Test-Driven Development

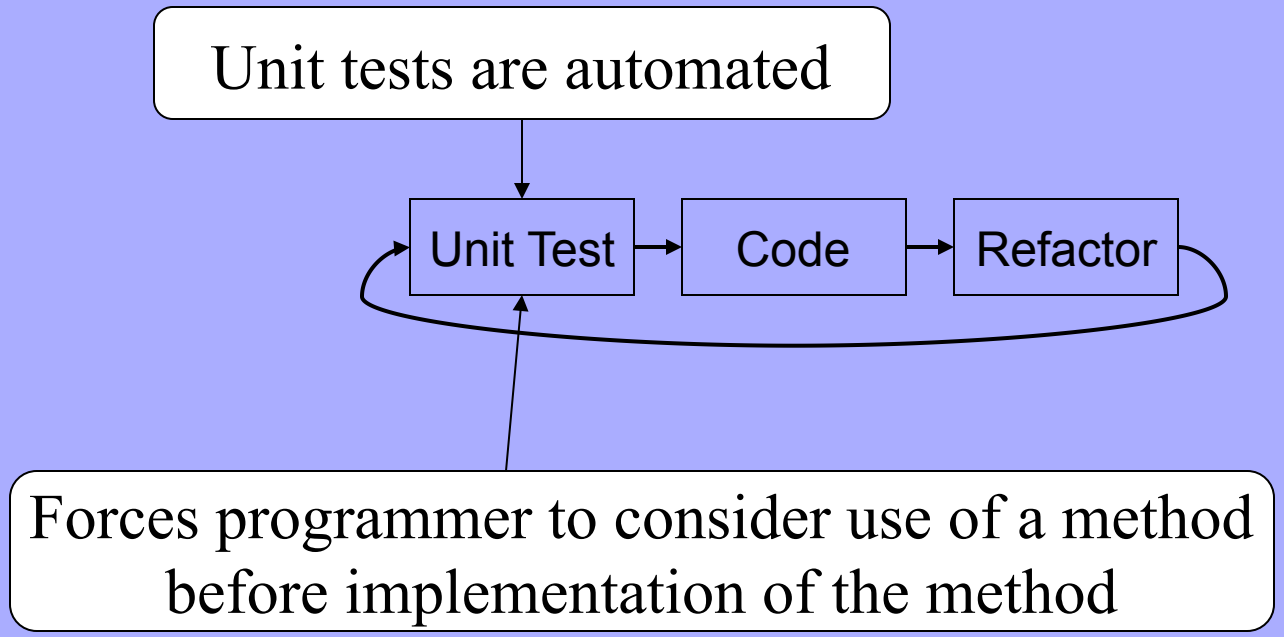
David Janzen

Outline

- What is TDD?
  - An example
  - Definitions
- What is not TDD?
- Where did TDD come from?
- Why should I use TDD?
- How can I apply TDD effectively?

What is Test-Driven Development?

- TDD is a design (and testing) approach involving short, rapid iterations of



TDD Example: Requirements

- Ensure that passwords meet the following criteria:
  - Between 6 and 10 characters long
  - Contain at least one digit
  - Contain at least one upper case letter

TDD Example: Write a test

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

public class TestPasswordValidator {
  @Test
  public void testValidLength() {
     PasswordValidator pv = new PasswordValidator();
     assertEquals(true, pv.isValid("Abc123"));
  }
}
```

Needed for JUnit

This is the teeth of the test

Cannot even run test yet because PasswordValidator doesn't exist!

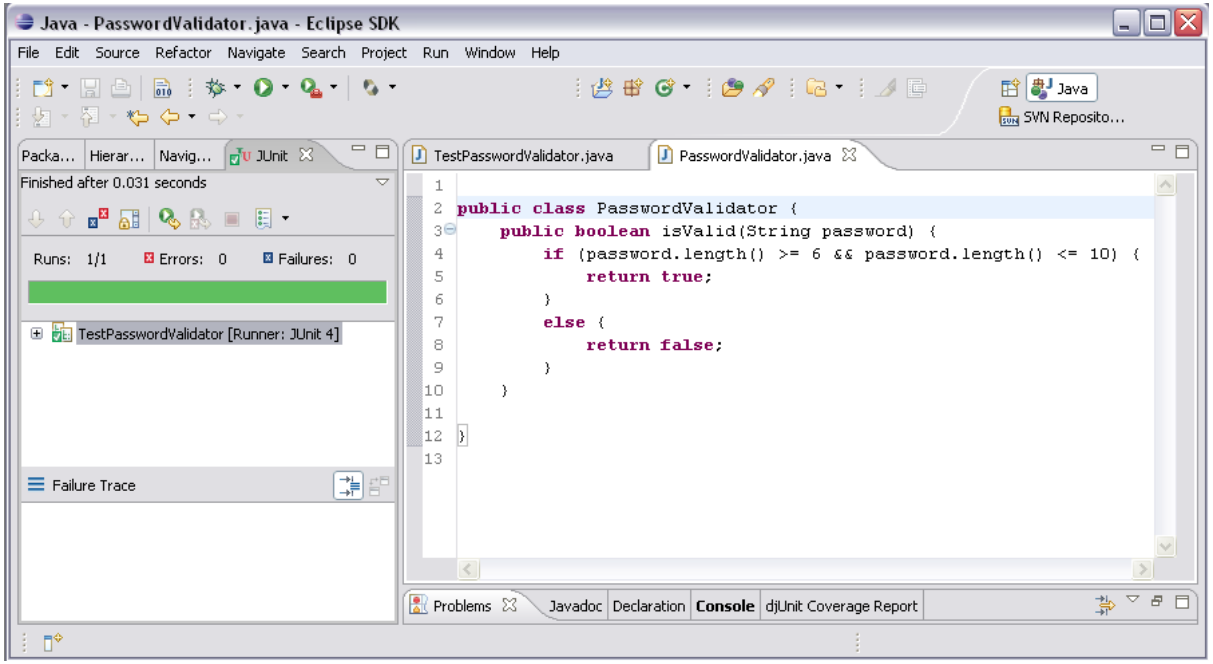
TDD Example: Write a test

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

public class TestPasswordValidator {
  @Test
  public void testValidLength() {
     PasswordValidator pv = new PasswordValidator();
     assertEquals(true, pv.isValid("Abc123"));
  }
}
```

Design decisions:
class name, constructor,
method name, parameters and return type

TDD Example: Write the code

```
public class PasswordValidator {
   public boolean isValid(String password) {
      if (password.length() >= 6 && password.length() <= 10) {
         return true;
      }
      else {
         return false;
      }
   }
}
```



TDD Example: Refactor

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

public class TestPasswordValidator {
  @Test
  public void testValidLength() {
     PasswordValidator pv = new PasswordValidator();
     assertEquals(true, pv.isValid("Abc123"));
  }
}
```

Do we really need an instance of PasswordValidator?

TDD Example: Refactor the test

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

public class TestPasswordValidator {
  @Test
  public void testValidLength() {
     assertEquals(true, PasswordValidator.isValid("Abc123"));
  }
}
```

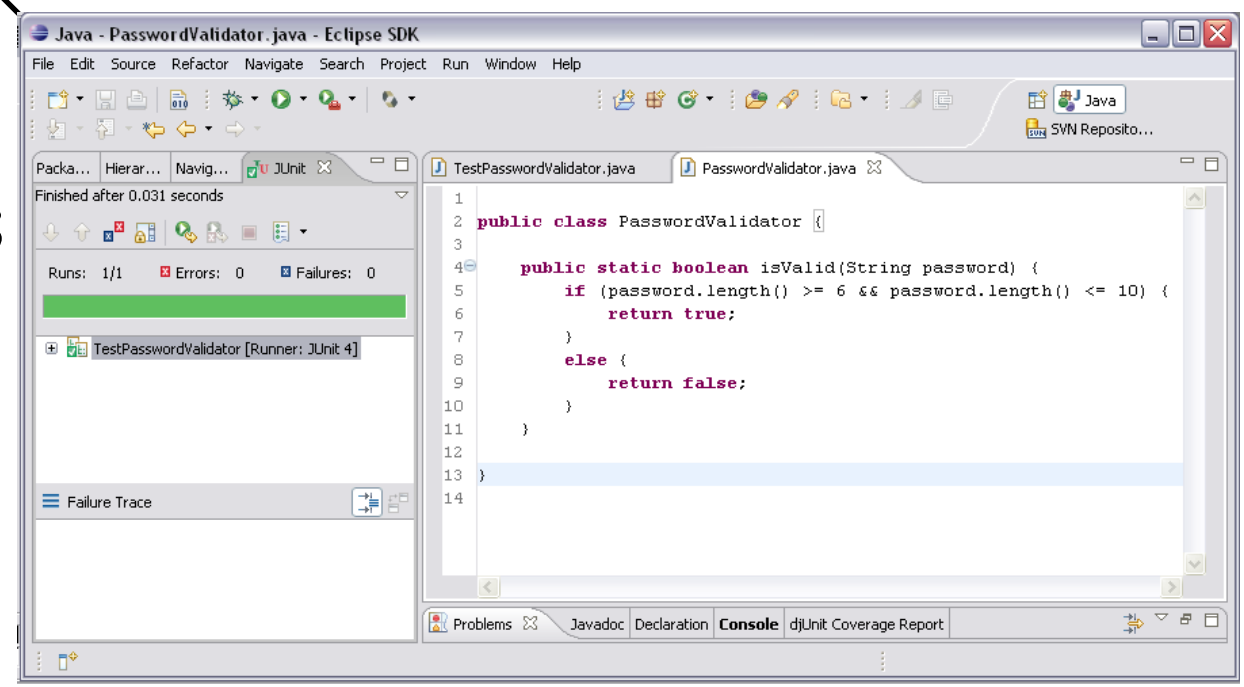
Design decision:
static method

What is Refactoring?

- Changing the *structure* of the code without changing its *behavior*
  - Example refactorings:
    - Rename
    - Extract method/extract interface
    - Inline
    - Pull up/Push down
- Some IDE's (e.g. Eclipse) include automated refactorings

TDD Example: Refactor the code

```
public class PasswordValidator {
   public static boolean isValid(String password) {
      if (password.length() >= 6 && password.length() <= 10) {
         return true;
      }
      else {
         return false;
      }
   }
}
```



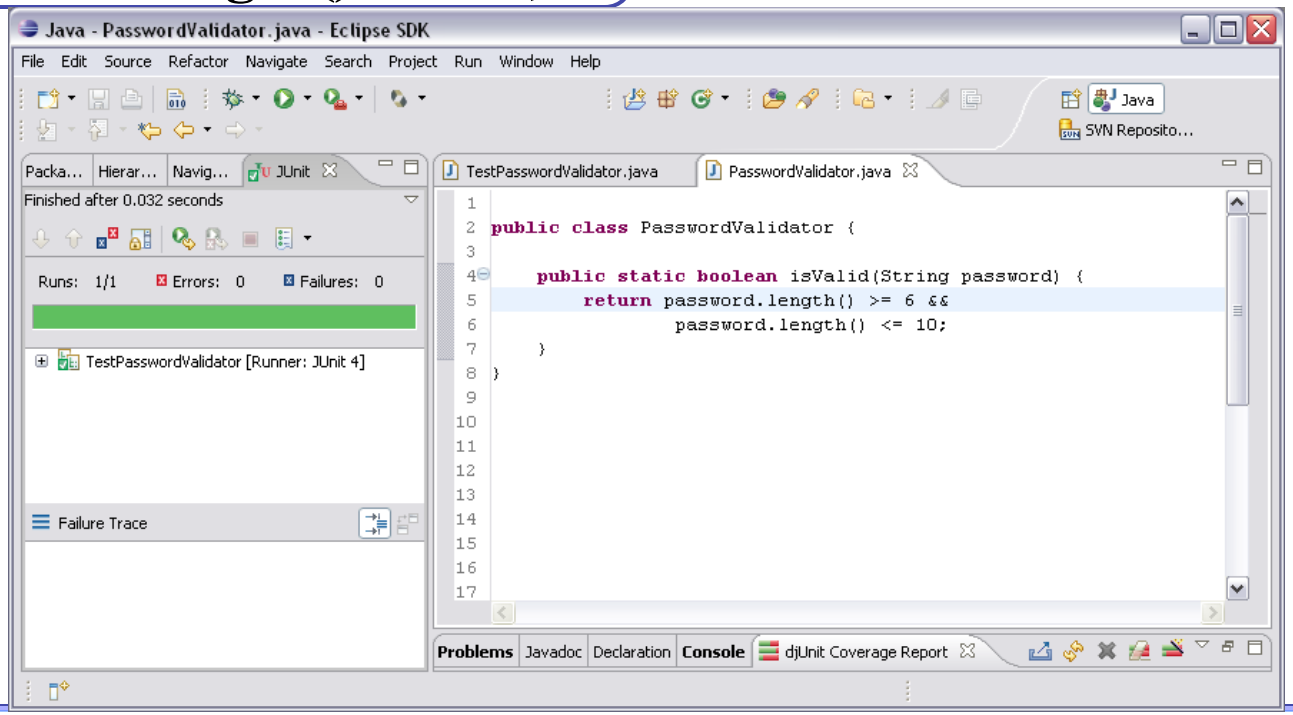
TDD Example: Refactor the code

```
public class PasswordValidator {
   public static boolean isValid(String password) {
      if (password.length() >= 6 && password.length() <= 10) {
         return true;
      }
      else {
         return false;
      }
   }
}
```

Can we simplify this?

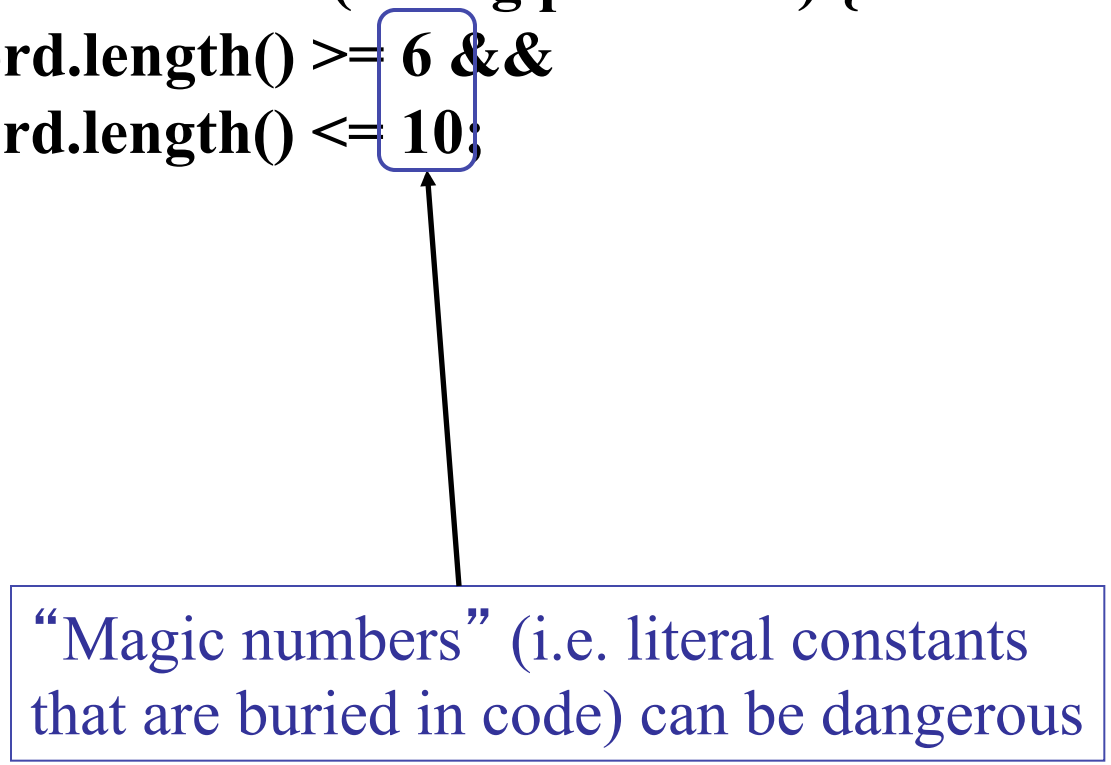
TDD Example: Refactoring #1

```
public class PasswordValidator {
   public static boolean isValid(String password) {
      return password.length() >= 6 &&
             password.length() <= 10;
   }
}
```



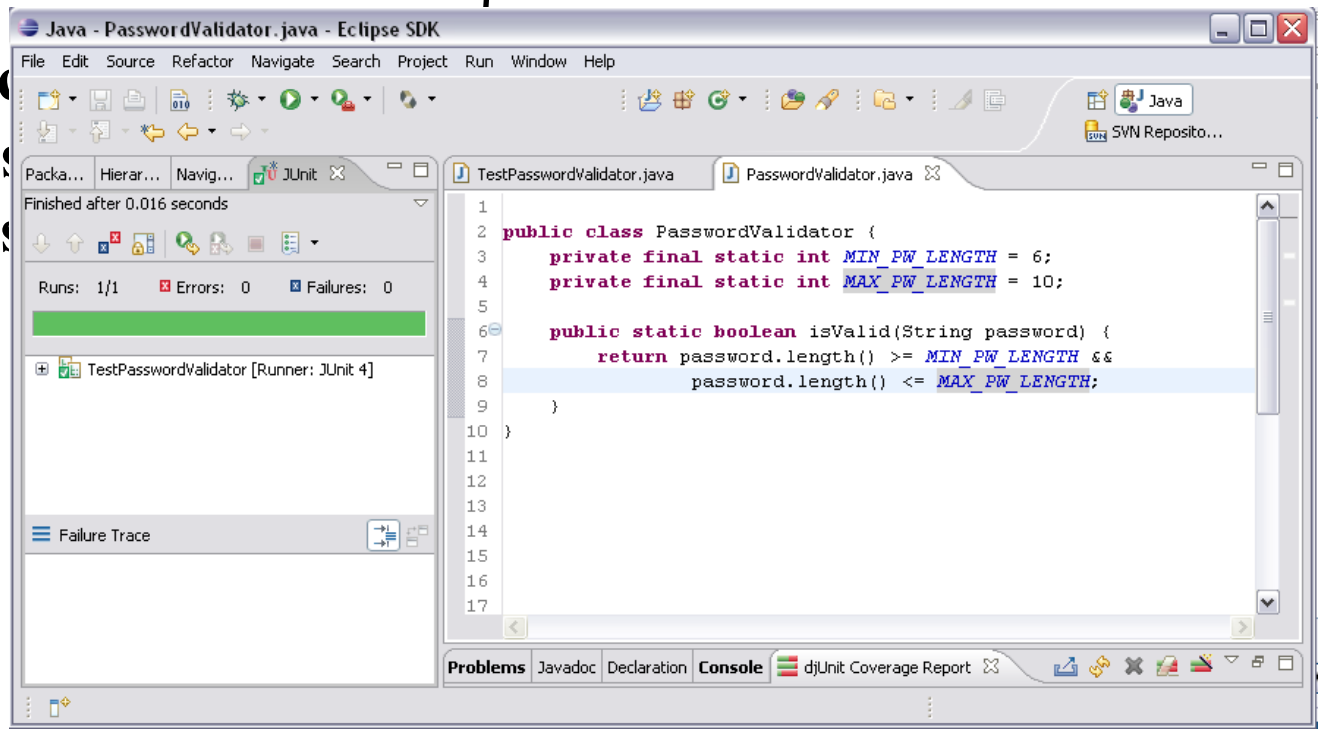
TDD Example: Refactoring #1

```
public class PasswordValidator {
   public static boolean isValid(String password) {
      return password.length() >= 6 &&
             password.length() <= 10;
   }
}
```



TDD Example: Refactoring #2

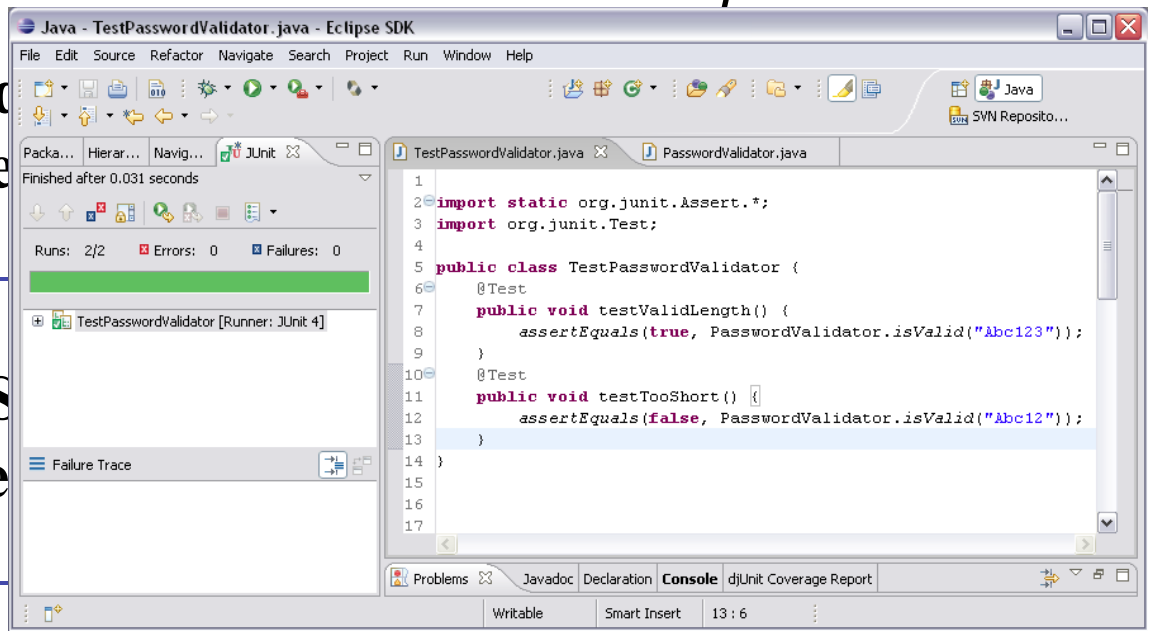
```
public class PasswordValidator {
    private final static int MIN_PW_LENGTH = 6;
    private final static int MAX_PW_LENGTH = 10;

    public static
      return pas
             pas
    }
}
```



TDD Example: Write another test

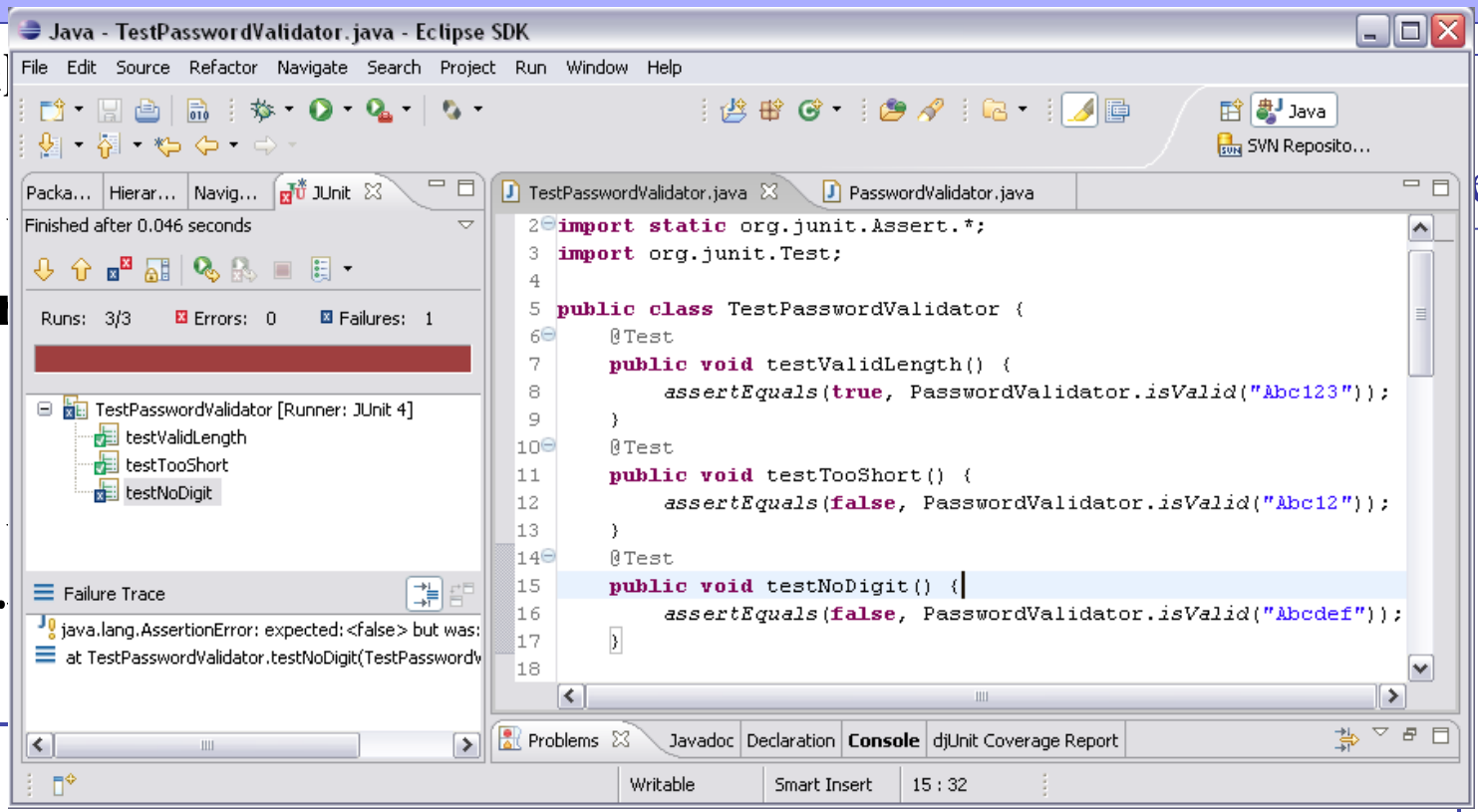
```
import static org.junit.Assert.*;
import org.junit.Test;

public class TestPasswordValidator {
  @Test
  public void testValic
     assertEquals(true
  }
  @Test
  public void testTooS
     assertEquals(false
  }
}
```

No design decisions; just unit testing



TDD Example: Write another test

```
public c
  @Test
  public
      asser
  }
  @Test
  public
      asser
  }
  @Test
  public void testNoDigit() {
      assertEquals(false, PasswordValidator.isValid("Abcdef"));
  }
}
```



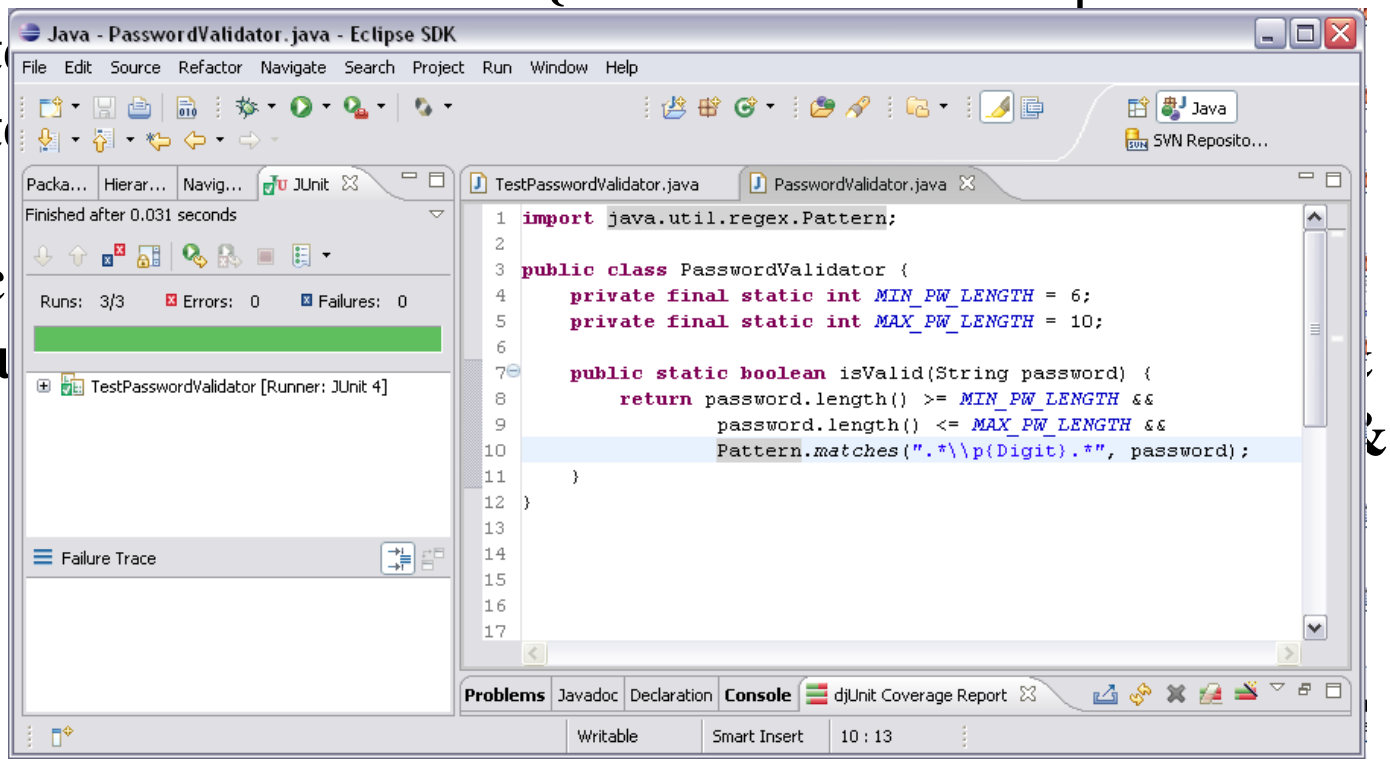
TDD Example: Make the test pass

```
public class PasswordValidator {
        private final static int MIN_PW_LENGTH = 6;
        private final static int MAX_PW_LENGTH = 10;

        public static boolean isValid(String password) {
            return password.length() >= MIN_PW_LENGTH &&
                    password.length() <= MAX_PW_LENGTH;
        }
    }
```

TDD Example: Make the test pass

```
import java.util.regex.Pattern;

public class PasswordValidator {
        privat
        privat

        public
            retu

        }
    }
```

Check for a digit



TDD Example: Refactor

```
import java.util.regex.Pattern;

public class PasswordValidator {
        private final static int MIN_PW_LENGTH = 6;
        private final static int MAX_PW_LENGTH = 10;

        public static boolean isValid(String password) {
            return password.length() >= MIN_PW_LENGTH &&
                   password.length() <= MAX_PW_LENGTH &&
                   Pattern.matches(".*\\p{Digit}.*", password);
        }
    }
```

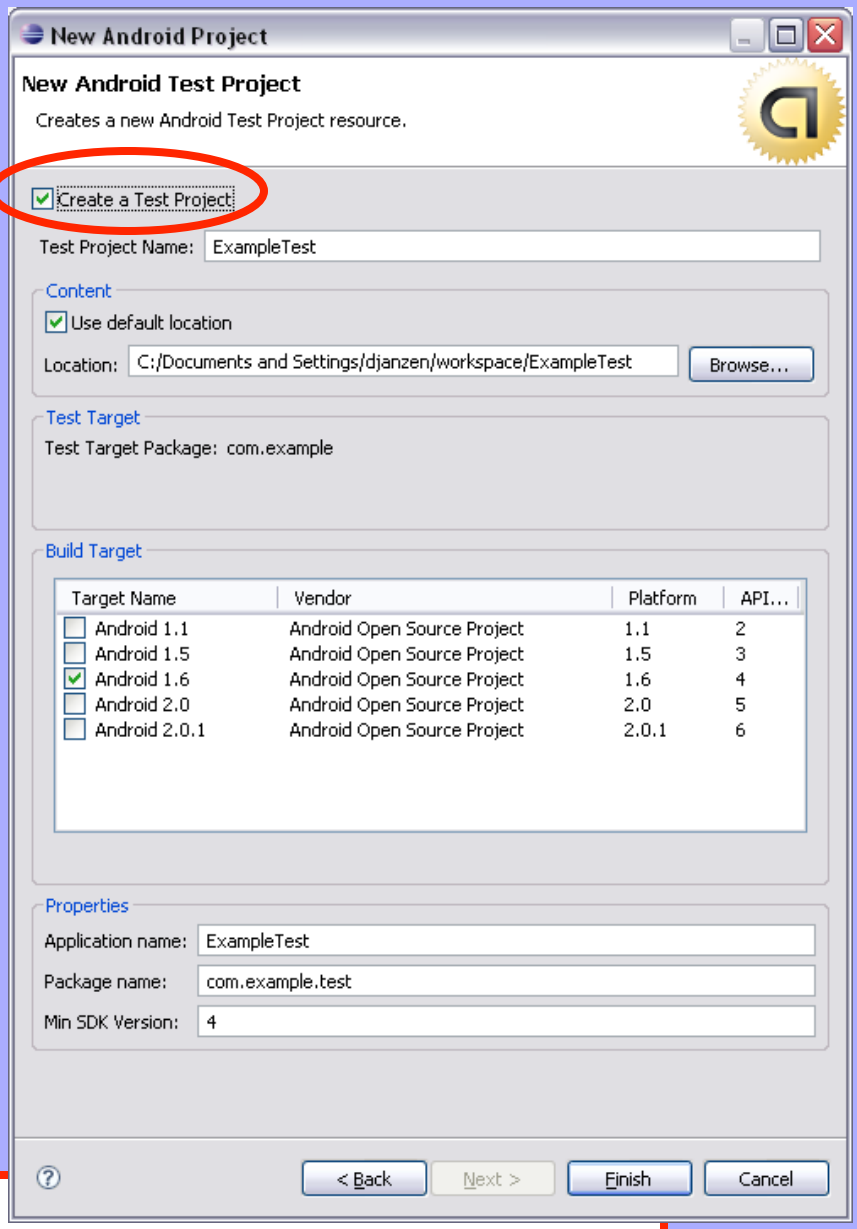
Extract methods for readability

TDD Example: Done for now

```
import java.util.regex.Pattern;
public class PasswordValidator {
  private final static int MIN_PW_LENGTH = 6;
  private final static int MAX_PW_LENGTH = 10;
  private static boolean isValidLength(String password) {
     return password.length() >= MIN_PW_LENGTH &&
               password.length() <= MAX_PW_LENGTH;
  }
  private static boolean containsDigit(String password) {
     return Pattern.matches(".*\\p{Digit}.*", password);
  }
  public static boolean isValid(String password) {
     return isValidLength(password) &&
               containsDigit(password);
  }
}
```

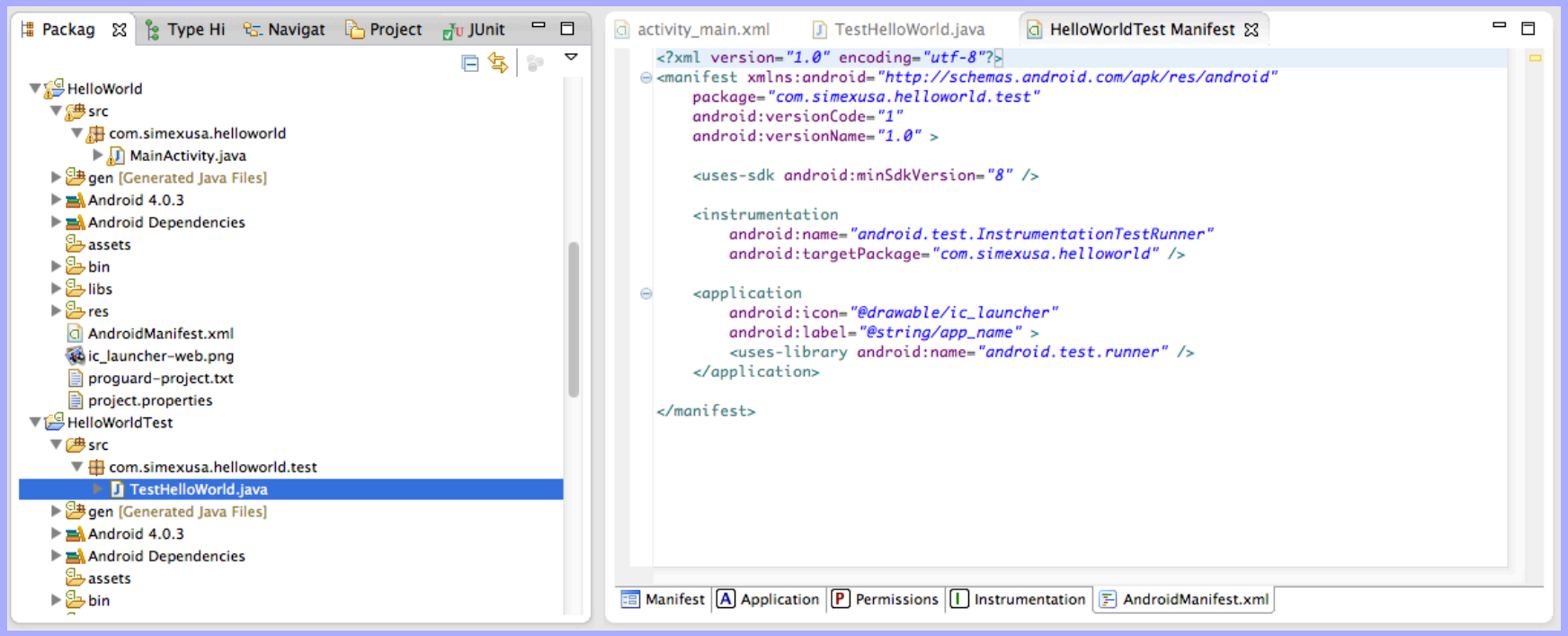
TDD in Android

- Android SDK integrates JUnit 3
  - not JUnit 4
- Many helper TestCase classes
- Recommended best practice to put tests in separate project but share folder
  - Eclipse "New Android Project" wizard will do this for you

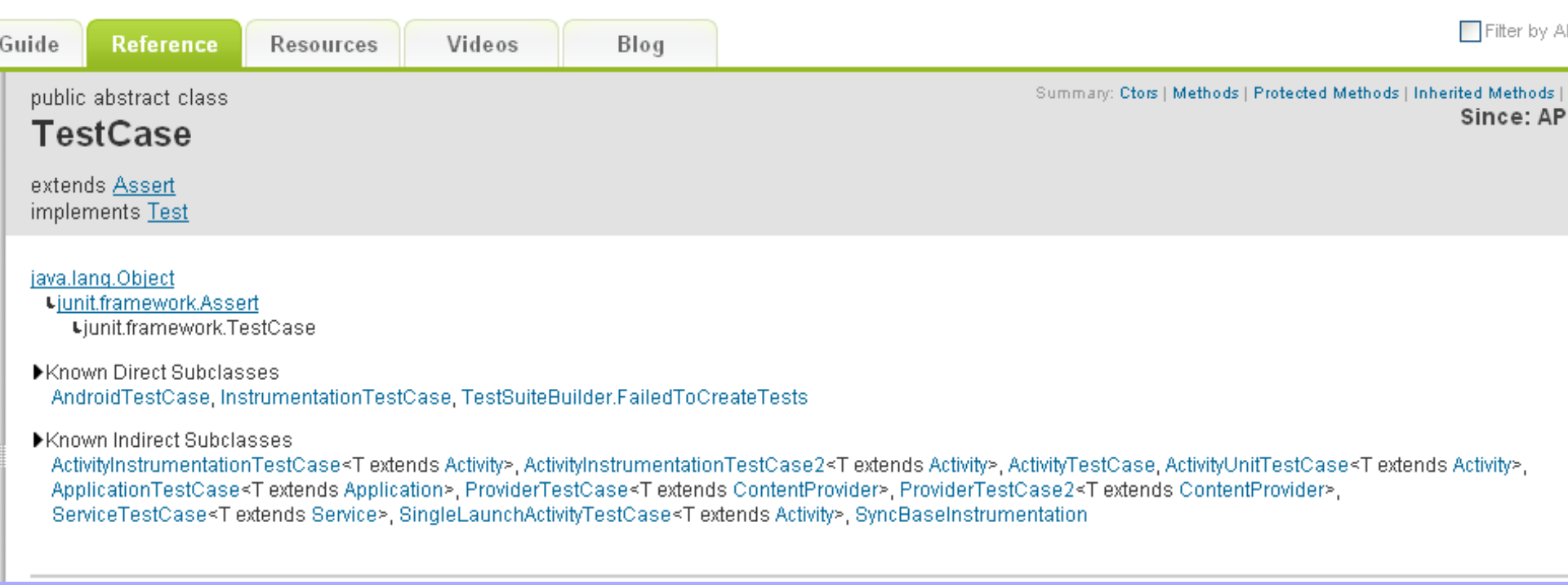


Beware if both src and test projects use same libraries

(see http://jimshowalter.blogspot.com/2009/10/developing-android-with-multiple.html)



Android TestCase Classes



Android TestCase Classes

- Basic JUnit tests
  - TestCase (run tests with assert methods)
- When you need an Activity Context
  - AndroidTestCase (see getContext())
- When you want to use a Mock Context
  - ApplicationTestCase (call setContext() before calling createApplication() which calls onCreate())

Android TestCase Classes

- When you want to test just one Activity
  - ActivityUnitTestCase (allows you to ask if the Activity has started another Activity or called finish() or requested a particular orientation)
- When you want to do a functional test on an Activity
  - ActivityInstrumentationTestCase2 (allows you to send key events to your Activity)

Android TestCase Classes

- When you want to test a Content Provider
  - ProviderTestCase2
- When you want to test a Service
  - ServiceTestCase
- When you want to stress test the UI
  - Monkey
    - http://d.android.com/guide/developing/tools/monkey.html

Android TestCase How-to

- Add instrumentation to AndroidManifest.xml

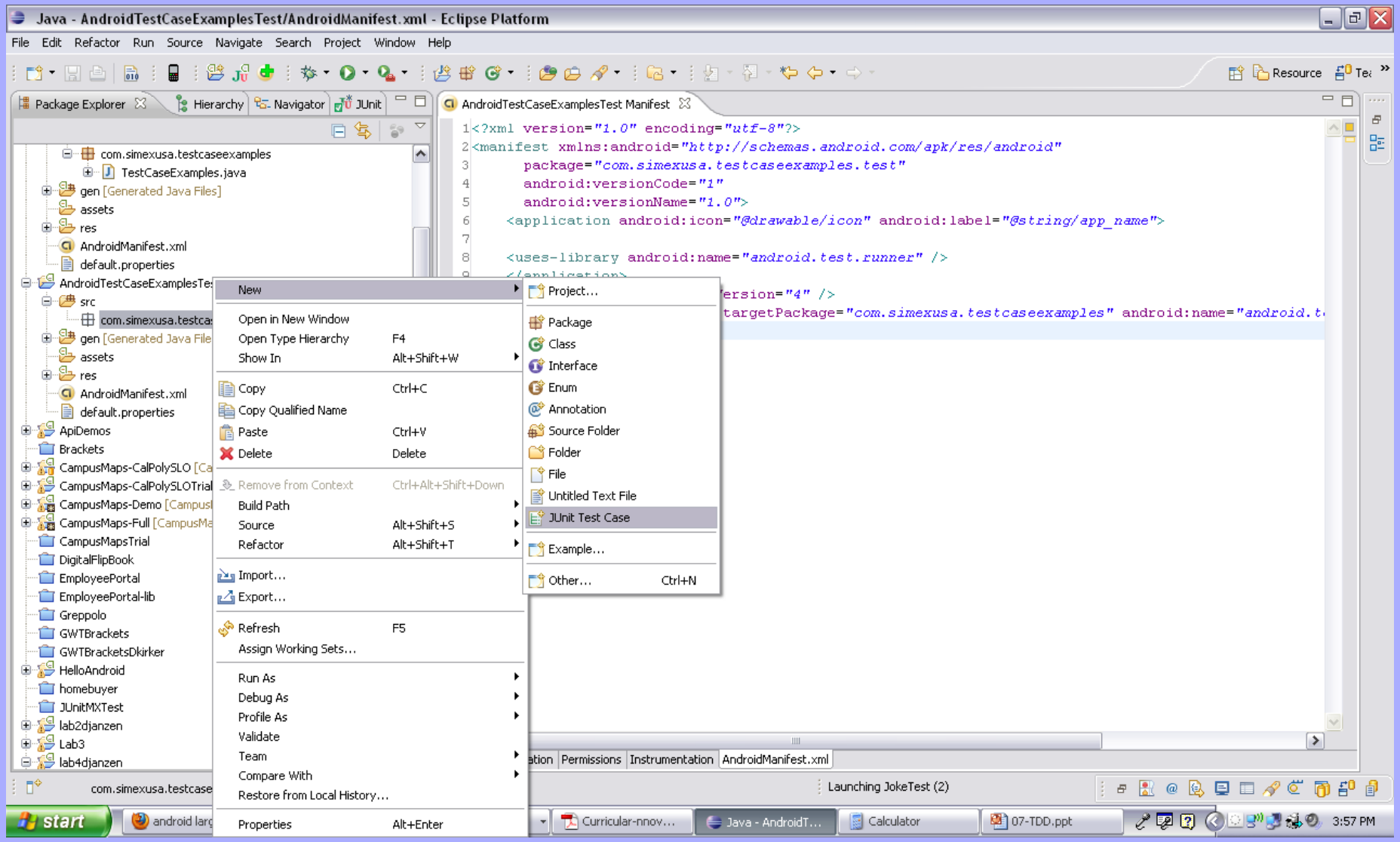
```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.simexusa.testcaseexamples"  android:versionCode="1"
    android:versionName="1.0">
   <application android:icon="@drawable/icon" android:label="@string/app_name"
     android:debuggable="true">
      <uses-library android:name="android.test.runner" />
      <activity android:name="SomeActivity android:label="@string/app_name">
         <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
         </intent-filter>
      </activity>
   </application>
   <uses-sdk android:minSdkVersion="3" />
   <instrumentation android:name="android.test.InstrumentationTestRunner"
                    android:targetPackage="com.simexusa.testcaseexamples"
                    android:label="Tests for my example."/>
</manifest>
```

Android TestCase How-to

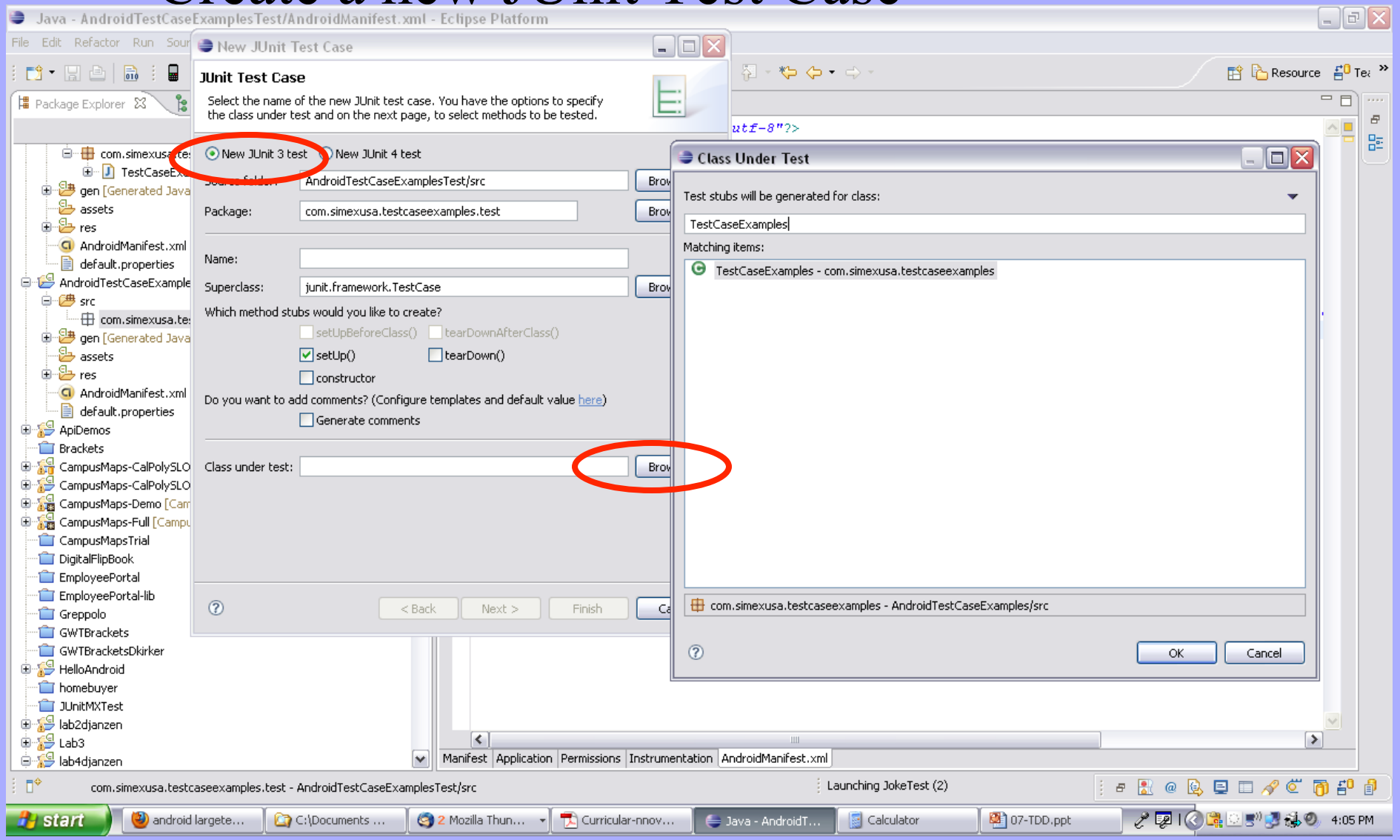
- Add instrumentation to AndroidManifest.xml
  - When creating a second project

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.simexusa.testcaseexamples.test"
    android:versionCode="1"
    android:versionName="1.0">
  <application android:icon="@drawable/icon" android:label="@string/app_name">

  <uses-library android:name="android.test.runner" />
  </application>
  <uses-sdk android:minSdkVersion="4" />
  <instrumentation android:targetPackage="com.simexusa.testcaseexamples"
    android:name="android.test.InstrumentationTestRunner" />
</manifest>
```

- Create a new JUnit Test Case



- Create a new JUnit Test Case

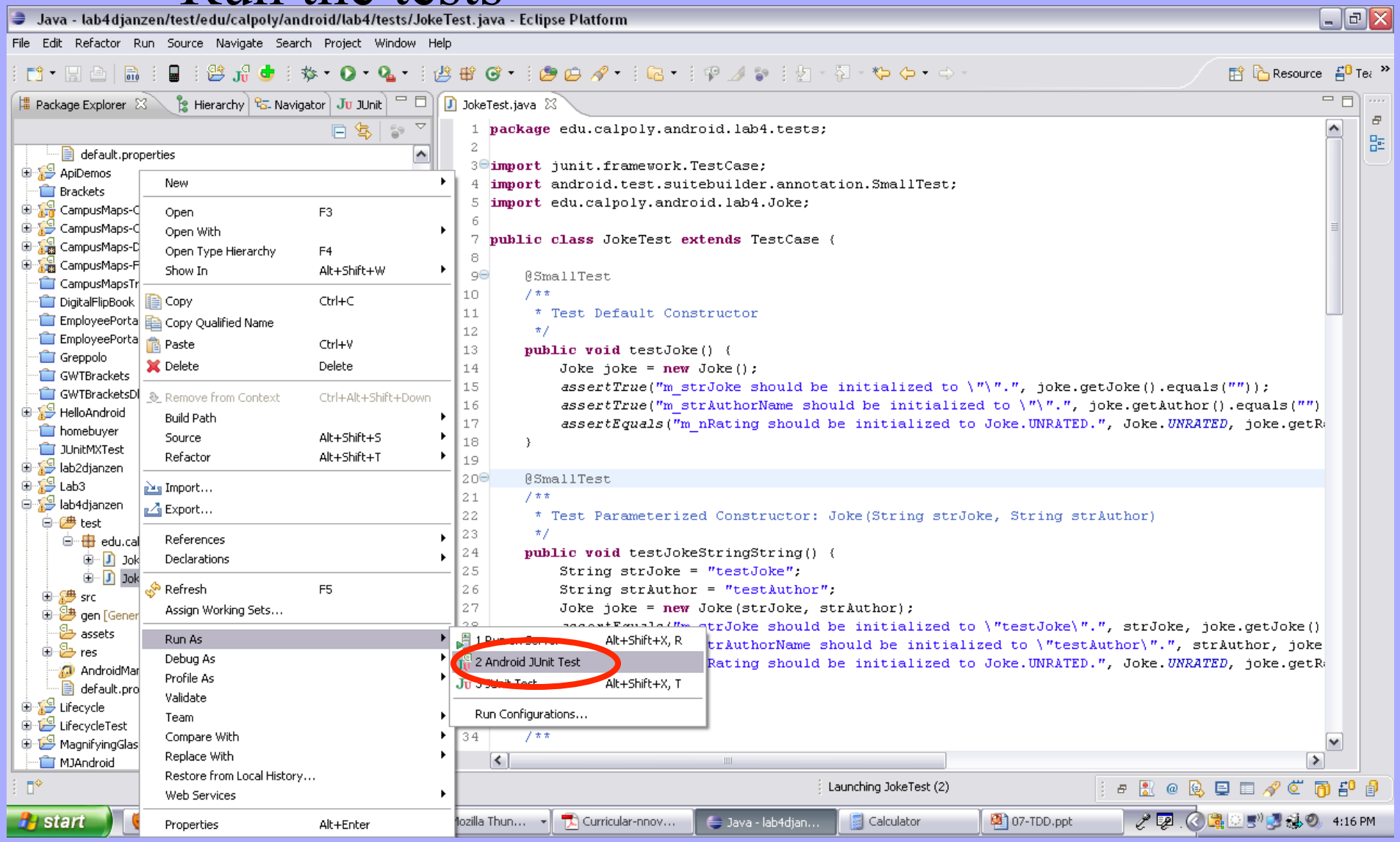


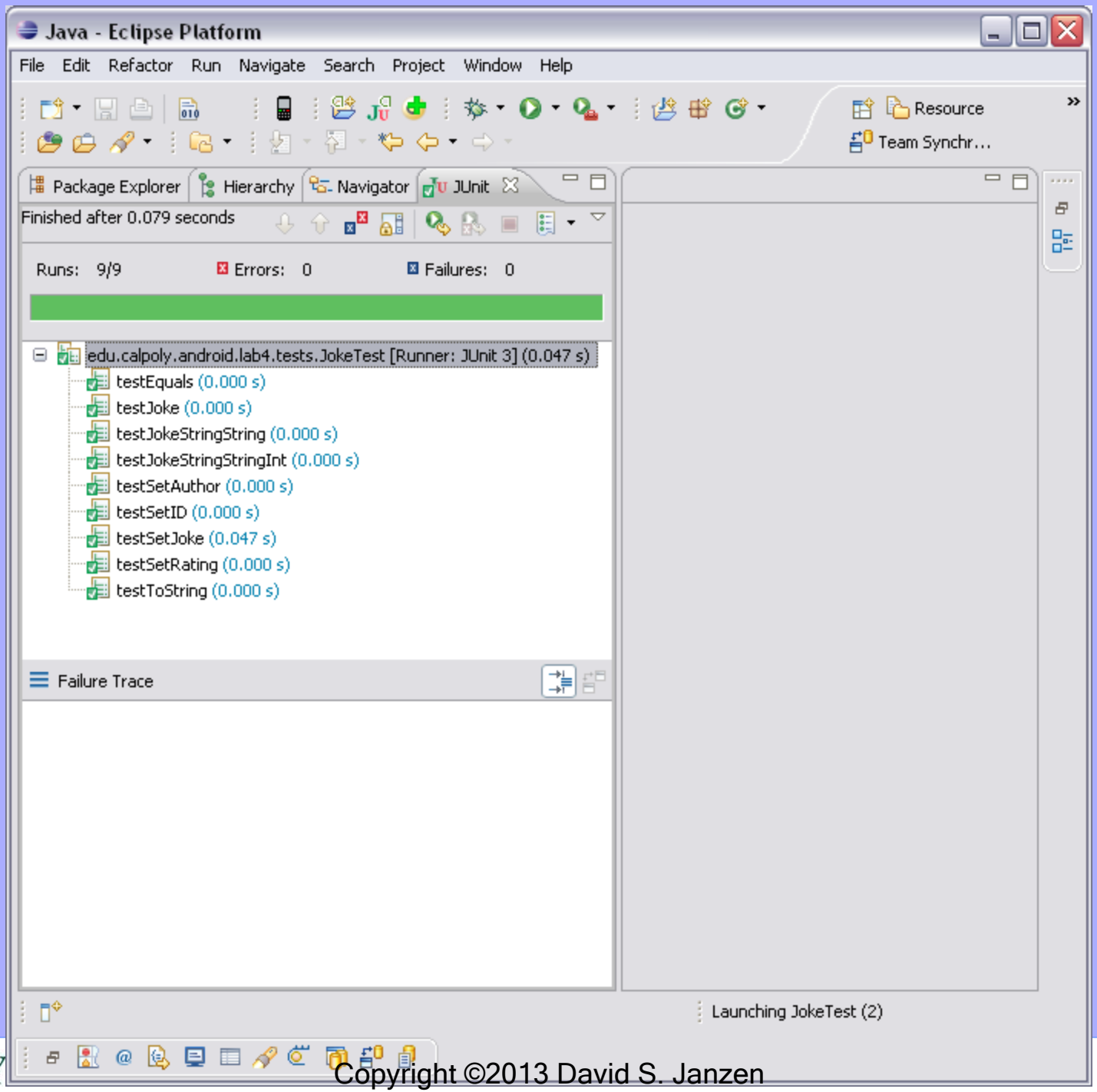
Testing POJO's

- Plain Old Java Objects
  - (i.e. independent of frameworks like Android or J2EE)

```
import junit.framework.TestCase;
import edu.calpoly.android.lab4.Joke;

public class JokeTest extends TestCase {

 public void testJoke() {
    Joke joke = new Joke();
    assertTrue("m_strJoke should be initialized to \"\".", joke.getJoke().equals(""));
    assertTrue("m_strAuthorName should be initialized to \"\".",
                joke.getAuthor().equals(""));
    assertEquals("m_nRating should be initialized to Joke.UNRATED.",
                Joke.UNRATED, joke.getRating());
 }
}
```

- Run the tests





JUnit 3 How-to

- Import the JUnit framework

```
import junit.framework.*;
```

- Create a subclass of TestCase

```
public class TestBank extends TestCase {
```

- Write methods in the form testXXX()
- Use assertXXX() methods

```
public void testCreateBank() {
    Bank b = new Bank();
    assertNotNull(b);
}
```

- Compile test and functional code; Run a TestRunner to execute tests; Keep the bar green!

Fixtures

- Notice redundancy in test methods

```
import junit.framework.TestCase;
public class TestBank extends TestCase {
    public void testCreateBank() {
        Bank b = new Bank();
        assertNotNull(b);
    }
    public void testCreateBankEmpty() {
        Bank b = new Bank();
        assertEquals(b.getNumAccounts(),0);
    }
}
```

- Common test setup can be placed in a method named setUp() which is *run before each test*

setUp()

```
import junit.framework.*;
public class TestBank extends TestCase {
    private Bank b;
    public void setUp() {
        b = new Bank();
    }
    public void testCreateBank() {
        assertNotNull(b);
    }
    public void testCreateBankEmpty() {
        assertEquals(b.getNumAccounts(),0);
    }
    public void testAddAccount() {
        Account a = new Account("John Doe",123456,0.0);
        b.addAccount(a);
        assertEquals(b.getNumAccounts(),1);
    }
}
```

setUp() is run before *each* test

tearDown()

- tearDown() is run after each test
  - Used for cleaning up resources such as files, network, or database connections

```
import junit.framework.TestCase;
public class TestBank extends TestCase {
    private Bank b;
    public void setUp() {
        b = new Bank();
    }
    public void tearDown() {
        b = null;
    }
    …
}
```

tearDown() is run after *each* test

Grouping Tests with @xTest

- Some tests run fast, others don't
  - You can separate them with @SmallTest, @MediumTest, @LargeTest

```
public class JokeTest extends TestCase {

@SmallTest
/**
 * Test Default Constructor
 */
  public void testJoke() {
     Joke joke = new Joke();
     assertTrue("m_strJoke should be initialized to \"\".", joke.getJoke().equals(""));
     assertTrue("m_strAuthorName should be initialized to \"\".",
                    joke.getAuthor().equals(""));
     assertEquals("m_nRating should be initialized to Joke.UNRATED.",
                    Joke.UNRATED, joke.getRating());
}
```

Running Tests with @xTest

- Run the tests with adb from the command line
  - http://developer.android.com/reference/android/test/InstrumentationTestRunner.html

```
C:\adb shell am instrument -w -e size small edu.calpoly.android.lab4/
    android.test.InstrumentationTestRunner


edu.calpoly.android.lab4.tests.dflt.JokeCursorAdapterTest:....
edu.calpoly.android.lab4.tests.dflt.JokeTest:.........
Test results for InstrumentationTestRunner=.............
Time: 1.975


OK (13 tests)
```

Testing Campus Maps

```
package com.simexusa.campusmaps_full;

import com.simexusa.campusmaps_full.CampusMap;
import com.simexusa.campusmaps_full.TranslatorUtility;
import junit.framework.TestCase;

public class TestTranslatorUtility extends TestCase {

  protected void setUp() throws Exception {
     super.setUp();
  }
  public void testTranslateLatToY() {
     double b1lat = 35.302518;
     double b2lat = 35.299365;
     int b1py = 445;
     int b2py = 840;
     double latitude = 35.299812;
     assertEquals(784,TranslatorUtility.latToCoordinate(latitude,b1lat,b2lat,b1py,b2py))
}
```

Testing Campus Maps

```
package com.simexusa.campusmaps_full;

import com.simexusa.campusmaps_full.CampusMap;
import com.simexusa.campusmaps_full.TranslatorUtility;
import junit.framework.TestCase;

public class TestTranslatorUtility extends TestCase {

  protected void setUp() throws Exception {
     super.setUp();
  }
  public void testTranslateLatToY() {
     double b1lat = 35.302518;
     double b2lat = 35.299365;
     int b1py = 445;
     int b2py = 840;
     double latitude = 35.299812;
     assertEquals(784,TranslatorUtility.latToCoordinate(latitude,b1lat,b2lat,b1py,b2py))
}
```

Test complicated methods

Testing Campus Maps

```
public void testSplit2() {
   String s = "go+180";
   String [] results = s.split("\\+");
   assertEquals(results[0],"go");
   assertEquals(results[1],"180");
}

public void testParser() {
   CampusMap [] maps = TranslatorUtility.parseMapData(
      "Bethel College|http://www.bethelks.edu/map/bcmap.png|" +
      "39.298664|39.296903|-76.593761|-76.590527|383|614|171|352\n");
   assertEquals(maps[0].title,"Bethel College");
}

public void testGetMaps() {
   CampusMap[] myCampusMaps = new CampusMap[5];
   TranslatorUtility.retrieveMapData("http://simexusa.com/cm/fav5defaultmapdata.txt",
                                     myCampusMaps);
   assertEquals(myCampusMaps[0].title,"Cal Poly - SLO");
}
```

Explore library API's

Verify it works like I expect

Functional tests
This one gets data from the web

Functional Testing

- ActivityInstrumentationTestCase2
  - Allows us to create/start an Activity
  - Get Views from the Activity (e.g. Buttons)
  - Run things on the UI thread (e.g. click Buttons)
  - Perform asserts in JUnit
- Other options
  - http://code.google.com/p/autoandroid/
  - Formerly Positron
    - Android + Selenium = Positron

```
public class FunctionalTest extends
                ActivityInstrumentationTestCase2<AdvancedJokeList> {
  public FunctionalTest() { super("edu.calpoly.android.lab2",
                                                  AdvancedJokeList.class);}
  protected void setUp() throws Exception {   super.setUp();  }
  public void testAddJoke() {
    ArrayList<Joke> m_arrJokeList = null;
    m_arrJokeList =  this.retrieveHiddenMember("m_arrJokeList",
                                          m_arrJokeList,getActivity());
    assertEquals("Should be 3 default jokes",m_arrJokeList.size(),3);
    getActivity().runOnUiThread(new Runnable() {
      public void run() {
        AdvancedJokeList theActivity = (AdvancedJokeList)getActivity();
        EditText et = (EditText)theActivity.
                findViewById(edu.calpoly.android.lab2.R.id.newJokeEditText);
        Button bt = (Button)theActivity.
                findViewById(edu.calpoly.android.lab2.R.id.addJokeButton);
        et.setText("This is a test joke");
        bt.performClick();
      }});
    getInstrumentation().waitForIdleSync(); // wait for the request to go through
    assertEquals("Should be 4 jokes now",m_arrJokeList.size(),4);
    assertEquals("Ensure the joke we added is really there",
            m_arrJokeList.get(3).getJoke(),"This is a test joke");
}
```

```
@SuppressWarnings("unchecked")
public <T> T retrieveHiddenMember(String memberName, T type, Object
sourceObj) {
    Field field = null;
    T returnVal = null;
    try {//Test for proper existence
        field = sourceObj.getClass().getDeclaredField(memberName);
    } catch (NoSuchFieldException e) {
        fail("The field \"" + memberName +
    "\" was renamed or removed. Do not rename or remove this member variable.");
    }
    field.setAccessible(true);

    try {//Test for proper type
        returnVal = (T)field.get(sourceObj);
    } catch (ClassCastException exc) {
        fail("The field \"" + memberName +
            "\" had its type changed. Do not change the type on this member variable.");
    }
```

```
   // Boiler Plate Exception Checking. If any of these Exceptions are
   // thrown it was because this method was called improperly.
  catch (IllegalArgumentException e) {
      fail ("This is an Error caused by the UnitTest!\n Improper user of
retrieveHiddenMember(...) -- IllegalArgumentException:\n Passed in the wrong
object to Field.get(...)");
   } catch (IllegalAccessException e) {
      fail ("This is an Error caused by the UnitTest!\n Improper user of
retrieveHiddenMember(...) -- IllegalAccessException:\n Field.setAccessible(true)
should be called.");
   }
   return returnVal;
}
```

Monkey

- Random stress testing
  - From http://d.android.com/guide/developing/tools/monkey.html
  - When the Monkey runs, it generates events and sends them to the system. It also *watches* the system under test and looks for three conditions, which it treats specially:
    - If you have constrained the Monkey to run in one or more specific packages, it watches for attempts to navigate to any other packages, and blocks them.
    - If your application crashes or receives any sort of unhandled exception, the Monkey will stop and report the error.
    - If your application generates an *application not responding* error, the Monkey will stop and report the error.

```
adb shell monkey -p edu.calpoly.lab2 -v 500
```

TDD and Android Resources

- Android SDK documentation
  - http://developer.android.com/reference/junit/framework/TestCase.html
- Tutorial:
  - http://dtmilano.blogspot.com/2008/01/test-driven-development-and-gui-testing.html
- Blogs:
  - http://dtmilano.blogspot.com/search/label/test%20driven%20development
  - http://jimshowalter.blogspot.com/2009/10/developing-android-with-multiple.html

Test-Driven Development

- **Short introduction[1]**
  - Test-driven development (TDD) is the craft of producing automated tests for production code, and using that process to *drive design* and *programming*. For every tiny bit of functionality in the production code, you first develop a test that specifies and validates what the code will do. You then produce exactly as much code as will enable that test to pass. Then you refactor (simplify and clarify) both the production code and the test code.

1. http://www.agilealliance.org/programs/roadmaps/Roadmap/tdd/tdd_index.htm

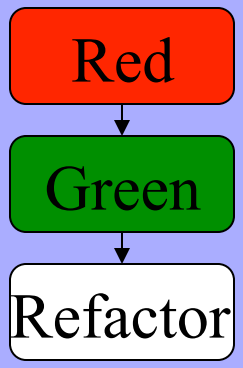
Test-Driven Development

- **Definition[1]**
  - Test-driven Development (TDD) is a programming practice that instructs developers to write new code only if an automated test has failed, and to eliminate duplication. The goal of TDD is "clean code that works."

1. "JUnit in Action" Massol and Husted.

- **The TDD Two-Step[2]**
  - Write a failing automatic test before writing new code
  - Eliminate duplication

- **The TDD Cycle[2]**
  - Write a test
  - Make it run
  - Make it right



2. "Test-Driven Development By Example" Beck.

Some Types of Testing

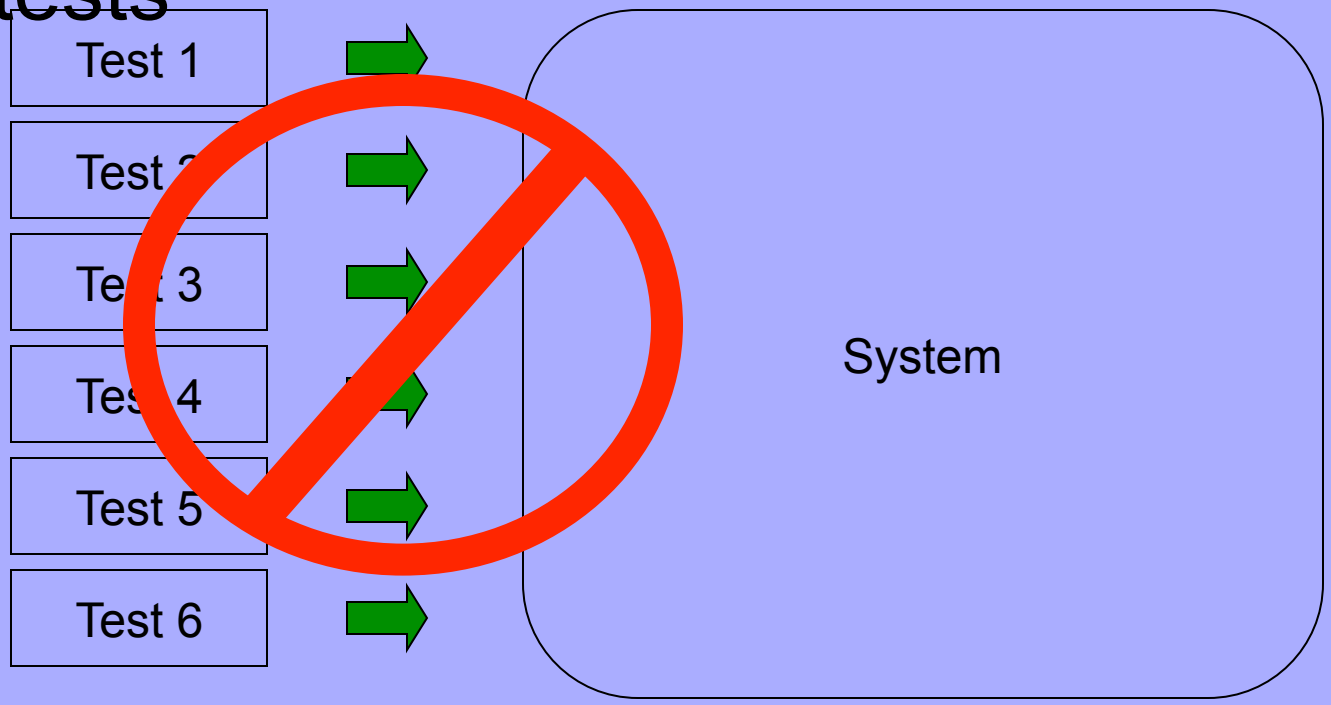
- Unit Testing ← TDD focuses here
  - Testing individual units (typically methods)
  - White/Clear-box testing performed by original programmer
- Integration and Functional Testing ← and may help here
  - Testing interactions of units and testing use cases
- Regression Testing ← and here
  - Testing previously tested components after changes
- Stress/Load/Performance Testing
  - How many transactions/users/events/… can the system handle?
- Acceptance Testing
  - Does the system do what the customer wants?

TDD Misconceptions

- There are many misconceptions about TDD
- They probably stem from the fact that the first word in TDD is “Test”
- TDD is **not about testing**,
  TDD is about **design**
  - Automated tests are just a nice side effect

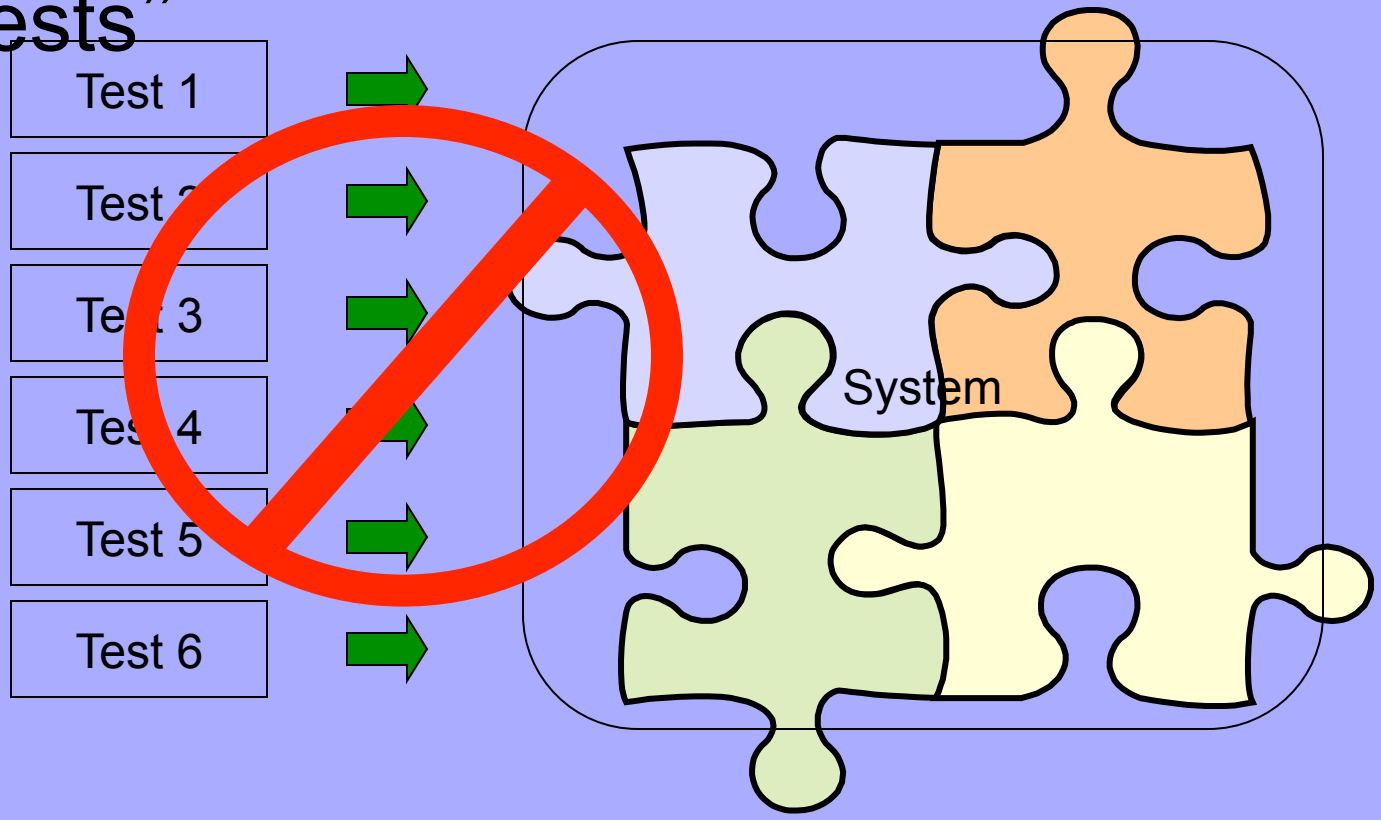
TDD Misconception #1

- TDD does not mean "write all the tests, then build a system that passes the tests"



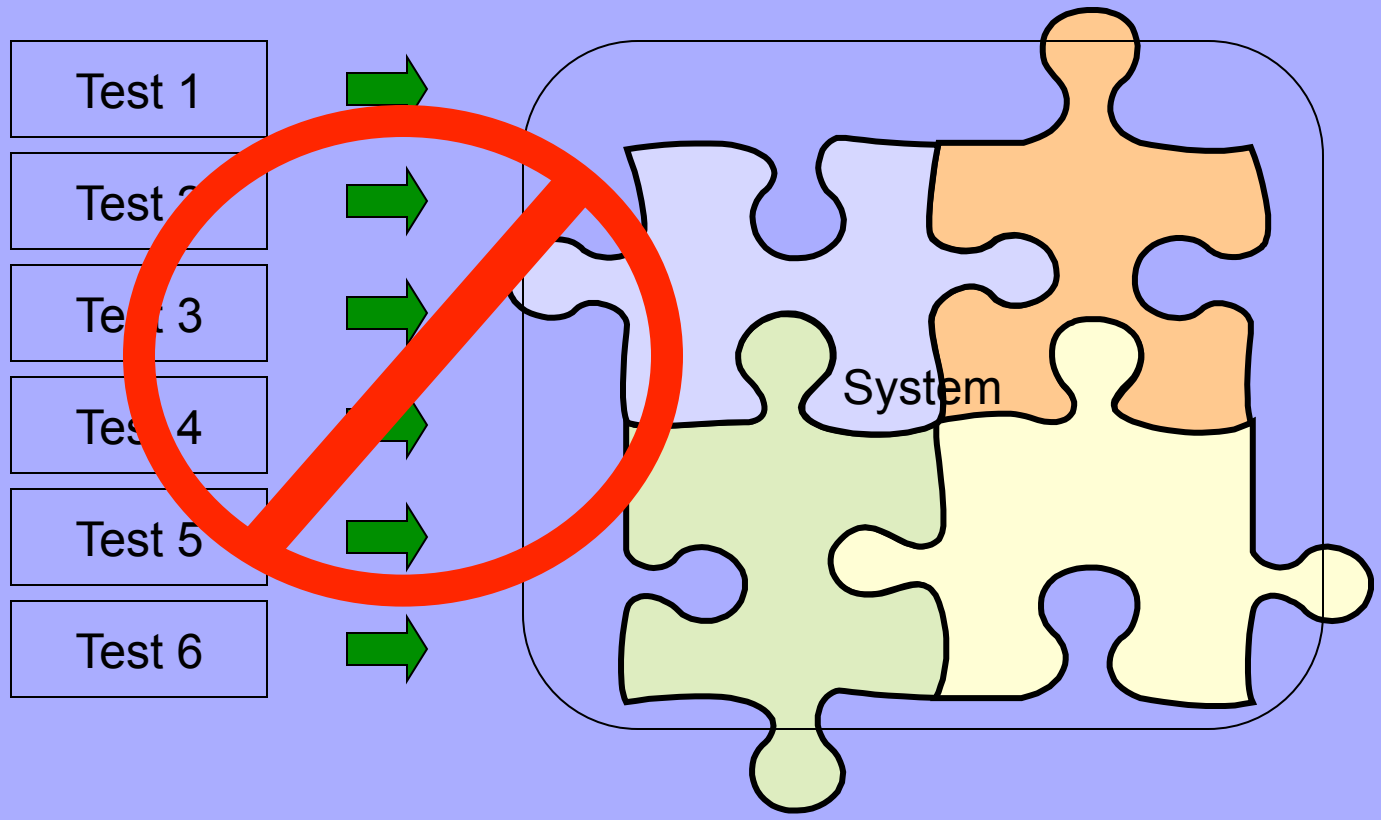
TDD Misconception #2

- TDD does not mean "write some of the tests, then build a system that passes the tests"



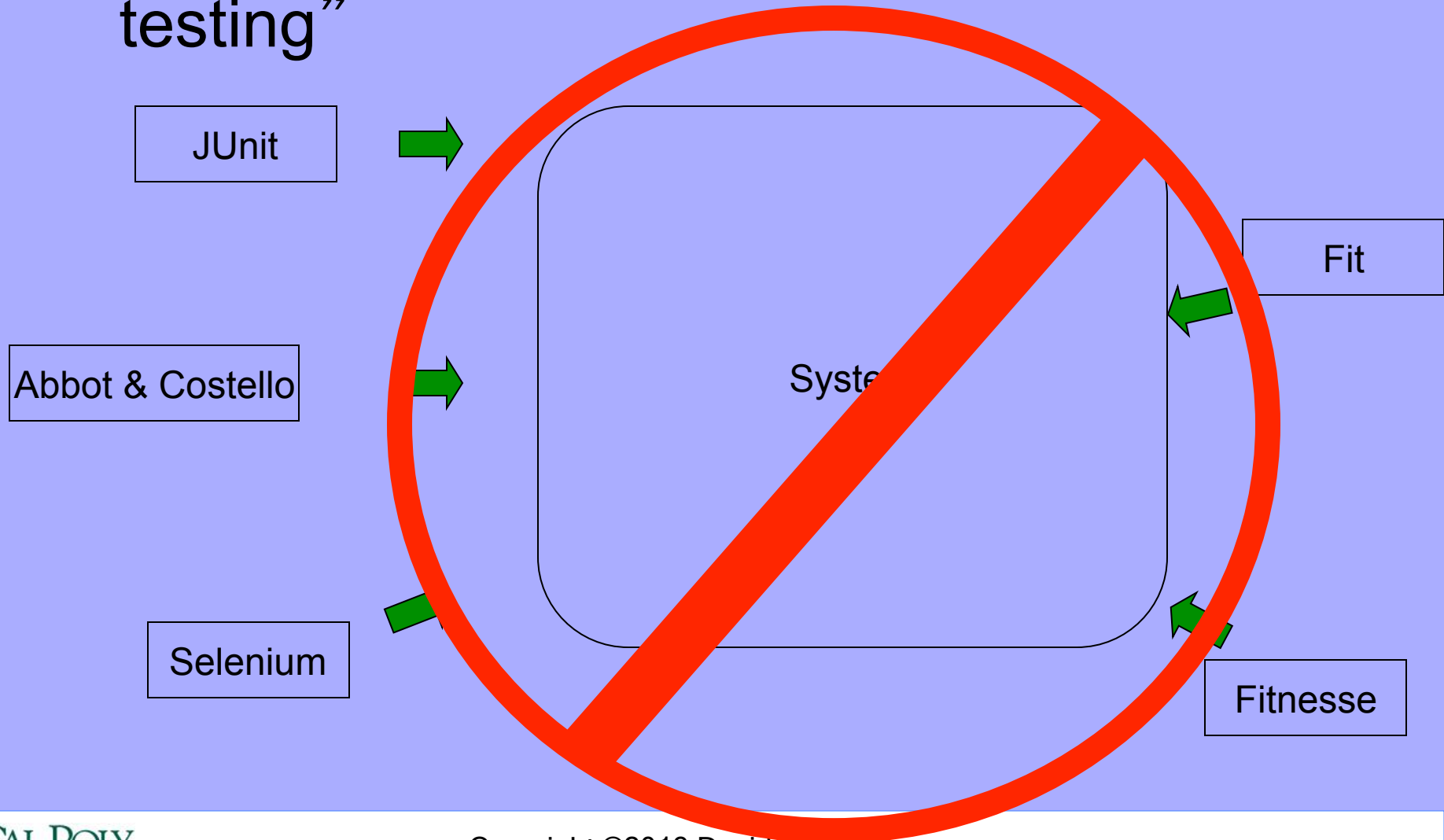
TDD Misconception #3

- TDD does not mean “write some of the code, then test it before going on”



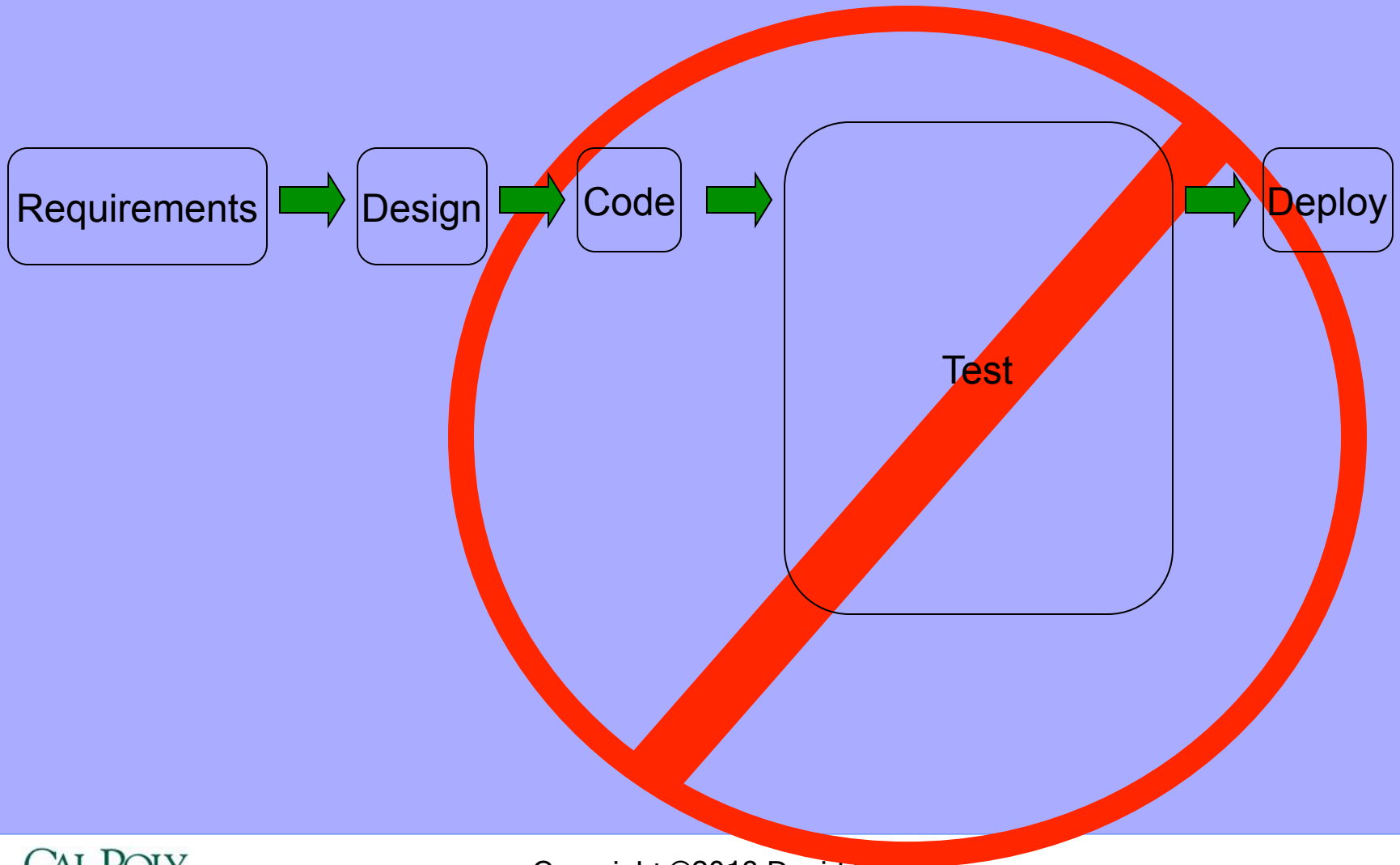
TDD Misconception #4

- TDD does not mean “do automated testing”



TDD Misconception #5

- TDD does not mean “do lots of testing”



TDD Misconception #6

- TDD does not mean "the TDD process"
- TDD is a *practice*

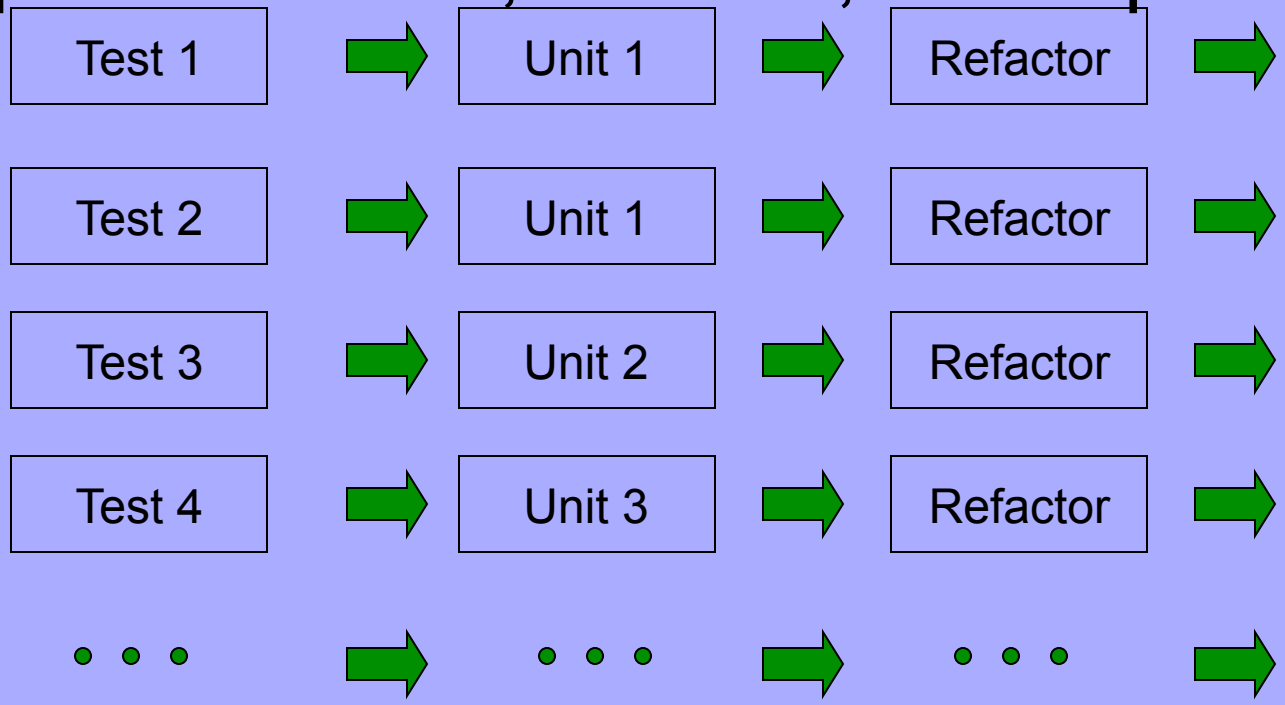
  (like pair programming, code reviews, and stand-up meetings)

  not a *process*

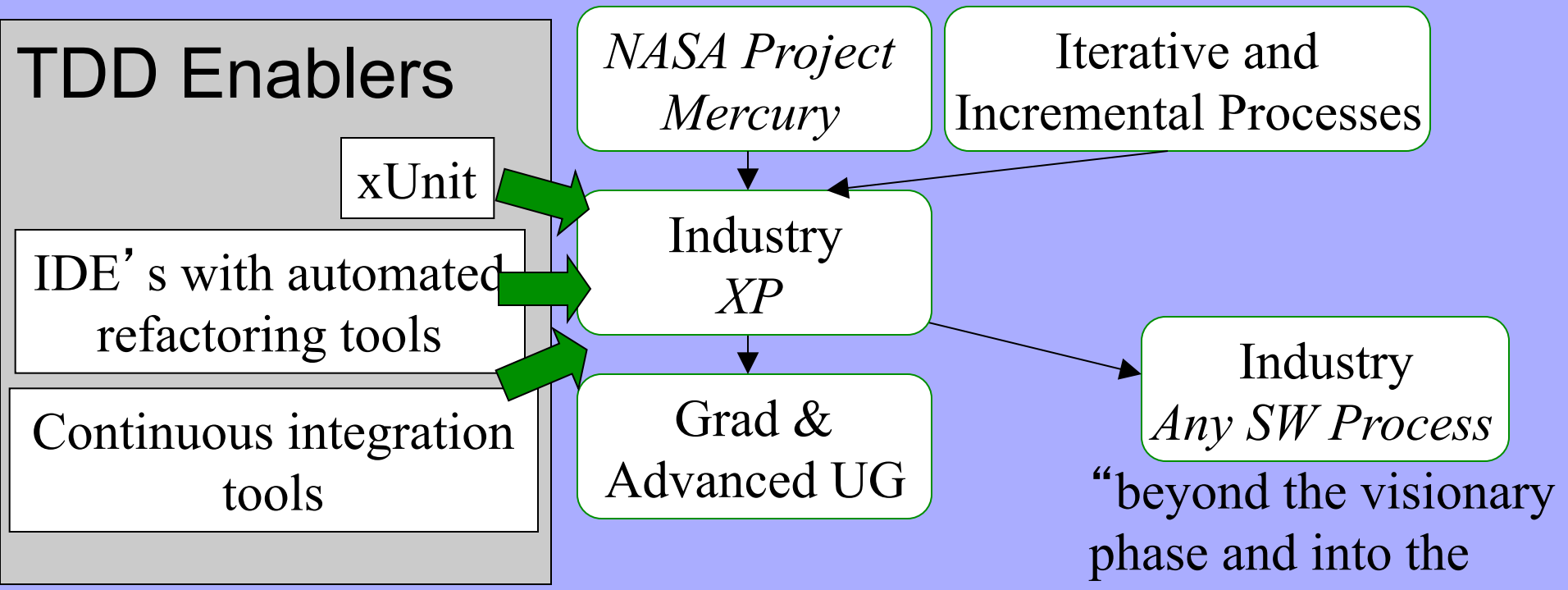
  (like waterfall, Scrum, XP, TSP)

TDD Clarified

- TDD means “write one test, write code to pass that test, refactor, and repeat”

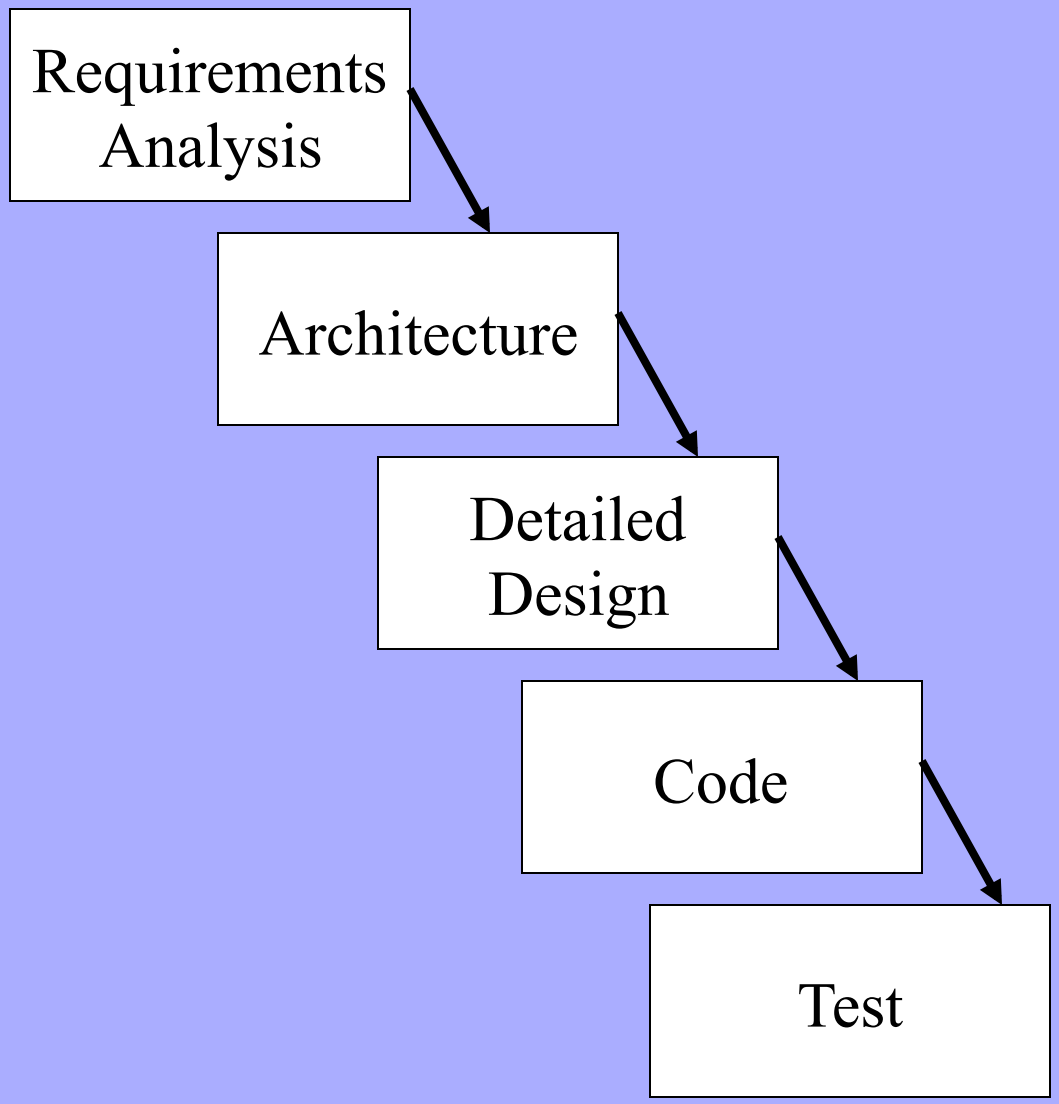


TDD History



"beyond the visionary phase and into the early mainstream"
IEEE Software 2007

Traditional Waterfall Process



eXtreme Programming (XP) Overview

- Kent Beck and Ward Cunningham
- First agile method to garner much attention
- Small teams
  - 20 max, ~10 optimal
- Roles
  - Customer
  - Coach
  - Programmers
  - Tester
  - Tracker
- Four Values
- Twelve Core Practices
  - Many are well-known

Extreme Best Practices

- If *customer feedback* is good
  - then have a customer always on-site
- If *code reviews* are good
  - then always perform code reviews through pair programming
- If *early integration* is good
  - then continuously integrate
- If *unit-testing* is good
  - then require unit-tests and do them first
  - plus make them automated so they are run often

XP Core Practices (v1)

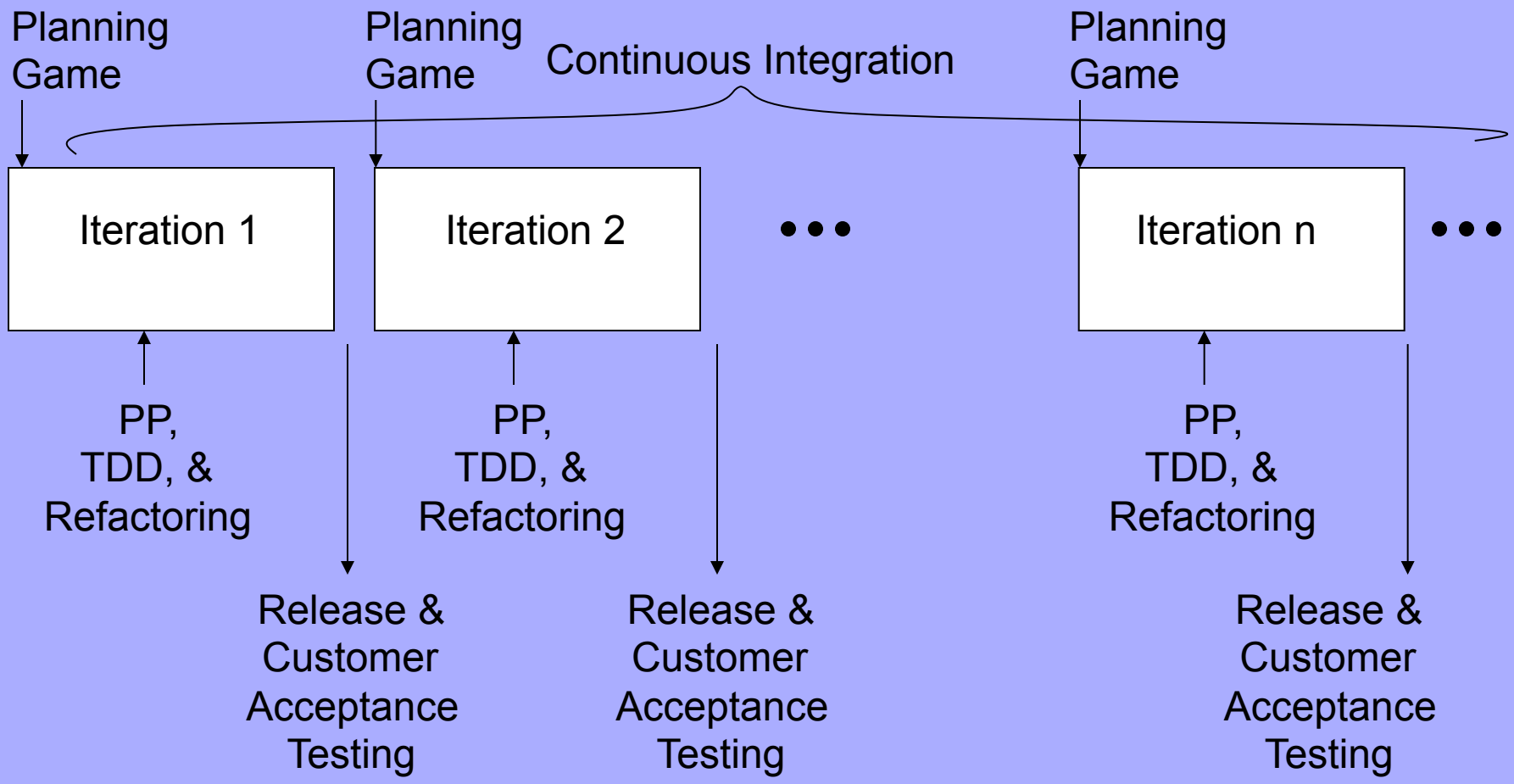
- Test-Driven Development
- Planning Game
- Whole Team/On-Site Customer
- Pair Programming
- Continuous Integration
- Design Improvement (Refactoring)
- Small Releases
- Simple Design
- Metaphor
- Collective Ownership
- Coding Standard
- Sustainable Pace

XP Values[1]

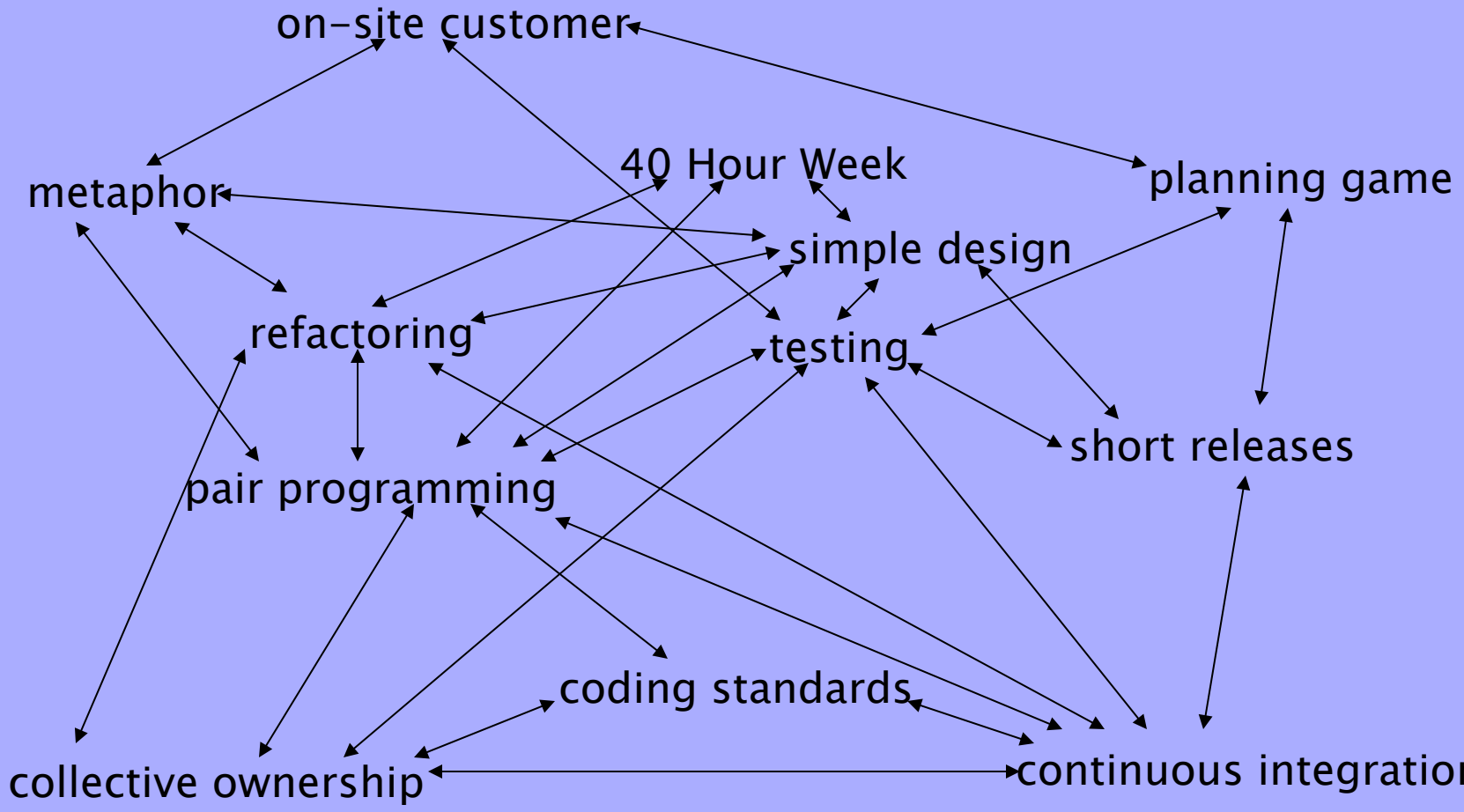
- Communication
  - Unit-testing, pair programming, task estimation
- Simplicity
  - YAGNI (You Aren't Going to Need It)
- Feedback
  - Tests tell us what works
  - User stories get estimated quickly
  - Early, regular releases enable customer tests
- Courage
  - Tests let us change anything and see what breaks
- Respect (new to v2)
  - Teammates must respect each other

1. Beck, "Extreme Programming Explained: Embrace Change", 2000

TDD in XP

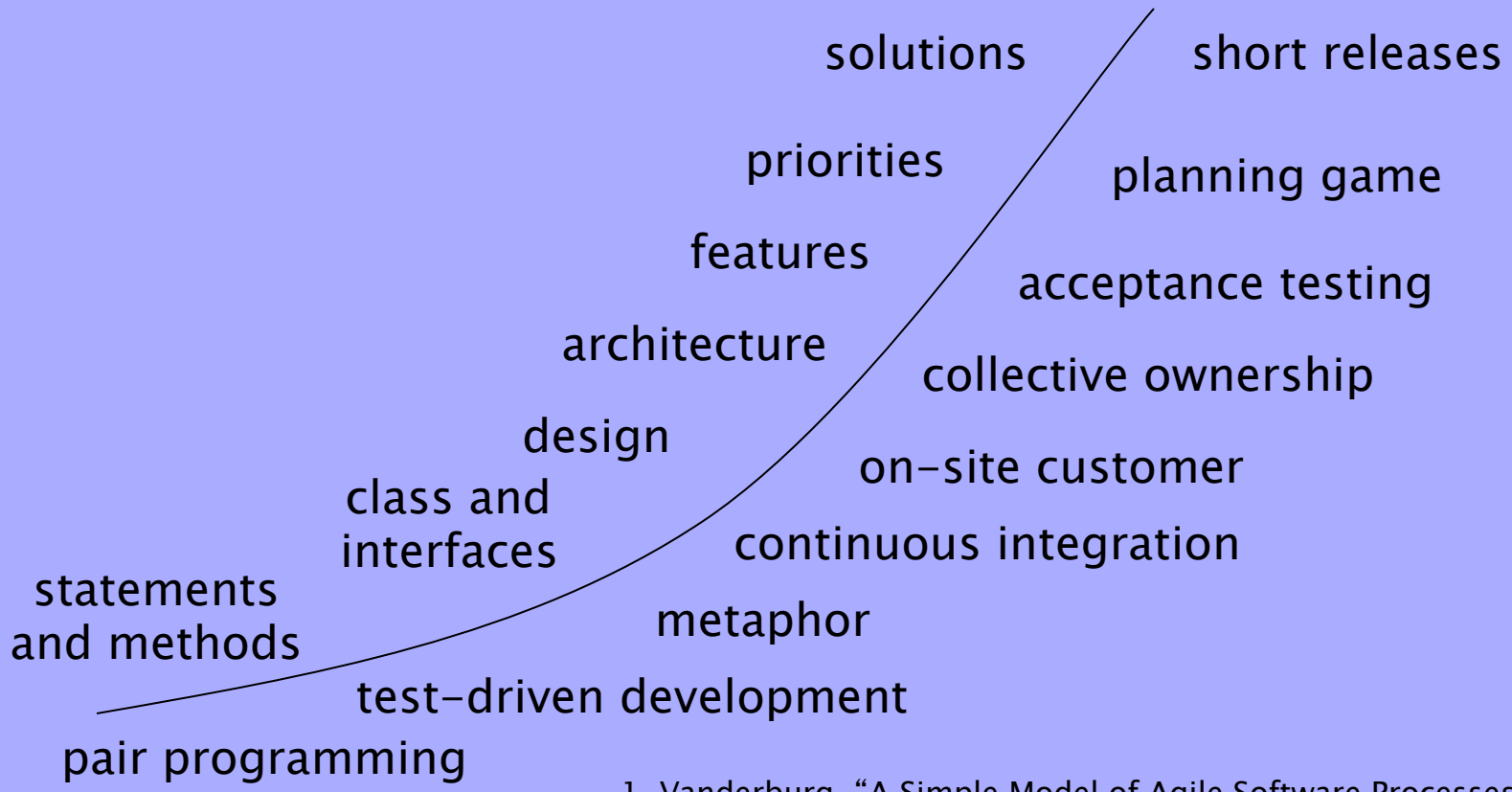


XP Practice Coupling[1]



1. Beck, Extreme Programming Explained: Embrace Change, 2000

XP Scale-Defined Practices[1]

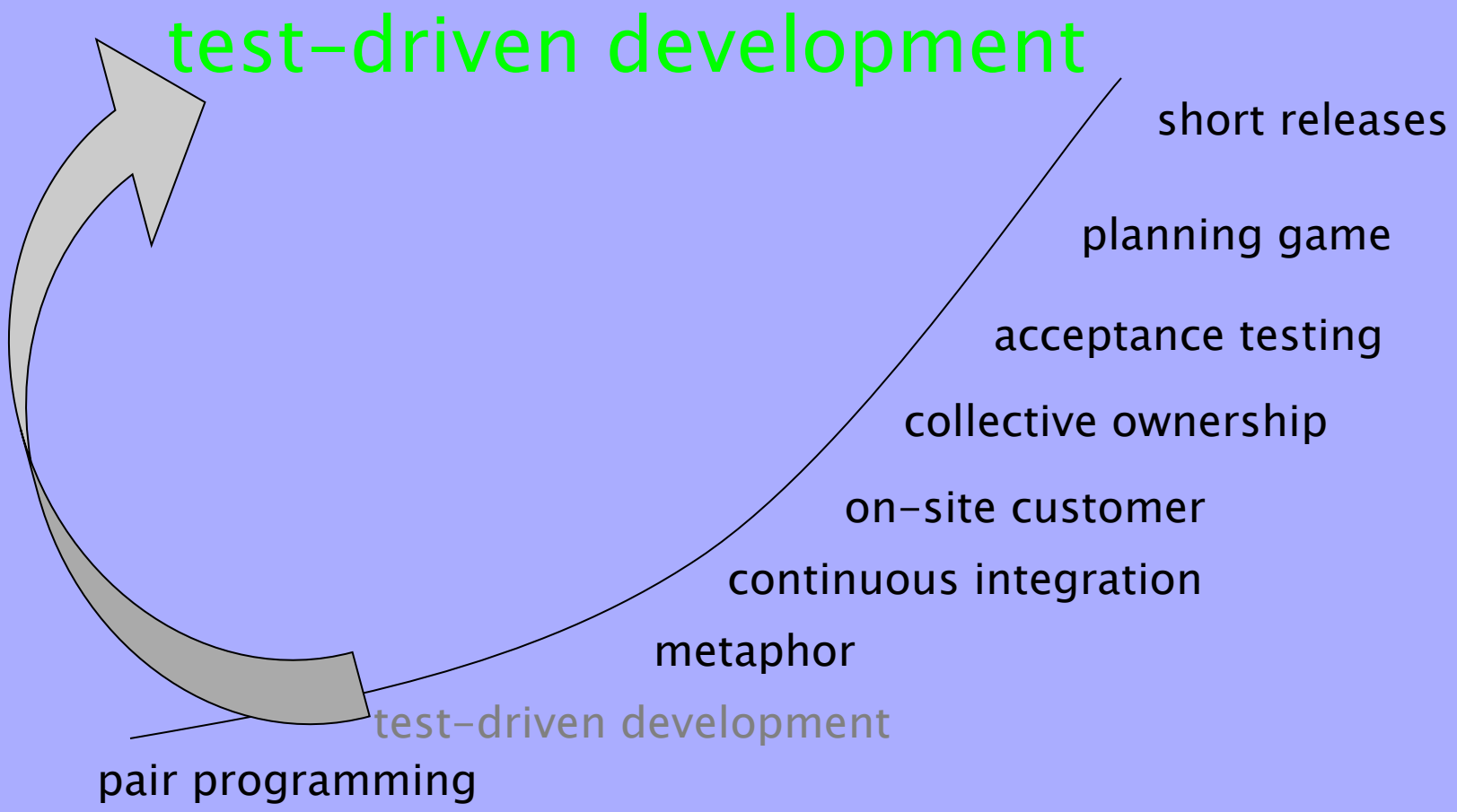


1. Vanderburg, "A Simple Model of Agile Software Processes", *OOPSLA*, 2005

XP Practices and Time Scales[1]

1. Vanderburg, "A Simple Model of Agile Software Processes", *OOPSLA*, 2005

Extracting TDD from XP



TDD in Software Development Lifecycle

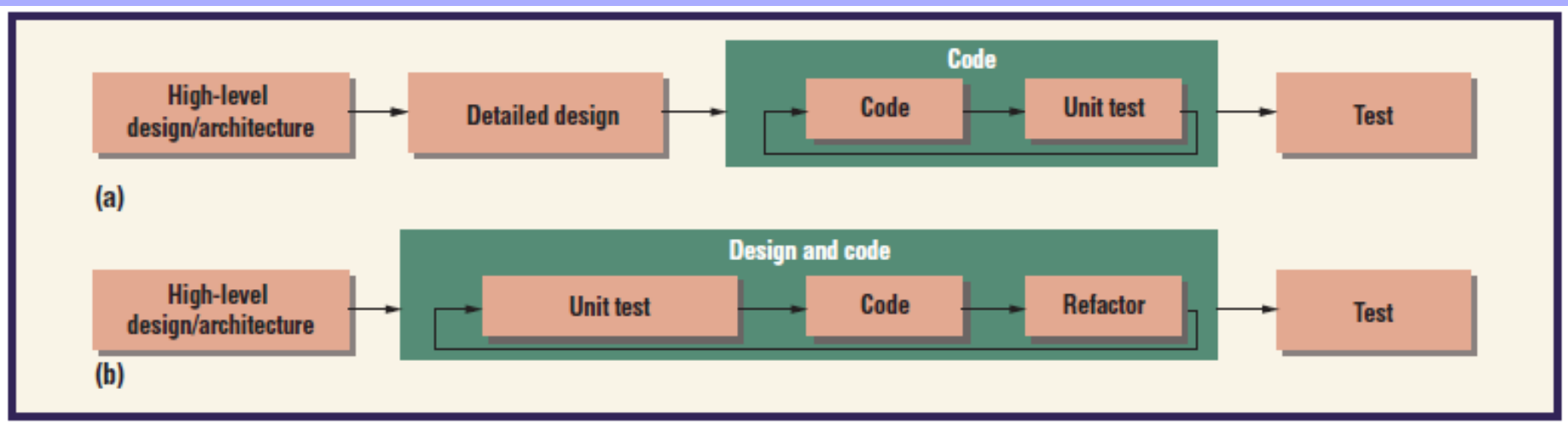
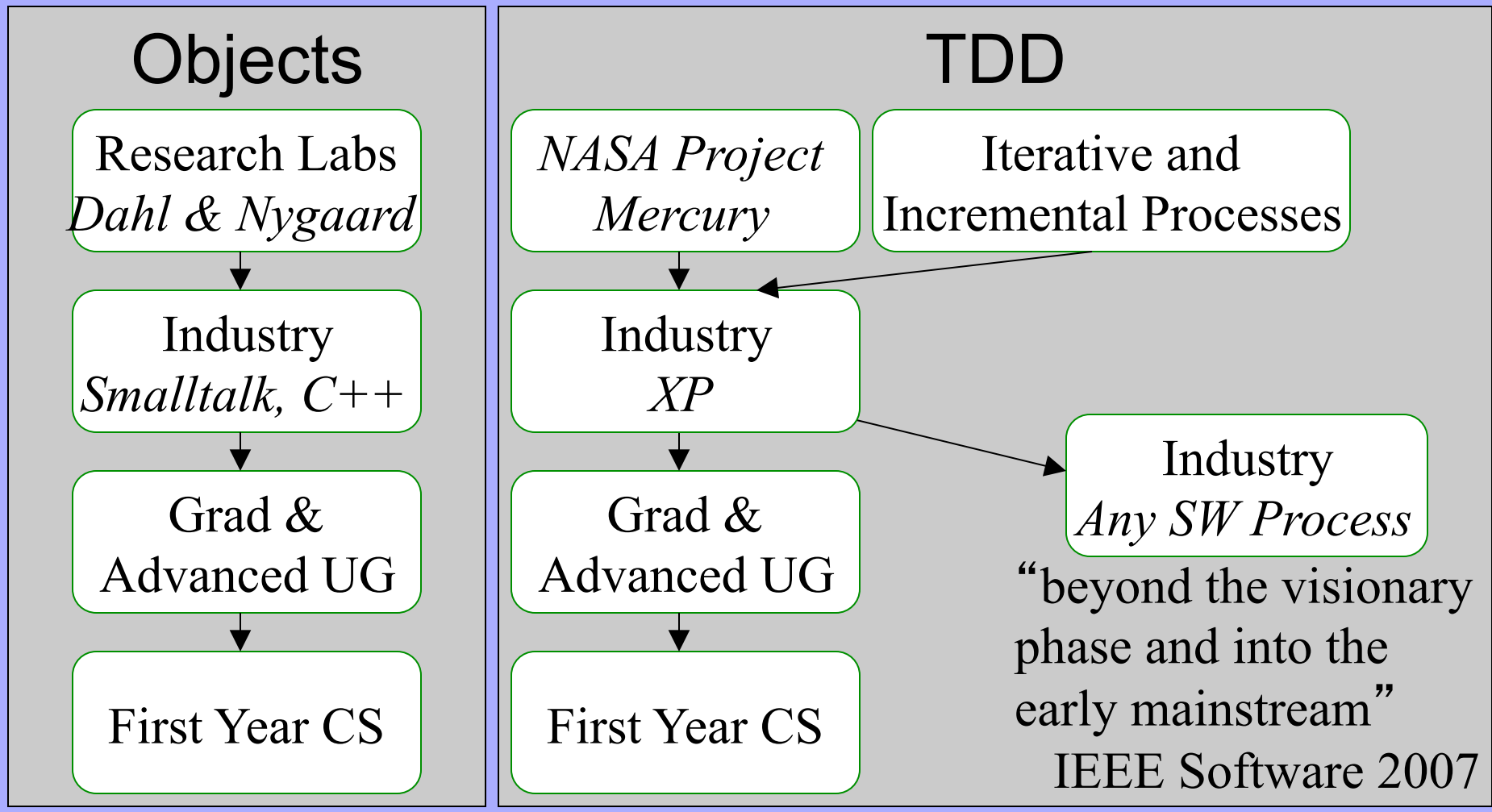


Figure 1. Development flow: (a) traditional test-last and (b) test-driven development/test-first flow.

TDD Future?



Why Test-Driven Development?

- Everybody else is using TDD
  - OK, not a great reason, but can't ignore it
  - Examples:
    - MS Silverlight 2 Beta 1 included >2000 tests, boasting >80% coverage for Controls.Test[1]
    - IEEE Software dedicated a 2007 edition to TDD
    - Wikipedia lists xUnit frameworks for 55 languages
    - Testimonials from companies such as Google, Intuit, and Salesforce.com (see Agile 200x for more)
    - Steve McConnell included TDD among his top ten software advances of the last decade

1. http://blogs.msdn.com/sburke/archive/2008/03/05/silverlight-2-beta-1-controls-available-including-source-and-unit-tests.aspx

Why Test-Driven Development?

- Promising Claims:
  - Courage
    - Automated tests allow immediate feedback to the implications of refactorings and defect fixes
  - Better Designs
    - Focuses on the use of code, not the implementation
    - Encourages simple designs
      - Less coupling, more cohesion
  - Increased Test Coverage
  - Teamwork
    - Tests are a form of code documentation
    - Unit tests can be completed in parallel, unlike integration tests which require complete units
  - Fun and addictive
    - Become Test-Infected and keep your code clean

TDD Evidence: Productivity and External Quality

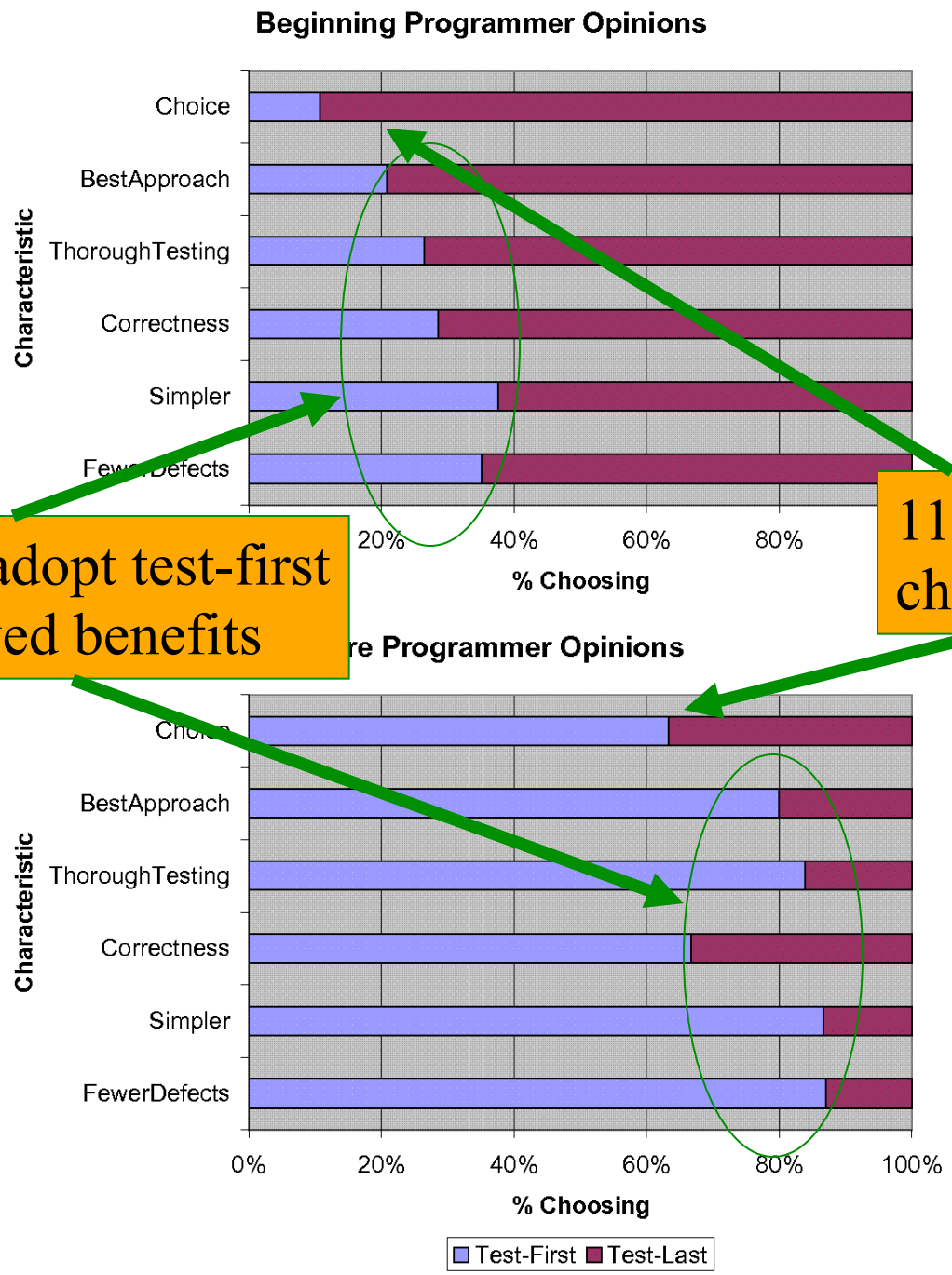
- May 2007 IEEE Software article summarized 9 industry studies and 9 academic studies
  - Industry study results
    - Most reported **increased effort**
      - Up to 100% on small projects, 5-19% on larger projects
    - Consistently reported **increased quality**
      - Up to 267% reduction in internal defect density
  - Academic studies a bit more mixed

TDD Evidence: Internal Quality

- March 2008 IEEE Software study (mine)
  - 15 projects over 5 years, 30+ KLOC
  - TDD produced **higher test coverage**
    - 30% higher line coverage, 78% branch coverage
  - TDD produced **smaller methods/classes**
    - 33% fewer LOC/method
    - 35% fewer methods/class
  - TDD produced **less complex methods/classes**
    - 54% lower average cyclomatic complexity (V(G))
    - 46% lower weighted methods per class (WMC)
  - Coupling and Cohesion results mixed

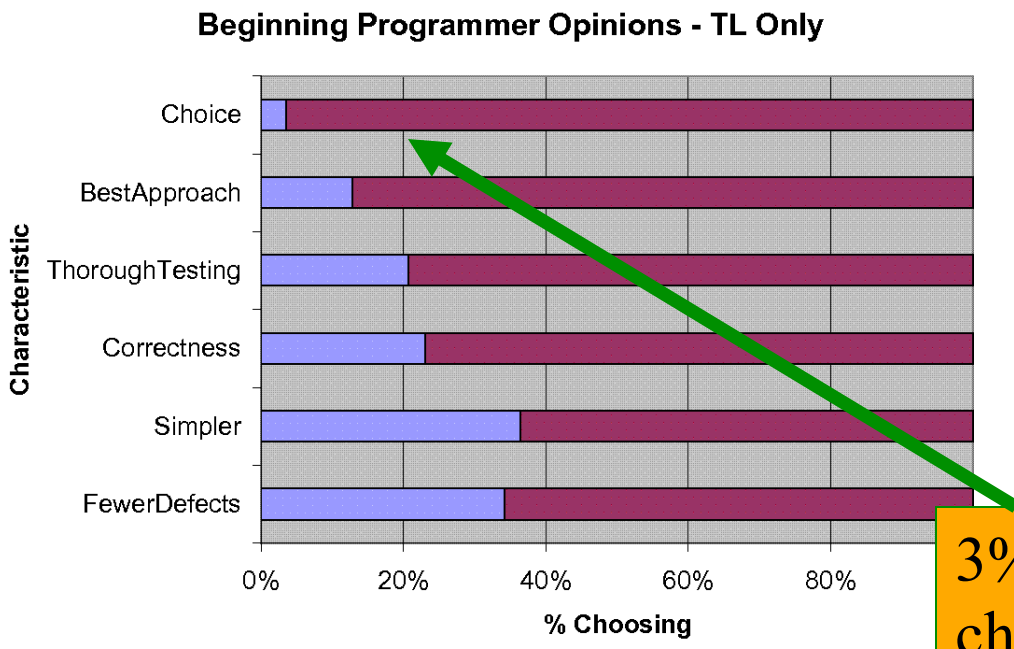
TDD Evidence: Opinions

- Study with ~150 students and industry professionals
  - Differences between early programmers and more mature programmers
  - TDD acceptance increased 47% with TDD experience
    - i.e. try it and you're more likely to like it

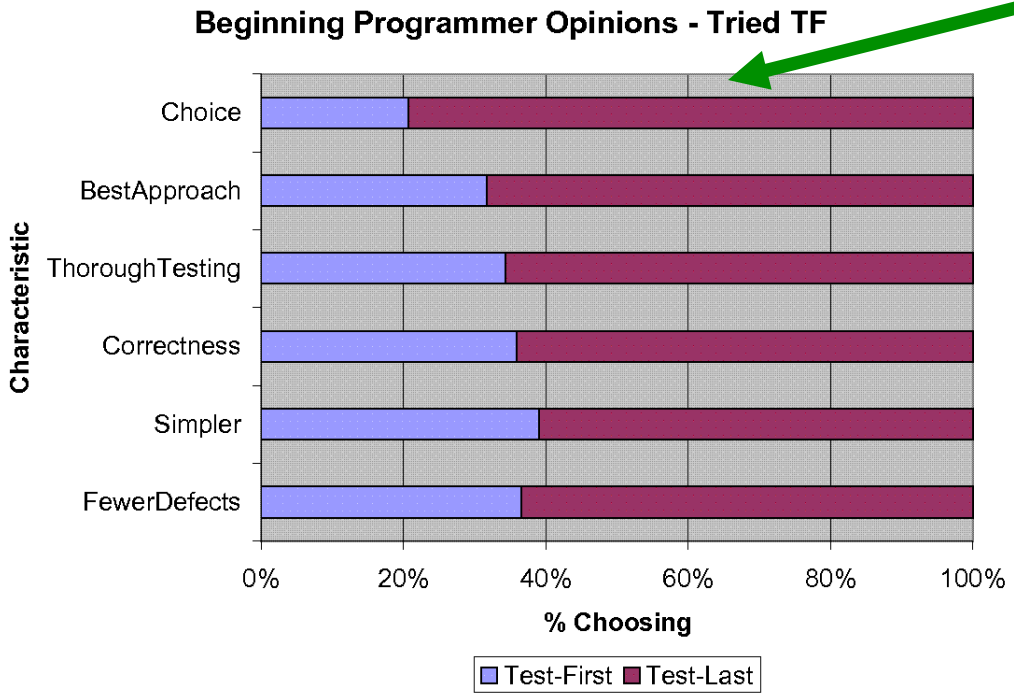


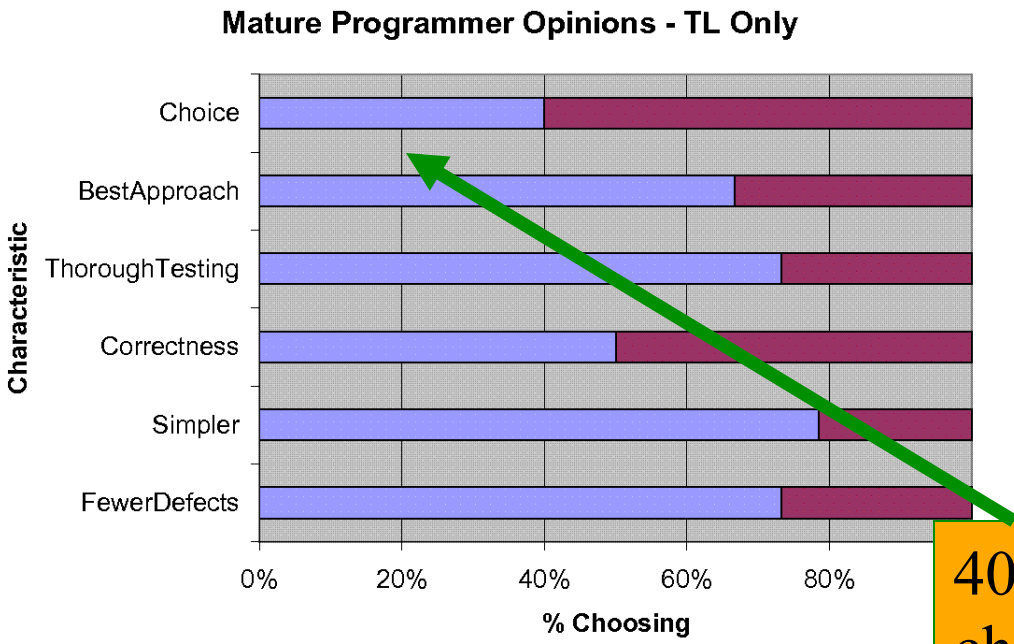
Reluctance to adopt test-first despite perceived benefits

11% vs 63% would choose test-first

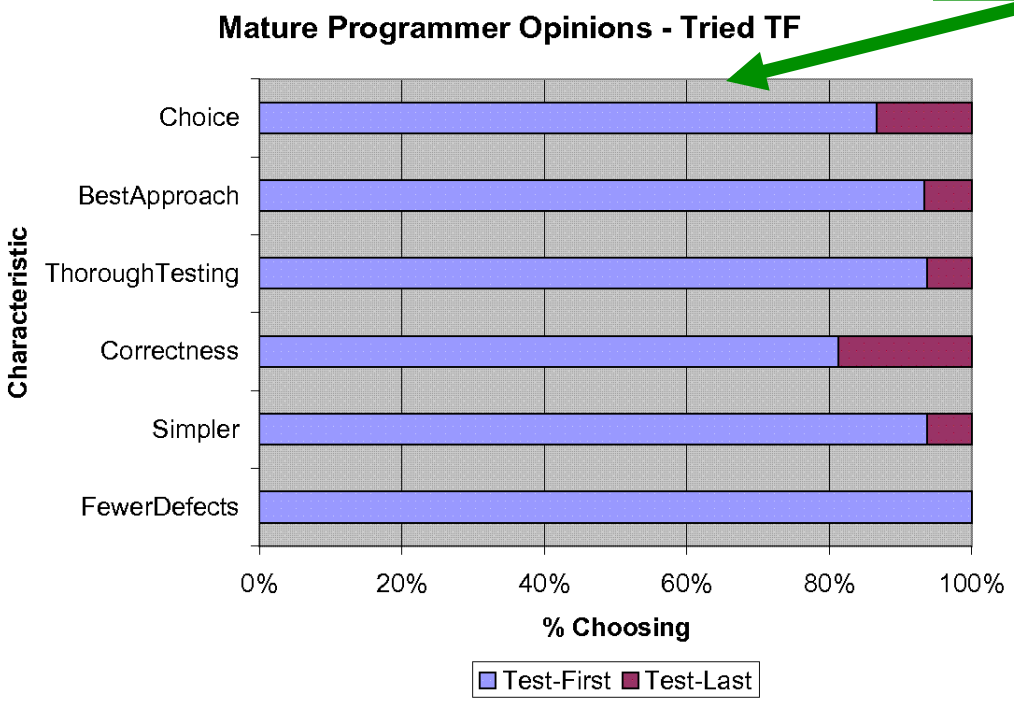


3% vs 21% would choose test-first





40% vs 87% would choose test-first



Anecdotal Evidence #1

- Environment:
  - Aircraft manufacturer
  - Small teams, informal process
  - Many Java web applications
- Experience:
  - TDD learning curve
  - Higher quality
  - Greatly simplified maintenance
    - Apps might be untouched for months
    - Tests provided documentation and safety net

Anecdotal Evidence #2

- Environment:
  - COTS provider
  - Small teams
- Experience:
  - TDD works with many processes (e.g. Scrum, TSP)
  - TDD integral to agility, continuous integration
  - Use code coverage as a primary metric

How can I apply TDD effectively?

- Try it
  - On a small scale
  - But give it some time
  - Don’t start with the UI
  - In maintenance,
    - Write a test to reveal a defect before fixing
    - Get code under test before changing it
  - Write tests to learn how an external library works
- Get some training or a mentor/coach

How can I apply TDD effectively?

- Learn TDD-related design patterns
  - Dependency injection
  - Test bus
- Learn TDD tools
  - xUnit
    - Fixtures, expecting exceptions
  - Mock object frameworks
- Integrate TDD into build/integration process
- Learn TDD patterns

Testing GUI's

- GUI's are hard to test
- Robots are software agents that can be programmed to execute tests on a GUI
  - Small changes in the UI can require big changes in the robot scripts
  - Robot scripts can be tedious and slow to build
  - Robot scripts can be slow to execute

Test Bus Architecture

- A test bus is a built-in access to API's that bypass the UI[1]

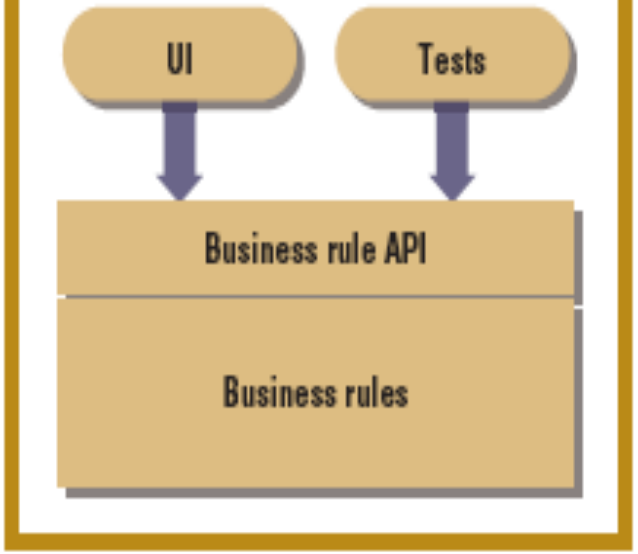


Figure 1. A testable system includes a test bus that can access the API independent of the UI.

1. Martin, "The Test Bus Imperative," IEEE Software, July 2005

TDD Patterns[1]

- Test List
  - Write the tests we think of on paper so we can focus on the test at hand
- Assert First
  - Start writing your test by writing the assert statement
- Explanation Test
  - Spread the use of automated testing by giving explanations to other developers in terms of tests
- Learning Test
  - Write tests for external libraries (Java SDK) to learn how they work
- Regression Test
  - When a new defect is reported, write a new test that would have caught it and consider why you didn't think of it the first time
- Do Over
  - When we get lost, sometimes the best thing to do is throw code away and start over
- Clean Check-in
  - Only check-in code that passes all the tests
- Broken Test
  - When programming alone, or when you want to remember what you were doing, leave a broken test so you go right to it when you come back

1. Beck, "Test-Driven Development by Example"