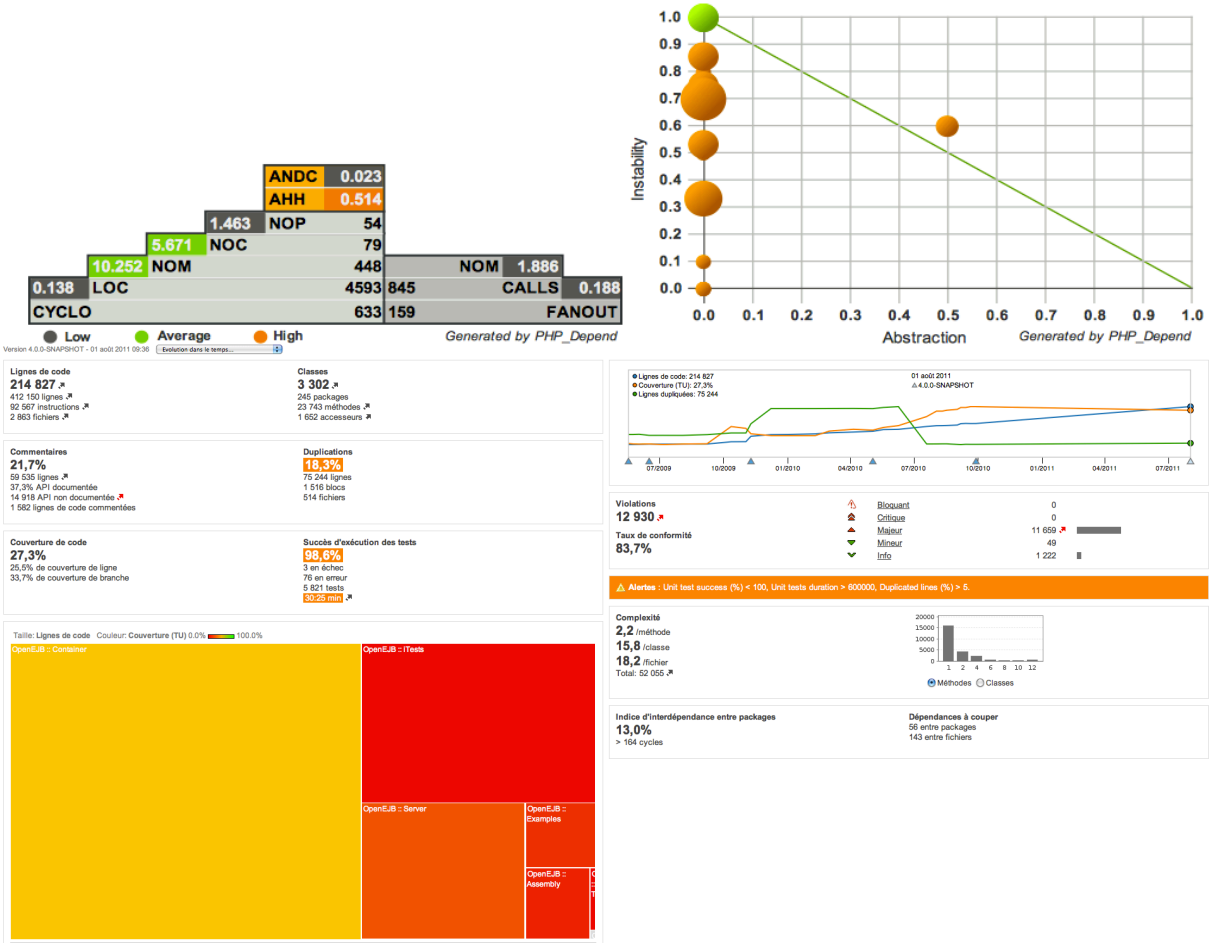
Averages.

NCSS average	Program NCSS	Classes average	Functions average	Javadocs average
19.13	187.00	0.00	4.38	3.75



3. Maintain organizational standards with code audit

Run tools such as PMD, PhpDepend, FxCop or clang analyzer that report on coding standards violations from your automated build.



To improve your code, use common refactoring patterns. See http://sourcemaking.com.

4. Reduce duplicate code

Reduce the amount of duplicate code in a code base is a very important issue for software maintenance.

Use tools (PMD-CPD) from your automated build to reduce duplicated code.

```
void printOwing(double amount) {
    printBanner();

    //print details
    System.out.println ("name:" + _name);
    System.out.println ("amount" + amount);
}
```

⇓

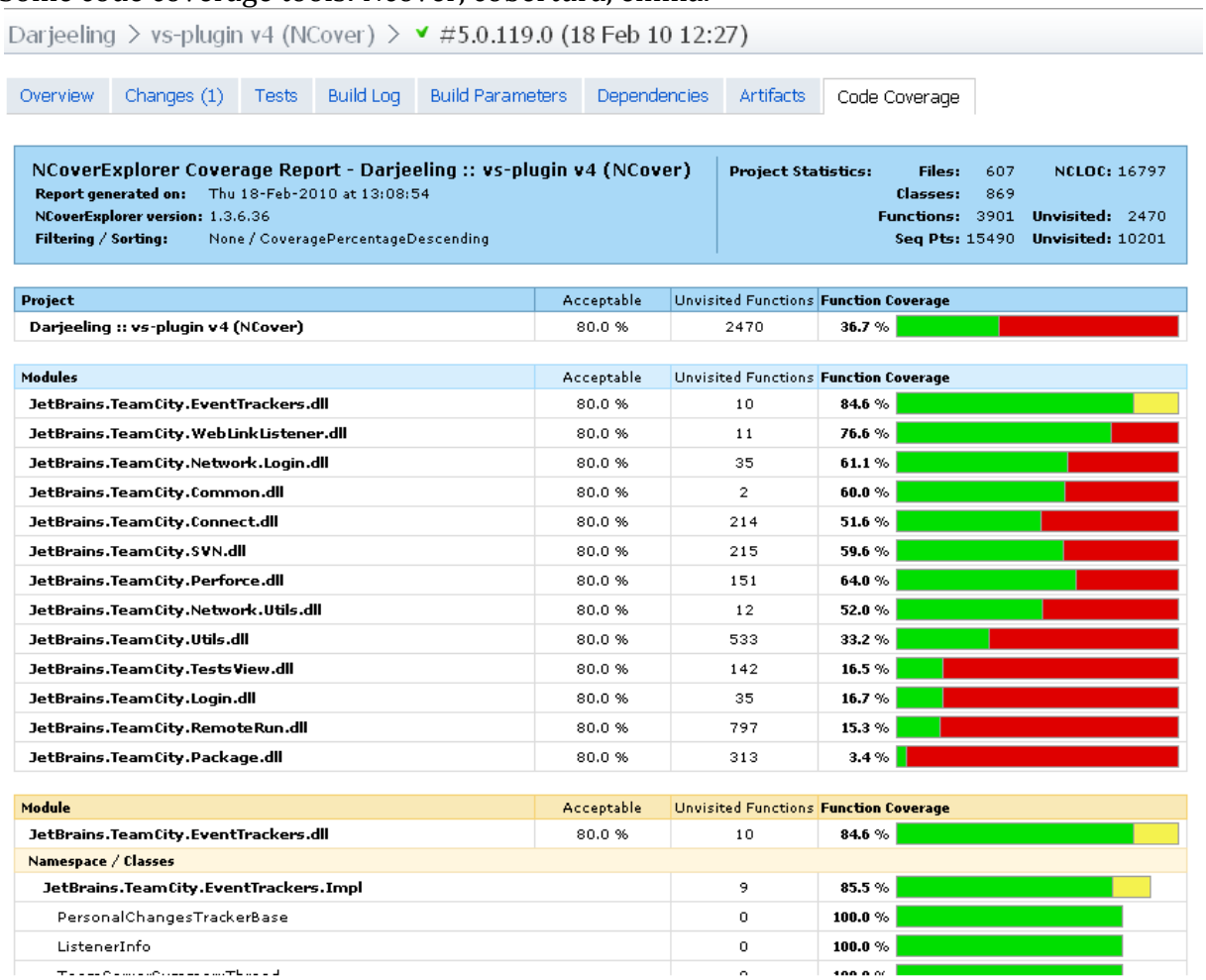
```
void printOwing(double amount) {
    printBanner();
    printDetails(amount);
}

void printDetails (double amount) {
    System.out.println ("name:" + _name);
    System.out.println ("amount" + amount);
}
```

5. Assess Code coverage

Determine areas that should use more tests by using code coverage analysis as part of your build.

Some code coverage tools: Ncover, cobertura, emma.



Darjeeling > vs-plugin v4 (NCover) > #5.0.119.0 (18 Feb 10 12:27)

Overview | Changes (1) | Tests | Build Log | Build Parameters | Dependencies | Artifacts | Code Coverage

NCoverExplorer Coverage Report - Darjeeling :: vs-plugin v4 (NCover)

Report generated on: Thu 18-Feb-2010 at 13:08:54
NCoverExplorer version: 1.3.6.36
Filtering / Sorting: None / CoveragePercentageDescending

Project Statistics: **Files:** 607 **NCLOC:** 16797 **Classes:** 869 **Functions:** 3901 **Unvisited:** 2470 **Seq Pts:** 15490 **Unvisited:** 10201

Project	Acceptable	Unvisited Functions	Function Coverage
Darjeeling :: vs-plugin v4 (NCover)	80.0 %	2470	36.7 %

Modules	Acceptable	Unvisited Functions	Function Coverage
JetBrains.TeamCity.EventTrackers.dll	80.0 %	10	84.6 %
JetBrains.TeamCity.WebLinkListener.dll	80.0 %	11	76.6 %
JetBrains.TeamCity.Network.Login.dll	80.0 %	35	61.1 %
JetBrains.TeamCity.Common.dll	80.0 %	2	60.0 %
JetBrains.TeamCity.Connect.dll	80.0 %	214	51.6 %
JetBrains.TeamCity.SVN.dll	80.0 %	215	59.6 %
JetBrains.TeamCity.Perforce.dll	80.0 %	151	64.0 %
JetBrains.TeamCity.Network.Utils.dll	80.0 %	12	52.0 %
JetBrains.TeamCity.Utils.dll	80.0 %	533	33.2 %
JetBrains.TeamCity.TestsView.dll	80.0 %	142	16.5 %
JetBrains.TeamCity.Login.dll	80.0 %	35	16.7 %
JetBrains.TeamCity.RemoteRun.dll	80.0 %	797	15.3 %
JetBrains.TeamCity.Package.dll	80.0 %	313	3.4 %

Module	Acceptable	Unvisited Functions	Function Coverage
JetBrains.TeamCity.EventTrackers.dll	80.0 %	10	84.6 %
Namespace / Classes			
JetBrains.TeamCity.EventTrackers.Impl		9	85.5 %
PersonalChangesTrackerBase		0	100.0 %
ListenerInfo		0	100.0 %

D. Get a continuous feedback

"As a general rule, the most successful man in life is the man who has the best information."

Benjamin Disraeli (1804-1881)

Without feedback, none of the other aspects of CI is useful.

You need rapid feedback to take immediate action and fix the problem before it propagates.

Beware of information overload : sending feedback to everyone on a project usually only causes everyone to ignore this information.

Old news is not really news at all.

The heart of continuous feedback is reducing the time between when a defect is introduced, discovered, and fixed.

They are various mechanisms to enable continuous feedback :

- Email, sound, visual devices, dashboards on wide screens, SMS, browser plugins, etc.

E. Deploy continuously

Without a successful deployment, a software does not really exists.
Deploy continuously enables to release working software any time, any place, with as little effort as possible.
As an example, Flickr, the photo sharing web site, releases software every 30mn in a good day.

1. Tag releases put into production to facilitate creation of release-bug-fixes branches
2. Produce a clean environment, free of assumptions. It is a matter of removing and reapplying software, scripts and configuration values to ensure that the environment is operating as expected. They are different levels but if you want to reduce risks, you should drop all including the Operating System.
3. Generate and label a build directly from the repository and install it on the target machine. It allows to bind the build to the source code.
4. Successfully run tests at all levels in a clone of the production environment
5. Create build feedback reports: tell which defects have been addressed, what are the new features, etc. You may add a file difference report.
6. Be able to rollback quickly if needed.

F. Further work

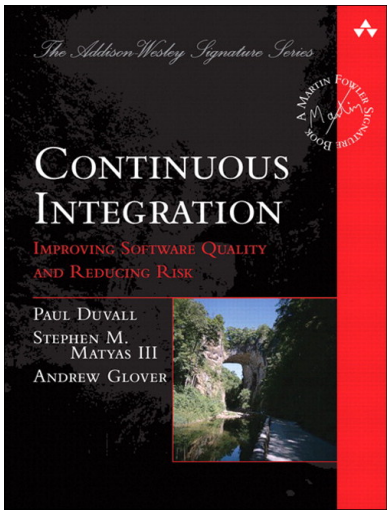
Distributed build
Staged build using

- https://wiki.jenkins-ci.org/display/JENKINS/Clone+Workspace+SCM+Plugin
- https://wiki.jenkins-ci.org/display/JENKINS/Splitting+a+big+job+into+smaller+jobs

VI. References

Wikipedia : http://en.wikipedia.org

Martin Fowler's article on Continuous Integration:
http://www.martinfowler.com/articles/continuousIntegration.html



http://sourcemaking.com/

VII. Appendix: Continuous Integration guideline

Top 10 rules

1. **Check-in regularly to mainline**

Continuous Integration is about integrating early and often. To take advantages of the CI methodology, you need to share your code with all developers in the mainline (core repository, trunk) quickly. It implies small tasks / increments.

2. **Have an automated test suite / use TDD**

Each piece of code should come with its automated tests. A good practice is to write tests before the code: Test-Driven Development.

3. **Always run tests locally before committing**

Yes, there is a server that runs test for you … but it does not prevent us to run a minimum set of tests in your development environment. The CI server is here to detect problems not seen in your development environment.

4. **Don't commit broken code to the mainline!**

A source code repository is not a disk backup. A (known) broken code should never be committed to the mainline.

5. **Don't check-in on a broken build**

If you commit on a broken build, you loose feedback on this integration. If there are several commits on a broken build, you may end with a lot of bugs introduced in the mainline and loose your time to track problems.

6. **Fix broken build immediately**

Fix broken builds should be the highest priority of the team. It prevents other developers to check-in and will end with a big integration task. The solution is to time-box the fix.

7. **Time-box fixing before reverting**

Define a reasonable time (for example 20 minutes) allowed to fix a broken build. Once this time elapsed, revert to the latest working version and take the time you need to fix it. Then commit again.

8. **Never go home on a broken build**

We don't want you to stay at work all the night! If you don't have time, revert, go home, and fix it the next day.

9. **Don't comment out failing tests**

We don't want broken builds but we need to be confident in our code robustness. We need all tests, more tests!

10. **Keep your build fast**

The key is to get a rapid feedback. So, you need to get your build fast (ideally <10 min). If you cannot speed the build up, use a staged build.