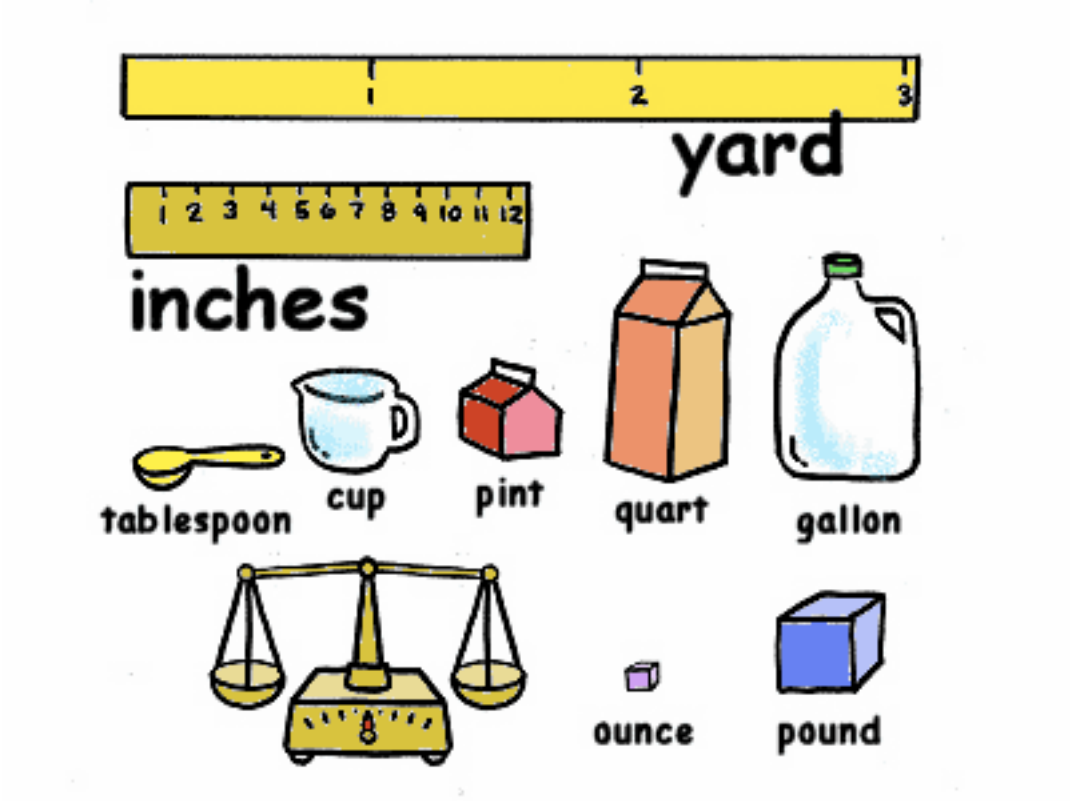
Unit Test

Common Excuses

- I’m not a tester!
- Customer is not paying for it
- We need to deliver yesterday.
- It takes too long to run the tests.
- Its a complex code, it cant be unit tested.
- We use agile.

Unit testing is not about finding bugs

What is a Unit ?



Depending on the type of language you are using, we might talk about modules, classes, functions, procedures or methods as the smallest unit.

In OO a unit is usually understood as a class or group of tightly coupled classes

The 3A Pattern

- Arrange
  - Prepare input
- Act
  - Call Method
- Assert
  - Verify output

```
public void test() {
        Math obj = new Math();
        int ans = obj.Add(10,20)
        assertEquals( ans, 30);
}
```

```
public void test() {

        Math obj = new Math();
        int ans = obj.Add(10,20)
        assertEquals( ans, 30);

        ans = obj.Sub(10,20)
        assertEquals( ans, -10);

        ans = obj.Mul(10,20)
        assertEquals( ans, 200);
}
```

Eager Test - Code Smell

The test verifies too much functionality in a single Test Method.

One Assert per Test

Test a single concept in each test function.

```
public void testAdd() {
    Math obj = new Math();
    int ans = obj.Add(10,20)
    assertEquals( ans, 30);
}
public void testSub() {
    Math obj = new Math();
    ans = obj.Sub(10,20)
    assertEquals( ans, -10);
}
public void testMul(){
    Math obj = new Math();
    ans = obj.Mul(10,20)
    assertEquals( ans, 200);
}
```

```
public void test() {

        Math obj = new Math();
        int ans = obj.Add(10,20)
        assertEquals( ans, 30);

        ans = obj.Add(100000,300000)
        assertEquals( ans, 400000);

        ans = obj.Add(-10,-10)
        assertEquals( ans, -20);
}
```

```
public void testAddWithSmallValue() {
    Math obj = new Math();
    int ans = obj.Add(10,20)
    assertEquals( ans, 30);
}
public void testAddWithLargeValue() {
    Math obj = new Math();
    ans = obj.Add(100000,300000)
    assertEquals( ans, 400000);
}
public void testAddWithNegativeValue() {
    Math obj = new Math();
    ans = obj.Add(-10,-10)
    assertEquals( ans, -20);
}
```

Data Driven Test using **JUnitParams**

```
@RunWith(JUnitParamsRunner.class)
public class MathTest {

    @Test
    @Parameters({
            "10, 20,   30",
            "0,0,0",
            "-10,10,0"   })
    public void addTest(int x,int y,int result) {
      Math obj = new Math();
      ans = obj.Add(x,y)
      assertEquals( ans, result);
    }
}
```

```
testPopHappyPath();
testPopEmptyStack();
testPushHappyPath();
testPushFullStack();
testPeek();
```

test fixture, not a method

```
public void testPushPop() {
    Stack stackUT = new Stack();
    Object expectedElement = new Object();
    assertEquals(expectedElement, stackUT.push(expectedElement).pop();
    assertTrue(stackUT.isEmpty());
}

public void testFILO() {
    Stack stackUT = new Stack();
    Object expectedOne = new Object();
    Object expectedTwo = new Object();
    stackUT.push(expectedOne);
    stackUT.push(expectedTwo);
    assertEquals(expectedTwo, stackUT.pop());
    assertEquals(expectedOne, stackUT.pop());
    assertTrue(stackUT.isEmpty());
}
```

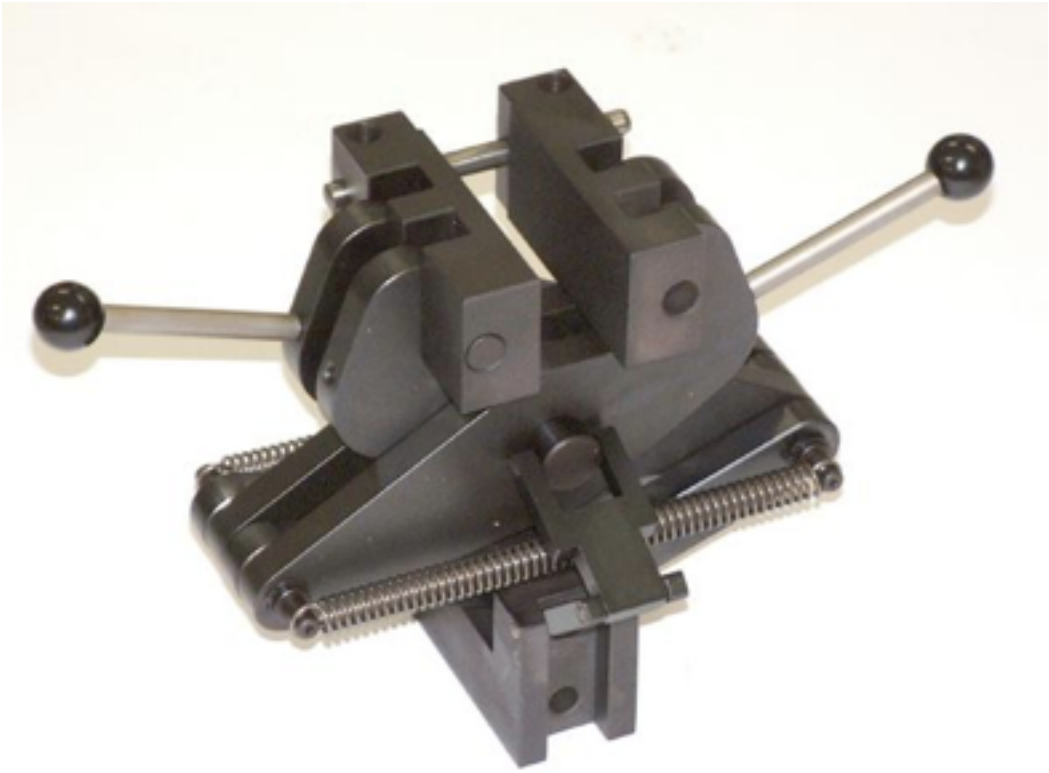
The contract is the thing you're going to test, not the implementation details.

```
public class TestClass
{
        public void testAdd() {
                        Math obj = new Math();
                        int ans = obj.Add(10,20)
                        assertEquals( ans, 30);
        }

        public void testWithdraw() {
                        Account obj = new Account();
                        bool res = obj.Withdraw(101,5000)
                        assertEquals( res, true);
        }

}
```

One Fixture Per Class



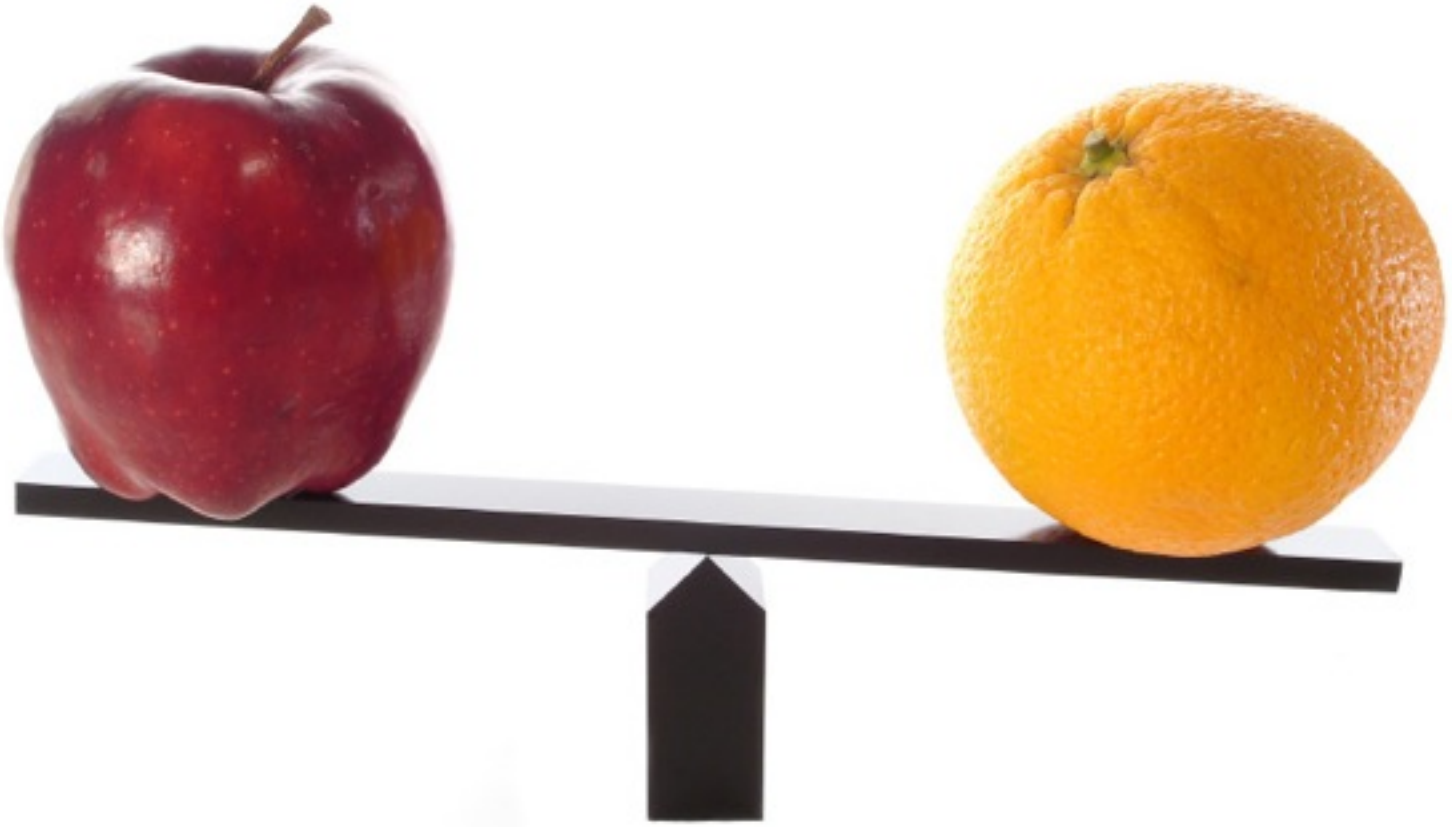
The term “fixture” means many things to many people

5 Steps

1. Setup
   - Initialize
2. Arrange
   - Prepare input
3. Act
   - Call Method
4. Assert
   - Verify output
5. Tear Down
   - Cleanup

```
public void testFlightMileage_asKm2() throws Exception {
	// set up fixture
	// exercise constructor
	Flight newFlight = new Flight(validFlightNumber);
	// verify constructed object
	assertEquals(validFlightNumber, newFlight.number);
	assertEquals("", newFlight.airlineCode);
	assertNull(newFlight.airline);
	// set up mileage
	newFlight.setMileage(1122);
	// exercise mileage translator
	int actualKilometres = newFlight.getMileageAsKm();
	// verify results
	int expectedKilometres = 1810;
	assertEquals( expectedKilometres, actualKilometres);
	// now try it with a canceled flight
	newFlight.cancel();
	try {
		newFlight.getMileageAsKm();
		fail("Expected exception");
	} catch (InvalidRequestException e) {
		assertEquals( "Cannot get cancelled flight mileage",e.getMessage());
	}
}
```

Keep Tests Clean



Test code is just as important as production code.
It is not a second-class citizen.

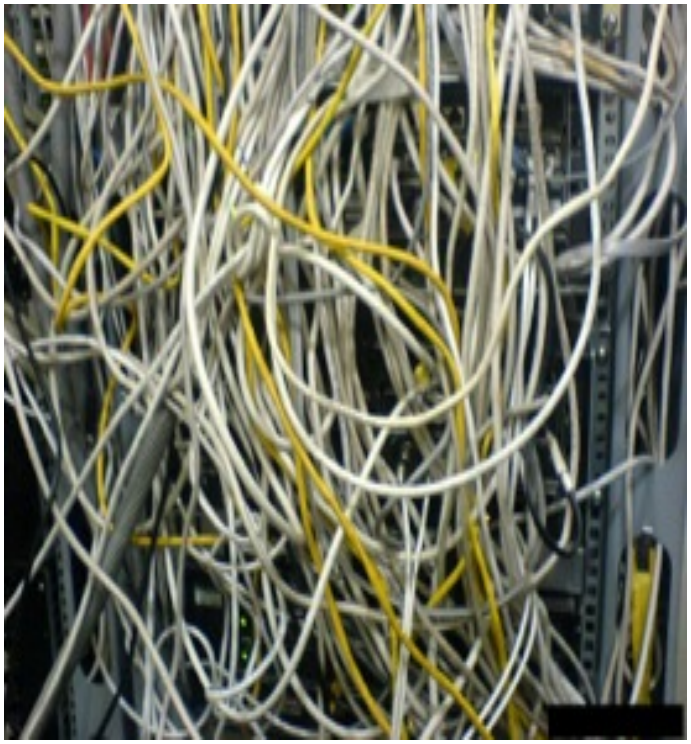
```
    // verify Vancouver is in the list
    actual = null;
    i = flightsFromCalgary.iterator();
    while (i.hasNext()) {
        FlightDto flightDto = (FlightDto) i.next();
        if (flightDto.getFlightNumber().equals(
        expectedCalgaryToVan.getFlightNumber()))
        {
            actual = flightDto;
            assertEquals("Flight from Calgary to Vancouver",
            expectedCalgaryToVan,
            flightDto);
            break;
        }
    }
}
```

Tests as Documentation

should act as documentation of how the system under test (SUT) *should behave*

Avoid logic in tests

- switch, if, else statements
- foreach, for, while loops



```
public void testWithConditionals() {
    String expectedLastname = "smith";
    List foundPeople = PeopleFinder.findPeopleWithLastname(expectedLastname);

    if (foundPeople != null) {
        if (foundPeople.size() == 1) {

                Person solePerson = (Person) foundPeople.get(0);
                assertEquals( expectedLastname,solePerson.getName());
        } else {
                fail("list should have exactly one element");
        }
    } else {
        fail("list is null");
    }
}
```

K
- Keep

I
- It

S
- Simple

Guard Assertion

```
public void testWithoutConditionals() throws Exception {
    String expectedLastname = "smith";
    List foundPeople = PeopleFinder.
        findPeopleWithLastname(expectedLastname);
    assertNotNull("found people list", foundPeople);
    assertEquals( "number of people", 1, foundPeople.size() );
    Person solePerson = (Person) foundPeople.get(0);
    assertEquals( "last name", expectedLastname,solePerson.getName() );
}
```

We replace an if statement in a test with an assertion that fails the test if not satisfied.

```
public void test() {
      try{
            Math obj = new Math();
                  int ans = obj.Add(10,20)
                  assertEquals( ans, 30);
          }
          catch(ArithmaticOverflowException e)
          {
                              fail("invalid data");
            }
}
```

Avoid Exception Handling

```
public void test() {
                    Math obj = new Math();
                int ans = obj.Add(10,20)
                assertEquals( ans, 30);
}
```

```
public void test() {
      try{
            Math obj = new Math();
                  int ans = obj.Add(109876,20678678)
                  fail("invalid data");


          }
          catch(ArithmaticOverflowException e)
          {
           }
}
```

```
 public void testScheduledState_shouldThrowInvalidRequestEx()
         throws Exception
{
    Flight flight = FlightTestHelper.getAnonymousFlightInScheduledState();
    try {
          flight.schedule();
          fail("not allowed in scheduled state");
     }
     catch (InvalidRequestException e) {
          assertEquals("InvalidRequestException.getRequest()","schedule",e.getRequest());
          assertTrue(  "isScheduled()", flight.isScheduled());
     }

}
```

```
@expectedException(typeof(ArithmaticOverflowException))
public void test() {
        Math obj = new Math();
            int ans = obj.Add(109876,20678678)
    }
```

```
public class Invoice
{
    private double getSubTotal()
    {
        ..........
    }
}
```

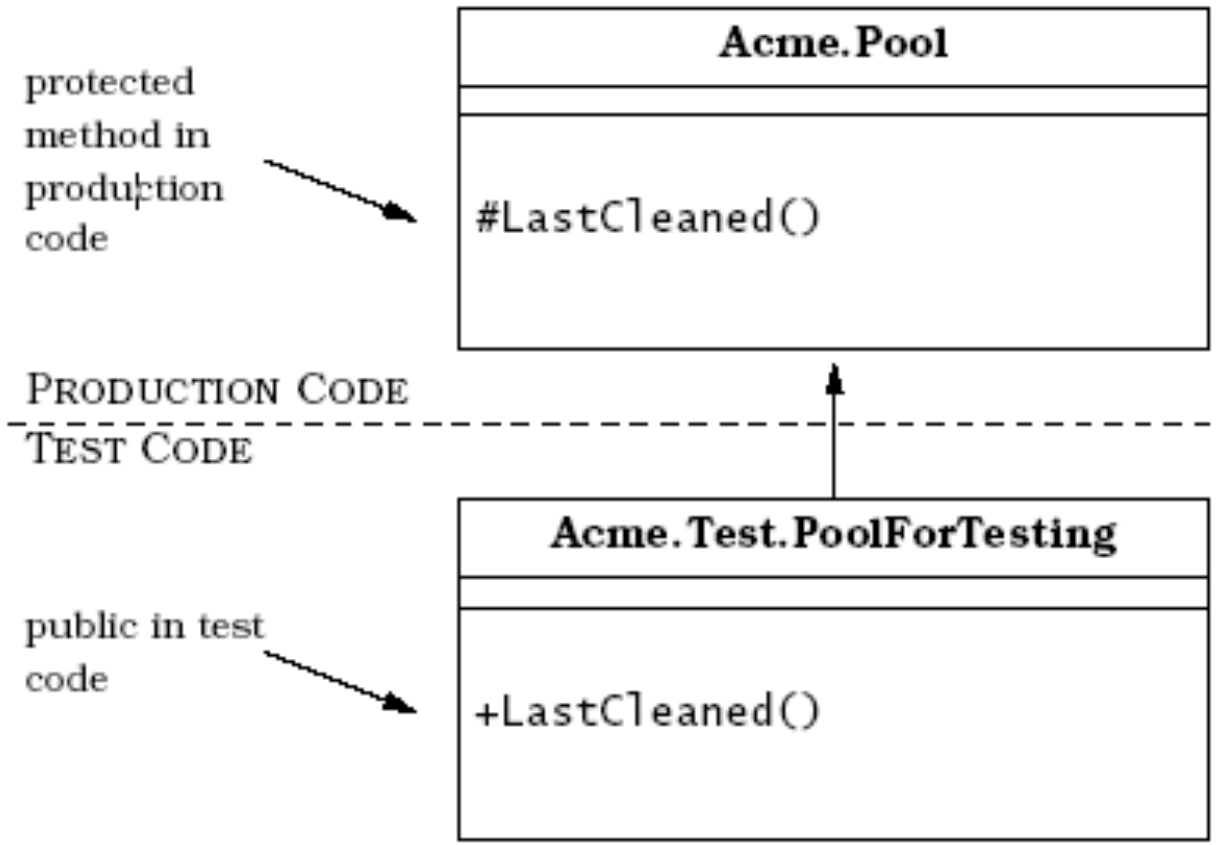
Don't test private methods

When you test a private method, you’re testing against a contract internal to the system, which may well change.

```
public class Invoice
{
    protected double getSubTotal()
    {
        ..........
     }
}
```

```
public class Customer
{
        public getName() {}
        public setName(v){}
        public getAge() {}
        public setAge(v) {}
}
```

Testing Protected Methods



```
public void testInvoice_addOneLineItem_quantity1_b() {
        // Exercise
        inv.addItemQuantity(product, QUANTITY);
        // Verify
        List lineItems = inv.getLineItems();
        assertEquals("number of items", lineItems.size(), 1);
        // Verify only item
        LineItem expItem = new LineItem(inv, product, QUANTITY);
        LineItem actual = (LineItem)lineItems.get(0);
        assertEquals(expItem.getInv(), actual.getInv());
        assertEquals(expItem.getProd(), actual.getProd());
        assertEquals(expItem.getQuantity(), actual.getQuantity());
}

public void testRemoveLineItemsForProduct_oneOfTwo() {
        // Set up
        Invoice inv = createAnonInvoice();
        inv.addItemQuantity(product, QUANTITY);
        inv.addItemQuantity(anotherProduct, QUANTITY);
        LineItem expItem = new LineItem(inv, product, QUANTITY);
        // Exercise
        inv.removeLineItemForProduct(anotherProduct);
        // Verify
        List lineItems = inv.getLineItems();
        assertEquals("number of items", lineItems.size(), 1);
        LineItem actual = (LineItem)lineItems.get(0);
        assertEquals(expItem.getInv(), actual.getInv());
        assertEquals(expItem.getProd(), actual.getProd());
        assertEquals(expItem.getQuantity(), actual.getQuantity());
}
```

The Silent Catcher AntiPattern

```
[Test]
[ExpectedException(typeof(Exception))]
public void ItShouldThrowDivideByZeroException()
{
   // some code that throws another exception yet passes
    the test
}
```

A test that passes if an exception is thrown.. even if the exception that actually occurs is one that is different than the one the developer intended.

Humble object

"xUnit Test Patterns"

extract the logic into a separate, easy-to-test component that is decoupled from its environment. After you tested this logic, you can test the complicated behaviour (multi- threading, asynchronous execution, etc...)

Keep Test Logic Out of Production Code

The production code should not contain any conditional statements of the if testing then sort

Regression Test

intends to ensure that changes (enhancements or defect fixes) to the software have not adversely affected it.

Regression testing can be performed during any level of testing (Unit, Integration, System, or Acceptance)

“any method that can break is a good candidate for having a unit test, because it may break at some time, and then the unit test will be there to help you fix it.”

Write Test before Bug Fix

Write a new test that causes the failure to happen and then make the test pass by fixing the defect.

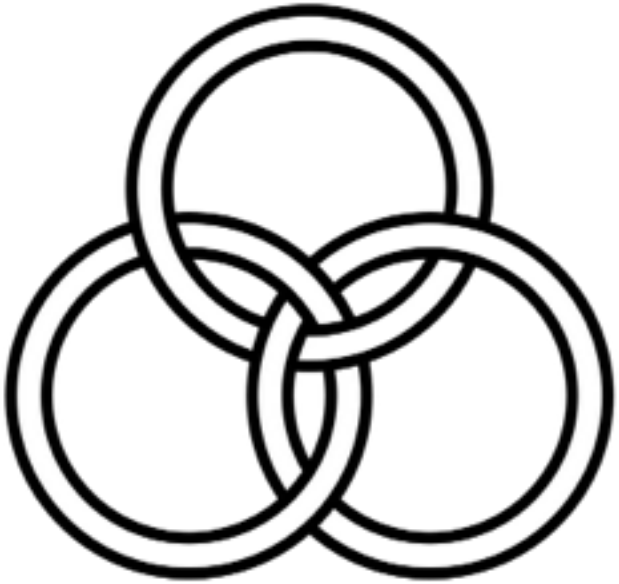
The Test With No Name Antipattern

```
public void test testForBUG123() {
    ....
}
```

The test that gets added to reproduce a specific bug in the bug tracker and whose author thinks does not warrant a name of its own. Instead of enhancing an existing, lacking test, a new test is created called testForBUG123.

Another good litmus test is to look at the code and see if it throws an error or catches an error. If error handling is performed in a method, then that method can break.

Minimize Test Overlap

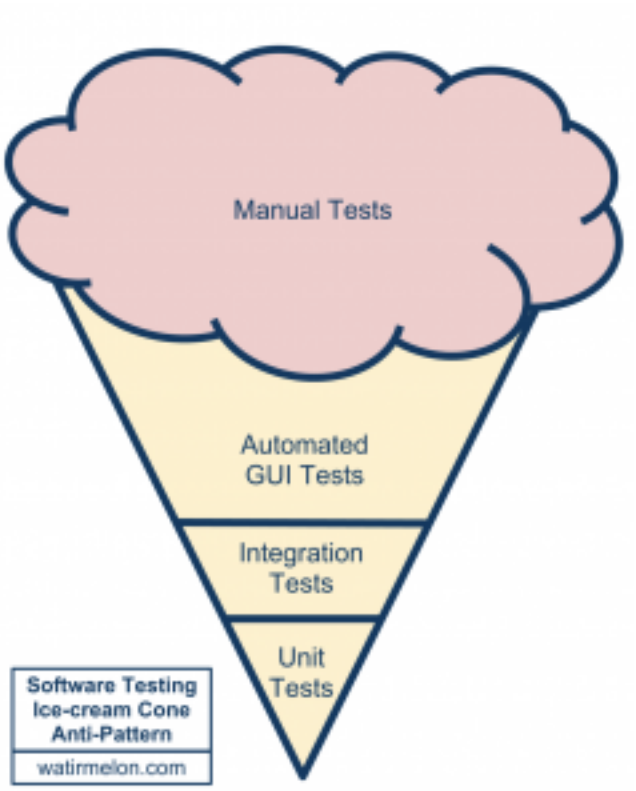


Having several tests verify the same functionality is likely to increase test maintenance costs and probably won't improve quality very much.

Unit testing is one of the most important components in legacy code work.

Traditionally,most if not all business logic and integration logic is tested through the GUI layer. With the GUI layer bring brittle, minor changes to the UI will create many tests to break.

Ice Cream Cone Anti Pattern

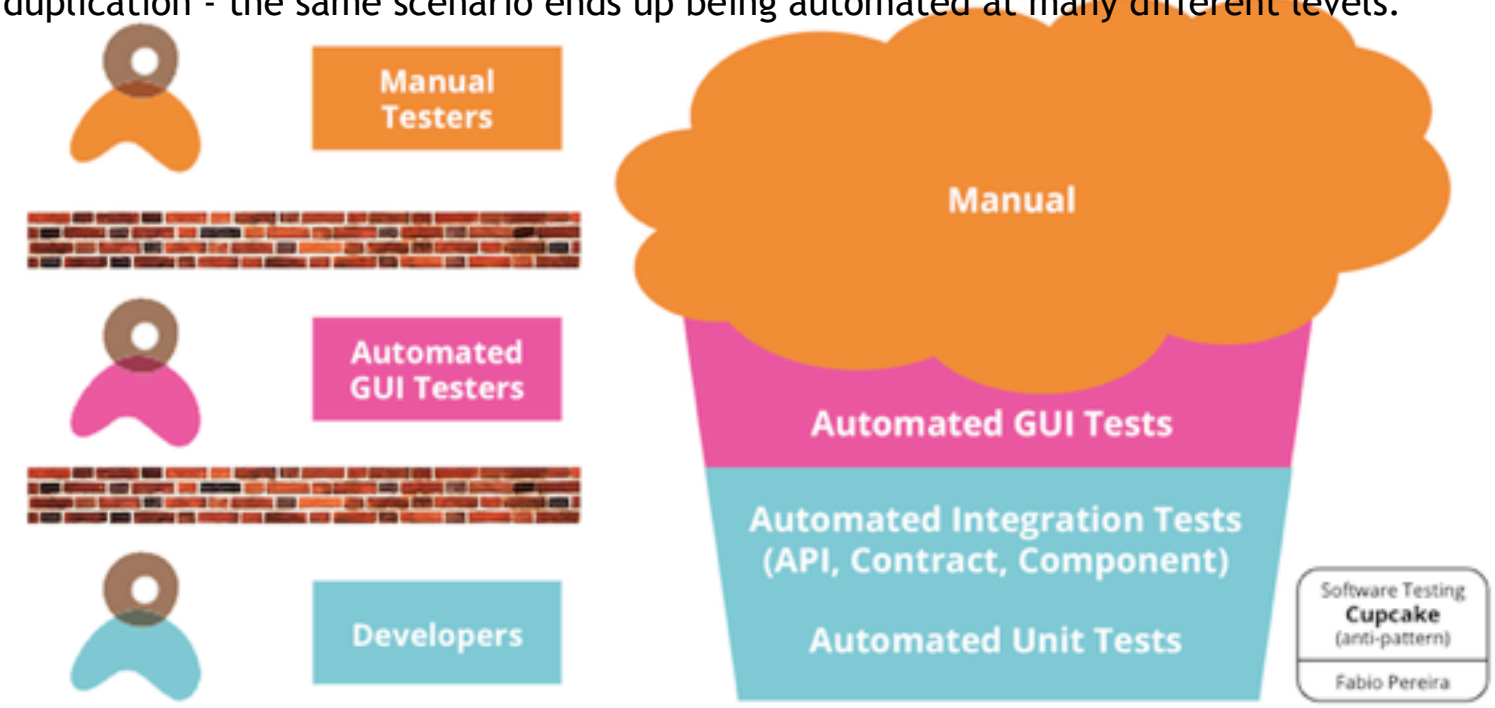


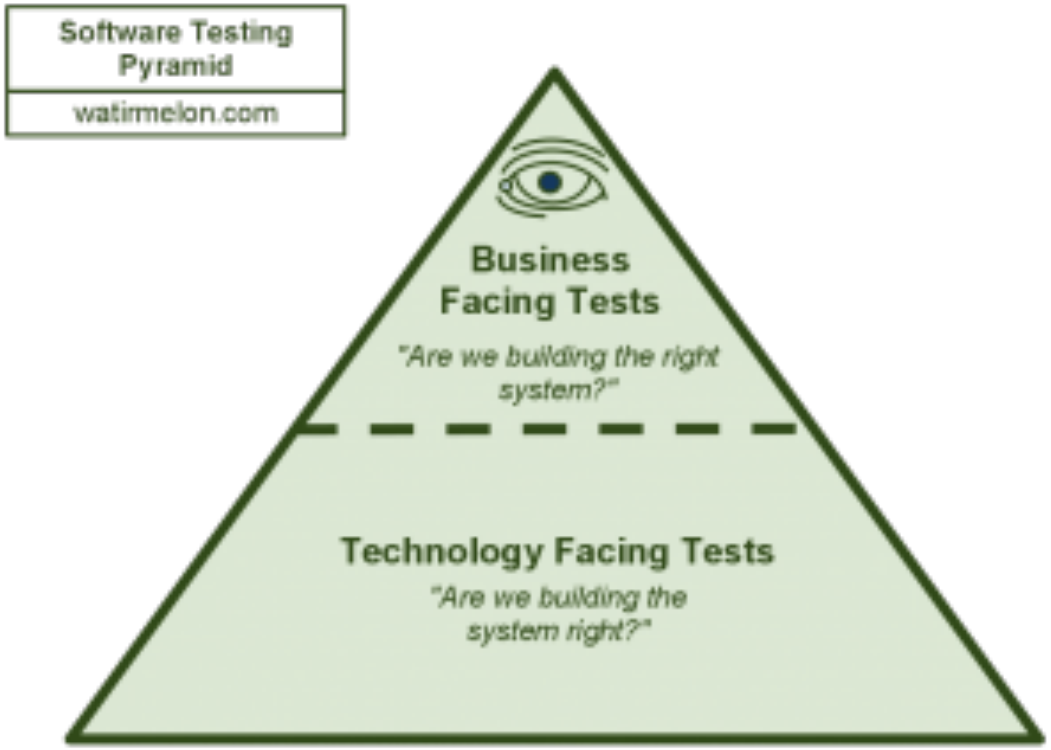
- happens when there is not enough low-level testing (unit, integration and component), too many tests that run through the Graphical User Interface (GUI) and an even larger number of manual tests.

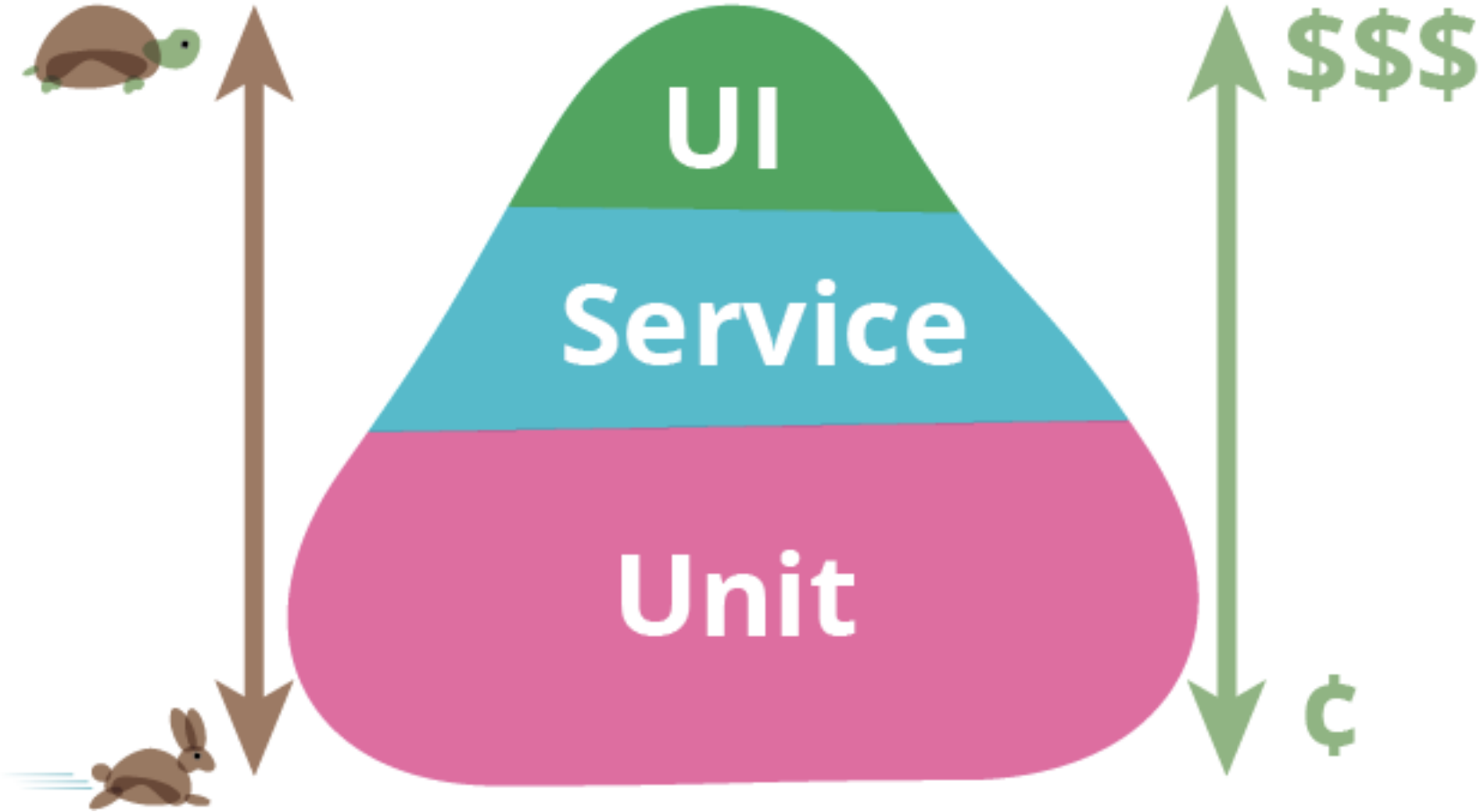
Cup Cake Anti Pattern

- Writing tests usually spans 3 teams:
- Developers write unit, integration and component tests.
- Another team writes black box tests through the GUI.
- Manual testers have a set of scenarios that they execute manually.

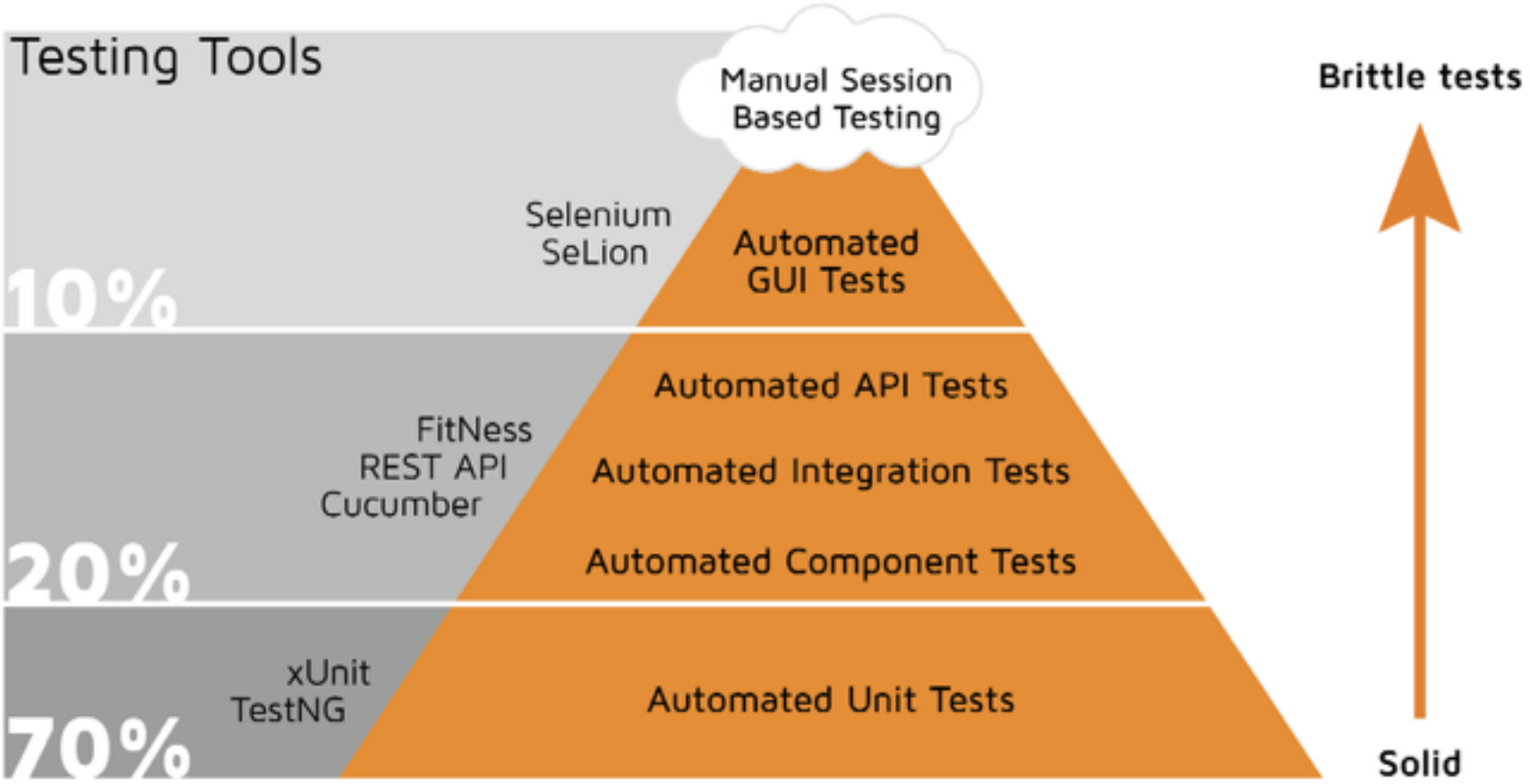
- This results in duplication - the same scenario ends up being automated at many different levels.



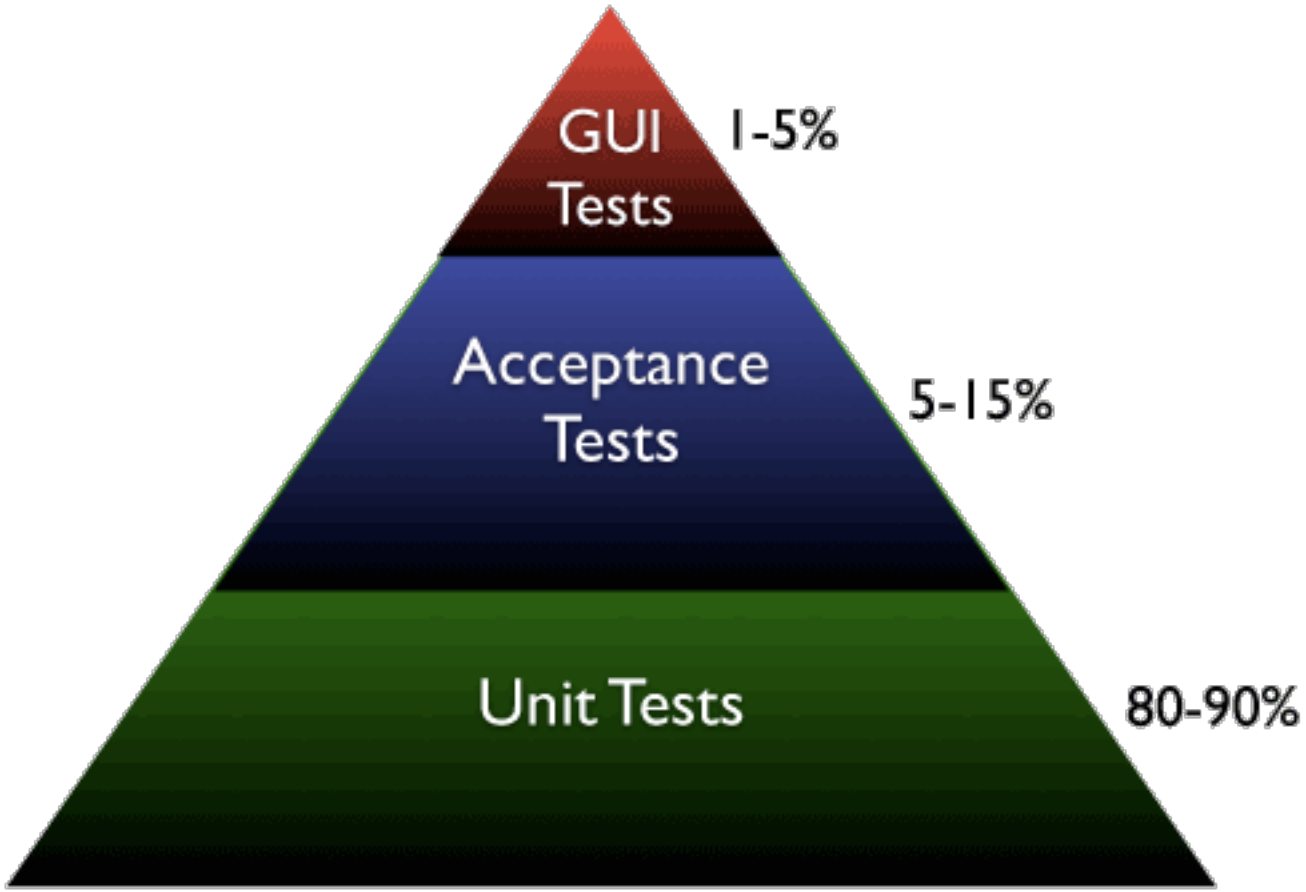

Software Testing Pyramid
watirmelon.com
Business Facing Tests
"Are we building the right system?"
Technology Facing Tests
"Are we building the system right?"

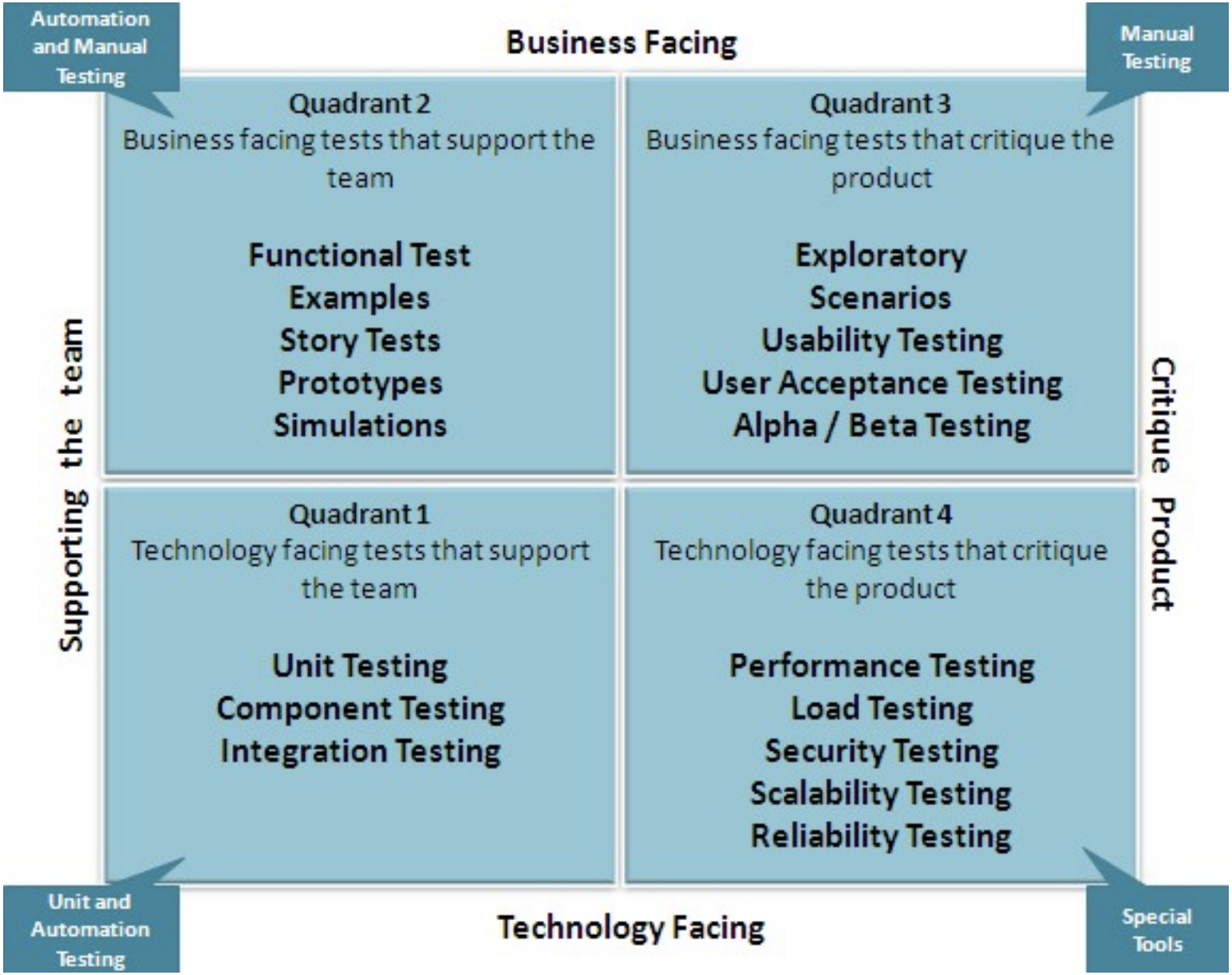

UI
Service
Unit
$$$
¢

Test Pyramid Strategy

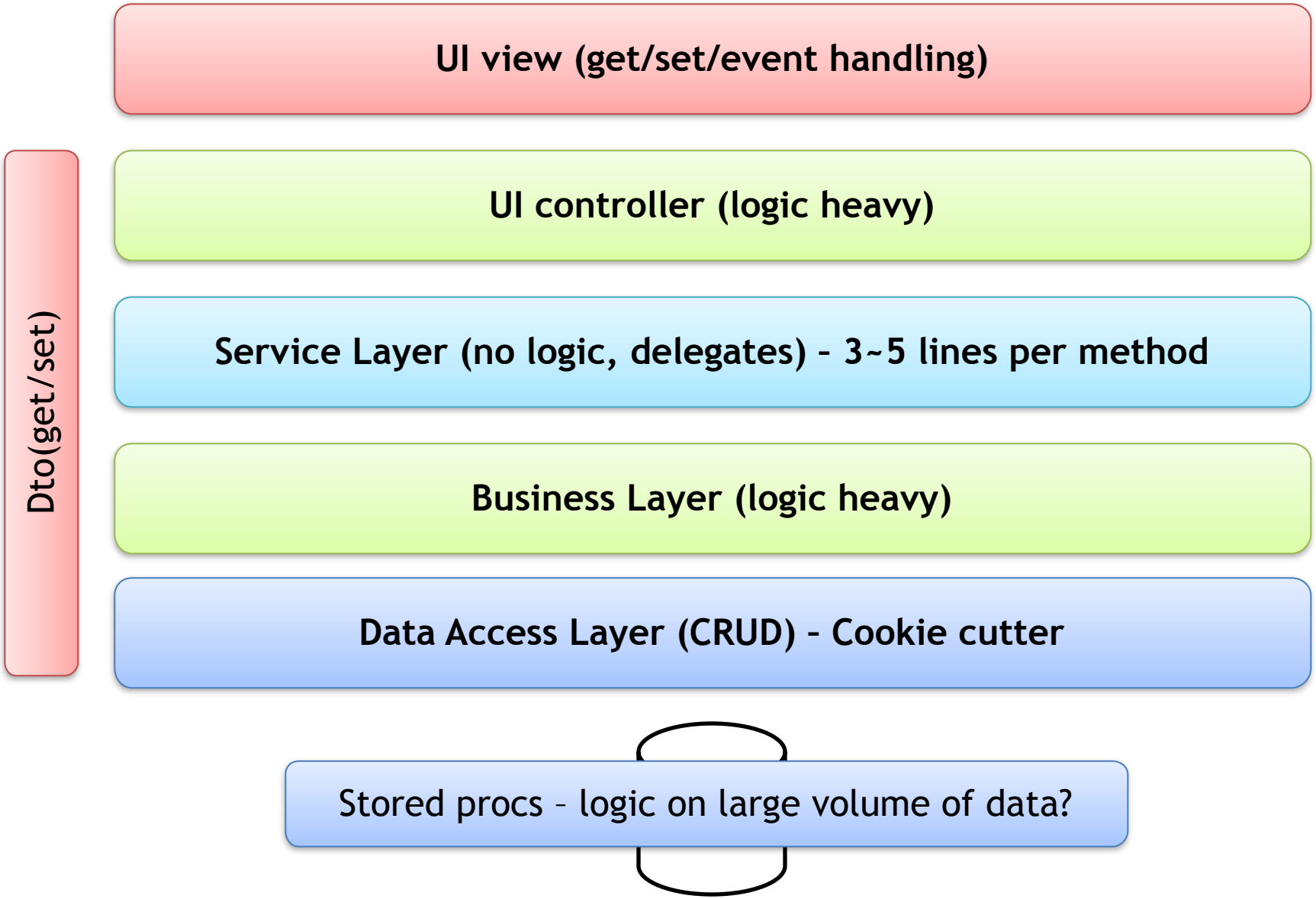


- move most of the tests down into the lower layers where UI changes will not effect the integrity of the tests.


GUI Tests
1-5%
Acceptance Tests
5-15%
Unit Tests
80-90%


Automation and Manual Testing
Business Facing
Manual Testing
Quadrant 2
Business facing tests that support the team
Functional Test
Examples
Story Tests
Prototypes
Simulations
Quadrant 3
Business facing tests that critique the product
Exploratory
Scenarios
Usability Testing
User Acceptance Testing
Alpha / Beta Testing
Supporting the team
Critique Product
Quadrant 1
Technology facing tests that support the team
Unit Testing
Component Testing
Integration Testing
Quadrant 4
Technology facing tests that critique the product
Performance Testing
Load Testing
Security Testing
Scalability Testing
Reliability Testing
Unit and Automation Testing
Technology Facing
Special Tools

When the team of developers, manual testers and GUI testers work together to achieve the same goal, collaborating and helping each other, we can achieve a much better testing strategy that will ensure the quality of our software.


UI view (get/set/event handling)
UI controller (logic heavy)
Service Layer (no logic, delegates) - 3~5 lines per method
Business Layer (logic heavy)
Data Access Layer (CRUD) - Cookie cutter
Dto(get/set)
Stored procs - logic on large volume of data?

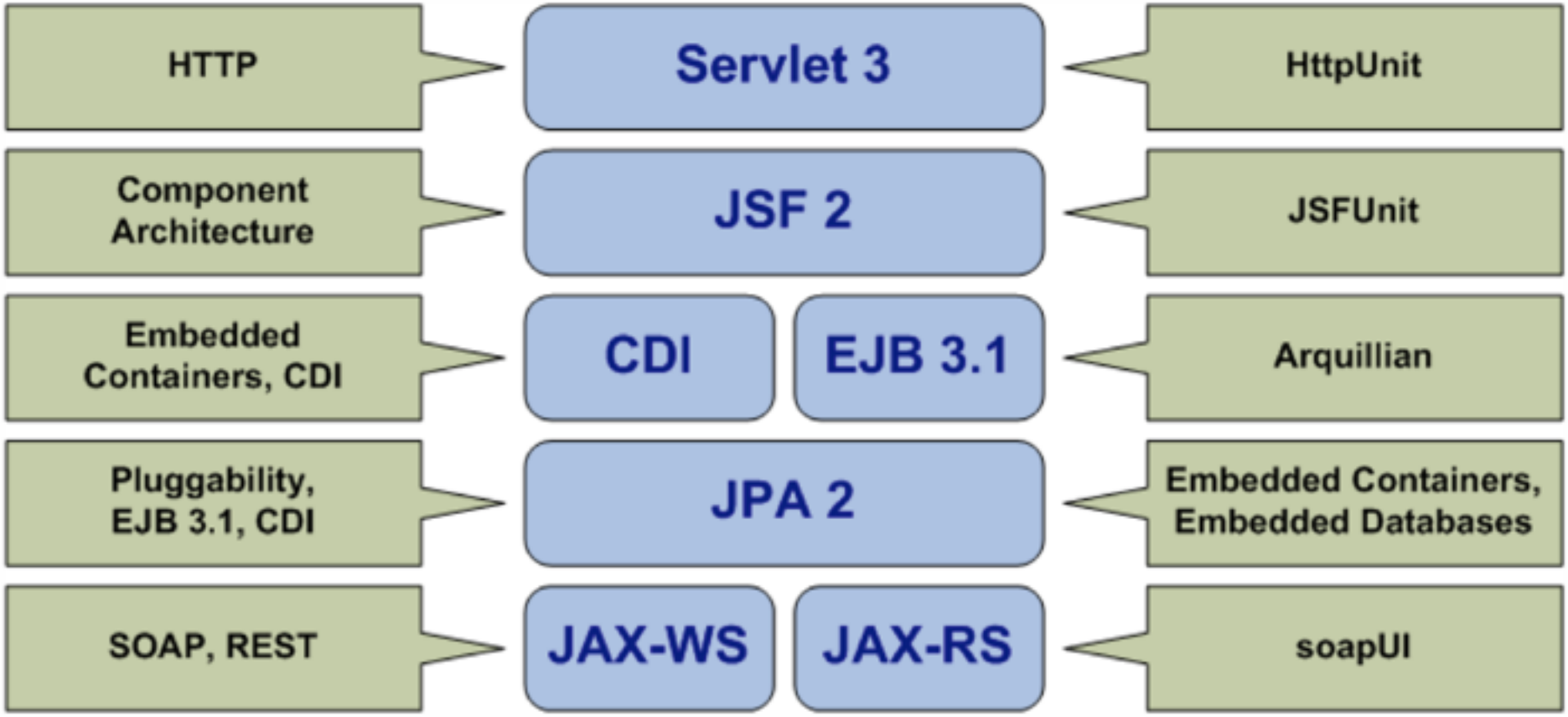
Minimize Untestable Code

GUI components
Multithreaded code

We can refactor the untestable code to improve its testability by moving the logic we want to test out of the class that is causing the lack of testability.

Remote Stored Procedure Test

whenever we have nontrivial logic in stored procedures. JUnit tests for each stored procedure via the JDBC PreparedStatement classes.


HTTP
Servlet 3
HttpUnit
Component Architecture
JSF 2
JSFUnit
Embedded Containers, CDI
CDI
EJB 3.1
Arquillian
Pluggability, EJB 3.1, CDI
JPA 2
Embedded Containers, Embedded Databases
SOAP, REST
JAX-WS
JAX-RS
soapUI

Unit tests are Fast

Single Test < 200 ms
Small Suite < 10s
All Test Suits < 10 mins

Unit tests must run quickly, encouraging developers to run them as often as possible.

Unit Tests are **Repeatable**

Tests should run in the production environment, QA environment, and your laptop while riding home on the train without a network.

What Unit Test will not do ?

- It talks to a database.
- It communicates across a network.
- It touches the file system.
- You have to do special things to your environment (such as editing configuration files) to run it.

Chain Gang Antipattern

- tests that must run in a certain order

Will not have Dpenedancy

They cannot have external dependencies to resource like networks, databases, file systems, webservice or third-party software.

"I should be testing the behavior of the method, not that it's calling the right sequence of methods"

Mockery Antipattern

a unit test contains so many mocks, stubs, and/or fakes that the system under test isn't even being tested at all, instead data returned from mocks is what is being tested.

```
public class StudentVO {
   private String name;
   private int rollNo;

   StudentVO(String name, int rollNo){
      this.name = name;
      this.rollNo = rollNo;
   }

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }

   public int getRollNo() {
      return rollNo;
   }

   public void setRollNo(int rollNo) {
      this.rollNo = rollNo;
   }
}
```

Easy tests Antipattern

```
public void() {
        ....
}
```

the code is difficult to test. As a result, you see lots of tests for things that are easy to test, and the real logic of the unit under test is ignored.

- Tests should help us improve quality.
- Tests should help us understand the SUT.
- Tests should reduce (and not introduce) risk.

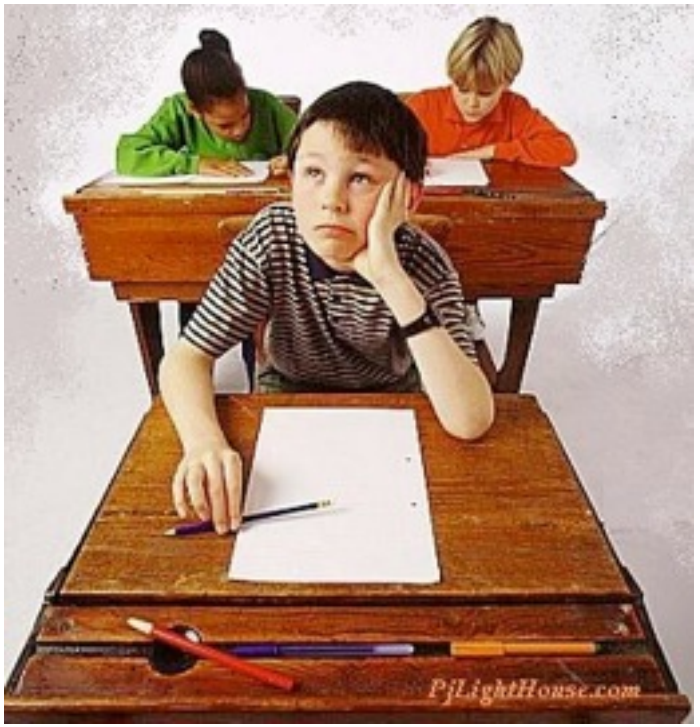
Philosophy

"Any program feature without an automated test simply doesn't exist."

from Extreme Programming Explained, Kent Beck

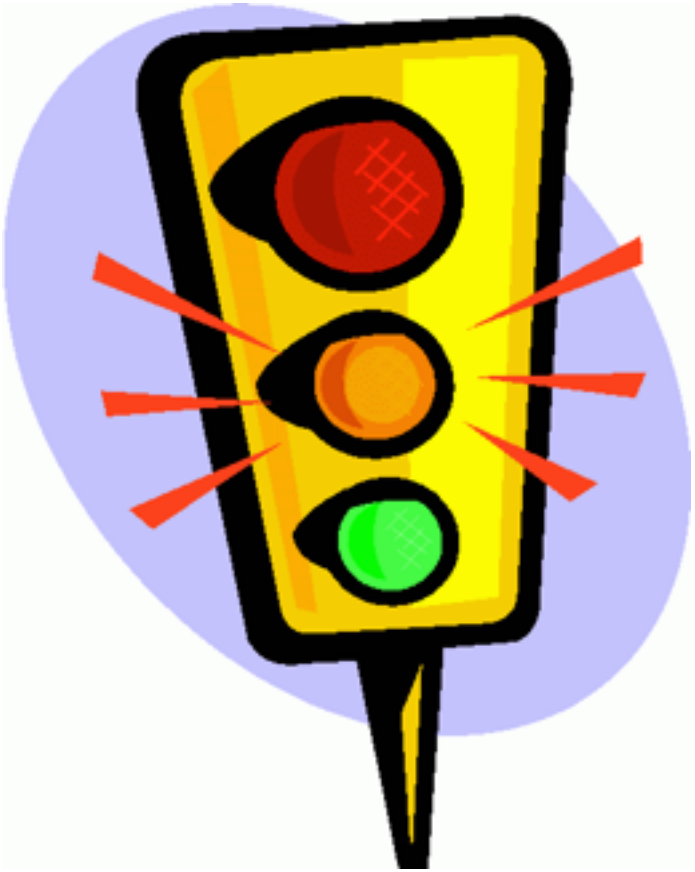
Write Test before Code

Do not write production code until you have written a failing unit test



Write Just Enough Code

You may not write more production code than is sufficient to pass the currently failing test.



Write Just Enough Test

Do not write more of a unit test than is sufficient to fail.

Appendix

UI level testing, **end-to-end functional coverage**	Selenium SeLion (Selemium extention) HP QTP SmartBear TestComplete MS UI Automation
API level testing	RestAPI FitNesse Cucumber
Unit level testing	xUnit TestNG

```
@RunWith(Arquillian.class)
public class InjectionTestCase {
  @Deployment
  public static JavaArchive createTestArchive() {
    return Archives.create("test.jar", JavaArchive.class)
        .addClasses(GreetingManager.class, GreetingManagerBean.class);
  }

  @EJB
  private GreetingManager greetingManager;

  @Test
  public void shouldBeAbleToInjectEJB() throws Exception {
    String userName = "Earthlings";
    Assert.assertEquals(Hello " + userName, greetingManager.greet(userName));
  }
}
```

```
@RunWith(Arquillian.class)
public class InjectionTestCase
{
  @Deployment
  public static JavaArchive createTestArchive() {
    return Archives.create("test.jar", JavaArchive.class)
        .addClasses(GreetingManager.class, GreetingManagerBean.class)
        .addManifestResource(new ByteArrayAsset(new byte[0])),
            ArchivePaths.create("beans.xml"));
  }

  @Inject GreetingManager greetingManager;

  @Inject BeanManager beanManager;

  @Test
  public void shouldBeAbleToInjectCDI() throws Exception {
    String userName = "Earthlings";
    Assert.assertNotNull("Should have the injected the CDI bean manager", beanManager);
    Assert.assertEquals("Hello " + userName, greetingManager.greet(userName));
  }
}
```

```
@Stateless
public class LibraryPersistentBean implements LibraryPersistentBeanRemote {


   @PersistenceContext(unitName="EjbComponentPU")
   private EntityManager entityManager;

   public void addBook(Book book) {
      entityManager.persist(book);
   }

   public List<Book> getBooks() {
      return entityManager.createQuery("From Book").getResultList();
   }
}
```

```
customer = new Customer();
accountManager = mock(AccountManager.class);
customer.setAccountManager(accountManager);
account = mock(Account.class);
when(accountManager.findAccount(customer)).thenReturn(account);
when(accountManager.getBalance(account)).thenReturn(balanceAmount200);
```

mock

```
public class PersonService
{
   private final PersonDao personDao;
   public PersonService( PersonDao personDao )
   {
      this.personDao = personDao;
   }
   public boolean update( Integer personId, String name )
   {
      Person person = personDao.fetchPerson( personId );
      if( person != null )
      {
         Person updatedPerson = new Person( person.getPersonID(), name );
         personDao.update( updatedPerson );
         return true;
      }
      else
      {
         return false;
      }
   }
}
```

- stubs are used to return canned data to your methods or functions under test, so that you can make some assertions on how your code reacts to that data

- Mocks are used to specify certain expectations about how the methods of the mocked object are called by your program: how many times, with how many arguments, etc.

- You should still use a mock library or framework even when you want to use stubs

Unfinished Test Assertion

How do we structure our test logic to avoid leaving tests unfinished?

```
void testSomething() {
    // Outline:
    // create a flight in ... state
    // call the ... method
    // verify flight is in ... state
    fail("Unfinished Test!");
}
```

Want to ensure that we don't accidentally forget to fill in the bodies of these tests if we get distracted.

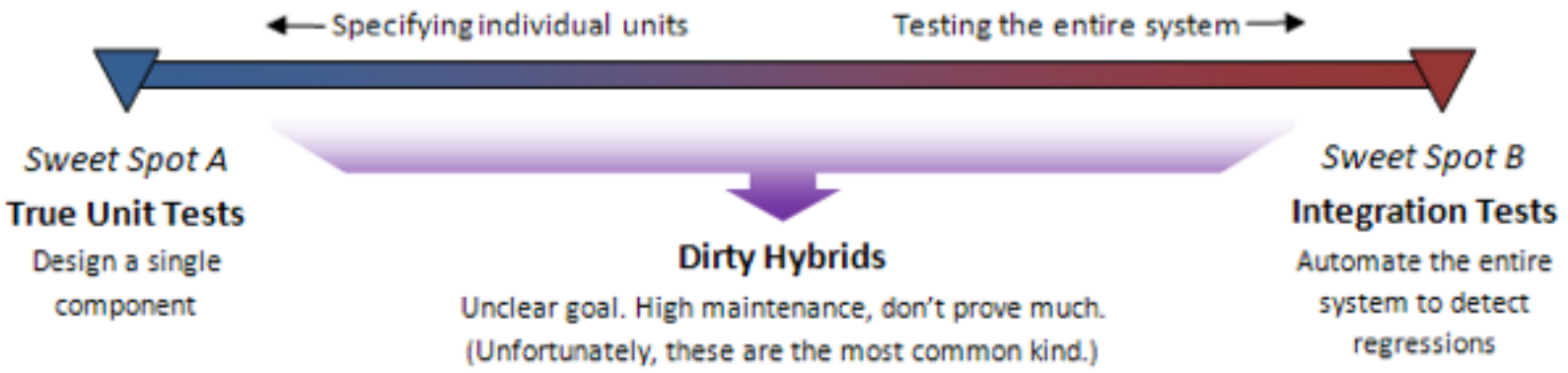
Unfinished Test Assertion

```
[Ignore]
void testSomething() {
    // Outline:
    // create a flight in ... state
    // call the ... method
    // verify flight is in ... state
}
```

Or Exclude the entire *Testcase Class from the AllTests Suite*

Unit Test
Integration Test
Acceptance Test

Unit v/s Integration



Dependency Injection

```
public void test() {
        string[] names= {“kitkat”,”perk”};

            Invoice obj = new Invoice();

            foreach(string name in names)
            {
                obj.AddItem(name);
            }

            List<LineItem> items = obj.getLineItems();

            int i = 0;
            foreach(LineItem item in items)
            {
                assertEquals( item.name, names[i] );
                i++;
            }
}
```

- Arquillian is all about testing your code inside a container.
- embedded containers, which typically run on the same JVM as your test case.
- managed containers, which Arquillian will start up for you during the test process and shut down after the tests are run but run in a different JVM.
- remote containers, which are assumed to be running prior to the test and will simply have deployments sent and tests executed.

- Arquillian uses protocols to capture results of the running test cases.
- Local: This assumes a test case is running in the same JVM and as a result only works with embedded containers
- Servlet 3.0: A protocol that leverages Servlet 3.0 functionality to pass the test results back to the caller
- JMX: This is only supported on JBoss AS 7.
- Arquillian's protocol framework is extensible.

- Each of the containers has its own configuration contained within arquillian.xml.

```
        Person person = new Person( 1, "Phillip" );
        when( personDAO.fetchPerson( 1 ) ).thenReturn( person );

        boolean updated = personService.update( 1, "David" );
        assertTrue( updated );
        verify( personDAO ).fetchPerson( 1 );
        ArgumentCaptor<Person> personCaptor =
ArgumentCaptor.forClass( Person.class );
        verify( personDAO ).update( personCaptor.capture() );
        Person updatedPerson = personCaptor.getValue();
        assertEquals( "David", updatedPerson.getPersonName() );
        // asserts that during the test, there are no other calls to the mock
object.
        verifyNoMoreInteractions( personDAO );
```

the ShrinkWrap
project exists to help dynamically build Java archive files. There are four primary archive types supported in ShrinkWrap: Java Archive (plain JAR files), Web Archive (WAR files), Enterprise Archive (EAR files), and Resource Adapters (RARs).

```
<dependency>
 <groupId>junit</groupId>
 <artifactId>junit</artifactId>
 <scope>test</scope>
</dependency>
```

Law of LSP

Unit test written for base class should work for derived class as well

Everything Is A Mock Anti Pattern

- There should be no need to mock a data structure, dto,

Data Driven Test

```
[Data("file.csv")]
public void test(int x,int y, int res) {

        Math obj = new Math();
        int ans = obj.Add(x,y)
        assertEquals( ans, res);
}
```

passing objects

```
protected Object[] paramsForObjectTest()
{
    return new Object[]
    {
        new Object[]
        {
            new Complex(1, 1), new Complex(2, 2), new Complex(3, 3)
        },
        new Object[]
        {
            new Complex(2, 1), new Complex(4, 2), new Complex(6, 3)
        }
    };
}

@Test
@Parameters(method = &quot;paramsForObjectTest&quot;)
public void checkForwardAddWorksWithyObjects(Complex a, Complex b, Complex expected)
{
    Complex actual = a.add(b);

    assertEquals(expected.getX(), actual.getX(), TOLERANCE);
    assertEquals(expected.getY(), actual.getY(), TOLERANCE);
}
```

Data Driven Test using **JUnitParams**

```
public class ParameterizedMethodDataProvider
{
    public static Object[] providePositiveCases()
    {
        return $(   $(new Complex(1, 1), new Complex(2, 2), new Complex(3, 3)),
                    $(new Complex(2, 1), new Complex(4, 2), new Complex(6, 3)));
    }

    public static Object[] provideNegativeCases()
    {
        return $(   $(new Complex(-1, -1), new Complex(-2, -2), new Complex(-3, -3)),
                    $(new Complex(-2, -1), new Complex(-4, -2), new Complex(-6, -3)));
    }
}

@Test
@Parameters(source = ParameterizedMethodDataProvider.class)
public void checkForwardAddWorksWithClassProvider(Complex a, Complex b, Complex expected)
{
    Complex actual = a.add(b);

    assertEquals(expected.getX(), actual.getX(), TOLERANCE);
    assertEquals(expected.getY(), actual.getY(), TOLERANCE);
}
```