Software Craft

Discover and Practice

Version Control

Definition

Version control, also known as 'revision control' or 'source control', is the process of managing changes to documents, programs and other stores of information. In software development, it's important that you carry out version control process on your source code so that you can recall specific versions or changes at a later time.

A version control system (VCS) allows you to manage file history, so you can roll back to previous states of a file if something goes wrong (Figure 1), maintain change logs and compare versioning issues.

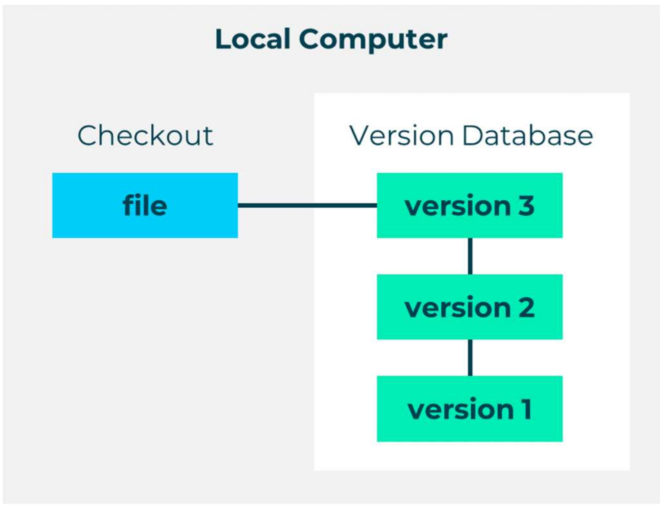


Figure 1

Benefits

Keeping on top of your version control can seem like a tedious task, but it'll be worth it in the long run. Using version control means that you can...

1. Keep track of code and changes, so you can avoid any confusion involved in mailing it around and integrating it.
2. Allow multiple people to work on a project at the same time by pushing changes to the central repository. A VCS will also merge changes together in files where there are conflicts.
3. Branch code to work on specific parts of it.

Version Control Systems

There are two main types of version control systems: centralised version control systems (CVCSs) and distributed version control systems (DVCSs).

A CVCS (Figure 2) allows multiple developers to collaborate on other systems such as CVS, Subversion and Perforce. It's made up of a single server that contains all of your version-controlled files, and clients can check out files from this source server.

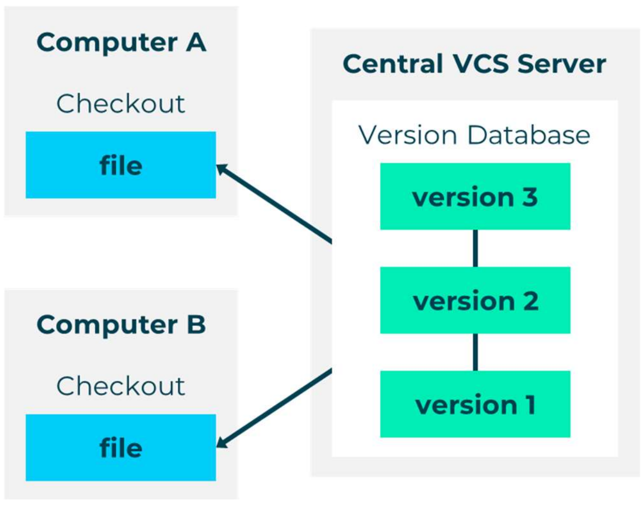


Figure 2

A DVCS (Figure 3) mirrors the central repository, so each time a user checks out that file, it's like creating a backup of the repository. They don't just check our the most local snapshot of a file. Git and Mercurial are examples of a DVCS.

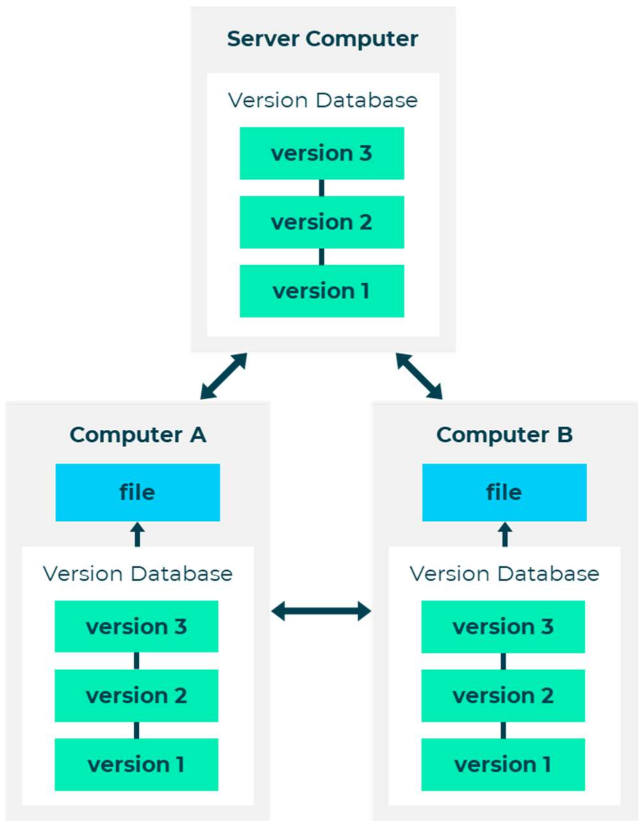


Figure 3

Git as a DVCS

Git is a powerful open-source version-control tool that you can use to track changes in your source code when you're developing software.

As a DVCS, its goals are:

- Speed
- Simplicity
- Strong support for non-linear development
- Full distribution
- Ability to handle large projects efficiently, e.g. Linux kernel

When using Git, you'll need to choose a hosting service to help you manage it. Some of the most popular Git hosting services used currently are GitHub, GitLab and BitBucket. There are five steps to changing something within a Git repository:

1. Create a repository on a hosting site, or on your own server.
2. Check out the repository to your own machine using `git remote add`.
3. Add some code.
4. Commit your changes to the local repository.
5. Push changes to the remote repository using `git push`.

Git Basics

Below, you can find a series of code snippets that will help you get started with using Git.

Initialising Repositories

To create a new subdirectory and a Git repository skeleton, use:

```
git init
```

Initialising Repositories with Existing Files

```
git add *.pp

git add README.md

git commit -m "Initial commit"
```

Cloning Repositories

Git can use a number of different protocols, including http and SSH:

```
git clone

git://github.com/resource
```

Recording Changes to a Repository

Each file in a Git directory can be **tracked** or **untracked** (Figure 4). Tracked filed are files that were in the last snapshot. They can be modified, unmodified or staged. Untracked files are everything else – this means that they aren't in your last snapshot or staging area.

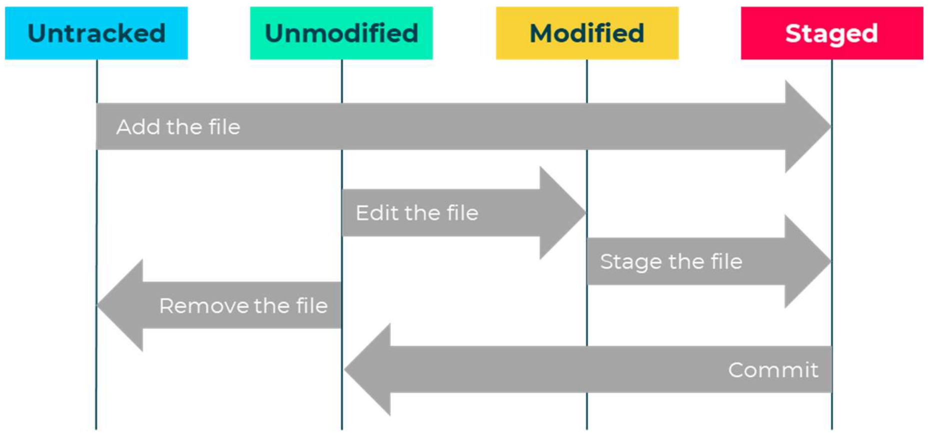


Figure 4

To determine which files are in which state, you should use the git status command. If you run this directly after a clone, you should see something like the following:

```
git status

On branch master

Nothing to commit, working directory clean
```

You can find more information on recording changes to a repository at the git-scm website.

Staging New or 'Modified' Files

To add a file, use:

```
 git add <filename>
```

To tell Git to ignore files or folders, name them:

```
.gitignore
```

Working with Remote Repositories

Git projects are often held on remote repositories, which hold versions of a project or dependencies on the web / network. To see configured remote repositories, run the following command:

```
git remote
```

If you've cloned a repository, you should see the origin. To add a repository, use:

```
git remote add
```

```
[shortname] [url]
```

'Shortname' becomes an alias for access to the repository.

Pushing to a Repository

To push your project to a repository, use:

You can also specify `-v`, which shows you the URL that Git has stored for the shortname to be expanded to.

In newer versions of Git, if you want to rename a reference, run:

```
git remote rename
```

For example:

```
git remote rename pb dave
```

```
git remote origin dave
```

It's worth mentioning that this changes your remote branch names, too. In the above example, what used to be referenced at pb / master is now at dave / master.

If you want to remove a reference for some reason - you've moved the server or are no longer using a particular mirror, or perhaps a contributor isn't contributing any more - you can use git remote rm:

```
git remote rm dave
```

```
git remote origin
```

Pulling from a Repository

To pull all the changes you've made to the repository, use:

```
git pull
```

It's good practise to do this before you push any changes, so that you get an up-to-date copy of the repository to push to and you can see any conflicts before you push them. You can also stash your changes before pulling the remote branch.

Creating a New Branch

Branching allows you to diverge from the main line of development without doing accidental damage to the main line. Git encourages a workflow that allows you to branch and merge, so Git branches are very lightweight compared to other VCSs. Thy build on core Git features, so when you commit you have a snapshot of current content and pointers to the current commits. Branches are based on your repository, or parents.

To create a branch, use:

```
git checkout -b newBranchName
```

To commit any changes to your code, use:

```
git commit -am "updated some file(s)"
```

To merge a branch back into the main line, use:

```
git checkout master

git merge newBranchName
```

Alternatively, you can use the following two commands:

```
git branch newBranchName

git checkout newBranchName
```

Alternatives to Git

Git is popular due to its open source nature, simplicity, and built-in tools. However, it's important to know that it isn't the **only** existing source control method. Popular alternatives include Subversion, CVS, Mercurial, Fossil, Veracity and SSH.

Subversion is very similar to Git in that you can add new files to the repository and pull files from it. However, Subversion is a CVCS.

The SHH File Transfer Protocol is a UNIX-based command interface and network protocol that allows you to access, transfer and manage files. It's a suite of three utilities (slogin, ssh and scp) that you can use to gain secure access to remote a remote computer. SSH ensures this security by encrypting and authenticating its commands with a digital certificate. SSH uses RSA public key cryptography for both connection and authentication.

Pair Programming

Definition

Pair programming is an agile technique that involves two developers working together as peers on the same project. One writes the code while the other critiques, provides guidance and conducts research, and both have co-responsibility for the outputs. The developers can switch roles when needed, but there's always one person doing each task. This is an effective strategy to help identify bugs and design problems, and maintain coding standards.

Benefits

Pair programming has a number of benefits. Working as a team means that you increase both your productivity, and your confidence in the outputs you're producing. You're also able to cross-learn, particularly if you switch roles with the other developer. This can reduce training costs and time within your organisation, as you are able to train each other. Two minds means two points of view working to solve problems, so you find more effective solutions more quickly, which helps you overcome blockers or obstacles more quickly. The continuous code reviews mean that you can give and receive feedback more quickly, so you can reduce and resolve bugs more quickly. Finally, the shared responsibility helps build trust across teams, and also means that you don't have to put the project on hold should a developer go on annual leave or fall ill.

Types of Pairing

Driver-Navigator

This is arguably the most established style of pairing. The driver focuses on tactical concerns related to the mechanics of the activity, such as typing, file handling, basic code implementation and syntax. The navigator focuses on the big picture, which means looking at broader concerns and checking the driver's work for mistakes.

Backseat Navigator

This style is similar to the Driver-Navigator, but the navigator is a little more tactical. The driver still types, but the navigator decides on the code structure, naming, and file management.

Tour Guide

In this style, the guide types and explains what they're doing. The tourist rarely intervenes. This can work with both an expert guide and novice tourist, and vice versa, with the tourist correcting when the guide makes a mistake or is unsure.

Ping-Pong

This style is used within test-driven development (TDD) and extreme programming (XP). Developer A writes a failing test and Developer B writes a code to pass. Then Developer B writes a failing test and Developer A writes the code to pass. This continues back and forth (like a game of ping pong!) until the user story is complete.

Cross-Functional

This style is used in embedded system development, where a developer works with a hardware engineer. It allows more time to work alone, and is ideal for developing a new system.

Distributed

This style works well for teams whose members are in different geographical locations. The developers work together via a collaborative real-time editor, shared desktop or remote pair programming plugin. It's more tiring than traditional pair programming as it often requires more hours or work. Tools for distributed pair programming include Mikogo, Trellis and Yuuguu.

Software Design Patterns

Definition

Software design patterns are reusable solutions to commonly occurring problems. They emerge over time as developers write code and establish best practises for certain situations. We usually use them to solve specific problems in code generation and interactions, rather than large-scale architectural problems.

Patterns can be really powerful if we apply them correctly in the right situations. The key is that they aren't prescriptive specifications for writing code – you have to know when it's appropriate to use them.

The Principle of Balance

Using software design patterns is a great way of grasping and implementing principles in your code, but you need to find the balance between making it too simple and needlessly complex. When your code is too simple, it's less flexible, and more fragile and repetitive. When it's too complex, it's harder to follow and implement.

Common Design Patterns

Common software design patterns include:

- Façade: a front-facing interface that hides a more complex code
- Proxy: provides a placeholder for another object
- Command: encapsulates a request as an object
- Observer: defines a one-to-many relationship between objects to notify all objects of any changes to one of them
- State: allows an object to change its behaviour when its internal state changes
- Strategy / Template: defines a family of algorithms, encapsulates each algorithm and makes the algorithms interchangeable within that family
- Factory: creates objects without exposing the instantiation logic to the client
- Singleton: a class self-instantiates the only instance that exists in the program

Factory and singleton are both included in the 'Gang of Four'. Let's take a look at what that is.

The 'Gang of Four'

In 1994, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides wrote a book called *Design Patterns: Elements of Reusable Object-Oriented Software*.

The publisher, Addison-Wesley, describes their work as follows:

> *'Capturing a wealth of experience about the design of object-oriented software, **four top-notch designers present a catalog of simple and succinct solutions to commonly occurring design problems**. Previously undocumented, these 23 patterns allow designers to create more **flexible, elegant, and ultimately reusable designs** without having to rediscover the design solutions themselves.'*

Note that we've added the bold emphasis for the purpose of this guide.

The book includes a comprehensive explanation and evaluation of 23 different design patterns, as well as details on how you can use them to design object-oriented software. The design patterns are categorised into three sections: creational, structural and behavioural.

Creational patterns provide ways to create instances of individual objects, or groups of related objects.

Structural patterns let developers establish the relationship between classes or objects through the use of inheritance and interfaces.

Behavioural patterns define how classes and objects are allowed to communicate with each other.

Testing

Unit Testing

Definition

Unit testing, also known as ‘component’ or ‘module testing, is a software testing method that tests individual sections of a source code. A ‘unit’ is the smallest testable part of a software. This could be a method, a database query, protocol or transaction, or a dynamic webpage, depending on the project you’re working on. We carry out unit tests to validate each unit of our software in isolation to make sure that they’re all functioning as they should.

Unit tests are an example of test-driven development (TDD), which means that they’re written before the code to be tested. Kent Beck, software engineer and creator of extreme programming, summarises TDD as allowing the developer to eliminate duplication by writing new code only if they first have an automated test.

The tests are made up of small pieces of code that the developers write, execute and maintain. The tests themselves are automated and ‘white box’ - this means that the test has complete knowledge of the internals of the component being tested. This contrasts with 'black box’ testing, in which only the interface of the component is being tested.

Benefits

Unit tests will tell you if your code really works as you develop it. This saves a lot of time, as you’ll detect any defects or bugs early on and can more easily localise the source of the bug.

Using continuous integration (CI) to run your tests, pull down the latest changes and push code to a remote depository also makes deployment much easier.

Long story short, unit testing gives you the confidence that your code is of high quality and will perform exactly as you want it to.

Core Concepts

'Test-first programming is the least controversial and most widely adopted part of Extreme Programming (XP). By now the majority of professional Java™ programmers have probably caught the testing bug'

– Elliotte Rusty Harold

So, let's take a look at some of the concepts that make unit testing such a popular method amongst software developers, and such a core practice of extreme programming (XP):

- You can adopt it within other methodologies
- TDD: the test is written before implementation, and tests drive the design of the application programming interface (API)
- It's automated and self-validating
- It's easy to maintain

In his book *Clean Code* (2009), Robert Martin captures the concepts in unit testing in the acronym FIRST:

Fast: they need to be fast so that developers can run them frequently

Independent: one test must not be dependent on, or be affected by, any other test

Repeatable: tests must be repeatable in every environment, e.g. development, QA, production

Self-validating: tests should either pass or fail automatically, without any manual intervention or inspection of output logs

Timely: tests should be written in a timely fashion (just before the application code – i.e. TDD)

Readable Tests: Coding by Intention

You should always code your tests consistently and with a well-defined, clear intent. There are four phases to doing this:

1. Setup / arrange: set up the initial state for the test.
2. Exercise / act: perform the action under test.
3. Verify / assert: determine and verify the outcome.
4. Clean-up: clean up the state created.

When working through each of these phases, you need to make sure that you express and document them clearly and thoroughly. This includes expres11sing your expected outcomes.

Right BICEP

You can use the acronym 'Right BICEP' as a guideline to help you remember areas that are important to test:

Right: Are the results right?

B: Are all the boundary conditions correct?

I: Can you check the inverse relationships?

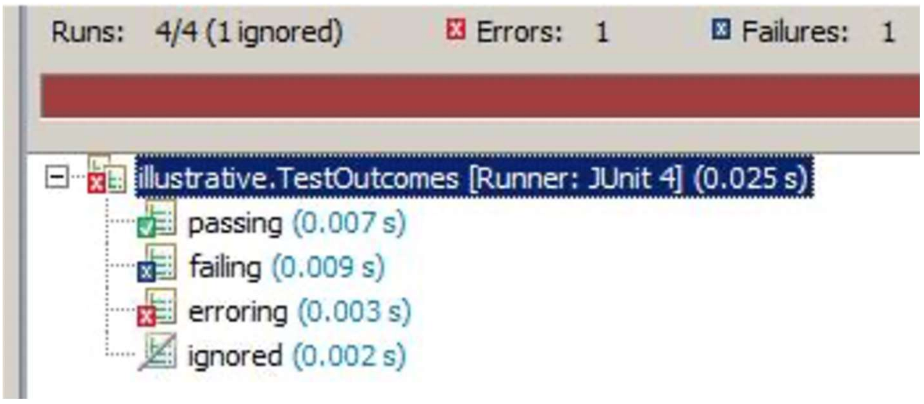
C: Can you crosscheck results using other means?

E: Can you force error conditions to happen?

P: Are performance characteristics within bounds?

Test Statuses

Each test that you carry out with have four statuses: passing, failing, erroring and ignored. These are illustrated in Figure 5 below. In this image, the red bar shows that the test has finished running. When you're running a test, this progress bar will be green.



```
Runs:   4/4 (1 ignored)        Errors:  1        Failures:  1

illustrative.TestOutcomes [Runner: JUnit 4] (0.025 s)
    passing (0.007 s)
    failing (0.009 s)
    erroring (0.003 s)
    ignored (0.002 s)
```

Figure 5

Passing: Ultimately, you want all of your tests to pass.

Failing: In TDD, you always start with a test that fails.

Erroring: The test neither passes nor fails. If this happens, then something has gone wrong and a run-time error has occurred, e.g. because a library jar hasn't been provided.

Ignored: @Test @Ignore

Manual Testing vs Automated Testing

Manual Testing	Automated Testing
Only certain people can execute the tests	Anyone can execute the test
Difficult to consistently repeat tests	Perfect for regression testing
Manual inspections can be error prone and aren't scalable	Series of contiguous testing, where the results of one test rely on the other
Doesn't aggregate, indicate how much code was exercised, or integrate with other tools (e.g. build processes)	The build test cycle is increased

Unit vs Component vs Integration

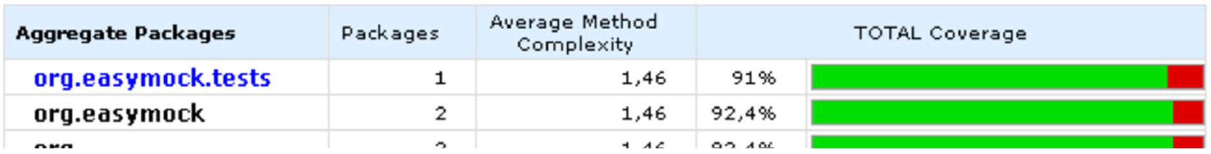
Unit Testing	Component Testing	Integration Testing
Ensures all of the features within the Unit (class) are correct	Similar to unit testing but with a higher level of integration between units	Involves the testing of two or more integrated components
Dependent / interfacing units are typically replaced by stubs, simulators or trusted components	Units within a component are tested as together real objects	
Often uses tools that allow component mocking / simulation	Dependent components can be mocked	

Code Coverage

Code coverage, or test coverage, is all about identifying what to test and **how well your tests cover your code base**. When designing a test, it's important to make sure that your conditions meet your organisation's coverage and completeness guidelines.

'White box' tests are tests where you have complete knowledge of the internals of the component being tested. This is the kind of test that you'll be carrying out, which means that **you should cover every line of code**. Clover is a tool commonly used for white box testing, and you can integrate it with Junit and Eclipse. It provides high-level summaries and line-by-line analysis of course code. Istanbul for JavaScript is another commonly used tool.

In ordinary development, you will most likely struggle to reach a 90% coverage rate (Figure 6). In TDD, however, your coverage should naturally be at least 90%.



Aggregate Packages	Packages	Average Method Complexity	TOTAL Coverage	
org.easymock.tests	1	1,46	91%	🟩🟥
org.easymock	2	1,46	92,4%	🟩🟥
[illegible]	[illegible]	[illegible]	[illegible]	🟩🟥

Figure 6

Coverage Metrics

There are four ways in which you can measure your code coverage: method, symbol, branch and cyclomatic complexity.

Method coverage measures the percentage of methods touched by your test. It's a very coarse-grained measure, which means that it can be inaccurate.

Symbol coverage measures the percentage of sequence points (statements) that have been exercised, but it doesn't show if both sides to an 'if' condition have been covered.

Branch coverage measures the percentage of completely executed blocks in a method. This means that both sides to an 'if' condition will be checked.

Cyclomatic complexity measures how many linearly independent paths there are through your code. It reflects the number of tests to get 100% branch coverage. If your method has a cyclomatic complexity of 15 or more, it's too complex and you should refactor it. We'll talk about refactoring in the next chapter.

Percentage Isn't Everything

You may have come across test generation tools that auto-generate unit tests with 100% coverage, but these tests are ultimately worthless. Your method might be covered, but can you be sure that all of the edge / corner cases have been adequately tested? And if you're using a get / set method that simply reads / writes a field, is it really going to add value to cover these? You'll most likely end up with a test class full of trivial get / set methods. This makes it incredibly difficult to identify the tests with real value. You should consider creating one of these methods as a helped to 'get things set up'. Once it's served this purpose, you can delete it. You can also use them as a means of detecting regions of code that haven't been tested.

Emma is one example of a code analysis tool that you can keep running it to make sure there aren't any regions of your application code which haven't be tested. Other tools check for conformity to coding rules, such as PMD and checkstyle. CodePro Analytix is a very powerful suite of code analysis tools that has been made freely available by Google. You can use it to not only measure code coverage and cyclomatic complexity, but also to detect regions of duplicated code, analyse dependencies between classes, etc. For more information on CodePro, visit the Google Developers website.

'100% test coverage does not guarantee that your application is 100% tested'

– Massol

Remember that failing tests count towards coverage just the same as passing tests do.

Quality over Quantity

It is easy to think that the more tests you have, the better, but this isn't the case. Having too many tests can cause the feedback cycle to slow down and increase your maintenance costs. This eventually diminishes any additional value. To avoid this, try not to duplicate any tests. Identify which ones are necessary (remember the 'Right BICEP' acronym?), and focus on those.

The quality of your tests is far more important than having lots of them. They need to be demanding. Jester, the JUnit test tester, is a tool that you can use to test the quality of your tests. However, it does this through mutating the source code and recompiling it, which is very slow. It's important that you can run your tests quickly, so you should use Jester in a targeted manner. Jester is also known for producing false positives by identifying changes to a source code that don't cause any tests to fail. In these cases, you may have to discount the results.

Testing Patterns

By now you should be fairly confident in your knowledge of testing methods and why we use them. Just as designing software has patterns, so does testing it. Let's take a look at some of the patterns that might help make the process a little easier for you.

Assertions

An assertion is an expression that encapsulates a testable logic about the product you're testing. It will be true, unless there's a bug in the program. Usually, you'll have exactly one assertion per test. A **resulting state assertion** is a standard test. This is the state that we expect. The following assertions are for when you aren't conducting a straightforward, standard test.

Guard Assertion

A guard assertion is a precondition that you set for the test to be correct, before applying a resulting state assertion. For example, a remove() method really removes something that was there:

```
assertThat(basket.list(), hasItem(new Product("4 Candles", 9)));

basket.remove(1);

assertThat(basket.list(), not(hasItem(new Product("4 Candles", 9))));
```

Matchers generally make a custom message superfluous, but here we can clarify the role of the guard assertion:

```
assertThat(basket.list(), describedAs("A pre-requisite for the test", hasItem(new Product("4
Candles", 9))));
```

With this method, you should write the assertion as your first step in writing the test method. Identify what the right answer is before you move on.

Remember that, ideally, you'll have one assertion per test. The guard assertion pattern illustrates the exception that proves the rule!

Delta Assertion

If you can't guarantee the absolute resulting state, test the delta (difference) between the initial and resulting states. Use the delta assertion where there is something slightly outside the control of your test; where its initial state isn't determinate, but where the operation you're testing will have a determinate effect upon it. Performance tests are often implicitly delta assertions because they take a timing before an operation, and the timing again after it, and assert that the difference is below a certain threshold.

Custom Assertion

If the code verifying your assumptions is particularly long, extract it to a method. The method should contain an assert(). If it contains logic, you should test that as well. Consider writing your method as a custom matcher, which will combine with other matchers:

```
assertThat("12345", not(isAPostCode()));

assertThat(codes, hasItem(isAPostCode()));
```

The custom assertion pattern helps your test code respect DRY ('don't repeat yourself', i.e. there aren't any duplications). For example, you might need to assert that all the cards in an initially generated deck are different, and then again, when the shuffle operation has been performed. Remember to be careful about introducing complexity into your tests, without having a test to verify that this functionality itself is correct.

Parameterised Test

Parameterised tests are an example of data-driven testing, where you use one test method for multiple data sets. The data is hard-coded in a 3D array, or comes from an external source (e.g. a file or db):

```
@Parameters

public static Collection makeData() {

 return Arrays.asList( new Object[][] {{ 1,  "Jan" },{ 3,  "Mar" }, { 12, "Dec" } });

}

// { input to fn, expected output }

public UtilsTest(int input, String output) {

   this.input = input;

   this.expected = output;
```

JUnit lets you set up parameterised tests with annotations. You need to annotate the test class itself as follows:

```
@RunWith(value=Parameterized.class)

public class UtilsTest {

  private int input;

  private String expected;

}
```

The test method looks ordinary, except that it uses instance variables of the class:

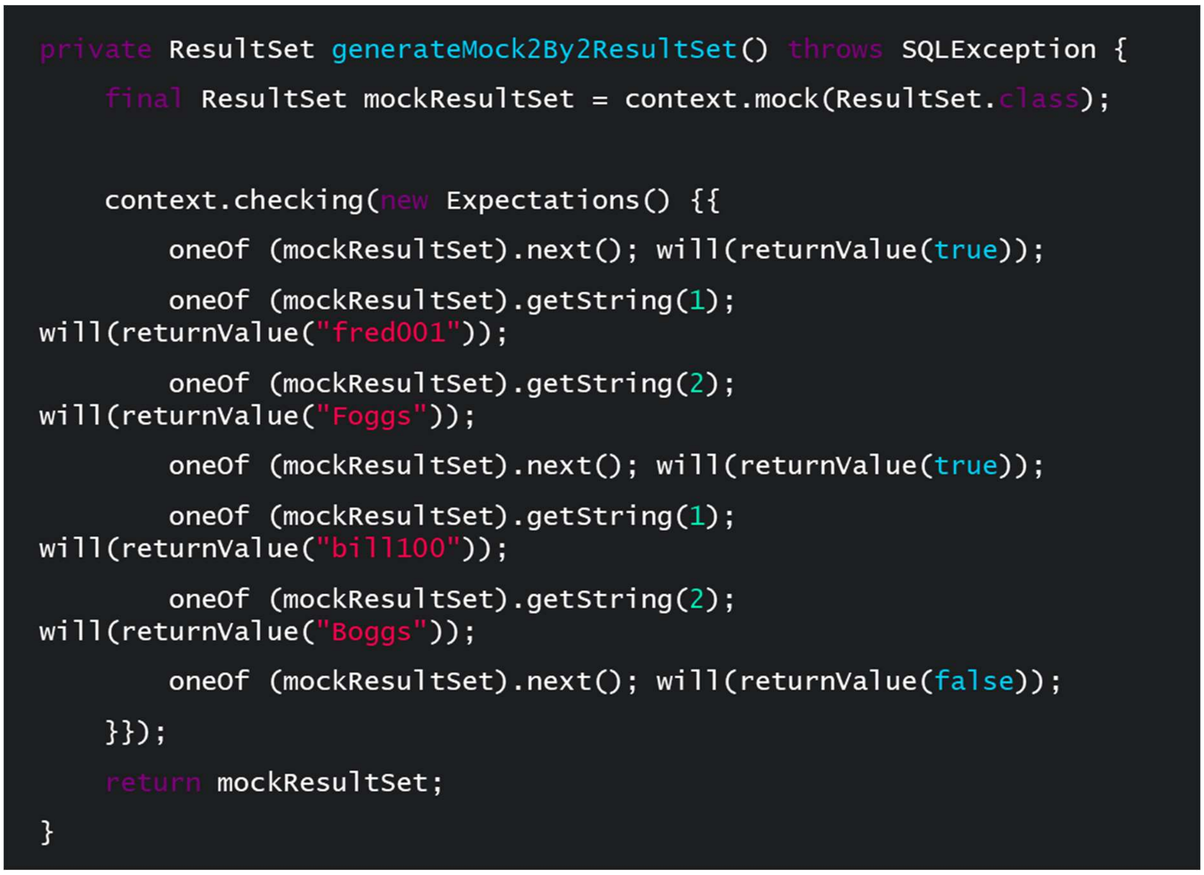
```
@Test

public void testGetMonthString() {

  assertEquals("Incorrect month String", expected, Utils.getMonthString(input));

}
```

For each array of values from the `@Parameters` annotated method, a test is run where those values are passed to the constructor, and any `@Before`, then `@Test`, then `@After` method in the class is invoked. You could also define a parameterised exception test, in which the `@Parameters` annotated method defines negative data for the function – e.g. a negative number, the corner case 0 (in many systems, January is indexed by 0), and a very large number. In these cases, there is no expected value to match with the input; we expect the function to throw an Exception.

Parameterised Creation Method

A parameterised creation method hides attributes that are essential to fixtures, but irrelevant to test. It involves factoring out fixture object creation from setup to PCM, and is useful when you're creating a complex mock, especially if you're going to use it in multiple tests.

With the **parameterised creation method**, you can encapsulate complex fixture creation into a self-explanatory method. The Figure 7 example uses the **JMock** framework to create something which fakes the ResultSet you might get by running a SQL query like "SELECT id, surname FROM users" against a database.



```
private ResultSet generateMock2By2ResultSet() throws SQLException {
    final ResultSet mockResultSet = context.mock(ResultSet.class);

    context.checking(new Expectations() {{
        oneOf (mockResultSet).next(); will(returnValue(true));
        oneOf (mockResultSet).getString(1);
will(returnValue("fred001"));
        oneOf (mockResultSet).getString(2);
will(returnValue("Foggs"));
        oneOf (mockResultSet).next(); will(returnValue(true));
        oneOf (mockResultSet).getString(1);
will(returnValue("bill100"));
        oneOf (mockResultSet).getString(2);
will(returnValue("Boggs"));
        oneOf (mockResultSet).next(); will(returnValue(false));
    }});
    return mockResultSet;
}
```

Figure 7

Extra Constructor

If your existing constructor hard-codes some dependencies, you can use the extra constructor pattern to inject dependencies by test (mock or stub). You do this by introducing a 'trap door' that makes your code easier to test:

'My car has a diagnostic port and an oil dipstick. There is an inspection port on the side of my furnace and on the front of my oven. My pen cartridges are transparent so I can see if there is ink left.'

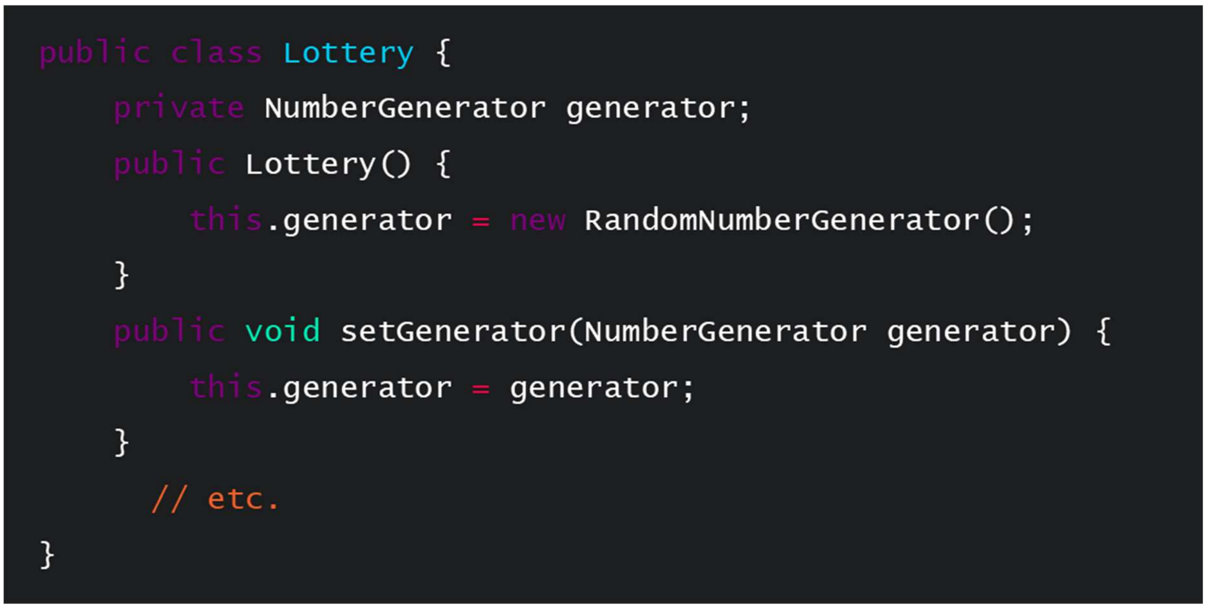
– Ron Jeffries

Constructor and Setter Injections

In a constructor injection, the API signals that the parameter(s) isn't optional – you need to supply it when creating the object.

In a setter injection, the API signals that the dependency is optional / changeable.

Let's go back to the number generator example we explored with mocks. With either a constructor or setter injection, you can have a default constructor which hard-codes the default dependency, e.g. a setter which allows that to be overridden (with a mock, in the test):



```
public class Lottery {
    private NumberGenerator generator;
    public Lottery() {
        this.generator = new RandomNumberGenerator();
    }
    public void setGenerator(NumberGenerator generator) {
        this.generator = generator;
    }
      // etc.
}
```

Figure 8

Test-Specific Subclass

When you're not permitted to modify an actual class, you can create a behaviour-modifying or state-exposing subclass. Not that this isn't usually a TDD approach. This patterns inherits a method to test as-is, and the test creates an instance of the testable subclass, injecting the mock.

Sub-classing is about where the dependency comes from; the key is that the code we wish to test is inherited untouched in the subclass.

An example of this could be in a lottery class. Default constructor creates a real `RandomNumberGenerator` and the `NumberGenerator` field has protected visibility. `TestableLottery` extends the lottery:

```
public TestableLottery(NumberGenerator generator) {

  this.generator = generator;

}
```

Factory

The factory pattern provides the means for the test to change the type returned. The test sets the factory to return the mock or stub without changing the application code semantics. You use this method for dependencies with 'extract and override': extract dependency creation to a factory, and use the override method in the test-specific subclass.

CUT uses the factory method. The CUT has:

```
public class Lottery {
   NumberGenerator generator;
   protected NumberGenerator getGenerator() {
      return new RandomNumberGenerator();
  }
  // etc.
}
```

The test-specific subclass can simply @Override the above factory method. Using Java inner classes, we don't even need a separate class for the test-specific subclass; you can create it, for example, as an anonymous inner class:

```
@Test public void generateArrayAsString() {
   final NumberGenerator mockGenerator =
                  createMock(NumberGenerator.class);
   Lottery lotto = new Lottery() {
      @Override protected NumberGenerator getGenerator() {
         return mockGenerator;
      }
   };
   expect(mockGenerator.generate(anyInt())).andReturn(2).times(6);
   // etc.
}
```

Object Mother

An object mother is a class that you can use to help create example objects that you can use in your tests, like with a factory pattern. However, it goes beyond the factory pattern in that it allows you to customise the objects you create, it provides methods to update the objects during the tests, and it allows you to delete the object from the database once you've completed your tests.

Test Doubles

Your code interacts with other things. These could be network resources, such as databases and web services, code being developed in parallel by another team, or containers. Objects that the class under test interacts with are called

collaborators, and substitutes for them used in testing are sometimes referred to as **test doubles** (like a stunt double). We use test doubles in integration testing if the collaborators don't exist yet, or aren't available for use in tests. They allow us to run the test in its real environment, and to test interactions between your class and the rest.

Mocks

Mocking is a method for testing your code functionality in isolation. It doesn't require database connections, file server reads or web services.

The mock object itself does the 'mocking' of the real service, returning dummy data. It does this by simulating the behaviour of a real method, external component. Your code calls methods on the mock object, which delivers results as set up by your tests.

Mocks have an expectation of the calls that they're likely to receive. If they receive a call they don't expect, they can throw an exception. During verification, they're checked to make sure that they received all of the calls they expected.

It's important to remember that you're not creating an object with state (records with mock data sets), you're creating an object which will respond to (method calls from) your code as if it had a certain state.

There are several dynamic mock frameworks in Java, including EasyMock, JMock, Mockito, Powermock and JMockit.

When to Use Mocks

In their work *Endo-testing: Unit Testing with Mock Objects*, Mackinnon, Freeman and Craig say that you should use mock objects when the real object…

- Has non-deterministic behaviour
- Is difficult to set up
- Has behaviour that is hard to cause (such as network error)
- Is slow
- Has (or is) a UI
- Uses a callback (tests need to query the object, but the queries are not available in the real object (for example, 'was this callback called?')
- Does not yet exist

If another team is writing the code, you may not be able to access the interfaces but, if you do have a set of agreed interfaces, mock them. Set the desired expectations, and from there you can **debug your code**.

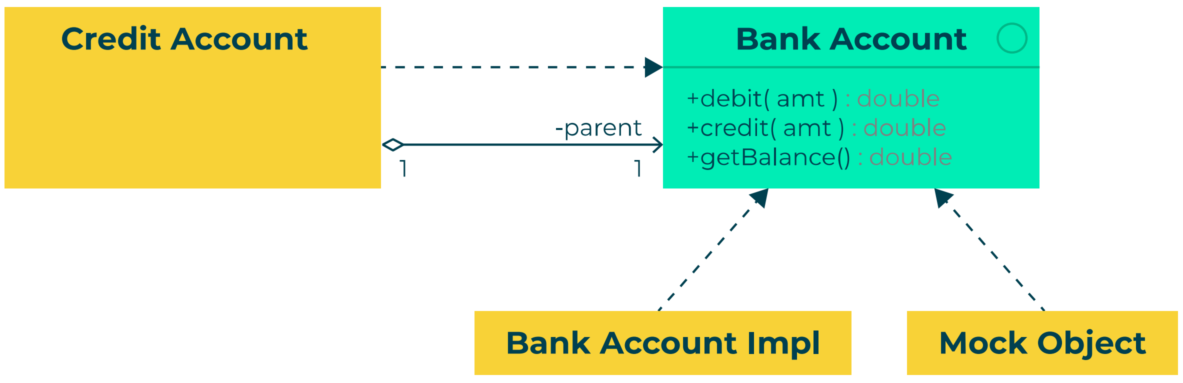


Figure 9

Drawbacks of Mocking

There are four main drawbacks to using mock objects.

1. **Mocks don't test interactions with container or between the components.** This means that you can't have full confidence that the code will run in target container, so you still need integration tests.
2. **Tests are coupled very tightly to implementation.** When you set expectations, you make the tests mirror internal implementation.
3. **Mocks don't test the deployment part of components.**
4. **Most frameworks can't mock static methods, final classes and methods.** However, you can mock static methods with a workaround. Powermock extends Easymock and Mockito to enable you to do this.

Stubs

Another type of test double, stubs are a controllable replacement for existing dependencies. Stubs provide pre-programmed answers to calls they receive during the test. They don't usually respond to anything outside of what you've programmed in for the test.

Stubs are good for course-grained testing, such as replacing a complete external system like a web server or database. However, they're often complex to write and have their own debugging and maintenance issues.

Mocks vs Stubs

Mocks and stubs are not necessarily mutually exclusive; it may be appropriate, depending on the class you're testing, to use a combination of both mocks and stubs.

Both mocks and stubs can be handwritten, simple classes – if they remain simple, they'll have the advantage of simplicity and readability over frameworks. A dynamic fake object is a mock or stub created at runtime – using a framework, rather than handwritten. jMock and EasyMock are two leading Java mock objects frameworks, but there are others. See here for more information on them. Also listed are mockito, rMock, and SevenMock. Some argue that 'isolation framework' is a better term than 'mocking framework', because it more clearly signals the intent: isolating the unit tests from their external dependencies.

Stubs enable tests by replacing external dependencies. Asserts are against the class under test, not the stub:

$$\begin{tikzcd}\text{Test} \arrow[r, leftrightarrow, "\text{asserts}"'] & \text{CUT} \arrow[r, leftrightarrow, "\text{interacts}"'] & \text{Stub}\end{tikzcd}$$

Figure 10

Using a mock is much like using a stub, except that the mock will record the interactions, which can be verified. Asserts are verified against the mock:

$$\begin{tikzcd}\text{CUT} \arrow[r, leftrightarrow, "\text{interacts}"'] & \text{Mock} \arrow[r, leftrightarrow, "\text{asserts}"'] & \text{Test}\end{tikzcd}$$

Figure 11

Mocks can be used for stubbing (as above), or a test can ask a mock to verify that the CUT interacts with the external dependency in the correct way.

Tests should test a single thing, so there should be at most one mock per test – all other fakes should be stubs. Having multiple mocks means you're not just testing a single thing.

Dependency Injection (DI)

Injecting dependencies means setting relations between instances. We do this to remove tight coupling and make our unit tests more flexible. To illustrate a dependency, you should always **use an interface instead of a concrete class:**

```
public interface IEngine { }

public class FastEngine implements IEngine { } ⍰ tight coupling
```

You then inject the interface into the class:

```
public class FastEngine {

   private IEngine engine;

   public FastEngine(IEngine engine) { this.engine = engine; }

}
```

Methodology

There are two ways of performing a dependency injection: by **constructor** or by **setter**. With either method, you need to inject concrete instances at runtime rather than creating them. You also need to find a way of managing or wiring the dependencies. In Java, the Spring framework lets you do this by supporting DI out of the box using annotations.

When using the constructor method, the value of the pet is set as below:

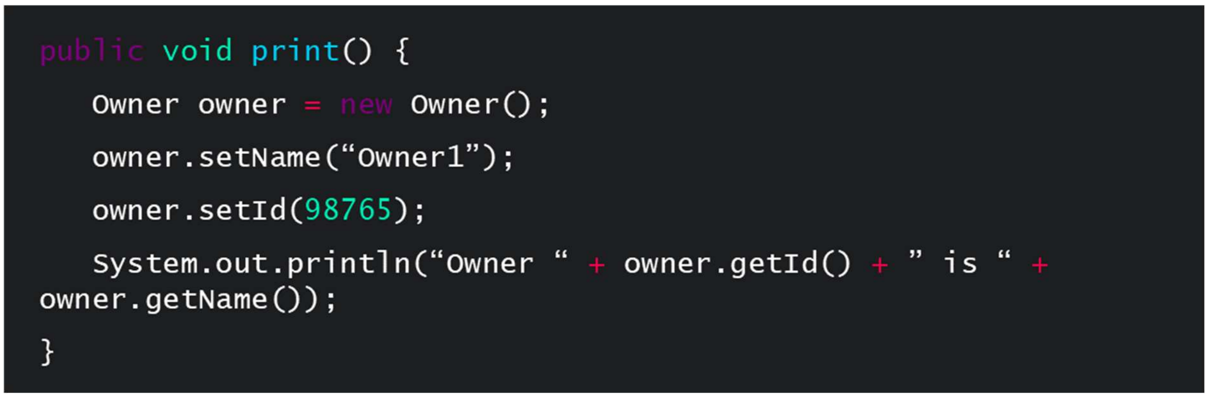
```
public class Owner {

  Pet pet;

  public Owner(Pet pet) {

    this.pet = pet;

  }

}
```

When using the setter method, the value of the pet is set as below:

```
public class Owner {

  Pet pet;

  public void setPet(Pet pet) {

    this.pet = pet;

  }

}
```

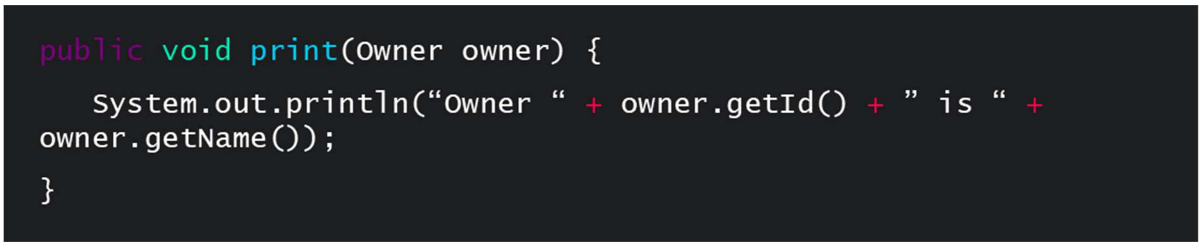
Example

Figure 12 shows our line of code without an injection, where Figure 13 shows the same code using an injection:



```
public void print() {
    Owner owner = new Owner();
    owner.setName("Owner1");
    owner.setId(98765);
    System.out.println("Owner " + owner.getId() + " is " +
owner.getName());
}
```

Figure 12



```
public void print(Owner owner) {
    System.out.println(“Owner “ + owner.getId() + ” is “ +
owner.getName());
}
```

Figure 13

Containers

When injecting complex dependencies, you can use a dependency injection container to inject the mock objects into your unit tests during unit testing.

Some examples of dependency injection containers include Spring DI, Butterfly DI Container, Dagger, Guice and PicoContainer.

If your dependencies are simple to mock out, then your tests won’t usually need a dependency injection container.

Refactoring with Existing Tests

Definition and Principles

Refactoring is the process of restructuring your code without changing its external behaviour. It preserves the software's functionality, but improves its design and structure. There are four key principles involved in refactoring:

1. Keep it small. Refactor in small increments to create a modest overhead for the work in the team.
2. Business catalysts. Make sure that you refactor at the right time for your organisational needs, not just whenever the team decide they want to do it!
3. Team cohesion. Your team needs to apply a high level of communication and teamwork.
4. Transparency. Your team needs to be completely open with stakeholders about the costs involved in refactoring.

Benefits

Refactoring is not the same as rewriting – it involves changing the **structure** of your code. Nowadays, you can refactor your code automatically through a tool such as Eclipse, which makes the process much quicker. But why do we refactor in the first place? Well, refactoring…

- Makes code easier to understand
- Improves code maintainability
- Increases quality and robustness
- Makes code more reusable
- Typically to make code conform to design pattern
- Many now automated through Eclipse, etc.
- Improves the design of software
- Makes it easier to find bugs, as code is cleaner

'Do. Not. Skip. Refactoring… The single biggest problem I've witnessed after watching dozens of teams take their first steps in test-driven development is insufficient refactoring.'

– *Lasse Koskela, Test Driven, p. 106*

Common Refactorings

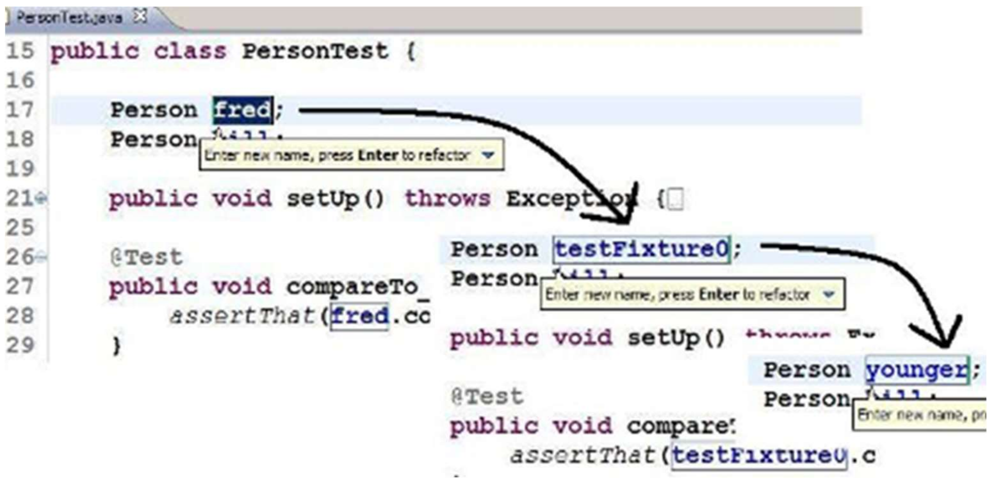
There are far too many refactoring for us to go into detail on all of them in this guide. For a comprehensive list, see Martin Fowler's Catalog of Refactorings. For the purpose of this course, we'll explore some of the most common refactorings.

Refactor-Rename

Using refactor-rename is much more efficient than manually editing your file. When you rename a variable, it renames all occurrences of the variable. When you rename a class, it renames the compilation unit too (the .java file), and all known references to the class. When you rename a method, it renames all known calls to that method. Finally, when you rename a package, it renames all the package declarations within.

At all times, you should be thinking about how you can make your code as self-documenting as possible. Replace unclear and ambiguous identifiers with clear and informative ones. For each variable, choose a name which reflects what it represents in the problem domain, not its implementation (e.g. in the enhanced for loop, nextClient rather than nextElement). Only choose one-letter identifiers for variables such as array indexes.

To do this in Eclipse, repeat <ALT> <SHIFT> R as in Figure 14 below, until you're satisfied you have an identifier that best reflects what the item represents.



PersonTest.java

```
public class PersonTest {

    Person fred;
    Person
    Enter new name, press Enter to refactor

    public void setUp() throws Exception {

    @Test
    public void compareTo_
        assertThat(fred.cc
    }
```

```
    Person testFixture0;
    Person
    Enter new name, press Enter to refactor

    public void setUp()

    @Test
    public void compare
        assertThat(testFixture0.c
```

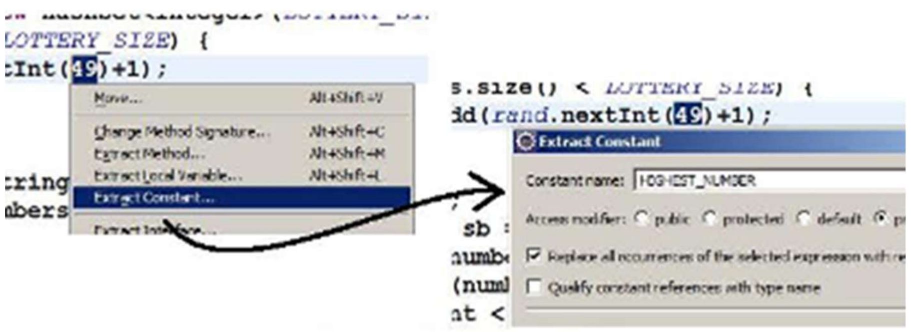
```
    Person younger;
    Person
    Enter new name, pr
```

Figure 14

Extract Constant / 'No Magic Numbers'

Avoid hard-coding literal values, like 49, into your code. This is because anyone coming along to maintain the code will be scratching their head wondering what this number represents. There's also likely to be the issue of maintainability – it's likely that this value will occur more than once, and you might need to revise it.

In Eclipse, highlight the number, then <CTRL> <SHIFT> T , the general context menu for refactoring (Figure 15). Select Extract Constant and choose a meaningful name. The constant will be declared as a class field, and the literal occurrence you selected will be replaced. Note that, if there are further occurrences of the value within the class, you need to search for them manually and replace them.



```
LOTTERY_SIZE) {
tInt(49)+1);
tring
mbers
```

Move... Alt+Shift+V
Change Method Signature... Alt+Shift+C
Extract Method... Alt+Shift+M
Extract Local Variable... Alt+Shift+L
Extract Constant...
Extract Interface...

```
s.size() < LOTTERY_SIZE) {
dd(rand.nextInt(49)+1);
 sb
numbe
(numl
at <
```

Extract Constant

Constant name: HIGHEST_NUMBER

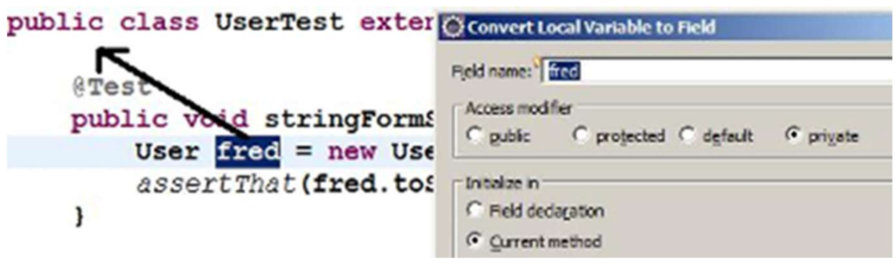
Access modifier: ( ) public ( ) protected ( ) default [✓ p]

[✓] Replace all occurrences of the selected expression with re

[ ] Qualify constant references with type name

Figure 15

For Convert Variable to Field (Figure 16), highlight a variable local to a method so that you can pull it to class field level. This could be part of a 'remove duplication' refactoring, such as in the case of a unit test fixture as illustrated below.



```
public class UserTest exter

    @Test
    public void stringForms
        User fred = new Use
        assertThat(fred.toS
    }
```

Convert Local Variable to Field

Field name: fred

Access modifier: ( ) public ( ) protected ( ) default [✓ private]

Initialize in: ( ) Field declaration [✓ Current method]

Figure 16

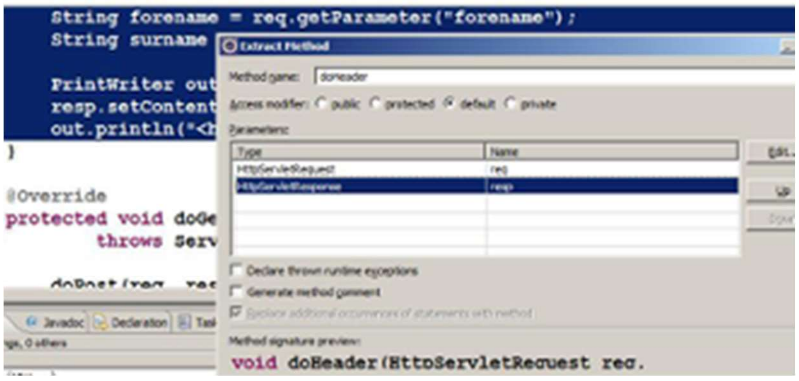
At this point, you need to make a decision. Since the variable is now a class-level field, what class accessibility should it have? You can choose from:

1. **Field declaration**, i.e. the whole line is in effect pulled out of the method.
2. **In the current method**, i.e. the declaration `User fred;` is pulled up to the class level and the initialisation remains where it is.
3. **In the class constructors**, i.e. as 2, but the initialisation statement will be moved to all and any constructors.

None of these is quite right for a Test class, so best is to accept 2 then manually move it to setUp().

Extract Method

You can use the extract method <ALT> <SHIFT> M to remove code duplication and to break your code up if it gets too long. You can also use it to improve the clarity of your code, by moving lines to a method which better expresses their intent. For example, in Figure 17, you might ask why the call reader.readLine() exists when it does nothing. In this case, you could extract it to method discardHeaderLine(). Your code then becomes self-documenting, which is much more efficient than adding comments!



```
    String forename = req.getParameter("forename");
    String surname
    PrintWriter out
    resp.setContent
    out.println("<h
}

@Override
protected void doGe
        throws Serv

    doPost(req, res
```

Extract Method

Method name: doHeader

Access modifier: ( ) public ( ) protected (•) default ( ) private

Parameters:

Type	Name
HttpServletRequest	req
HttpServletResponse	resp

☐ Declare thrown runtime exceptions

☐ Generate method comment

☑ Replace additional occurrences of statements with method

Method signature preview:

```
void doHeader(HttpServletRequest req,
```

Figure 17

For the purpose of illustration, let's extract some lines of code into a method, and then modify the method to make it as testable as possible. We start with some servlet code (inside a doGet() or doPost() method):

```
String forename = req.getParameter("forename");

String surname = req.getParameter("surname");

PrintWriter out = resp.getWriter();

resp.setContentType("text/html");

out.println("<h4>Author: " + surname + ", " + forename + "</h4>");
```

Eclipse detects that, if it's to make a method out of this, it'll need two input parameters, and will be void return type. Notice the method signature preview at the bottom of the dialogue. When you click OK for the refactoring, those lines will be replaced by a call to your new method, i.e. doHeader(req, resp).

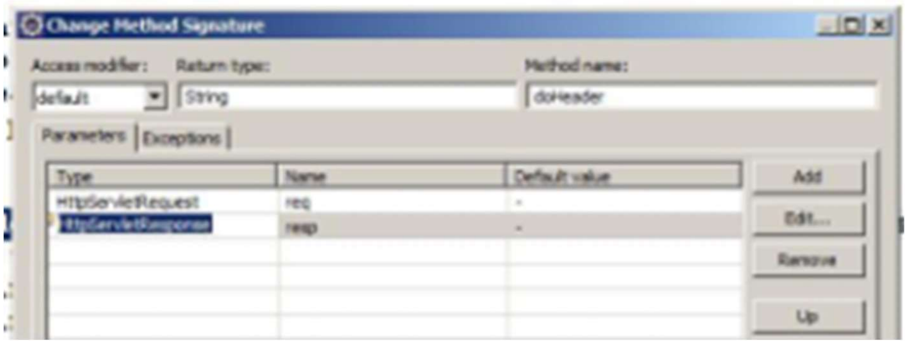
Notice the option in the dialogue to replace additional occurrences of statements with method. If Eclipse had detected that those same lines were repeated elsewhere in this class, this option would make sure that those lines would be replaced too.

Extract Method for Testability

There are six steps to this refactoring (Figure 18):

1. Highlight lines.
2. Invoke refactoring.
3. Enter new method name and check accessibility.
4. If necessary:
    - Introduce local variable for return value
    - Get method code to compute return value as appropriate
    - Return the return value and adjust method return type
    - Adjust the call to the new method
5. Change method signature.

6. Make the new method as cohesive as possible.


Change Method Signature

Access modifier: default
Return type: String
Method name: doHeader

Parameters | Exceptions

Type	Name	Default value
HttpServletRequest	req	-
HttpServletResponse	resp	-

Figure 18

The method you create initially is void, but you're aiming for testability, so ideally you want a method that returns something to you. With this kind of method, you can declare a String or StringBuilder local variable, build it up in the method, and return it, rather than printing it out to the PrinterWriter out.

This also means that the method will have no need for its second input parameter, the Response object from which the Writer is obtained. You can use a second refactoring, Change Method Signature..., to make it simpler and more cohesive. This will give you something like the following:

```
String doHeader(HttpServletRequest req) throws IOException {

   String returnValue = null;

   String forename = req.getParameter("forename");

   String surname = req.getParameter("surname");

   returnValue = "<h4>Author: " + surname + ", " + forename + "</h4>";

   return returnValue;

}
```

You can adjust where it's called to out.print(doHeader(req)); and mock the HttpServletRequest instance that's passed to it. You can then assert what string you expect out of it.

Extract Class

When your class is too large, you can break it into smaller classes with cohesive behaviour. Alternatively, if there is related functionality in multiple classes, you could break your large class into smaller functionality-dedicated classes that are easier to maintain.

Replace Inheritance by Delegation

This refactoring is also called 'Favour Composition over Inheritance'. Let's go through an example...

Suppose: class Deck<Card> extends ArrayList<Card>

Reasoning: a Deck is a list of Cards.

However, this is wrong - the relationship is has-a , not is-a.

This doesn't expose the unnecessary methods of ArrayList.

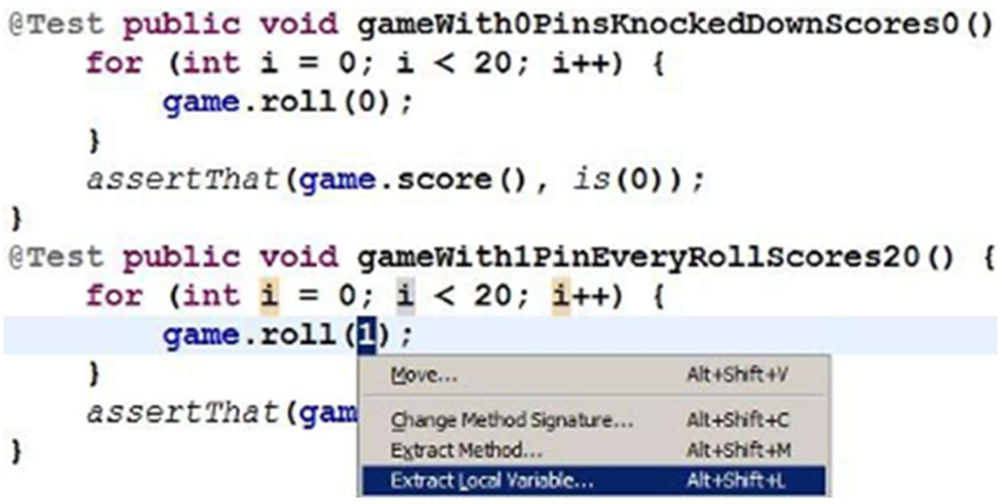
You should expose only methods a Deck needs, and delegate their implementation to the contained ArrayList.

Remove Duplication

You don't need to be an adherent of TDD or XP to recognise the importance of Remove Duplication. Duplication of logic or responsibility is like coding kryptonite - it introduces the risk of making a change in one place but not another, which is a huge problem when it comes to maintenance.

Remember DRY – Don't Repeat Yourself! There are four steps to removing duplications. Let's take a look at how it works:

1. Imagine that you have two blocks of code that are almost identical. First, extract value(s) where they differ to variable(s) (Figure 19).
2. These will become input parameter(s) to a single common method.
3. Place declaration of local variable `int pins = 1` outside loop.
4. Apply Extract Method refactoring to the loop.



```
@Test public void gameWith0PinsKnockedDownScores0()
    for (int i = 0; i < 20; i++) {
        game.roll(0);
    }
    assertThat(game.score(), is(0));
}
@Test public void gameWith1PinEveryRollScores20() {
    for (int i = 0; i < 20; i++) {
        game.roll(1);
    }
    assertThat(gam
}
```

Move... Alt+Shift+V
Change Method Signature... Alt+Shift+C
Extract Method... Alt+Shift+M
Extract Local Variable... Alt+Shift+L

Figure 19

In this example, which is a variant of the one in 'Uncle Bob' Martin's [Bowling Game Kata](), the test methods would end up looking like this:

```
@Test public void gameWith0PinsKnockedDownScores0() {

  roll20(0);

  assertThat(game.score(), is(0));

}
```

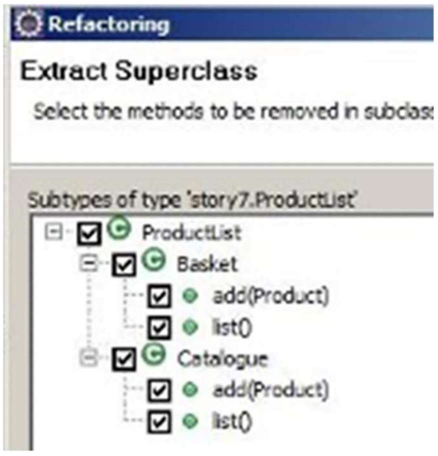
The commonality between the two locks of code would be captured in the following method:

```
private void roll20(int pins) {

  for (int i = 0; i < 20; i++) {

    game.roll(pins);

  }

}
```

Extract Superclass

Extract Superclass is another refactoring for removing duplication. There are a few steps to this refactoring, so let's go through an example.

1. Suppose that Basket and Catalog have commonality: both have a List of Products, methods to `add()` and `list()`. Choose one, e.g. Basket > Extract Superclass (Figure 20).



Refactoring

Extract Superclass

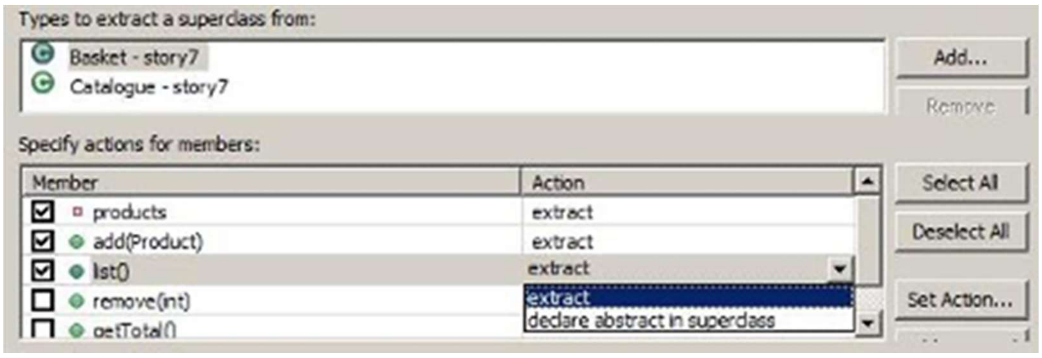
Select the methods to be removed in subclas

Subtypes of type 'story7.ProductList'

```
☑ ProductList
  ☑ Basket
    ☑ add(Product)
    ☑ list()
  ☑ Catalogue
    ☑ add(Product)
    ☑ list()
```

Figure 20

2. Add the other types to extract a superclass from.
3. Select your methods and files to be extracted (Figure 21).
4. Basket etc. will extend a new superclass.



Types to extract a superclass from:

Basket - story7
Catalogue - story7

Specify actions for members:

Member	Action
☑ products	extract
☑ add(Product)	extract
☑ list()	extract
☐ remove(int)	extract / declare abstract in superclass
☐ getTotal()	

Figure 21

Eclipse has a three-step wizard for the Extract Superclass refactoring:

1. Name to the superclass to be created, e.g. ProductList. If you already have the second class about to become its subclass, in the 'Types to extract a superclass from' add it. Check off the members which will be pulled into the new superclass.
2. Explicitly check off which methods are to be removed from the soon-to-be-subclasses.
3. Preview the changes.

Coding to Interfaces

You know how it works by now – let's walk through an example (Figure 22) of this refactoring!

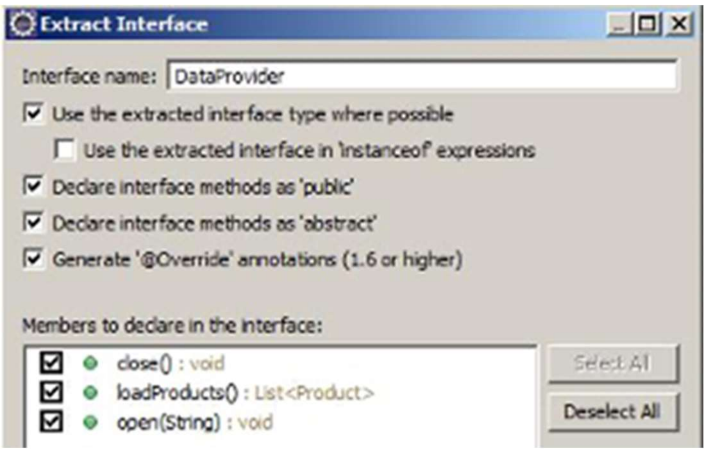
1. Suppose a class needs a Repository / DAO:

   ```
   CSVFileProvider provider = new CSVFileProvider();
   ```

2. Candidate for decoupling interface and implementation:

   ```
   DataProvider provider = new CSVFileProvider();
   ```

3. Plug in different implementations without affecting the rest of your code.
4. Choose method names which are neutral about data source.
5. Choose exception type at same level of abstraction.



Interface name: DataProvider

☑ Use the extracted interface type where possible

☐ Use the extracted interface in 'instanceof' expressions

☑ Declare interface methods as 'public'

☑ Declare interface methods as 'abstract'

☑ Generate '@Override' annotations (1.6 or higher)

Members to declare in the interface:

☑ close() : void

☑ loadProducts() : List<Product>

☑ open(String) : void

Figure 22

Coding to interfaces is widely seen in uses of the Collections API:

```
List<Product> products = new ArrayList<Product>();
```

If you look in the API docs for java.util.List, you'll find 25 different operations you may want to perform on a list-like data structure. Crucially, there are different algorithmic implementations of this concept, and each has its advantages and disadvantages. That means one or other implementation may be more effective, depending on the sorts of operations you're going to perform on it. By declaring the variable products to be the type of the interface, you know that you can perform all the operations you want on it – but if subsequently a different realisation of that concept of List turns out to be relevant, you can plug that in with no impact on the rest of your code.

Data providers are another classic example. Imagine that in version 1 of your application, the data source is to be a flat file, but version 2 will need it to be something else, like a database or a web service. Design your provider class with this flexibility in mind: call a method open() rather than openFile(). Don't swallow exceptions, but don't throw IOExceptions either (what good will that be to a version that access a RDBMS?). Create your own Exception type at the appropriate level of abstraction – e.g. DataAccessException. Finally, follow the 'catch and rethrow' pattern: catch the low-level IOException and encapsulate it within an instance of the higher-level exception type, which is thrown.

Legacy Code

Legacy code is a type of application system source code that's no longer supported. However, the term can also be used for any code that you don't understand and that you're finding difficult to change. In this case, you might ask yourself, 'Should I refactor or rewrite this code?'

Lots of developers enjoy rewriting the code rather than fixing existing code, but a total rewrite only makes sense if you fully understand the full requirements of the system. The existing codebase could be so huge and such a mess that actually refactoring it would be more trouble than it's worth. Refactoring requires you to fully understand the codebase. If it's poorly written, it's going to be difficult to read and understand.

You have three options here:

1. Leave the old code alone and write more legacy code. This is a very common approach, but is also the **wrong** approach as it makes the code harder to update and maintain.
2. Set some time and resources aside to completely rewrite the system from scratch.
3. Approach the system in a pragmatic way and slowly and incrementally improve it – this is less intrusive.

Seams

Seams allow you to substitute classes and functions. With object seams, the dependency situation is based on wither inheritance or interface implementation.

This example is based on the substitution principle:

```
public void ProcessAccount(AccountProcessor proc, Account acc)

{

}

...

class TestAccountProcessor: public AccountProcessor

{

   // Substitute implementation

}
```

When working with legacy code, try to substitute instead of changing the code. If you have to change it, change the smallest amount possible.

You can use linker seams to define different builds by varying the classpath. For can define a substituted set of classes within the package, and change the classpath to create two different builds.

Pre-processor seams are based on pre-processor directors managing the substitution. As the name implies, this type of seam requires a pre-processor.

Refactoring Without Tests

Code that has little or no test coverage is usually badly designed. It makes it hard to know if your code changes with break other parts of the application, which in turn makes it harder to write tests. This becomes a vicious circle or sloppy code and low test coverage (Figure 23):

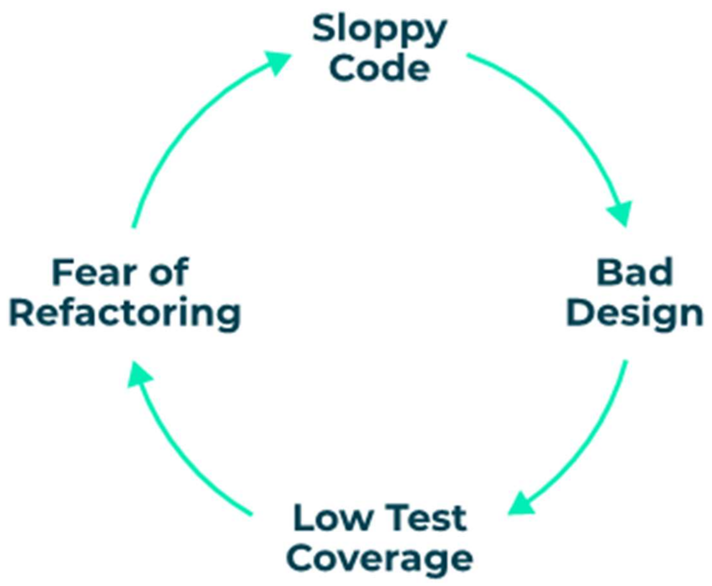


Figure 23

You should always try to have unit tests in place before you start to refactor your code, but sometimes it's necessary to refactor without having unit tests.

In some cases, you may be told not to refactor code if it has no unit tests. If this happens, individuals would have to put in the effort of creating unit tests before the code can be refactored. The effort to do this is too high, so it simply doesn't get done and the code stays unimproved.

Many tools have automatic refactoring options built-in. For example, in Eclipse-IDE you can right click on a source file and select the refactor option.

Small step refactoring is the process of making very small simple steps that are so trivial, there is almost no chance of making a mistake. This can end up making a big refactor, as the whole process of continuing to make small steps creates a net effect.

Testing Techniques

Test-Driven Development

Definition

'Test-first programming is the least controversial and most widely adopted part of Extreme Programming (XP). By now the majority of professional Java™ programmers have probably caught the testing bug'

– Elliotte Rusty Harold

Test-driven development (TDD), or test-driven design, is a software development process that follows a short, repeating cycle of turning requirements into test cases, then improving the code to pass the tests.

It's a core practise of XP, and you can adopt it within other methodologies. In test-driven development, the test is written before implementation and drives the design of the API.

Benefits

There are a huge number of benefits to test-driven development that result in its popularity among software developers. These include the following:

- Build up library of small tests that protect against regression bugs*
- Extensive code coverage
    - No code without a test
    - No code that isn't required
- Almost completely eliminates debugging, which makes up for time spent developing tests
- Tests as developer documentation
- Confidence, not fear
    - Confidence in quality of the code
    - Confidence to refactor

*A regression bug is a defect which stops some bit of functionality working, after an event such as a code release, or refactoring.

Process and Strategy

Traditional development often involves creating tests almost as an afterthought, 'if there's time'. This often means that the product has a high number of defects, which leads to a lengthy testing phase after a release is frozen. But this costs more than just time – it's a lot more expensive to fix a bug discovered at this stage than one discovered when it's first introduced to the code. There's also the issue of maintainability: when you have a legacy code that 'works', it becomes a lot more difficult to make any changes without breaking the code.

Test-driven development defines a different approach that helps to solve some of these problems. There are four steps to TDD:

1. **Write a failing test.** Design the API for the code to be implemented. In tests, using an API is the best way to evaluate your design.
2. **Write just enough code to pass the test.** This minimises code bloat and keeps you focused on satisfying the requirement embodied in the test.
3. **Pass the test.**
4. **Refactor.** Change the code without changing the functionality.

In his article The Three Rules of TDD, 'Uncle Bob' Martin describes three rules of test-driven development that work in parallel to these guidelines:

You are not allowed to write any production code unless it is to make a failing unit test pass.

You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.

You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

Red-Green-Refactor

The four steps to TDD are also known as the 'Red-Green-Refactor workflow'. This workflow, illustrated in Figure 24, outlines the same way of working, but combines steps three and four into the 'green' finalisation section.

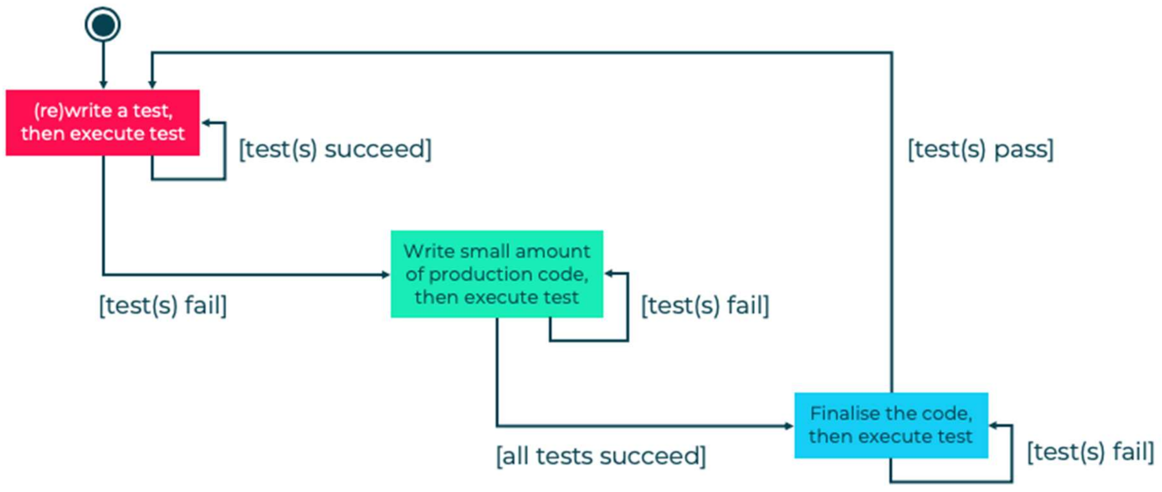

(re)write a test, then execute test
[test(s) succeed]
[test(s) pass]
Write small amount of production code, then execute test
[test(s) fail]
[test(s) fail]
Finalise the code, then execute test
[all tests succeed]
[test(s) fail]

Figure 24

TDD Strategies

In his book *Test-Driven Development by Example*, Kent Beck outlines three strategies for TDD that encourage developers to work in small steps, continually re-running their tests:

1. Fake it. Return a constant and gradually replace constants with variables.
2. Obvious implementation. If a quick, clean solution is obvious, type it in!
3. Triangulation. Locate a transmitter by taking bearings from two or more receiving stations. Only generalise your code when you have two or more different tests.

Triangulation (Figure 25) can be a little difficult to get your head around at first. Imagine that you have three people: one in a boat (A), one in a lighthouse (B) and one in a tower (C). The person on the boat can determine their position on a chart by taking compass bearings to the lighthouse and the tower. Or conversely, observers at B and C can work out the location of the boat by taking bearings to it from their known points on the shore and sharing their readings. In fact, sailors are taught to take a three-point fix, with charted landmarks that are as widely separated as possible, for better accuracy.

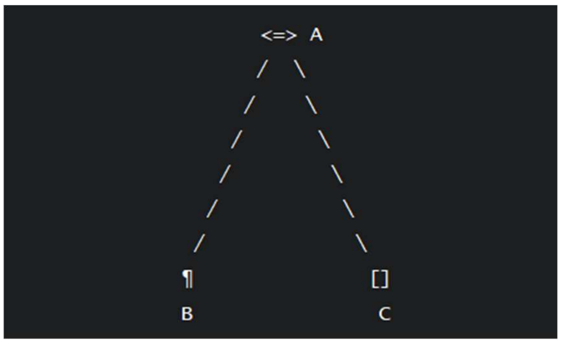


Figure 25

Testing Heuristics

'Heuristic' describes any practical approach to problem solving. It might not be perfect, but it's good enough to reach an immediate, short-term goal. In chapter 26 of *Test-Driven Development by Example*, 'Red Bar Patterns', Kent Beck suggests methods of breaking down the seemingly mountainous task of developing new functionality in a test-driven way into small, tractable steps. There are five of these methods:

1. **Test List:** write a list of all tests you know you have to write.
2. **Starter Test:** Start with the case where the output should be the same as the input.
3. **One Step Test:** Start with the test that'll teach you something and you're confident you and implement.
4. **Explanation Test:** ask for and give explanations in terms of tests. This method is primarily for communicating within the development group,

clarifying the requirements for some item of functionality by expressing them precisely in terms of tests.

5. **Learning Tests:** check your understanding of a new API by writing tests.

For example,

'a poster on the Extreme Programming newsgroup asked about how to write a polygon reducer test-first. The input is a mesh of polygons and the output is a mesh of polygons that describes precisely the same surface, but with the fewest possible polygons. "How can I test-drive this problem since getting a test to work requires reading Ph.D. theses?"'

In this case, you'd use the Starter Test method. The output should be the same as the input and some configurations of polygons are already normalised. You can't reduce them any further. The input should be as small as possible, i.e. a single polygon or an empty list of polygons.

Behaviour-Driven Development

Definition

Behaviour-driven development (BDD) is an agile software development process that encourages different parts of the business to collaborate on a project. It focuses on the stakeholders' perspective of the application and encouraging change:

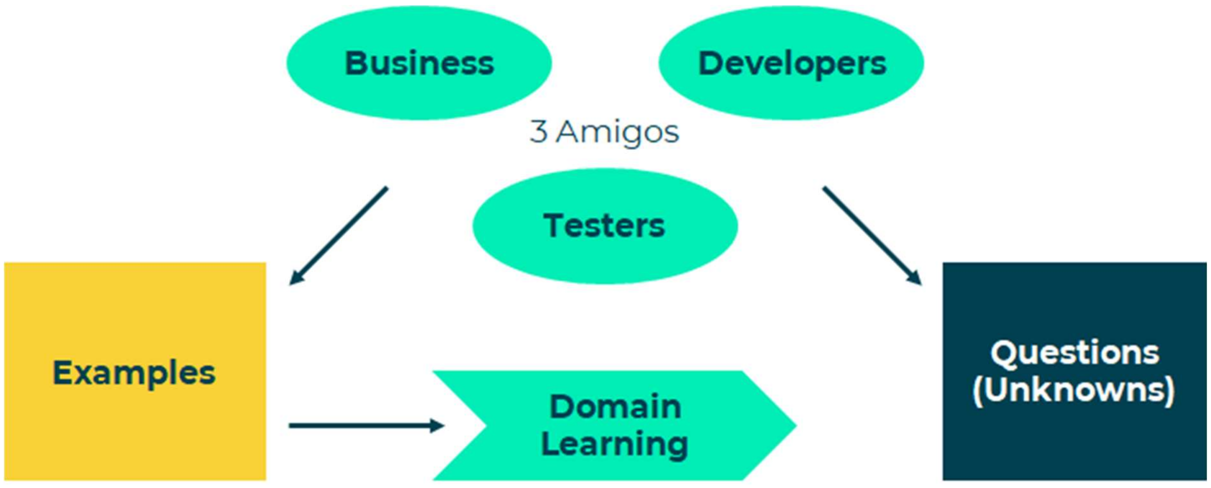


Figure 26

Involving the 'three amigos' (Figure 26) allows for the development of the key examples adding to the domain learning, and crucially the knowledge of the 'known unknowns'.

Because of the variety of people involved, BDD uses a language that all stakeholders can understand. Living documentation expresses the requirements as a set of **features**, and acceptance criteria as a set of **scenarios** (examples).

Each scenario provides an example to illustrate how the feature should work. Gherkin is a common language that you can use to define features and their scenarios – we'll talk about Gherkin and 'Given, When, Then' in more detail a little later in this chapter.

Process

There are five stages in the behaviour-driven development process:

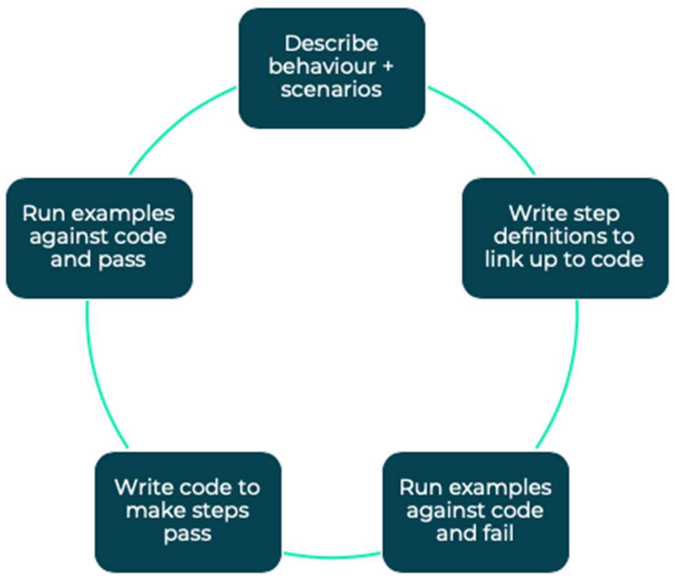

Describe behaviour + scenarios
Write step definitions to link up to code
Run examples against code and fail
Write code to make steps pass
Run examples against code and pass

Figure 27

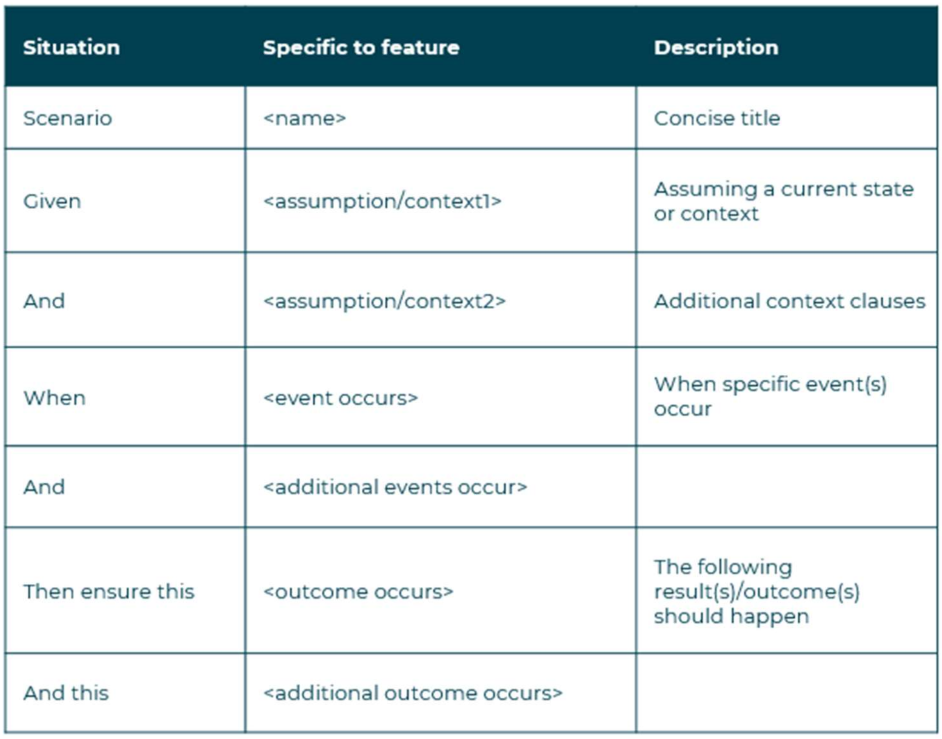
As illustrated in Figure 27, BDD is an iterative process that allows change. It begins with the definition of a requirement as a feature and the specification of the behaviours in terms of scenarios. This is written in a ubiquitous (universal, easy-to-understand) language like Gherkin, but is essentially a set of English sentences using a common set of keywords.

You can export each feature into a feature file, where you can use it to link up to the software you're developing. To do this, you write a specialist script called a **step definition**. The step definition references the features and scenarios described in the feature file and then hooks into the actual software that is being developed. Here, it performs relevant tests based on whatever the scenario requires.

All of this happens before the software is even written so, at first, all of the tests fail! But as you write more of the code, more scenarios pass until, eventually, the whole story passes.

Defining Scenarios

Scenarios (or examples) allow us to illustrate all the possible things that could happen and the circumstances in which they could happen. Each scenario takes the form of a sequential number followed by a name. We usually write them using the 'Given, When, Then' format, though there are further clauses that you can include if you need to (Figure 28). **Given** indicates the current state or context of the domain **When** one or more events occur, **Then** a set of results or outcomes are expected to happen.



Situation	Specific to feature	Description
Scenario	<name>	Concise title
Given	<assumption/context1>	Assuming a current state or context
And	<assumption/context2>	Additional context clauses
When	<event occurs>	When specific event(s) occur
And	<additional events occur>	
Then ensure this	<outcome occurs>	The following result(s)/outcome(s) should happen
And this	<additional outcome occurs>	

Figure 28

Scenarios are relevant to actors and stakeholders, and each scenario can be traced through to one or more tests. You can use multiple scenarios to illustrate how a feature should respond in differing circumstances.

It's really important to use a common language to define your scenarios so that they can be linked directly to automated testing via a bit of clever development cross wiring. In essence, the behaviour definition is driving the development, so it's important that everyone understands the scenario definitions.

Writing Scenarios

When writing a scenario, it's best to start with the 'happy' or 'all is well' path:

Given my shopping basket is empty

When I add the book 'BDD is Fun' to the basket

Then the shopping basket contains 1 copy of 'BDD is Fun'

Once you've done this, you can add in 'edge' scenarios to demonstrate the variations of behaviour:

Given my shopping basket contains the book 'BDD is Fun'

When I add another copy of book ‘BDD is Fun’ to the basket

Then the shopping basket contains 2 copies of ‘BDD is Fun’

Best Practice

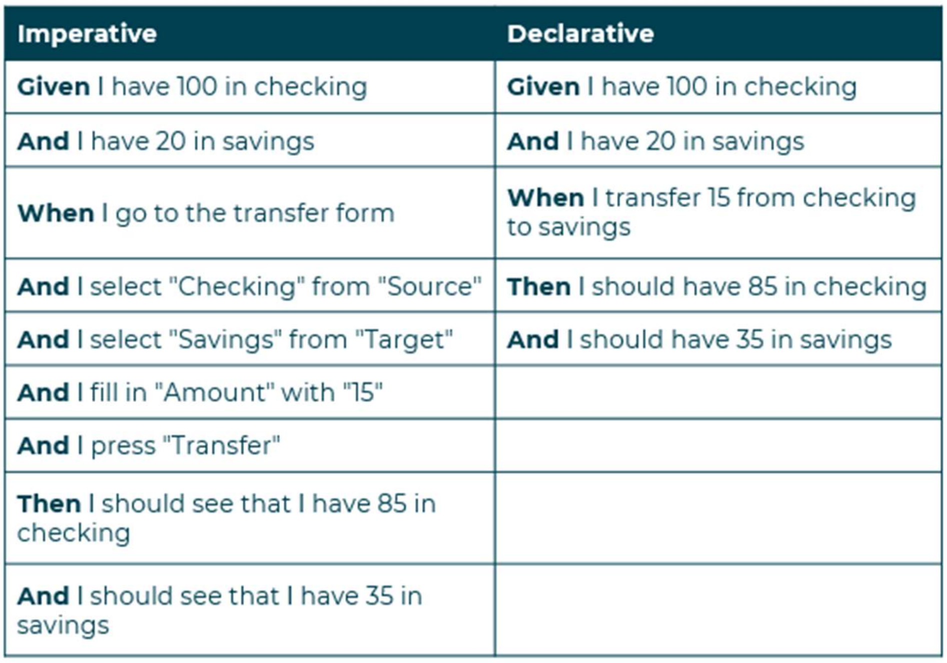
When writing scenarios, there are some golden rules that you need to follow:

- Write in present tense
- Always use business language
- Express the clauses as readable sentences
- Use only the situations shown in Figure 28 - ‘or’ doesn’t exist!
- Keep to one feature per story, and keep to max. 12 scenarios per feature
- Capitalise Gherkin keywords Given, When, Then, etc.
- Capitalise all titles

You’ll often see scenarios written out using one of two styles: **imperative** and **declarative** (Figure 29).

Imperative is usually more prescriptive, longer and more minutely detailed. It takes less for granted. This style forces developers working with the scenario to write link up code at a much lower level of granularity. This can be beneficial because it means the scenario writers can write many variations on the same scenarios to test different kinds of conditions. It's often also easier to work in new scenarios without a lot of extra development effort.

The declarative style is less prescriptive and allows the developers to write specific links to individual scenarios. This gives less flexibility to the scenario writer.



Imperative	Declarative
Given I have 100 in checking	**Given** I have 100 in checking
And I have 20 in savings	**And** I have 20 in savings
When I go to the transfer form	**When** I transfer 15 from checking to savings
And I select "Checking" from "Source"	**Then** I should have 85 in checking
And I select "Savings" from "Target"	**And** I should have 35 in savings
And I fill in "Amount" with "15"	
And I press "Transfer"	
Then I should see that I have 85 in checking	
And I should see that I have 35 in savings	

Figure 29

Using Backgrounds

You can use Background to avoid repeating the same clauses continually. But be careful not to use Background to set up complicated states, unless that state is actually something the client needs to know.

For example, if the user and site names don't matter to the client, use a higher-level step such as 'Given I am logged in as a site owner'.

Keep your Background section short - your client needs to actually remember this stuff when reading the scenarios! If the Background is more than 4 lines long, consider moving some of the irrelevant details into higher-level steps. If the Background section has scrolled off the screen, the reader no longer has a full overview of what's happening. Think about using higher-level steps, or splitting the *.feature file.

Because it needs to be memorable, you should make your Background section vivid. Use colourful names and try to tell a story. The human brain keeps track of stories much better than it keeps track of names like 'User A', 'User B', 'Site 1', and so on.

For example:

> **Given** the user has entered a valid pin
>
> **And** the account is open
>
> Scenario: User transfers cash from Savings to Checking
>
> **Given** I have 100 in checking
>
> **And** I have 20 in savings

Validating Scenarios

When you're ready to validate your scenario, ask yourself these five questions:

1. What is the intent?
2. Have we understood the expected behaviour?
3. Is the expected behaviour clearly expressed?
4. Can the rule be clearly understood when reading the scenario, and is there enough detail?
5. How many rules are defined in the scenario? Remember, it should ideally just be one!

Examples

Scenario: Account is in credit

> **Given** the account is in credit

And the card is valid

And the dispenser contains cash

When the customer requests cash within their balance

Then the account is debited

And cash is dispensed

And the card is returned

You can include data to be more illustrative:

Given the customer has 100 in checking

And the customer has 20 in savings

When 15 is transferred from checking to savings

Then there is 85 in checking

And there is 35 in savings

You can also include data in the form of a table:

Given the input ‘<input>’

When the calculator is run

Then the output should be ‘<output>’

input	output
2 + 2	4
98 + 1	99
255 + 390	645

Writing User Stories

User stories are your software system requirements, written from the perspective of your user. They usually contain a definition and your acceptance criteria.

Definitions typically use the ‘Connextra’ format: **As an X, I want Y, So that Z.** Here, X is a person, ‘persona’ or role, Y is an action to be performed or a feature, and Z is a benefit.

Acceptance criteria are conditions that have to be satisfied for the story to conclude successfully. They help create a definition of done (I.e. what the ‘done’ or finished product will look like), and they make the story testable. The acceptance criteria have to be met as specified by the product owner, stakeholders, designers / architects and the development team.

Requirements and acceptance criteria written in a pure narrative form can be ambiguous. However, you can use examples of how a process should work based on different scenarios to make things clearer. Examples are often much more illustrative and cover many more likely eventualities than could ever be communicated using a raw textual description:

- 'In the best case scenario, this is how it should work…'
- 'It could also work like this…'
- 'Even this could happen! This is what the response should be…'

Examples should illustrate what could happen, under what circumstances, and what should be done when it happens.

Cucumber and Gerkin

Cucumber

According to its own website, Cucumber is…

' a tool that supports Behaviour-Driven Development (BDD).'

Cucumber can be used in almost any development language. It reads executable specifications written in plain text and validates that the software does what those specifications say. The specifications consist of multiple examples, or scenarios. For example:

Scenario: Breaker guesses a word

Given the Maker has chosen a word

When the Breaker makes a guess

Then the Maker is asked to score

Each scenario is a list of steps for Cucumber to work through. Cucumber verifies that the software conforms with the specification and generates a report indicating success or failure for each scenario.

Gherkin

In order for Cucumber to understand the scenarios, they must follow some basic syntax rules, called **Gherkin**. You can download and install Cucumber from the installation page on their website. The downloads are rated as follows:

- Official - hosted by Cucumber
- Semi-official - hosted elsewhere but uses components from Cucumber

- Unofficial - hosted elsewhere and doesn't use any components from Cucumber
- Unmaintained - official implementations but unmaintained

Cucumber works by breaking scenarios down into step definitions, which can then be put into actual code, as follows:

> Given the balance is 500
>
> When the user withdraws 200
>
> Then the balance is 300

```
@Given("the balance is $bal")

public void setBalance(double bal) {

   myCal=new Calculator(bal);

}

@When("the user withdraws $amount")

public void withdrawAmount(double amount) {

   myCal.Withdraw(amount);

}

@Then(" the balance is $bal")

public void testResult(double bal) {

   Assert.assertEquals(bal, myCal.getBalance());

}
```

A feature broken down into its constituent scenarios is linked to step definitions that are typically written in a programming language such as Python, Ruby, JavaScript, Java or C#. The step definitions may use a driver library that can talk to a web browser or a mobile phone simulator and simulate touches, mouse clicks or key presses. They can just as easily link directly to code as unit tests or test of web services or micro services.

In the example above, the step definition is using a bank account class and checking that the balance that is being used to initialise it is in fact setting the balance property of the account.

Notice how the value 500 is being passed in from the scenario to the step definition `step_set_balance` as a parameter called `newBal`.

There are intermediary steps between Gherkin scenarios and code or user interfaces via a UI automation api. Many development environments provide utilities to generate step definitions from the Gherkin script. If you use parameters carefully, you can reuse your step definitions:

```
@When("the user withdraws $amount")

public void withdrawAmount(double amount) {

   myCal.Withdraw(amount);

}

// becomes @When("the user $action $amount")

public void withdrawAmount(string action, double amount) {

   if (action == "withdraws"){

      myCal.Withdraw(amount);

   } ...

}
```

You can execute a Gherkin script either from the command line, using a Cucumber engine, or from within the development environment as unit tests (if the IDE supports it).

Applying Gherkin to Acceptance Criteria

Let's take a look at an example of this. Say your customer requirement is 'Books can be added to the cart', Gherkin would describe the criteria more formally as:

> **Given** my shopping basket is empty...
>
> **When** I add the book 'BDD is Fun' to the basket...
>
> **Then** the shopping basket should contain 1 copy of 'BDD is Fun'

Gherkin uses the following keywords to define examples:

- Feature
- Background
- Scenario
- Given
- When
- Then
- And
- But

These work in the same way as we discussed in the previous section on writing scenarios.

Automation

Developers can use a **Cucumber** engine to run Gherkin. This engine may link to a framework to talk to browsers, simulators etc.

Mobile developers use tools like **Appium**, and web developers can use **Selenium** to hook into browsers. Step definitions run commands via Selenium Api.

Code Smells

Coding isn't always straightforward, but there are some basic principles that you can follow, as well as a few things to avoid, which will help you set off on the right foot every time.

Common Code Smells

Long methods make code hard to maintain and debug – consider breaking your code up into smaller methods.

Refuse request, which is when a class inherits from a base class and doesn't use any of the inherited methods.

Data clumps, which occur when multiple method calls take the same parameters.

Duplicate code, which can mean that you fix a bug, only for the same bug to then resurface somewhere else in the code!

Other common code smells include middle man and primitive obsession. For more details on these, we recommend that you read this article by Georgina McFayden.

SOLID Principles

The SOLID principles are part of Robert 'Uncle Bob' Martin's wider set of suggested software design principles. They are usually applied to object oriented code design, and are as follows:

S - Single Responsibility Principle

- A class should only have a single responsibility
- There should be only one trigger to mean a change in the class is necessary

O - Open-Closed Principle

- Code should be open for extension but closed for modification

L - Liskov Substitution Principle (aka design by contract)

- 'Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program'
- In other words, you must be able to substitute a base class with any class that extends it

I - Interface Segregation Principle

- Interfaces shouldn’t be made generic; it’s better to have specific interfaces specific to the client that will implement them

D - Dependency Inversion Principle

- ‘Depend upon abstractions, not concretions’

Coding Standards and Practices

The Three Virtues

The three virtues of a programmer listed below come from Larry Wall, the original author of the Perl programming language. Of course, they're tongue-in-cheek... true laziness requires hard work! These virtues all have serious points, and are not confined to Perl.

Laziness

Laziness is the virtue that makes a programmer hate to code the same thing twice, which is why reusable code is born:

- Programmers hate answering the same questions over and over, so write good documentation
- Programmers hate reading documentation, so write code clearly
- Programmers write tools and utilities to make the computer do all the work

Now swap out 'laziness', which has negative connotations, for 'efficiency'. Reusing code between programs and projects not only saves money in development time, it also reduces the amount of testing to be done and, ultimately, results in more reliable programs.

Impatience

The impatient programmer gets the computer to do the work. So, rather than expecting the user to copy items between applications (which is slow and error-prone), the impatient programmer writes a program to do it. If there is no tool or utility to do what you need, then write one (that's how Perl came about).

- Programmers hate a computer that is lazy - an impatient programmer's code anticipates a need

Hubris

Hubris, or excessive pride, is often considered a sin. What is meant here is that the author takes personal responsibility for their own program code. Hubris means we care about what we do.

- Programmers have pride in programs that no one will criticise

The rest of this chapter is about how we achieve these virtues, in particular on looking ahead beyond the initial development of a program and considering what will happen when we release it into the wild.

Good Practice

The golden rule of programming is to always follow the standards of your organisation. Ask to see the coding standards or guidelines, then make sure that your code always uses consistent naming, effective commenting and proper formatting.

It's important that you follow the coding standards defined by your organisation above all others mentioned in this chapter.

So now we've covered the golden rule, let's take a look at some further good practices…

1. Remember 'Rubbish in – rubbish out'. If you let bad data in, you'll get bad data out.
2. Choose the smallest data type for the job. You don't need a 64-bit integer to store a person's age!
3. Always assign and operate on like data types.
4. Remember floating point issues.
5. Use constants, not literal numbers, where possible.
6. Create variables with the shortest scope and lifetime.
7. Make sure that objects are allocated and available for release as soon as possible.
8. Use variables for one purpose only.
9. Make sure that functions only perform one task. To help with this, check the data passed into a function is valid, and don't make functions dependent on global variables.
10. Make sure that classes have a single responsibility, and identify what that is. To help with this, avoid making classes do things they shouldn't be responsible for and, if possible, avoid making a class dependent on other concrete classes.
11. Automate your tests!

Naming Conventions

Using a naming convention for your own identifiers has so many advantages that just about every set of programming standards mentions them. It is part of your job to follow the company conventions - even if you disagree with them. Your program code is not the place to argue about conventions.

A naming convention will help avoid using a name which the language, or a 3rd-party library, already uses – these are called name collisions. It also helps the names stand-out against the background noise of program code. Naming conventions can indicate properties of the thing being named.

In order for names to be descriptive, they often have to consist of several words. How do we separate those words when white-space is not allowed within an identifier? There are several options. Some old languages (and old versions of existing languages) have limits as low as eight characters for identifiers. In these cases, many programmers use names consisting of initials, e.g. AFAIK. Most modern languages have no effective limit on the size of an identifier, or that limit is so large that it is unlikely to be exceeded. Longer variable names have no effect on program performance - especially if the code is compiled or minimised for production.

Camel case is where the first letter of each word is in uppercase, forming a visual boundary between words without the need for a space character. Opinions vary on whether the first letter should be upper or lower case; when upper case is used, it's often called PascalCase:

```
lowerCamelCase

UpperCamelCase (PascalCase)
```

Some people argue that CamelCase is difficult to read, so an alternative is to separate words using an underscore. A hyphen isn't normally allowed because it can be confused with a minus sign, as in x-axis (x minus axis?). Using lower case names with underscores is popular in C and UNIX:

```
all_lower_case
```

You should also avoid all UPPERCASE, as these are commonly used for macro names.

It doesn't really matter which style you use, so long as you are consistent. Reading program code is not the easiest task in the world, and having inconsistent conventions makes it harder. Remember that, if you're modifying someone else's program, you should use their original style as much as possible.

Single character names are acceptable for temporary variables. For example, i, j, k, and l are often used for loop counts, although lower-case letter l (ell) can be difficult to distinguish from the number 1 (one), depending on the font, so is probably best avoided, as is the letter O (oh).

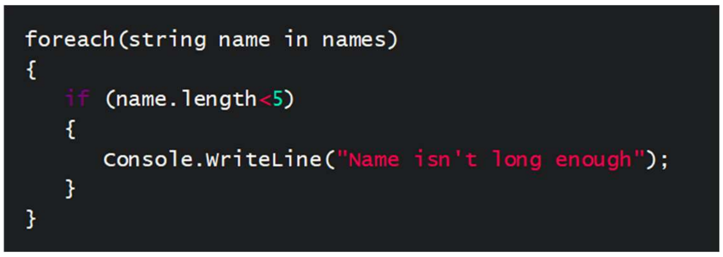
Comments

Comments are essential in your code and are expected, but don't overdo it. Comments are not an excuse for poor code!

- Always comment the intent of your code
- Don't comment self-evident coding structures
- Make sure you comment work-arounds and quick fixes – ideally, rewrite them to be optimal
- Always update comments when you update code
- Remove comments from scripts before release

Formatting

Good formatting makes code clearer. Format your code to allow it to flow naturally. Proper indentation makes it easier to identify code blocks, but some languages use other formats. For example, Python uses indentation, but C-based languages use braces. If using braces, you should allow each brace to sit on its own line and use indentation (Figure 30). You should always use the convention for the language you're writing in.



```
foreach(string name in names)
{
   if (name.length<5)
   {
      Console.WriteLine("Name isn't long enough");
   }
}
```

Figure 30

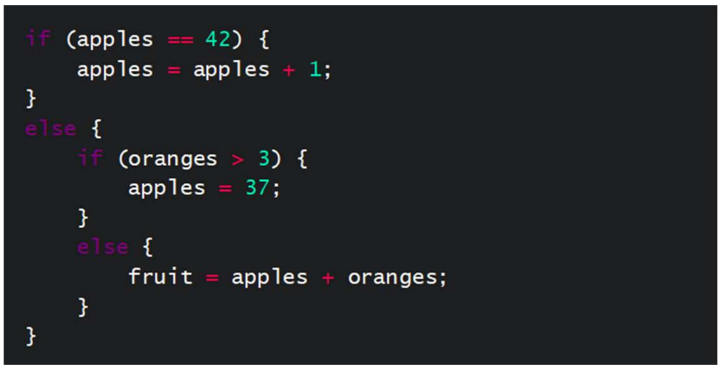
Readability and Style

Generally, more time is spent on maintenance than development, so make sure you document what you do. Code that is obvious today is not obvious tomorrow.

You should avoid 'clever' one-liners – they're rarely faster, and can sometimes actually be slower. On top of this, they're often difficult to debug.

Remember, developers and programmers like to **KISS** - Keep It Simple and Straightforward.

It is easy to spot a program written by an experienced programmer. It will be easy to read, well commented, laid out with plenty of white-space, and use consistent indentation. By 'indentation', we mean placing spaces inside blocks of code, as illustrated in Figure 31.



```
if (apples == 42) {
    apples = apples + 1;
}
else {
    if (oranges > 3) {
        apples = 37;
    }
    else {
        fruit = apples + oranges;
    }
}
```

Figure 31

How many spaces you should use for indentation is something that programmers argue over. Four is a good number, but if everyone else uses three, five, or six, then go along with them. Don't use tabs though - the size of a tab varies between tools and users, and is not portable.

A program written by a novice is difficult to read, with no indentation, no white-space, and no comments. Try to make your programs easy to read and easy to modify. It might be you who has to implement a change six months after the code was written, but it might be someone else. If it's someone else, then you can feel pride in your code - hubris.

The Elements of Programming Style by Brian Kernighan and P.J.Plauger uses Fortran and PL/1, but is applicable to most languages (with a few exceptions).

Error Handling

The key with error handling is to remember that if anything can go wrong, it will. Because of this, you should specifically test error conditions.

- If an opportunity exists to test for an error - take it!
    - When calling a library routine
    - When getting any data from the outside
- Always report the error to an expected location
    - Write errors to an error stream, not the normal output stream
    - Users probably don't need the full story, but make sure it's available to those that do
- It's a common hacker's trick to cause a program to fail
    - Can result in the display of sensitive data, or the opening of a backdoor

Even if your program code is perfect and never does anything wrong, you still need to handle errors which occur elsewhere. What if the user enters an invalid value? What if the file you are trying to use doesn't exist, or you don't have permissions to use it? What if a library routine you are calling gets an error?

For the majority of programmers, their least favourite chore is testing and handling error conditions. Often there are many ways for something to fail, resulting in exception code that can dwarf the core program logic.

One of the traditional approaches to signalling errors is to return some kind of status value from a function. There are a number of problems with this: it can lead to cascading if-else-if code that can be hard to read, and return values are too easy to ignore.

Aborting the program is a last resort. It's up to the caller to determine what the failure policy should be, and for the called component to detect any failure.

The solution is to **represent exceptions as objects**. As objects, they can be self-describing, whereas something like a simple integer value (as used in some languages) isn't very informative as it says nothing about the details of an exception. Exceptions can be caught by defining a handler, but how you do that, and how you raise and catch exceptions, is language specific.

Get the computer to collect all the diagnostic information available, particularly if you are lazy and impatient.

Programming for Change

Defensive programming is a style of programming intended to reduce the possibility of mistakes during the maintenance phase of a program's life-cycle - remember, this phase will be much longer than the development phase. That maintenance is often carried out by people other than the original developer.

Defensive programming is a technique which requires some thought and a rather pessimistic outlook. The motto is: **never make assumptions**.

Coding using a clear, easy to understand, style is the foundation of defensive programming, but it doesn’t stop there. When testing a number of alternative values, always include a test at the end if it matches none of those - even if, logically, if can't be anything else. At the very least you would have caught an error.

Using well named constant or read-only variables to hold numbers is good, but they must be defined in a ‘well-known' or obvious location in the code. Of course, documentation helps as well.

Artificial limits are often imposed when you create a fixed length data item. For example, how much memory should be allocated for someone's name? 24 bytes? 1024 bytes? If you don’t allocate enough, then data may be lost. On the other hand, allocate too much and you waste system resources. This is where dynamic memory allocation becomes useful - decide at run-time how much memory to use. The down-side is often a performance penalty.

Help and Support

If your organisation has its own standards then you should always follow those in preference to any other guidelines - including any we have mentioned in this course! Style is subjective. However, most programming languages offer advice on programming style:

- Python has PEP 008 rules
- Perl has perlstyle documentation and PerlCritic
- Java has Sun's code conventions
- Google Style Guides are available for several languages

Programming is clearly hard work. So what does a **lazy** programmer do? Write a program which imposes style or spots bad practice. There are several, many are based upon an old UNIX/C program called 'lint', which looked explicitly for bad coding style. Modern versions are much more sophisticated and can spot some types of bugs as well. Few go as far as Perl's perlcritic program (and website) which offers an opinion of your code at different levels of criticism: gentle, stern, harsh, cruel, or brutal! There is a serious point here – it's an easy trap to feel over-protective towards your own programs. It is important to listen to, and sometimes to seek, criticism in order to grow your skills. Only then can you attain the virtue of **hubris**.

Many integrated development environments, and even simple editors, encourage good style by automatically indenting, giving hints, and so on. They are only helper tools, though – it's still your responsibility to write clearly.

Continuous Integration

Definition

Continuous integration is the practice of building software in medium-large sized teams where team members integrate their work frequently and run as many tests as possible. Martin Fowler, author of the article 'Continuous Integration', defines it as:

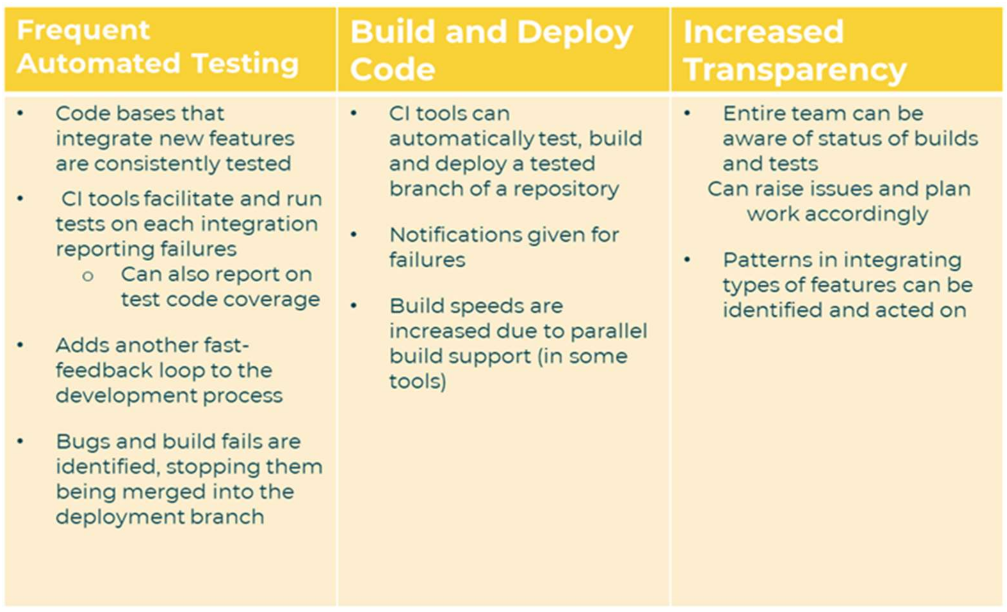
'a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly.'

How

The how is quite simple. Continuous integration works as developers frequently introduce new features to a piece of software being built. Code is regularly and automatically added to the base code. As such, test-driven development methodologies are used to verify that the new features work; this is the safest method as it runs a very low risk of breaking any code. Ultimately, continuous integration automates the building process.

Why

There are three key reasons as to why continuous integration proves to be effective. Figure 32 below illustrates these reasons.



Frequent Automated Testing	Build and Deploy Code	Increased Transparency
• Code bases that integrate new features are consistently tested • CI tools facilitate and run tests on each integration reporting failures ○ Can also report on test code coverage • Adds another fast-feedback loop to the development process • Bugs and build fails are identified, stopping them being merged into the deployment branch	• CI tools can automatically test, build and deploy a tested branch of a repository • Notifications given for failures • Build speeds are increased due to parallel build support (in some tools)	• Entire team can be aware of status of builds and tests Can raise issues and plan work accordingly • Patterns in integrating types of features can be identified and acted on

Figure 32

Best Practices

There are two main best practices to keep in mind when implementing continuous integration.

Maintaining a single source repository

All integrated code should be kept within an up-to-date, secure repository that can be updated and added to. It should be easy to access for those who are integrating and it should also contain everything required for the build, such as tests and compilation scripts. This means no more emailing code around!

Everyone commits to the mainline every day

When each change is integrated and tested, the change must be committed to.

Securing Jenkins

In Jenkins 2.0+, security is enabled by default. Although, you can manage users from the 'Manage Jenkins' menu. Before this update, Jenkins allowed anyone to create and build new jobs. For security purposes, this was very impractical. It ultimately would have allowed anyone with access to the internet to use a seemingly private build server!

Access Control

When you first access Jenkins, like any good open-source automation server, it will ask you to set an initial username and password. However, it doesn't - and shouldn't - stop there! Jenkins allows you to hook onto existing systems such as Unix user groups. It also enabled Lightweight Directory Access Protocol (LDAP) so existing Windows logins can be used in addition to having its own user database, making authorised access easy but very secure. Logged in users have overarching access but project-based security can be enabled if required.

It's important to note that Jenkins needs to run as root or User must belong to group root and chmod g+r/etc/shadow needs to be done in order to enable Jenkins to read/etc/shadow.

CI Tools

- **Jenkins**: A free, open-sourced tool with an active community
- **Circleci**: Free tiers available, charged options for volume users
- **TeamCity:** Maintained by the JetBrains team, has free and paid-for services
- **GitLab:** Can be configured to use their services or of your own choice (in-house or cloud-based), has free and paid-for services
- **TravisCI:** Free for open-source projects, private projects are chargeable

Jenkins Concepts

Introduction

Jenkins is a CI tool that was first released in 2005 as the 'Hudson Project'. It was created by Sun Microsystems and bought out by Oracle, who changed the name to Jenkins for the open source-project. Both the Hudson and Jenkins project are still active but development has deviated.

The aim of the Jenkins tool was to be easily accessible, easy to install and easy to use. Other focuses were:

- Plugins to connect to source control
- Builds projects using the project setup (works with any language)
- Public / company facing dashboard lets everyone know the current build status

Purpose

Jenkins is ultimately a comprehensive solution for testing and building code. It can even deploy code onto servers (providing they already exist!). Figure 33 below shows a typical Jenkins workflow.

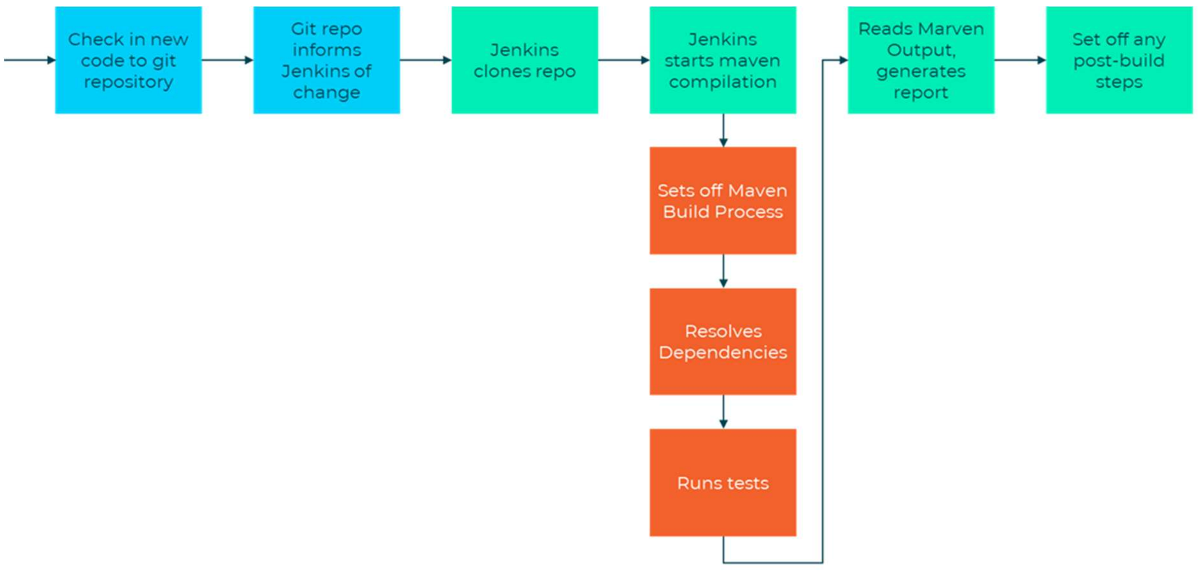

Check in new code to git repository
Git repo informs Jenkins of change
Jenkins clones repo
Jenkins starts maven compilation
Reads Marven Output, generates report
Set off any post-build steps
Sets off Maven Build Process
Resolves Dependencies
Runs tests

Figure 33

The Jenkins workflow allows companies to follow best practices for continuous integration. This means that the build is automated, self-testing, fast, deployed automatically and gives you transparent results.

It also means that installation and configuration are easy because you can do these through the GUI, with no need for XML files. Additionally, many plugins are available for any combination of source control, compiling, testing and deployment.

Distributed Jenkins

The way Jenkins runs builds is quite simple. It uses different agents, or threads, on a singular machine. By default, Jenkins has two threads available for you to use. However, whilst this works with small-medium scale projects, larger projects may need more capacity. This is because a singular machine will eventually run out of memory / CPU capacity. So, Jenkins can run as a distributed system with a single master controlling one or more nodes. It can build a project on multiple operating systems or architectures.

It's also important to look at the different ways in which Jenkins allows you to manage nodes, it can:

- Launch node via SSH (usually on unix systems)
- Control Windows as a service
- Launch via Java Web Start
- Write our own script to launch it

Advantages to Distributed Builds

There are three main advantages to using distributed builds:

1. One server can handle multiple projects.
    - Single point to see how the builds are going
2. You can specify jobs to target specific machines.
    - You may want one job to build on Windows only
    - Can identify specific nodes by name to use
    - Alternatively, you can leave the pool open and the master will select which node to use
3. Not a lot of overhead for setting up.
    - Jenkins does not need to be installed on the node
    - Can pull more online or take them offline easily
    - Jenkins can automatically install all the tools it needs

Scheduling and Node Monitoring

Jenkins will always follow certain rules when deciding on which node to use. These rules are:

1. Use a machine the project is configured to stick to.
2. Use the same computer that the project was built on before.
3. Long builds are moved to nodes.

The master can monitor the health of whichever node is used. It can do so by overseeing if builds over many projects start to fail on this node. It can also check disk space and response time and can pull nodes offline if there's a problem. It's also worth noting that plugins can add more monitors.

Setting up a Distributed Build

Setting up the Master

From the main dashboard you can manage Jenkins and manage nodes. You can also add a new node, though your first will always be a permanent node. This configuration can then be copied by others, if need be. You can also set the number of executor threads. Always specify a directory for Jenkins to use on the node, i.e:

/home/ec2-user/jenkins

C:\jenkins

When setting up the master, you can also decide how much you want to use the node and how much you want to control the node, using 'Usage' and 'Launch Method'.

Note that Java web start is easiest for windows hosts.

Windows Nodes

You can run the node program via Java Web Start or through installing a service. After setting up the node on the master, go to the link given by the master on the remote machine. Then, click the Java Web Start button.

If you already have Jenkins installed as a service, you can connect directly to the machine using a username and password. However, this isn't recommended. This way, you're prone to bugs. As such, you're better off using the Java Web start.

Distributed Jenkins Servers

Jenkins Servers uses agents to scale systems. By default, Jenkins will use one agent which will then build by using one core, as Jenkins architecture is typically master and agent. However, it may need to build for different architectures in order to scale appropriately. For instance, it may need to build for:

- Windows / Mac / Linux
- Android / IOS / Windows Phone

Other qualities attributed to Jenkins master / node servers are that they can exist anywhere and that a single report can be generated over all the builds. The master can monitor the health of any connected node.

Jenkins Best Practices

Jenkins have their own set of best practices listed on their website. However, as a starting point, you should...

1. Always secure Jenkins. Don’t leave your server open to abuse by others, and at least enable Jenkins’ own user database.
2. Backup Jenkins’ regularly. We have the scm sync mod doing this for us now!
3. Use ‘file fingerprinting’ to manage dependencies. Keep track of which version of each project is used by what.
4. The most reliable builds will be clean builds, which are built fully from Source Code Control. Enable the build to be pulled directly from git, and don’t include pre-compiled files if you can get away with it.
5. Integrate tightly with your issue tracking system, like JIRA or bugzilla, to reduce the need for maintaining a change log.
6. Integrate tightly with a repository browsing tool like FishEye if you’re using Subversion as source code management tool. Subversion is slower than Git. This allows Jenkins to quickly work out what has changed since the previous build.
7. Configure your job to generate trend reports and automated testing when running a Java build. Maven will run tests automatically if you use the package command.
8. Set up Jenkins on the partition that has the most free disk-space. Jenkins can keep all the settings, logs and builds in the JENKINS_HOME directory for use later.
9. Archive unused jobs before removing them - you can always resurrect a job, but you can’t undelete it.
10. Setup a different job / project for each maintenance or development branch you create. This will help you detect problems quickly when developing in parallel. Alternatively, you can use the matrix project.
11. Allocate a different port for parallel project builds and avoid scheduling all jobs to start at the same time.
12. Set up email notifications mapping to all developers in the project, so that everyone on the team has their pulse on the project's current status.
13. Report failures as soon as possible.
14. Write jobs for your maintenance tasks, such as cleanup operations, to avoid full disk problems. You can create cron jobs to deal with some of these bits.
15. Tag, label, or baseline the codebase after the successful build so that everyone knows which version they are looking at or demonstrating.
16. Configure Jenkins bootstrapper to update your working copy prior to running the build goal / target.
17. In larger systems, don't build on the master - we have other nodes for this. Set the executor count to zero on the master.

Continuous Delivery and Deployment

Definition

Continuous delivery (CD) is a software engineering approach associated with DevOps that encourages teams to produce software in short cycles. This makes sure that you can release software reliably at any given time. The aim of CD is to build, test and release software at a faster, continual pace (Figure 34). Jez Humble, author of *Continuous Delivery* and co-author of *The DevOps Handbook*, describes it as:

'The ability to get changes - features, configuration changes, bug fixes, experiments - into production or into the hands of users safely and quickly in a sustainable way'

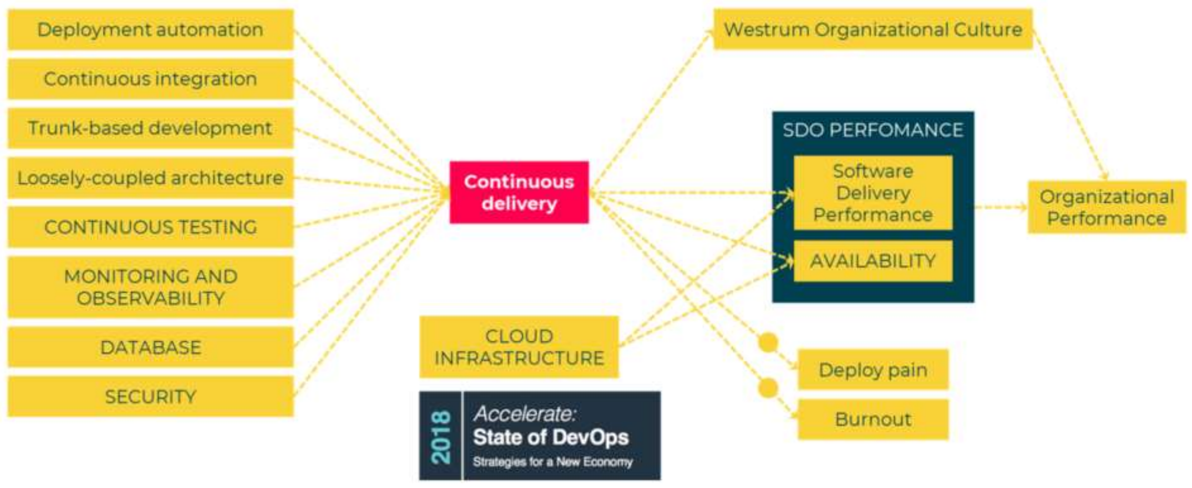


Figure 34

Continuous Deployment

Continuous deployment means much the same thing. Both focus on:

- Successful, repeatable deployment of code
- Deployment to server after the build has run and been verified
- Automation of the server setup / connection
- Minute by minute code releases, rather than months at a time

However, with continuous deployment, the trigger for production is **manual**, as illustrated in Figure 35 below.

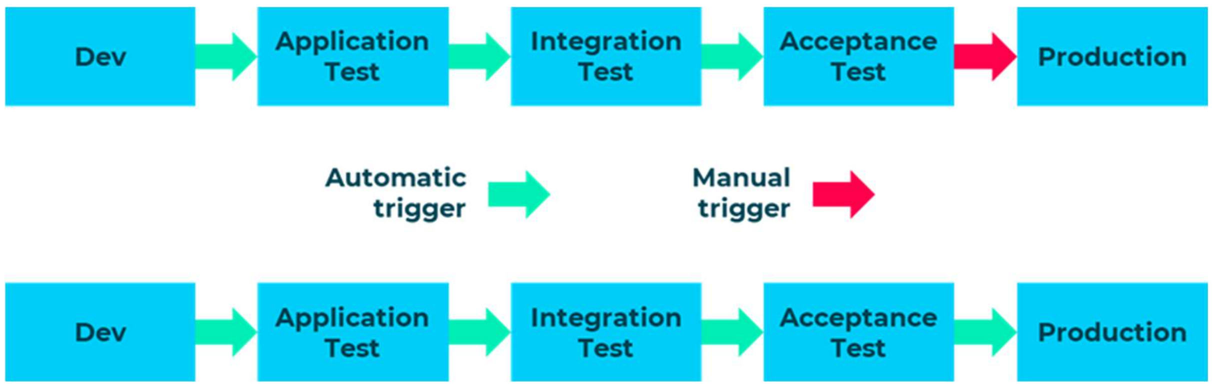


Figure 35

Three Tiers of Culture

There are three tiers of culture in relation to CD. These are:

Enterprise Executive

Executive leadership culture affects the enterprise's strategic approaches. Ultimately, it impacts the decision making processes surrounding continuous delivery.

Cross-Functional Teams

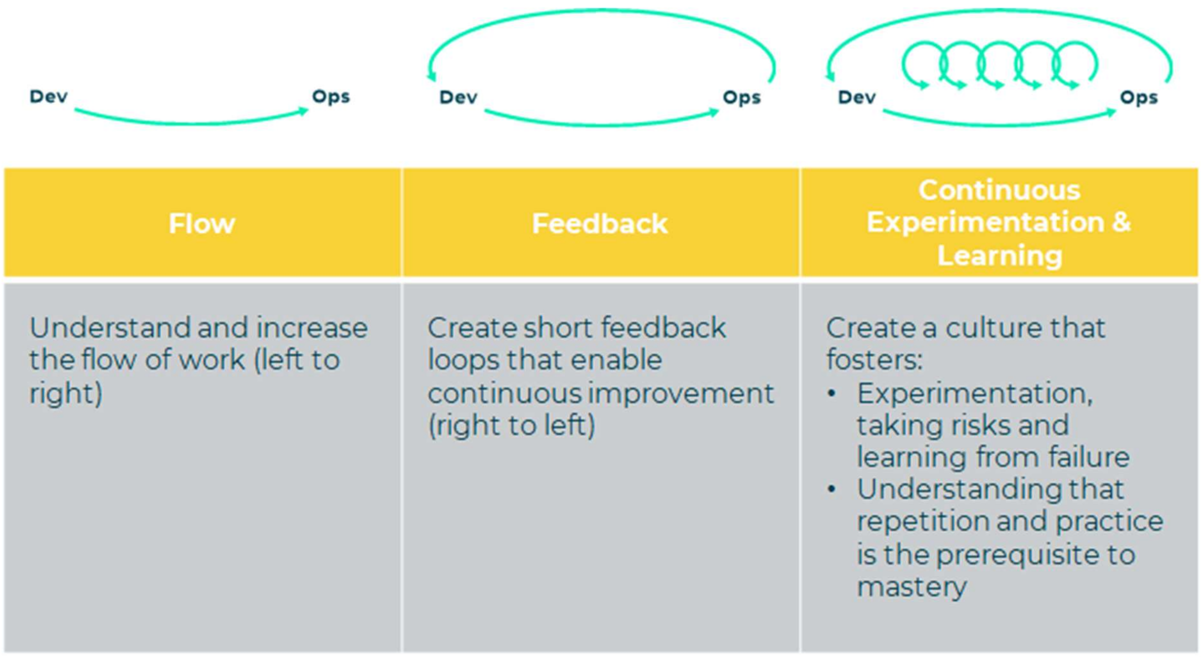
The culture and behaviour of cross-functional teams affects the choices made by teams. Overall, it impacts workflows and practices that are critical to continuous delivery.

Design Micro-Teams

The culture and behaviour of smaller teams affects workflows and practices that are critical to any design process suitable for continuous delivery.

Three Methods of Continuous Delivery

To implement continuous delivery, it's important to consider the three ways in which this can be done. Figure 36 below illustrates the three ways in which you can implement continuous delivery.



Flow	Feedback	Continuous Experimentation & Learning
Understand and increase the flow of work (left to right)	Create short feedback loops that enable continuous improvement (right to left)	Create a culture that fosters: • Experimentation, taking risks and learning from failure • Understanding that repetition and practice is the prerequisite to mastery

Figure 36

DevOps

DevOps is a key asset to continuous delivery. Development and operations can often seem divided, despite having a common goal: effective production. This division means that communication between the two can become confusing and unreliable. DevOps is a collaborative effort between the two parts of an organisation, and it helps pave the way for a smooth rollout (Figure 37 and 38).

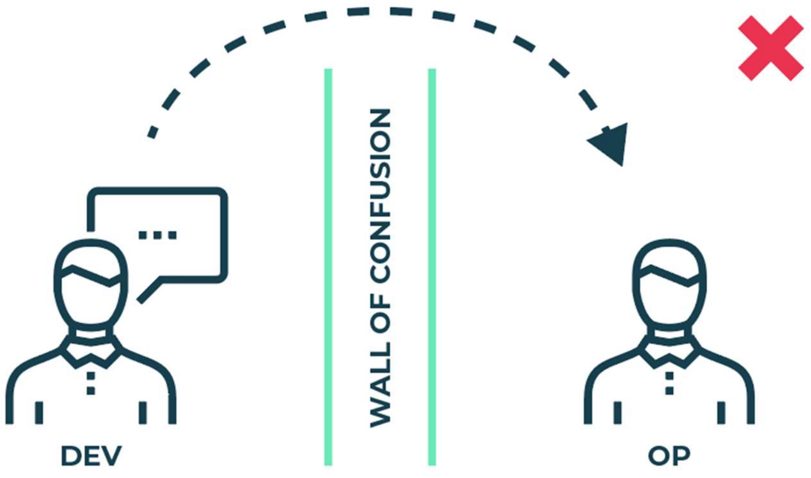


Figure 37

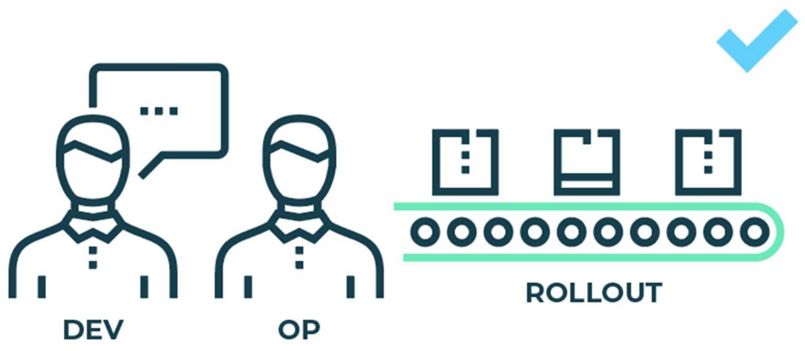


Figure 38

Concepts and Practices

The acronym CALMS outlines the five key concepts of DevOps:

C - Culture

A - Automation

L - Lean

M - Measurement

S – Sharing

So remember to keep CALMS and practise DevOps!

DevOps practices focus on maintaining a fast-paced, fluid production line. Continued integration allows for the principle of continuous delivery to produce something of value for the user. Other key DevOps practices include self-service configuration, automated provisioning and release management, automated acceptance testing (we already know the advantages of automated tests over manual), and version control across all production artefacts.

There are five key patterns within DevOps, all of which support continuous integration and deployment:

1. **Feature toggles (flags)** allow you to expose and hide features in a solution before they're ready for release, without changing the code.
2. **Branch by abstraction** is the technique of making changes gradually so that you can release the system regularly while still making changes. This is particularly useful for large-scale changes.
3. **Dark launching** is the process of releasing certain features to your users prior to the full release. It allows you to obtain user feedback and test for bugs incrementally.
4. **Canary release** is the process of rolling out changes across smaller sets of users before making them available to everyone. It's named after the

technique of using canaries in mines to give an early warning of any toxic gases in the air.

5. **Blue-green deployment** (or A / B deployment) involves running two identical production environments at the same time, so that you can run tests in one environment while the other is live. This means that there's less downtime when you're working on a new version.

Roles

There are many roles within DevOps because it spans across such wide business area. Some examples include:

- CDA – Continuous delivery architect
- DevOps evangelist
- Release manager
- Automation architect
- Software developer / tester
- Experience assurance (XA) professional
- Security engineer
- Utility technology player

As you can see, your role as a software developer technician sits amongst these DevOps roles, so it's a concept and way of working that you should be familiar with.

Culture of Trust

Operationally, DevOps builds a culture of high trust. This comes about through regular peer reviews of any production changes, proactive monitoring of the production environment and a win-win relationship between development and operations. This means that the outcomes will also be win-win for both parts of the organisation.