Q & A

Test

PBA Softwareudvikling/BSc Software Development

Tine Marbjerg

Spring 2018

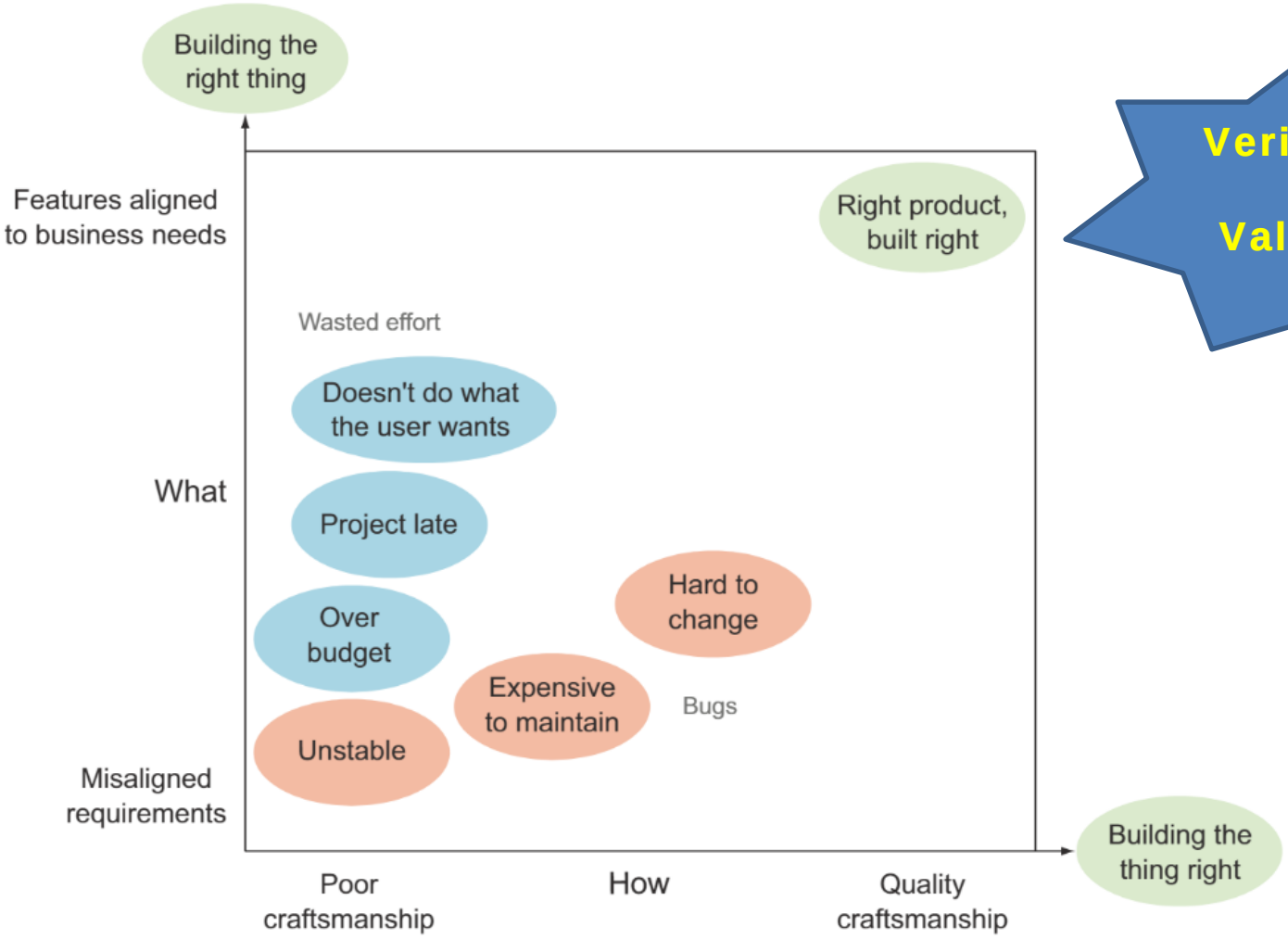
Overall Exam Topics

- Fundamentals of Testing
- Testing and Software Life Cycle
- Static Techniques
- Test Design Techniques
- Unit Testing
- Integration Testing
- System Testing
- Stress and load testing

General about platforms and tools

- Platforms and tools
  - You probably know by now, but you don't have to use:
    - Java
    - JUnit
    - Exact same tools as presented in class
- Exam questions structure
  - 80 – 20 %
  - general & practical focus
- Be prepared to show code and tests

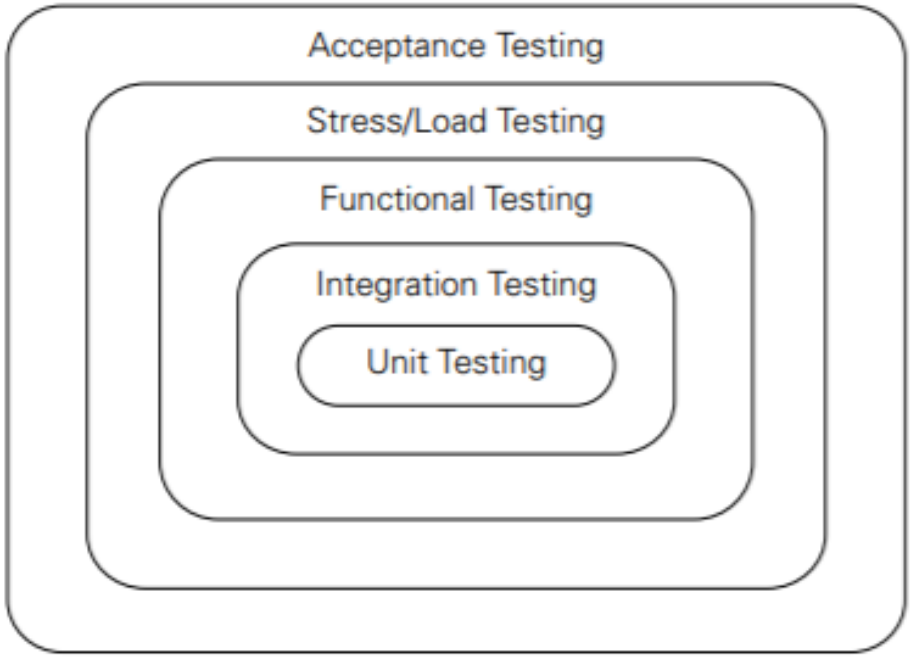
Build features well AND the right features



Verification & Validation

The five types of tests

- There are others non-functional test types, but these have been our focus:



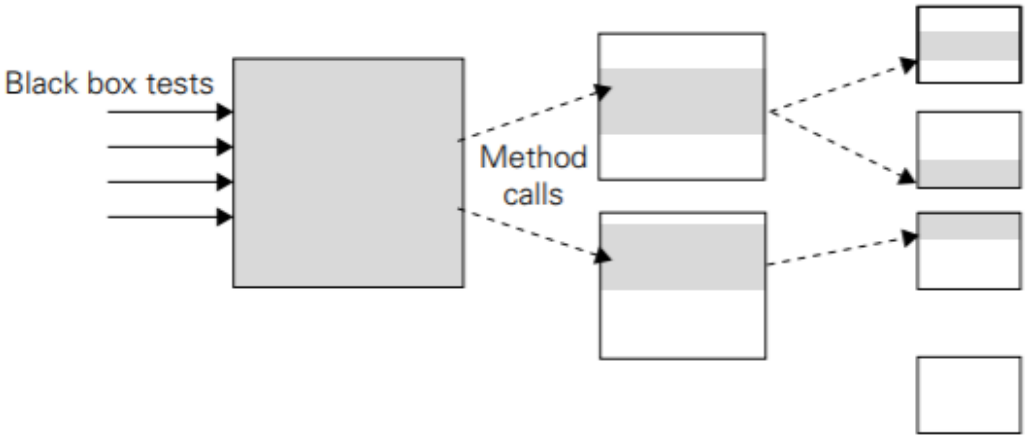
The outermost tests have broadest scope

The need for unit tests

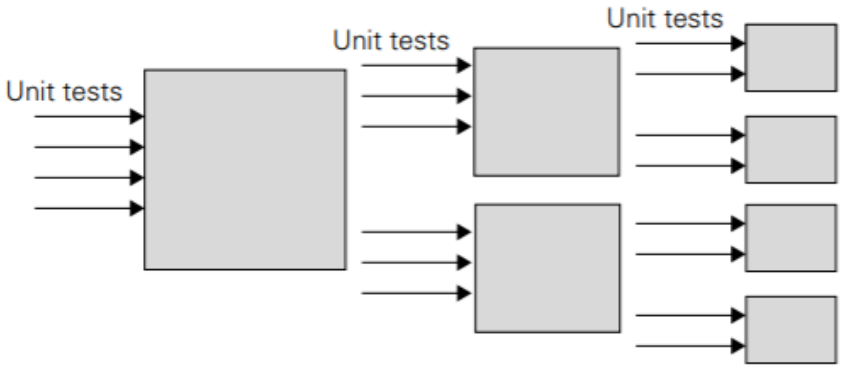
1. They allow greater test coverage than functional tests
2. Increase team productivity
3. Detect regressions and limit the need for debugging
4. They give us the confidence to refactor and make changes in general
5. Improve implementation
6. Document expected behavior
7. Enable code coverage and other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

1. Greater test coverage than functional tests

- Functional tests typically cover 70 % of the application:

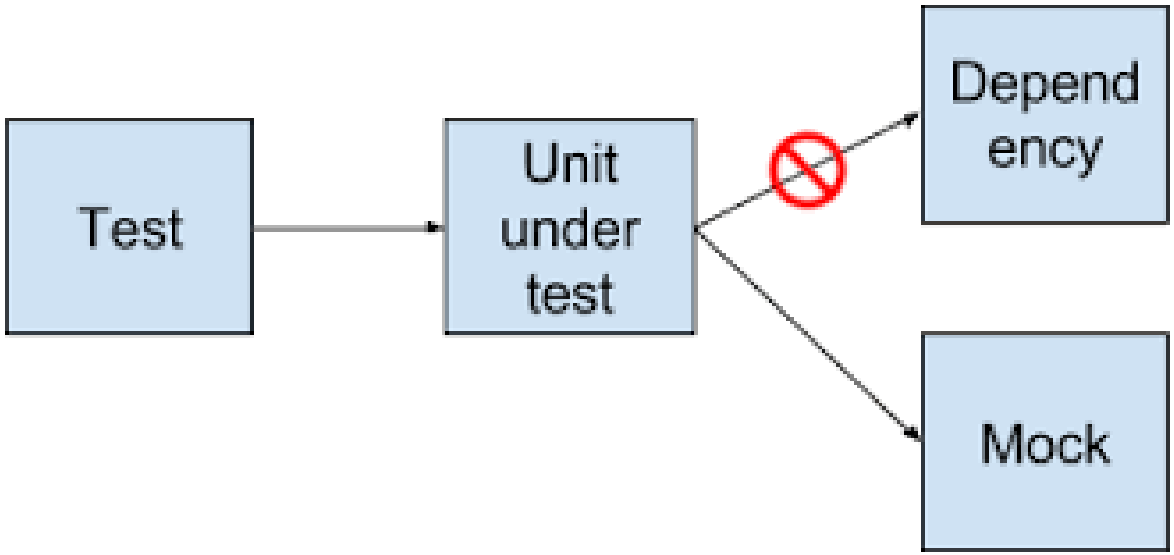


- Unit tests can achieve higher coverage (you can control input to each method and behavior of secondary objects):



2. Increase team productivity

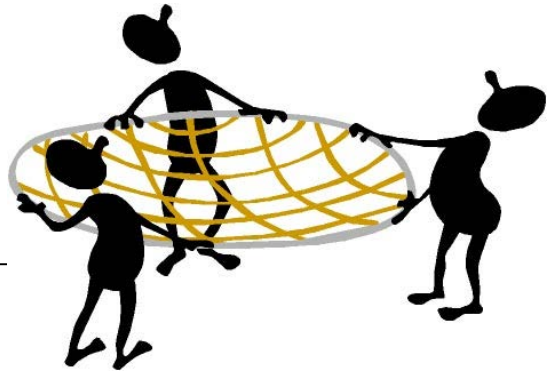
- You don’t have to wait for all other team members to finish their work



3. Detect regressions and limit the need for debugging

- A passing unit-test **confirms** your code works
- It gives you the **confidence** to modify your existing code, either for refactoring purposes or to add/modify new features.
- A good suite of unit tests reduces the need to debug an application to find out what's failing. Whereas 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
- Why would you risk spending hours of debugging time (and putting the delivery at risk) only to improve the implementation design, or change a method name
- Unit tests provide a safety net that gives you the courage to refactor.

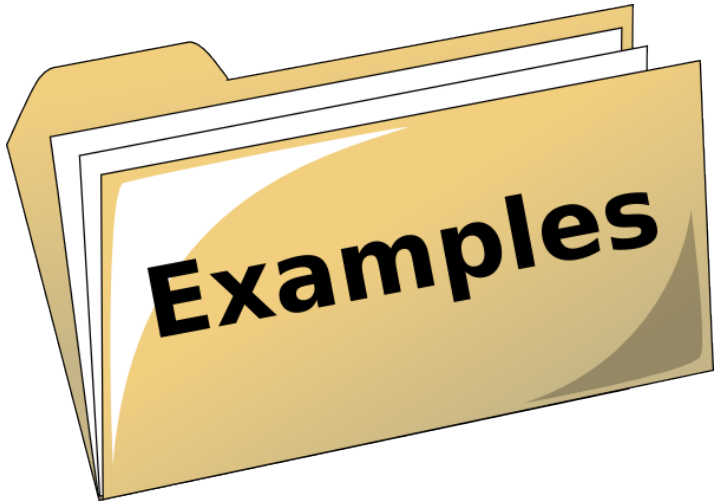
5. Improve implementation

- Unit tests are a first-rate client of the code they test
- They force the API under test to be flexible and to be unit-testable in isolation

- You usually have to refactor your code under test to make it unit-testable (or use TDD)

6. Document expected behavior

- Examples better than 300 page documentation describing the API



- Unit tests must always be up to date

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

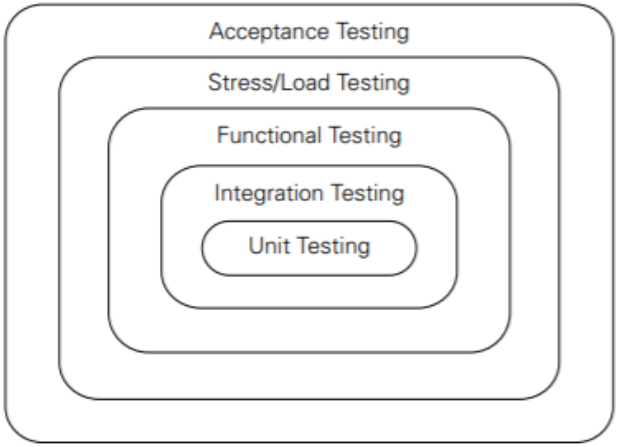
Source

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 comes after unit testing

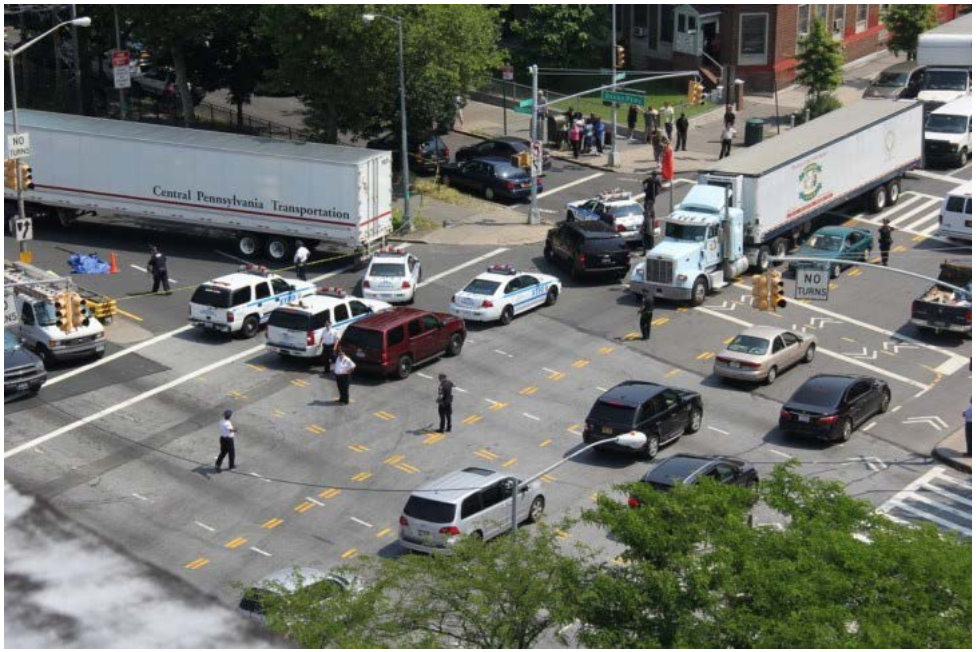
Quality control steps:



- 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

Why do integration testing?

- Just as more traffic collisions occur at intersections, the points where units interact are major contributors of software accidents



- *Ideally, integration tests should be defined before the units are coded.*
- *Being able to code to the test, increases a programmer's ability to write well-behaved objects ☺*

How to define 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)

3 levels of integration testing – JUnit in Action chap. 4

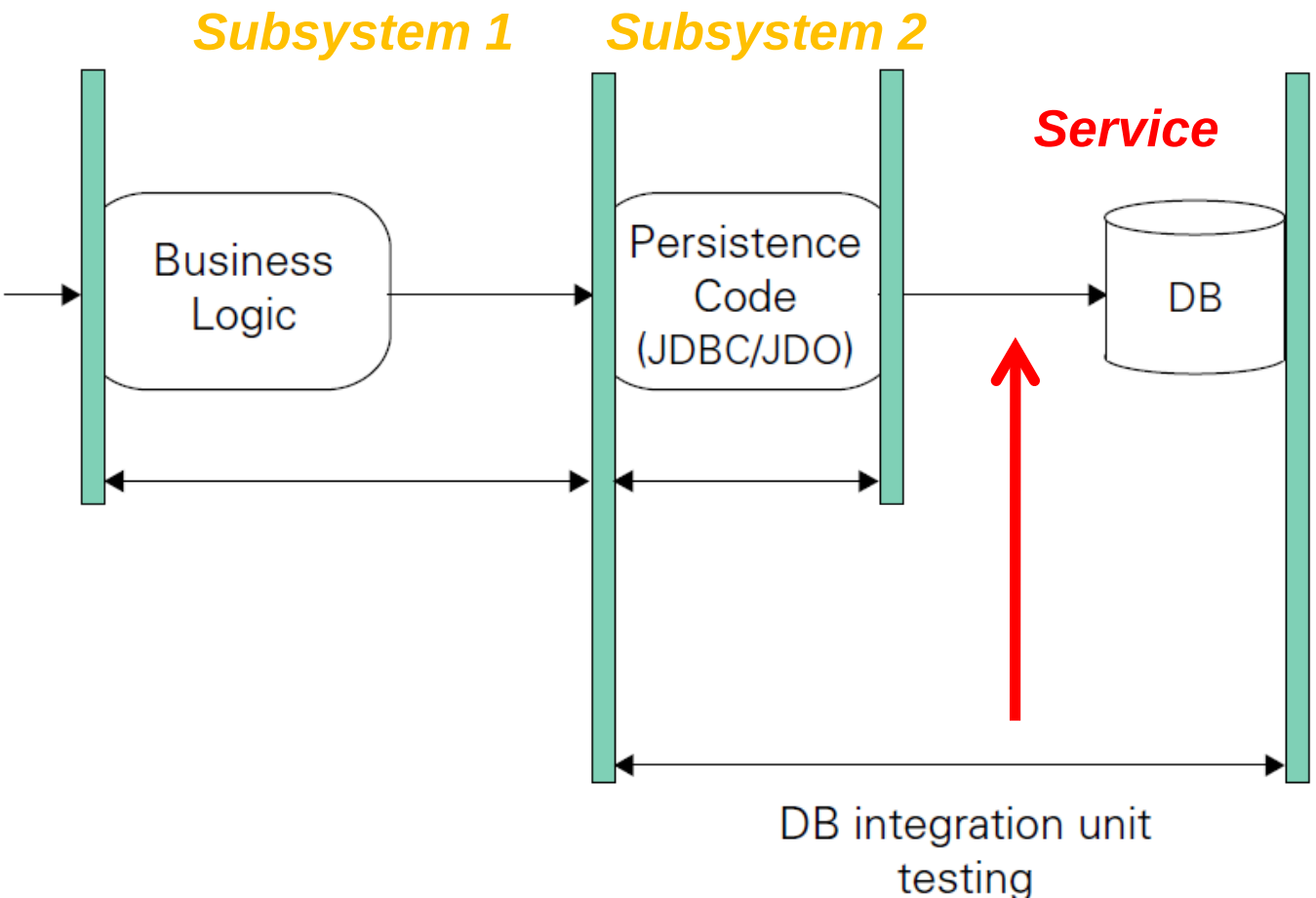
- Because there are many things with which you might want to integrate, the word integration can mean different things under different circumstances:

Table 4.1 Testing how objects, services, and subsystems interact

Interaction	Test description
Objects	The test instantiates objects and calls methods on these objects.
Services	The test runs while a servlet or EJB container hosts the application, which may connect to a database or attach to any other external resource or device.
Subsystems	A layered application may have a front end to handle the presentation and a back end to execute the business logic. Tests can verify that a request passes through the front end and returns an appropriate response from the back end.

Integration test at subsystem + service level

- Test connects to DB (or attach to external resource or device):



Integration testing approaches

- Big Bang!
- Top-down Incremental Test
- Bottom-up Incremental Test
- Functional Incremental Test

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

Leave out integration testing?

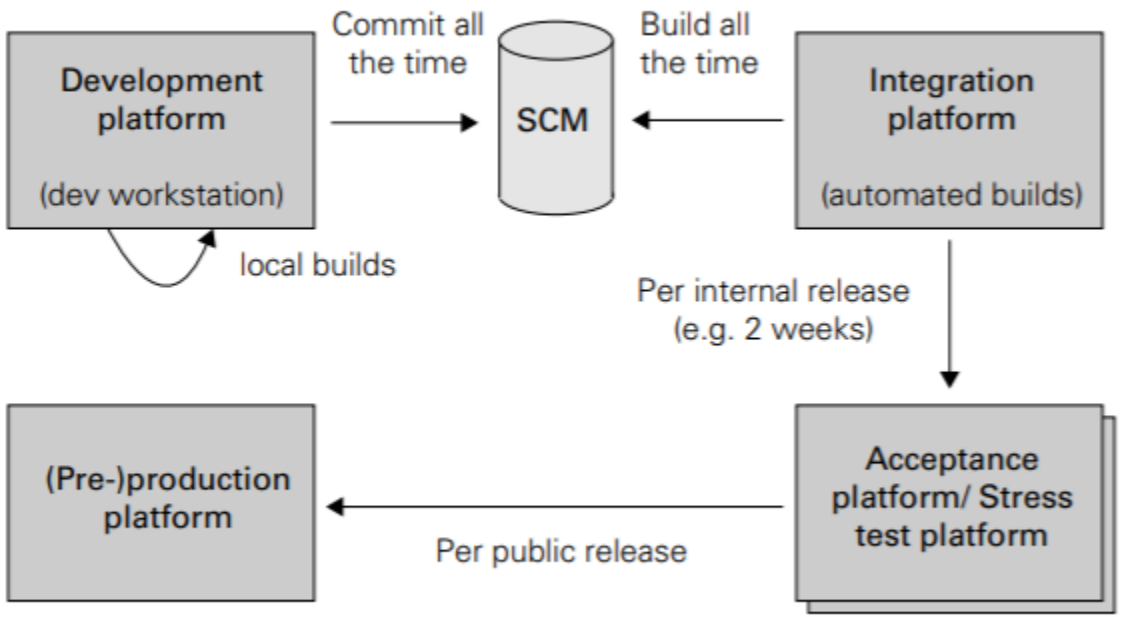
- DB driver should not be considered a risk – we expect third-party code to work
- Unit tests verify correct use of persistence API
- Hitting the database is slow
- Setting up test data must be managed

- But when only unit testing, we make *assumptions* about:
  - the database schema
  - the query statements used
  - the object-relational mappings

At some point we need to test if the assumptions are correct

Testing in the development cycle 1

Testing occurs at different places and times during development cycle:



Development platform—where the coding happens

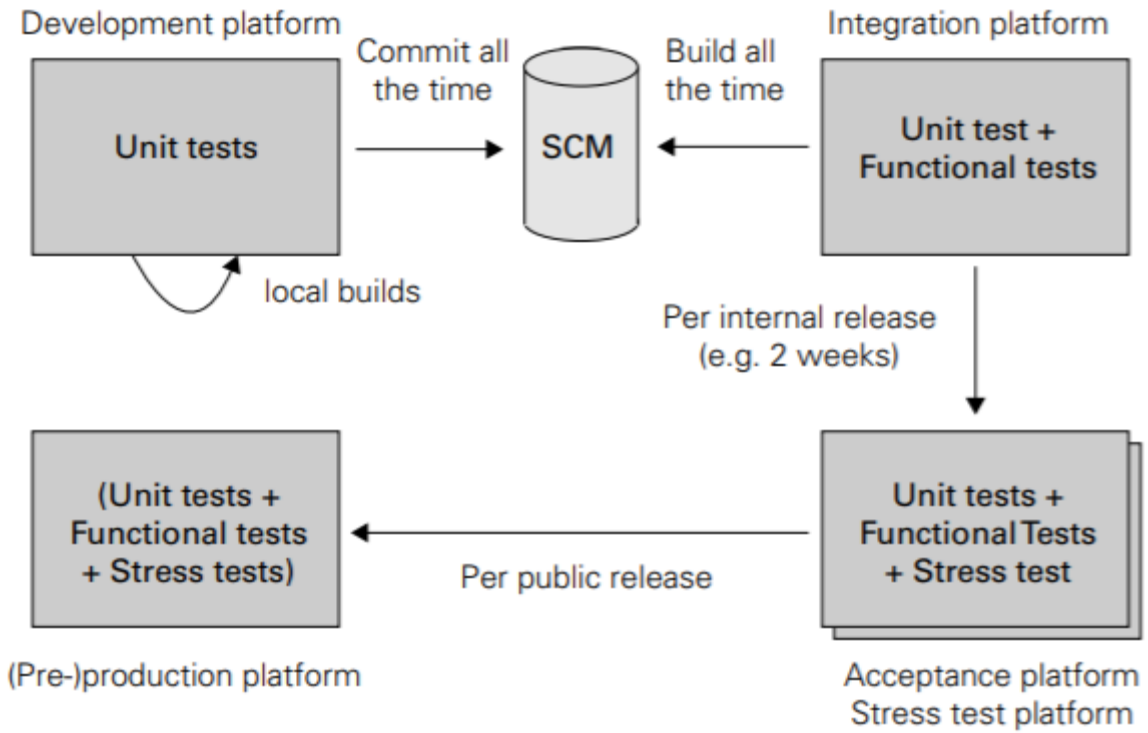
Integration platform builds the application from its different pieces and ensure that they all fit together

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 2

Test types on the different platforms



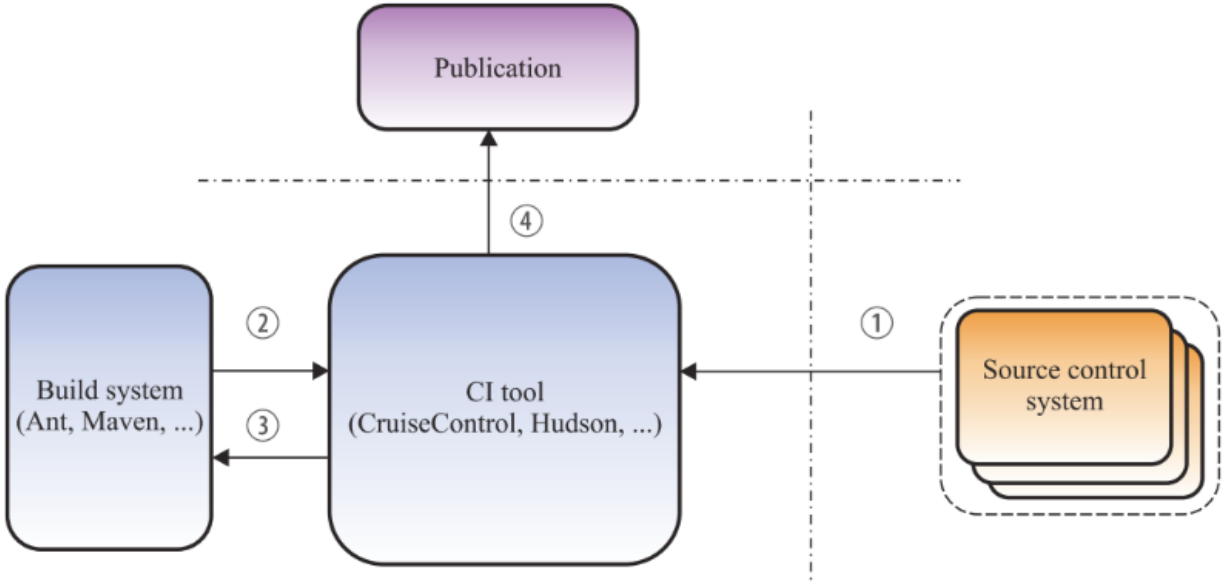
Development platform: unit tests (tests in isolation) – quick from IDE + your automated build before commit to version control sys.

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 extreme close to prod.

Continuos integration testing (fall semester curriculum)

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



Test Strategies & Techniques

- Measuring test coverage
  - Black box + white box testing can be combined
- Writing testable code
  - Avoid complexity, make code readable & testable
- Test-driven Development
  - Is more a programming practice that has automated tests as a result

Writing Testable Code (JUnit in Action section 5.2)

- 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 sta
  - ❑ Singletons pros and cons
  - ❑ Favour generic methods

Public APIs are contracts

- Never change the signature of public method!
- Why not?
  - Public methods become the articulation points of an application among components, open source projects, and commercial products that might not even know of one another's existence.

Reduce dependencies

- Reduce dependencies of other classes as much as possible!

- Why?
  - Tests become more complicated if classes to test depend on many other classes

Reduce dependencies (Example - a)

- Problem in example:
  - Every time we instantiate the **`Vehicle`** object, we also instantiate the **`Driver`** object
  - Application logic is mixed with instantiation code (factories)

```
class Vehicle {
   Driver d = new Driver();

   boolean hasDriver = true;

   private void setHasDriver(boolean hasDriver)
   {
        this.hasDriver = hasDriver;
   }
}
```

Reduce dependencies (Example - b)

- Solution to example:
  - Pass a **Driver** *instance* to the **Vehicle** class
  - Separates application logic and instantiation logic → We can mock any type of **Driver** implementation

```
class Vehicle {
   Driver d;
   boolean hasDriver = true;

   Vehicle(Driver d) {
        this.d = d;
   }
   private void setHasDriver(boolean hasDriver) {
        this.hasDriver = hasDriver;
   }
}
```

Create simple constructors

- Avoid logic in the constructor!
- Why?
  - If we mix **instantiation logic** and **other logic**, we always end up with a predefined object state
  - This mix can be hard to test, because we want to do this for each test case … and the wanted state might not be the same each time:
    - ❑ **Instantiate the class to test**
    - ❑ **Set the class into a particular state**
    - ❑ Assert the final state of the class

Arrange
Act
Assert

Follow the Principle of Least Knowledge

- The Law of Demeter states that one class should know only as much as it needs to know (encapsulation)

- Why?
  - To make **objects less dependent** on the internal structure of other objects, so these other object can be changed without reworking their calling objects →
  - **Information hiding** makes software more maintainable and adaptable

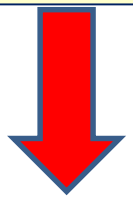
Follow the Principle of Least Knowledge (example)

- Problem in example:
  - Class **Vehicle** knows **Context.getDriver** method

```
class Vehicle{
   private Driver driver;

 Vehicle(Context context) {
        this.driver = context.getDriver();
   }
}
```

Solution: Require objects, don't search for objects



```
… Vehicle(Driver driver) {
     this.driver = driver;
   }…
```

Avoid hidden dependencies and global state

- Avoid global objects if they are not coded for shared access

- Why?
  - Can give unintended consequences
  - Client might expect exclusive access to global object

Avoid hidden dependencies and global state (example)

- Problem in example:
  - DBManager implies a global state.
  - Reservation object hides dependency upon a database manager.

```
public void reserve() {
        DBManager manager = new DBManager();
        manager.initDatabase();
        Reservation r = new Reservation();
        r.reserve();
}
```

- Solution to example:
  - Make use of DBManager explicit (as input parameter)

```
public void reserve() {
        DBManager manager = new DBManager();
        manager.initDatabase();
        Reservation r = new Reservation (manager);
        r.reserve();
}
```

Singletons pros and cons

- Pro
  - object is only instantiated once
- Cons
  - Can't call and test a private method directly (the constructor is private in Singleton pattern).
    - Solution
      - Rely on code coverage
      - Change access modifier while testing
      - Reflection
  - Introduces global state into code
    - when you provide access to a global object, you share not only that object but also any object to which it refers

Singleton implementation

```
public class Singleton {
   private  static Singleton INSTANCE;
   private  Singleton() {}
   public static Singleton getInstance() {
      if(INSTANCE == null) {
         INSTANCE = new Singleton();
      }
      return INSTANCE;
   }
}
```

Favour generic methods

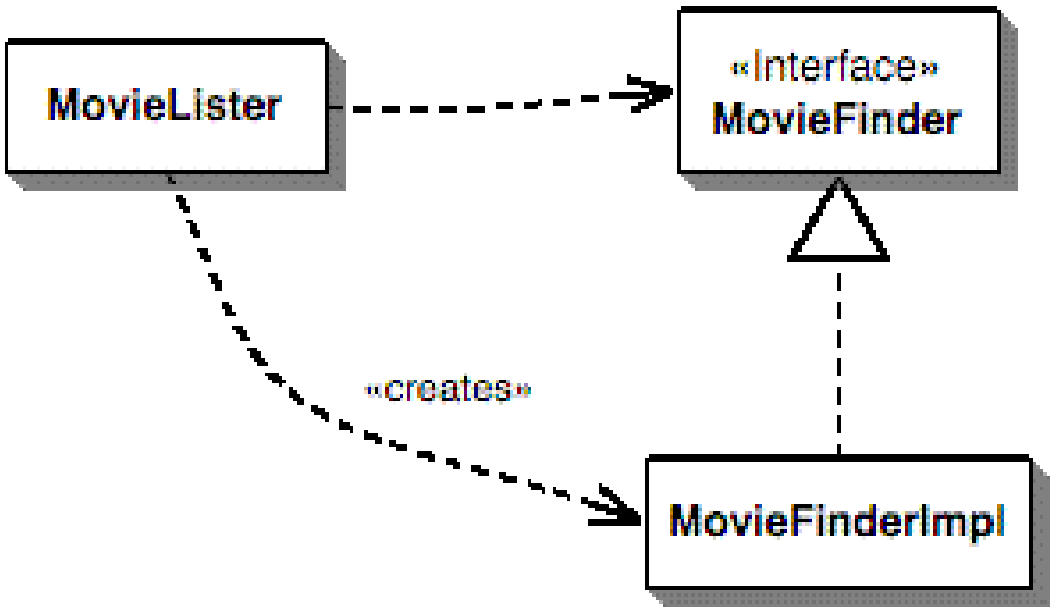
- Avoid static methods

- Why?
  - No ability to use polymorphism *(things are what they are at compile time)*
    - You cannot substitute code with test code
    - No code reuse for application either

    - OBS! Utility methods like **`Math.sqrt`** come naturally static

Favour generic methods - example

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



Favour composition over inheritance

- Do not reuse code with inheritance

- Why?
  - Inheritance hierarchy is rigid and we cannot change hierarchy *(sub class is always of certain superclass type – freedom is fixed at compile time)*
  - With composition we can compose objects differently → can switch from one state of our objects to another, i.e. more testable

  - So…
    Inheritance for polymorphism?

    Inheritance for reuse?

Favour polymorphism over conditionals

- Avoid long **`switch`** and **`if`** statements
- Why?
  - Conditional logic increases complexity

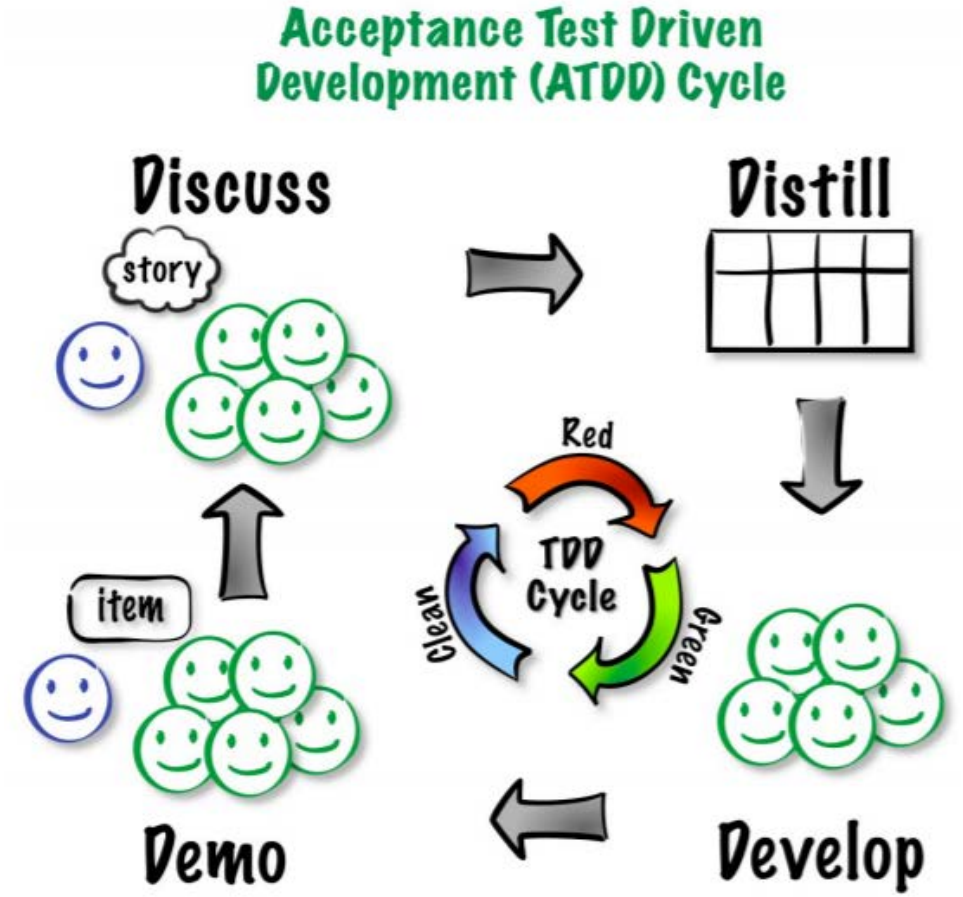
Favour polymorphism over conditionals - example

- Problem: Bad example with conditionals (TATH; listing 5.6)

```
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

Driving Development with Tests



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

Discuss Requirements

We ask the business stakeholder requesting the feature questions intended to elicit acceptance criteria:

“What should happen if a user enters an insecure password?”

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

“What are examples of ‘symbols’?”

“What about spaces?”

“What about dictionary words with obvious substitutions that meet the criteria but still may be insecure, like ‘p@ssw0rd’?”

“What about existing accounts?”

“How will you know this feature ‘works’?”

Distill Tests in a Framework-Friendly Format

Examples

Valid passwords that should result in SUCCESS: “p@ssw0rd”, “@@@000dd”, “p@ss w0rd”, “p@sw0d”

Invalid passwords that should result in the error message, “Passwords must be at least 6 characters long and contain at least one letter, one number, and one symbol.”: “password”, “p@ss3”, “passw0rd”, “p@ssword”, “@@@000”

Format example for framework that can read HTML file:

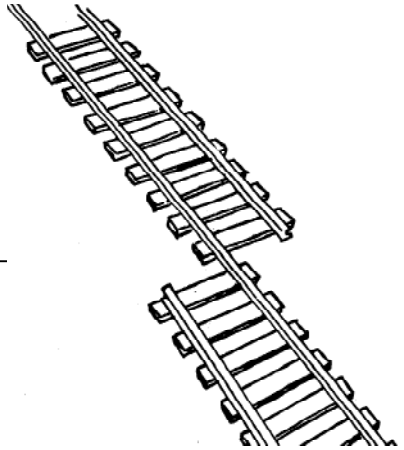
Test Case	Action	Argument
Verify valid and invalid passwords	Password Should Be Valid	p@ssw0rd
	Password Should Be Valid	@@@000dd
	Password Should Be Valid	p@ss w0rd
	Password Should Be Invalid	password
	Password Should Be Invalid	p@ss3
	Password Should Be Invalid	passw0rd
	Password Should Be Invalid	@@@000

Develop the Code (and Hook up the Tests)

Test cases:

Test Case	Action	Argument	Argument
Verify valid and invalid passwords	Create Login	fred	p@ssw0rd
	Message Should Be	SUCCESS	
	Attempt to Login with Credentials	fred	p@ssw0rd
	Message Should Be	Logged In	
	Create Login	fred	@@@000dd
	Message Should Be	SUCCESS	
	Attempt to Login with Credentials	fred	@@@000dd
	Message Should Be	SUCCESS	
	...etc...		

Static test techniques 1



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

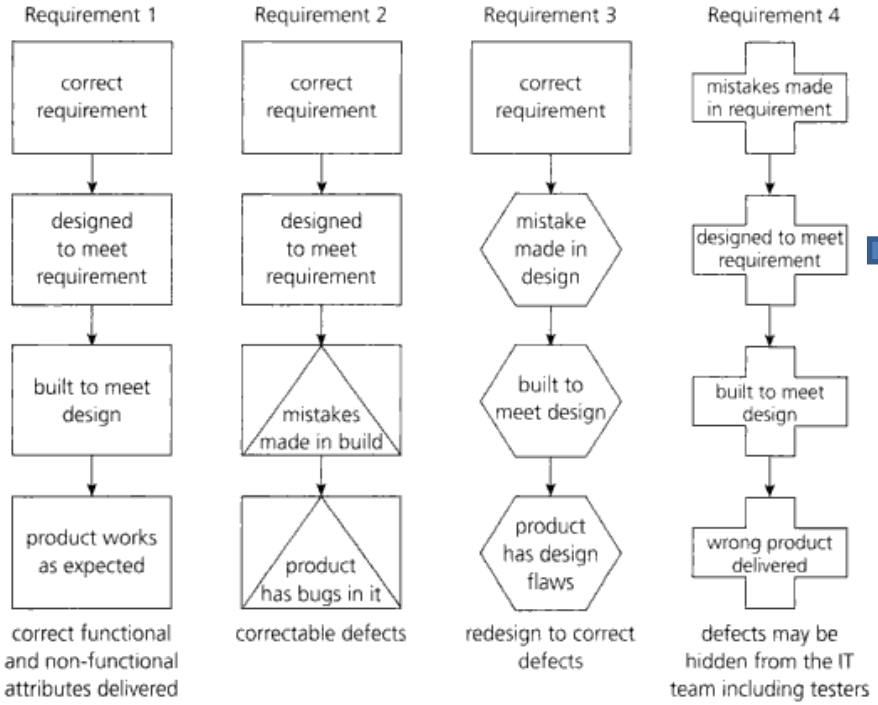
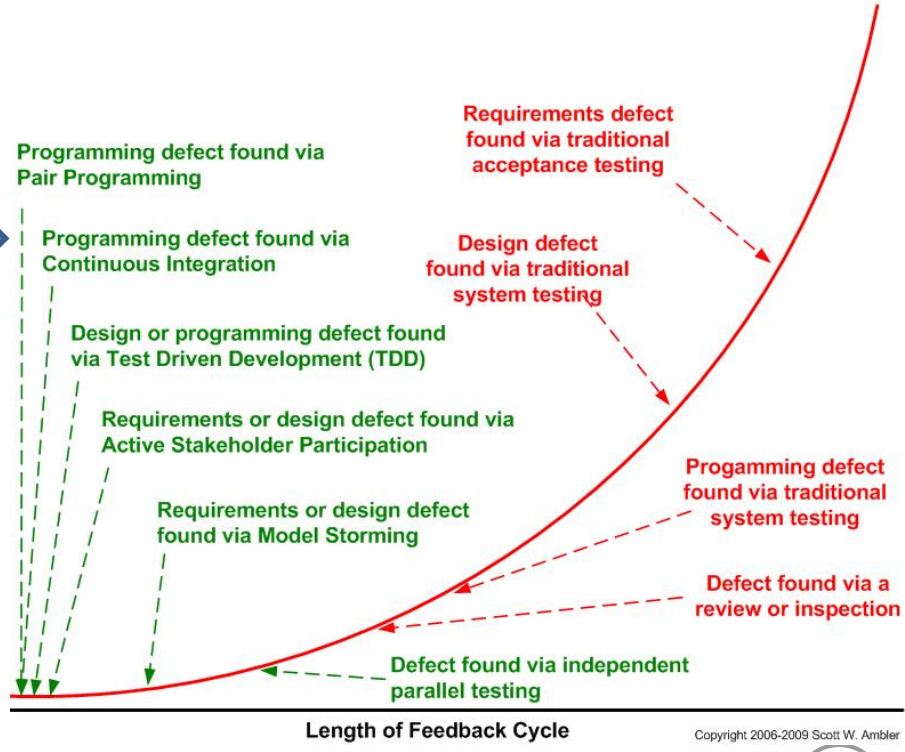


FIGURE 1.1 Types of error and defect



Static test techniques 2

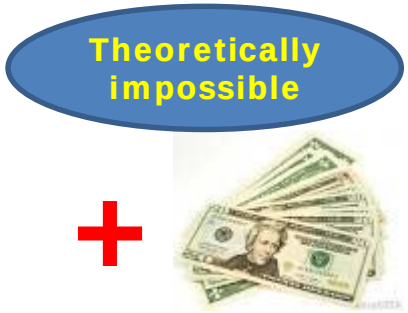
- **Review**–different types & degree of formality
  - **Walkthrough** -Led by the author who presents his artifacts
  - **Technical review** - Discussion meeting to achieve consensus about the technical content of an artifact
  - **Inspection** – formal review roles and guidelines

- **Static code analysis** - Code metrics
  - Cyclomatic complexity (no. of decisions in program), depth of inheritance, Class coupling etc.
  - Know your tool
  - Know about purpose of structural measurement
  - Prepare good examples (or use screen dumps from assignments)

Test Case Design 1

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 2

- **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
- **Experience-based techniques**
  - People's knowledge, skills, and background is prime contributor to test conditions and test cases

Test Case Design 3 -Black-Box Techniques

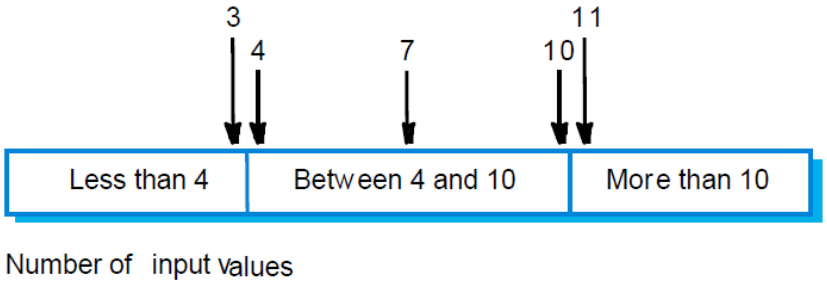
- Why and how
  - Equivalence partitioning
  - Boundary analysis
  - Decision tables
  - State transition
  - Use case based (see system test)

Use your assignment solutions!!!

- All techniques should lead you to test cases!
  - described in simple tables for manual test or
  - written in code (automated test, like JUnit)

Test Case Design 4 – purpose of techniques

- Equivalence partitioning
- Boundary analysis



- Decision tables

TABLE 4.4 Decision table with combinations and outcomes				
Conditions	**Rule 1**	**Rule 2**	**Rule 3**	**Rule 4**
Repayment amount has been entered	T	T	F	F
Term of loan has been entered	T	F	T	F
Actions/Outcomes				
Process loan amount	Y	Y		
Process term	Y		Y	

- State transition

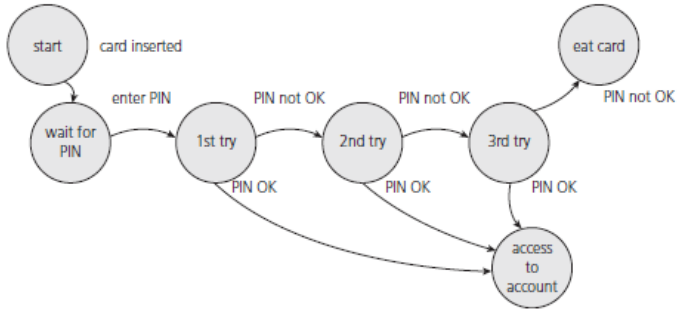


FIGURE 4.2 State diagram for PIN entry

Test Case Design 5 - Test Case Examples

Table template:

Test Case	Special Notes	Input			Expected Output		
		Var 1	Var 2	...	Var 1	Var 2	...
TC001							
TC002							
...							

JUnit example:

```
@Test
public void testGetPart()
{
  Part p = pm.getPart(10506);
  assertTrue("Get Part failed1", p != null);
  assertTrue("Get Part failed2",p.getPno()== 10506);
}
```

Test Case Design 6 – State Transition

- When some aspect of system/component can be described as ‘state machine’
  - Any system where you get a different output for the same input, depending on what has happened before, is a finite state system.
- A state transition model has four basic parts:
  - The **states** that the software may occupy (open/closed or funded/insufficient funds);
  - The **transitions** from one state to another (maybe not all transitions allowed);
  - The **events** that cause a transition (e.g. closing a file or withdrawing money);
  - The **actions** that result from a transition (e.g. an error message or being given your cash).

Test Case Design 7 – State Transition

- You can model use cases, sub systems, but often a class
- You want to illustrate legal and illegal transitions
- States for bug in bug tracking system:
  - New
  - Assigned
  - Closed
  - Reopened
- Have you tested for invalid transitions?

Test Case Design 8 – State Transition

- State diagram

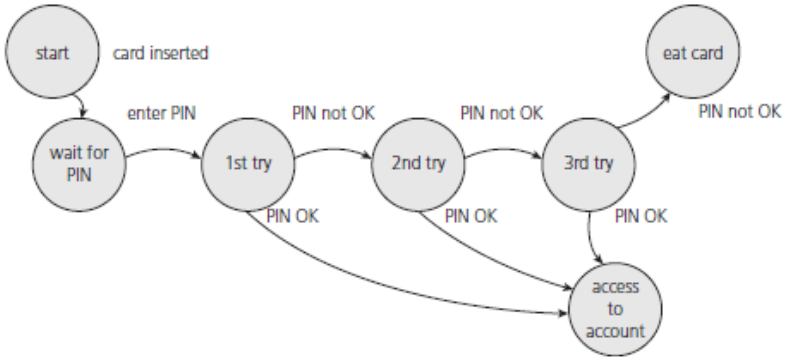


FIGURE 4.2 State diagram for PIN entry

- State table

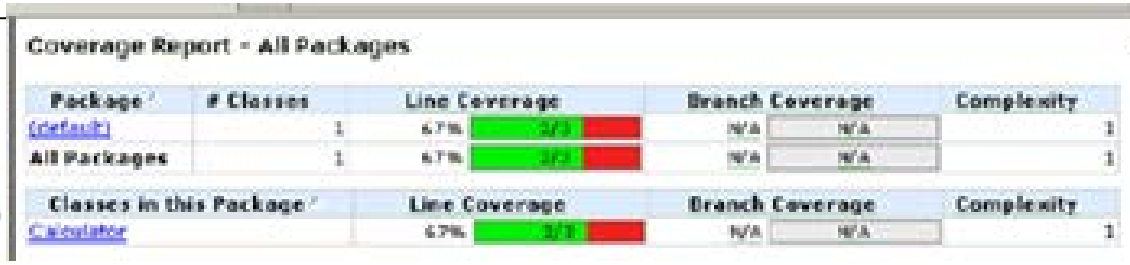
TABLE 4.9 State table for the PIN example

	Insert card	Valid PIN	Invalid PIN
S1) Start state	S2	–	–
S2) Wait for PIN	–	S6	S3
S3) 1st try invalid	–	S6	S4
S4) 2nd try invalid	–	S6	S5
S5) 3rd try invalid	–	–	S7
S6) Access account	–	?	?
S7) Eat card	S1 (for new card)	–	–

Test Case Design 9 -White-Box Techniques

Code coverage

$$\text{Coverage} = \frac{\text{Number of coverage items exercised}}{\text{Total number of coverage items}} \times 100\%$$



Use your assignment solutions!

- Statement coverage
  - your tests have exercised all statements in the code.
- Decision coverage
  - every true and false outcome of each condition is tested
- Remember that test coverage can cover non-code issues, e.g. structural items such as how many menu items have been tested or how many help screens.

Test Case Design 10–Experience-based Techniques

- Error guessing
  - A complement to more formal techniques
  - Depends on skills of tester (experience & gut feeling)
    - Examples: division by zero, blank input, empty file
  - No rules
- Fault attacks
  - Evaluate reliability by attempting to force specific failures to occur
- Exploratory testing
  - Popular in agile context
  - Based on mission charter: new tests adapts to previous test results
  - Time boxed & hands-on
  - Requires experience
  - Different styles

Testing & System Development Methods 1

Test models: Test is part of software development life cycle

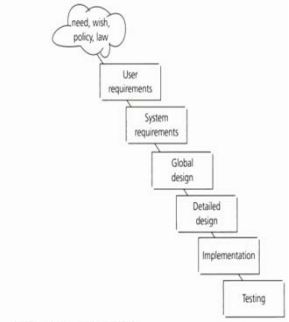


FIGURE 2.1 Waterfall model

- Waterfall
  - In the end of the cycle
- V-model
  - Specifies testing activities in each phase of the cycle

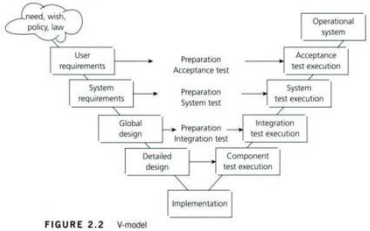
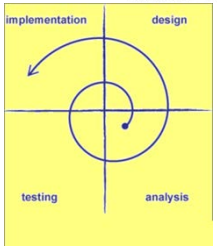
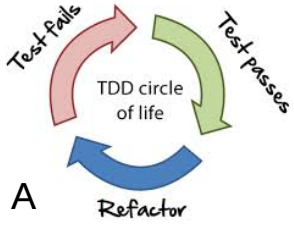


FIGURE 2.2 V-model

- Spiral testing
  - Applied to software increments in each iteration



- Agile testing
  - E.g. XP - a. test throughout process; b. customer involvement; c. testers & developers collaborate; d. test often and in small chunks
- Test-driven development
  - Requirements specified as tests



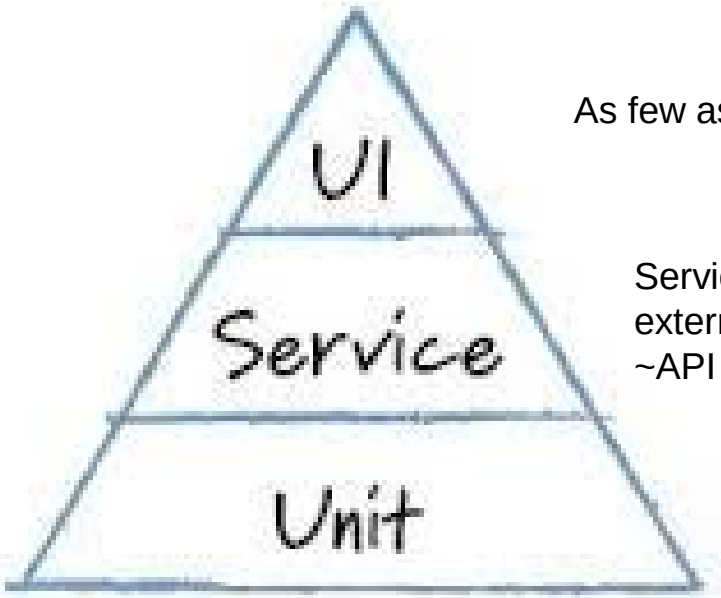
Testing & System Development Methods 2

Test levels

- Unit testing
  - Every component is tested independently: components are stable
- Integration testing
  - Internal interfaces (components & sub systems)
  - External systems
  - Interfaces are stable
- System testing (Functional testing)
  - Is the system complete and correct?
  - E.g create new customer; delete customer info etc.
  - More ‘mechanical’ than user testing, e.g. all CRUD operations in one test activity
- User testing
  - Fit for purpose?
  - Real life usage –scenarios (more natural flow that system testing)
- Acceptance testing
  - More basis for decision than actual testing: final approval of system

Testing & System Development Methods 3

Effort of test automation



As few as possible – brittleness & time consuming

Services separate from UI – both sub system and external integration tests (forgotten layer in agile) ~API testing

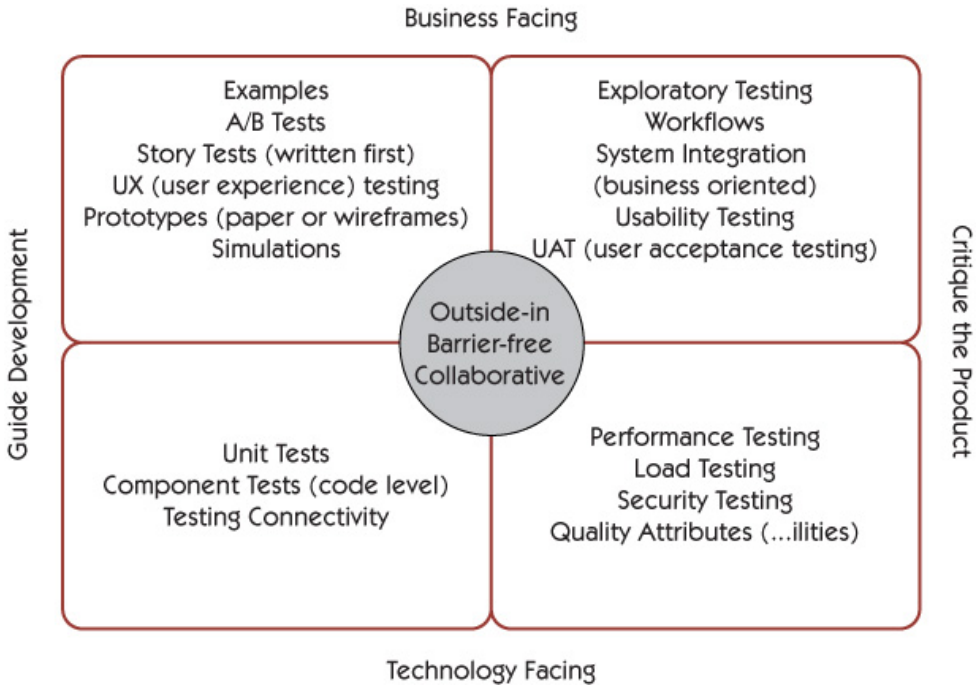
Foundation of automated test strategy - Specific feedback to developer – bug at line 47!

Testing & System Development Methods 4

- 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
    - Otherwise reorganize tests; get new hardware; run concurrently on VM's

Testing & System Development Methods 5

Agile Test Quadrants



Test Management 1

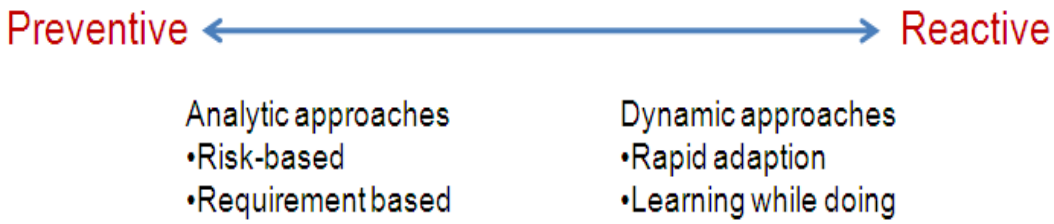
- Central Test Management activities
  - “Choosing” test organization and roles
  - Test planning and estimation
  - Test monitoring
  - Configuration Management
  - Incident management

Test Management 2

Planning

- Careful planning can reduce costs by
  - early development of test cases (V model)
  - Reduce elapsed time by parallelizing tasks (V model)
  - (Early) reviews
  - Agile techniques to get requirements right early

- Test Strategy
  - Management decides how much the organization will invest in system reliability
  - Test strategy is the general way testing will happen within each test level, independent of project, across organization.
  - Approaches



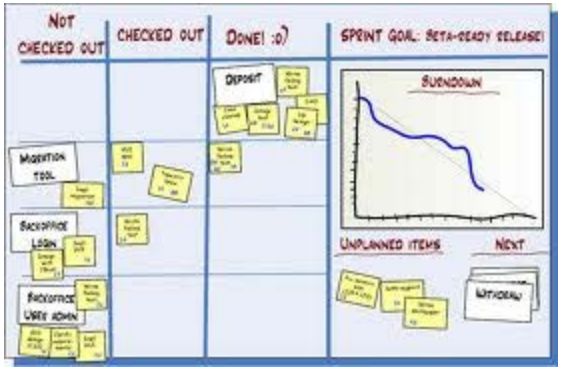
Test Management 3

Configuration management

- Configuration management is the discipline of managing and controlling change in evolution of software system
  - git, svn, cvs etc.
- Most configuration management systems have Build Management
  - Automatic building of system as developers create new versions of the components
  - maven, gradle, ant etc.

Test Management 4 - Tools

- Consider tools that you would introduce in project if you were “test manager” (in charge of sw quality):
  - Planning and monitoring
    - Make direction and results visible
  - Incident management
  - Test automation
  - Coverage measurement
  - Static analysis tool
  - Other?



Test Management 5

System Criticality influences on

- testing effort
- choice of test model

Level 1 Life critical systems

- People can be hurt due to system defects

Level 2 Mission critical systems

- System controls irreversible processes , e.g. production in factory

Level 3 Management Information Systems

- Administrative systems where corrections can be made at reasonble price

Level 4 “List” systems

- Produce information at service level and defect doesn’t go into other systems