POWERING POTENTIAL

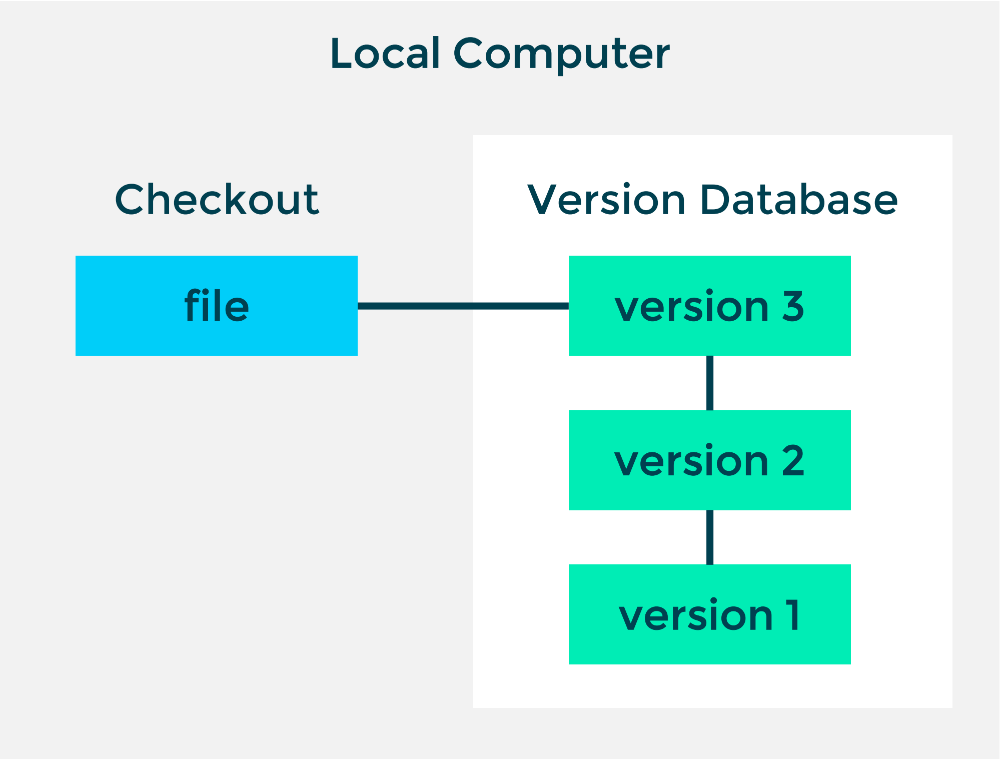
VERSION CONTROL

What is Version Control?

Version control is the process of recording changes to files.

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
- Maintain change logs
- Compare versioning issues



Local Computer

Checkout

Version Database

file

version 3

version 2

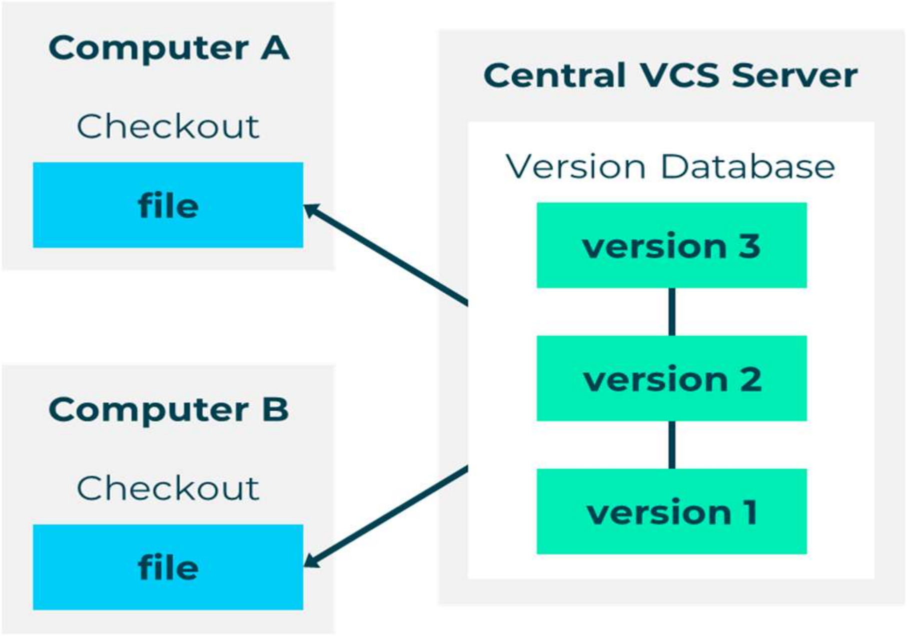
version 1

Benefits of Version Control

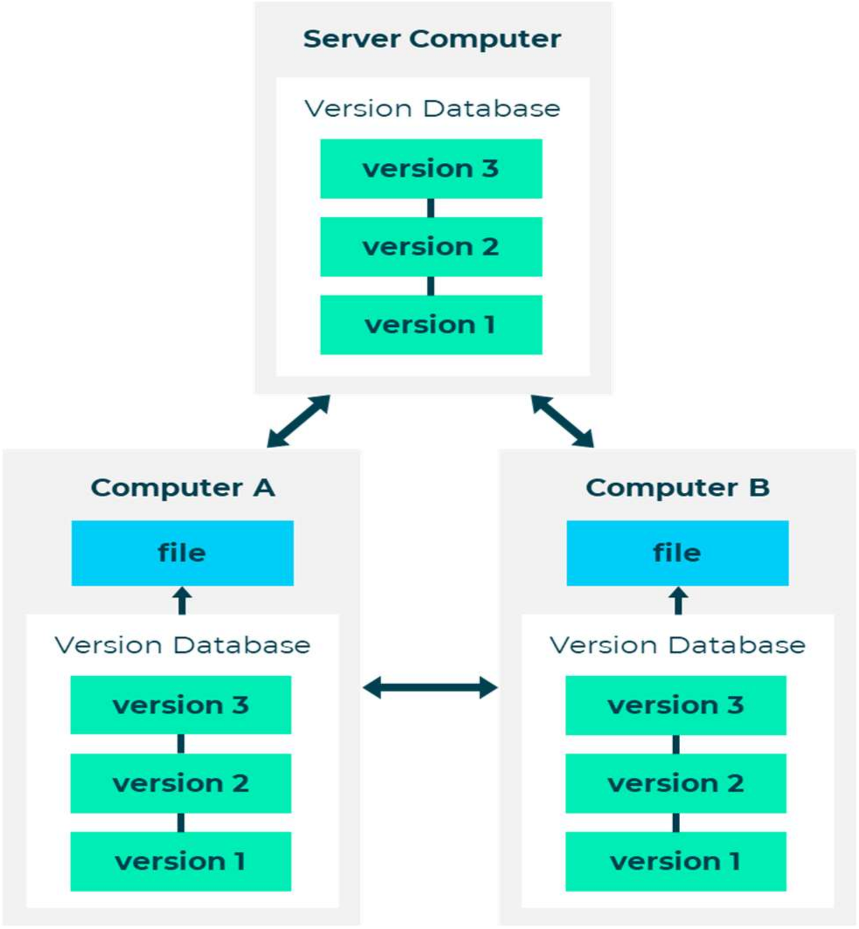
1. **Keep track of code and changes.**
   - Automated version management
   - One copy of the code everyone has access to
   - No more mailing around code and confusion trying to integrate it
2. **Multiple people can edit a single project at the same time.**
   - Push changes to the central repository
   - Merge together changes in files where there are conflicts
3. **Branch code to work on specific parts.**
   - Version 2.3 doesn’t need to die because someone else wants to look at version 3

Types of Version Control Systems

Centralised Version Control System (CVCS)



Distributed Version Control System (DVCS)



GIT AS A DVCS

GIT AS A DVCS

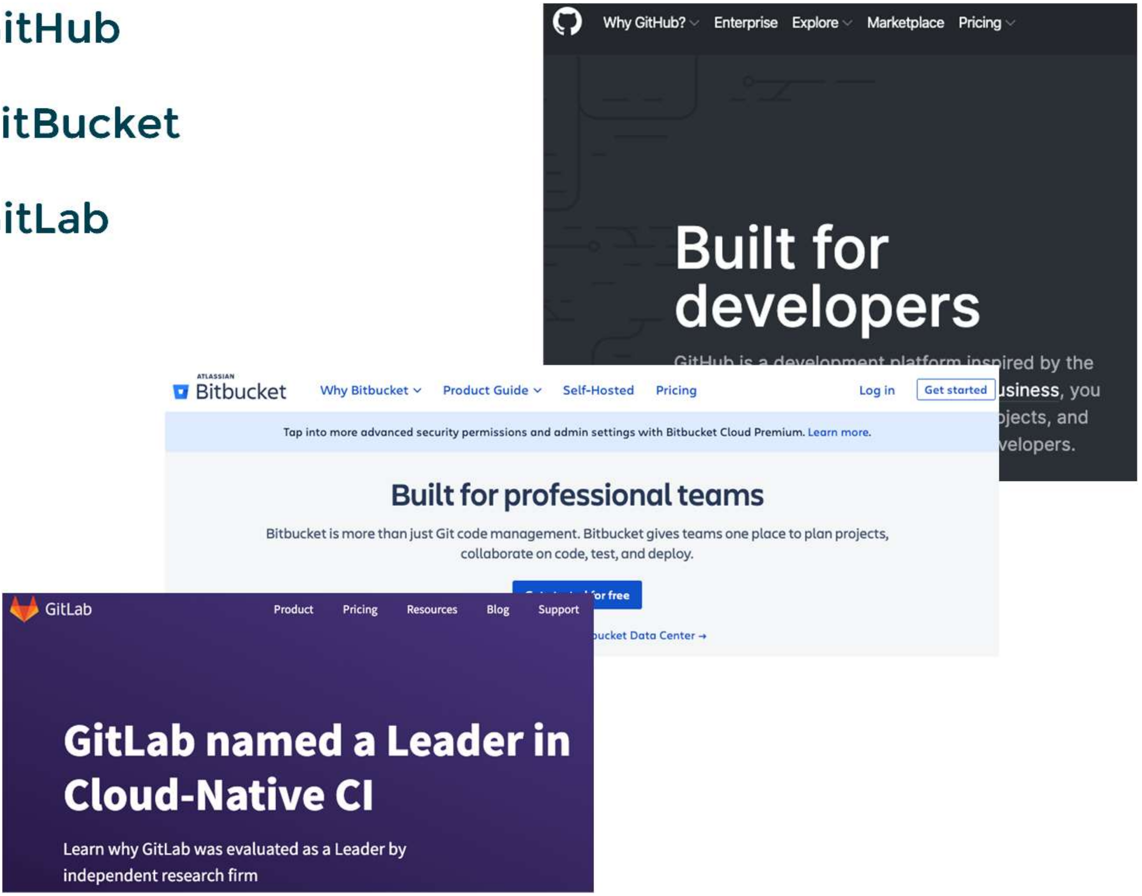
Git is a powerful version-control tool. Its origins are is Linux Development, so it's open source.

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

CHOOSING A HOSTING SERVICE FOR GIT

- GitHub
- BitBucket
- GitLab



Built for developers

GitHub is a development platform inspired by the
usiness, you
jects, and
velopers.

Tap into more advanced security permissions and admin settings with Bitbucket Cloud Premium. Learn more.

Built for professional teams

Bitbucket is more than just Git code management. Bitbucket gives teams one place to plan projects, collaborate on code, test, and deploy.

or free

bucket Data Center →

GitLab named a Leader in Cloud-Native CI

Learn why GitLab was evaluated as a Leader by independent research firm

USING A HOSTING SERVICE FOR GIT

To make changes to a Git repository, follow these steps:

1. Create a repository on a hosting site, or 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

INITIALISING REPOSITORIES

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

```
git init
```

INITIALISING A REPOSITORY WITH EXISTING FILES

```
git add *.pp
```

```
git add README.md
```

```
git commit -m "Initial commit"
```

CLONING AN EXISTING REPOSITORY

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

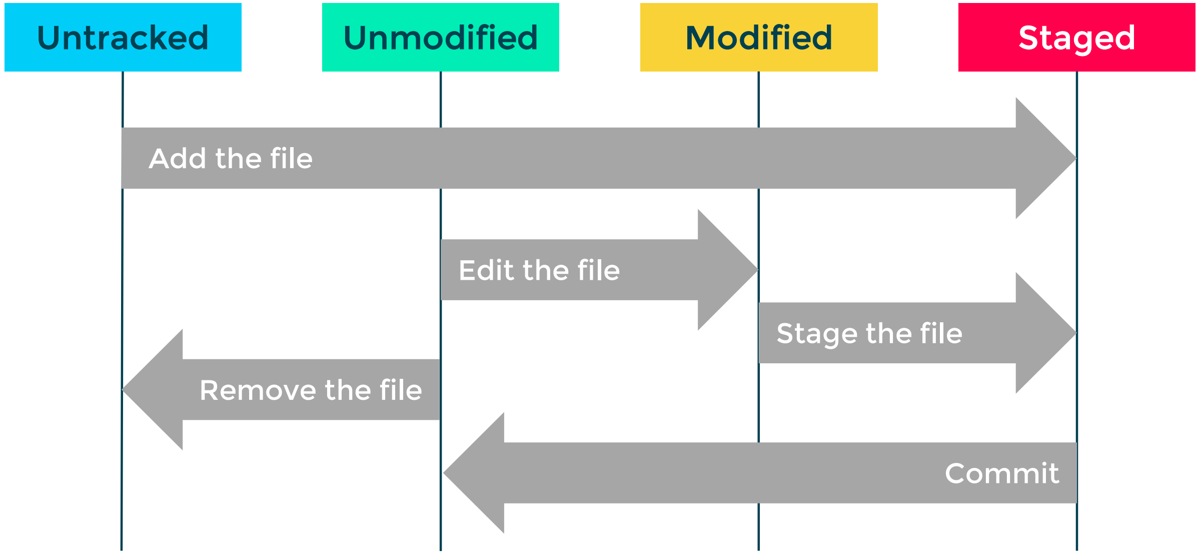
```
git clone git://github.com/resource
```

RECORDING CHANGES TO A REPOSITORY

Each file in a Git directory can be **tracked** or **untracked.**

1. Tracked files are files that were in the last snapshot. They can be unmodified, modified or staged.

2. Untracked files are everything else. They're not in your last snapshot or staging area.



RECORDING CHANGES TO A REPOSITORY, CONT.

The main tool you use to determine which files are in which state is the git status command. If you run this command directly after a clone, you should see something like this:

```
git status
On branch master
Nothing to commit, working directory clean
```

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

Remote repositories hold versions of a project or dependencies on the web / network.

To see configured remote repositories run the following command:

```
git remote
```

If you have 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 upstream, use:

```
git push origin master
```

`-v` shows you the URL that Git has stored for the shortname to be expanded to.

To rename a reference, run:

```
git remote rename
```

e.g.

```
git remote rename pb dave
```

```
git remote origin dave
```

PUSHING TO A REPOSITORY, CONT.

```
git remote rename
```

changes your remote branch names, too, e.g. pb / master is now dave / master.

To remove a reference for some reason, use git remote

```
git remote rm dave
```

```
git remote origin
```

PULLING FROM A REPOSITORY

To pull all the changes made to the repository, use:

```
git pull
```

Pull the repository before pushing changes

- You get an up to date copy of the repo to push to
- You can see any conflicts before they are pushed
- You can `stash` your changes before pulling the remote branch

CREATING A NEW BRANCH

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
```

CREATING A NEW BRANCH ALTERNATIVE

Rather than use the `checkout –b` command shown above, you could use the following two commands:

```
git branch newBranchName
```

```
git checkout newBranchName
```

ALTERNATIVES TO GIT

Git is popular due to:

- Open source nature
- Simplicity
- Context switching between branches easier
- Local staging area for commits
- GUI tools available such as Sourcetree
- Built-in tools in eclipse

…but it isn’t the only option. Alternatives include:

- Subversion
- CVS
- Mercurial
- Fossil
- Veracity
- SSH

LAB

"Git in 5 minutes"

POWERING
POTENTIAL

PAIR
PROGRAMMING

What is Pair Programming?

- Two developers work together on one project
- One developer codes while the other reviews
- Co-responsibility for outputs
- Developers can switch roles when needed
- Effective for identifying bugs and design problems, and maintaining coding standards

Image: By Kabren - Own work, CC BY-SA 3.0, [https://commons.wikimedia.org/w/index.php?curid=21903816]

BENEFITS OF PAIR PROGRAMMING

- Increased productivity
- Increased confidence
- Cross-learning
  - Reduced training costs and time
- Multiple points of view
  - Faster issue resolution
  - Fewer obstacles
- Continuous code reviews
  - Give feedback and reduce bugs more quickly
- Shared responsibility
  - Trust
  - Backups if a developer is on annual leave or falls ill

TYPES OF PAIRING

- Driver-Navigator
- Backseat navigator
- Tour guide
- Ping-pong
- Cross-functional
- Distributed

LAB

"Pair Programming concrete practice"

Afterwards, we'll discuss...

- Which pair programming method(s) you tried
- What went well
- What didn't go so well

SOFTWARE PATTERNS

What are Software Design Patterns?

- Reusable solutions to commonly occurring problems
- Emerge over time as developers write code and establish best practices
- Solve particular problems in code generation and interactions
- Powerful if used in the correct way applied in the correct circumstance

```
elif _operation == "MIRROR_Y":
    mirror_mod.use_x = False
    mirror_mod.use_y = True
    mirror_mod.use_z = False
elif _operation == "MIRROR_Z":
    mirror_mod.use_x = False
    mirror_mod.use_y = False
    mirror_mod.use_z = True

    #selection at the end -add back the deselec
mirror_ob.select= 1
modifier_ob.select=1
bpy.context.scene.objects.active = modifier_ob
print("Selected" + str(modifier_ob)) # modifier
    #mirror_ob.select = 0
```

The Principle of Balance

Find the balance between…

Too simple

Too complex

COMMON DESIGN PATTERNS

- Facade
- Proxy
- Command
- Observer (subsumed into .NET as events)
- State
- Strategy / Template Method
- Factory
- Singleton

THE ‘GANG OF FOUR’

WHO WERE THE ‘GANG OF FOUR’?

Design Patterns: Elements of Reusable Object-Oriented Software (1994)

- Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides

‘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.’

- Addison-Wesley publishers, emphasis added for this session

GANG OF FOUR PATTERNS

Three types of pattern:

- Creational
- Structural
- Behavioural

POWERING
POTENTIAL

TESTING

UNIT TESTING

What are Unit Tests?

- A software testing method that tests individual sections of source code
- Unit = the smallest testable part of the software
- Allows us to validate units in isolation
- Test-driven development – written before the code to be tested
- Automated
- Developers write, execute and maintain test code

BENEFITS OF UNIT TESTING

- Shows you that the code works as you develop it
- Detect bugs or defects early on
- Easily localise the source of the bug
- Continuous integration (CI) makes deployment easier

- TL;DR: Confidence in code quality and functionality

KEY CONCEPTS

- Core practice of XP
- Can be adopted within other methodologies
- TDD: test written before implementation and tests drive API design
- Automated and self-validating
- Easy to maintain

F.I.R.S.T.

Unit tests must be…

- Fast
- Independent
- Repeatable
- Self-validating
- Timely

- Robert Martin, *Clean* Code, 2009

READABLE TESTS: CODING BY INTENTION

Four phases:

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.

Each phase should be:

- Clearly expressed, inc. your expected outcomes
- Well documented

RIGHT B.I.C.E.P

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

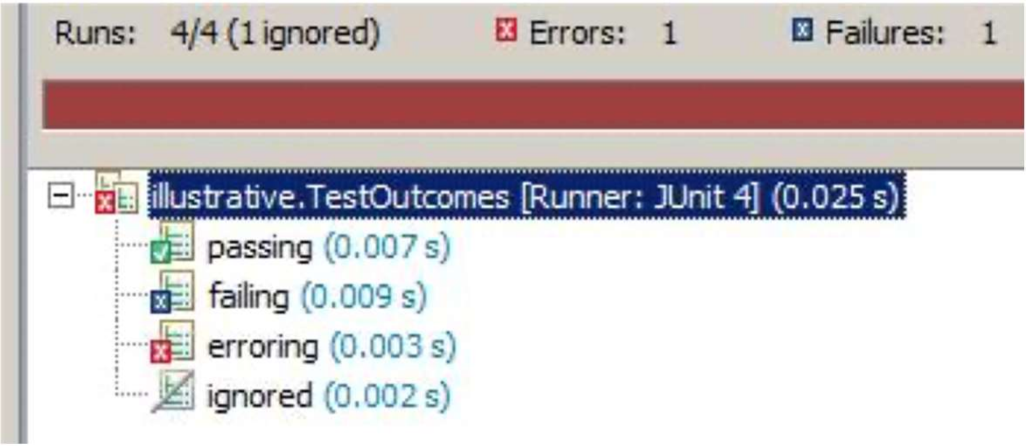
Passing: ultimately, all our tests must pass

Failing: in TDD, always start with a test which fails

Erroring: test neither passes nor fails

- Something has gone wrong, a run-time error has occurred; e.g. necessary library jar has not been provided

Ignored: `@Test @Ignore`



```
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)
```

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	

LABS

"Unit Testing" lab

+

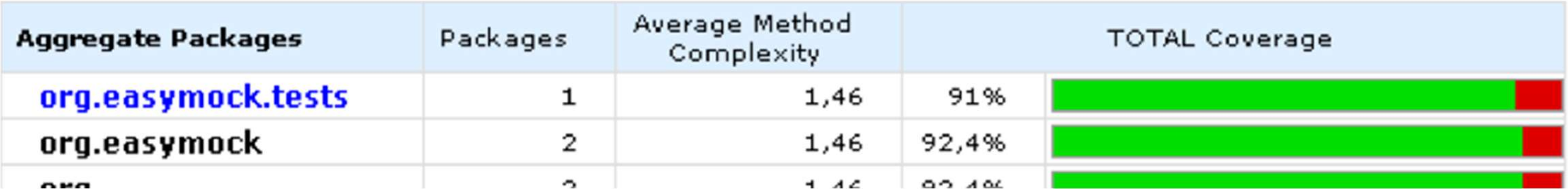
"Readable Tests" lab

CODE COVERAGE

What is Code Coverage?

Ask: How well do the tests cover the code base?

- White box tests – you should cover every line of code
- Tools: Clover and Istanbul
- Acceptable percentage of coverage
  - Ordinary development: struggle to get to 90%
  - TDD: 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%	🟩🟥
org	2	1,46	92,4%	🟩🟥

COVERAGE METRICS

1. Method coverage
   - % of methods covered
   - Very coarse-grained, i.e. inaccurate
2. Symbol coverage
   - % of sequence points (statements) that have been exercised
3. Branch coverage
   - % of completely executed blocks in a method
4. Cyclomatic complexity
   - How many linearly independent paths through the code there are

PERCENTAGE ISN'T EVERYTHING

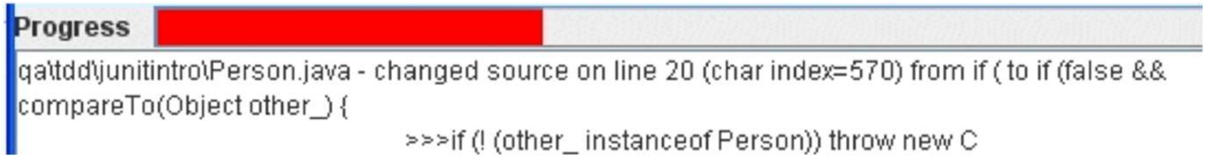
‘100% test coverage does not guarantee that your application is 100% tested’

– Massol

- Test generation tools which auto-generate unit tests with 100% coverage, but the tests are worthless
- Test class full of trivial Get / Set methods which make it hard to see where the tests with real value are
- Use as a helper to ‘get things set up’ and delete it when it’s served that purpose and detect regions of code which are not tested
- Failing test counts towards coverage the same as passing test

QUALITY OVER QUANTITY

- ‘The more tests you have, the better’ – **FALSE!**
- Jester – JUnit test tester (http://jester.sourceforge.net/)



Progress

```
qa\tdd\junitintro\Person.java - changed source on line 20 (char index=570) from if ( to if (false &&
compareTo(Object other_) {
                                        >>>if (! (other_ instanceof Person)) throw new C
```

- Slow, so use it in a targeted manner
- Produces false positives

LAB

"Code Coverage" Lab

TESTING PATTERNS

ASSERTIONS

Expressions that encapsulate a testable logic about the product you’re testing.

TYPES OF ASSERTION

Resulting state assertion: a standard test; the state that we expect.

Guard assertion: assert a precondition for the test to be correct, and follow it with the resulting state.

Delta assertion: if you can’t guarantee the absolute resulting state, test the delta (difference) between the initial and resulting states.

Custom assertion: helps your test code respect DRY (‘don’t repeat yourself’, i.e. there aren’t any duplications).

PARAMETERISED TEST

- Data-driven testing: one test method, multiple data sets
- Data hard-coded in 2D array, or from an external source

```
@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;
}
```

PARAMETERISED TEST

JUnit 4 - you can set up parameterised tests with annotations:

```
@RunWith(value=Parameterized.class)
public class UtilsTest {
  private int input;
  private String expected;
}
```

The test method uses instance variables of the class:

```
@Test
public void testGetMonthString() {
  assertEquals("Incorrect month String", expected,
Utils.getMonthString(input));
}
```

PARAMETERISED CREATION METHOD

- Hides attributes essential to fixtures, but irrelevant to the test
- Factor out fixture object creation from setUp to PCM
- Useful when creating a complex mock, esp. if it will be used in multiple tests
- Consider also Builder Pattern

PcM example

```
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;
}
```

EXTRA CONSTRUCTOR

- If existing constructor hard-codes some dependencies, use an extra constructor to inject dependencies by test (mock or stub)
- Introduces a 'trap door' to make 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

- **Constructor injection:** API signals that the parameter(s) isn’t optional, you must supply it when creating the object
- **Setter injection:** API signals the dependency is optional / changeable
- With either 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.
}
```

TEST-SPECIFIC SUBCLASS

- Create a behaviour-modifying or state-exposing subclass
- Not usually a TDD approach
- E.g. lottery class
  - Default constructor creates real `RandomNumberGenerator`
  - `NumberGenerator` field has `protected` visibility
  - `TestableLottery` extends `Lottery`

```
public TestableLottery(NumberGenerator generator) {
    this.generator = generator;
}
```

- Inherits method to test as-is
- Test creates instance of testable subclass, injecting mock

FACTORY

Factory provides means for test to change type returned

- Test sets factory to return mock or stub
- Application code semantics completely unchanged

Use factory pattern for dependencies

- E.g. CUT uses factory method
  - Extract and override: extract dependency creation to factory
  - Override method in test-specific subclass
  - Or: Client asks factory class for dependency

```
public Lottery() {
   generator = NumberGeneratorFactory.create();
}
```

See pages 17-18 in your learner guide for an example of the factory pattern used by CUT.

OBJECT MOTHER

- Create example objects that you can use in your tests
  - Customise the objects you create
  - Update the objects during the tests
  - Delete the object from the database once you’ve completed your tests
- Factors out creation of business objects to factory class, or just a class containing fixtures to avoid duplication
- e.g. a set of standard ‘personas’:

```
private static List<Skill> bobskills = Arrays.asList(CARPENTER, PLUMBER);
public static Builder BOB = new Builder("Bob", bobskills);
```

LAB

"Test Patterns" Lab

TEST DOUBLES

WHAT ARE TEST DOUBLES?

- Your code interacts with others things
  - Network resource – database, web service, etc.
  - Code being developed in parallel by another person/team
  - Container (e.g. objects with lifecycle methods)

- **Collaborators:** objects with which the class under test interacts with
- If the collaborator doesn't exist yet or isn't available, use a substitute called a **test double**

Test doubles allow you to…

- Run the test in its real environment
- Test interactions between your class and the rest

WHAT IS MOCKING?

- A method for testing your code functionality in isolation
- Does not require:
  - Database connections
  - File server reads
  - Web services
- The mock object itself does the ‘mocking’ of the real service, returning dummy data
- The mock object simulates the behaviour of a real method, external component

MOCK OBJECTS

> Mock: ‘an object created to stand in for an object that your code will be collaborating with. Your code can call methods on the mock object, which will deliver results as set up by your tests’
>
> – Massol, p. 141

To mock a database ResultSet:

- You are not creating an object with state (records with mock data sets)
- You are creating an object which will respond to (method calls from) your code as if it had a certain state

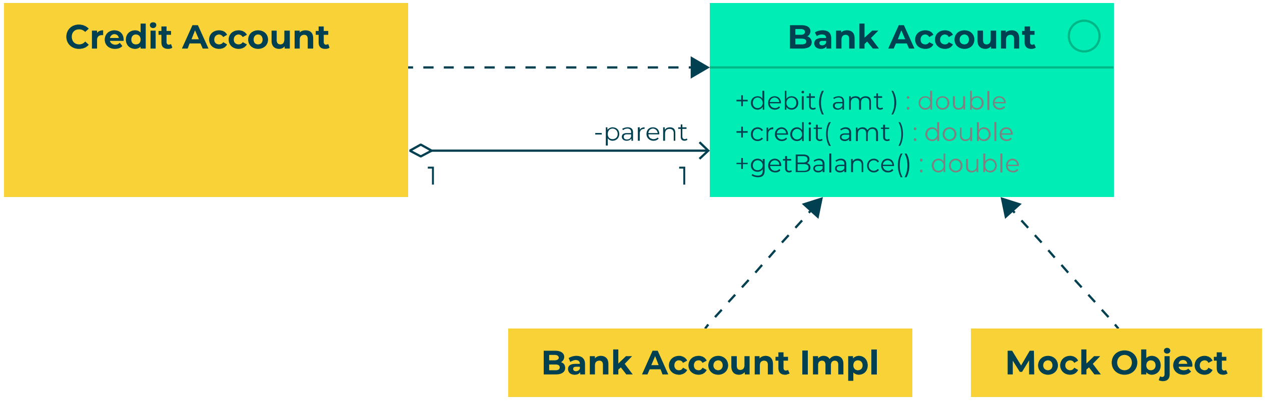
Several Dynamic Mock Frameworks in Java:

- EasyMock
- JMock
- Mockito
- Powermock
- JMockit and more

WHEN TO 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



Credit Account

-parent

1

1

Bank Account

+debit(amt) : double
+credit(amt) : double
+getBalance() : double

Bank Account Impl

Mock Object

From *Endo-testing: Unit Testing with Mock Objects,* Mackinnon, Freeman and Craig

DRAWBACKS OF MOCK OBJECTS

1. Mocks don’t test interactions with container or between the components.
2. Tests are coupled very tightly to implementation.
3. Mocks don’t test the deployment part of components.
4. Most frameworks can’t mock static methods, final classes and methods

The first three points here are adapted from Massol, *JUnit In Action*, p. 171.

STUBS

Stub: controllable replacement for existing dependency

- A class which is (ideally) the simplest possible implementation of the logic in the real code
- Provide pre-programmed answers to calls they receive
- Won’t respond to anything outside of what you’ve programmed

Good for:

- Coarse-grained testing, e.g. replacing a complete external system

Drawbacks:

- Often complex to write
- Introduce their own debugging & maintenance issues

MOCKS VS STUBS

- Not mutually exclusive; you can combine their use
- Both can be handwritten, simple classes – simplicity and readability are at an advantage over frameworks
- Stubs enable tests by replacing external dependencies
- Asserts are against the Class Under Test, not the stub

```
Test <–––––> CUT <–––––> Stub
   asserts        interacts
```

- 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

```
CUT <–––––> Mock <–––––> Test
   interacts       asserts
```

WHAT IS DEPENDENCY INJECTION (DI)?

- Injecting dependencies means setting relations between instances
- It helps to remove tight coupling
- Use interfaces instead of concrete classes to illustrate a dependency:

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

- You ‘inject’ the interface into the class:

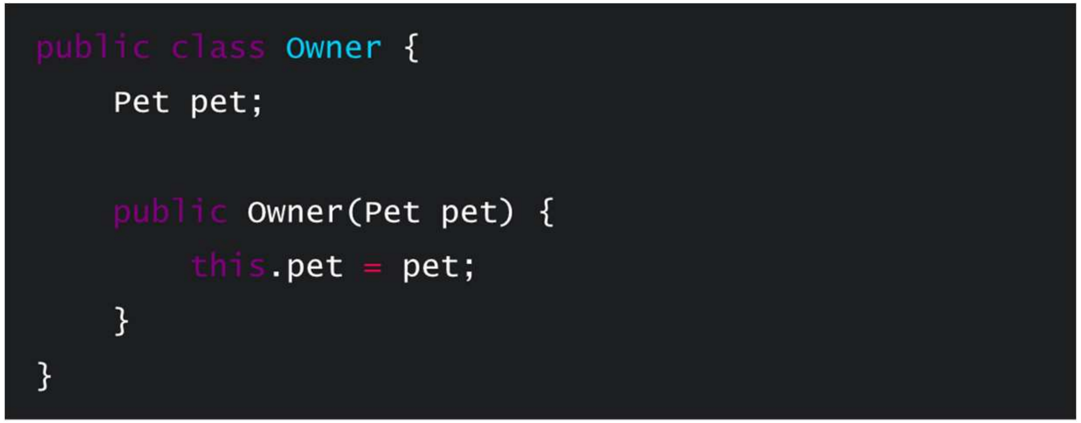
```
public class FastEngine {
   private IEngine engine;
   public FastEngine(IEngine engine) { this.engine = engine; }
}
```

Dependency Injection Methodology

- Rather than creating concrete instances, inject them at runtime
- Need a way to manage (or wire) the dependencies

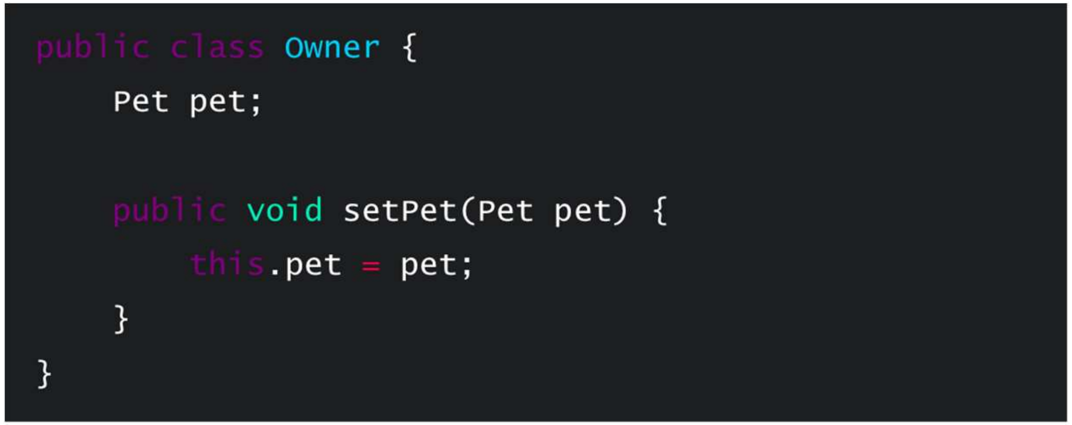
  Java’s Spring framework

- Two ways to perform dependency injection:



```java
public class Owner {
    Pet pet;

    public Owner(Pet pet) {
        this.pet = pet;
    }
}
```

Constructor



```java
public class Owner {
    Pet pet;

    public void setPet(Pet pet) {
        this.pet = pet;
    }
}
```

Setter

DEPENDENCY INJECTION EXAMPLE

Without injection:

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

Using injection:

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

DEPENDENCY INJECTION CONTAINERS

- Containers inject the mock objects into your unit tests during unit testing
- Some DI containers include:
  - Spring DI
  - Butterfly DI Container
  - Dagger
  - Guice
  - PicoContainer
- Many unit tests don’t require a DI container if their dependencies are simple to mock out

LAB

"Test Doubles" Lab

POWERING POTENTIAL

REFACTORING WITH EXISTING TESTS

PRINCIPLES OF REFACTORING

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

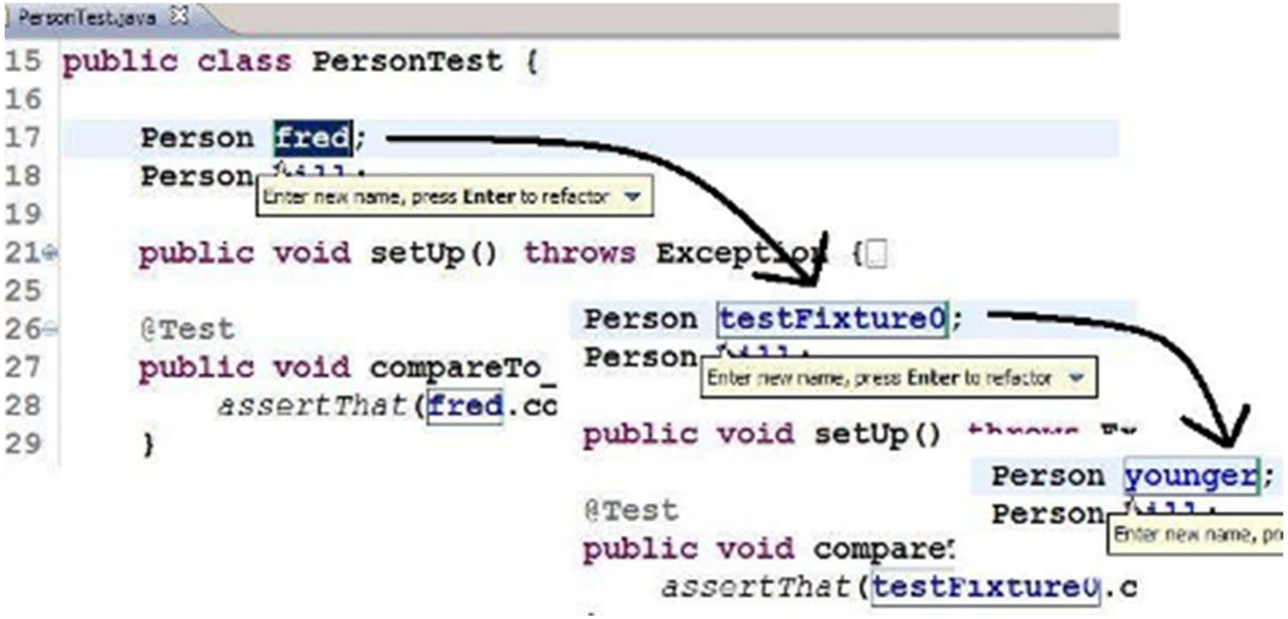
Taken from: [http://www.agileadvice.com/2016/03/24/scrumxplean/refactoring-4-key-principles/]

BENEFITS OF 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
- Refactoring ≠ Rewriting
- Improves the design of software
- Makes it easier to find bugs, as code is cleaner
- Many now automated through Eclipse, etc.

COMMON REFACTORINGS: REFACTOR-RENAME

Repeat <ALT> <SHIFT> R until you're satisfied you have an identifier that best reflects what the item represents.



PersonTest.java

```
public class PersonTest {

    Person fred;
    Person bill;

    public void setUp() throws Exception {

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

Enter new name, press Enter to refactor

```
Person testFixture0;
Person bill;

public void setUp() throws Ex

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

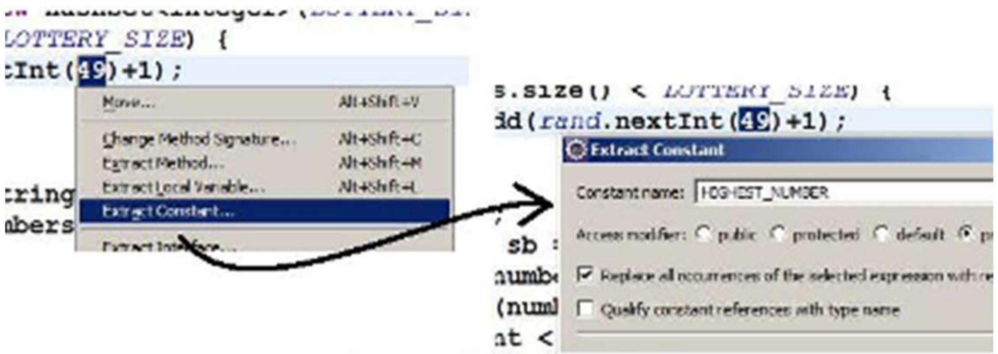
Enter new name, press Enter to refactor

```
Person younger;
Person bill;
```

Enter new name, pr

Common Refactorings: Extract Constant / ‘No Magic Numbers’

- Highlight literal (e.g. int or String), then <ALT> <SHIFT> T


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...
Extract Constant

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

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

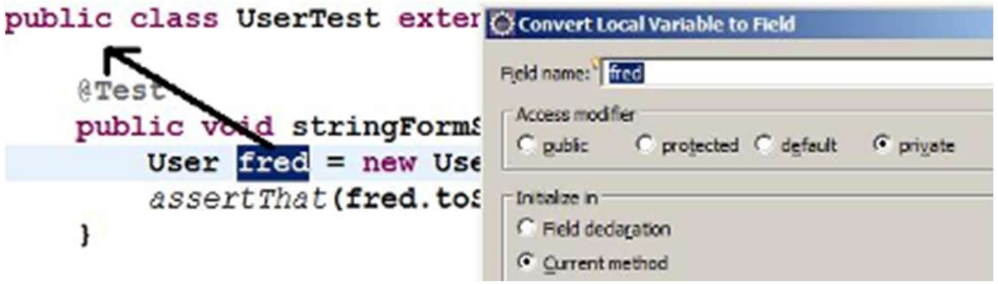
Constant name: HIGHEST_NUMBER

Access modifier: ○ public ○ protected ○ default ● p

☑ Replace all occurrences of the selected expression with re

□ Qualify constant references with type name

- Convert Local Variable to Field



```
public class UserTest exter

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

Field name: fred

Access modifier

○ public ○ protected ○ default ● private

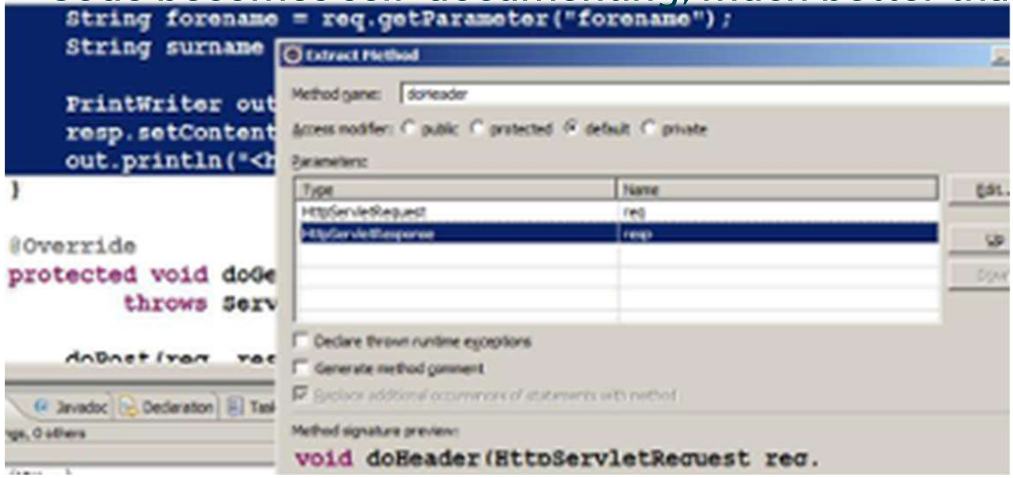
Initialize in

○ Field declaration

● Current method

COMMON REFACTORINGS: EXTRACT METHOD

- Eclipse <ALT> <SHIFT> M
- Remove code duplication
- Break up overly long methods
- Clarity: move lines to a method which expresses the intent
  - Why is there this call reader.readLine() which does nothing?
  - Extract to method: discardHeaderLine()
  - Code becomes self-documenting; much better than comments



```
    String forename = req.getParamoter("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

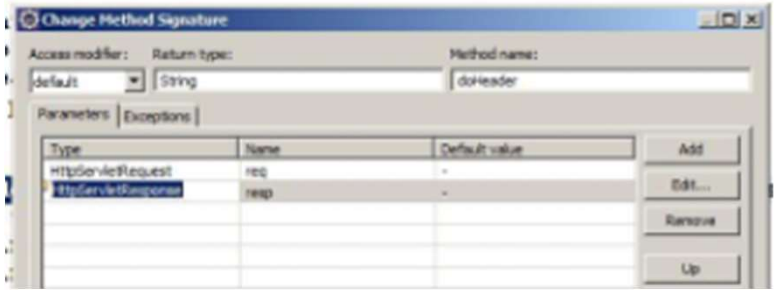
☑ Replace additional occurrences of statements with method

Method signature preview:

`void doHeader(HttpServletRequest req,`

COMMON REFACTORINGS - EXTRACT METHOD FOR TESTABILITY

1. Highlight lines.
2. Invoke refactoring.
3. Enter new method name, 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

Type	Name	Default value
HttpServletRequest	req	-
HttpServletResponse	resp	-

COMMON REFACTORINGS: EXTRACT CLASS

Break a large class into smaller classes based on:

- Cohesive behaviour
- Related functionality

COMMON REFACTORINGS: REPLACE INHERITANCE BY DELEGATION

aka Favour Composition over Inheritance

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

Reasoning: a Deck is a list of Cards

Wrong: relationship is `has-a`, not `is-a`

Doesn’t expose unnecessary methods of `ArrayList`

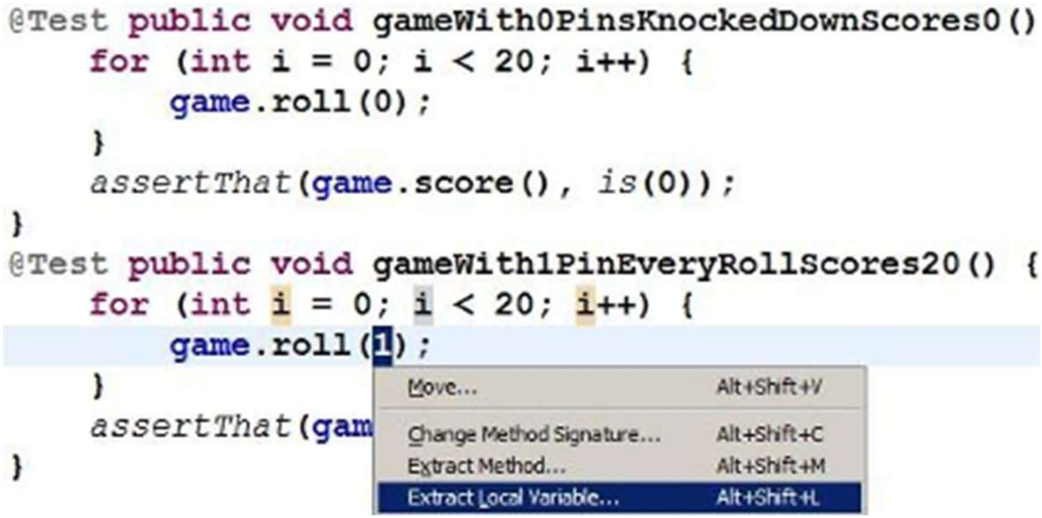
Expose only methods a `Deck` needs, and delegate their implementation to the contained `ArrayList`

COMMON REFACTORINGS: REMOVE DUPLICATION

DRY: Don’t Repeat Yourself

E.g. two blocks of code which are almost identical:

- Extract value(s) where they differ to variable(s)
- Will become input parameter(s) to single common method
- Place declaration of local variable int pins = 1 outside loop
- Apply Extract Method refactoring to thr 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

COMMON REFACTORINGS: REMOVE DUPLICATION

This example (a variant of the one in Robert Martin's Bowling Game Kata), the test methods would end up looking like:

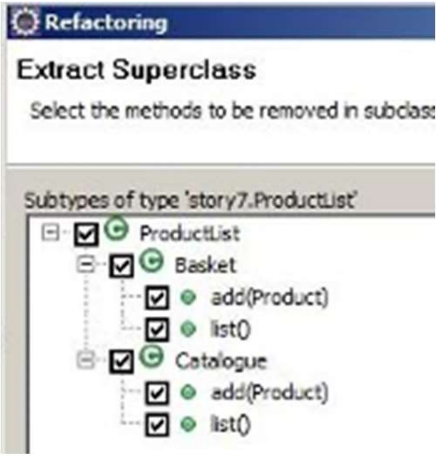
```
@Test public void gameWith0PinsKnockedDownScores0() {
   roll20(0);
   assertThat(game.score(), is(0));
}
```

with the commonality between the two blocks of code captured in the method:

```
private void roll20(int pins) {
   for (int i = 0; i < 20; i++) {
      game.roll(pins);
   }
}
```

Common Refactorings: Extract Superclass

1. Suppose Basket and Catalog have commonality.
   - Both have a List of Products, methods to `add()` and `list()`
2. Choose one, e.g. Basket -> Extract Superclass.
3. Add the other types to extract a superclass from.
4. Select methods, fields to be extracted.
5. Basket, etc. will extend new superclass.

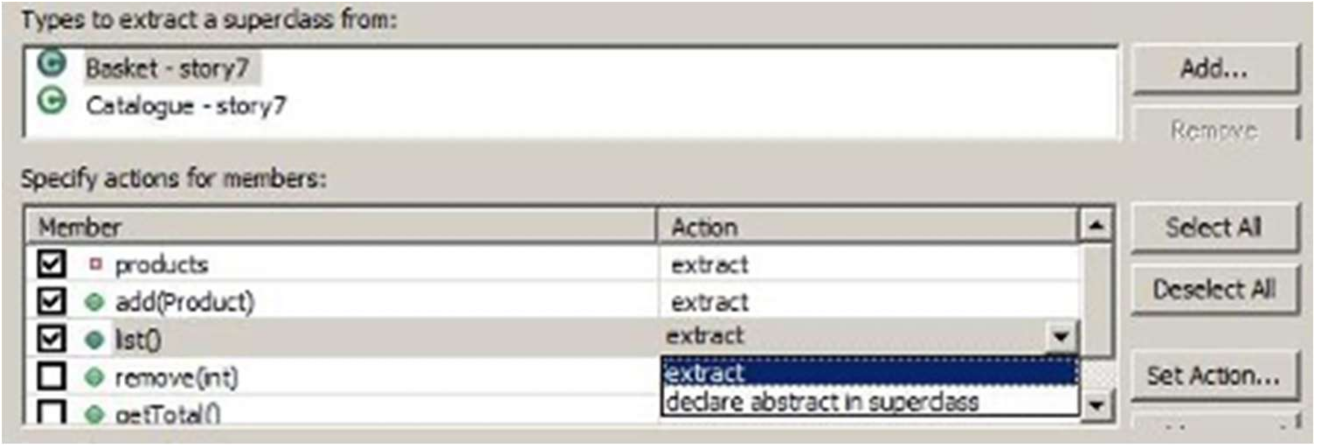


Extract Superclass

Select the methods to be removed in subclas

Subtypes of type 'story7.ProductList'

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



Types to extract a superclass from:

- Basket - story7
- Catalogue - story7

Specify actions for members:

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

COMMON REFACTORINGS: CODING TO INTERFACES

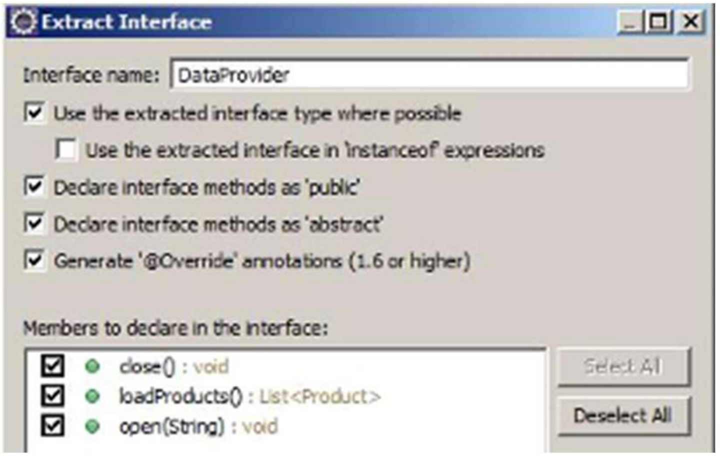
Suppose a class needs a Repository / DAO:

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

- Candidate for decoupling interface & implementation:

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

- Plug in different implementations w/o affecting rest of code.
- Choose method names which are neutral about data source.
- Choose exception type at same level of abstraction.



Extract Interface

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

LEGACY CODE

- Code that you don’t understand… Is it better to refactor the code or to rewrite?
- A total rewrite only makes sense if you fully understand the full requirements of the system
- Refactoring requires you to fully understand the codebase

Three options:

1. Leave the old code alone and write more legacy code.
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 for substitution of classes and functions.

Object Seams

Dependency Substitution based on either inheritance or interface implementation.

Example

This example is based on the substitution principle:

```
public void ProcessAccount(AccountProcessor proc, Account acc)
{
}
...
class TestAccountProcessor: public AccountProcessor
{
   // Substitute implementation
}
```

SEAMS

For legacy code:

- Don't change, substitute when possible
- If you have to, change the smallest amount of code possible

Linker Seams

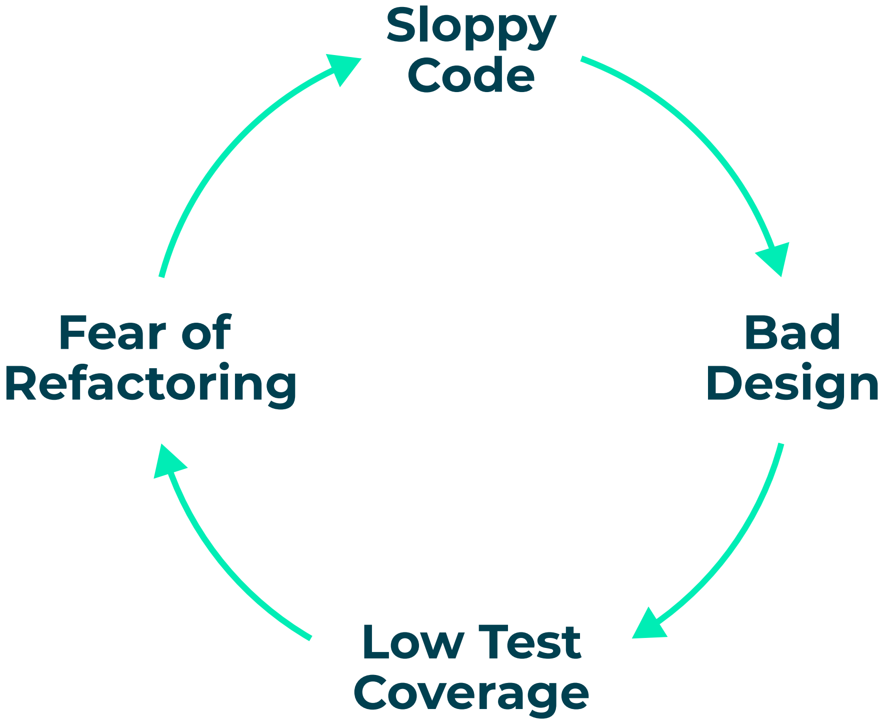
- Different Builds can be defined by varying the classpath
- For package class com.qa.mainframe, we could:
  - Define a substituted set of classes within the package
  - Change the classpath to create two different builds

Pre-processor Seams

- Based on pre-processor directors managing substitution Requires a pre-processor

Refactoring with little or no test coverage

- Code that has little or no test coverage is usually badly designed
- Makes it hard to know if your code changes with break other parts of the application
- This makes it harder to write tests



Base image from [https://codeclimate.com/blog/refactoring-without-good-tests/]

REFACTORING WITHOUT TESTS

- It is usually a good idea to have unit tests in place before you start to refactor some code
- It is sometimes necessary to refactor without having unit tests. You may be told not to refactor code if it has no unit tests
  - The code stays unimproved as it’s too much effort to create the unit tests required
- Automatic refactoring
  - Many tools have refactor options built-in, e.g. Eclipse-IDE
- Small step refactoring
  - Make very small simple steps that are so trivial, there is almost no chance of making a mistake. This process of making small steps creates a net refactoring effect

LAB

"Refactoring with existing tests" Lab

Afterwards, we'll discuss what you tried...

TESTING TECHNIQUES

TEST-DRIVEN DEVELOPMENT

WHAT IS TDD?

- Repeating cycle of turning requirements into test cases and improving code to pass the tests
- Core practice of XP
- Can be adopted within other methodologies
- Test written before implementation
- Tests drive design of API

TDD BENEFITS

- 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.

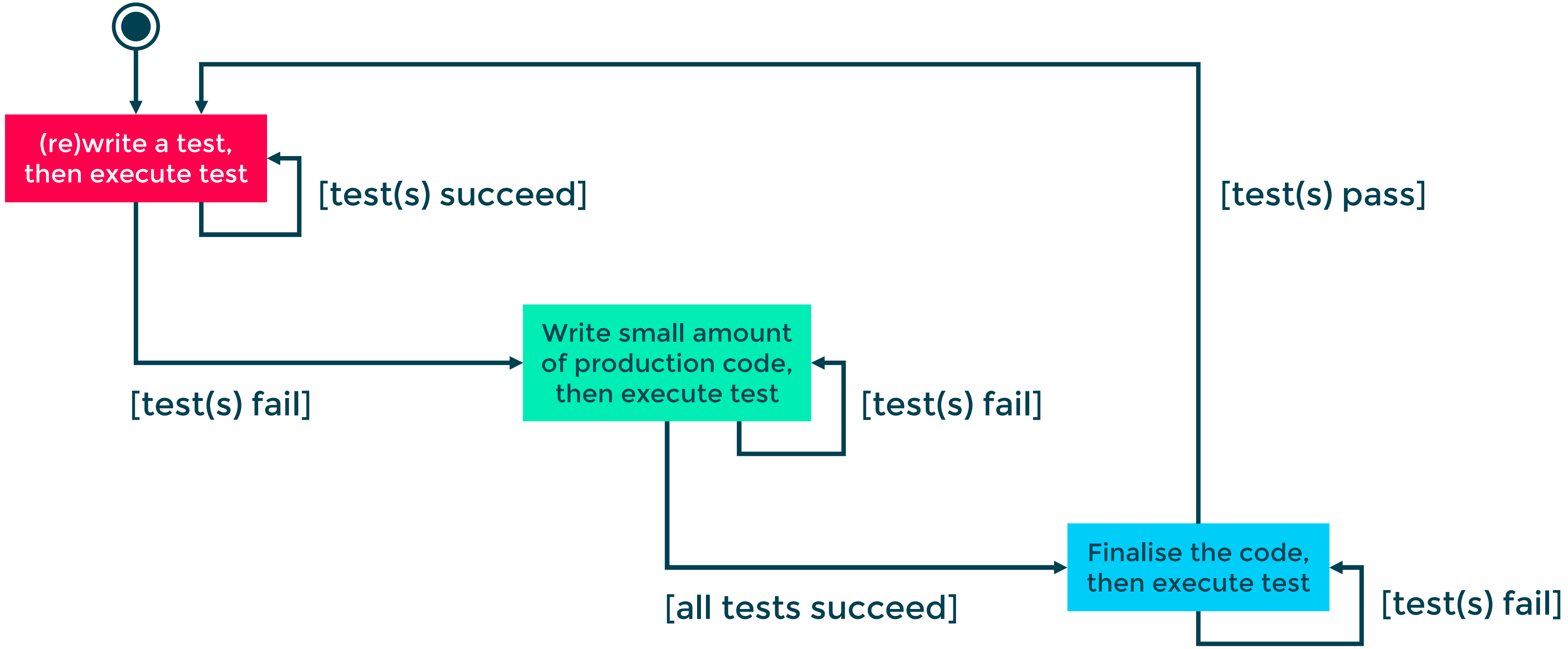
TDD STEPS

1. Write a failing test.
2. Write just enough code to pass the test.
3. Pass the test.
4. Refactor.

Different to traditional development…

- Tests as an afterthought
- High level of defects
- Lengthy testing phase
- High time and monetary costs
- Poor maintainability

The Red - Green - Refactor Workflow



(re)write a test, then execute test

[test(s) succeed]

[test(s) fail]

Write small amount of production code, then execute test

[test(s) fail]

[all tests succeed]

Finalise the code, then execute test

[test(s) fail]

[test(s) pass]

Beck’s TDD Strategies

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.**

The point is to get developers to work in very small steps, continually re-running the tests.

```
    <==> A
      / \
     /   \
    /     \
   /       \
  /         \
 /           \
¶             []
B              C
```

BECK'S TESTING HEURISTICS

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

LAB

"Test Driven Development (TDD)" Lab

Afterwards, we'll discuss what you thought...

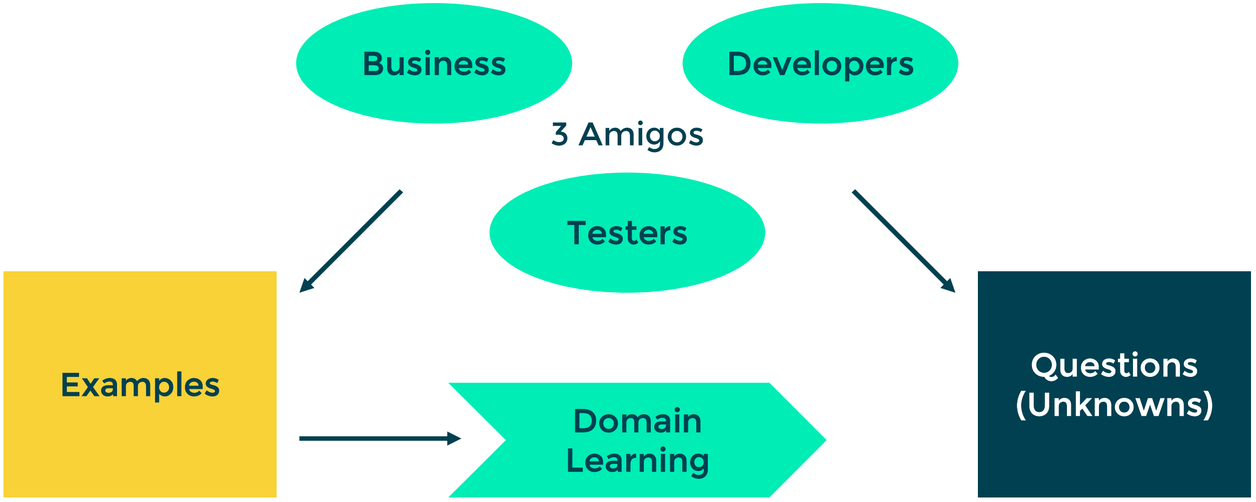
BEHAVIOUR-DRIVEN DEVELOPMENT

WHAT IS BDD?

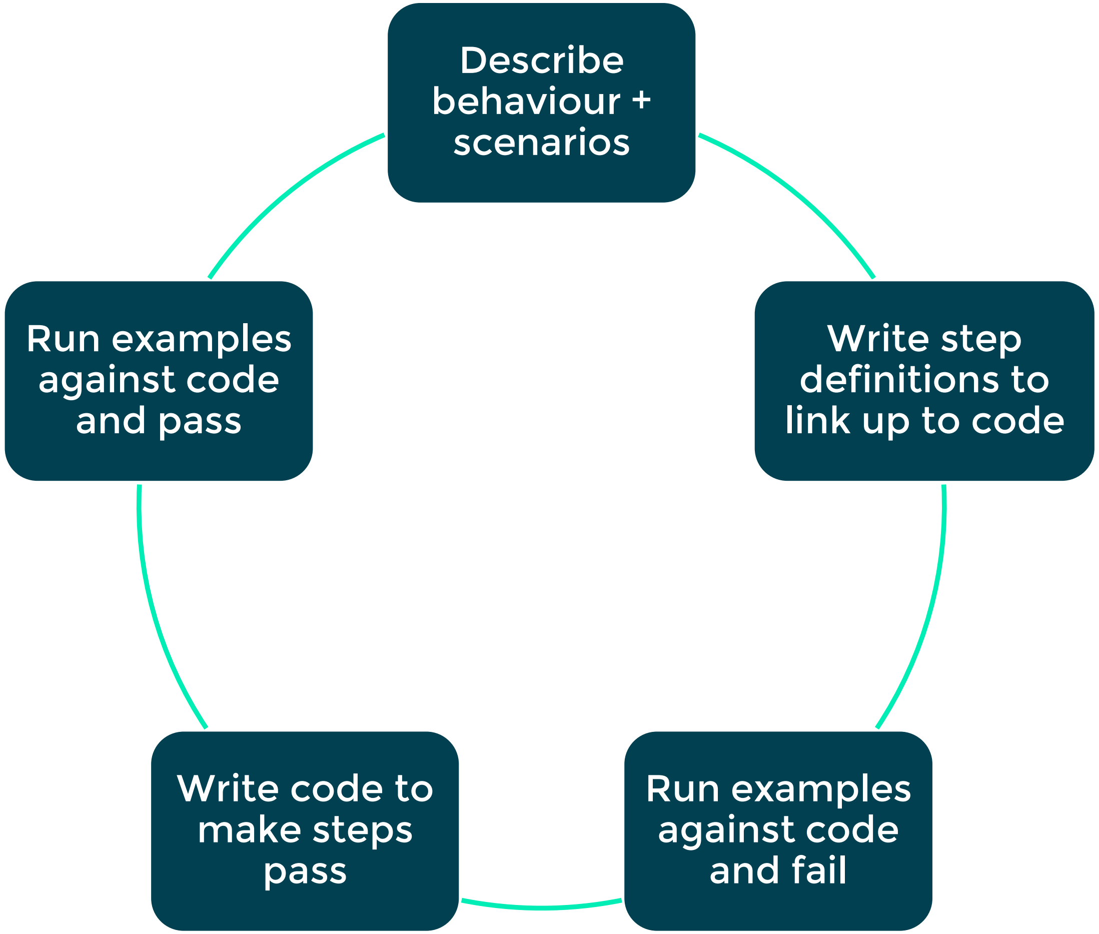
BDD is about conversation and collaboration

Language understandable to all stakeholders:

- Requirements = features
- Acceptance criteria = scenarios
- Scenarios illustrate how features work



BDD 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

DEFINING SCENARIOS

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>	

WRITING SCENARIOS

1. Write the ‘happy path’ scenario:

   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"

2. Then add alternative ‘edge’ scenarios:

   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 PRACTICES

- 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

IMPERATIVE VS DECLARATIVE

Imperative Style

- What should happen, including a detailed description of the steps
- Developers write matching step definitions at a lower level
- Low level step definitions are often reusable from other scenarios
- Less readable / harder to understand overall

Declarative Style

- What should happen, not necessarily how its achieved
- Less detail, more generalised style
- Developers write higher level step definitions
- Step definitions are less reusable and more specific to the scenario
- More readable / easier to understand

USING BACKGROUNDS

Use Background to avoid repeating the same clauses continually.

- Keep your Background section short and relevant
- Don’t use Background to set up **complicated states**, unless that state is actually something the client needs to know
- If the Background section has scrolled off the screen, think about using higher-level steps, or splitting the *.feature file
- Make your Background section vivid and try to tell a story

```
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

Ask these 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!

WRITING USER STORIES

A user story usually contains:

- Definition
- Acceptance criteria

Definitions typically use the ‘Connextra’ format:

As an X

I want Y

So that Z

Acceptance criteria are conditions that have to be satisfied for the story to conclude successfully

- Definition of done
- Makes the story testable

USING EXAMPLES

User stories with raw acceptance criteria can miss the overall intent.

Examples of how it should work make things clearer:

- ‘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 illustrate:

- What could happen
- Under what circumstances it might happen
- What should be done when it happens

CUCUMBER AND GHERKIN

‘Cucumber is a tool that supports Behaviour-Driven Development (BDD).’

Cucumber reads executable specifications written in plain text and validates that the software does what those specifications say.

The specifications consist of **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
```

The scenarios must follow some basic syntax rules, called **Gherkin**.

STEP DEFINITIONS

Scenarios are broken into step definitions:

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

STEP DEFINITIONS

```
@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());
}
```

APPLYING GHERKIN TO ACCEPTANCE CRITERIA

Customer requirement: ‘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’

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

Gherkin keywords

AUTOMATION

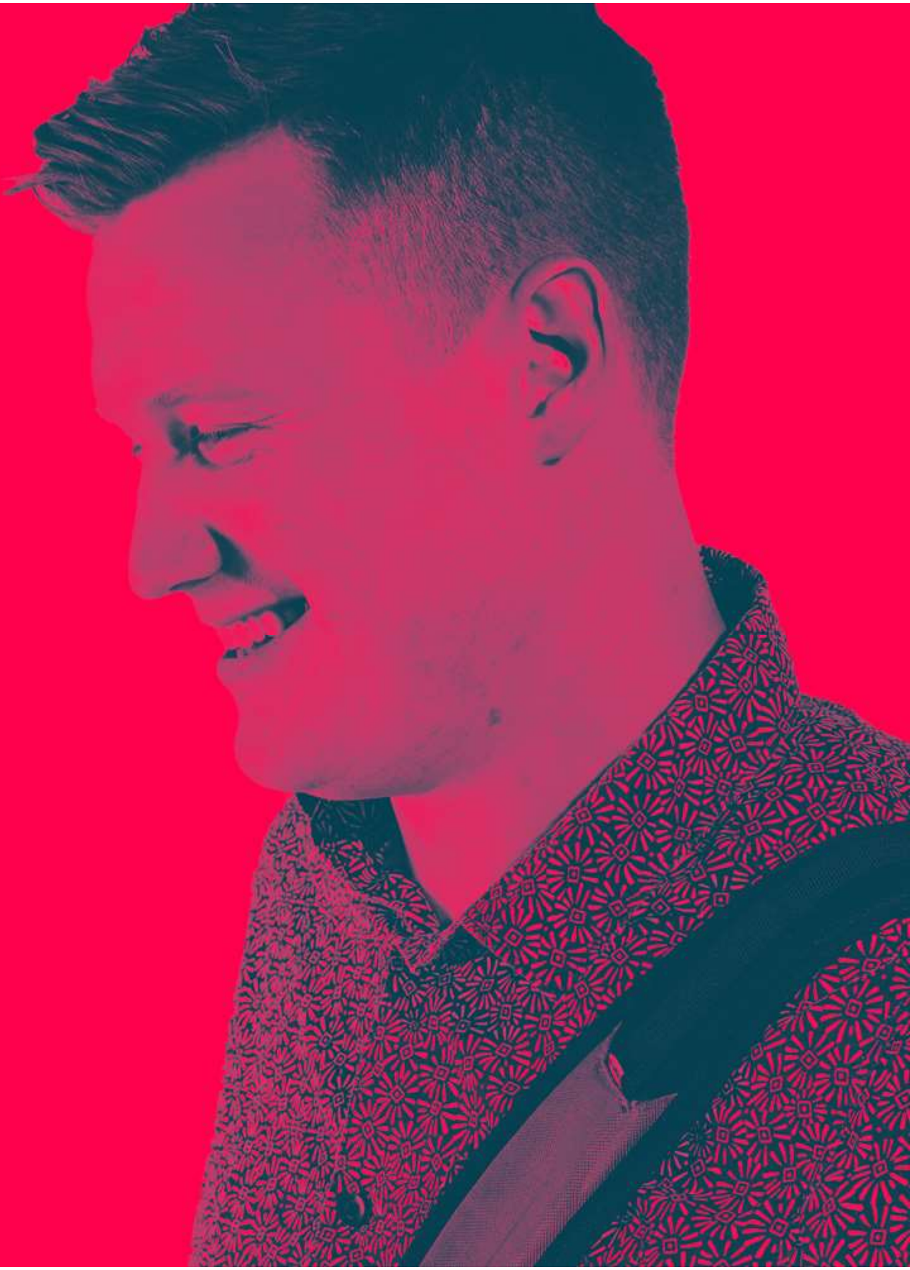
- Developers can use a **Cucumber** engine to run Gherkin. This engine may link to a framework to talk to browsers, simulators etc.
- Mobile developers use tools like **Appium**
- Web developers can use **Selenium** to hook into browsers. Step definitions run commands via Selenium Api.

LAB

"Behaviour Driven Development (BDD)" Lab

Afterwards, we'll discuss what you thought...

CODE SMELLS



COMMON CODE SMELLS

Long methods
Make code hard to maintain and debug – consider breaking up into smaller methods

Refuse bequest
When a class inherits from a base class and doesn’t use any of the inherited methods

Data clumps
When multiple method calls take the same parameters

Duplicate code
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.

UNCLE BOB'S SOLID PRINCIPLES

S - Single Responsibility Principle

O - Open-Closed Principle

L - Liskov Substitution Principle

I - Interface Segregation Principle

D - Dependency Inversion Principle

LAB

"Code Smells" Lab

Afterwards, we'll discuss what you tried...

CODING STANDARDS AND PRACTICES

THE VIRTUES OF A PROGRAMMER

Laziness

- Hates answering the same questions over and over, so writes good documentation
- Hates reading documentation, so writes code clearly
- Writes tools and utilities to make the computer do all the work

Impatience

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

Hubris

- Has pride in programs that no one will criticise

THE GOLDEN RULE

- Follow the standards of your organisation

- Ask to see the coding standards / guidelines

Then be sure your code always uses:

- Consistent naming
- Effective commenting
- Proper and effective code formatting

GOOD PRACTICE

1. Remember ‘Rubbish in – rubbish out’.
2. Choose the smallest data type for the job.
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.
10. Make sure that classes have a single responsibility, and identify what that is.
11. Automate your tests!

NAMING CONVENTIONS

Rules for the naming of identifiers:

- For data types - structures, classes, etc.
- For data items - constants, variables
- For code fragments - functions, methods, libraries

Naming conventions makes code easier to read:

- Easier to distinguish your items from those of a 3rd party
- Helps avoid clashes with reserved words

Data variables and constants should be clearly named:

- Functions should state their intent
- Avoid vague functions names like `calculate` preferring more specific `calculate_invoice_total`

COMMENTS

- Always comment the intent of your code
- Don't comment self evident coding structures
- Make sure you comment work-arounds and quick fixes
- Always update comments when you update code
- Remove comments from scripts before release

FORMATTING

Good formatting makes code clearer!

- Format your code to allow your code to naturally flow
- Languages like Python use indentation to specify blocks
- C Based languages use braces

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

READABILITY AND STYLE

- More time is spent on maintenance than development
  - Document what you do
  - Code that is obvious today is not obvious tomorrow
- Avoid 'clever' one-liners
  - They are rarely faster, sometimes slower
  - Often difficult to debug
- KISS - Keep It Simple and Straightforward

ERROR HANDLING

- If anything can go wrong - it will
  - 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

PROGRAMMING FOR CHANGE

- Defensive programming
  - Where changes are less likely to cause problems
  - Includes good naming conventions
  - Clarity of style makes the code easy to read
- Making your program flexible
  - Avoid artificial fixed limits
  - Make no assumptions on variable type sizes, word length, etc.
  - Adhere to standards as much as possible
    - Using language extensions ties your program to one vendor
- All this makes changing a program less work
  - Great for the virtue of laziness!

HELP!

- Many languages have style checkers and rules
  - 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
- Code analysis tools are often based on Lint
  - An old C-based tool, now much enhanced
  - Microsoft Visual Studio includes Code Analysis tool FxCop
- Some editors and IDEs assist with style as you type

LAB

"Coding Style Guides" lab

Afterwards, we'll discuss your thoughts...

CONTINUOUS INTEGRATION (CI)

WHAT IS CI?

- Building software and exercising it with as many tests as possible
- Developers work on a new feature
- It is added to the main code base automatically
- Test driven development methodologies used to verify feature works and doesn’t break anything
- Automating the build process

BENEFITS OF CI

1. Frequent automated testing.
2. Build and deploy code.
3. Increased transparency.

BEST PRACTICES

1. Maintain a single source repository
   - No more emailing code around!
   - Should contain everything required for the build including test and compilation scripts
2. Everyone commits to the mainline every day
   - Commit each change when it works

Adapted from Martin Fowler's article Continuous Integration

CI TOOLS
(IN NO PARTICULAR ORDER)

Jenkins - a free, open-source too with a very active community

circleci - free tiers available, charged options for volume users

TeamCity - maintained by the JetBrains team, it has free and paid-for services

Bamboo - part of the Atlassian suite of development tools and services, a paid-for service

GitLab - can be configured to use their services or on your own choice (in-house or cloud-based), with free and paid-for services

TravisCI - free for open-source project, private projects are chargeable

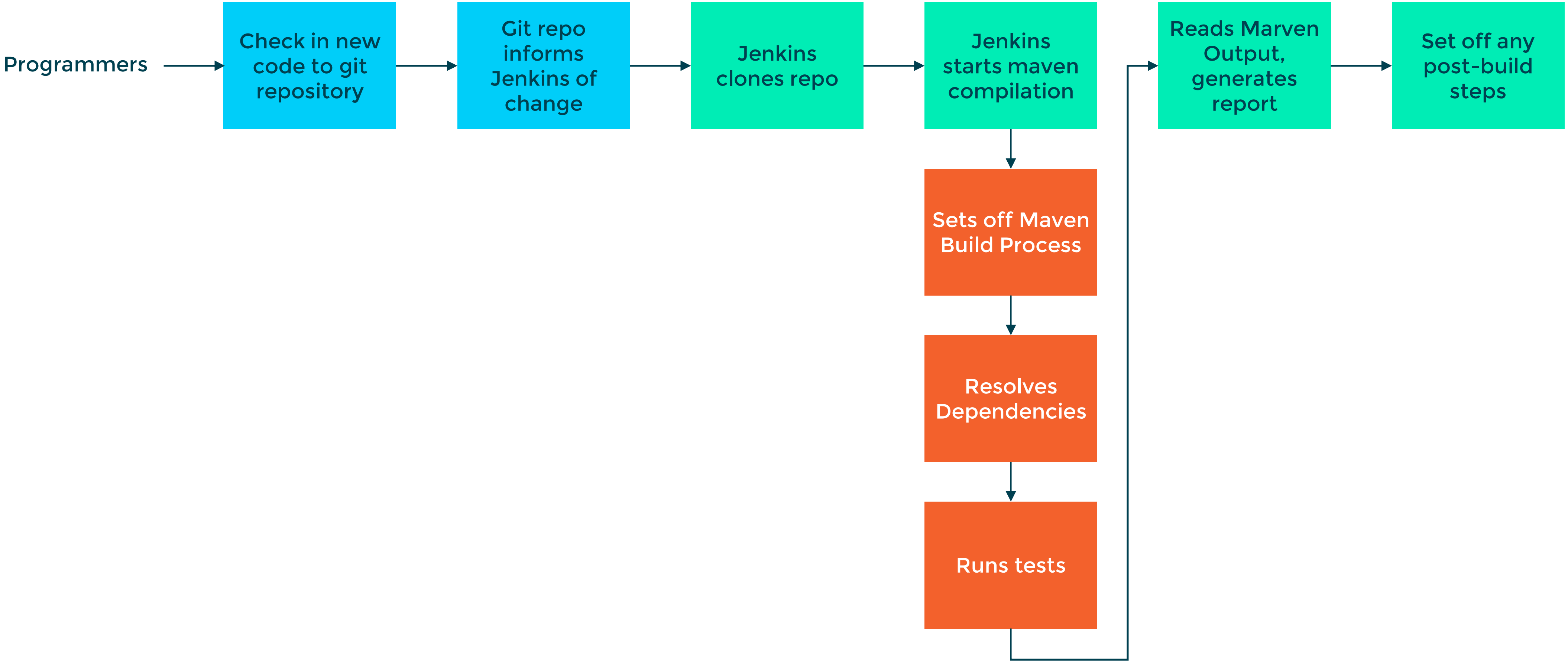
JENKINS CONCEPTS

INTRODUCING JENKINS

Jenkins is a one stop shop for testing and building code. It can even deploy code onto servers if they already exist.

- First released in 2005 as the Hudson project
- Created by Sun Microsystems
- Oracle bought the company and so had control of the Hudson name
- Name changed to Jenkins for the open source project
- Both the Hudson and Jenkins project are still active but development has diverged
- Created as a CI / CD tool
- Aim was to be easy to install and use
- Plugins to connect to source control
- Builds projects using the project setup – so can work with any language
- Public/company facing dashboard lets everyone know the status of the current build

Jenkins Workflow



Programmers

Check in new code to git repository

Git repo informs Jenkins of change

Jenkins clones repo

Jenkins starts maven compilation

Sets off Maven Build Process

Resolves Dependencies

Runs tests

Reads Marven Output, generates report

Set off any post-build steps

BENEFITS OF JENKINS

- Allows companies to follow some of the best practices for CI/CD
  - Automate the build
  - Make the build self testing
  - Keep the build fast
  - Make sure everyone can see the results of the build
  - Automate deployment
- Easy installation and config
  - Can be done all through the GUI – no need for more XML files
- Many plugins available for any combination of source control, compiling, testing and deployment
  - Still a very active project!
  - IDE Integration, orchestration tools, style checkers

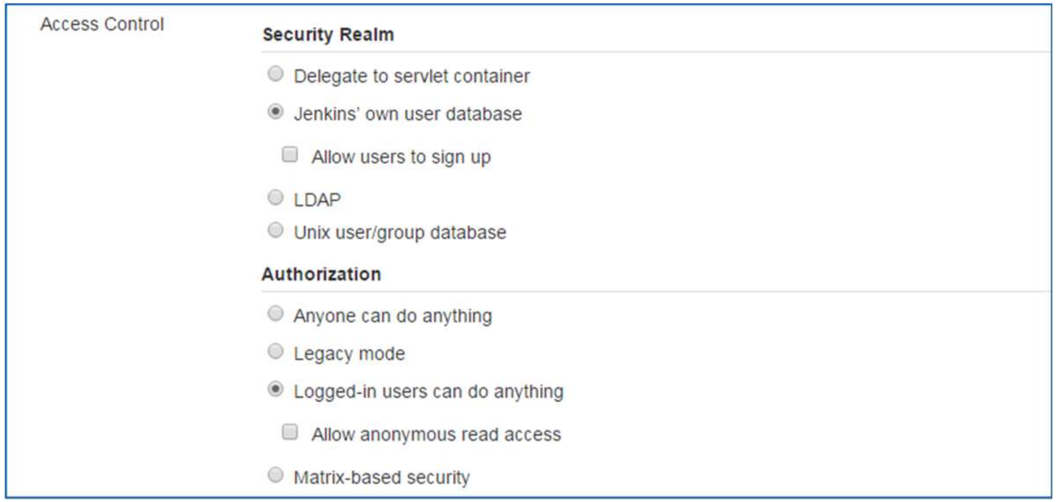
DISTRIBUTED JENKINS SERVERS

- Jenkins uses agents to scale systems
  - By default there is one agent which will build using one core
  - May need to build for different architectures
    - Windows / Mac / Linux
    - Android / iOS / Windows Phone
- Master / Node servers
  - Can exist anywhere
- Single report can be generated over all the builds

SECURING JENKINS

- Security is enabled by default in Jenkins 2.0+
  - Originally, Jenkins would allow anyone to create and build jobs

- You want to restrict who can trigger new builds and setup new jobs on your server
  - Otherwise random people on the internet will be able to use your build server!



Access Control

Security Realm

○ Delegate to servlet container

● Jenkins' own user database

☐ Allow users to sign up

○ LDAP

○ Unix user/group database

Authorization

○ Anyone can do anything

○ Legacy mode

● Logged-in users can do anything

☐ Allow anonymous read access

○ Matrix-based security

SECURING JENKINS

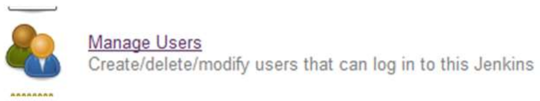
Access Control

- Jenkins can hook into existing systems
  - Unix user groups
    - Either Jenkins needs to run as root or User jenkins needs to belong to group root and chmod g+r / etc / shadow needs to be done to enable Jenkins to read /etc / shadow
- LDAP
  - Lightweight Directory Access Protocol
  - Use existing logins for Windows
- Jenkin's own user database
  - We can allow users to sign up or restrict this
  - When you first access Jenkins it will let you set an initial username and password
- Authorisation
  - Logged in users can do anything – covers most cases
  - You can enable project based security if required

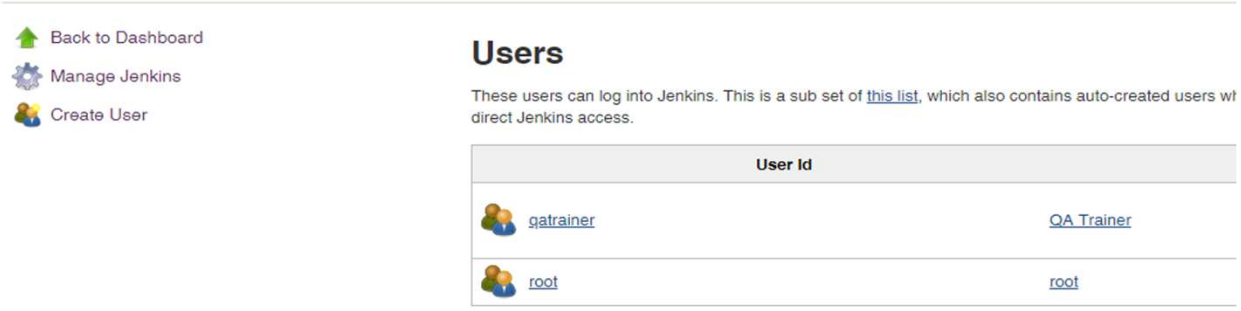
SECURING JENKINS

Manage Users

Add users manually from the Manage Jenkins menu



Manage Users
Create/delete/modify users that can log in to this Jenkins



Users

These users can log into Jenkins. This is a sub set of this list, which also contains auto-created users wh
direct Jenkins access.

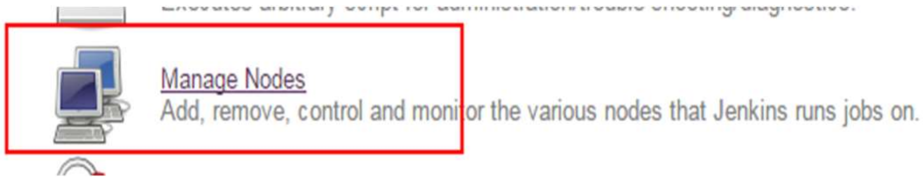
User Id	
qatrainer	QA Trainer
root	root

BENEFITS OF DISTRIBUTED JENKINS

- Jenkins runs builds using different agents, or threads on a machine
- When we have large projects we may need more capacity
- Jenkins offers different ways to manage nodes
- One server can handle multiple projects
- You can specify jobs to target specific machines
- Not a lot of overhead for setting up
- Jenkins always follows some rules when working out which node to use
- The master can monitor the health of any connected node

SETTING UP THE MASTER

- From the main dashboard
  - Manage Jenkins
  - Manage Nodes



Executes arbitrary script for administration/trouble-shooting/diagnostics.

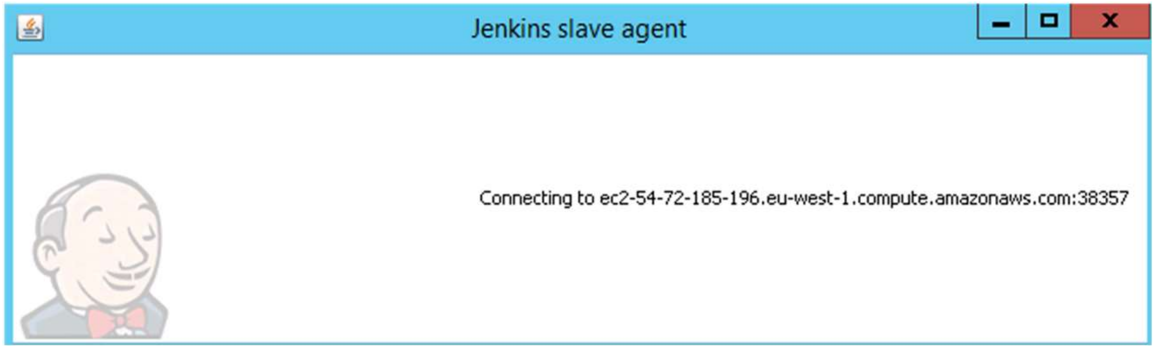
Manage Nodes

Add, remove, control and monitor the various nodes that Jenkins runs jobs on.

- Add a new node
  - Your first will always be a permanent node
  - Others can copy this configuration if they want
  - Number of executor threads can be set
  - You need to specify a directory for Jenkins to use on the node
    - `/home/ec2-user/jenkins`
    - `C:\Jenkins`
- Usage – say how much you want to use the node
- Launch Method – How you want to control the node
  - Java web start is the easiest for Windows hosts

WINDOWS NODES

- We can run the node program via Java Web Start or through installing a service
  - Setup the node on the master
  - Go to the link given by the master on the remote machine
  - Click the Java Web Start button
- If you already have Jenkins installed as a service we can connect directly to the machine using a username and password
  - Not recommended – prone to bugs
  - Better to use the Java Web Start



Connecting to ec2-54-72-185-196.eu-west-1.compute.amazonaws.com:38357

LINUX NODES

- Needs to have SSH password access enabled
  - Make sure you install Git!

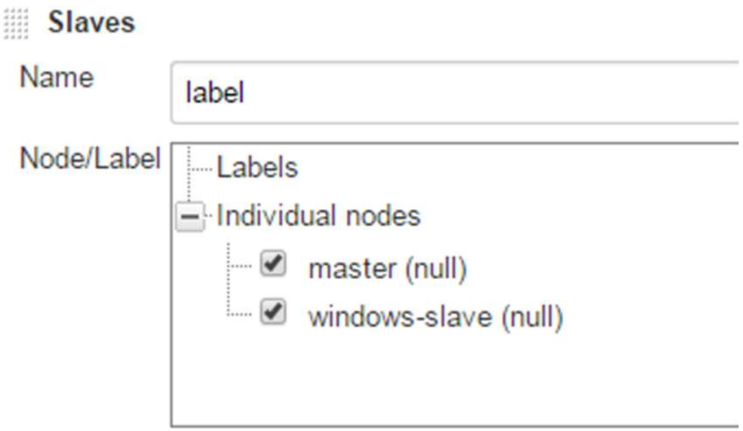
```
#Add user
$ sudo adduser jenkins
$ sudo passwd jenkins

#Change the SSH login – Set PasswordAuthentication to yes
$ sudo vim /etc/ssh/sshd_config

#restart the service
$ sudo service sshd restart
```

- Then when you create a node give it the username, password and IP address of the host

Setting up a Distributed Build

Collating the results

- We can have multiple jobs for one source repository
  - Define which machines to build on
  - Separate success/failures
- Combine them with multi-configuration (Matrix) projects
  - Allow you to build the same project using more than one node
    - Concurrent or consecutively
  - When configuring the project you get to decide which nodes to use
  - Jenkins will take care of the rest – including setting up maven (and the JVM if you have an oracle account) on the target machines



Slaves

Name: label

Node/Label:
- Labels
- Individual nodes
  - ☑ master (null)
  - ☑ windows-slave (null)

JENKINS BEST PRACTICES

- Secure and backup Jenkins regularly
- Use file fingerprinting to manage dependencies
- Enable the build to be pulled directly from git
- Integrate tightly with your issue tracking system
- Integrate tightly with a repository browsing tool if you’re using Subversion
- Configure your job to generate trend reports and automated testing
- Set up Jenkins on the partition that has the most free disk-space
- Archive unused jobs before removing them

JENKINS BEST PRACTICES

- Setup a different job / project for each maintenance or development branch you create
- Allocate a different port for parallel project builds
- Set up email notifications mapping to all developers in the project
- Report failures as soon as possible
- Write jobs for your maintenance tasks, such as cleanup operations, to avoid full disk problems
- Tag, label, or baseline the codebase after the successful build
- Configure Jenkins bootstrapper to update your working copy prior to running the build goal / target
- In larger systems, don't build on the master

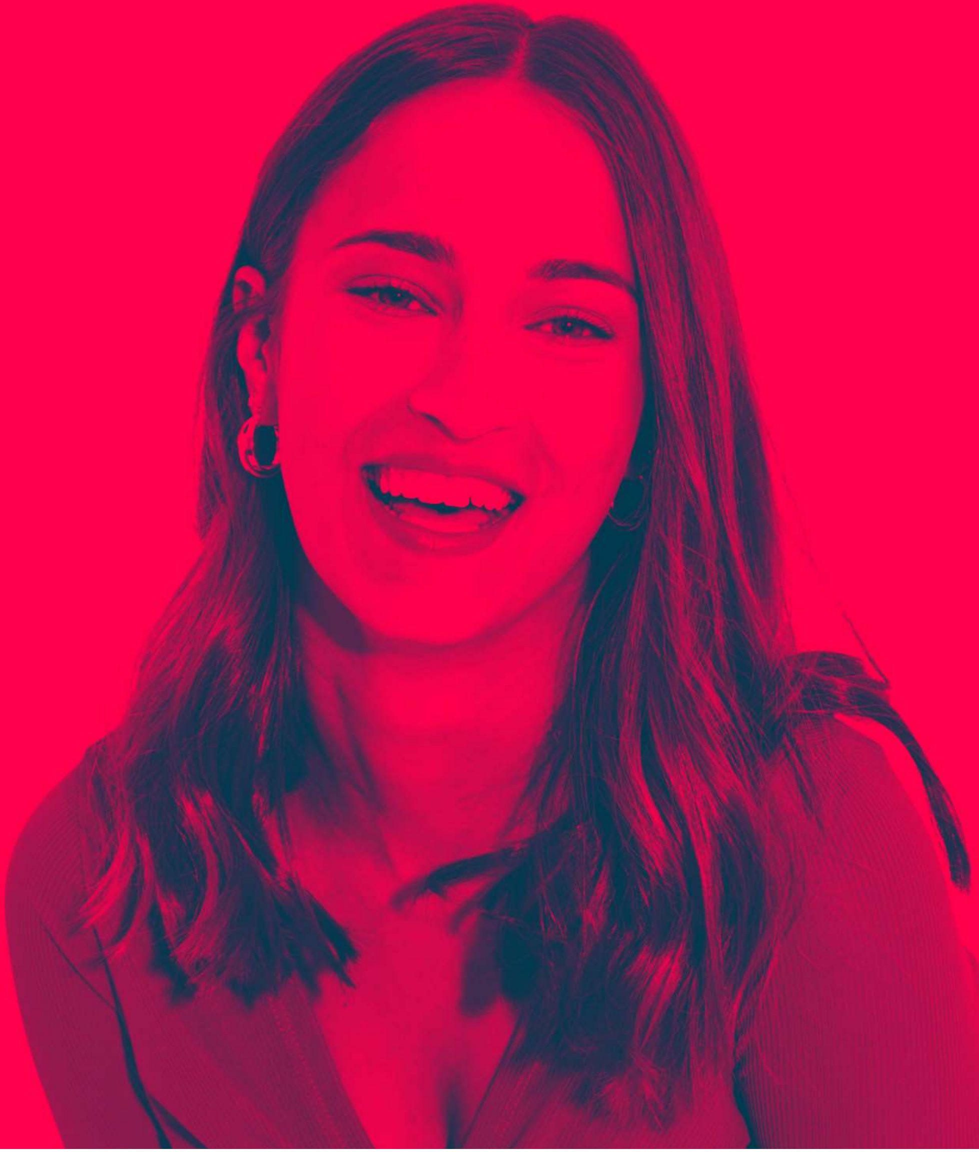
LAB

"Continuous Integration with Jenkins" lab

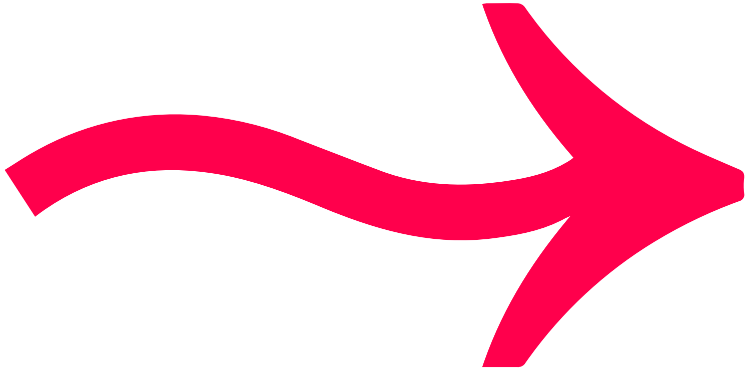
Afterwards, we'll discuss how it went...

POWERING
POTENTIAL

CONTINUOUS DELIVERY AND DEPLOYMENT



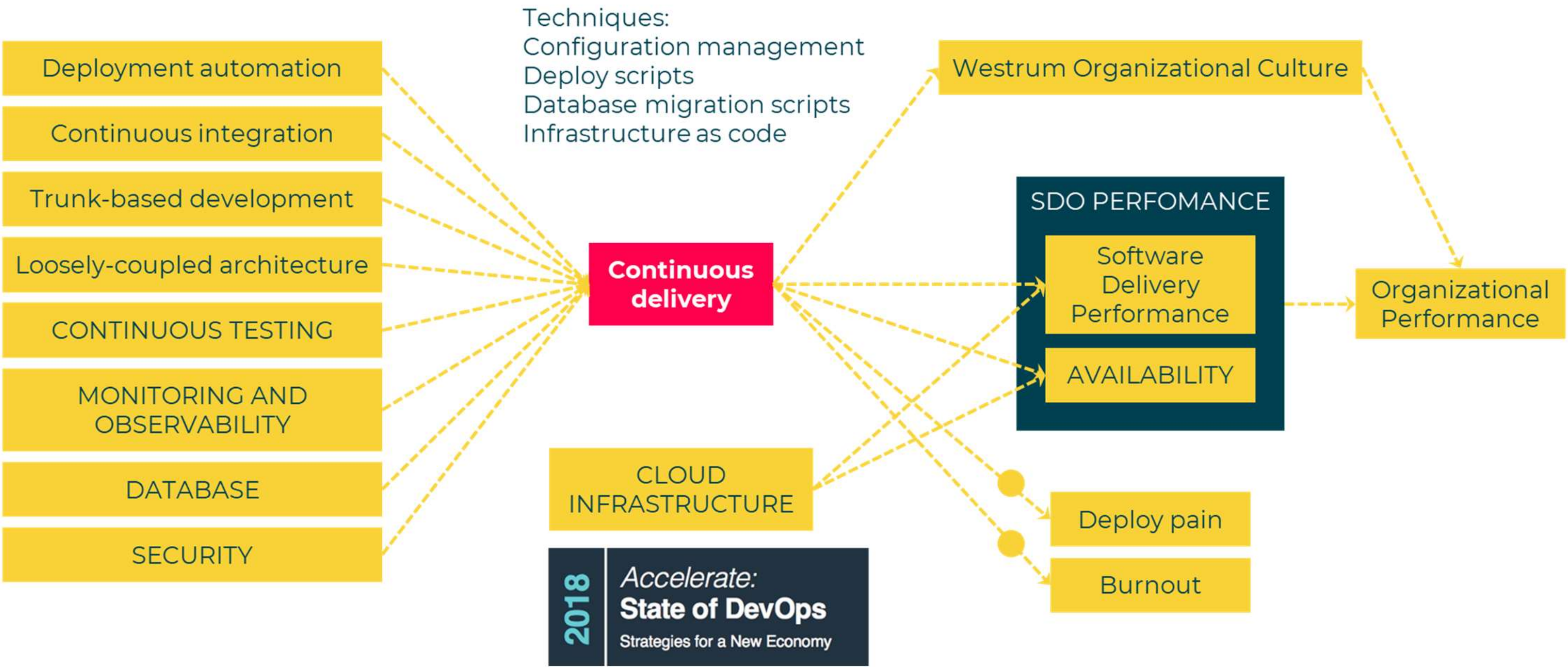
What is Continuous Deployment / Delivery?



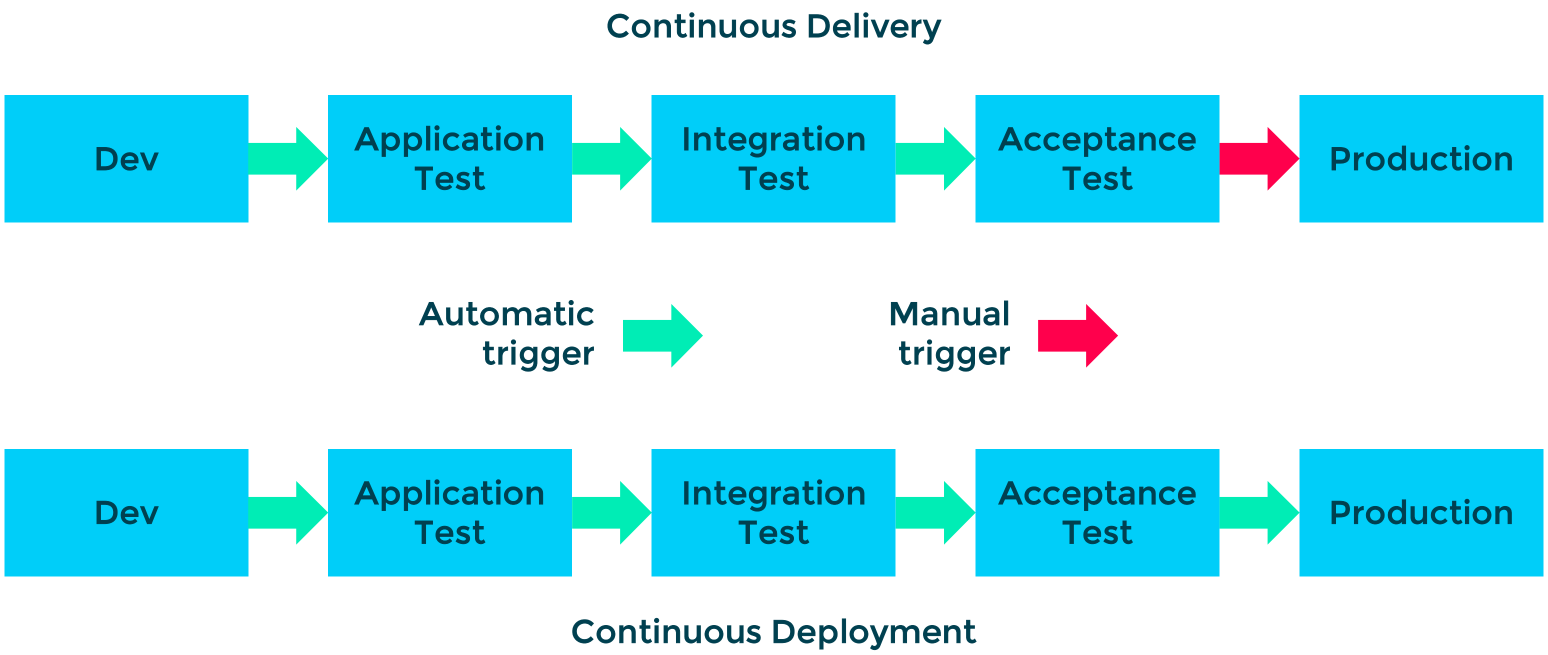
- A software engineering approach that allows teams to produce software in short cycles
- Successful, repeatable deployment of code
- Deployment to server after the build has run and been verified
- Automation of the server setup / connection
- Minute by minute code releases, rather than months at a time

Benefits of Continuous Delivery

CD can lead to higher IT and business performance.



Continuous Delivery vs Continuous Deployment



Continuous Delivery

Dev → Application Test → Integration Test → Acceptance Test → Production

Automatic trigger →

Manual trigger →

Dev → Application Test → Integration Test → Acceptance Test → Production

Continuous Deployment

From: Mirco Hering: notafactoryanymore.com, author of ‘DevOps for the Modern Enterprise’

CD and DevOps Culture

Culture Requirements for DevOps

Collaboration
- Supporting interactions between people

Affinity
- Relationships between teams and across the industry

Tooling
- Effective use of tools accelerates culture effects

Scale
- Organization changes throughout their lifecycles

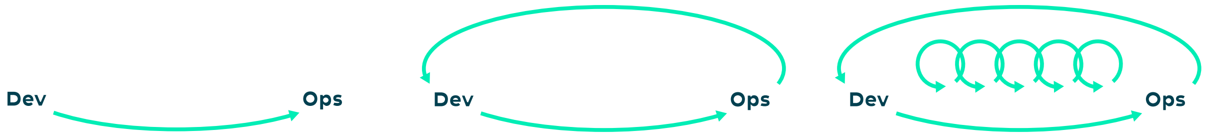
Three Tiers of Culture (CD)

Enterprise Executive
- Executive leadership culture affects the enterprise strategic approaches and decision making processes for continuous delivery

Cross-Functional Teams
- The culture and behaviour of cross-functional teams affects team choices, workflows and practices that are critical to continuous delivery

Design Micro-Teams
- The culture and behaviour of small teams affects workflows and practices that are critical to designs suitable for continuous delivery

THREE METHODS OF 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

WHAT IS DEVOPS?

DevOps allows for a collaborative effort between development and operations, making production smoother. It focuses on:

C - Culture

A - Automation

L - Lean

M - Measurement

S – Sharing

WHAT IS DEVOPS?

DevOps Patterns

- Feature toggles (flags)
- Branch by abstraction
- Dark launching
- Canary release
- Blue-green deployment

DEVOPS PRACTICES

- Self-service configuration
- Automated provisioning
- Continuous build
- Continuous integration
- Continuous delivery
- Automated release management
- Incremental testing
- Version control for all production artefacts
- Continuous integration and deployment
- Automated acceptance testing

DEVOPS CULTURE OF TRUST

Encouraged by…

- Regular peer reviews of any production changes
- Proactive monitoring of the production environment
- Win-win relationship between development and operations

POWERING
POTENTIAL

THANK YOU

Hope you enjoyed this
learning journey.