Q & A

TEST

PBA SOFTWAREUDVIKLING/
BSC SOFTWARE DEVELOPMENT

Christian Nielsen cnls@cphbusiness.dk

Tine Marbjerg tm@cphbusiness.dk

SPRING 2019

OVERALL EXAM TOPICS

- Testing in Software Life Cycle (fundamentals of testing)
- Testability
- Test Case Design (including static techniques)
- Unit Testing
- Integration Testing
- System Testing
- Non-functional Testing (performance)
- Continuous Integration / Continuous Deployment

GENERAL ABOUT EXAM

- Be prepared to show code and tests, including demo of exam project
- Platforms and tools: Java, JUnit etc., presented in class *or similar*.
- Exam 25 minutes (individual oral examination without preparation)
  - Two short presentations (5 minutes each), followed by a discussion.
  - *Presentation 1 – Topic*
    - The student draws a topic and makes a presentation based on the learning objectives and the assignments of the semester.
  - *Presentation 2 – Semester project*
    - The student presents a self-chosen topic based on the semester project.

TOPIC PRESENTATION EXAMPLES

- Testability
- Integration Testing
- Be able to handle production and test databases separately
- Non-functional Testing
- xUnit Framework
- Static Test Techniques
- Agile Test Quadrants
- Automated System Testing
- Test Automation
- Stubbing and Mocking
- Black Box Design Techniques

EXAM TOPICS AND SP ASSIGNMENTS

Topic	Assignment ID	Assignment
Test Case Design	1	Test cases
Unit Testing	2	Unit testing
Testability	3	Testability
Test Case Design	4	Specification based testing techniques
Testability	5	Dependencies
Testability / Unit testing	6	TDD
Integration testing	7	Integration testing
Functional testing	8	Functional testing
Non-functional testing	9	Non-functional testing
CI/CD	10	CI/CD

Note: System Testing & Fundamentals of software Life Cycle are covered by the exam project

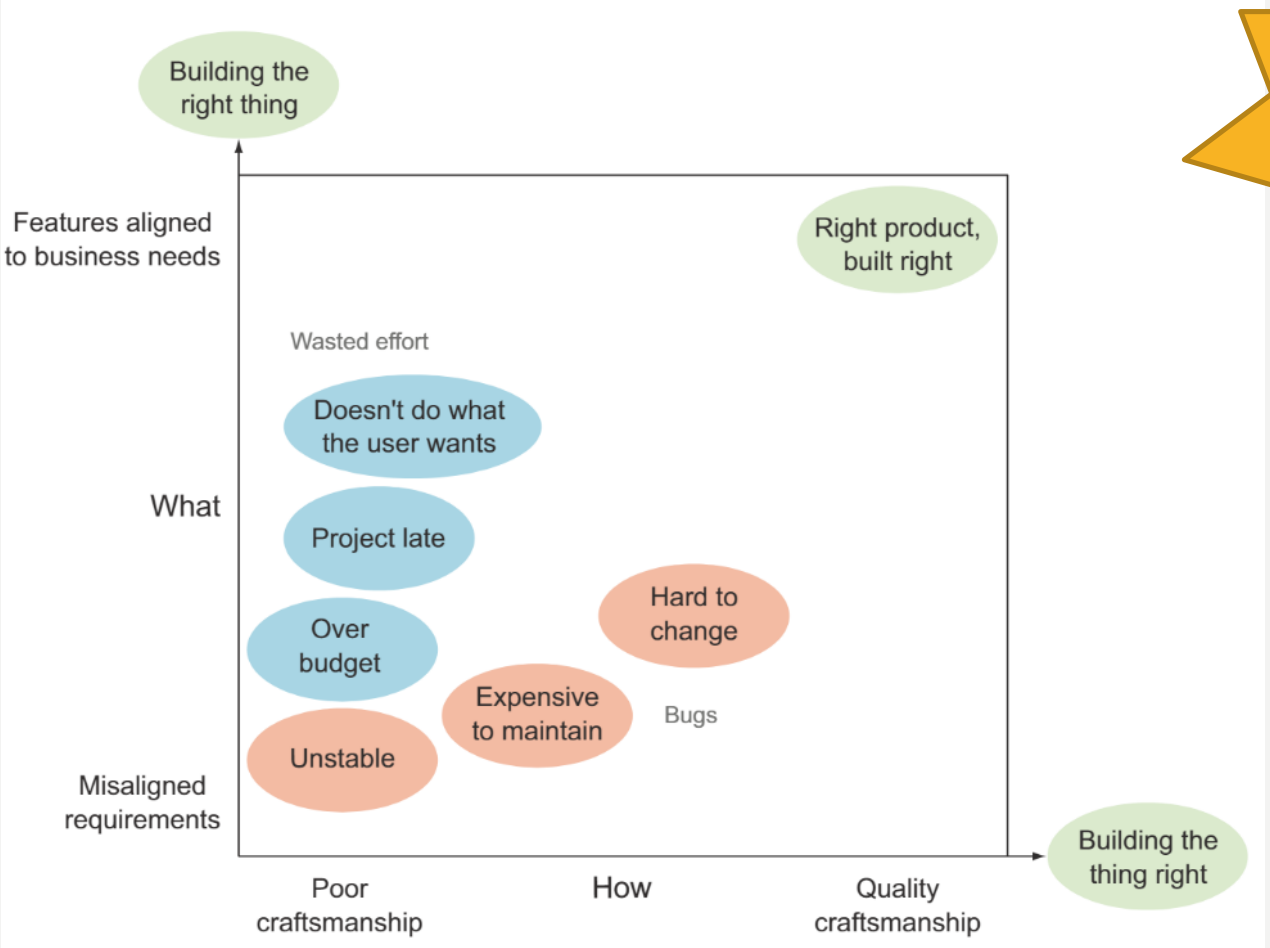
NON - EXHAUSTIVE LIST OF TOPICS

WALK THROUGH MAJOR ASPECTS OF DEVELOPER TESTING

20-05-2019

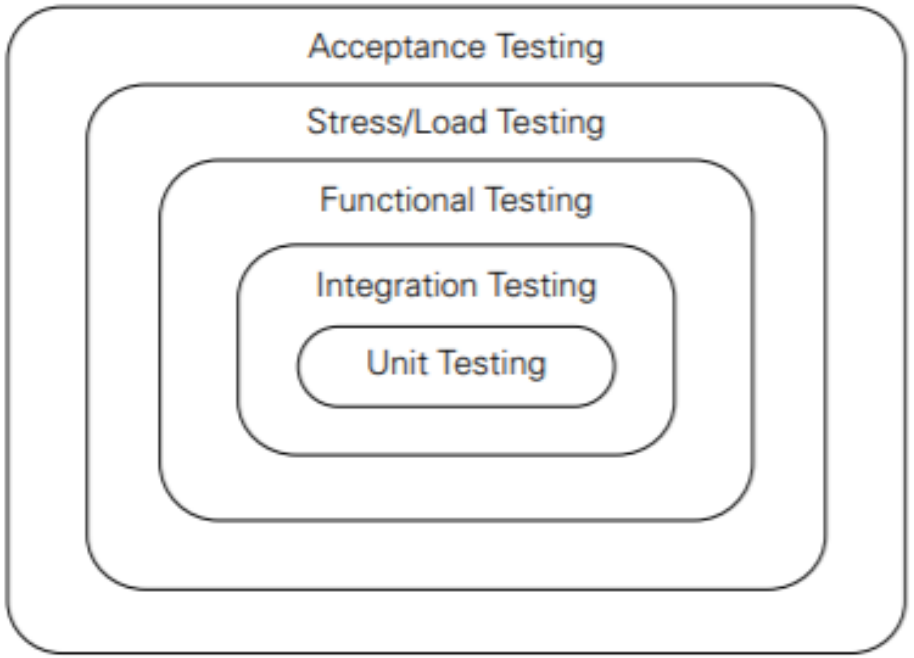
Q & A

FUNDAMENTALS: BUILD THE RIGHT THING AND BUILD IT RIGHT



Verification & Validation

THE FIVE TYPES OF TESTS



The outermost tests have broadest scope

NB!

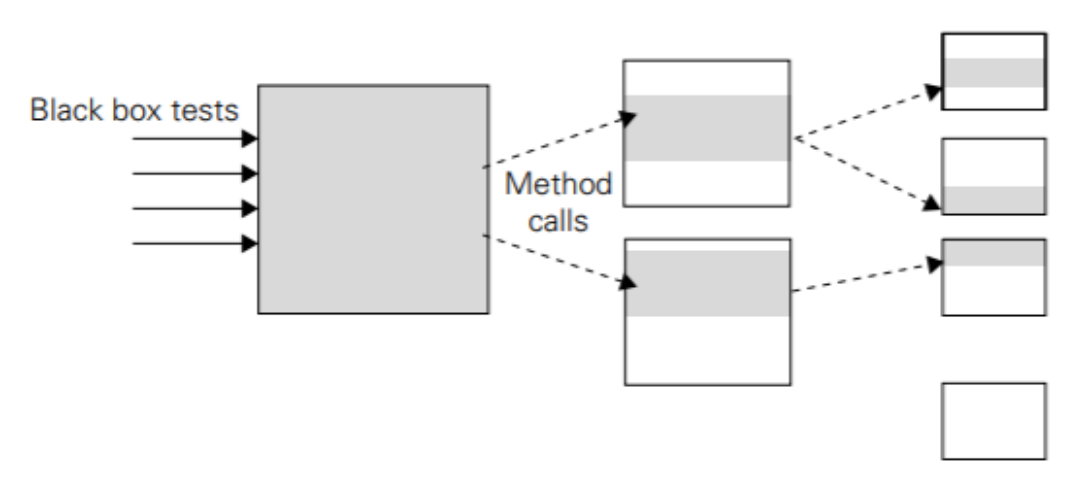
There are others non-functional test types, but performance has been our focus

Functional testing ~system testing

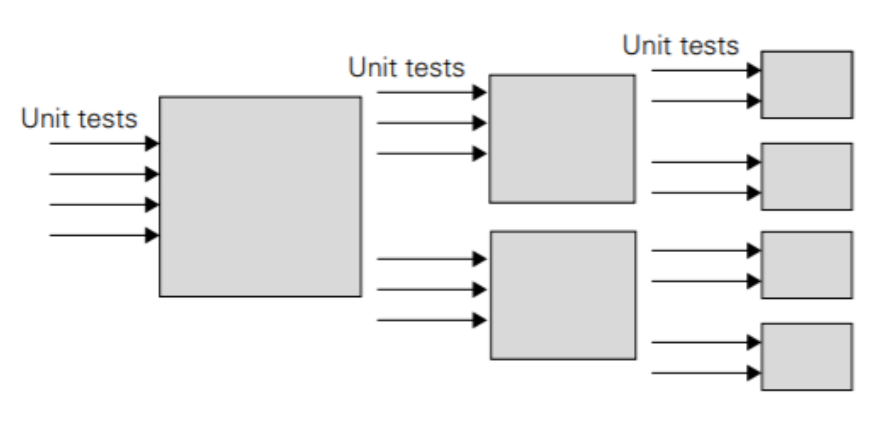
THE NEED FOR UNIT TESTS

1. They allow greater test coverage than functional tests
2. Increase team productivity
3. Detect regressions and limit the need for debugging
4. Confidence to refactor and make changes in general
5. Improve implementation
6. Document expected behavior
7. Enable code coverage and other metrics

1. GREATER TEST COVERAGE THAN FUNCTIONAL TESTS

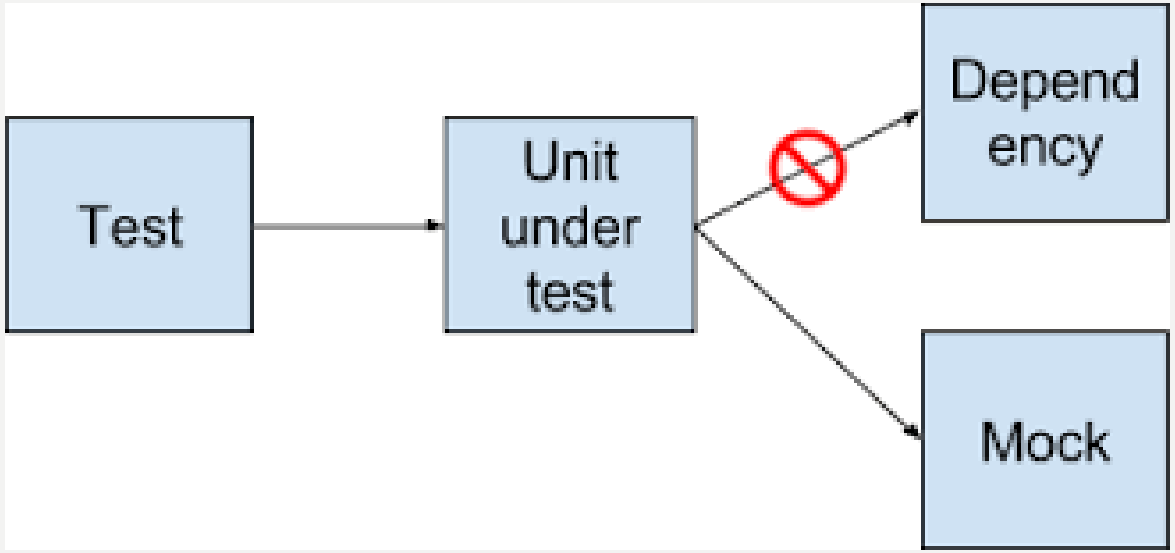


Unit tests can achieve higher coverage (you can control input to each method and behavior of secondary objects):



2. INCREASE TEAM PRODUCTIVITY

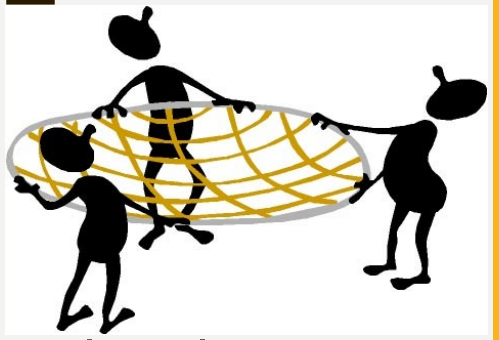
- You don't have to wait for all other team members to finish their work



3. DETECT REGRESSIONS AND LIMIT THE NEED FOR DEBUGGING

- A passing unit-test **confirms** your code works
- Increase confidence in changing the code - Instant feedback
- Well-written unit tests reduce the need to debug to find root cause.
  A functional test will tell you that a bug exists *somewhere* in the implementation

4. REFACTOR WITH CONFIDENCE



- Without unit tests, it is difficult to justify refactoring, because there is always a relatively high chance that you may break something in the process

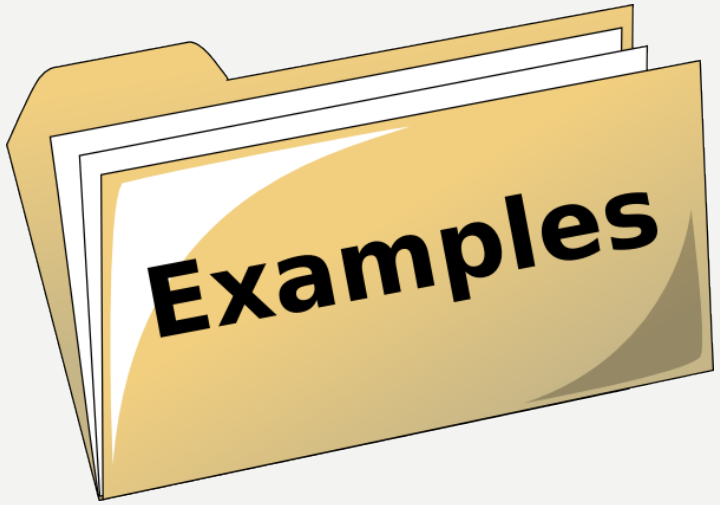
- Unit tests provide a safety net that gives you the courage to refactor.

5. IMPROVE IMPLEMENTATION

- Unit tests are a first-rate client of the code they test
- Force the API under test to be flexible and testable in isolation
- TDD is good way to build testability and quality from day one

6. DOCUMENT EXPECTED BEHAVIOR

- Examples better than 300 page documentation describing the API



7. CODE COVERAGE AND OTHER METRICS

File Coverage Summary

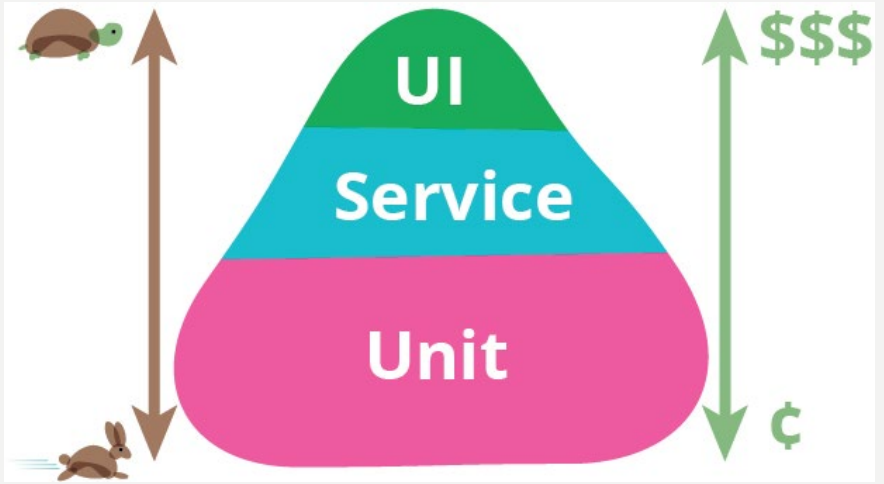
Name	Classes	Methods	Lines	Conditionals
Cell.java	100% 1/1	67% 2/3	71% 5/7	50% 3/6

Source

com/ciwithhudson/gameoflife/domain/Cell.java

```
1     package com.ciwithhudson.gameoflife.domain;
2
3     /**
4      * A single cell, which can be alive or dead.
5      */
6   819 abstract public class Cell {
7
8         public abstract Boolean isAlive();
9
10        public Boolean isDead() {
11    0       return !isAlive();
12        }
13
14        public abstract Cell nextGeneration(int neighbourCount);
15
16        public static Cell fromChar(char cellValue) {
17    92      if (cellValue == LivingCell.SYMBOL) {
18    35          return new LivingCell();
19    57      } else if (cellValue == DeadCell.SYMBOL) {
20    57          return new DeadCell();
21            }
22    0       throw new IllegalArgumentException("Illegal cell value character: " + cellValue);
23        }
24
25    }
```

INTEGRATION TESTING



- First, make individual unit tests work well
- Then … what happens when units of work are combined in workflow?
- We need to test the interaction between components = integration testing

DEFINITION OF INTEGRATION TESTING

Another way of putting it:

It is *not* a unit test if …

- It talks to the database.
- It communicates across the network.
- It touches the file system.
- You have to do special things to your environment to run it.

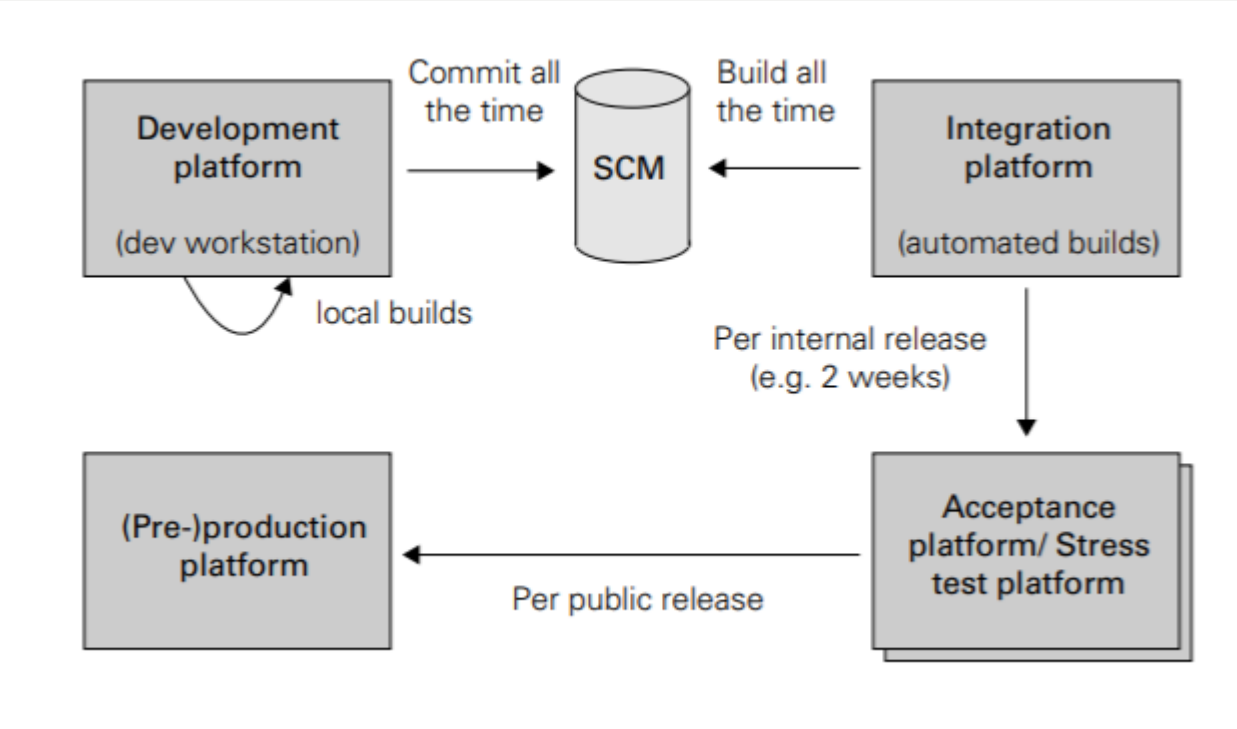
Source: Test-Driven Development (Koleska)

INTEGRATION TESTING APPROACHES

- Big bang
  - Everything is finished before integration testing (no need for simulation)
  - Time-consuming and difficult to find root cause
- Top-down
  - GUI testing early – feedback from user
  - Stubs needed
- Bottom-up
  - Drivers needed
- Incremental
  - Functions are testing incrementally
  - Incremental feedback from user

TESTING IN THE DEVELOPMENT CYCLE PLATFORMS

Testing occurs at different platforms during development cycle:



Development platform—where the coding happens

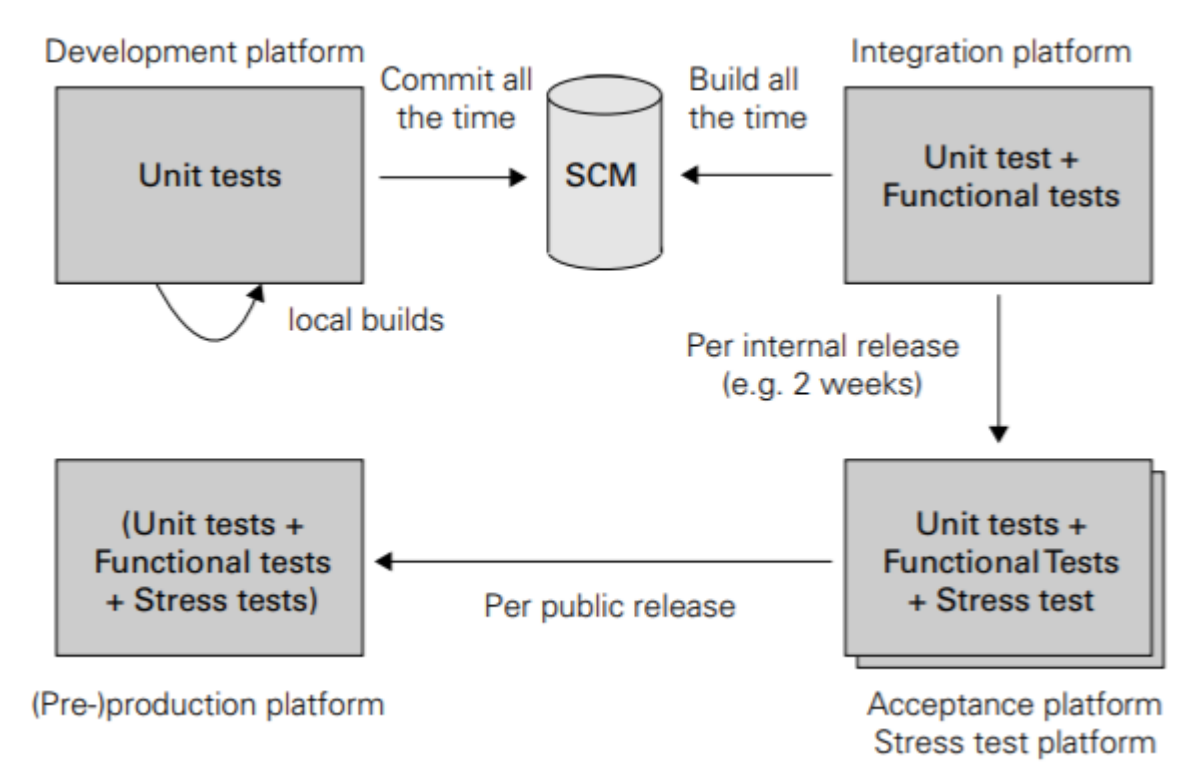
Integration platform builds the application from its different pieces and ensure that they all fit together

Acceptance platform is where the project's customers accept the system

Stress platform exercises the system under load and verifies scalability

TESTING IN THE DEVELOPMENT CYCLE TEST TYPES

Test types on the different platforms:



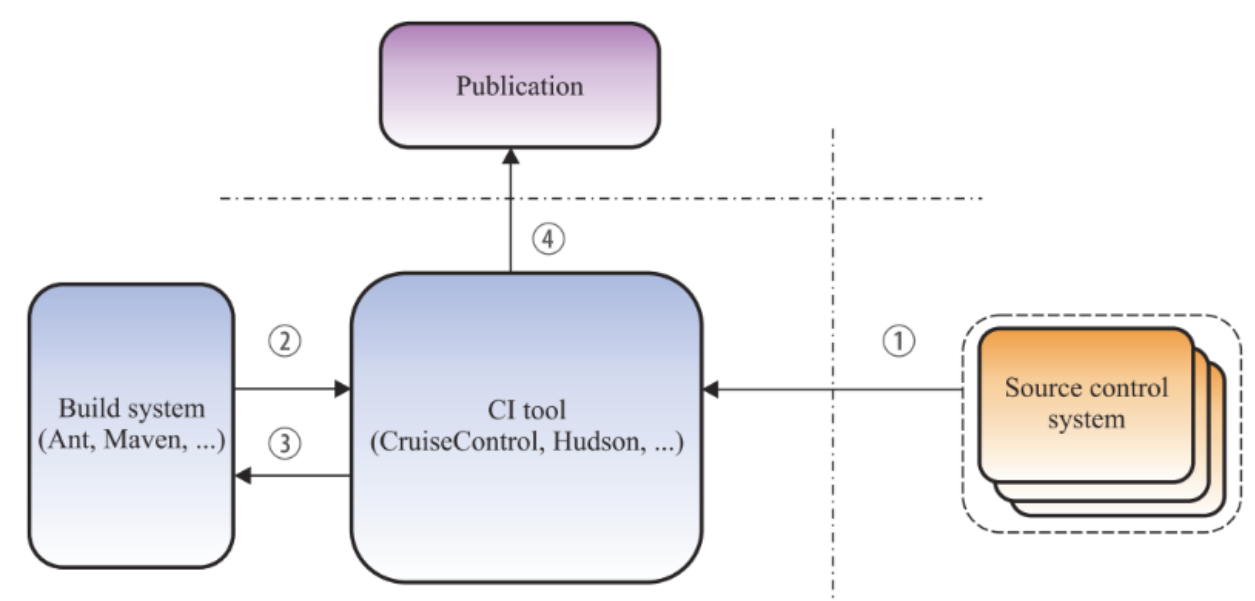
Development platform: unit tests (tests in isolation) – quick from IDE

Integration platform: all types of unit and functional tests automatically (time is less important). Maybe only subset (maybe not all ext. systems available)

Acceptance & **Stress** platform(s): same tests – platform extremely similar to production.

CONTINUOUS INTEGRATION TESTING

1. Check out project from source control system
2. Build each of the modules and execute all unit tests to verify that each module works in isolation
3. Execute integration tests to verify that modules integrate as expected
4. Publish the results from the tests executed



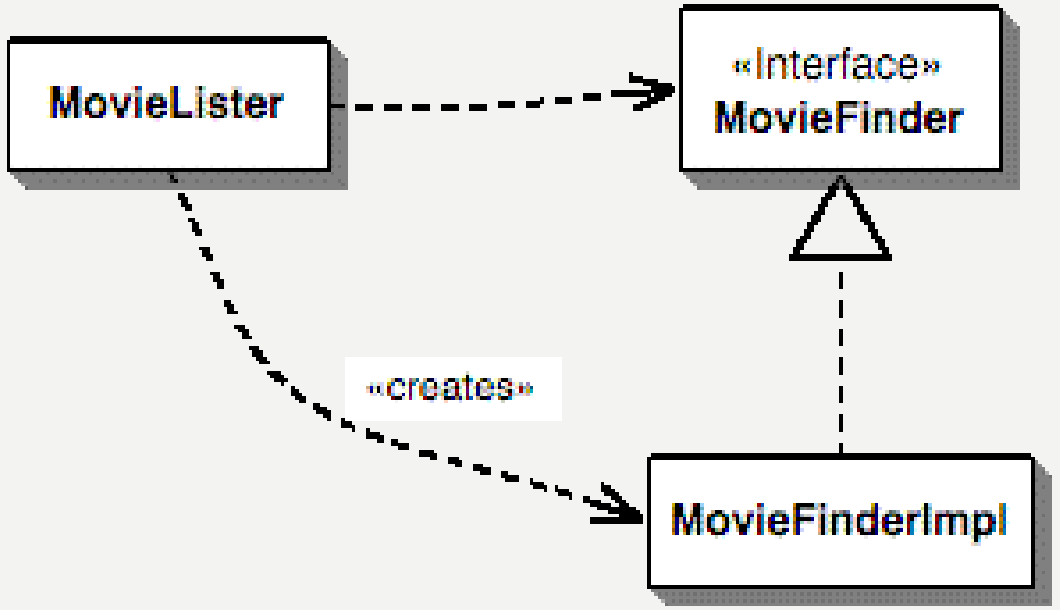
WRITING TESTABLE CODE

- The fundamental value of testable design is code that is easy to test

- Best practises:
  - ❑ Public APIs are contracts
  - ❑ Reduce dependencies
  - ❑ Create simple constructors
  - ❑ Follow the Principle of Least Knowledge
  - ❑ Avoid hidden dependencies and global state
  - ❑ Singletons pros and cons
  - ❑ Favour generic methods

FAVOUR GENERIC METHODS - EXAMPLE

- Polymorphism offers more flexible code than static methods
  - `MovieLister` can use `MovieFinderImpl` or stub/mock implementation



FAVOUR POLYMORPHISM OVER CONDITIONALS

- Avoid long **`switch`** and **`if`** statements (increases complexity)
- Remember, we have tools to measure CC.

```
public class DocumentPrinter {
  [...]
    public void printDocument() {
     switch (document.getDocumentType()) {
        case  Documents.WORD_DOCUMENT:
          printWORDDocument();
          break;
        case Documents.PDF_DOCUMENT:
          printPDFDocument();
          break;
        case Documents.TEXT_DOCUMENT:
          printTextDocument();
          break;
        default:
          printBinaryDocument();
          break;
     }
    }
    [...]
}
```

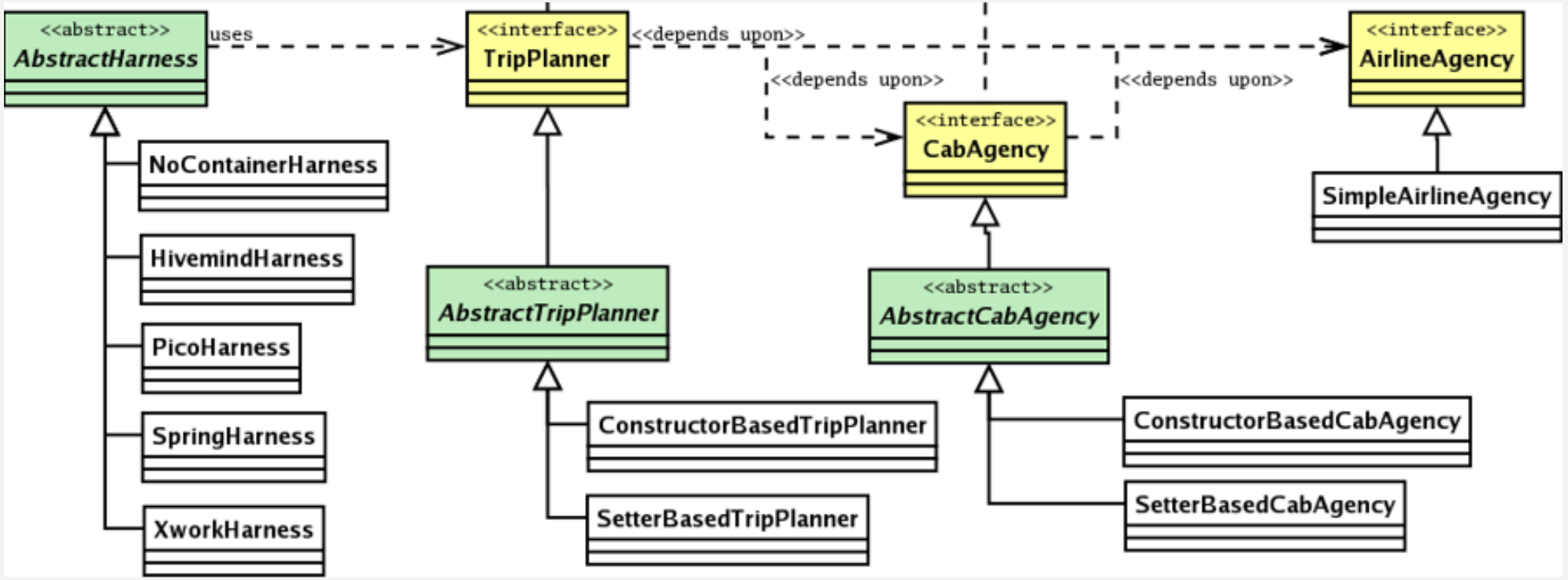
Solution: When you see a long conditional statement, think of polymorphism;
That means breaking down into several smaller classes for each document type with each their implementation of **`printDocument()`** method

DEPENDENCY INJECTION

- How to code
- How to configure dependencies (inversion of control)
- Stubs vs. mocks
- State based testing vs. behavior based testing

A ”NO CONTAINER” EXAMPLE OF INVERSION OF CONTROL

A trip planner depends upon a cab agency and an airline agency to plan a trip



A ”NO CONTAINER” EXAMPLE 2

Property file defines actual implementations to be used

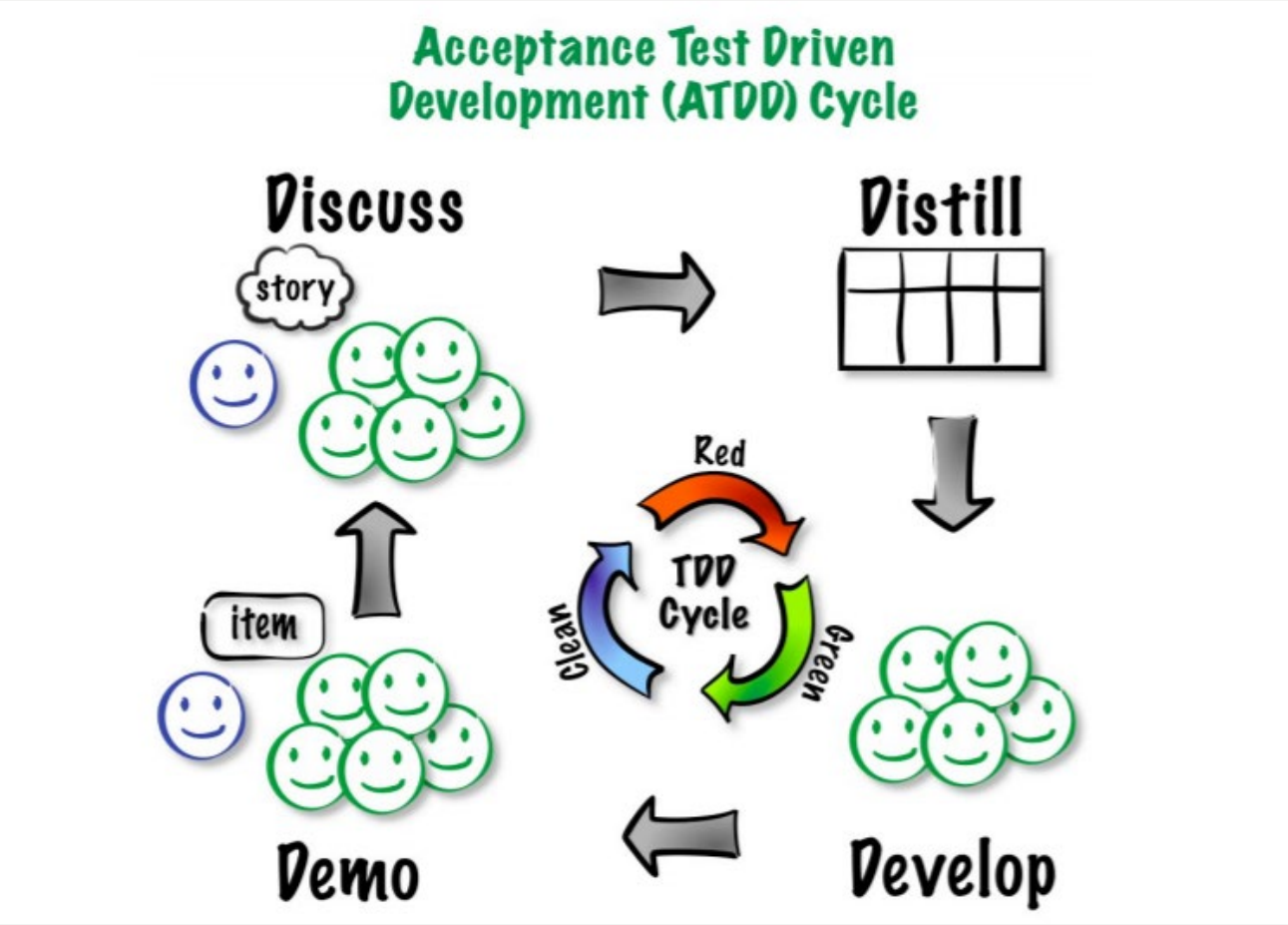
```
airline-agency-class = tdddemo.SimpleAirlineAgency
cab-agency-class = tdddemo.ConstructorBasedCabAgency
trip-planner-class = tdddemo.ConstructorBasedTripPlanner
```

A ”NO CONTAINER” EXAMPLE 3

Assembler code

```
Properties prop = new Properties();
prop.load(this.getClass().getResourceAsStream("/nocontainer-agency.properties"));
String trip = prop.getProperty("trip-planner-class"); //get all properties
…
Class tripPlannerClass = Class.forName(trip); // get all classes
…
if (TripPlanner.class.isAssignableFrom(tripPlannerClass)) {
        Constructor constructor = tripPlannerClass.getConstructor(new Class[]{AirlineAgency.class, CabAgency.class});
        tripPlanner = (TripPlanner) constructor.newInstance(new Object[]{airlineAgency, cabAgency});
}
```

DRIVING DEVELOPMENT WITH TESTS



Source: http://testobsessed.com/wp-content/uploads/2011/04/atddexample.pdf

EXPOSE UNCERTAINTY EARLY

- ATDD (accept criteria)
- BDD (By example)

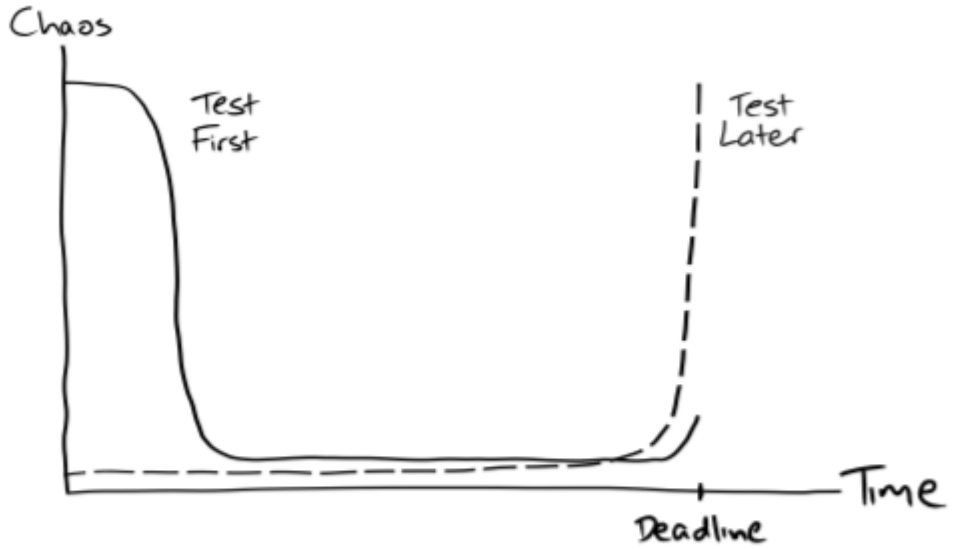


Figure 4.4 *Visible uncertainty in test-first and test-later projects*

"What about spaces?"

"What are examples of 'symbols'?"

"What should happen if a user enters an insecure password?"

"Can you give me examples of passwords you consider secure and insecure?"

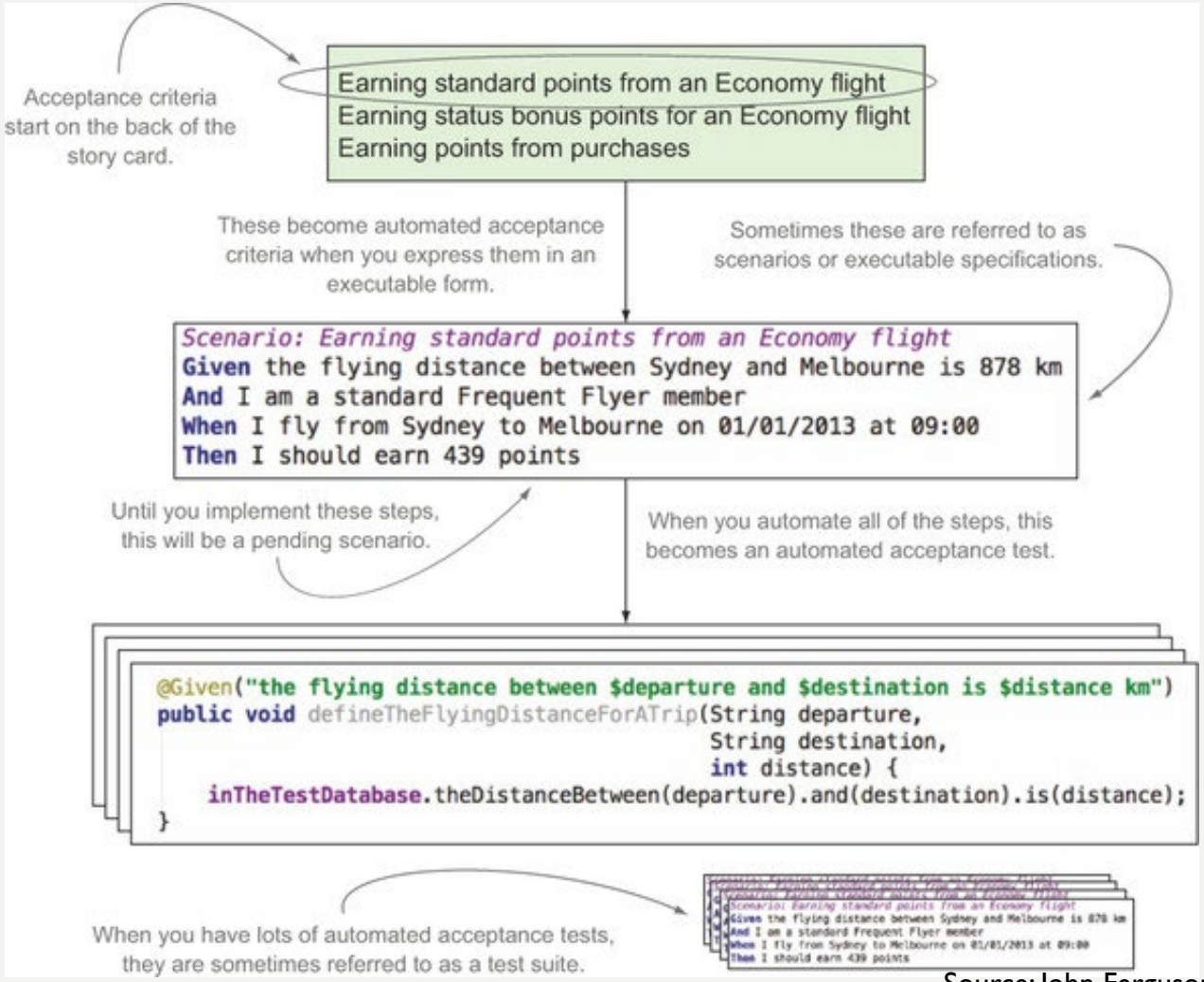
ACCEPTANCE CRITERIA TURNED INTO EXECUTABLE SPECIFICATIONS

BDD at acceptance test level:

Excutable tests written in a

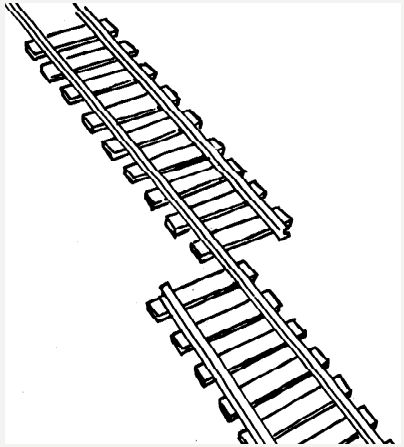
given_when_then

format (Gherkin)



Source: John Ferguson Smart- BDD in Action

STATIC TEST TECHNIQUES



- Reviews & Static Code Analysis
  - Why & when to use
  - Doing things 'messy' →technical debt: – changes later gets harder

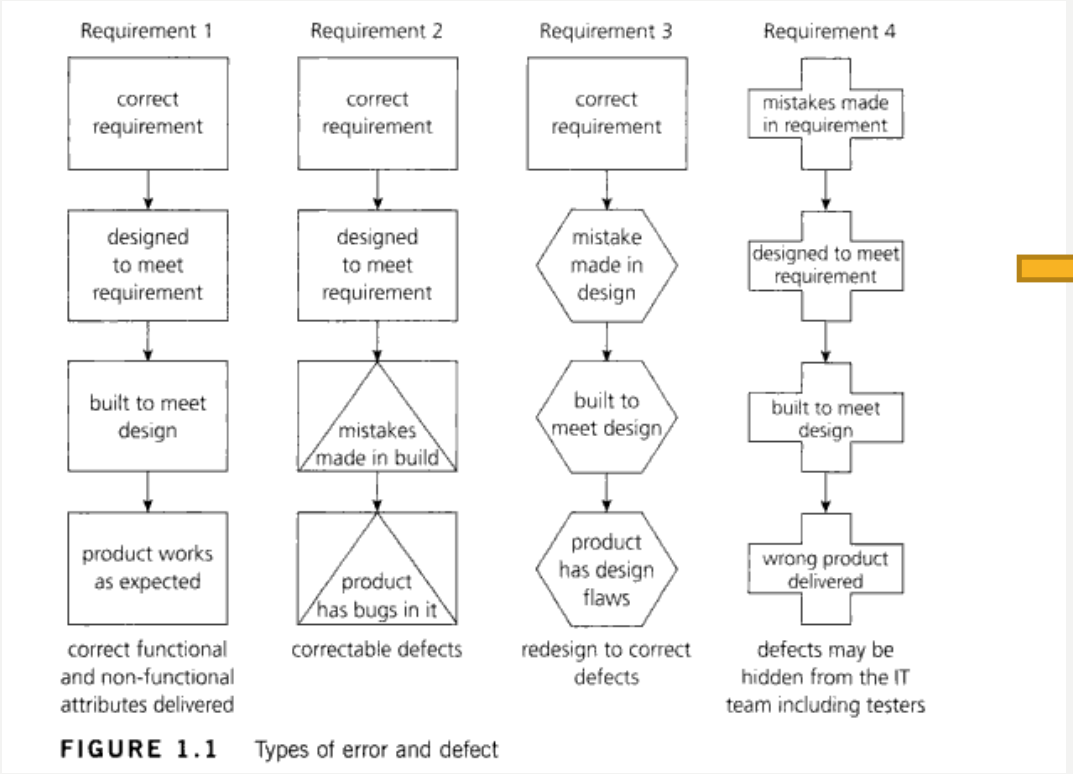
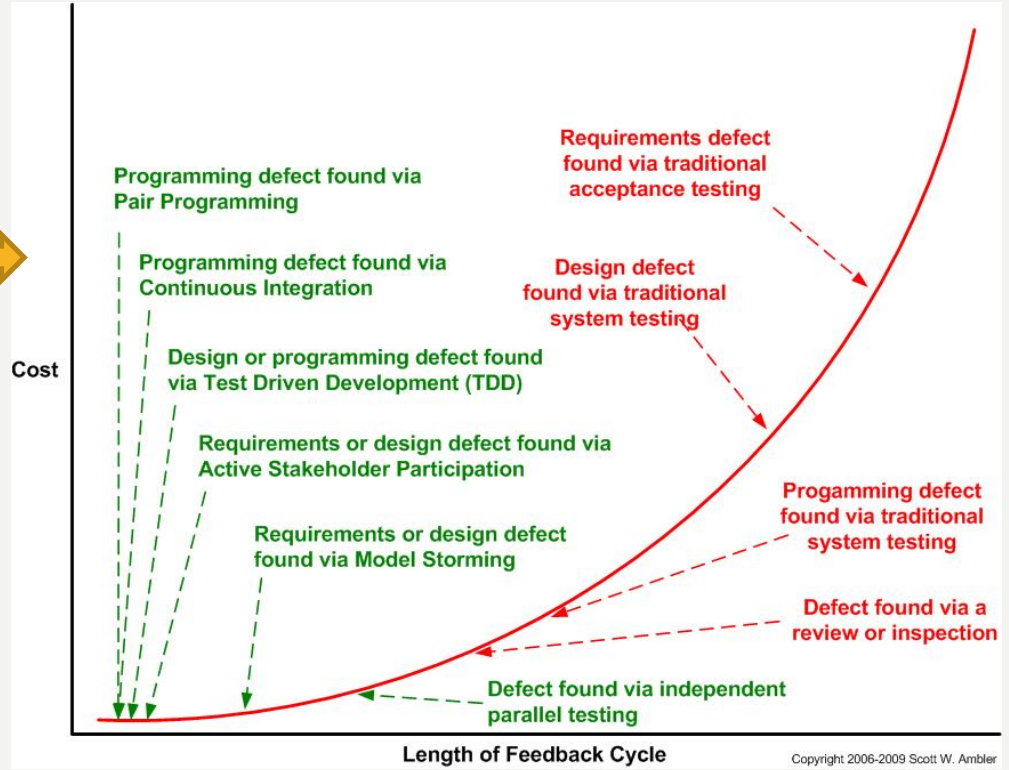


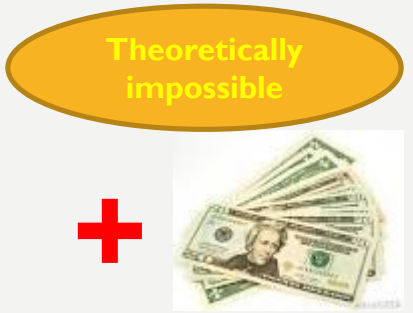
FIGURE 1.1 Types of error and defect



TEST CASE DESIGN

The fundamental problem of testing software

- We cannot make exhaustive testing →
  - Need to have a clever testing methodology
- Therefore, tests must be carefully designed



The process is as follows:

1. Test analysis

   Identify test conditions (i.e. something we could test)

2. Test design

   Specify test cases

3. Test implementation

   Specify test procedures (scripts)

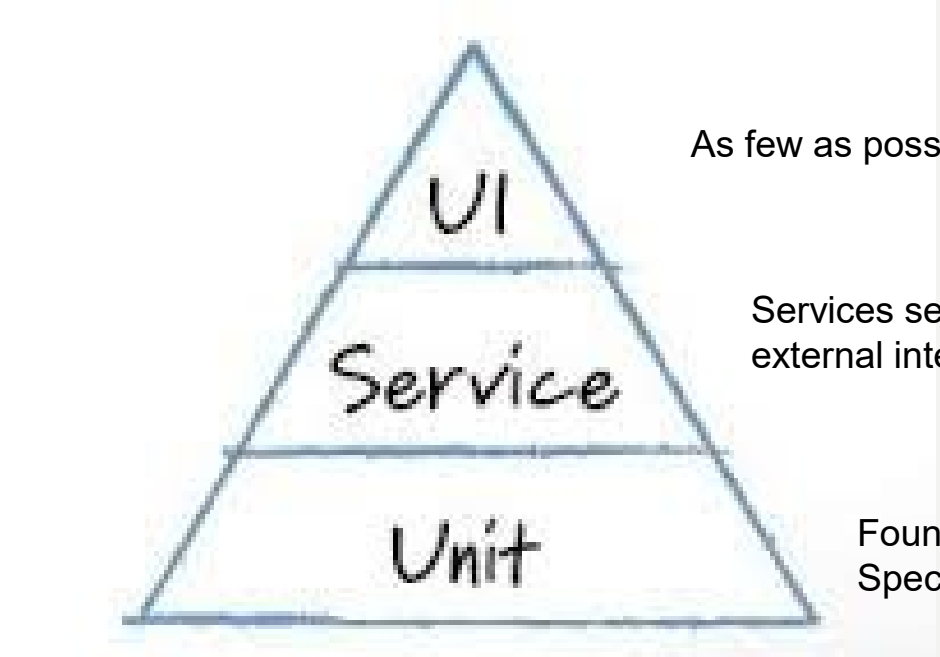
TEST CASE DESIGN TECHNIQUES

- **Specification-based /black-box techniques**
  - focuses on determining whether or not a system/ component does what it is supposed to do based on its functional requirements
  - appropriate at all levels of testing where a specification exists

- **Structure based / white-box techniques**
  - Internal structure is used to derive test cases
  - primarily used for unit and integration testing
  - Especially good if tool support for code coverage

TESTING IN SOFTWARE CYCLE

Effort of test automation



As few as possible – brittleness & time consuming

Services separate from UI – both sub system and external integration tests ~API testing

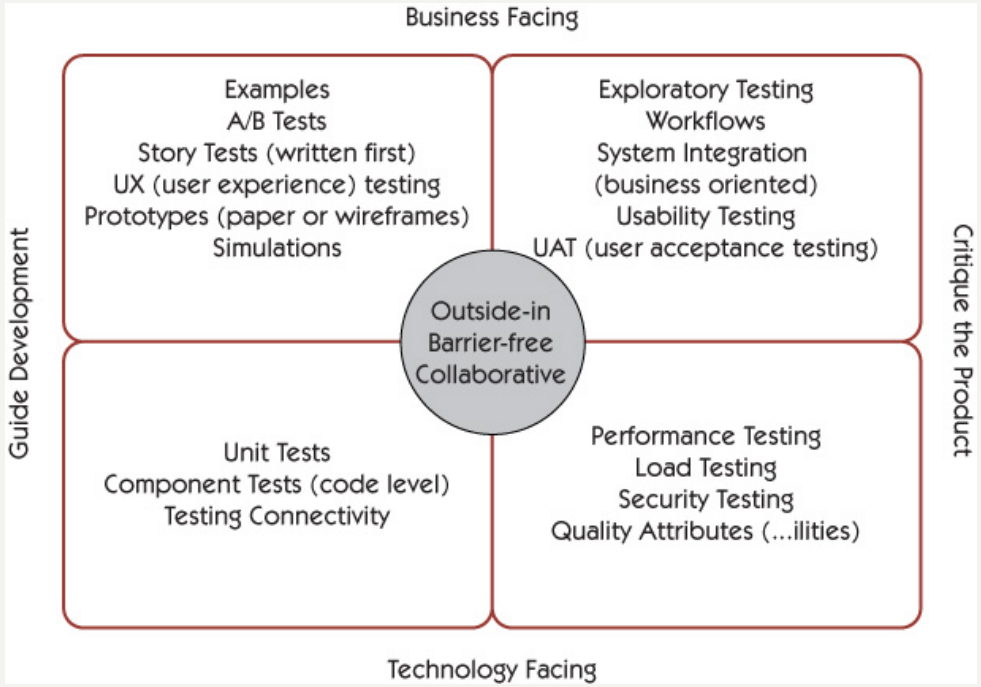
Foundation of automated test strategy -
Specific feedback to developer – bug at line 47!

TESTING IN SOFTWARE CYCLE

- Test roles and organization
  - Independent testers
  - Integrated test team
- Test tools in general - description and demonstration
  - Range from development tools to test manager tools
- Automated tests make regression testing much easier
  - Fast feedback
  - Balance between test coverage and speed
    - JUnit test suites < 10 minutes
    - Higher-level test suites < two hours (continuous integration server/automation engine)
    - Otherwise reorganize/reduce tests; get new hardware; run concurrently on VM's; in cloud;

TESTING IN SOFTWARE CYCLE

Agile Test Quadrants



TEST STRATEGIES & TECHNIQUES

- Measure test coverage
  - Black box + white box testing can be combined
- Write testable code
  - Avoid complexity, make code readable & testable
  - Do Test-Driven Development - ss more a programming practice that has automated tests as a result
- Rely on:
  - Test Pyramid for test automation principles
  - Agile Test Quadrants to get "all around the clock"
  - Static test techniques (reviews, static code analysis, coding standard)
  - CI / CD