1. Unit Testing (with Java and Junit)
2. Test Driven Development (with Java)

Technical Excellence 2018

Neueda

Contents

1. Introduction to JUnit
2. TDD and Agile Development
3. Unit Testing with Mock Objects

Neueda

Demos and Practical Exercises

- The course is based around labs and working examples
  - To demonstrate common techniques, and to investigate some more subtle points
- You can take away all the samples and practical code at the end of the course

Neueda

Any Questions?

- If there's anything you don't understand, or anything you want to investigate further…

Neueda

Introduction to JUnit

Neueda

Contents

- Objectives
  - To review JUnit for developers
- Contents
  - Unit testing and the xUnit family
  - JUnit overview
  - JUnit 4 annotations
  - Exception testing
  - How to write good tests
- Exercise

Neueda

Writing Manual Tests

- If you adopt a "manual testing" approach…
  - You write a test harness for the Class Under Test
  - The `main()` method creates an instance of a class, calls methods, and displays results via lots of `System.out.println()`s
- Drawbacks:
  - Not structured; have to hand-craft each time
  - Not necessarily repeatable; may not work in 2 weeks' time
    - Anyone should be able to run the tests at a click of a button, and see straightaway whether they passed/failed
  - Probably don't exercise all the code
  - No standardised reporting
    - Visual inspection of console output; inherently risky (may miss fails)
    - Cannot aggregate tests and aggregate reporting
  - Don't integrate with other tools (build process, coverage)

Neueda

In short: there is *no framework* (Roy Osherove, *The Art of Unit Testing*, p. 23)

Acronyms sometimes used:

SUT: System Under Test

CUT: Class Under Test

Unit Testing

- A Unit can be
  - a method
  - a database query, stored proc, or transaction
  - a dynamic web page
- A Unit Test:
  - Tests one behaviour that is expected from an object
  - Is automated, self-validating, consistent/repeatable, independent, readable, easy to maintain, and fast
- The xUnit Framework
  - JUnit and TestNG for Java
  - NUnit for .NET
  - Test::Unit for Perl …

Neueda

Unit Testing is also known as Component or Module Testing. The concept originated in the 1970s, and the initial concept of what constitutes a “unit” was probably quite open-ended. It is a form of "White Box" testing – in other words, where 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.

Robert Martin (*Clean Code*, Ch. 9) captures the modern conception of unit tests in the acronym FIRST:

- **F**ast: they need to be fast so that developers will run them very frequently
- **I**ndependent: one test must not be dependent on, or be affected by, any other test
- **R**epeatable: they must be repeatable in every environment: development, QA, production
- **S**elf-Validating: they should either pass or fail automatically, not require any manual intervention or manual inspection of output logs
- **T**imely: they should be written in a timely fashion (just before the application code – i.e. TDD)

JUnit: Principal Java xUnit framework

- Developed by:
  - Kent Beck (Extreme Programming - XP)
  - Eric Gamma (Design Patterns)
- There are two versions in common usage:
  - JUnit 3 (main package: `junit.framework`)
  - JUnit 4 (main package: `org.junit`) – we'll be using this
- How to run:
  - Command line
    - `java org.junit.runner.JUnitCore my.pkg.AllTests`
  - IDE: Eclipse, IntelliJ, NetBeans
  - Standalone GUI: AWT, Swing – JUnit 3 only
  - Build scripts: Ant task, Maven goal

Neueda

Another way the essence of the xUnit framework can be captured:

- Arrange objects: create and set up as required
- Act on an object: invoke the method being tested
- Assert: what is expected

Example CUT (Class Under Test)

```
public class Person implements Comparable<Person> {

    private String givenName;
    private String familyName;
    private int age;

    // 3 arg constructor, getters, …

    @Override
    public int compareTo(Person other) {
        int otherAge = other.age;
        return this.age - otherAge;
    }

    @Override
    public String toString() {
        return familyName + ", " + givenName;
    }
}
```

Neueda

A very common operation is performing a sort on a collection of objects. Typically, sort is implemented in terms of making pairwise comparisons of objects in the collection: to arrange a list in order we need to know how any two objects in the list compare. We need to define a comparison relation which will behave like this:

For an arbitrary pair of objects, for the notion of ordering we are defining:

- If the first comes before the second: return a negative number (e.g. –1)
- If the two are equivalent: return 0
- If the first comes after the second: return a positive number (e.g. +1)

The `Person` class here implements `Comparable<Person>` – in other words, it commits to defining an inherent notion of what it is for one `Person` to be before another if we were to sort them. We are building an inherent notion of order into the class, by implementing `compareTo()`. This definition is in terms of age:

- If the other person is older, return negative
- If the other person is the same age, return 0
- If the other person is younger, return positive

Generating a Test Stub in Eclipse

- Eclipse makes it easy to generate test stub automatically for a CUT
  - Right click CUT -> New -> JUnit Test Case
  - 1st dialog: which general JUnit methods do you want?
    - `setUp()` is usually sufficient
  - 2nd dialog: which methods do you want to test?
    - Just select `compareTo()` here

```
public class PersonTest {

    @Before
    public void setUp() throws Exception {
    }

    @Test
    public void testCompareTo() {
      fail("Not yet implemented");
  }
}
```

Neueda

Follow the steps in the slide, to automatically create a test class for a Class Under Test in Eclipse.

The example shown in the slide is for JUnit 4, whereby test methods are identified by `@Test` annotations. In JUnit 3, test methods are identified by a special name – any method name that starts with `test` is assumed to be a test method.

Additional techniques:

- You can generate a CUT from test class, which is a useful approach when strictly following TDD. See later for details!
- You can generate a blank test class (just complete the 1st dialog above).

Java Language Features used in JUnit 4

- Annotations

```
@Before
public void setUp() {
}
```

```
@Test
public void testCompareTo() {
}
```

- Suppressing warnings

```
@SuppressWarnings(value= {"serial"})
```

- Static imports
  - Import static members from another class
  - Allows you to use member name directly, without classname prefix

```
import static java.lang.System.out;
import static java.lang.System.currentTimeMillis;
 …
out.println(currentTimeMillis());
```

Neueda

JUnit 4 uses annotations to designate special methods in your test class, such as start-up methods, tear-down methods (see later), and the test methods themselves. You can also use annotations in your own code, such as:

- `@Override` – indicates that a method overrides a method in the superclass.
- `@SuppressWarnings` – tells the compiler not to generate specific types of error messages, which can be useful during the early stages of development.

JUnit 4 also makes extensive use of static imports. With static imports, static members of a class – fields and methods – can be imported directly into another class, making the code cleaner.

JUnit Assert Methods

- JUnit defines lots of assert methods
  - These are like if-tests, they verify a condition is true
  - In JUnit 4, they are `static` members of `org.unit.Assert`
- The assert methods are heavily overloaded… for example:
  - `assertEquals(boolean expected, boolean actual)`
  - `assertEquals(Object exp, Object act)`
  - `assertEquals(String msg, Object exp, Object act)`
- Additional useful methods:
  - `assertSame()/assertNotSame()`
  - `assertTrue()/assertFalse()`
  - `assertNull()/assertNotNull()`
  - `fail(String message)`

Neueda

`assertEquals()` is overloaded for all the primitives, plus `Object`, and `String` – so 10 versions – then double that for the three-argument version which starts with a String message. For the two items compared, always give the expected one first, then the computed one, for this is the order JUnit will use in error reporting (as in "java.lang.AssertionError: expected:<7.0> but was:<8.0>").

Always use the String version of an `assert()` method, so that the reason for failure will be stated if the assert fails.

For full details about all the methods in the `Assert` class, see:
`http://junit.sourceforge.net/javadoc/org/junit/Assert.html`

Example JUnit 4 Test Class

```
import org.junit.*;
import static org.junit.Assert.*;

public class PersonTest {
  Person tom, dick, harry;

  @Before
  public void setUp() throws Exception {
    tom = new Person("Tom", "Smith", 29);  // Ditto for other fixtures
  }

  @Test
  public void testCompareTo() {
    assertEquals("Tom same age as Dick ", 0, tom.compareTo(dick));
    assertTrue("Tom younger than Harry", tom.compareTo(harry) < 0);
    assertTrue("Harry older than Dick",  harry.compareTo(dick) > 0);
  }

  @Test
  public void testToString() {
    assertEquals("Smith, Tom", tom.toString());
  }
}
```

Neueda

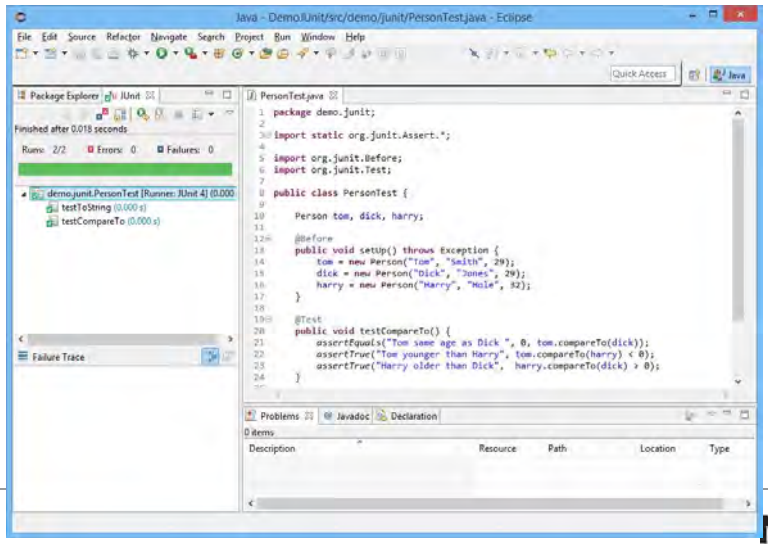
Here's a simple test class, to test our `Person` class from earlier in the chapter. Note the following points:

- We declare several `Person` variables, to facilitate our testing.
- The `setUp()` method is annotated with `@Before`. The code in this method will be executed *before each test* method in our test class. Note that the actual name of this method is unimportant – what matters is that it's annotated with `@Before`.
- The test methods are annotated with `@Test`. There are 2 test methods in this simple example – a real test class might contain dozens of test methods. Note that the actual names of the test methods is unimportant, from a technical point of view – it's the `@Test` annotation that makes them "test methods". (We'll return to the subject of choosing meaningful names for test methods later).
- Our test methods use various `assertXxx()` methods. Remember, these are static methods in the `org.junit.Assert` class.

If multiple classes have common setUp()/tearDown() code, abstract this out to a common superclass. Don't put any `@Test` methods in this superclass (else they will be inherited in each sub-class and therefore run multiple times). The common `@Before` method will be called before any `@Before` methods in a subclass. Naturally, if the method is called `setUp()` in both the sub and super class, the former will override the latter. A class can have multiple `@Before` methods (say `setUp()` and `init()`), but apart from this case of inheritance, it would not be a good design to have two set-up methods defined in the same class.

Running a Test Class

- Right-click in the test class, and select Run As > JUnit Test
  - JUnit displays the outcome of the tests

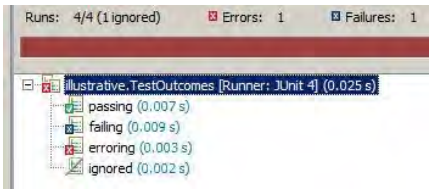


Neueda

The screenshot above shows a successful test outcome – the green bar shows all the tests passed, and the tests are listed below.

Four Statuses of a Test

- passing
  - Ultimately all tests must pass
- failing
  - In TDD, we always start with a test that fails
- erroring
  - Test neither passes nor fails, i.e. a run-time error has occurred
  - E.g. necessary library jar has not been provided
- ignored
  - A test annotated with `@Test @Ignore`



Neueda

The `@Ignore` annotation is more useful than it might first seem. If you know you have to write a test, but can't get round to writing it just yet, you can add this annotation alongside `@Test`. The essential point is that you get a constant reminder, as shown in the screen-shot above, that there is a test that still needs tending to. This is much better than the alternatives:

- If you left off `@Ignore` and just had an empty test body, the test would trivially pass, and you might easily overlook it.
- If you left off `@Ignore` and put a `fail()` in as Eclipse does, that wouldn't be right either.
- If you commented out `@Test`, the method would be there in your test class but it wouldn't show up in your test summary, so you might forget about it.

Note that the `@Ignore` annotation can be qualified with a string reason, as in `@Ignore("WiP")`, etc. Under no circumstances do this kind of thing:

```
@Test
public void testXyzMethod() {
   // TODO: how do we test this?
   assertTrue(true);
}
```

Exception Testing 1: Explicit try/catch

- If you want to test a method that throws an exception…
  - One approach is to wrap the call in a `try/catch` block, and deliberately cause the exception to occur
  - You can then assert that the expected exception occurred

```
@Test
public void addTime_InvalidInput_ThrowsWithMessage() {

  try {
    time.addTime(24, 20, 2, 30);
    fail("An exception should have been thrown");
  }
  catch(IllegalArgumentException actual) {
    assertEquals("Bad time: 24:20", actual.getMessage());
  }
}
```

Neueda

Note the explicit `fail()` assertion after the call to the tested method: if execution reaches this line of code, the test must be made to fail – because in this scenario, an exception should have been thrown, so we should not reach that point.

Exception Testing 2: Annotation-based

- JUnit 4 offers another way to test exceptions…
  - In the `@Test` annotation, set the `expected` property to the type of exception you expect to occur
  - The test succeeds if and only if the expected exception occurs

```
@Test(expected=IllegalArgumentException.class)
public void addTime_InvalidInput_Throws() {
  time.addTime(24, 20, 2, 30);
}
```

- Advantages of the annotation approach:
  - Simpler and cleaner than `try/catch` blocks
  - Good for initial exception tests

- Disadvantages of the annotation approach:
  - Doesn't give you access to the actual exception object…
  - So you can't access exception info (message, inner exception, etc.)

Neueda

Annotation style is much cleaner, and so is useful for an initial “getting things set up” test. But the explicit `try/catch` structure is still necessary if you need access to the exception object and its properties. For instance, a standard constructor for an `Exception` is one which takes another `Exception` object which is its cause (this is also known as "inner exceptions"). This allows you, as a developer, to define a higher level exception class to wrap a lower-level one, or a run-time exception to wrap a checked exception. Thus, getting access to the expected exception object allows us to make appropriate assertions about it.

If you want to test a method that throws a *checked* exception (rather than a *runtime* exception), an annotation-based test method can simply declare that it throws the exception as follows (note the `throws` clause at the end of the method signature):

```
@Test(expected=FileNotFoundException.class)
public void testFileInput() throws FileNotFoundException {
  File f = new File("nonexistant.txt");
  new FileInputStream(f);
}
```

A `@Test` with the `expected` property is a rare example of a test that does not need an explicit assertion. The key point is that a test should always have conditions under which it would fail (otherwise it is not really *testing* anything), and in this case failure to throw the expected exception type is the failure condition.

How to Write Good Tests (1 of 2)

- DON'T: don't write a test method that attempts to test all aspects of a method in one go!
  - E.g. don't write a test method such as `testCompare()`
  - … that contains a whole bunch of asserts to test every aspect
  - … because if the first assert fails, the others won't be exercised!
- DO: write a test method that tests a specific behaviour
  - Use a naming convention such as <methodName>_<Scenario> or <methodName>_<StateUnderTest>_<ExpectedBehaviour>
  - e.g. `isValidFileName_ValidFile_ReturnsTrue()`

Neueda

JUnit 4 encourages a subtle but important shift of emphasis in the way we conceive of tests. The naming convention here follows Osherove's proposal *The Art of Unit Testing*, p. 29. Whatever naming style you follow, what's important is that test method names can't be too long.

One popular style is to structure tests into a Given/When/Then format. Bob Martin (Clean Code, Ch. 9) gives this example of breaking up a long test method:

```
@Test
public void getPageHierarchyAsXml() {
  givenPages("PageOne", "PageOne.ChildOne", "PageTwo");
  whenRequestIsIssued("root", "type:pages");
  thenResponseShouldBeXml();
}
```

How to Write Good Tests (2 of 2)

- DO: Make unit tests as fine-grained as possible
  - Test one object, one behaviour at a time
  - Keep tests independent of each other
- DO: Provide a `toString()` method in the CUT
  - Good: "expected:<30/5/2011> but was:<31/5/2011>"
  - Bad!!!: "expected:<pkg.Date@785> but was:<pkg.Date@786>"
- DO: Watch out for tests introducing their own complexity
  - How do we know the test is correct?

Neueda

Here are some more recommendations on how to write good tests. We strongly advise you to follow these guidelines!

Summary

- JUnit is the most popular tool for unit testing in Java
- JUnit 4 uses `@Test` annotations (JUnit 3 uses `testXxx()` methods)
- JUnit integrates well with Eclipse and ant build scripts

- Remember FIRST – tests must be:
  - Fast
  - Independent
  - Repeatable
  - Self-Validating
  - Timely

Neueda

References

- The Art of Unit Testing - with Examples in .NET
  - Roy Osherove (Manning Publications) ISBN 1933988274
- JUnit in Action
  - Vincent Massol, (Manning Publications) ISBN 1930110995
- Clean Code – A Handbook of Agile Software Craftsmanship
  - Robert Martin, (Prentice Hall) ISBN 0132350882
- http://junit.org
- http://www.planetgeek.ch/wp-content/uploads/2011/02/Clean-TDD-Cheat-Sheet-V1.2.pdf/

Neueda

Any Questions?

Neueda

TDD and Agile Development

Neueda

TDD and Agile Development

- Objectives
  - Explain the motivation, process and core techniques of TDD
- Contents
  - Test Driven Development
  - TDD Process, Strategies, Benefits
  - Common Refactorings
  - Testing Patterns
  - Extreme Programming

Neueda

Traditional Development

- Code – Test – Refactor
- Tests are often an afterthought!
  - "If there's time" / leave to testers
- Problem: High level of defects
  - Lengthy testing phase after a release is frozen
  - Cost of fixing a bug discovered at that stage is far higher than if the bug was caught when introduced into code
- Problem: Poor maintainability
  - Legacy spaghetti code that "works"
  - Can't be touched – fear of breaking

Neueda

Test Driven Development

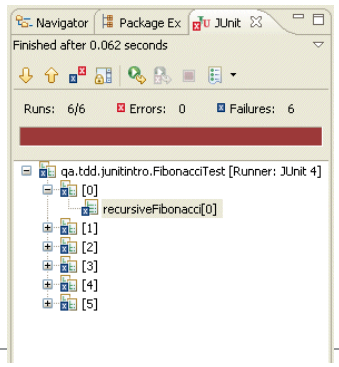
- Core practice of eXtreme Programming (XP)
- Can be adopted within other methodologies
  - "Test-first programming is the least controversial and most widely adopted part of XP. By now the majority of professional Java programmers have probably caught the testing bug" - Elliotte Rusty Harold
- What it means: You write a test before you write the implementation
  - Tools and techniques make TDD very rigorous process
- Aka Test-Driven Design
  - Tests drive the design of the API

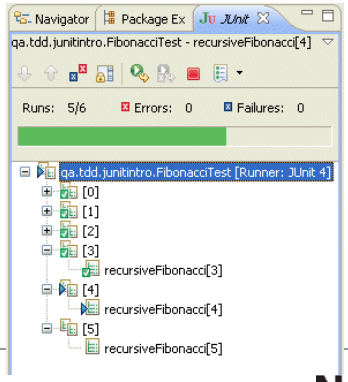
Neueda

Test Driven Development

- Test – Code – Refactor
  - 1. Write new code only if you first have a failing automated test
  - 2. Eliminate duplication

- Red – Green – Refactor





Neueda

The graphic on the right illustrates that the famous green bar is indeed a progress bar. Here we see JUnit running a sequence of six tests; four have completed successfully, the fifth is indicated as currently running, and the sixth is shown as yet to run.

TDD Process

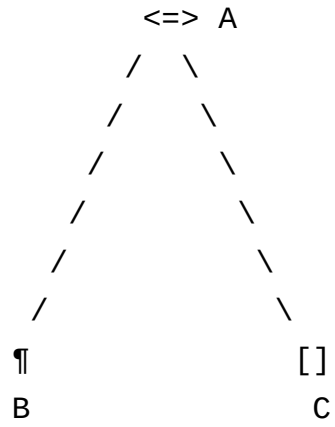
- First write the test
  - Designing the API for the code to be implemented
  - Using an API (in tests) is the best way to evaluate its design
- Write just enough code for the test to pass
  - Minimises code bloat
  - Keeps you focussed on satisfying the requirement of the test
- Refactor: change code without changing functionality
  - "A disciplined technique for restructuring code, altering its internal structure without changing its external behaviour" – Fowler
- Develop in small iterations
  - Test a little, code a little

Neueda

TDD Strategies

- Fake it
  - Return a constant
  - Gradually replace constants with variables
- Obvious implementation
  - If a quick, clean solution is obvious, type it in
- Triangulation
  - Locating a transmitter by taking bearings from 2 or more receiving stations
  - Only generalise code when you have 2 or more different tests
- The point is to get developers to work in very small steps, continually re-running the tests

Neueda



Triangulation: someone on the boat at A can determine their position on a chart by taking compass bearings to the lighthouse at B and the tower at C. 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.

TDD - Benefits

- Build up a library of small tests that protect against regression bugs
  - Tests act as developer documentation
- Extensive code coverage
  - No code without a test
  - No code that is not required
- Almost completely eliminates debugging
  - More than offsets time spent developing tests
- Confidence, not fear!
  - Confidence in the quality of the code
  - Confidence to refactor

Neueda

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

Code Smell

- Code Smell: symptom of something wrong with code
  - Not necessarily a problem, but needs consideration
  - Typically an anti-pattern for a refactoring
- Large class
  - A class that has too much in it; aka God-like object
- Feature envy
  - A class that uses the methods of another class excessively
- Inappropriate intimacy
  - Class A has dependencies on implementation details in class B
- Switch statement
  - May point to better design using polymorphism

Neueda

"Code smell" has become a common phrase in the XP/TDD community, promoted in Martin Fowler's *Refactoring: Improving the Design of Existing Code*. The slide lists four representative examples. *Switch statement* illustrates that these are only indicative; a particular use of a switch statement may be a perfectly appropriate flow control construct.

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
- Many refactoring techniques are automated through Eclipse etc.

- Refactoring ≠ Rewriting

Neueda

According to Koskela:

"**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." (*Test Driven*, p. 106)

Martin Fowler's site `http://refactoring.com` is a useful resource. For example, see the catalogue of refactorings at the following URL:

- `http://www.refactoring.com/catalog/index.html`

Another on-line catalogue of refactorings is available at the following URL:

- `http://industriallogic.com/xp/refactoring`

“Trivial” Refactorings

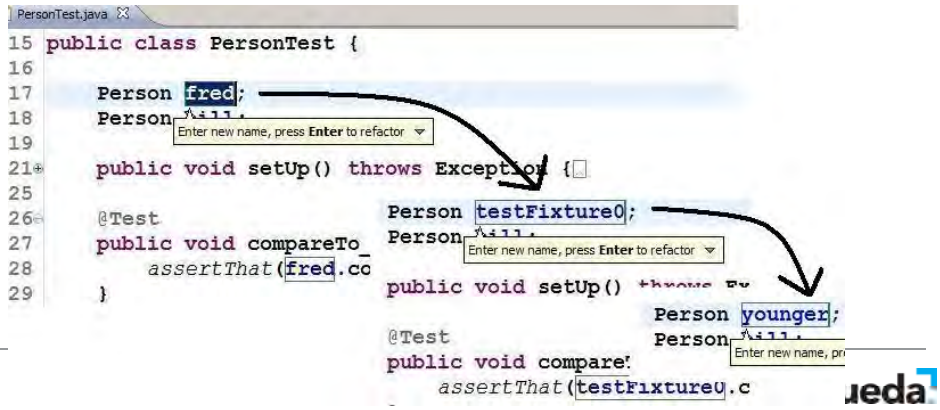
- Koskela: "such low-level refactorings are the fundamental building blocks to achieving larger refactorings. These "larger" refactorings are typically those that deal with moving the responsibilities around in your code, introducing or removing an inheritance hierarchy, or making other similar changes that (usually) involve more than one class. In the end, all refactorings can be reduced into a series of smaller steps, such as renaming a variable"
  - *Test Driven: Practical TDD and Acceptance TDD for Java Developers*, p. 26

- Conway: "Of course, it's not hard to look up the perlfunc manual and learn about the special semantics of substr assignments, so their impact on maintainability is marginal. Then again, almost every maintainability issue is, by itself, marginal. It's only collectively that subtleties, clevernesses, and esoterica begin to sabotage comprehensibility. And it's only collectively that obviousness, straightforwardness, and conformity to standards can help to enhance it. Every small choice when coding contributes in one direction or the other"
  - *Perl Best Practices*, p. 165

Neueda

Common Refactorings (1 of 3)

- Refactor Rename
  - In Eclipse: <ALT> <SHIFT> R
- Keep doing it, until you are satisfied you have an identifier that best reflects what the item represents



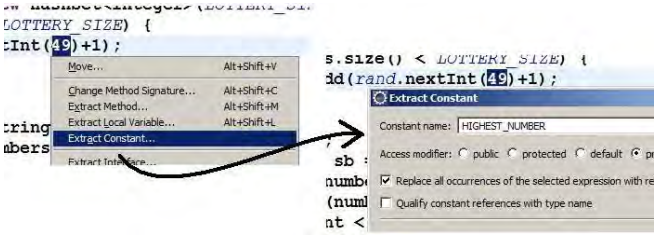
At all times, think how you can make your code maximally self-documenting. Aim to replace unclear and ambiguous identifiers with clear and informative ones. For a variable, choose a name that reflects what it represents in the problem domain, not its implementation (e.g. in the enhanced for loop, `nextClient` rather than `nextElement`). Only choose one-letter identifiers for variables such as array indexes.

Refactor-rename is much smarter than a manual edit of the file:

- Rename a variable: renames all occurrences
- Rename a class: renames the compilation unit too (the `.java` file)
- Rename a method: renames all known calls to that method
- Rename a package: renames all the package declarations within

Common Refactorings (2 of 3)

- Extract Constant – aka “No Magic Numbers”
  - Highlight a literal (e.g. int or String), then <ALT> <SHIFT> T



- Convert Local Variable to Field
  - Highlight a local variable, then <ALT> <SHIFT> T

Neueda

You should avoid hard-coding literal values, like 49, into your code. The essential issue is to do with the comprehensibility of the code: someone coming along to maintain it will be scratching their head wondering what this number represents. There is also likely to be the issue of maintainability : the value will probably occur more than once, and maybe it will need to be revised.

In Eclipse, highlight the number, then <ALT> <SHIFT> T, the general context menu for refactoring. 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: if there are further occurrences of the value within the class, you need to search for them manually and replace them.

Another common refactoring is to convert a local variable into a field (i.e. an instance variable). This could be part of a “remove duplication” refactoring – certainly in the case of a unit test fixture, as illustrated. After you've converted the variable is a class-level field, how should it be initialized? The options are:

i. In the field declaration – i.e. the whole variable declaration statement is effectively pulled out of the method.

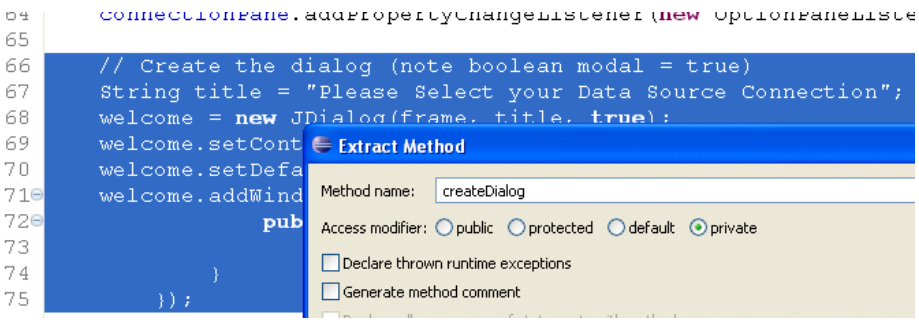
ii. In the current method – i.e. the declaration `User fred;` is pulled up to the class level and the initialisation remains where it is.

iii. In the class constructors – i.e. as ii), but the initialisation statement will be moved to all and any constructors.

None of these options is quite right for a test class, so the best option is to accept ii) and then manually move the initialization to the `setUp()` method.

Common Refactorings (3 of 3)

- Extract Method
  - If a method gets too long (e.g. > 10 lines), pull out part of it
  - Highlight the code you want to extract, then <ALT> <SHIFT> M



- Could be a single line
  - Why is there this call `reader.readLine()` that does nothing?
  - Extract to method: `discardHeaderLine()`

Neueda

One option for a method with many lines of code is to break it into functional chunks with carriage returns ("Code in Paragraphs", as *Perl Best Practices* puts it), and then add a comment to each chunk explaining what it does (as seen in the above image). But Astels for one counts these kinds of comment as a Code Smell (p.19), or later, as merely some deodorant to mask the smell of the long method (p.30).

Methods should not become too long. If you're following TDD, this is unlikely to happen. But just how long is too long? It's not simply about counting the number of statements. Consider the following example:

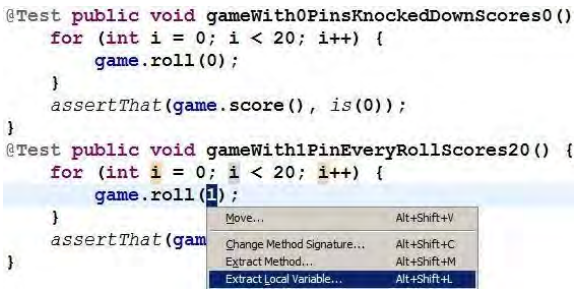
- Imagine you have some code that reads data from a .csv file.
- In this case, it's envisaged that the .csv files all have a header line with the column names, which are irrelevant and should not be read in as data.
- Thus the code contains a statement such as `reader.readLine()`, to skip the header line.

The call to `reader.readLine()` is very important, but its purpose might not be very obvious to other developers. Thus we can refactor the code by extracting the statement and wrapping it up in a well-named method that makes the purpose self-explanatory.

To extract a method in Eclipse, simply highlight the lines in question, then *Refactor -> Extract Method*. If the section of code has a single assignment to a local variable, Eclipse will offer that as the return value of the extracted method.

Remove Duplication

- DRY: Don't Repeat Yourself!
  - E.g. two blocks of code that are almost identical
  - Extract value(s) where they differ to variable(s)
  - Will become input parameter(s) to a single common method



  - Place declaration of local variable `int pins=1` outside the loop
  - Apply Extract Method refactoring to the `for` loop

Neueda

In this example (a variant of the one in 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));
}
```

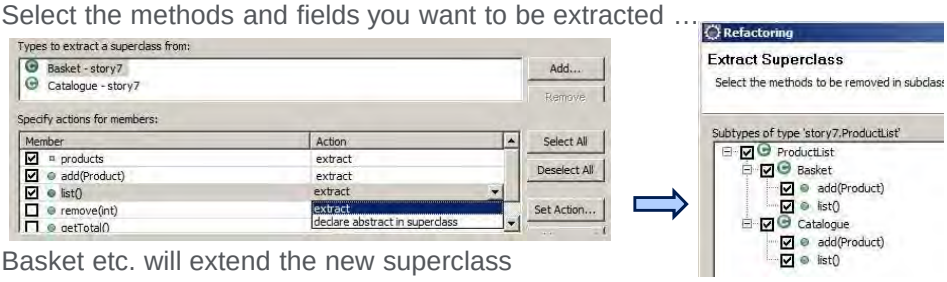
The commonality between the two blocks of code would be captured in the method as follows:

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

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 anathema to good code, because of the risks it introduces for maintenance: the risk of making a change in one place but not another.

Extract Superclass

- Another "Remove duplication" refactoring
- Suppose `Basket` and `Catalog` have commonality
  - Both have a `List` of `Products`, plus `add()` and `list()`
- Choose e.g. `Basket`, and Refactor -> Extract Superclass…
  - Add the other class (in Types to extract a superclass from)
  - Select the methods and fields you want to be extracted …



  - Basket etc. will extend the new superclass

Neueda

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

1. Give a name to the superclass to be created, e.g. `ProductList`. If you've already implemented other classes that you want to inherit from the new subclass, then add them via the *Types to extract a superclass from* list, and "check" the members that you want to be pulled into the new superclass.
2. Explicitly "check" any methods that are to be removed from the soon-to-be-subclasses.
3. Preview the changes.

In performing code reviews, I have come across some real horror stories!

- The identical method in classes in “adjacent” packages (i.e. package a.b.c.m and a.b.c.n)
- Two methods that differ in only a couple of String values
- Two methods that differ in their parameter lists by types ABC and XYZ, where these have a common supertype.
- Two methods or constructors, the bulk of whose definitions are identical, but differ in that one has an extra parameter.

Coding to Interfaces

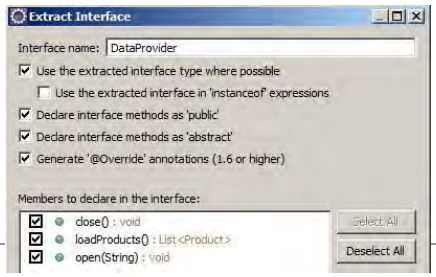
- Suppose a class needs a Repository/DAO

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

- This is a candidate for decoupling interface/impl[n]

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

  - Choose method names that are neutral about data source
  - Choose exception type at the same level of abstraction



Neueda

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 will see declared there all the sorts of operations you could 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 preferable, depending on the sorts of operations you are most likely to want to perform on it. By declaring the `products` variable to be the type of the interface (i.e. `List`), you know that you can perform all the operations you want on it – but if subsequently a different implementation of the concept of `List` turns out to be relevant, you can plug that in with no repercussions for the rest of your code (e.g. you could modify the declaration of `products` so that it's a `LinkedList<Product>` object rather than an `ArrayList<Product>` object).

Data providers are another classic example. Maybe in version 1 of our application the data source is to be a flat file. Maybe version 2 will need it to be a database. Or who knows what – maybe a web service. Design your provider class with this flexibility in mind: for example, define a method named `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 database?) Create your own Exception type at the appropriate level of abstraction – e.g. `DataAccessException` – and 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 then thrown.

Common Refactorings

- Extract Class
  - Remedy for a class that has become too large!
    - Break it into smaller classes with cohesive behaviour
  - Or if related functionality is in multiple classes
    - Put into a dedicated class where it's easier to maintain

- 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`

Neueda

Test Smell

- Hard-to-test Code
  - More likely with legacy code developed test-last than test-first
  - E.g. highly coupled code because of hard-coded dependencies
- Assertion Roulette
  - Cannot tell which of several assertions caused a test failure
  - E.g. Eager Test: single `@Test` is doing too much
    - Testing too much functionality; too many asserts
  - E.g. cannot replicate the failure on another machine
    - Resource optimism: dependant on environmental resource
- Obscure Test = difficult to understand at a glance
  - Too much in the test – e.g. Eager Test
  - Too little – hard to see cause/effect between fixture and assertion

Neueda

Badly written testing code can be as smelly as application code. Some of the remedies to the smells highlighted here are addressed in the pages on Unit Testing Best Practices, in the JUnit chapter. Some remedies are provided in the following pages on unit test patterns, and in part those remedies are expanded on in the chapter on mock objects. For instance, with highly coupled code we break the dependencies by using patterns such as constructor injection, setter injection, and test-specific subclass.

These test smells are discussed in

`http://xunitpatterns.com/Test%20Smells.html`

Testing Patterns (1 of 6)

- Resulting State Assertion
  - Standard test: this is the state we expect
- Guard Assertion
  - Assert precondition for the test to be correct
  - Then Resulting State Assertion
  - e.g. a `remove()` method really removes something that was there

```
assertThat(basket.list(), hasItem(new Product("iPad", 9)));
basket.remove(1);
assertThat(basket.list(), not(hasItem(new Product("iPad", 9))));
```

  - Matchers generally make 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("iPad", 9))));
```

Kent Beck recommends writing the assert as the first step in writing a test method – i.e. start by identifying "What is the right answer?"

Recall that the ideal is to have exactly one assertion per test. But this is not an absolute, and the guard assertion pattern is a good illustration of the exception that proves the rule.

Testing Patterns (2 of 6)

- Delta Assertion
  - If absolute Resulting State cannot be guaranteed…
  - … then test the delta between the initial and resulting states

- Custom Assertion
  - If the code verifying assumptions is long, extract it to a method
  - Note that the method contains an `assert()`

```
private void assertCardsAllDifferent(Deck deck) {

    Set<Card> set = new HashSet<Card>();

    int deckSize = deck.getSize();
    for (int i = 0; i < deckSize; i++) {
        set.add(deck.serveCard());
    }

    assertThat(set.size(), is(52));
}
```

Neueda

The Delta Assertion pattern would be used where there is something slightly outside the control of the test, where its initial state is not determinate, but where the operation we are testing will have a determinate effect upon it. Performance tests are often implicitly delta assertions, in that they take a timing before an operation, and the timing again after it, and assert that the difference is below a certain threshold.

The Custom Assertion pattern helps our test code respect DRY, in that there may be some common functionality that otherwise would be duplicated. In the envisaged example, perhaps we need to assert that all the Cards in an initially generated Deck are different, and again, when the shuffle operation has been performed. Obviously one has to be careful about introducing complexity into the tests, without having a test to verify that this functionality itself is correct.

Testing Patterns (3 of 6)

- Parameterised Test
  - Data-driven testing: one test method, multiple data sets
  - Data hard-coded in 2D array, or from external source (file, db)
  - Annotate test class: `@RunWith(value=Parameterized.class)`

```
@Parameters
public static Collection makeData() {
  return Arrays.asList( new Object[][] {
                              { 1,  "Jan" },
                              { 2,  "Feb" }, …
                              { 12, "Dec" },
                              });
}              // { input to function, and expected output }

public UtilsTest(int input, String output) {
  this.input = input;
  this.expected = output;
}
```

Neueda

JUnit 4 provides for parameterised tests to be set up with annotations. The test class itself must be annotated as follows:

```
@RunWith(value=Parameterized.class)
public class UtilsTest {
  private int input;
  private String expected;
  // Plus the code shown in the slide above …
}
```

The test methods in your test class look like ordinary test methods, except that they use instance variables of the class.

```
@Test
public void testGetMonthString() {
  assertEquals("Bad 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 in to the constructor, and any `@Before`, then `@Test`, then `@After` method in the class invoked. Equally importantly, we 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.

Testing Patterns (4 of 6)

- Mock Objects
  - For testing objects with expensive or complex collaborators
- Fowler identifies two types of distinctions
  - State verification: Assert against resulting state of CUT
  - Behaviour verification: Verify correct calls were made by CUT
- Classical vs. Mockist TDD:
  - Classical TDD: Use real collaborators in tests wherever possible
  - Mockist TDD: Always use mocks
- Example: file system interaction
  - Classical TDD: Use real file if overhead not high
  - Mockist TDD: Mock the file system interaction

Neueda

Martin Fowler (“Mocks Aren’t Stubs”):

“This coupling leads to a couple of concerns. The most important one is the effect on Test Driven Development. With mockist testing, writing the test makes you think about the implementation of the behavior – indeed mockist testers see this as an advantage. Classicists, [which MF identifies himself as] however, think that it’s important only to think about what happens from the external interface and to leave all consideration of implementation until after you’re done writing the test.

Coupling to the implementation also interferes with refactoring, since implementation changes are much more likely to break tests than with classic testing.”

Testing Patterns (5 of 6)

- Parameterised Creation Method
  - Factor-out fixture object creation from `setUp()` to a parameterized creation method
  - Hides attributes that are essential to fixtures but irrelevant to test
  - Useful when creating complex mock, especially if it will be used in multiple tests
- Object Mother
  - Factor-out creation of business objects to a 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);
```

Neueda

With the Parameterised Creation Method pattern, we can encapsulate complex fixture creation into a self-explanatory method. Here's an example using the JMock framework to create something that 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;
}
```

Testing Patterns (6 of 6)

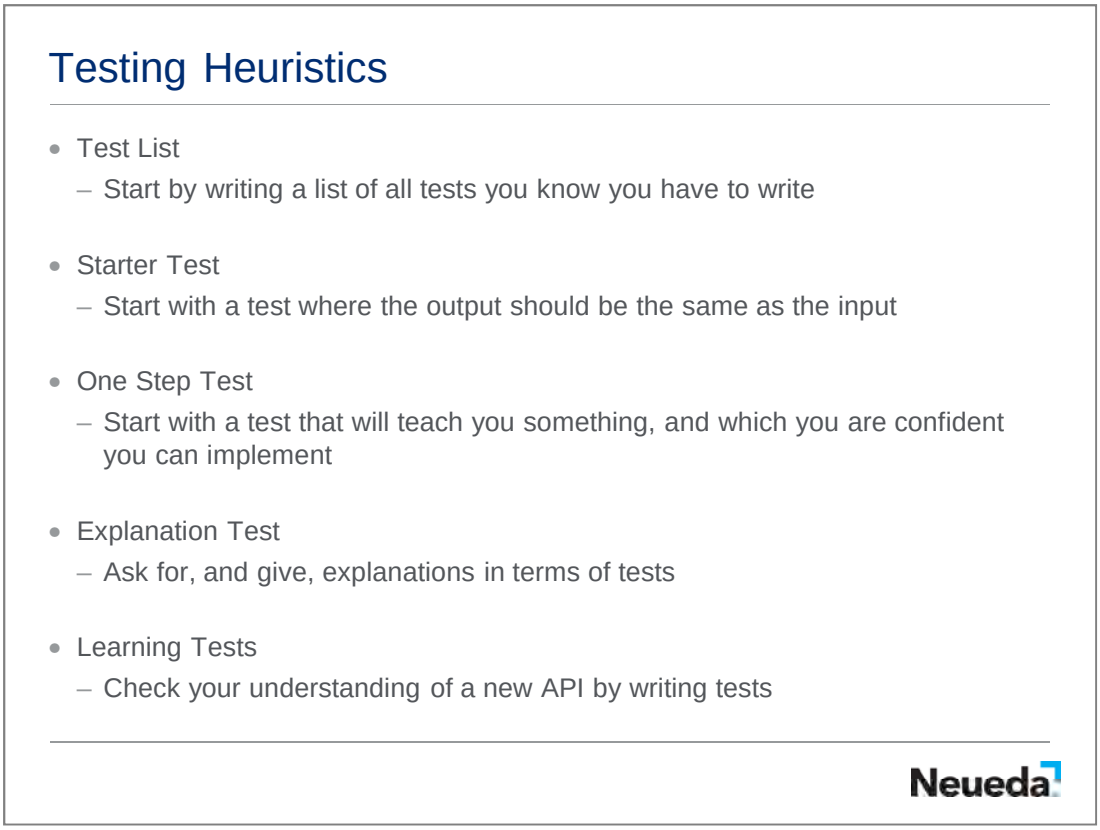
- Extra Constructor
  - If an existing constructor hard-codes some dependencies, then it might be useful to define an extra constructor, so that the test can inject a suitable object
  - Effectively, we've introduced a "trapdoor" to make the code easier to test:
  - "My car has a diagnostic port and 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
- Test-specific Subclass
  - Not permitted to modify actual class
  - Create behaviour-modifying or state-exposing subclass
  - More likely when testing didn't drive development

Neueda

Ron Jeffries is quoted by Massol, JUnit in Action p.148, originally from

http://tech.groups.yahoo.com/group/testdrivendevelopment/message/3914

He continues "And if I find it useful to add a method to a class to enable me to test it, I do so. It happens once in a while, for example in classes with easy interfaces and complex inner function (probably starting to want an Extract Class)."



These mainly come from Kent Beck's *Test Driven Development*, Chapter 26 "Red Bar Patterns". They're primarily suggestions for breaking down a seemingly mountainous task of developing some new functionality in a test-driven way, into very small, tractable steps.

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?”

Starter Test provides an answer:

- The output should be the same as the input. Some configurations of polygons are already normalized, incapable of further reduction.
- The input should be as small as possible, like a single polygon, or even an empty list of polygons."

Explanation Test is primarily for communicating within the development group, clarifying the requirements for some item of functionality by expressing them precisely in terms of tests.

Agile Development

- Welcome changing reqs, even late in development
- Deliver working software frequently
  - From a couple of weeks to a couple of months
- The most efficient and effective method of conveying info within a development team is face-to-face conversation
- Continuous attention to technical excellence and good design enhances agility
- Simplicity – maximize the amount of work not done!

Neueda

These are some of the 12 key principles enumerated on

http://agilemanifesto.org

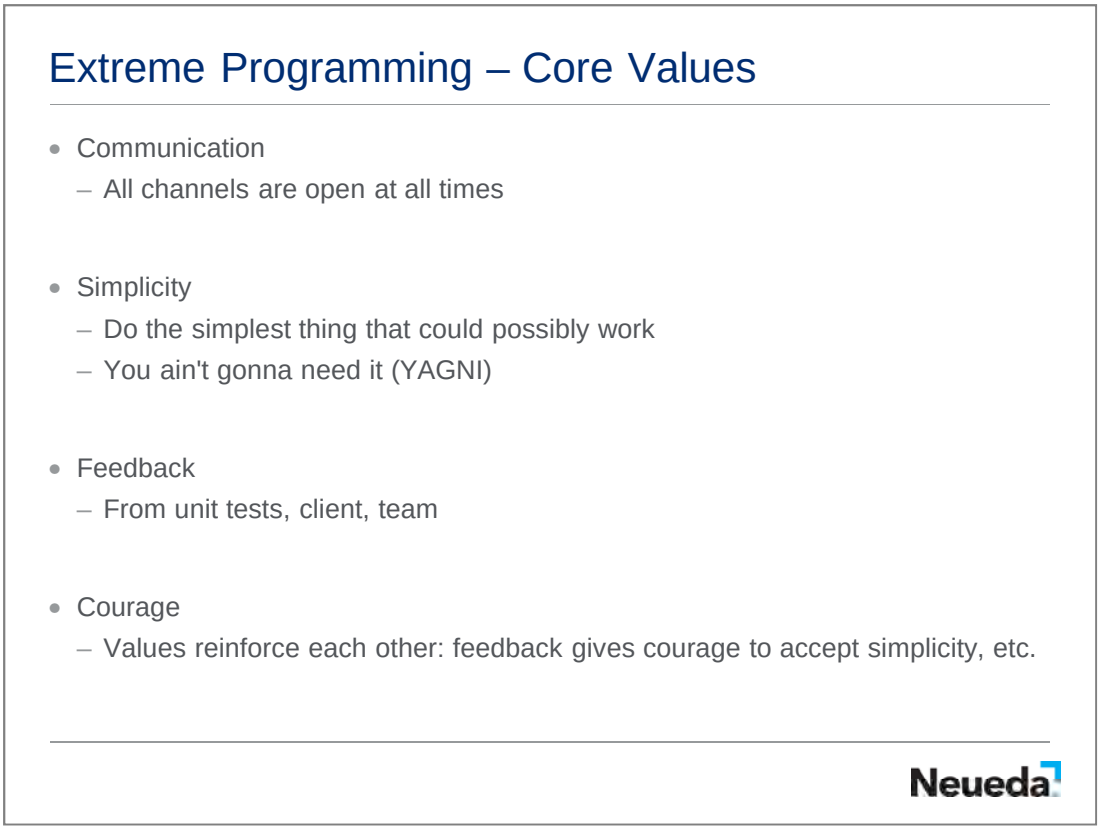
See also

http://www.waterfall2006.com

One way agile teams reflect on what they've done is to perform a retrospective, e.g. at the end of a development sprint. Someone draws a large cross on a flip-chart or whiteboard, dividing it into quadrants, with a symbol in each something like the following:

- Smiley face – what went well/what we would do again
- Sad face – what didn't go so well/what we wouldn't do again
- Light bulb or ! – what we learnt
- Question mark – what still puzzles us

All members of the team then contribute their thoughts which get written into the relevant quadrant.



Extreme Programming is one example of an Agile software development methodology. The Agile movement gained momentum in the light of perceived drawbacks to traditional approaches to software development. Common to all Agile approaches to software development is a commitment to very short iterations of development cycles, typically two to four weeks, and an acceptance of change. Agility implies the ability to adapt quickly to changing project needs.

YAGNI is an important part of the extreme, test-driven discipline. What it means is: don't go beyond implementing the immediately necessary functionality, starting to anticipate functionality you believe will be needed. If you are truly following TDD, you would be writing code beyond the constraints embodied in your current set of tests. If the functionality is needed, you will implement it when the time comes – but right now, you will be wasting time and effort.

Kent Beck added a fifth value in the second edition of *Extreme Programming Explained*:

Respect

- team members are to respect each other and their work

Extreme Programming Practices

- Pair Programming
  - One programmer codes, the other reviews; regularly swap roles
  - E.g. 1 writes test, 2 codes solution and writes next test; swap
  - E.g. swap on fixed time period (e.g. 25 mins)
- Collective Ownership
  - Anyone can change any code anywhere in system at any time
  - You may come back to code you wrote some weeks ago and find it's all changed, methods deleted, etc.
  - As opposed to the traditional model: a sole developer works on a module for months
- The Planning Game
  - Meeting to determine scope of this iteration/sprint

Neueda

These are some of the 12 practices set out by Kent Beck in *Extreme Programming Explained* . Another is the Agile emphasis on short release cycles. But the most widely adopted of all XP practices is TDD.

Summary

- TDD: big change to developers' mind-set and practices
- Test – Code – Refactor
  - Write a test
  - Make it run
  - Make it right
- Develop in small increments
- Follow unit testing best practices
  - Make tests clear, maintainable, fine-grained, independent
  - Ensure good coverage of negative (abnormal flow, errors) as well as positive (normal flow)

Neueda

References

- Test Driven Development – By Example
  - Kent Beck (Addison Wesley) ISBN 0321146530
- xUnit Test Patterns: Refactoring Test Code
  - Gerard Meszaros (Addison Wesley) ISBN 0131495054
  - http://xunitpatterns.com
- Refactoring: Improving the Design of Existing Code
  - Martin Fowler (Addison Wesley) ISBN 0201585672
- http://www.testdriven.com
- http://www.infoq.com

Neueda

Any Questions?

Neueda

Unit Testing with Mock Objects

Neueda

Unit Testing With Mock Objects

- Objectives
  - Introduction to Mock Objects, and an example using EasyMock
- Contents
  - Motivation for using Mock Objects
  - Stubs and Mocks
  - Worked examples
  - How to use EasyMock
  - When to use Mock Objects
  - Drawbacks of Mock Objects
- Exercise

Neueda

Mock Objects: Introduction

- What if you want to unit test:
  - A class whose lifecycle is managed by a container
  - A class that accesses a database
  - A class that access a network resource
- One possibility: run it in its real environment
  - Integration testing
  - Important to test interactions between your class and the rest
- But: other resource(s) …
  - May not be available yet
  - May not be available for use in tests
- You want to test the logic of your class in isolation

Neueda

Roy Osherove: "An *external dependency* is an object in your system that your code under test interacts with, and over which you have no control. (Common examples are filesystems, threads, memory, time, and so on.)" *The Art of Unit Testing*, p. 50.

Objects with which the class under test interacts are also known as its *collaborators*, and substitutes for them used in testing are sometimes referred to as *test doubles*. Osherove uses the term *fake* to cover both stubs and mocks.

Stubs

- Stub: controllable replacement for existing dependency
  - "provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test ... may also record information about calls" – Fowler
- A class that is the simplest possible implementation of the logic in the real code
- Good for coarse-grained testing
  - Replacing a complete external system, e.g. web server, database
- But often complex to write
  - Introduce their own debugging and maintenance issues
  - Not so suitable to fine-grained unit testing

Neueda

The quote from Martin Fowler's comes from his article "Mocks Aren't Stubs", which can be found at

- http://martinfowler.com/articles/mocksArentStubs.html

Worked Example: Lottery

- Task: to generate a String of six different random numbers, in format "9 – 41 – 13 – 7 – 25 – 38"
- Original version: untestable

```
Set<Integer> nums = new HashSet<Integer>(LOTTERY_SIZE);

while (numbers.size() < LOTTERY_SIZE) {
  numbers.add(rand.nextInt(HIGHEST_NUMBER)+1);
}

int count = 0;
StringBuilder sb = new StringBuilder();
for (Integer number : numbers) {
  sb.append(number);
  if (++count < numbers.size()) sb.append(" - ");
}

System.out.println(sb.toString());
```

Neueda

We're going to work with the example of simulating a Lottery, because it allows us to bring out some of the core issues with mocks and stubs, without the distractions of working with an extra API.

All of the code for this example is located in the `DemoMock` project, in packages named `stubsandmocks.versionX`.

Our starting point is the `Lottery` class in the `stubsandmocks.version0` package. The `Lottery` class has a single method, `main()`, that declares the following fields:

```
private static final int LOTTERY_SIZE = 6;
private static final int HIGHEST_NUMBER = 49;
private static Random rand = new Random();
```

The `main()` method starts by exploiting the fact that a `Set` rejects duplicates, as a way of ensuring the 6 lottery numbers will be distinct. The second half of `main()` is then concerned with the correct formatting of the results. It starts with a `StringBuffer` and `count` variable (not shown above), and builds up the string with the numbers spaced apart in the format required.

Refactoring for Testability

- Reorganize `main()` into more testable parts
  - `generateRandomSet()`
    - Returns a `Set<Integer>`
  - `formatNumbers()`
    - Takes a `Set<Integer>` and returns a `String`
    - Formatting logic can be tested with a determinate set of ints
- Factor-out the number generation into a separate method:

```
static int generate(int limit) {
  return rand.nextInt(limit)+1;
}
```

Neueda

The code and tests shown here are located in the `stubsandmocks.version1` package (see the `src` and `tests` folders).

When you take TDD to heart you will naturally be writing code in small, testable units, favouring methods which return a value (which can be tested) rather than methods whose return type is void. Here our imagined starting point was not developed that way, and we are having to manipulate it into a more testable form. By separating out the formatting logic we can test it with a set of numbers which the test can control, e.g.

```
@Test
public void formatIsHyphenSeparatedSequenceOfNumbers() {
  Set<Integer> numbers = new HashSet<Integer>();
  numbers.addAll(Arrays.asList(20, 30, 40));
  assertTrue(Lottery.formatNumbers(numbers)
                    .matches("^\\d+ - \\d+ - \\d+$"));
}
```

Refactoring to Use a Manual Stub (1 of 2)

- Declare an interface to represent number generation

```
public interface NumberGenerator {
  public int generate(int limit);
}
```

- Implement a "stub" version of the interface, for use in tests

```
public class NumberGeneratorStub implements NumberGenerator {
  private int number = 0;
  public int generate(int limit) { return number++; }
}
```

- Lottery class can now use Constructor Injection:

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

Neueda

The code and tests shown here and the following slide are located in the `stubsandmocks.version2` package (see the `src` and `tests` folders).

The real lottery runner can inject an instance of the interface that generates real random numbers, i.e. its `main()` could look like this:

```
Lottery lotto = new Lottery(new RandomNumberGenerator());

System.out.println(lotto.formatNumbers(lot.generateRandomSet(
)));
```

The test method that uses this lottery is shown on the next slide. We remove the dependency on something outside the control of the test, and achieve a test that has a guarantee of repeatability.

Refactoring to Use a Manual Stub (2 of 2)

- Functionality of `formatNumbers()` can be tested
  - The test class injects `NumberGeneratorStub` into `Lottery`

```
@Test
public void formatNumbersReturnsNumbersInLottoFormat() {
  Set<Integer> nums = lotto.generateRandomSet();
  String expected = "0 - 1 - 2 - 3 - 4 - 5";
  assertThat(lotto.formatNumbers(nums), is(expected));
}
```

- The stub can even record interactions of CUT
  - E.g. tests can verify that CUT made expected number of calls
  - Coincidentally number used for dummy data in this case is exactly that, so `verifyCallsMade()` can return it

```
assertThat(stub.verifyCallsMade(), is(6));
```

Neueda

The `LotteryTest` class starts like this, i.e. injecting a stub instance into the lottery:

```
private Lottery lotto;

@Before
public void setUp() {
  lotto = new Lottery(new NumberGeneratorStub());
}
```

Incidentally, the real random number generator method can also be tested using the Hamcrest matchers. The test serves as a constraint, and documents the fact that the range is strictly greater than zero but inclusive of the upper bound

```
@Test
public void generateGivenLimit10ReturnsNumberInRange() {
  NumberGenerator gen = new RandomNumberGenerator();
  for (int i = 0; i < 10; i++) {
    assertThat(
          generator.generate(10),
          allOf(greaterThan(0),lessThanOrEqualTo(10)));
  }
}
```

Mock Objects (Mocks)

- 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

- Common mocking frameworks available:
  - jMock
  - EasyMock
  - Mock Objects
  - Mockito
  - Rmock
  - SevenMock
  - Etc…

Neueda

Mocks and stubs are not necessarily mutually exclusive; it may be appropriate, depending on the class being tested, to combine use of a mocks and stubs.

Mocks and stubs can be handwritten, simple classes – if they remain simple they will 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. Here are some popular mocking frameworks:

- jMock – `http://jmock.org`
- EasyMock – `http://www.easymock.org/`
- Mock Objects – `http:/www.mockobjects.com`
- Mockito – `http://code.google.com/p/mockito`
- RMock – `http://rmock.sourceforge.net`
- SevenMock – `http://seven-mock.sourceforge.net`

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 vs. Mocks

- Stubs…
  - Enable tests by replacing external dependencies
  - Asserts are against the CUT (not against the stub), so a stub won't make the test fail
    - Test <————> CUT <————> Stub
    - Asserts interacts
- Mocks…
  - Test will use a mock to verify that the CUT interacted with the external dependency in the correct way
  - Asserts are against the mock
    - CUT <————> Mock <————> Test
    - interacts Asserts

Neueda

Mock object: "a fake object ... that decides whether the unit test has passed or failed. It does so by verifying whether the object under test interacted as expected with the fake object. There's usually no more than one mock per test." (Osherove, p. 84)

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

EasyMock: Introduction

- On the next few slides, we'll show how to use EasyMock
  - This is one of the most popular Java mocking frameworks
- General process for using EasyMock in a JUnit test case:
  - Create required mock objects, e.g. via `createMock()`
  - Create an instance of the CUT, replacing dependencies on real collaborators with mocks
  - Set expectations on mock(s):
    - Say what method calls you expect to occur
    - Fake the behaviour of these methods by returning the values you expect for each method call
  - Call `replay()`, which indicates expectation-setting has finished
  - Invoke a method on CUT, as with any JUnit test
  - Call `verify()`, to verify if the mock get the calls we expected
  - Optionally, make assertions (as before)

Neueda

Note that all the best mock object frameworks are type-safe: the mock really is an instance of whatever interface it mocks. Your favourite Eclipse keyboard shortcut, <CTRL> <SPACEBAR> will show you that the mock object can have just the same methods invoked on it as any other instance (of a class which is an implementation) of the interface.

For this reason, best practice suggests you adopt a naming convention like starting all your mock object identifiers with `mock`, to make it clear which are the mocks and which the real objects in your code.

On the next few slides we'll present a complete example of how to use EasyMock to test our `Lottery` class. The code and tests are located in the `stubsandmocks.version3` package (see the `src` and `tests` folders).

Note:

To use EasyMock, you must ensure `easymock.jar` is on the classpath, and that you import the `org.easymock.EasyMock` class. The `EasyMock` class contains a whole bunch of important static methods, e.g. `createMock()`, `expect()`, `replay()`, and `verify()`.

EasyMock: How to Write a Test

- Here's an example of what a test looks like when you use EasyMock
  - The following slides describe this code in more detail

```
@Test
public void generateRandomSetReturnsLotteryNumbers() {

  NumberGenerator mockGenerator = createMock(NumberGenerator.class);

  for (int i = 1; i < 7; i++) {
    expect(mockGenerator.generate(anyInt())).andReturn(i);
  }
  replay(mockGenerator);

  Lottery lotto = new Lottery(mockGenerator);
  Set<Integer> numbers = lotto.generateRandomSet();
  assertThat(lotto.formatNumbers(numbers), is("1 - 2 - 3 - 4 - 5 - 6"));

  verify(mockGenerator);
}
```

Neueda

This case is slightly atypical in that we need to say to the mock, in effect

```
expect(mockGenerator.generate(49)).andReturn(1);
expect(mockGenerator.generate(49)).andReturn(2);
expect(mockGenerator.generate(49)).andReturn(3);
expect(mockGenerator.generate(49)).andReturn(4);
expect(mockGenerator.generate(49)).andReturn(5);
expect(mockGenerator.generate(49)).andReturn(6);
```

We need the mock to return different numbers every time `generate()` is called. Otherwise we could have expressed that very succinctly as

```
expect(mockGenerator.generate(49)).andReturn(1).times(6);
```

The `for` loop is there simply to set up that series of six expectations, as the variable `i` is incremented on each iteration.

Notice that the argument matcher `anyInt()` rather than the hard-coded value `49` makes the test more flexible, without undermining its essential rigour. Using `anyInt()` means that if the `Lottery` highest number was changed to a value other than 49, these expectations would not break.

EasyMock: Creating Mock Objects

- There are several ways to create a mock object in EasyMock:
  - `createStrictMock()`
    - There must be no unexpected method calls
    - The order of the calls must be as per expectations
  - `createMock()`
    - There must be no expected calls
    - The order of the calls can be different to the order of expectations
  - `createNiceMock()`
    - There can be method calls from the CUT which were not expected
    - The order is not checked

Neueda

Here are some guidelines, to help you find the right balance for your tests:

- Tests must not be trivial – too easy to pass.
- Tests must not be too severe (aka “brittle”) – fail too easily even when legitimate changes are made to the CUT.
- General advice – avoid strict mocks.

EasyMock: Setting Expectations

- Expectations-setting phase
  - nice mock will return `null`, or `0`, or `false` for unexpected calls
- For a `void` method, use `expectLastCall()`

```
mockResponse.setContentType("text/html");
expectLastCall();
```

- For a method that throws exc^{n}, use `andThrow()`

```
expect(mockResponse.getWriter()).andThrow(new IOException());
```

- If you want to specify the cardinality, use:
  - `anyTimes()`
  - `atLeastOnce()`
  - `times(int count)`
  - `times(int min, int max)`

Neueda

The return type of an `expect()` call is the interface `IExpectationSetters`, and it is here that the method `andReturn()` is found (whose return type is also `IExpectationSetters`).

EasyMock: Replay and Verify

- Replay phase
  - `replay(mockRequest, mockResponse);`
  - This indicates you've finished setting expectations, and are ready to check if the expected method invocations are actually made
- Invoke method on instance of tested class
- Verification phase
  - `verify(mockRequest, mockResponse);`
  - This verifies that the tested object did perform all the method calls that were expected of it
- Optionally, `reset()` a mock object
  - Allows you to reuse for a new set of expectations etc.

Neueda

Verification can be factored out into the `tearDown()` method, if appropriate:

```
@After
protected void tearDown() throws Exception {
  verify(mockRequest, mockResponse);
}
```

EasyMock: Variations

- Classes can be mocked (not just interfaces)
  - Needs a code-generation library
  - Add `cglib-nodep` and `objenesis` jars to the classpath
- Make your test class extend `EasyMockSupport`
  - `replayAll()` – puts all mocks into replay mode
  - `verifyAll()` – verifies all mocks; can be put into `tearDown()`
- Use argument matchers to make your tests flexible

```
expect(mockRequest.getParameter(endsWith("name"))).andReturn("A");

mockWriter.println(startsWith("<h4>"));
expectLastCall();
```

Neueda

Easymock's argument matchers are static methods in the `EasyMock` class, and include:

- `anyObject()` – expect any `Object`
- `isA()` – takes a class literal (e.g. `String.class`), and will be satisfied with any instance of that class
- `anyBoolean()` etc. for all the other primitives – will be satisfied with any value of the appropriate type
- `aryEq()` – check for array equivalence (same length and equal elements)
- `contains(String substring)` – expect a `String` with this substring
- `startsWith(String prefix)` – expect a `String` starting with this prefix
- `find(String regex)` – expect a `String` with substring matching regex
- `matches(String regex)` – expect a `String` that matches the regex
- `cmpEq()` – expect an equivalent `Comparable` object
- `gt(), lt(), geq(), leq()` – perform greater-than, less-than, greater-than-or-equal, and less-than-or-equal tests. These are defined for primitive types and for `Comparable` objects.

EasyMock: Errors

- Any mock: got less than it expected
  - Test fails, message e.g.: expected: 2, actual: 1
  - Without `verify()`: test wrongly passes
- Non-nice mock: got more than it expected
  - Test fails, message e.g.: expected: 1, actual: 2
  - Fails even if `verify()` is overlooked
- Any mock: didn't call `replay()`
  - Test *errors* – `IllegalStateException`

Neueda

Mock objects are not the easiest framework to get your head round initially, and one of the things that can make them confusing are the sometimes obscure errors and error messages you encounter.

Remember that tests error if something is fundamentally wrong in the way the test is setup, and that includes omitting a library that is needed at runtime, such as the `cglib` and its related library `objenesis`.

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 – e.g. "was this callback called?"
  - Does not yet exist

Neueda

The list above is taken from Mackinnon, Freeman and Craig “Endo-testing: Unit Testing with Mock Objects”.

Drawbacks of Mock Objects

- Don't test interactions with container or between the components
  - Don't give full confidence that code will run in target container
  - i.e. still need integration tests
- You need an excellent knowledge of API being called
  - Expectation-setting makes tests mirror internal implementation
- Objects are mocked, not classes
  - So `static` methods need workaround
- You can't mock `final` classes/methods
  - e.g. `java.util.Scanner` and `java.io.Console`

Neueda

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

Summary

- Mock objects enable small, focussed unit tests
  - This promotes refactoring
- Stubs contain business logic; Mock objects do not
  - Mocks do only what they are set up to do in the tests
- Mock objects allow you to probe a class
  - Determine if a method was called that was not expected
  - Determine if a method was not called that was expected
  - Verify the order and number of times methods are called
- EasyMock: widely adopted mock object framework
  - Integrates easily with JUnit
  - Others: jMock, Mockito, RMock, SevenMock, …

Neueda

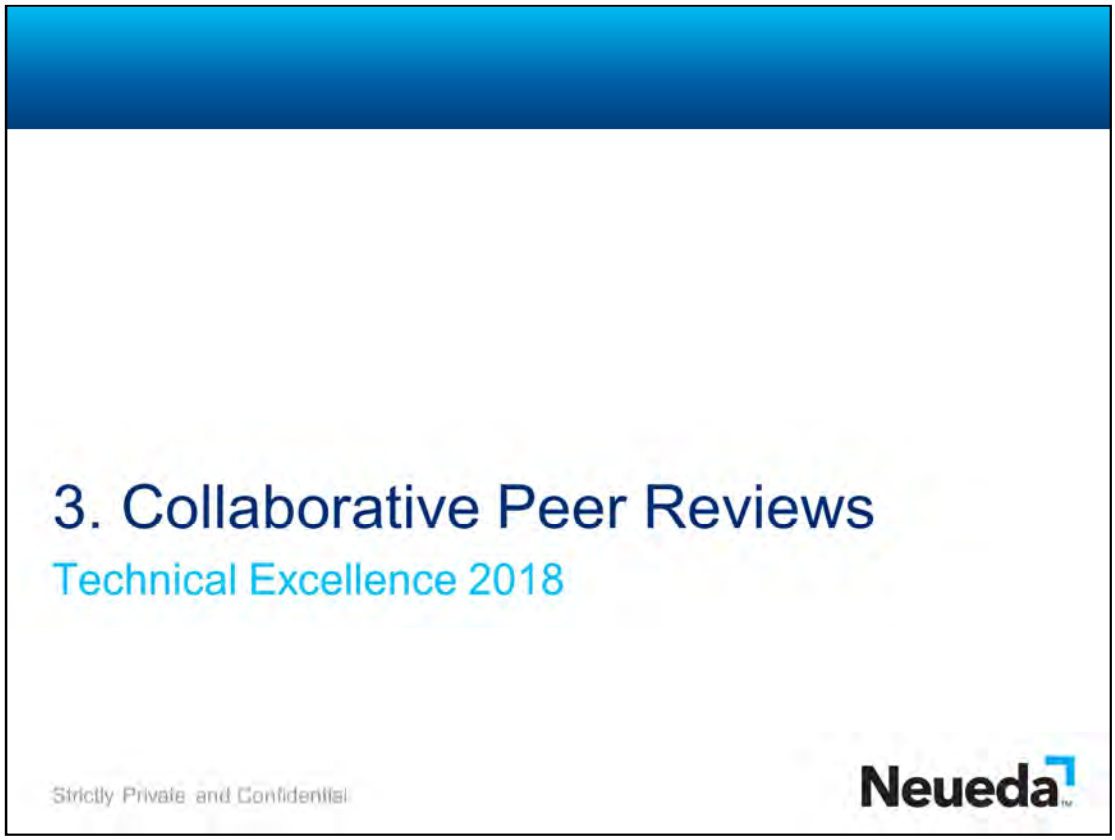
References

- http://easymock.org
- http://www.mockobjects.com (and http://jmock.org)
- http://tech.groups.yahoo.com/group/easymock
- Pragmatic Unit Testing in Java with JUnit
  - Andy Hunt, Dave Thomas (Pragmatic Bookshelf) ISBN 0974514012
- Test Driven Development: A Practical Guide
  - David Astels (Prentice Hall) ISBN 0131016490

Neueda

Any Questions?

Neueda

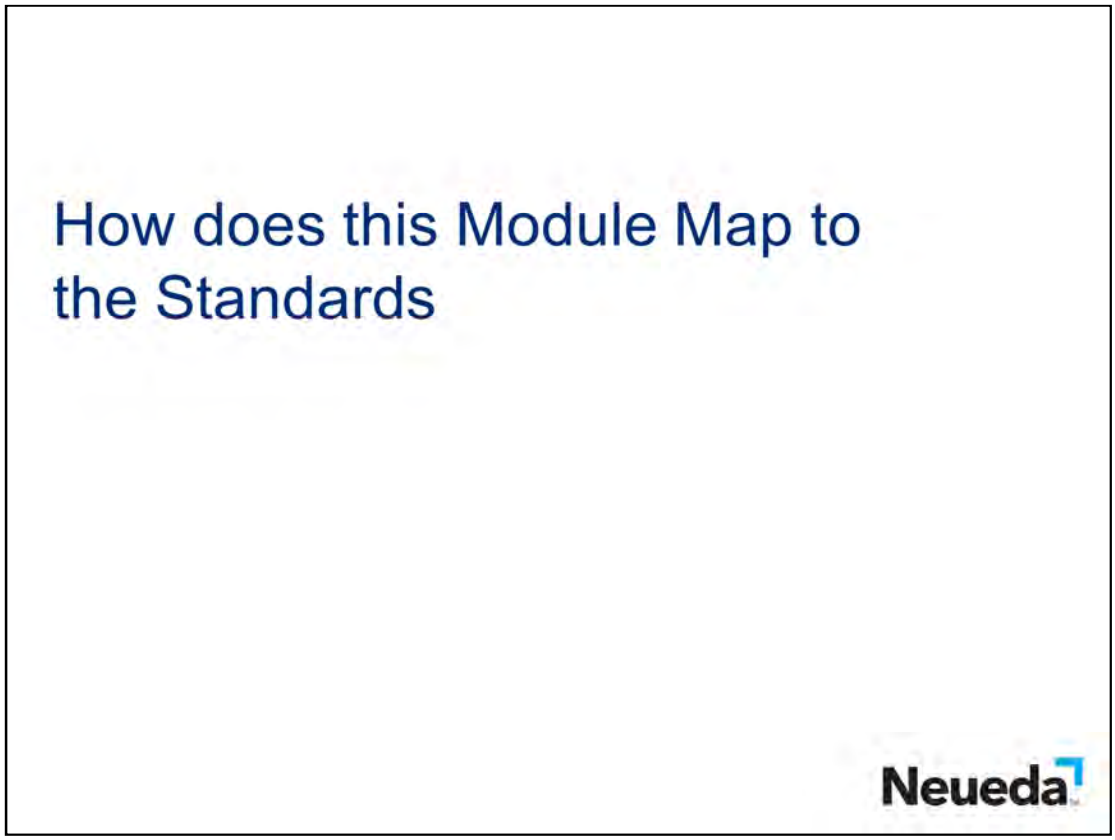


- One of the most powerful quality practices available to the software industry is to have colleagues of the author of a software work product examine that product for possible problems. This activity is called a peer review.

- Almost everyone who has ever written a computer program has asked a colleague to come look over his shoulder and help him find an elusive problem. This is the simplest type of peer review. We often cannot find errors in our own work simply because we are too close to the work. We can all benefit from an outside point of view.

- This course addresses the broad domain of software peer reviews, focusing on a specific type of highly structured review called an inspection.

- While you will not always have every work product you create reviewed, and you will not always perform a review using the full rigor of the inspection technique, it’s important for every software organisation to develop a culture in which people get a little help from their friends to make their work products better.


How does this Module Map to the Standards
Neueda

Agenda

- The Case For Peer Reviews
- Types of Review
- The Softer Side to Peer Reviews
  - Resistance to Reviews
- Practice Review
- Automating Reviews

Neueda


1. The Case for Peer Reviews

What is a Software Peer Review?

An Examination of a Software Work Product by People other than Its Author in Order to Identify Defects (Departures from Specifications or from Standards) and Improvement Opportunities

Neueda

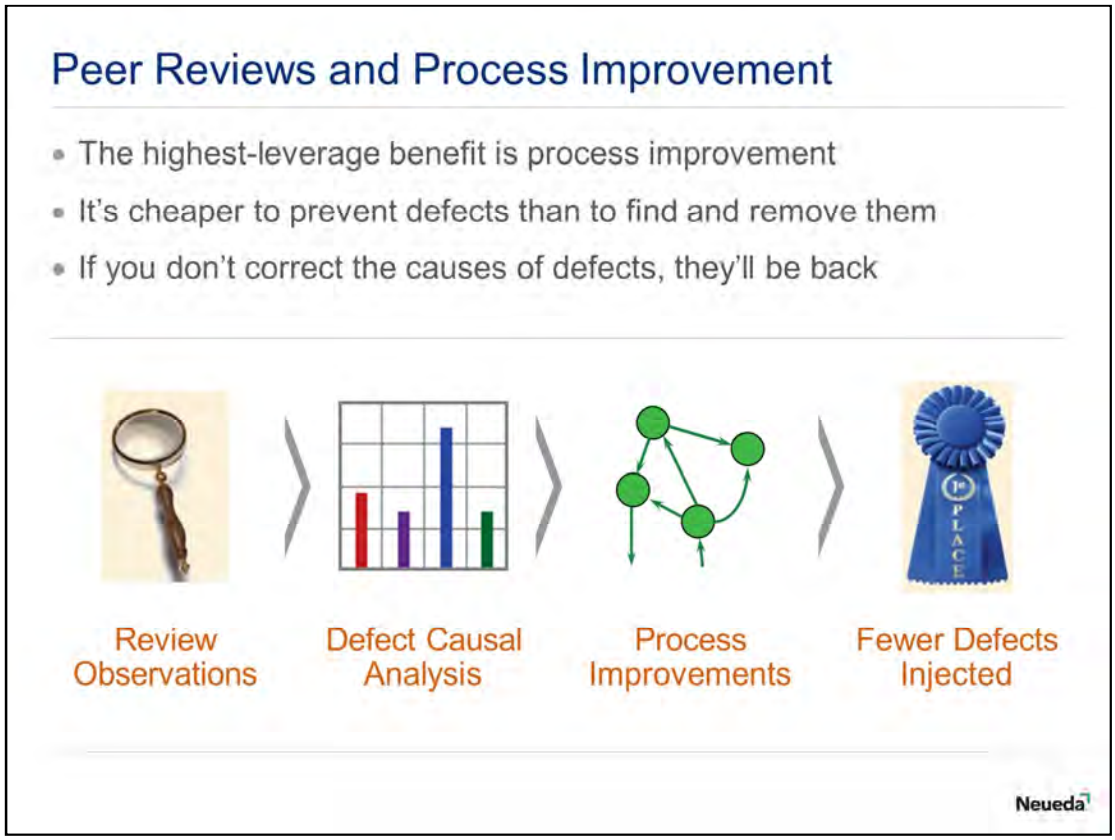
- We use the terms “fault,” “bug,” “defect,” and “error” interchangeably in this class.
- Technical work needs reviewing for the same reason that pencils need erasers: people make mistakes.
- Although people are good at catching some of their own errors, many errors that escape the originator can be spotted by someone else.
- The specific purpose of these peer reviews is to find problems, not to: report status, make project decisions, educate people about a project or product, or find someone to blame for project problems.

Objectives of Peer Reviews

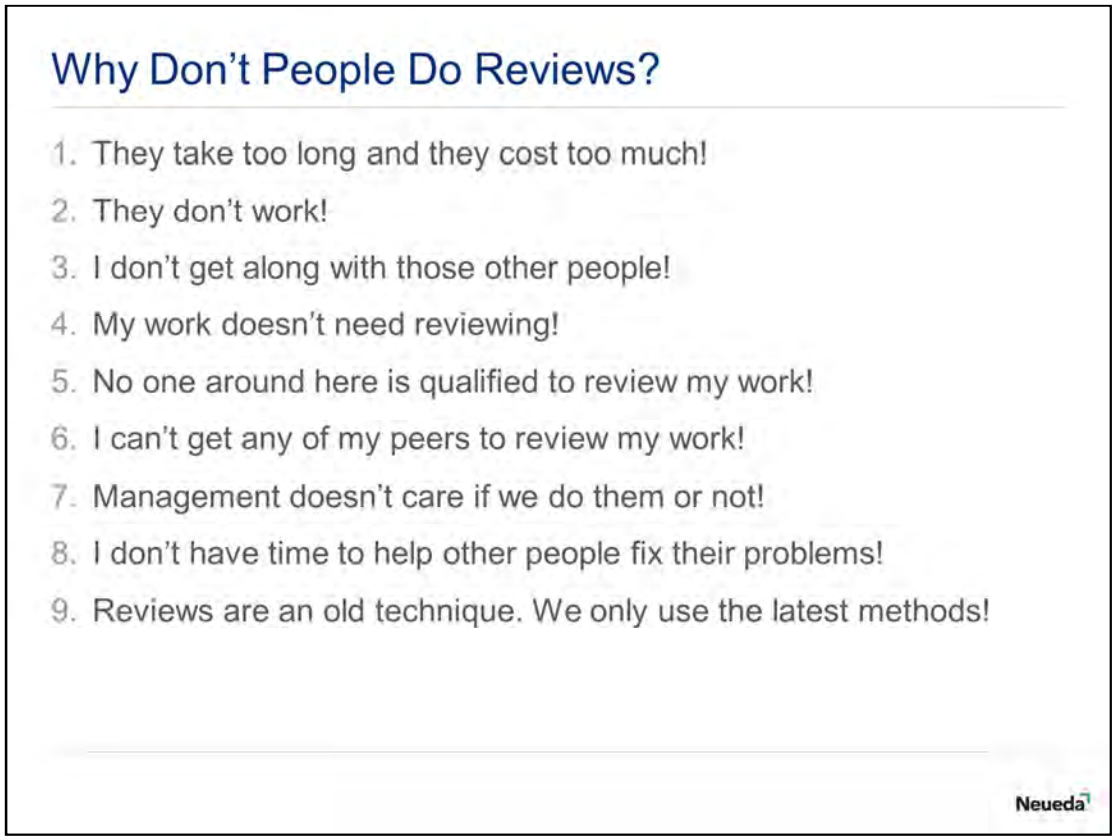
- To reveal errors in function, logic, or implementation
- To ensure that a work product satisfies its specification
- To check for conformance to standards
- To give managers insight into product quality
- To communicate technical information in a project
- To enable someone besides the author to support or modify a work product
- To identify process improvement opportunities

Neueda

- Here are some reasons why people hold software peer reviews.
- There's another benefit, too. I have learned something from every review I've participated in, either as the creator of the work product or as a reviewer. Reviews provide a great opportunity for seeing how other people do work similar to yours so you can find new ideas for doing a better job on the next task you undertake.



- People often emphasize the defect-removal capabilities of peer reviews.
- This is an important benefit, but even more powerful is the ability to analyse patterns of defects found during peer reviews to identify ways to improve your software development process that will lead to fewer such defects being created in the future.



- Here are some excuses that people offer for not performing peer reviews. In most cases they really are excuses, but sometimes resistance of the types shown here indicates a cultural problem in the organisation or shortcomings in the way the reviews are being performed.
- It's important to recognise that reviews are not free. They take time and consume resources. The rationale for performing reviews is that the benefits can greatly outweigh the costs. That is, performing reviews is cheaper than not performing reviews and then dealing with the cost of the defects lingering in the work product farther down the line.
- Learning how to review work takes training and experience. The first few reviews you perform may not yield the kind of benefits that you can ultimately achieve with more experience.
- Reviews can in fact cost more than they yield in a few cases. If the review is performed ineffectively, if defects are found are not corrected, or if there aren't any defects in the product in the first place, then the review could be a waste of time. Most of the time, though, they are well worth while.

Cultural Barriers to Peer Reviews

- Fear of public ridicule or criticism
- Previous negative review experiences
- Fear that management will hold defects against the author
- Attitude that "my work doesn't need reviewing"
- Concern that reviews will slow the project down
- Belief that testing is faster
- Review outcomes that managers overrule
- National or team cultures that avoid personal criticism

Neueda

- Asking someone else to look at your work and tell you what you did wrong is a learned, not instinctive, behavior.
- Here are some of the cultural barriers that must be overcome if individuals and teams are to hold effective peer reviews.
- The first time someone walks out of a peer review feeling personally attacked or professionally insulted is the last time that person will voluntarily submit his work for review.
- Reviews do not slow the project down; bugs slow the project down. In those cases, the bugs are there whether you decide to look for them or not and you'll have to deal with them eventually. It's not a matter of "you can pay me now or you can pay me later." It's "you can pay me now or you can pay me a lot more later".
- The misperceptions about reviews, such as the notion that testing is always better, can be addressed through education.
- The cultural problems can perhaps be overcome if managers create a supportive, quality-driven environment that rewards people who ask their peers for input and who willingly review the colleagues' work.

Risks of Criminalising Defects

- Developers might not submit their work for review
- Developers might refuse to review someone else's work
- Reviewers might not point out defects they see
- Authors might hold unofficial "pre-reviews"
- Review teams might debate whether or not something is a defect
- Authors might review small bits of work to avoid finding too many bugs
- The culture will have an implicit goal of finding few defects

Neueda

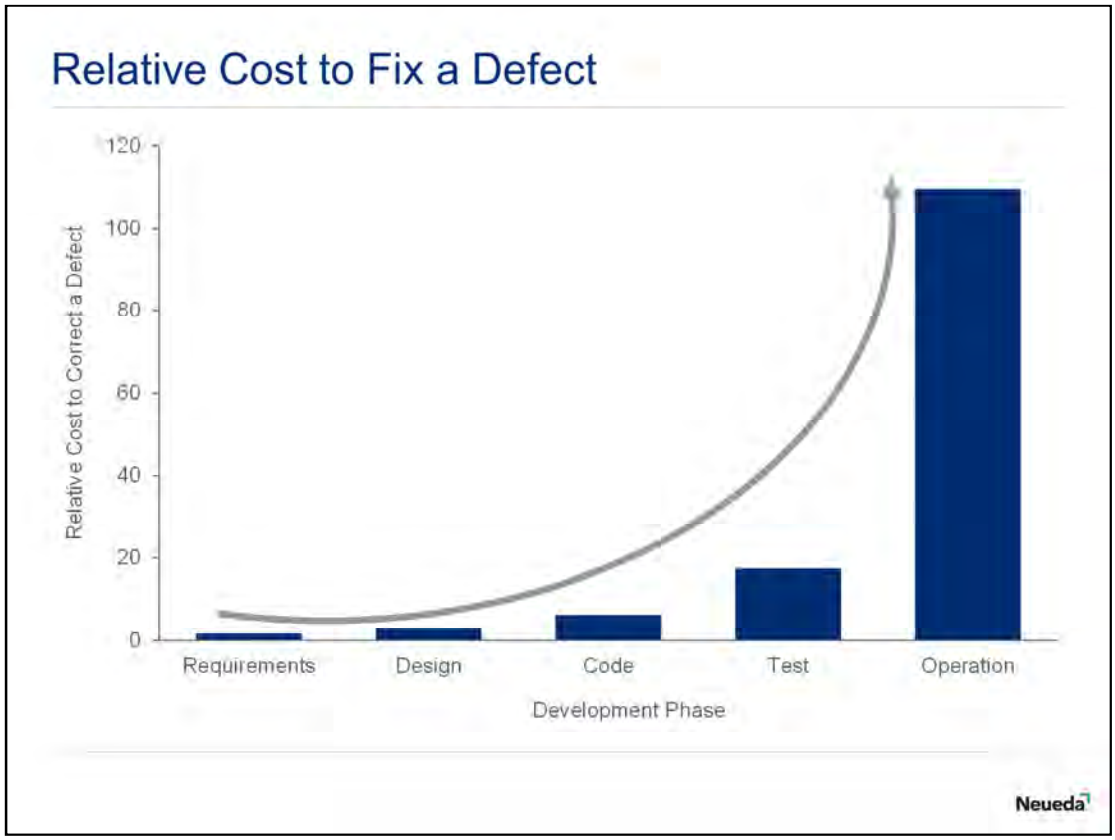
- I once received an e-mail from a quality engineer who said that his organisation had been successfully performing inspections for about two years. Now, though, the development manager wanted to use metrics from the inspections as input to the developers' performance appraisals. If more than 5 defects were found in an inspection of one of your work products, this would count against you at performance evaluation time.
- Naturally, this made the developers very nervous. The quality engineer who wrote to me was looking for some arguments to use to deflect the development manager away from this dangerous course of "criminalising defects". Here are some of the undesired behaviors that could result from this bad plan.
- This is an example of "dysfunctional measurement", in which measuring something changes people's behavior in an unexpected and undesired way. You must never use inspection data either to reward or to punish individuals.

Using Risk to Select Items to Review

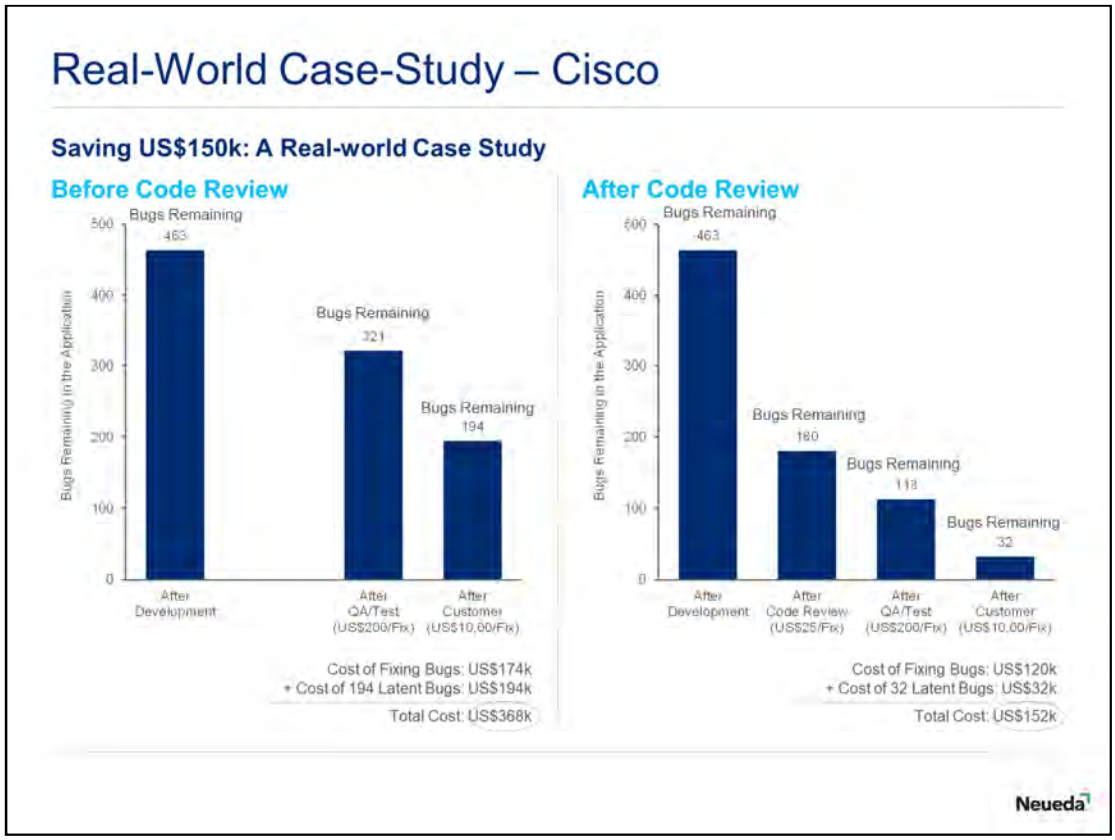
- Most mistakes are small; context determines severity
- Risk = (Probability of a defect) x (Impact if there's a defect)
- Focus limited review effort on high-risk portions of product
  - New technology or techniques
  - Complex logic or algorithms
  - Mission – or safety-critical components
  - Many or dangerous failure modes
  - Components intended for reuse
  - Created by less experienced people
  - Key architectural components
  - Components that affect multiple parts of product

Neueda

- Consider a simple typographical error in an if/then block in source code that can make certain code unreachable. This type of error poses a low risk if the unreachable code does something minor like changes the color of the text on the screen. However, it's high risk if the unreachable code turns on the life support system in a hospital room. This is why context determines the potential severity of an error.
- Because you probably won't have the time and resources to carefully review every part of every product created on a software project, use risk analysis to focus your attention. Think about the parts of the product that could pose a greater risk of either having undiscovered defects or having defects cause serious problems and wasted time down the road. Those parts of the product are excellent candidates for being reviewed by other team members.



- If it costs US$1 to fix a defect found in the requirements phase, it costs US$2.40 in the design phase, US$25 in coding, US$200 in testing, and US$1000 (according to this data) if the requirements defect was not found until the product was released into operation.
- Costs increase because of the time required to make multiple changes due to an error, rebuild, retest, redocument, reinstall, customer support (like help desk), mailing new disks to customers, maintaining a Web site to download patches, etc.
- The earlier we find an error, the less damage it can do to the system.



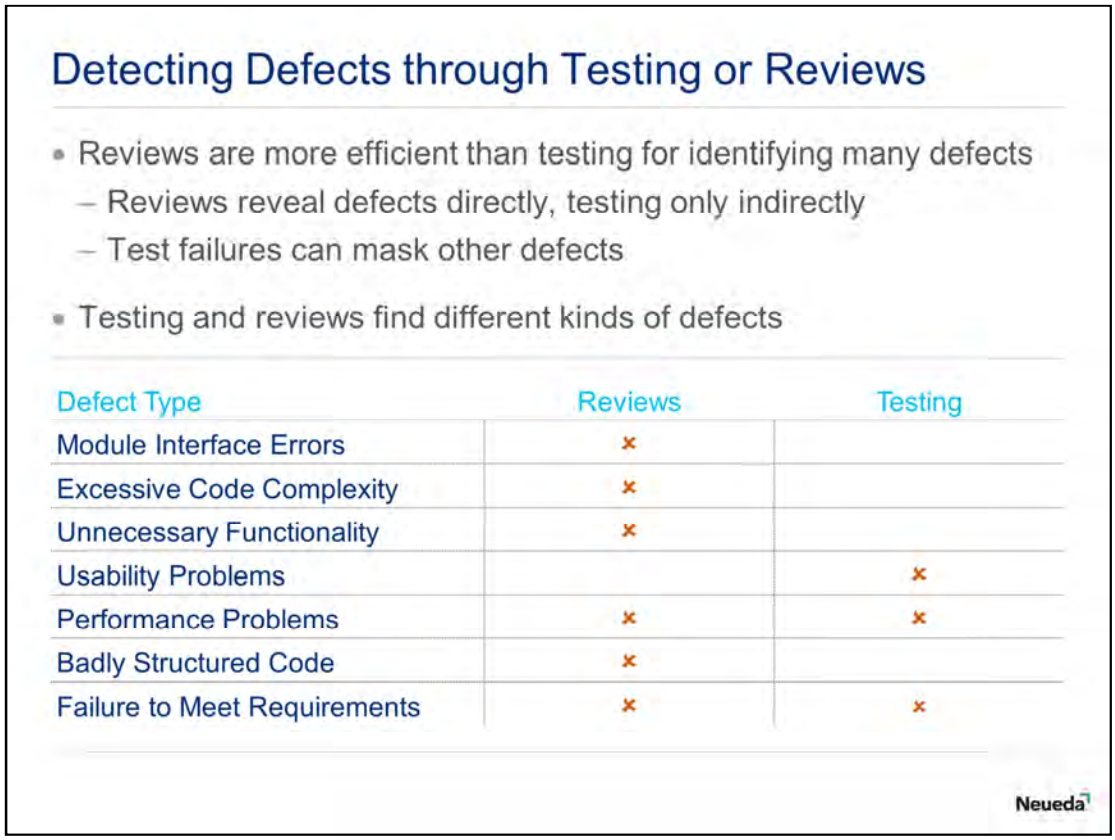
- If it costs US$1 to fix a defect found in the requirements phase, it costs US$2.40 in the design phase, US$25 in coding, US$200 in testing, and US$1000 (according to this data) if the requirements defect was not found until the product was released into operation.
- Costs increase because of the time required to make multiple changes due to an error, rebuild, retest, redocument, reinstall, customer support (like help desk), mailing new disks to customers, maintaining a Web site to download patches, etc.
- The earlier we find an error, the less damage it can do to the system.

Published Benefits from Software Inspections

Company	Cost Impact	Productivity Impact	Quality Impact
Aetna Insurance Company		25% increase in coding productivity	Inspections found 82% of errors
AT&T Bell Laboratories	Cost of finding errors reduced 10X by inspections	14% increase	10X improvement
Bell Northern Research	Avoided 33 hours of maintenance/defect found	Inspections 2–4X faster than testing	Found 80% of all defects by inspection
Bull HN Information Systems		Late stage testing period shortened because defects removed earlier by inspection	Inspections found 70–75% of all defects
Hewlett-Packard	Save US$21.4 million/year; 10X return on investment	Reduced time to market by 1.8 months	
IBM	Cost of rework of defects found by inspection is 10% that by testing	1 hour of inspection time saved 20 testing hours and 82 defect correction hours	
Jet Propulsion Laboratory	US$7.5 million savings from 300 inspections		
Litton	Saved 63.4 staff hours per inspection		76% fewer defects in integration
Microsoft		Cost to find and fix errors by testing is 4X more than by code inspection	
Motorola	Saved US$2.5 million in one year on one project		

Neueda

- Many articles have been published on inspection techniques and results since inspection method was first announced in the mid-1970s. Here is some data collected from the published software literature.
- These benefits are for doing formal inspections, not informal “look this over for me” reviews. Because people rarely collect data from informal reviews, it’s hard to know how effective they are.
- The limited data available all indicates that inspections are more effective at finding defects than are walkthroughs or other less formal types of peer reviews.

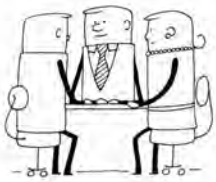


- It's not obvious to many people why reviews are more efficient than testing for finding defects, although several studies have shown that this is the case for many types of defects. The two main reasons are shown here.
- When you execute a test, you do not spot a defect. Instead you observe a failure in which the system does not behave as you expected it to under some conditions. Then you have to do some debugging to find the underlying defect (often called a fault) that led to the failure. In contrast, during review you're looking directly at the defect.
- Sometimes you encounter a failure that blocks you from being able to continue the testing process until that underlying fault is corrected. This can slow down the testing process. However, defects that you find during review do not usually prevent you from finding other defects during that same review.
- I'm not suggesting that you abandon testing in favor of reviews. Testing and reviews find different kinds of defects, so both techniques should be part of the developer's quality tool kit.

Informal and Formal Reviews

Informal Reviews

- No defined process or participant roles
- Usually ad hoc, rather than planned
- Examples: Walkthrough, peer deskcheck, passaround

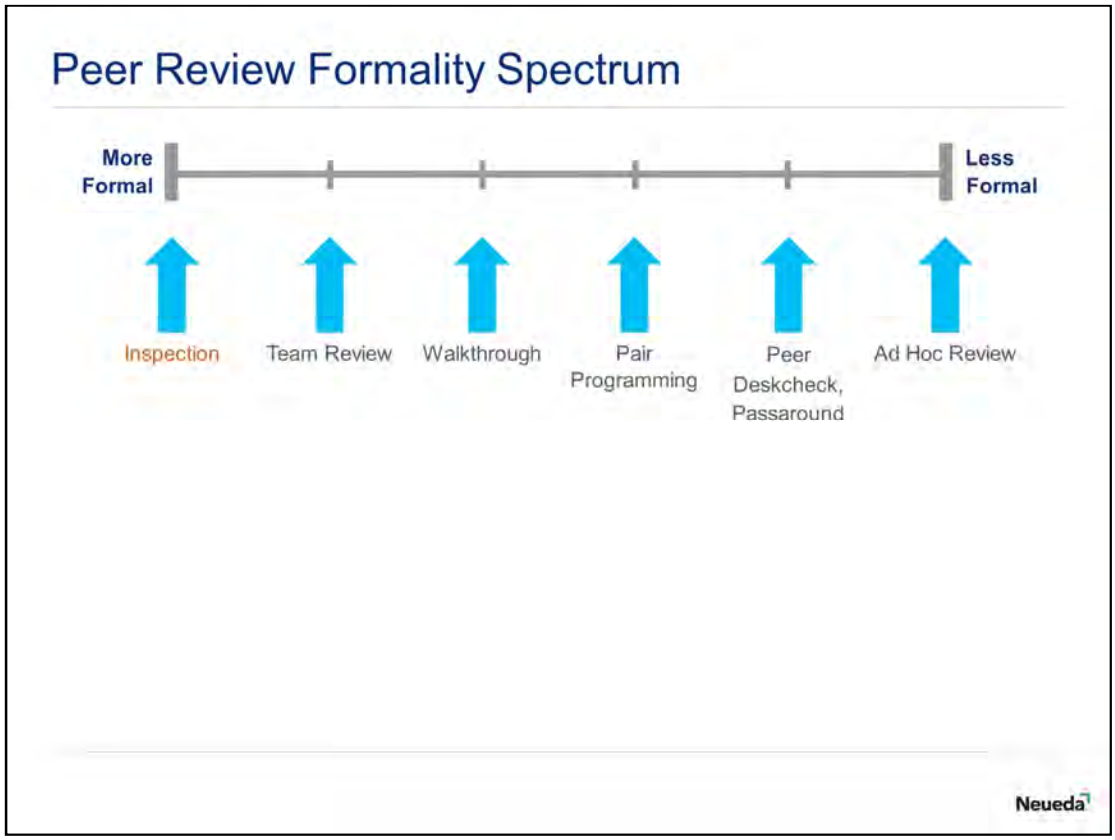


Formal Reviews

- Defined review objectives
- Follow a documented review process
- Defined participant roles and trained team
- Checklists, rules, or analysis methods used to find defects
- Management report produced on status of the product
- Data collected for quality control and process management

Neueda

- There are various types of both formal and informal reviews.
- I'll talk more about benefits and shortcomings of both later in the course.
- Some more characteristics of formal reviews: the review has stated objectives; the meeting is preplanned and is structured; the participants come prepared.

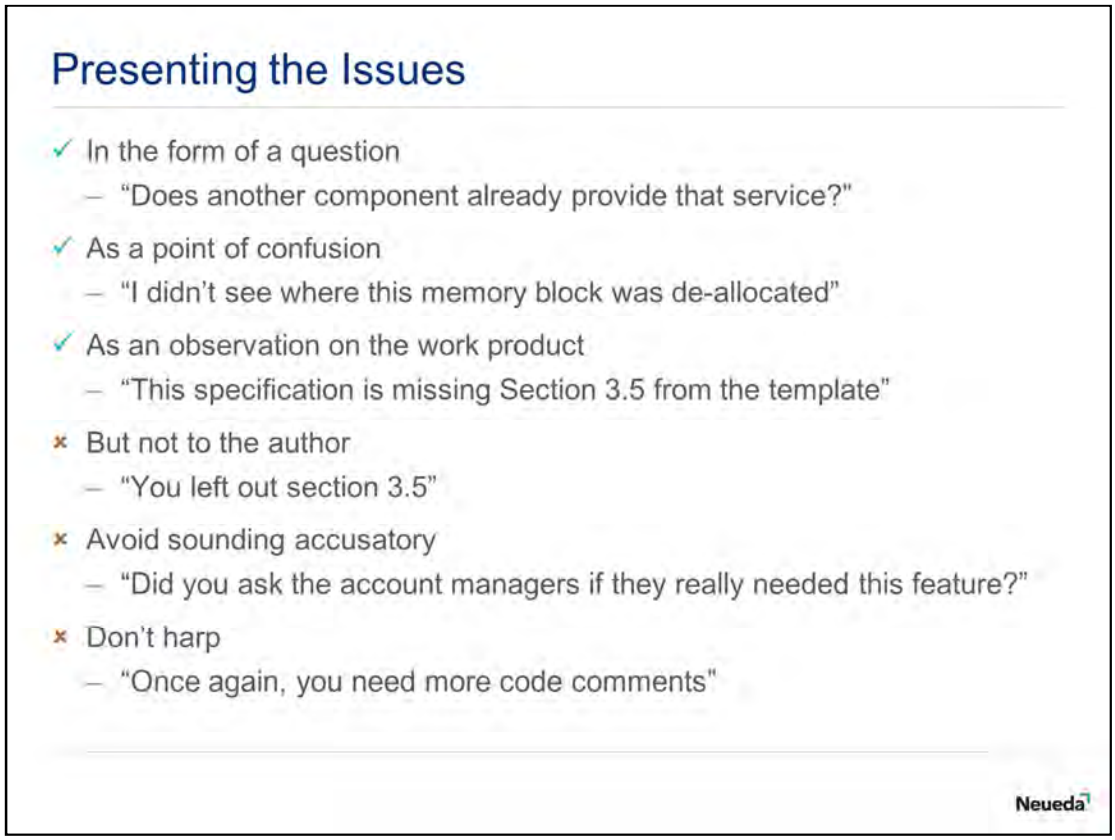


- This scale shows the range of formality for different kinds of peer reviews. The inspection is the most formal approach. Learn how to do inspections, instead of hoping to move to inspection after doing other, less formal reviews. We'll talk about the other review methods later.
- The team review has many of the characteristics of a formal inspection, but uses a simplified process.
- A walkthrough is an informal review the author leads.
- Pair programming is an aspect of Extreme Programming, in which 2 developers work together and continuously review each other's work. This is valuable that is not the same thing as having an independent observer look at the work from a fresh perspective.
- In a peer deskcheck, you ask a colleague to look at your work product. Peer deskchecks could use systematic defect detection methods and keep records.
- A passaround is a multiple, concurrent peer deskcheck.

- You typically want to have four perspectives represented in an inspection:
- The author of the work product and perhaps a peer of the author
- The author of any predecessor specification document (supplier)
- Anyone who will be using this work product in a downstream process

(customer, or “victim,” of the product, such as a tester who has to write tests from a requirements specification)

- Anyone who is responsible for related work products to which this one will interface in some way



- The way you make comments during a review has a big impact on how receptive the author is to what you have to say.
- I like to point out observations I made about the product, or things that confused me.
- Notice how the word "you" doesn't appear in the examples of good ways to present issues. Talk about the work product or what you saw; don't talk about what the author did or didn't do.

Peer Reviews and Diversity

- Frank criticism isn't acceptable in some cultures
  - Publicly airing defects in a meeting is uncomfortable
  - Offering suggestions to managers isn't done
- Distance separation requires distributed reviews
  - Need a capable moderator
  - Rotate meeting times to spread the inconvenience
  - Hold face-to-face initial meeting to establish rapport
- If necessary, collect input anonymously
  - Use peer deskchecks or passarounds

Neueda

- You need to consider the culture of the organisation when determining whether peer reviews will work and how best to incorporate them into the team's practices.

Some Best Practices for Code Reviews

1. Review fewer than 200–400 lines of code at a time
2. Aim for an inspection rate of less than 300–500 LOC/hour
3. Take enough time for a proper, slow review, but not more than 60–90 minutes
4. Authors should annotate source code before the review begins
5. Establish quantifiable goals for code review and capture metrics so you can improve your processes
6. Checklists substantially improve results for both authors and reviewers
7. Verify that defects are actually fixed!
8. Managers must foster a good code review culture in which finding defects is viewed positively
9. Beware the “Big Brother” effect
10. The Ego Effect: Do at least some code review, even if you don’t have time to review it all

Neueda

- Final thought to take away: Just as we cannot test quality in, we cannot review quality in.
- We can only use reviews to find places where the quality is not as high as it should be, and to help us learn how to do a better job next time.
- This is the place to play the Collaborator Short Demo Video

Review Exercise

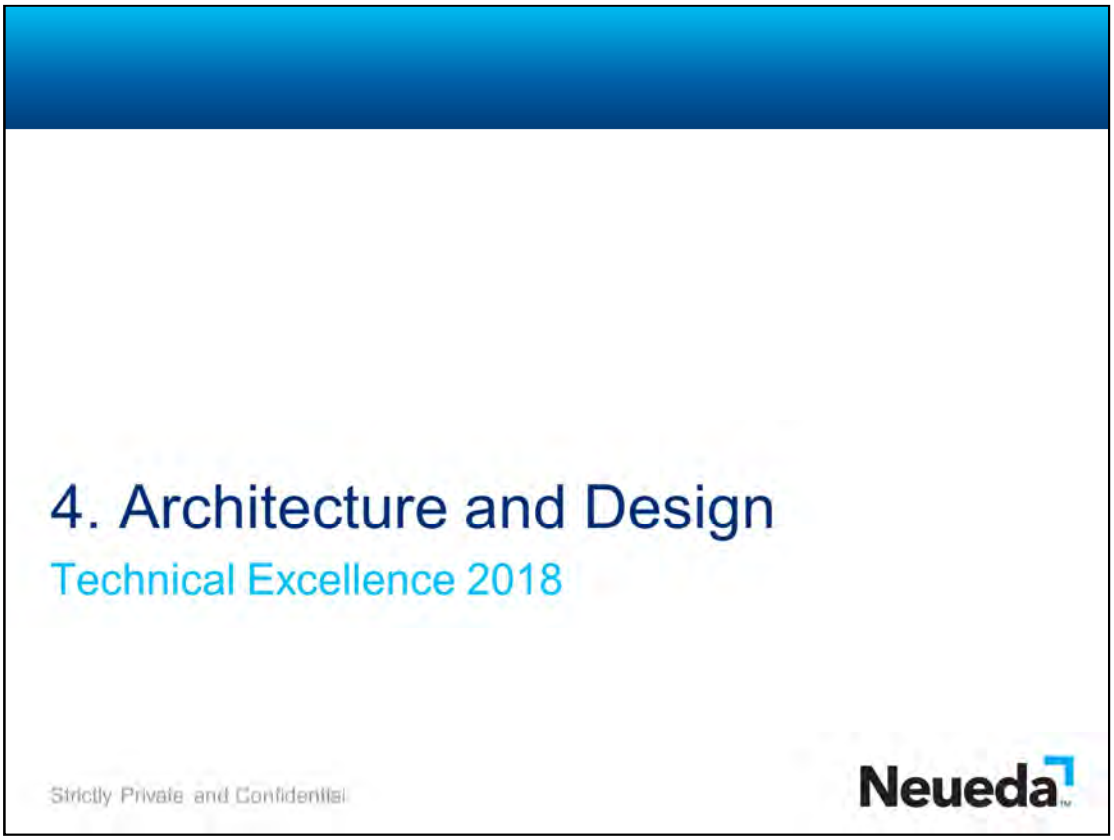
- We have provided a series of artefacts for review
- Depending upon your expertise you can select any from the following
  - Requirements Specification
  - Test plan
  - Java Component
  - C# Component
  - Run Book

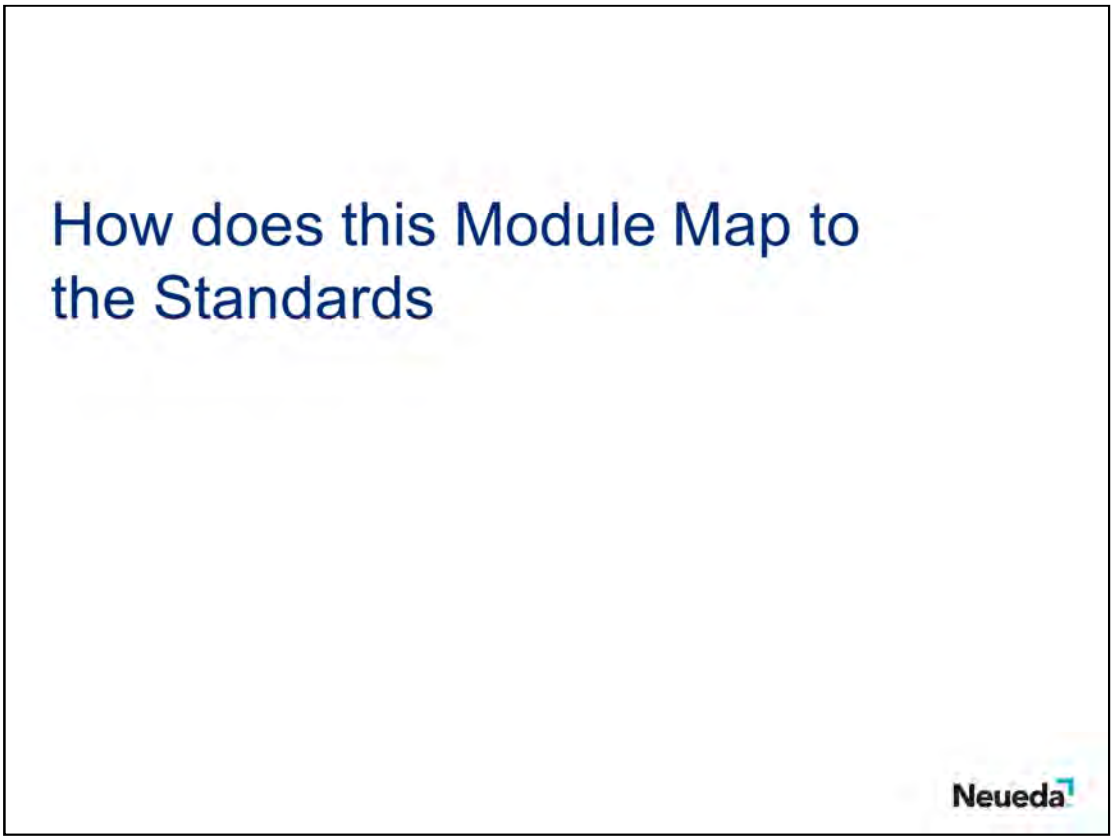
Neueda

For Further Reading

- Gilb, Tom and Dorothy Graham. Software Inspection. Reading, Mass.: Addison-Wesley, 1993
- Radice, Ronald A. High Quality Low Cost Software Inspections. Andover, Massachusetts: Paradoxicon Publishing, 2002
- René, Marc, and Kathy Walker. "In Other Words…,"Better Software, vol. 6, no.2 (February 2004), pp. 18–21
- Wheeler, David A., Bill Brykczynski, and Reginald N. Meeson, Jr. Software Inspection: An Industry Best Practice. Los Alamitos, California: IEEE Computer Society Press, 1996
- Wiegers, Karl. "The Seven Deadly Sins of Software Reviews," Software Development, vol. 6, no.3 (March 1998), pp. 44–47
- Wiegers, Karl E. Peer Reviews in Software: A Practical Guide. Reading, Mass.: Addison-Wesley, 2002

Neueda

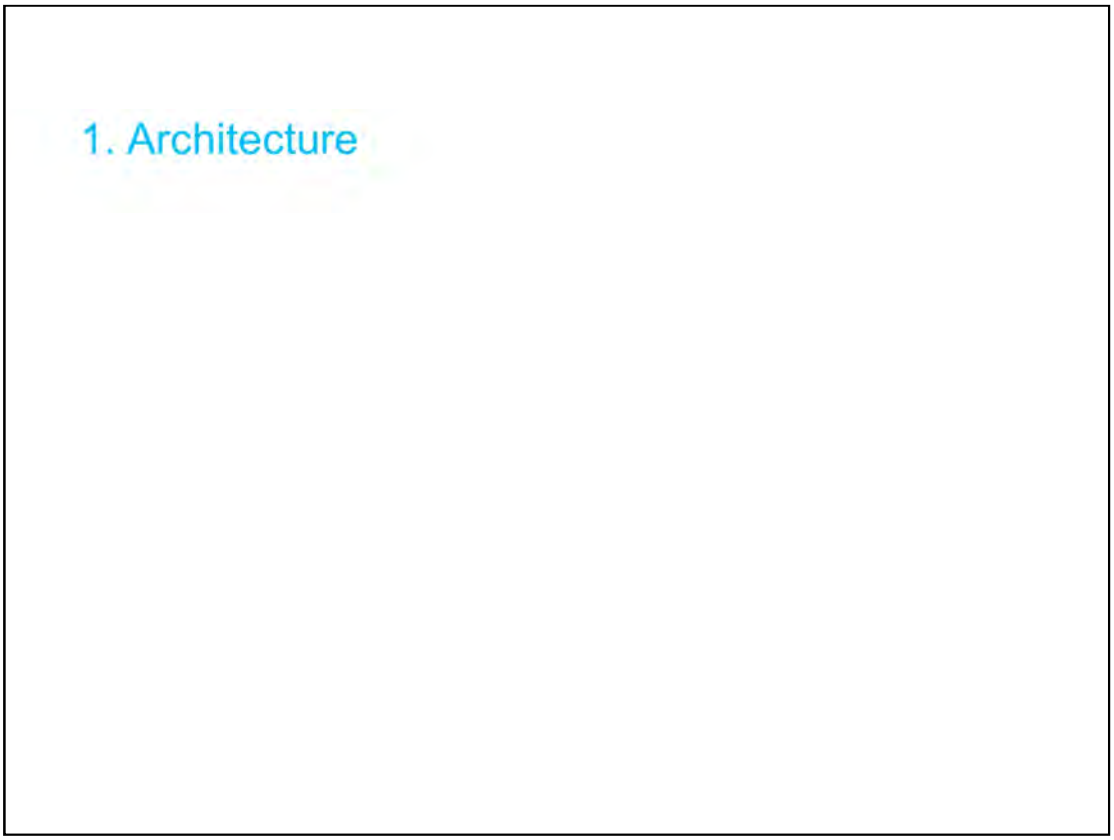

4. Architecture and Design
Technical Excellence 2018
Strictly Private and Confidential
Neueda


How does this Module Map to the Standards
Neueda

Objectives

- Software Architecture
- Network Architecture
- Software Design Patterns

Neueda


1. Architecture

Key Questions

- What do we mean by architecture?
- What are some properties of a good architecture?
- How and when should code be reused?
- How can patterns aid design reasoning?

Neueda

What Do We Mean by Architecture?

- Various people have attempted to define architecture
- The following pages show some quotes on what architecture is

Neueda

[ ]

The architecture of a software system defines that system in terms of computational components and interactions among those components

Mary Shaw and David Garlan
Carnegie Mellon University

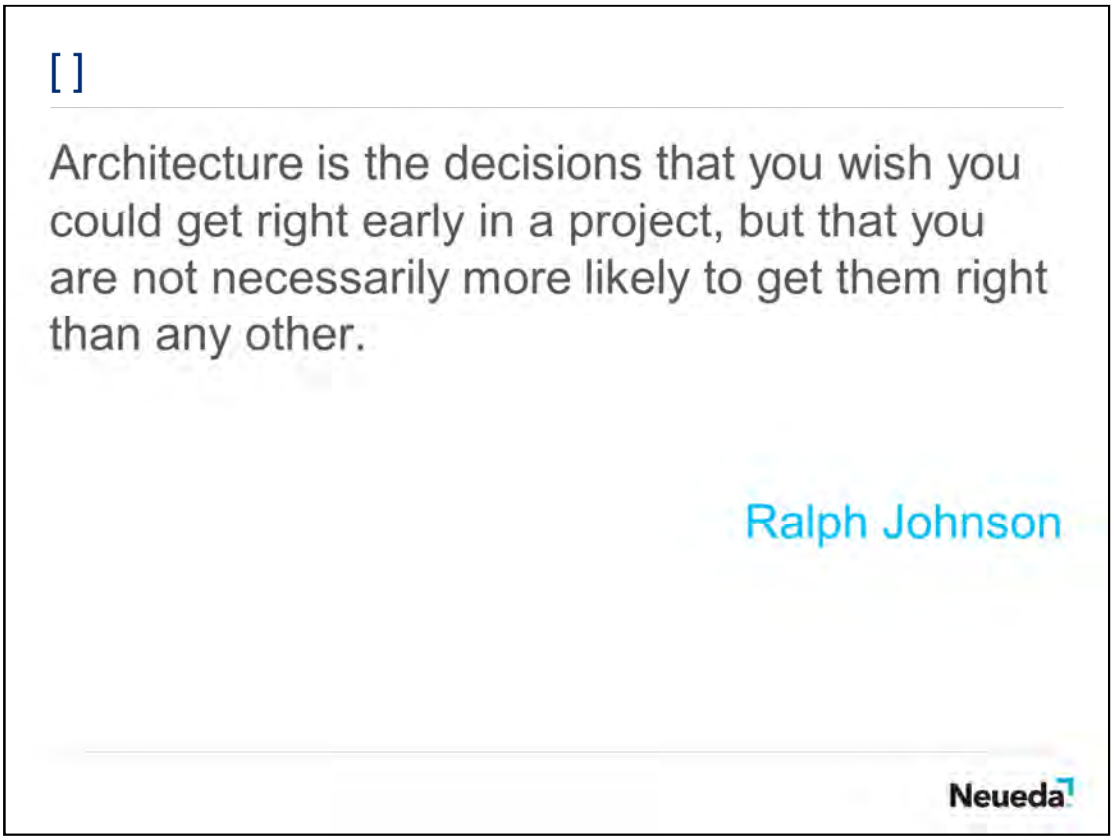
Neueda

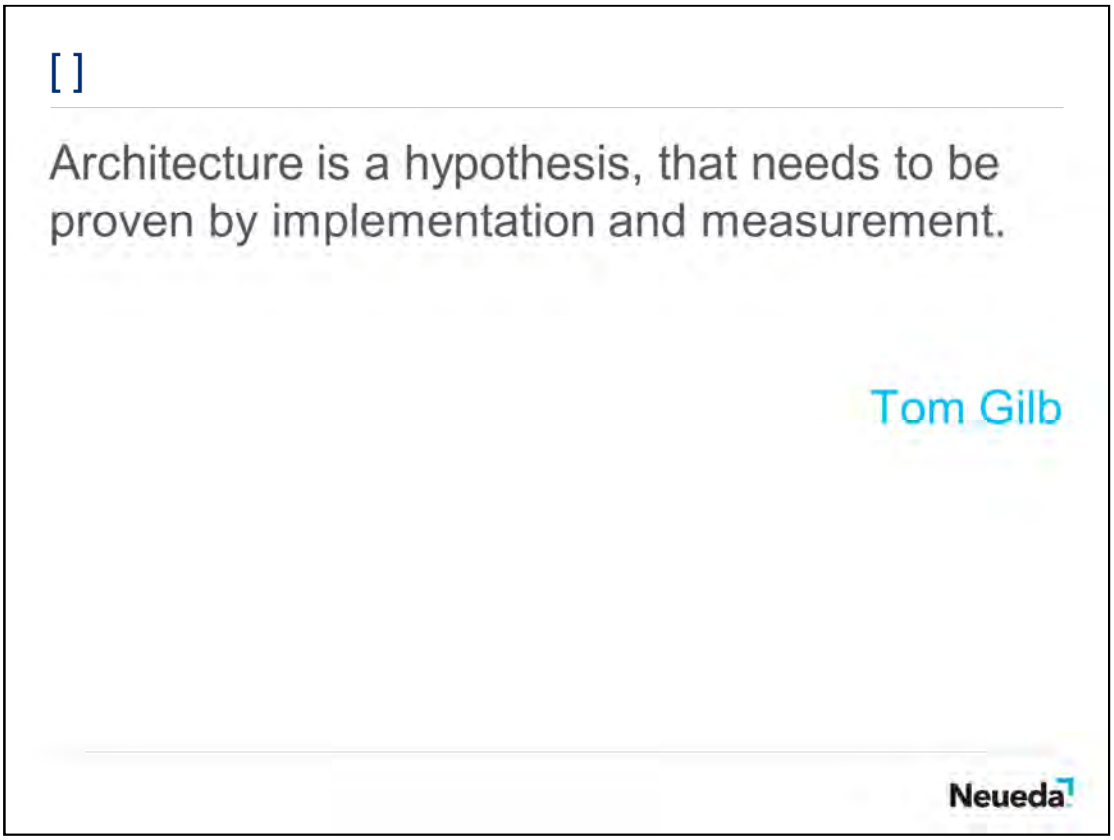
[ ]

"Architecture" is a term that lots of people try to define, with little agreement. There are two common elements: One is the highest-level breakdown of a system into its parts; the other, decisions that are hard to change.

Martin Fowler

Neueda


[ ]
Architecture is the decisions that you wish you could get right early in a project, but that you are not necessarily more likely to get them right than any other.
Ralph Johnson
Neueda


[ ]
Architecture is a hypothesis, that needs to be proven by implementation and measurement.
Tom Gilb
Neueda

[ ]

There are standard precautions that can help reduce risk in complex software systems. This includes the definition of a good software architecture based on a clean separation of concerns, data hiding, modularity, well-defined interfaces, and strong fault-protection mechanisms.

Gerard J Hozmann
"Mars Code", CACM 57(2)

cacm.acm.org/magazines/2014/2/171689-mars-code/fulltext

Neueda

[ ]

If you think good architecture is expensive, try bad architecture.

Brian Foote and Joseph Yoder
Big Ball of Mud

Neueda

Architecture in Summary

- Architecture represents the design decisions that are hard to change
- All systems have an architecture, whether intended or not, whether recognised or not, whether good or not
- An architecture should be considered a set of hypotheses to be evaluated
- Architecture involves invention and discovery (and surprise)
- Architecture is influenced by process and vice versa

Neueda

Impact of Architecture

- Architecture impacts the non-functional requirements of a system including
  - Performance
  - Reliability
  - Flexibility
  - Maintainability

Neueda

Performance

- Computer performance is characterised by the amount of useful work accomplished by a computer system or computer network compared to the time and resources used. Depending on the context, high computer performance may involve one or more of the following
  - Short response time for a given piece of work
  - High throughput (rate of processing work)
  - Low utilisation of computing resource(s)
  - High availability of the computing system or application

http://en.wikipedia.org/wiki/Computer_performance

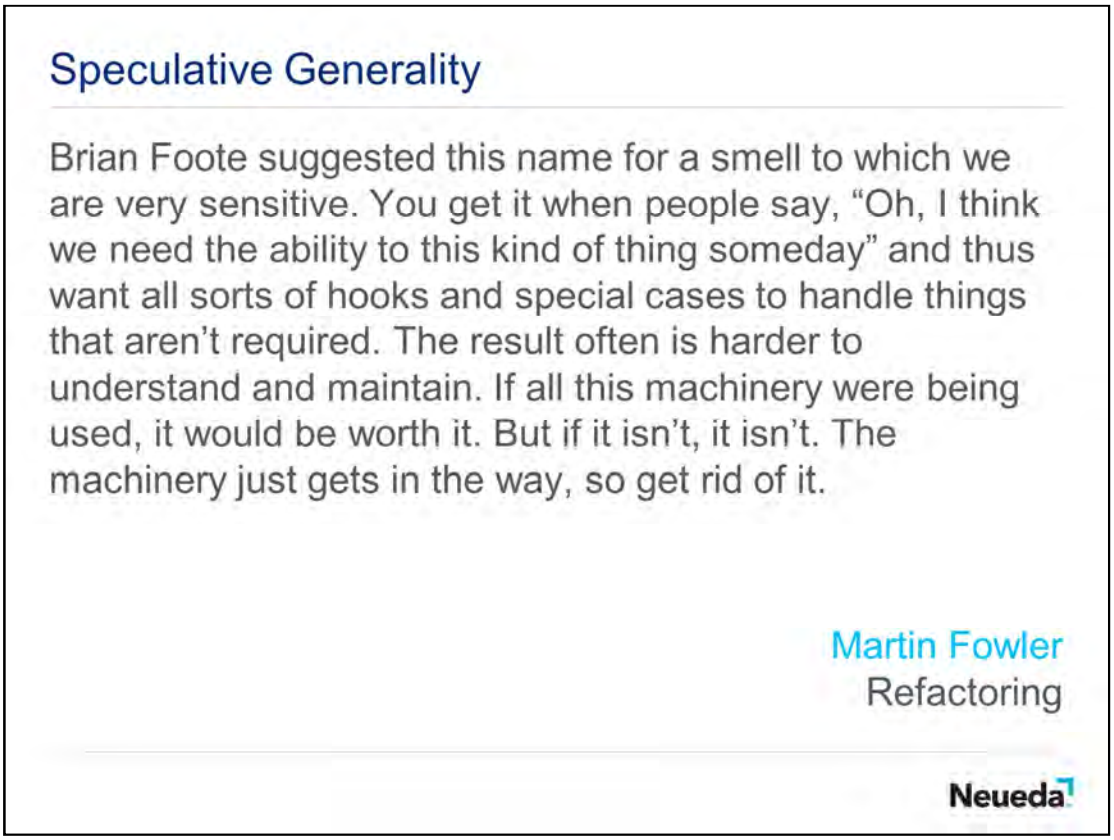
Neueda

Flexibility

Because the design that occurs first is almost never the best possible, the prevailing system concept may need to change. Therefore, flexibility of organisation is important to effective design.

Fred Brooks

Neueda



YAGNI – you ain’t gonna need it!

Layers

Define one or more layers for the software under development, with each layer having a distinct and specific responsibility.

Frank Buschmann, Kevlin Henney and Douglas C Schmidt
Pattern-Oriented Software Architecture, Volume 4: A Pattern Language for Distributed Computing

Neueda

Software Architecture Best Practices

- An architecture should support the changes that it experiences
- Do not use speculation to add extra complexity to the architecture
- Aim for use before reuse
- Reduce dependencies before increasing reuse
  (reuse introduces dependencies)
- Reusable components need tests, documentation and clear ownership and support

Neueda

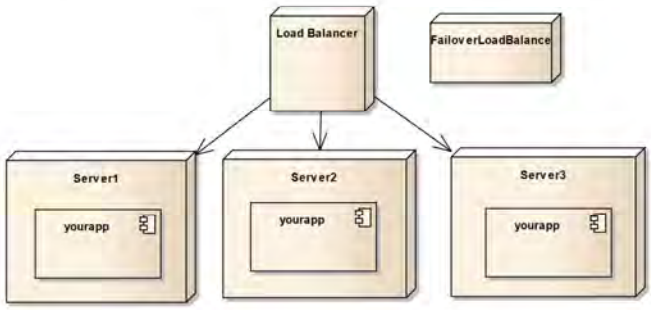
Network Architecture

- Migrating applications to cloud platforms is driving a renewed focus on architecture and new principles
  - All services should support horizontal scaling
  - All services should be event driven
  - All services should target zero downtime
  - Resiliency – applications pass annual continuity of business test and critical applications are FORT tested
- To support these principles
  - Servers must be stateless
  - Servers must be decoupled

Neueda

Stateless Servers

- Consider the following architecture
- If any of one of these servers fail, it will not matter as new ones can be spun up (potentially automatically)
  - If however state is stored on these servers then server failure is a problem

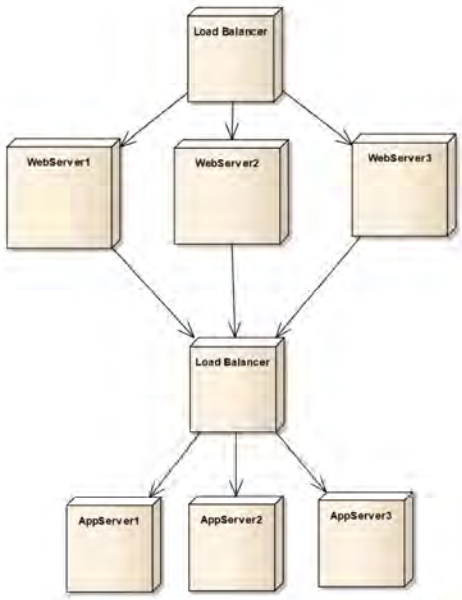


Neueda

Ask how this architecture relates to the standards on the previous page.

Loose Coupling Sets You Free

- Servers should be decoupled
- Allows servers to be added and removed from the fleet depending upon load
- In the example shown, Web servers do not know about individual app servers

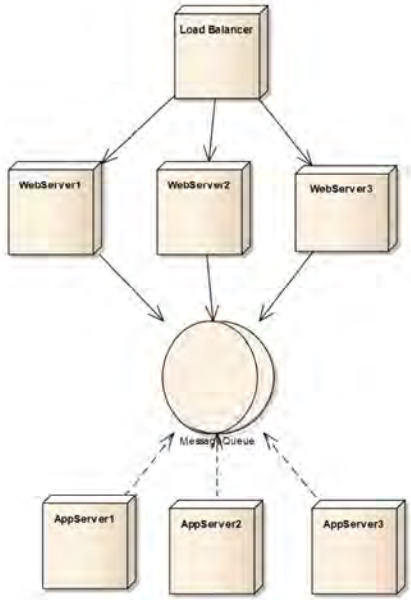


Neueda

Ask how this architecture relates to the standards on the previous pages.

Decoupling with Messaging

- A message queue can be used resulting in further decoupling and an event driver architecture
  - Now the Web servers can function even if no app servers are running at all
  - The Queue would fill up until the app servers came back online



Neueda

Ask how this architecture relates to the standards on the previous pages.

2. Software Design Patterns

[ ]

A pattern is more than just a solution structure, so its audience must also have a sense of the context, the forces, and the consequences that are associated with a solution.

Neueda

What are Patterns?

- A pattern expresses a reusable design idea or approach
- A pattern is not just a solution: It involves reasoning from problem to solution; it includes a rationale
- A collection of patterns forms a style
- Appropriate use of patterns is based on seeing a fit between a problem and its context and one or more solutions
- Inappropriate use of patterns comes from cargo cult application of design ideas without regard for context

Neueda

Creational Patterns

- Creational patterns are used to create objects in a manner suitable to the situation
  - Singleton
  - Factory
  - Abstract Factory
  - Builder
  - Prototype

Neueda

Exercise 1

- A common example used in books and online for creational patterns is a Pizza Factory
- Your task is to come up with some candidate designs for how a Pizza Factory might be built

Neueda

Pizzas

- There are several different pizzas on the menu
  - Ham and Pineapple
  - Meat Feast
  - Vegetarian
- Each Pizza consists of a
  - Base
  - Topping
  - Sauce

Neueda

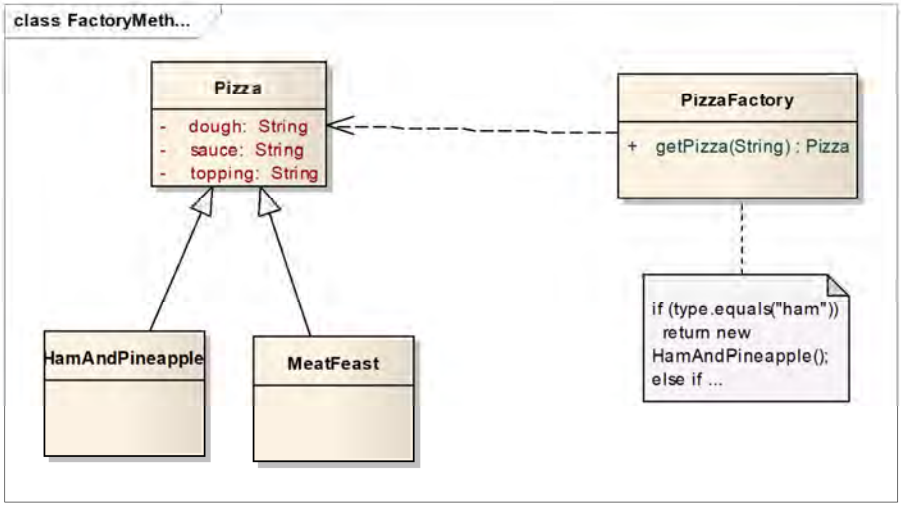
Deliverables

- Create a class diagram using your preferred notation to define how your pizzas will be made

Neueda

Solution 1: Factory

- You could simply use the Factory pattern



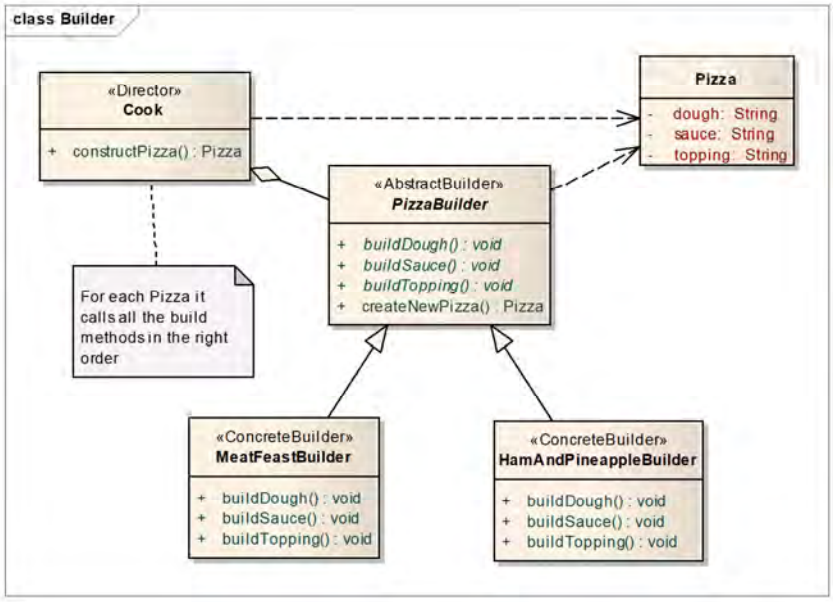
Neueda

Solution 2: Builder

- The Builder pattern allows you to abstract the building of a complex object into steps
  - Step 1: Do the base
  - Step 2: Do the sauce
  - Step 3: Do the topping

Neueda

Builder UML



Neueda

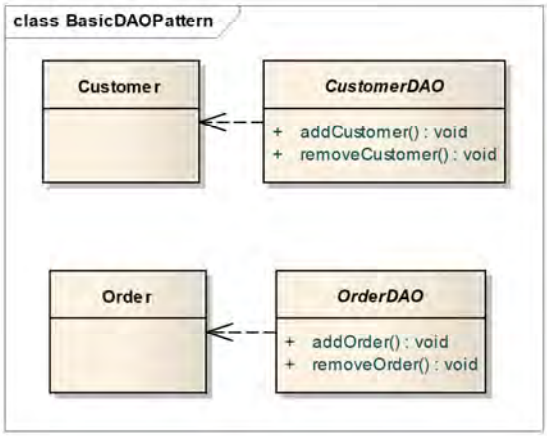
Builder Participants

- The Concrete Builder classes have methods to build the various parts of the pizza
- The Director class uses a ConcreteBuilder and works with their methods to build the various bits of Pizza
  - The Director does not know which Concrete Builder they are using

Neueda

Exercise 2 – Introduction

- You may have heard of the Data Access Object pattern
- This pattern involves classes that carry out database access on behalf of entity classes



Neueda

Exercise 2

- DAOs can be built using different data access technologies
  - E.g. Hibernate or JDBC
- How could you build your DAO library, so that you would only have to change one line of code to switch from say a Hibernate based set of DAOs to a JDBC based set of DAOs?

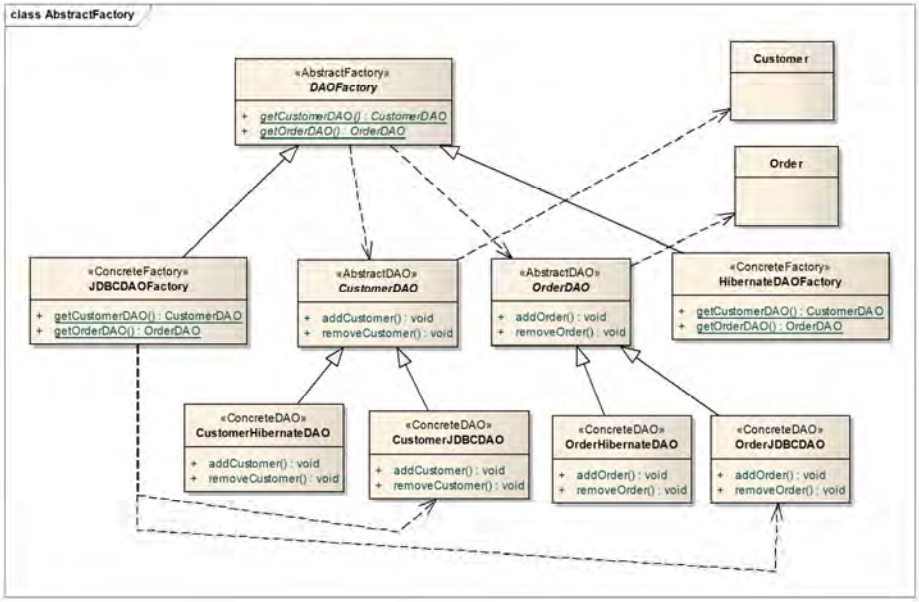
Neueda

DAO Solution

- The pattern that helps to create a family of objects, like a family of DAOs is the Abstract Factory Pattern

Neueda

[ ]


class AbstractFactory
«AbstractFactory»
DAOFactory
+ getCustomerDAO() : CustomerDAO
+ getOrderDAO() : OrderDAO
Customer
Order
«ConcreteFactory»
JDBCDAOFactory
+ getCustomerDAO() : CustomerDAO
+ getOrderDAO() : OrderDAO
«AbstractDAO»
CustomerDAO
+ addCustomer() : void
+ removeCustomer() : void
«AbstractDAO»
OrderDAO
+ addOrder() : void
+ removeOrder() : void
«ConcreteFactory»
HibernateDAOFactory
+ getCustomerDAO() : CustomerDAO
+ getOrderDAO() : OrderDAO
«ConcreteDAO»
CustomerHibernateDAO
+ addCustomer() : void
+ removeCustomer() : void
«ConcreteDAO»
CustomerJDBCDAO
+ addCustomer() : void
+ removeCustomer() : void
«ConcreteDAO»
OrderHibernateDAO
+ addOrder() : void
+ removeOrder() : void
«ConcreteDAO»
OrderJDBCDAO
+ addOrder() : void
+ removeOrder() : void

Neueda

Behavioural Patterns

- Behavioural patterns are used to model standard solutions for capturing the behaviour of object oriented systems
  - Chain of Responsibility
  - Command
  - Iterator
  - Mediator
  - Publish/Subscribe
  - State
  - Strategy
  - Template Method
  - Visitor

Neueda

Exercise 3

- Problem statement
  - You have a hierarchy of classes containing a number of behaviours which must occur in a specific order
  - However, you want the flexibility to be able to change the actual behaviour implementations in the concrete classes but keep the order
- How do you design for that?

Neueda

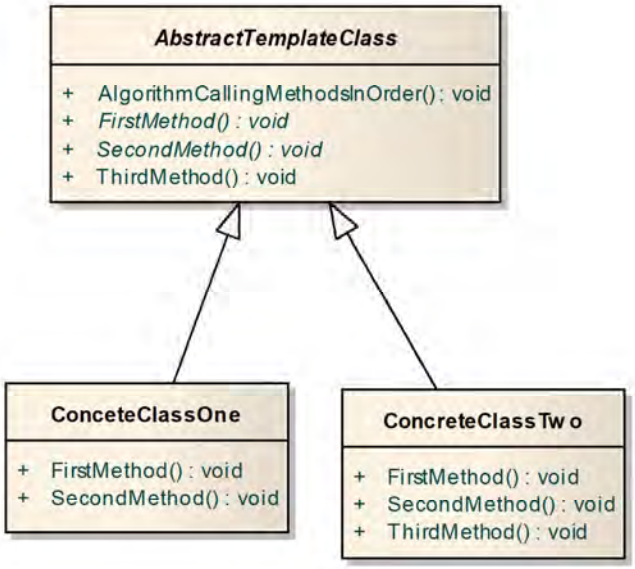
The Template Pattern Solution

- The Template pattern solves this problem
  - A Template class contains a set of algorithm methods which could be abstract or concrete
  - A Template class also contains a method that calls each of these methods in the correct order
  - Subclasses to the template then override any/all of the algorithm methods

Neueda

Template Pattern UML

- Here we see the Template class
- Here we see the subclasses overriding certain behaviours



Neueda

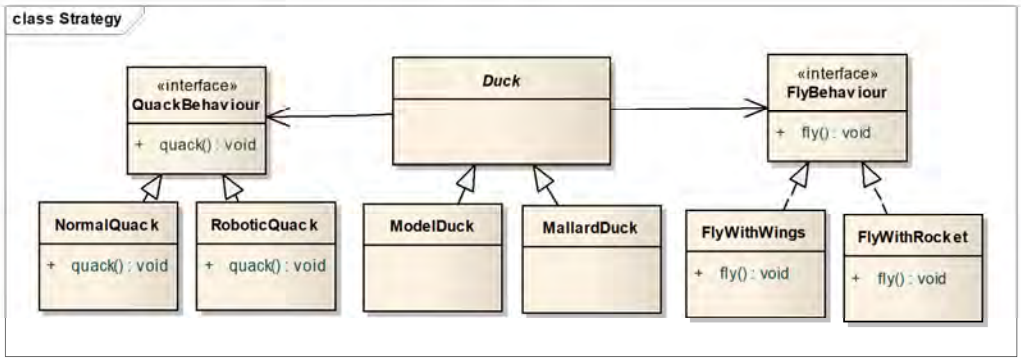
Exercise 4

- Problem Statement
  - You have a number of classes that can share specific behaviours
  - You want to be able to 'plug in' different behaviours into your classes
  - To use the example from Head First Patterns, you want to create many different types of Duck, each with 'pluggable' fly and quack capabilities

Neueda

Solution: Strategy Pattern

- Making ducks fly and quack with strategy



Neueda

Strategy

- Each behavioural strategy like fly() is modelled as a different class
- The behaviours are then 'plugged in' to the different types of class that need the strategies (the Ducks)

Neueda

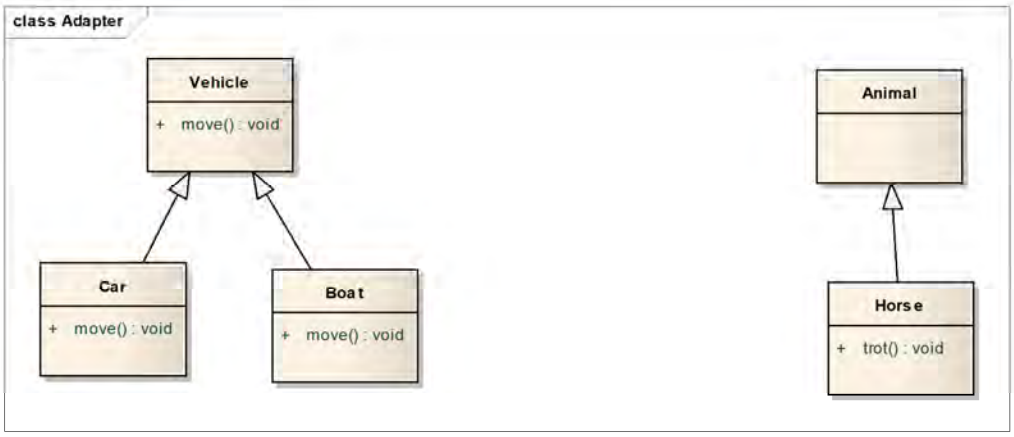
Structural Patterns

- Structural patterns address the structuring of class relationships to solve structural OO problems
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Façade
  - Flyweight

Neueda

Exercise 5

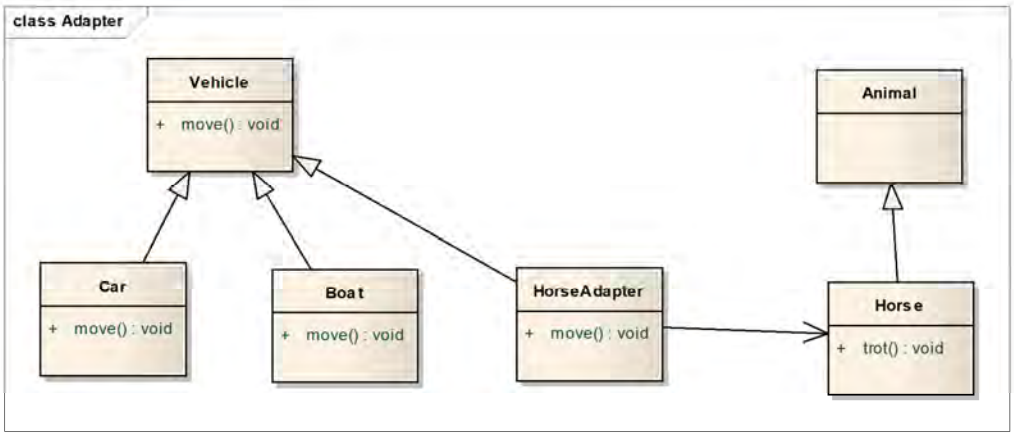
- We want a Horse to be a vehicle, but it is already an Animal! How do we solve it?



Neueda

Solution – Adapter

- The Adapter pattern is a simple but powerful design pattern to allow a class to be plugged into a different hierarchy



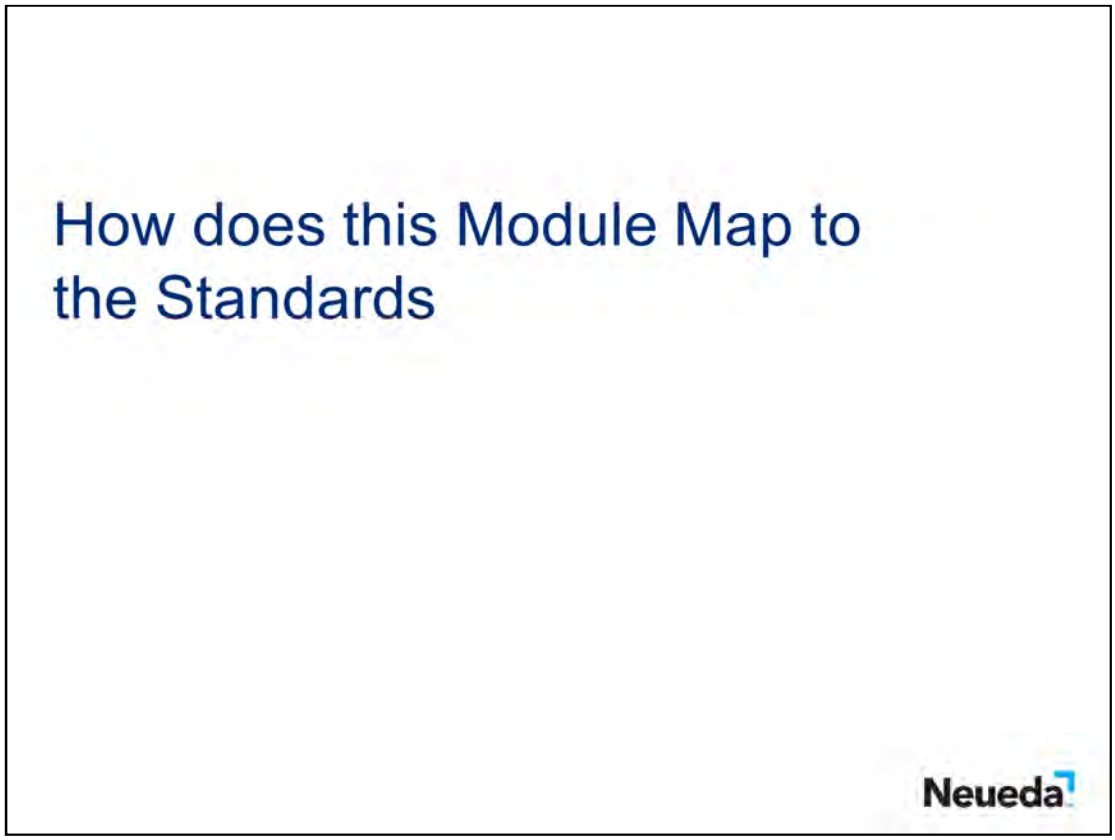
Neueda

Summary

- Software Architecture
- Network Architecture
- Software Design

Neueda


5. Continuous Integration, Delivery and Deployment
Technical Excellence 2018
Strictly Private and Confidential
Neueda


How does this Module Map to the Standards
Neueda

The Standards

- Change – deploy releases subject to appropriate risk, quality and rollback standards
- Single Click Deployment – build automated, reliable and repeatable releases

Neueda

Quick Poll No.1

Safe?

- Are you using version control?

Quick?

- Can you release new version of your software in one day?

Quick and Safe?

- Can you release new, well-tested version of your software in one day?

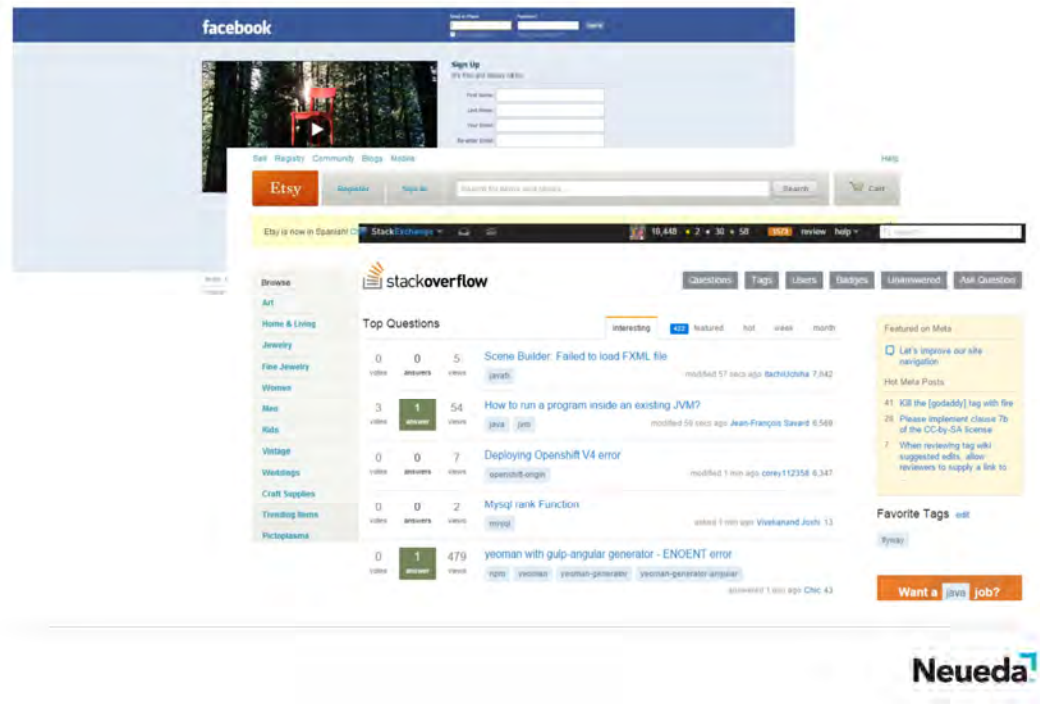
Neueda

Quick Poll No.2

- How often do you deploy into production?
- At least
  - Once a year
  - Every 3 months
  - Every Month
  - Every 2 weeks
  - Every week
  - Multiple times a week
  - Every day
  - Multiple times a day

Neueda

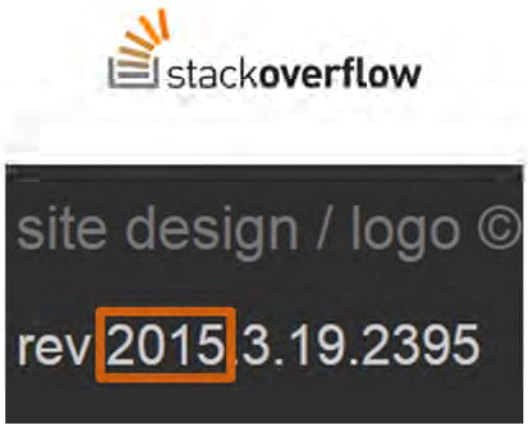
[ ]


facebook
Sign Up
Etsy
stackoverflow
Top Questions
Scene Builder: Failed to load FXML file
How to run a program inside an existing JVM?
Deploying Openshift V4 error
Mysql rank Function
yeoman with gulp-angular generator - ENOENT error
Favorite Tags
Want a java job?
Neueda

[ ]

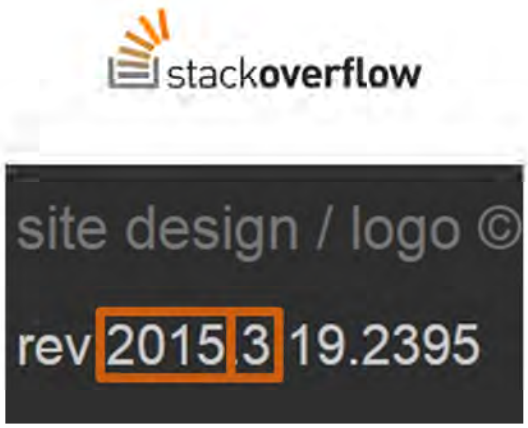

stackoverflow
site design / logo ©
rev 2015.3.19.2395
stackoverflow.com
Neueda

[ ]


stackoverflow
site design / logo ©
rev 2015.3.19.2395
stackoverflow.com

Neueda

[ ]


stackoverflow
site design / logo ©
rev 2015 3 19.2395
stackoverflow.com

Neueda

[ ]


stackoverflow
site design / logo ©
rev 2015.3.19.2395
stackoverflow.com
Neueda

1. Development Now
2. Solution – Continuous Integration
3. Requirements
4. Benefits
5. Continuous Delivery
6. Continuous Deployment
7. Transition
8. Notes from the Field
9. Deployment Issues
10. Deployment Best Practices – Documentation

citi

Development Now

- Each developer has feature branches
  - If the version control is used at all
- Features are deployed when completed
- Integration issues

- Small test suite

Neueda

Problems

- Bringing software into production is hard
- Takes a lot of time
- Error prone

Neueda

1. Development Now
2. Solution – Continuous Integration
3. Requirements
4. Benefits
5. Continuous Delivery
6. Continuous Deployment
7. Transition
8. Notes from the Field
9. Deployment Issues
10. Deployment Best Practices – Documentation

citi

Continuous Integration?

"Continuous Integration is 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."

– Martin Fowler

Neueda

Change the Workflow!

- Check-out/update
- Code
- Build and test locally
- Update (merge)
  - Retest if changed
- Commit
- Continuous Integration server takes over …

Neueda

Change the Versioning!

- No ‘feature’ branches
- Temporary ‘developer‘ branches
- Good to test crazy ideas
  - Branch and throw away
- Trunk must always compile
- Avoid big scary merges

Neueda

How to Handle Features

- No ‘feature’ branches
- Features can be toggled on and off via deployment or compilation configuration
  - Also helps with Continuous Delivery/Deployment
- Keep features small
- Improve features interactively
  - Introduce early, then improve

Neueda

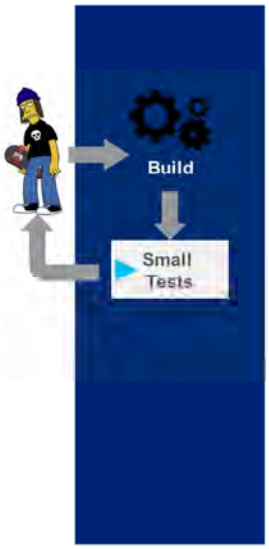
Testing

- Automate everything
  - If it hurts, do it more often. Continuously
  - Fail fast
- Integration testing
- Unit testing
- Functional testing
- Application testing
- Mobile testing

- Whatever you don't test against, will happen

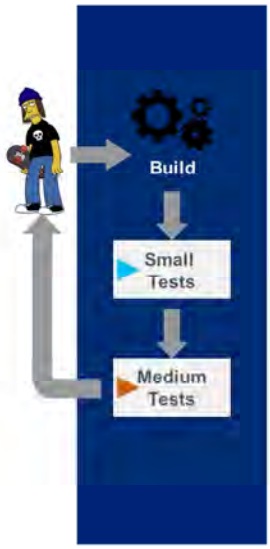
Neueda

[ ]



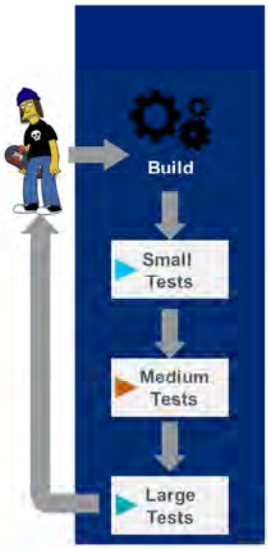
Neueda

[ ]



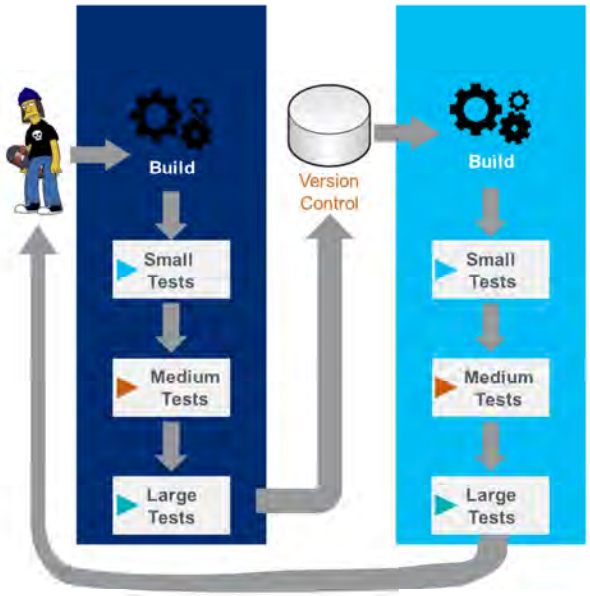
Neueda

[ ]



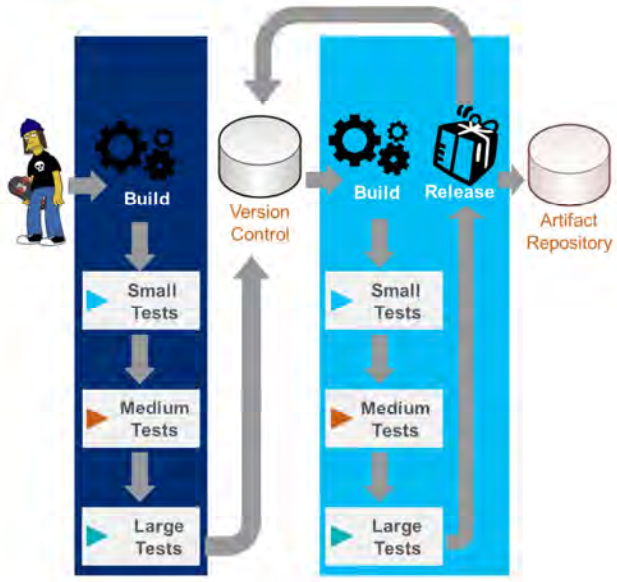
Neueda

[ ]


Build
Small Tests
Medium Tests
Large Tests
Version Control
Build
Small Tests
Medium Tests
Large Tests

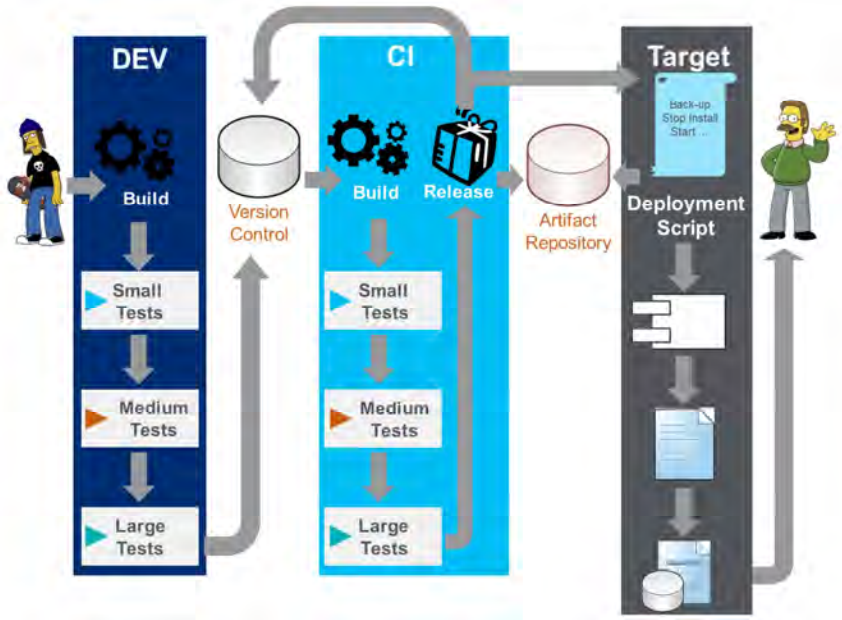
Neueda

Do Not Deploy Snapshots



Neueda

[ ]


DEV
Build
Small Tests
Medium Tests
Large Tests
Version Control
CI
Build
Release
Small Tests
Medium Tests
Large Tests
Artifact Repository
Target
Back-up
Stop Install
Start ...
Deployment Script

Neueda

1. Development Now
2. Solution – Continuous Integration
3. Requirements
4. Benefits
5. Continuous Delivery
6. Continuous Deployment
7. Transition
8. Notes from the Field
9. Deployment Issues
10. Deployment Best Practices – Documentation

citi

Requirements

- Source code repository (version control)
  - SVN, Git, Mercurial, …
- Project build
  - MSBuild
  - …
- Testing
- Continuous Integration server

Neueda

CI Server in Essence

```
while true do
begin
    if change_checked_into_vcs then
    begin
        if not build then
            report_error;
        if not test then
            report_error;
    end;
    sleep;
end;
```

Neueda

CI Servers

- Jenkins
  - Hudson fork
  - Java
  - Commercial support – Cloudbees
- CruiseControl.Net
  - C#
  - XML configuration: (
- FinalBuilder, Automated Build

Neueda

CI Recommendation

- Use a separate server (or VM)
  - For CI, or
  - For CI + build, or
  - For CI + build + test

Neueda

Central Dogma

- Build early, build often
  - On every check-in
  - Check in early, check in often

Neueda

1. Development Now
2. Solution – Continuous Integration
3. Requirements
4. Benefits
5. Continuous Delivery
6. Continuous Deployment
7. Transition
8. Notes from the Field
9. Deployment Issues
10. Deployment Best Practices – Documentation

citi

Benefits

- Brings order into chaos
- Everything could be achieved without the Continuous Integration, but …
- CI is the great enforcer

Neueda

Benefits (Cont'd)

- Code is always in the consistent state
- Code always compiles
- Automatic tests
- Automatic feedback on production readiness

Neueda

Code Always Compiles

Neueda

Code Always Compiles (Cont'd)

- Code should always build and test
  - Continuous Delivery

Neueda

1. Development Now
2. Solution – Continuous Integration
3. Requirements
4. Benefits
5. Continuous Delivery
6. Continuous Deployment
7. Transition
8. Notes from the Field
9. Deployment Issues
10. Deployment Best Practices – Documentation

citi

Continuous Delivery?

"The essence of my philosophy to software delivery is to build software so that it is always in a state where it could be put into production. We call this Continuous Delivery because we are continuously running a deployment pipeline that tests if this software is in a state to be delivered."

– Jez Humble, Thoughtworks

Neueda

CI <> CD

- CD = CI + fully automated test suite
- Not every change is a release
  - Manual trigger
  - Trigger on a key file (version)
  - Tag releases!
- CD – It is all about testing!

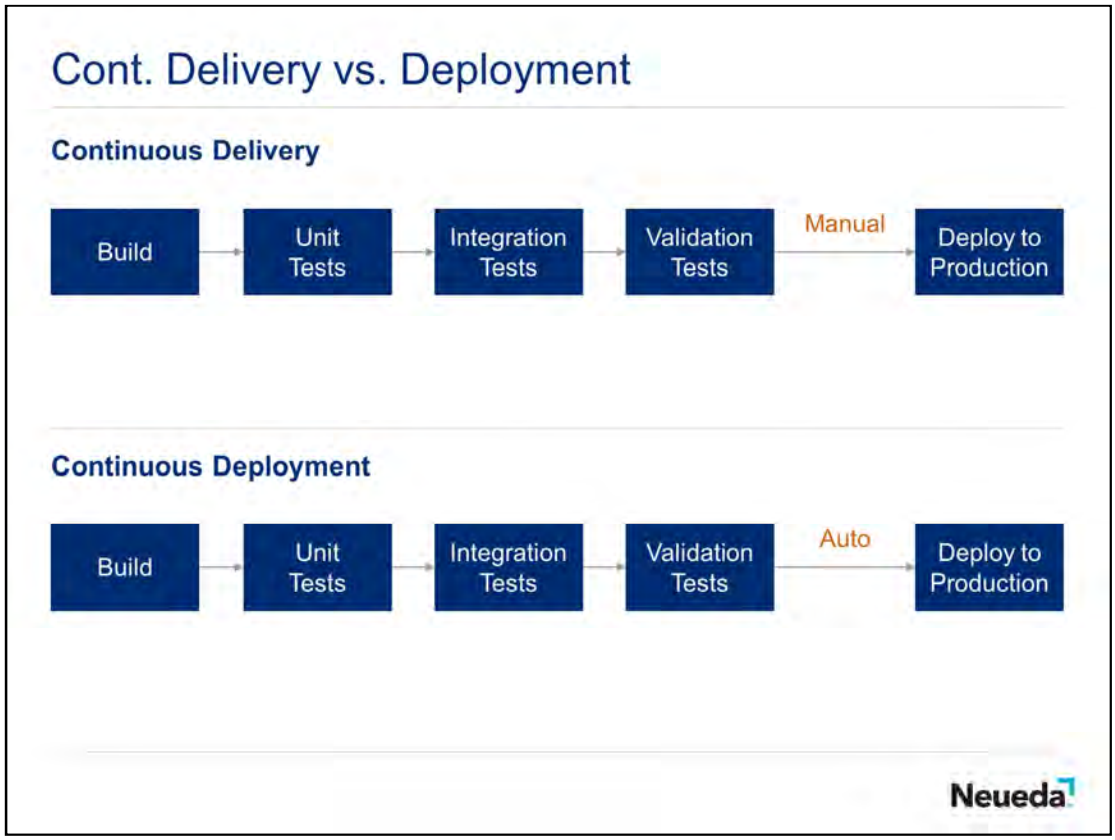
Neueda

Consider this

"How long would it take your organisation to deploy a change that involves just one single line of code?"

– Mary and Tom Poppendieck,
Implementing Lean Software Development

Neueda


Cont. Delivery vs. Deployment
Continuous Delivery
Build
Unit Tests
Integration Tests
Validation Tests
Manual
Deploy to Production
Continuous Deployment
Build
Unit Tests
Integration Tests
Validation Tests
Auto
Deploy to Production
Neueda

1. Development Now
2. Solution – Continuous Integration
3. Requirements
4. Benefits
5. Continuous Delivery
6. Continuous Deployment
7. Transition
8. Notes from the Field
9. Deployment Issues
10. Deployment Best Practices – Documentation

citi

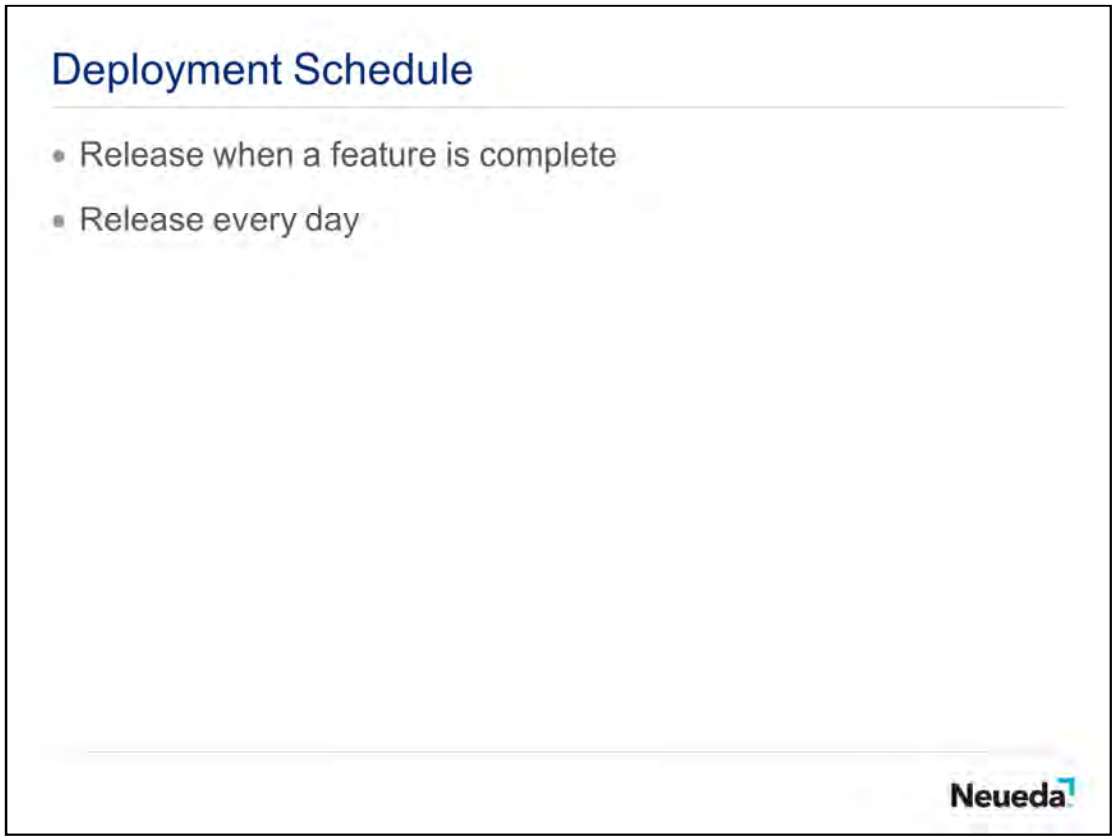
Continuous Deployment

Neueda

A Word of Warning

- Continuous delivery is doable
- Continuous deployment is a hard problem

Neueda



Hanselman: daily VM release

Deployment Strategies

- Zero-downtime deployment (and roll-back)
- Blue-green
  - Two environments
  - Install on one. Switch. Switch back on problems
- Canary release
  - Deploy to subset of servers
- Real-time application state monitor!

Neueda

Problems

- Technical
  - Databases
    - Schema migration
    - Revert!
    - Change management software
  - Configuration
- Human
  - Even more important
  - Automatic deployment = great fear
  - Customers don't want software to constantly change

Neueda

1. Development Now
2. Solution – Continuous Integration
3. Requirements
4. Benefits
5. Continuous Delivery
6. Continuous Deployment
7. Transition
8. Notes from the Field
9. Deployment Issues
10. Deployment Best Practices – Documentation

citi

How to Introduce

- Gain expertise
  - First step accomplished – You are here
- Automate the build
- Introduce tests
- Prove the concept
  - Introduce CI system
  - Run it in parallel to existing infrastructure
  - Give it time
  - Show the win-win

Neueda

1. Development Now
2. Solution – Continuous Integration
3. Requirements
4. Benefits
5. Continuous Delivery
6. Continuous Deployment
7. Transition
8. Notes from the Field
9. Deployment Issues
10. Deployment Best Practices – Documentation

citi

Project Experiences

- Continuous Integration is relatively easy
  - It is all about communication
- Continuous Delivery is harder
  - Some things are hard to test automatically
  - You need people dedicated to writing tests

Neueda

Implementation

- Run everything in VM
- Back-up!

Neueda

Software

- Jenkins
  http://jenkins-ci.org
- CruiseControl.NET
  http://www.cruisecontrolnet.org
- Final Builder
  http://www.finalbuilder.com/finalbuilder.aspx
- Automated Build Studio
  http://smartbear.com/products/

Neueda

Books

- Continuous Delivery
  http://www.amazon.com/dp/0321601912
- Continuous Integration
  http://www.amazon.com/dp/0321336380
- Implementing Lean Software Development
  http://www.amazon.com/dp/0321437381
- Release It!
  http://pragprog.com/book/mnee/release-it

Neueda

References

- http://en.wikipedia.org/wiki/Continuous_integration
- http://en.wikipedia.org/wiki/Continuous_delivery
- http://martinfowler.com/articles/continuousIntegration.html
- http://continuousdelivery.com
- http://www.hassmann-software.de/en/continuous-integration-hudson-subversion-delphi/
- http://edn.embarcadero.com/article/40962
- http://thundaxsoftware.blogspot.com/2011/07/continuous-integration-for-your-delphi.html
- http://nickhodges.com/post/Getting-Hudson-set-up-to-compile-Delphi-Projects.aspx

Neueda

1. Development Now
2. Solution – Continuous Integration
3. Requirements
4. Benefits
5. Continuous Delivery
6. Continuous Deployment
7. Transition
8. Notes from the Field
9. Deployment Issues
10. Deployment Best Practices – Documentation

citi

Discussion

- You are given a WAR file (Java web application) or a ASP.NET MVC application to be deployed
- What do you need to know?

Neueda

Discussion (Cont'd)

- What are the biggest challenges that you have experienced with deploying applications in Allstate?

Neueda

1. Development Now
2. Solution – Continuous Integration
3. Requirements
4. Benefits
5. Continuous Delivery
6. Continuous Deployment
7. Transition
8. Notes from the Field
9. Deployment Issues
10. Deployment Best Practices – Documentation

citi

Documentation

- Effective Deployment and Maintenance requires documentation
- But how and how much?
- Remember the agile manifesto?
  - "We value working software over comprehensive documentation"

Neueda

The RunBook

- One common approach to help with any software project is to have a runbook
- The runbook helps everyone to understand things like
  - How do I deploy and start an application?
  - How do I know it has deployed correctly?
  - How do I solve common problems?
  - What are the parameters of operation?

Neueda

RunBook Purpose

- What the software should do
- Its performance parameters/response times
- The modules that comprise the system
- The database(s) it uses/The services it uses
- The interfaces it offers
- Who the contacts are for the project
- What to do if something goes wrong
- How to start and test the application

Neueda

RunBook Sample Headings

- Overview
  - What does it do
- Deployment
  - How do I deploy it
- Configuration
  - What configuration does it require
- Database
  - What databases does it use and what is their design
- Monitoring
  - How is the application to be monitored
- Changes and maintenance
  - How are updates deployed and where are the logs

Neueda

RunBook Sample Headings (Cont'd)

- Back-up and recovery
  - How are back-ups done and what are the DR contingencies
- Further information
  - Any other pertinent information
- Troubleshooting Guide
  - Step by step troubleshooting instructions
  - Remember, this will be used possibly in a stressful situation – so make it clear
- Contacts
  - Who do we contact if escalation is required

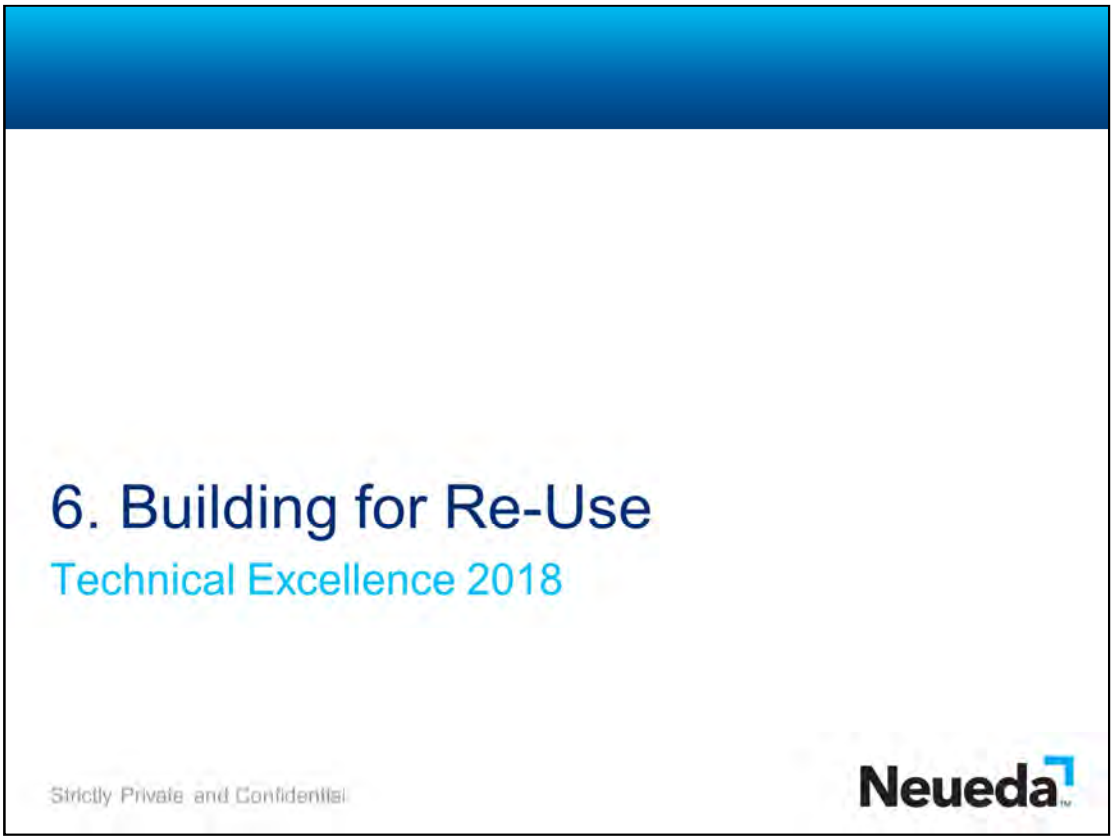
Neueda

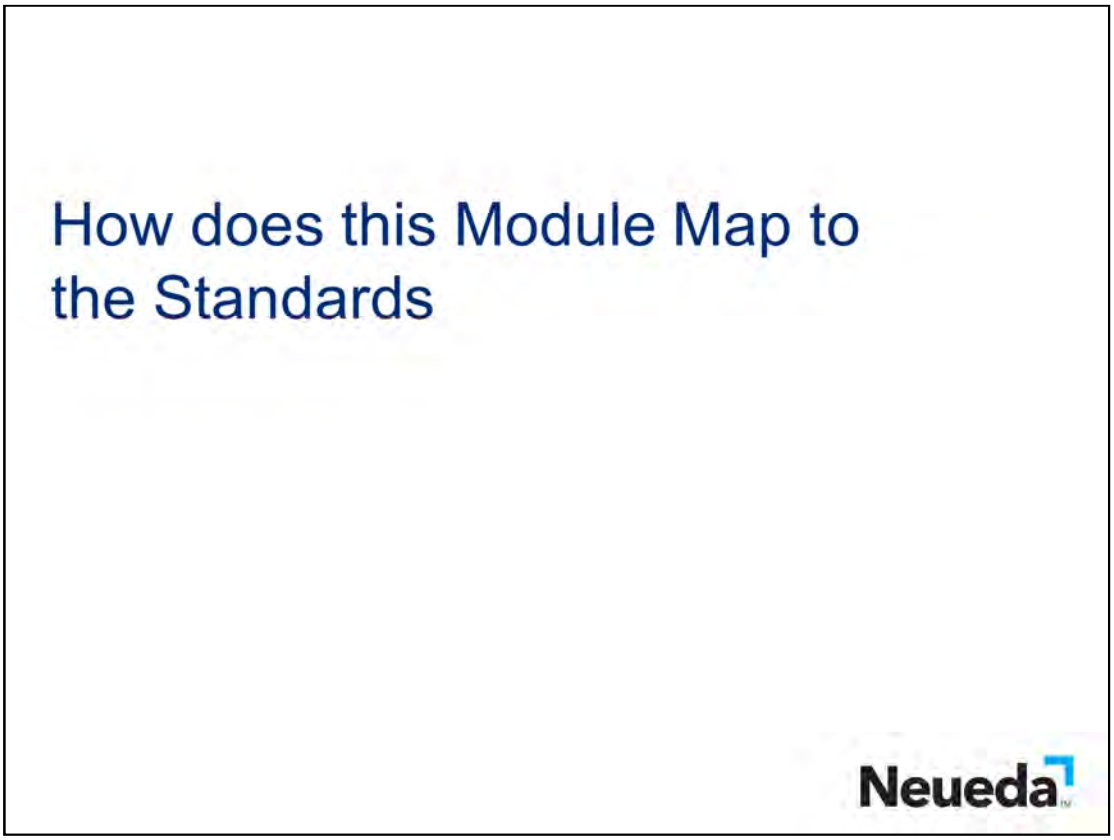
Exercise

- In groups of 3 or 4
  - Review the provided RunBook
  - What is good about this RunBook?
  - What is not so good about this RunBook?
  - What further information would you like to see?

- Present back your findings to the rest of the class

Neueda

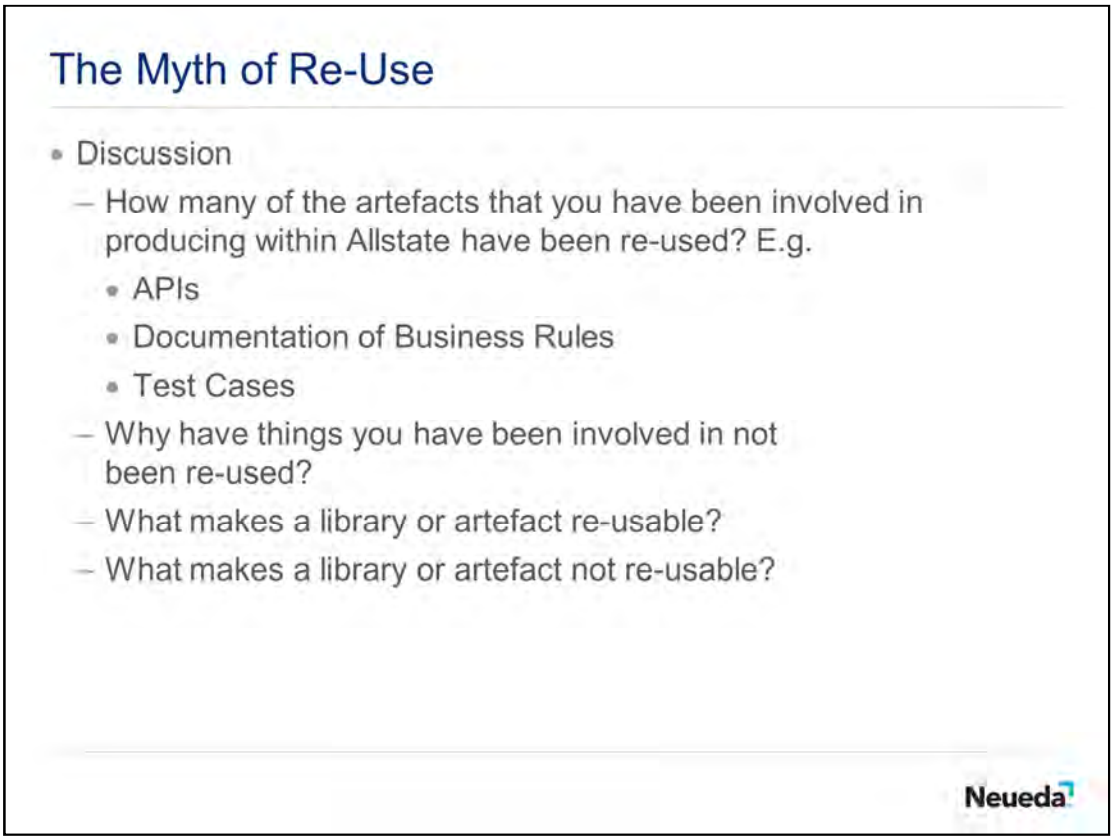

6. Building for Re-Use
Technical Excellence 2018
Strictly Private and Confidential
Neueda


How does this Module Map to the Standards
Neueda

Objectives

- The Myth of Re-Use
- Five Characteristics of Re-Usable Artifacts
  - Simple and Consistent API
  - Well Documented
  - Well Supported
  - Modular
  - Robust Exception Handling and Logging
- Allstate Examples

Neueda



The purpose of this slide is to set the scene and to get people thinking about re-use. Feel free to use these questions to provoke discussion. You can either get small groups within the class to discuss and then feedback or simply ask some or all of these questions to solicit feedback. You might want to consider writing the responses on a flipchart and then referring back to them during the remainder of the module.

Re-Use within Allstate

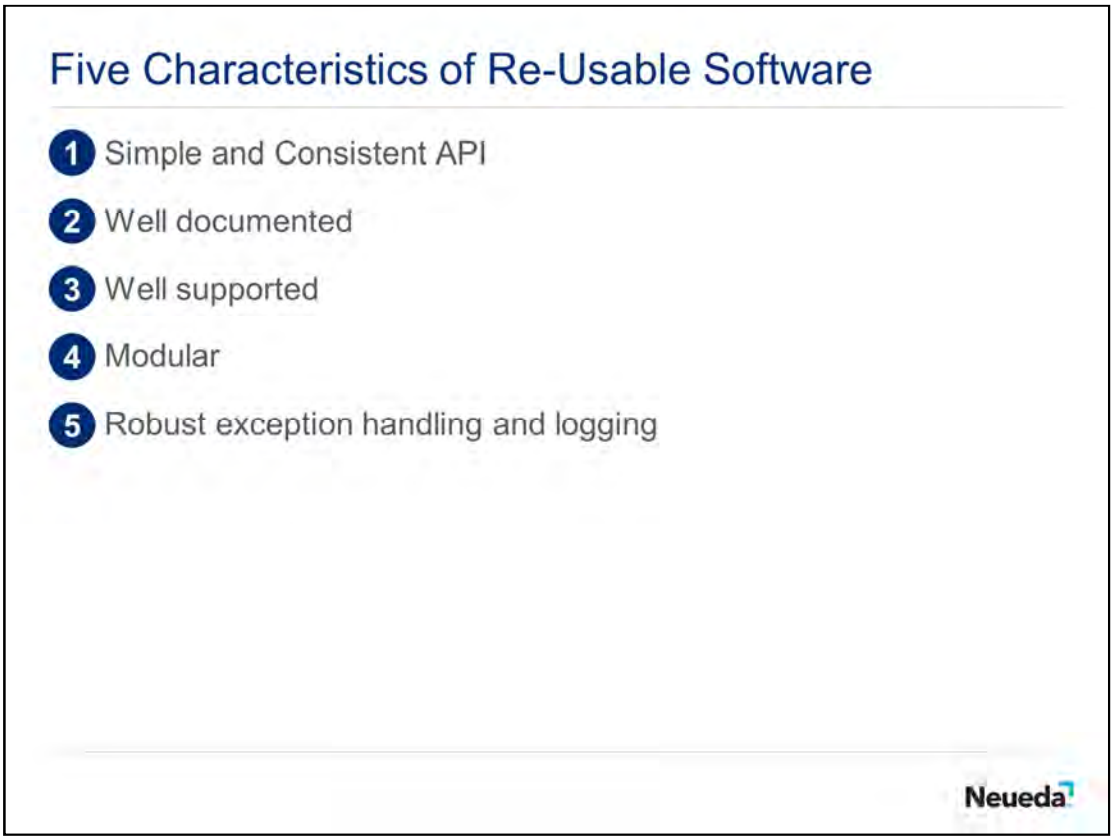
- A proficient developer will
  - "Explore opportunities for reuse; either finding and adopting an existing solution or creating a new asset that others can reuse"

Neueda

The Myth of Re-Use

- Not everything needs to be built for re-use
  - Ad hoc SQL scripts
    - Will not match any other DB structure
  - Many Shell scripts
    - Will be specific to the environment you are working on
  - User interfaces
    - Will be specific to a particular application so cannot be reused
  - Anything else?
- Building for re-use when re-use is not going to happen is unnecessarily time consuming and overly complex
  - Note that this is not the same as building for maintenance and support which is Almost Always important

Neueda


Five Characteristics of Re-Usable Software
1 Simple and Consistent API
2 Well documented
3 Well supported
4 Modular
5 Robust exception handling and logging
Neueda

1 Simple and Consistent API

- One of the biggest barriers to reuse is knowing how to reuse whatever the API is
- If an API is inconsistent in approach, or very complicated, then you will be less inclined to reuse it and build something else that does the same thing
- Inconsistent APIs are problematic to use
- Caveat – don't make it too simple. Just make it as simple as it can be
  - "I pretty soon discovered that "simple" just wasn't going to cut it. So now we have a solution that is unashamedly complex. But hopefully no more complex than it needs to be. And it turns out that this is the best way to simplify life for the actual user"

Gavin King (Creator of Hibernate)

Neueda

The Java 7 Date API as an Example

- The Oracle Web site says this
  - "Some of the date and time classes exhibit quite poor API design. For example, years in java.util.Date start at 1900, months start at 1, and days start at 0– not very intuitive"
  - In fact, Oracle get it wrong (months start at 0, days at 1)
  - Date d = new Date(1 November 2015)
    - What date does this represent?
- Developers in the end, created their own libraries
  - Joda-Time
  - Date4J
- Finally, a new library is now in Java 8, so developers can revert back to using the standard APIs

Neueda

Naming: Method Name Length

Which is Better?

A.

```
Document.remove (String tag);
```

B.

```
Document.removeTag (String tag);
```

In Client Code …

```
myDocument.remove (“Breaking News”);
```

or

```
myDocument.removeTag (“Breaking News”)
```

http://theamiableapi.com/2011/11/01/api-design-best-practice-write-helpful-documentation/

Neueda

2 Well Documented

- Many libraries and frameworks are very clever, but without documentation they are very hard to use and this acts as another barrier to re-use
- Discussion: What documentation is required?
  - API documentation
  - FAQ
  - Test cases
  - Requirements specification

Neueda

Documentation

- Do Developers use API documentation?
  - Yes!
- How do they use it?
  - Just-in-time learning
  - Remembering details
  - Extension of Knowledge
  - Experimenting with code
- What form(s) should your documentation take?

http://theamiableapi.com/2011/11/01/api-design-best-practice-write-helpful-documentation/

Neueda

Complete Documentation

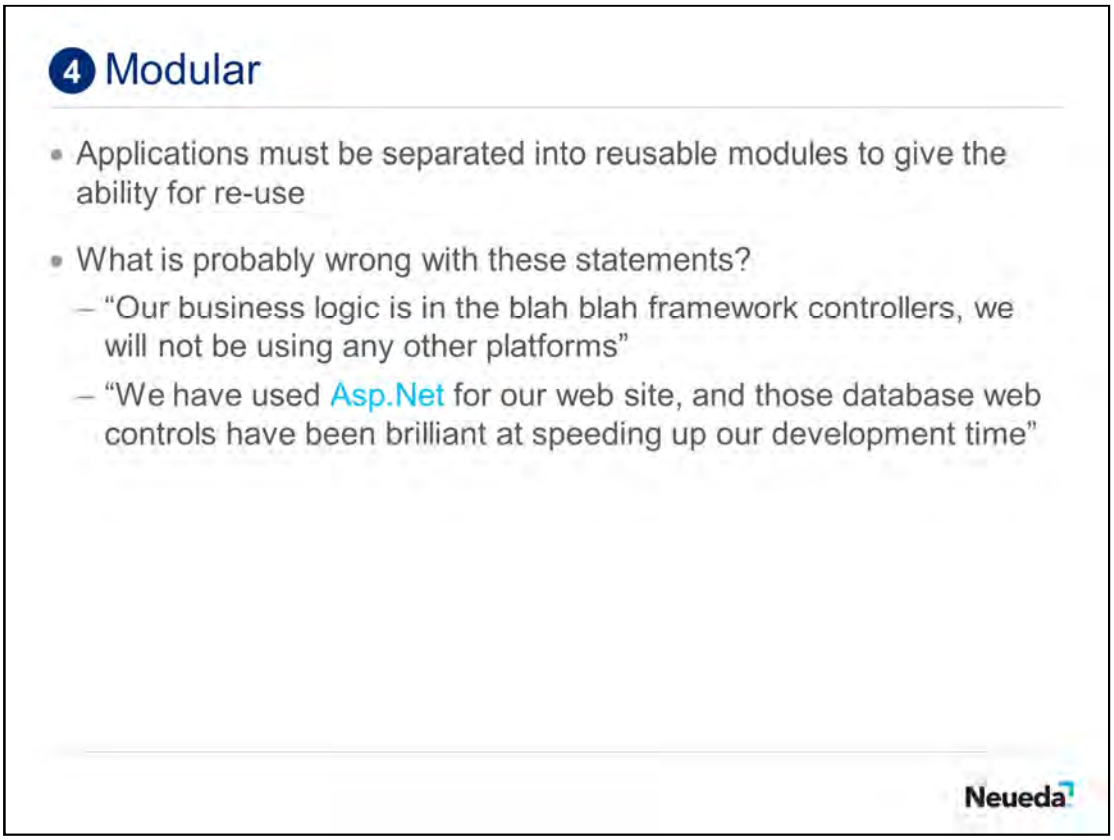
Document	Activities	Content	Organisation
Reference Manual	• Remembering details • Extension of knowledge	• Package overviews • Class overviews • Method descriptions	• Structural top-down • Alphabetical Index
Cookbook	• Experimenting with code • Extension of knowledge • Just-in-time learning	• List of use cases • Description of use cases • Code snippets	• By use-case
Programmer's Guide	• Just-in-time learning • Extension of knowledge	• Introductory overview • Glossary • Concepts • Conventions • Design patterns • Other …	• By subject • Alphabetical Index
Code Example or Tutorial	• Experimenting with code	• Source files • Build or project files • Other resources • Video (optional)	• Development project
FAQ or Knowledge Base	• Extension of knowledge	• Clarifications • Tips • Traps • Known issues	• Questions and answers

Neueda

3 Well Supported

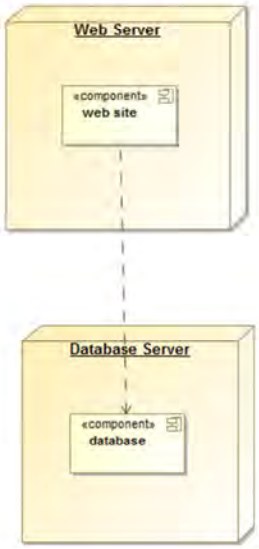
- To use an API, you want to know
  - Who maintains it?
  - Who owns it?
  - Who do I ask if it isn't working?
  - Are there others who use it?
  - Is there a forum where I can ask questions?

Neueda


4 Modular
• Applications must be separated into reusable modules to give the ability for re-use
• What is probably wrong with these statements?
– "Our business logic is in the blah blah framework controllers, we will not be using any other platforms"
– "We have used Asp.Net for our web site, and those database web controls have been brilliant at speeding up our development time"
Neueda

Non Modular Architecture

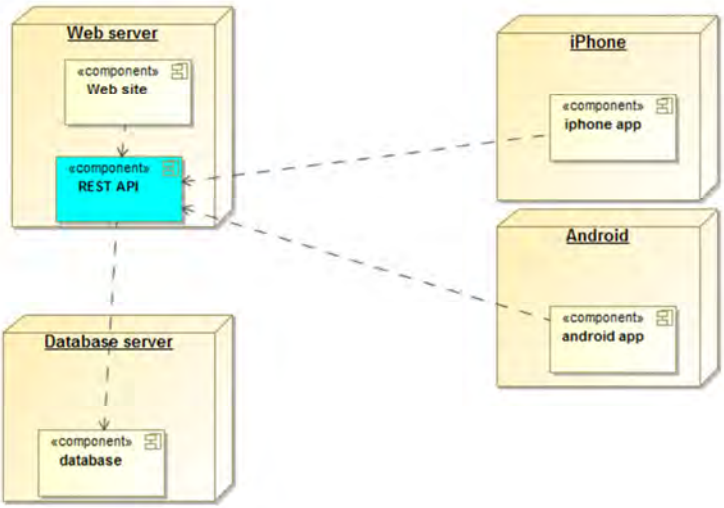
- You build a web site using one of the common Web platforms such as Asp.Net or JSF
  - For example an online auction site
- How will you build the mobile app?
  - Where is the API for the business logic?
  - How can you reuse it for mobile?



Neueda

Modular Example

- This architecture (using a REST API) provides re-use for the business logic for mobile apps

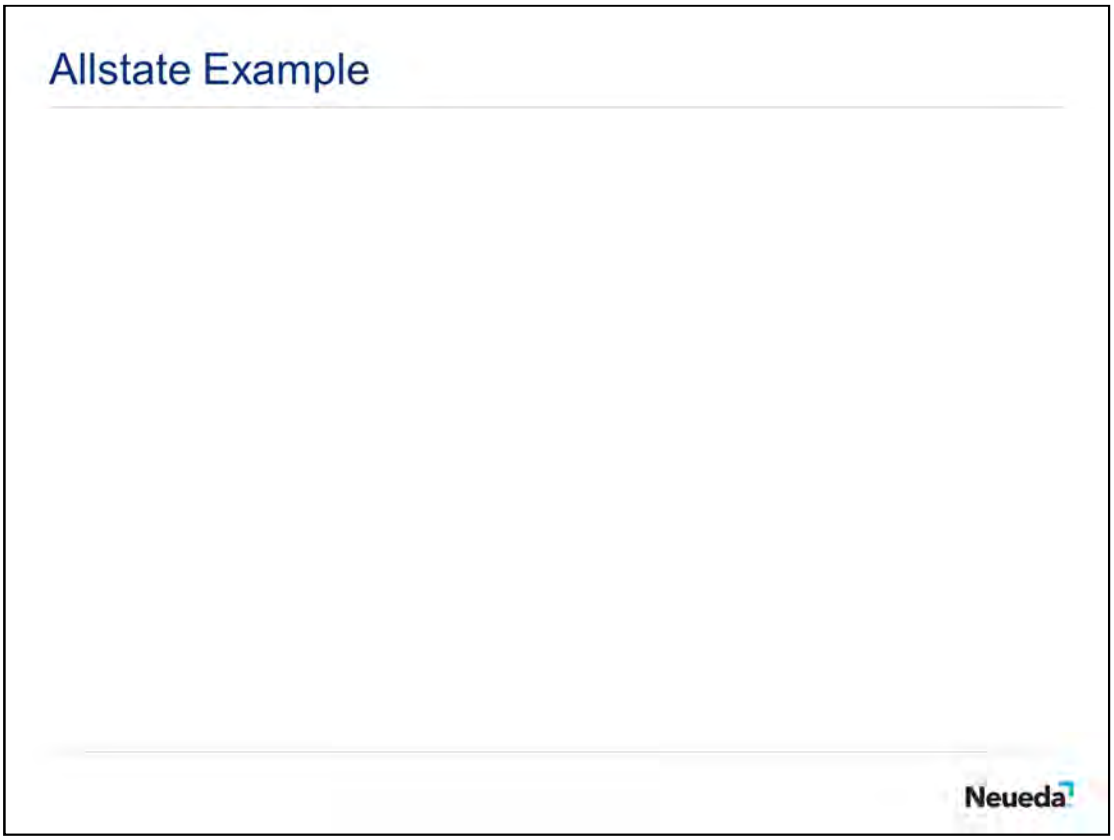


Neueda

5 Robust Exception Handling and Logging

- Any API that is to be reused must have a consistent exception handling model applied
- Any API that is to be reused must have appropriate logging capabilities
- A good example of this would be Hibernate for the Java and .NET platform
  - Loggers are configurable for all aspects of Hibernate functionality
  - There is an clear exception model with many specific exception classes representing the different kinds of problems you can encounter

Neueda


Allstate Example
Neueda

Some Practical Reasons Why Reuse May Succeed

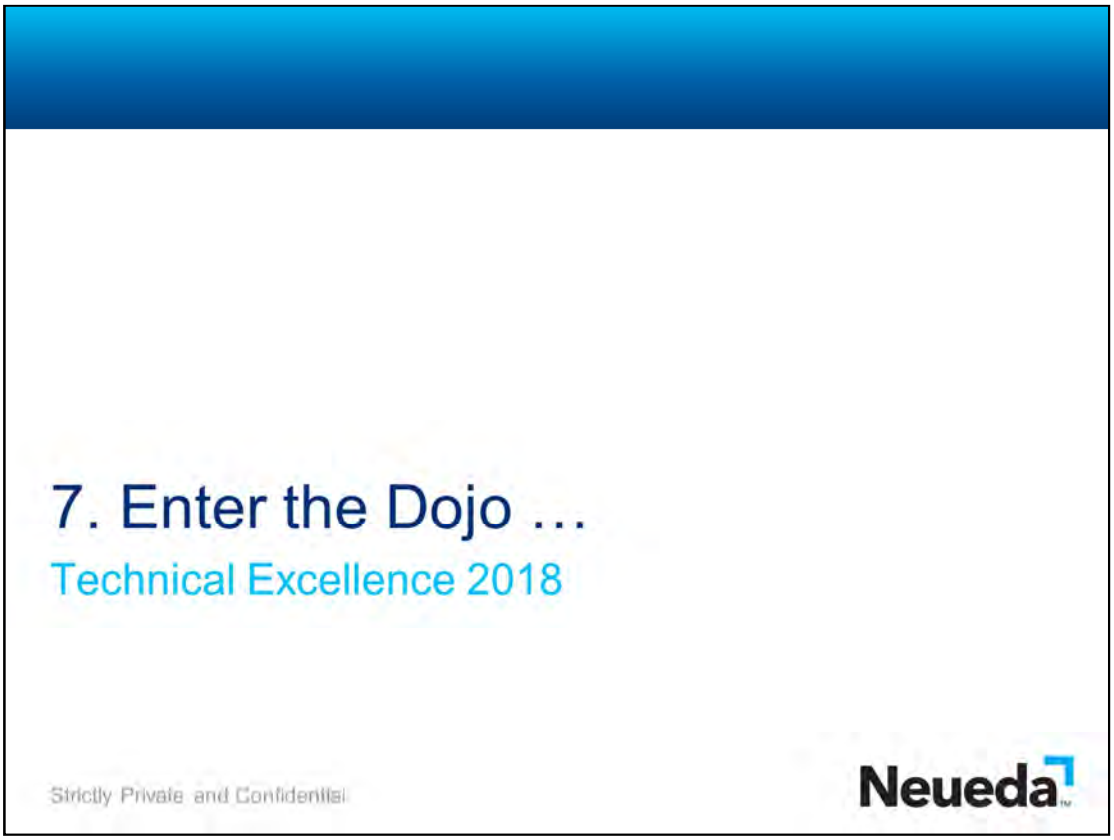
- Answers the demand not addressed by other applications
- Cross platform REST APIs and the team implemented clients in .Net, Java, JS, Objective C
  - Simplicity of usage/API (Enable, LogException). Will probably take 15 min
  - Simple on-boarding
  - Good wiki pages and documentation
- SSL support to support applications deployed on clients' computers
- Does not crash. If host crashes – recovers exceptions when goes back

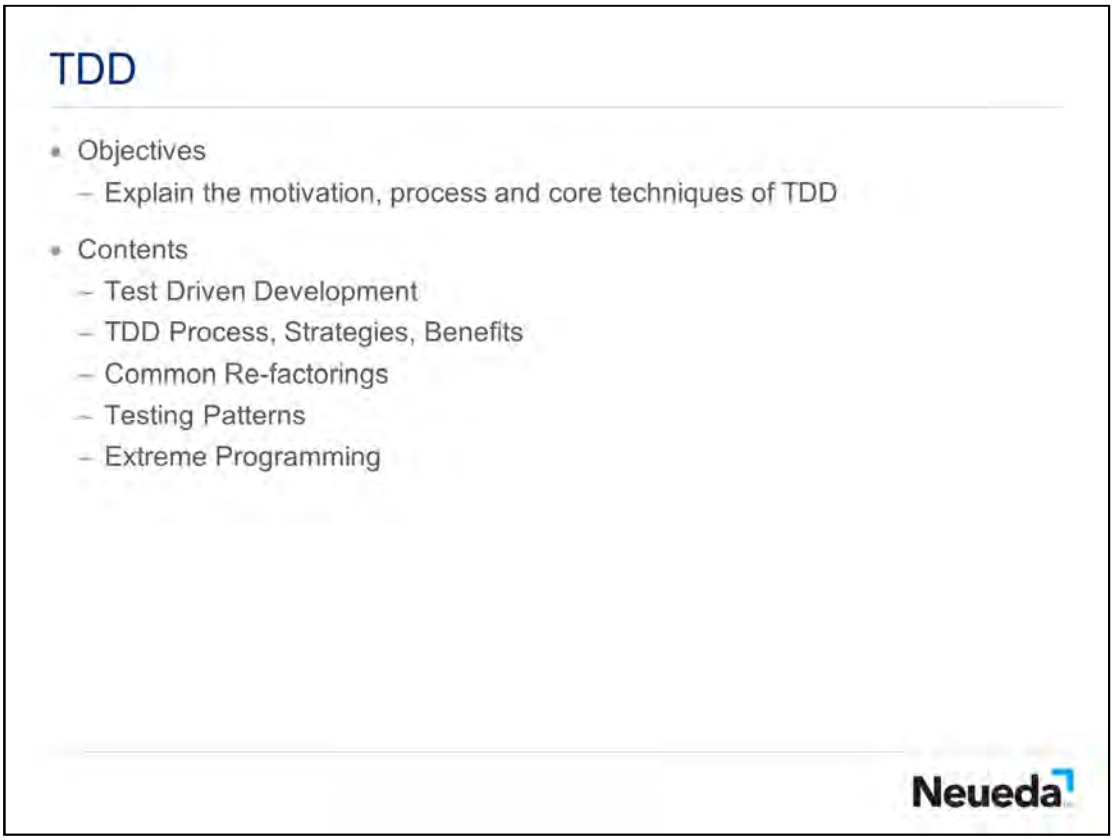
Neueda

Summary

- The Myth of Re-Use
- Five Characteristics of Re-Usable Artefacts
- Simple and Consistent API
- Well Documented
- Well Supported
- Modular
- Robust Exception Handling and Logging
- Allstate Examples

Neueda

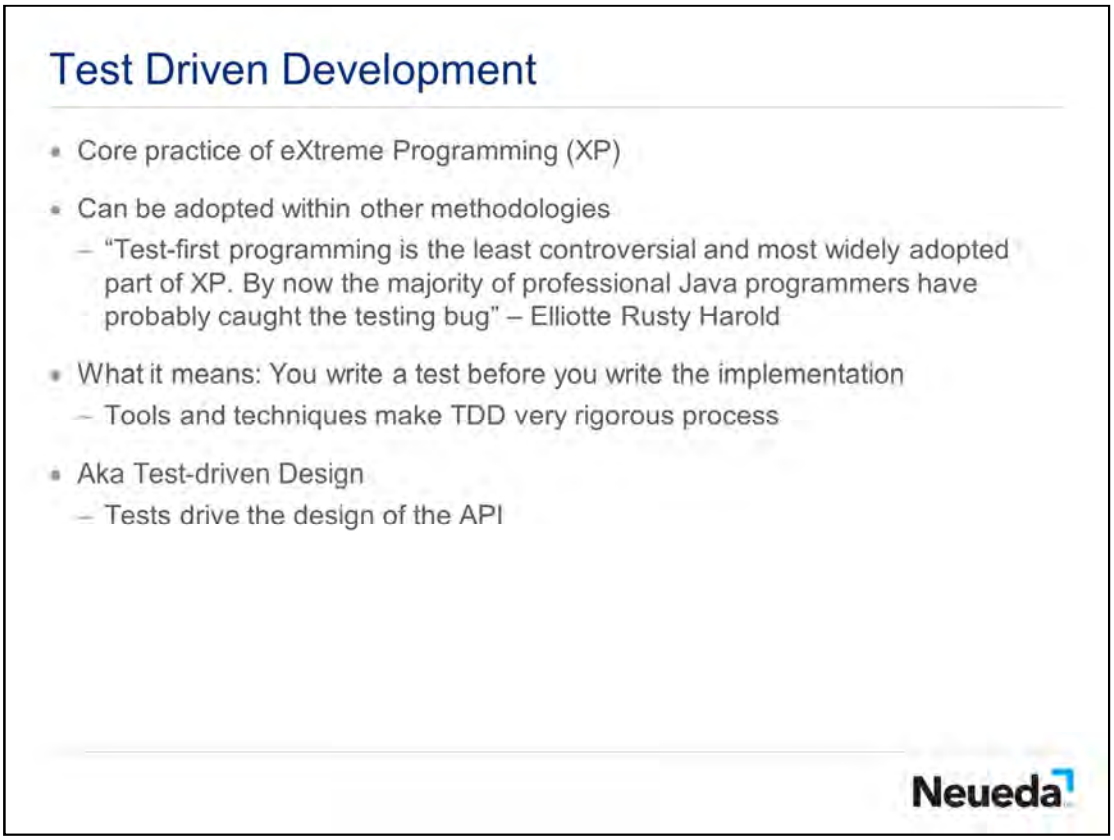

7. Enter the Dojo …
Technical Excellence 2018
Strictly Private and Confidential
Neueda


TDD
Objectives
Explain the motivation, process and core techniques of TDD
Contents
Test Driven Development
TDD Process, Strategies, Benefits
Common Re-factorings
Testing Patterns
Extreme Programming
Neueda

Traditional Development

- Code – Test – Refactor
- Tests are often an afterthought!
  - "If there's time"/leave to testers
- Problem: High level of defects
  - Lengthy testing phase after a release is frozen
  - Cost of fixing a bug discovered at that stage is far higher than if the bug was caught when introduced into code
- Problem: Poor maintainability
  - Legacy spaghetti code that "works"
  - Can't be touched – fear of breaking

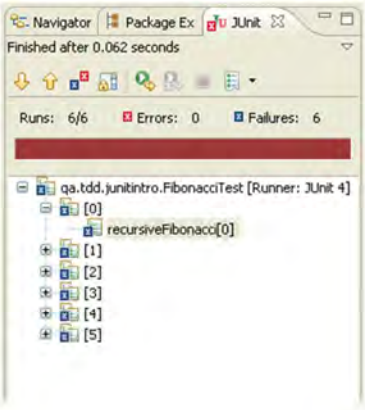
Neueda

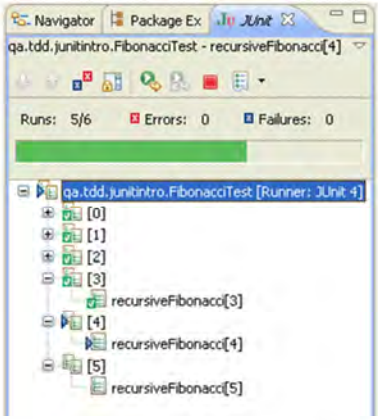


The graphic on the right illustrates that the famous green bar is indeed a progress bar. Here we see JUnit running a sequence of six tests; four have completed successfully, the fifth is indicated as currently running, and the sixth is shown as yet to run.

Test Driven Development (Cont'd)

- Test – Code – Refactor
  - 1. Write new code only if you first have a failing automated test
  - 2. Eliminate duplication
- Red – Green – Refactor





Neueda

TDD Process

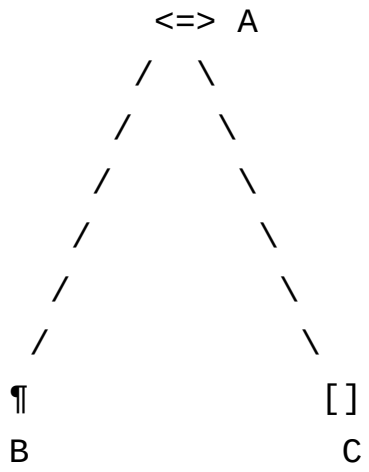
- First write the test
  - Designing the API for the code to be implemented
  - Using an API (in tests) is the best way to evaluate its design
- Write just enough code for the test to pass
  - Minimises code bloat
  - Keeps you focussed on satisfying the requirement of the test
- Refactor: Change code without changing functionality
  - "A disciplined technique for restructuring code, altering its internal structure without changing its external behaviour" – Fowler
- Develop in small iterations
  - Test a little, code a little

Neueda

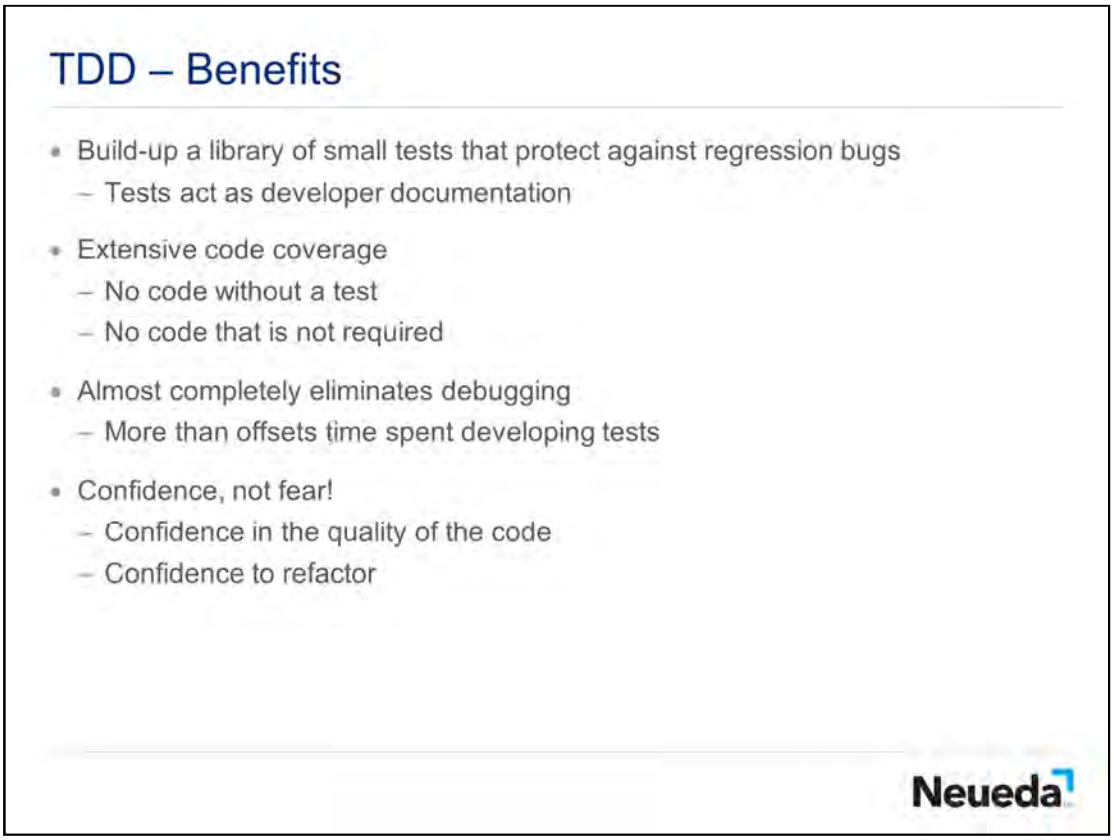
TDD Strategies

- Fake it
  - Return a constant
  - Gradually replace constants with variables
- Obvious implementation
  - If a quick, clean solution is obvious, type it in
- Triangulation
  - Locating a transmitter by taking bearings from two or more receiving stations
  - Only generalise 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

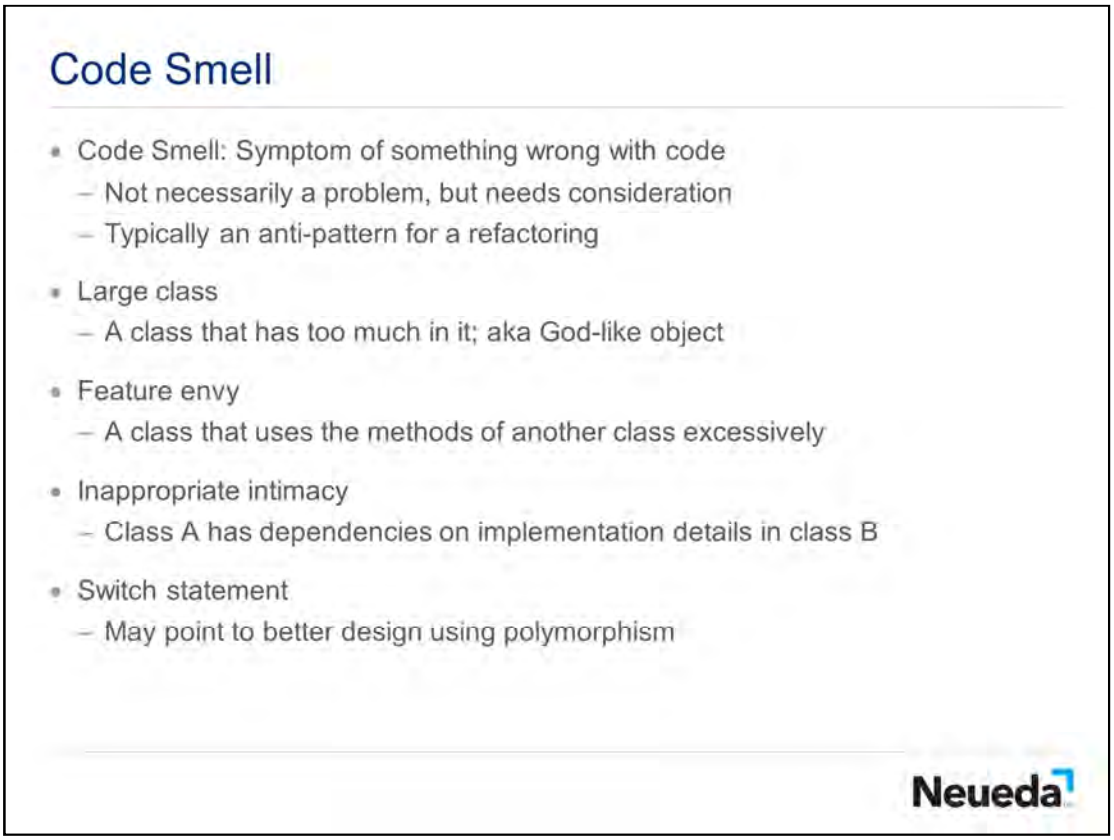
Neueda



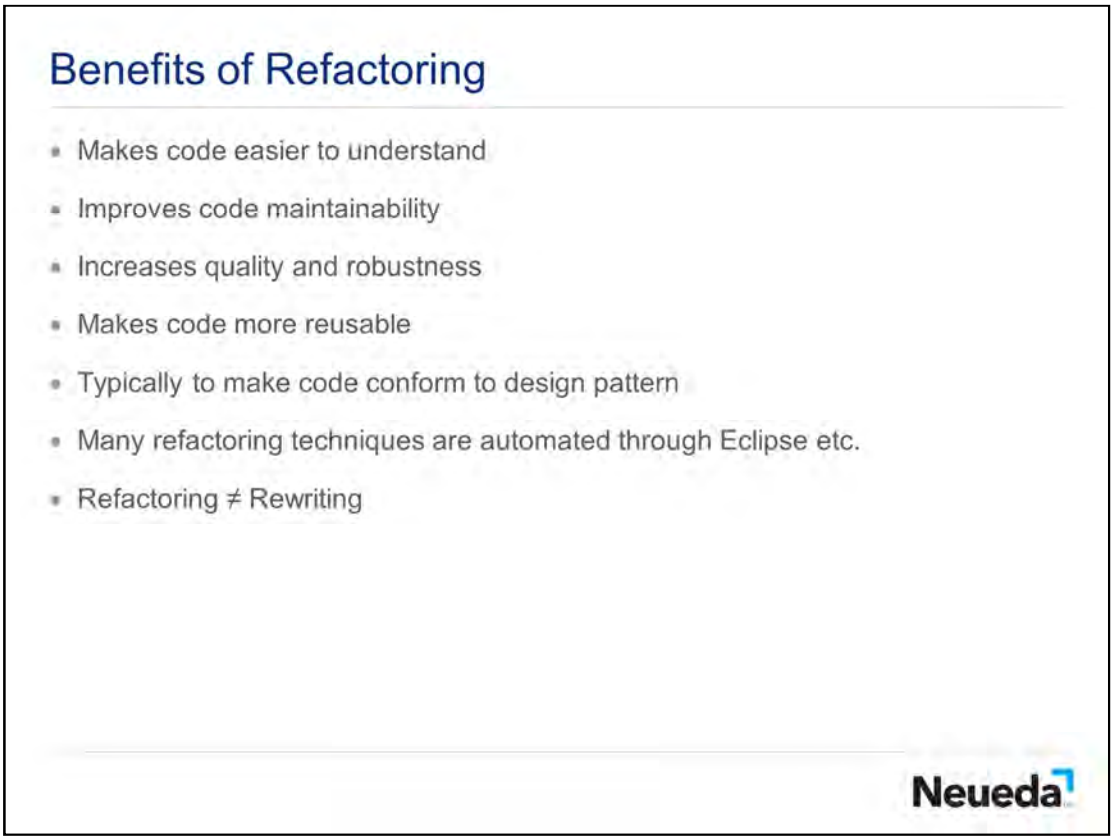
Triangulation: someone on the boat at A can determine their position on a chart by taking compass bearings to the lighthouse at B and the tower at C. Or conversely, observers at B and C can work out the location of the boat by taking bearings to it from their known points on the shore and sharing their 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.



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



"Code smell" has become a common phrase in the XP/TDD community, promoted in Martin Fowler's *Refactoring: Improving the Design of Existing Code*. The slide lists four representative examples. *Switch statement* illustrates that these are only indicative; a particular use of a switch statement may be a perfectly appropriate flow control construct.



According to Koskela:

"**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." (*Test Driven*, p. 106)

Martin Fowler's site `http://refactoring.com` is a useful resource. For example, see the catalogue of refactorings at the following URL:

`http://www.refactoring.com/catalog/index.html`

Another on-line catalogue of refactorings is available at the following URL:

`http://industriallogic.com/xp/refactoring`

Summary

- TDD: Big change to developers' mind-set and practices
- Test – Code – Refactor
  - Write a test
  - Make it run
  - Make it right
- Develop in small increments
- Follow unit testing best practices
  - Make tests clear, maintainable, fine-grained, independent
  - Ensure good coverage of negative (abnormal flow, errors) as well as positive (normal flow)

Neueda

References

- Test Driven Development – By Example
  - Kent Beck (Addison Wesley) ISBN 0321146530
- xUnit Test Patterns: Refactoring Test Code
  - Gerard Meszaros (Addison Wesley) ISBN 0131495054
  - http://xunitpatterns.com
- Refactoring: Improving the Design of Existing Code
  - Martin Fowler (Addison Wesley) ISBN 0201485672
- http://www.testdriven.com
- http://www.infoq.com

Neueda

Becoming an Exemplary Developing Engineer

- Is it by attending this course?
  - If only it were that easy!
- Like anything, practice is what makes a master

Neueda

The Cyber Dojo

- Jon Jagger created cyber-dojo.org to be a place to
  - Promote deliberate practice for software developers
- Cyber-dojo is where programmers practice programming
- Cyber-dojo is not an individual development environment
- Cyber-dojo is a shared learning environment
- You practice by going slower and focusing on improving rather than finishing

Neueda

Enter the Dojo

- You will now enter the Dojo and begin your first practice

Neueda