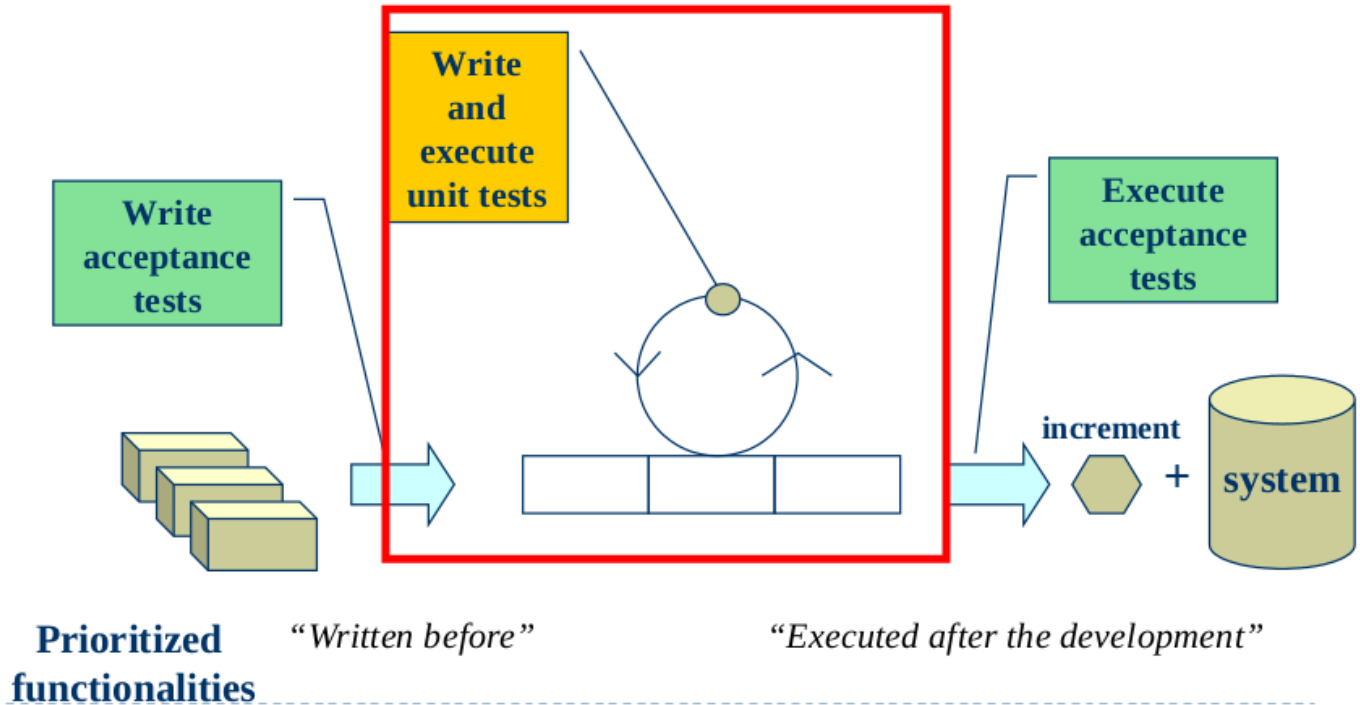
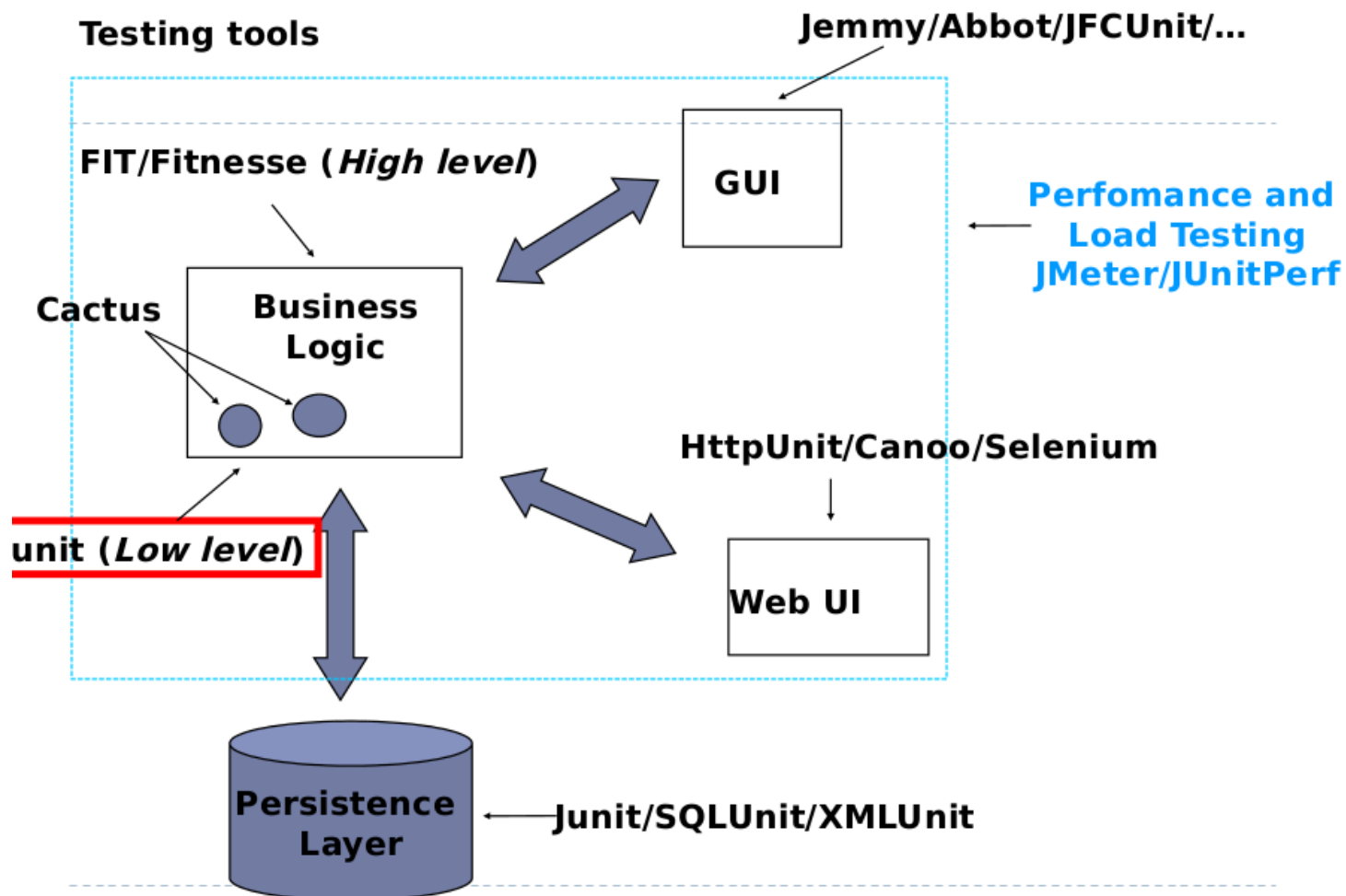


# Iterative Software development

---





## Testing with JUnit

---

- ▶ Junit is a **unit test environment** for Java programs developed by *Erich Gamma* and *Kent Beck*.
  - ▶ Writing test cases
  - ▶ Executing test cases
  - ▶ Pass/fail? (expected result = obtained result?)
- ▶ Consists in a **framework** providing all the tools for testing.
  - ▶ framework: set of classes and conventions to use them.
- ▶ It is **integrated into eclipse** through a graphical plug-in.

Annotation	Description
@Test public void method()	The annotation @Test identifies that a method is a test method.
@Before public void method()	Will execute the method before each test. This method can prepare the test environment (e.g. read input data, initialize the class).
@After public void method()	Will execute the method after each test. This method can cleanup the test environment (e.g. delete temporary data, restore defaults).
@BeforeClass public void method()	Will execute the method once, before the start of all tests. This can be used to perform time intensive activities, for example to connect to a database. Methods annotated with this annotation need a static modifier to work with JUnit.
@AfterClass public void method()	Will execute the method once, after all tests have finished. This can be used to perform clean-up activities, for example to disconnect from a database. Methods annotated with @AfterClass need a static modifier to work with JUnit.
@Ignore	Will ignore the test method. This is useful when the underlying code has been changed and the test case has not yet been adapted. Or if the execution time of this test is too long to be included.
@Test (expected = Exception.class)	Fails, if the method does not throw the named exception.
@Test(timeout=100)	Fails, if the method takes longer than 100 milliseconds.

Statement	Description
fail(String)	Let the method fail. Might be used to check that a certain part of the code is not reached. Or to have failing test before the test code is implemented.
assertTrue(true) / assertTrue(false)	Will always be true / false. Can be used to predefine a test result, if the test is not yet implemented.
assertTrue([message], boolean condition)	Checks that the boolean condition is true.
assertEquals([String message], expected, actual)	Tests that two values are the same. Note: for arrays the reference is checked not the content of the arrays.
assertEquals([String message], expected, actual, tolerance)	Test that float or double values match. The tolerance is the number of decimals which must be the same.
assertNull([message], object)	Checks that the object is null.
assertNotNull([message], object)	Checks that the object is not null.
assertSame([String], expected, actual)	Checks that both variables refer to the same object.
assertNotSame([String], expected, actual)	Checks that both variables refer to different objects.

# Introduction to Unit Testing



Unit testing is a software testing method by which individual units of source code.

“Focus on testing whether a method follows the terms of its *API contract*”

“Confirm that the method accepts the expected range of input and that the returns the expected value for each input”

# Starting from scratch



The simple calculator class

```
public class Calculator {  
    public double add(double number1, double number2) {  
        return number1 + number2;  
    }  
}
```

# A Simple Test Calculator Program

```
public class CalculatorTest {  
    public static void main(String[] args) {  
        Calculator calculator = new Calculator();  
        Double result = calculator.add(10,50);  
        if(result != 60) {  
            System.out.println("Bad result: " + result);  
        }  
    }  
}
```



# A (Slightly) Better Test Program

```
public class CalculatorTest {  
  
    private int nbError = 0;  
  
    public void testAdd() {  
        Calculator calculator = new Calculator();  
        double result = calculator.add(10, 50);  
        if(result != 60) {  
            throw new IllegalStateException("Bad result: " + result);  
        }  
    }  
}
```

# Unit Testing Best Practices



- Always Write Isolated Test Case
- Test One Thing Only In One Test Case
- Use a Single Assert Method per Test Case
- Use a Naming Conventions for Test Cases
- Use Arrange-Act-Assert or Given-When-Then Style

# What is JUnit?



is a simple, open source framework to write and run repeatable tests. It is an instance of the xUnit architecture for unit testing frameworks.

Latest Stable Version: 4.12 (18 April 2016)

# Understanding Unit Testing frameworks

Unit testing frameworks should follow several best practices. These seemingly minor improvements in the *CalculatorTest* program highlight **three** rules that all unit testing frameworks should follow.



Each unit test run independently of all other unit tests.



The framework should detect and report errors test by test.



It should be easy to define which unit tests will run.

# JUnit design goals

The JUnit team has defined three discrete goals for the framework



The framework must help us write useful tests.



The framework must help us create tests that retain their value over time.



The framework must help us lower the cost of writing tests by reusing code.

# Downloading and Installing JUnit



Plain-Old JAR

***ma*ven**

MAVEN

# Testing with JUnit

JUnit has many features that make it easy to write and run tests. You'll see these features at work throughout this example.

- Separate test class instances and class loaders for each unit test to avoid side effect.
- JUnit annotations to provide resource initialization and reclamation methods: `@Before`, `@BeforeClass`, `@After` and `@AfterClass`
- A variety of assert methods to make it easy to check the results of your tests.
- Integration with popular tools like Ant and Maven, and popular IDEs like Eclipse, NetBeans, IntelliJ, and JBuilder

# The JUnit CalculatorTest Program

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

public class CalculatorTest {
    @Test
    public void testAdd() {
        Calculator calculator = new Calculator();
        double result = calculator.add(10,50);
        assertEquals(60, result, 0);
    }
}
```





# Exploring Core JUnit



We need a reliable and repeatable way to test our program.

We need to grow our test as well.

We need to make sure that we can run all of our tests at any time, no matter what code changes took place.

# The Question Becomes ...



How do we run multiple test classes?

How many assert methods provided  
by the JUnit Assert class?

How do we find out which tests  
passed and which ones failed?

# JUnit Assert Method Sample

assertXXX method	What it's used for
<code>assertArrayEquals("message", A, B)</code>	Asserts the equality of the A and B arrays.
<code>assertEquals("message", A, B)</code>	Asserts the equality of object A and B. This assert invokes the equals() method on the first object against the second.
<code>assertSame("message", A, B)</code>	Asserts that the A and B objects are the same object. (like using the == operation)
<code>assertTrue("message", A)</code>	Asserts that the A condition is true.
<code>assertNotNull("message", A)</code>	Asserts that the A object is not null.

# JUnit Core Objects

JUnit Concept	Responsibilities
Assert	Lets you define the conditions that you want to test. An assert method is silent when its proposition succeeds but throws an exception if the proposition fails.
Test	A method with a <code>@Test</code> annotation defined a test. To run this method JUnit constructs a new instance of the containing class and then invokes the annotation method.
Test class	A test class is the container for <code>@Test</code> methods.
Suite	The suite allows you to group test classes together.
Runner	The Runner class runs tests. JUnit 4 is backward compatible and will run JUnit 3 tests.

# Running Parameterized Tests

[...]

@RunWith(value=Parameterized.class)

**public class** ParameterizedTest {

**private double** expected;

**private double** valueOne;

**private double** valueTwo;

    @Parameters

**public static** Collection<Integer[]> getTestParameters() {

        return Arrays.asList(new Integer[][] {

# Running Parameterized Tests (Cont)

```
        {2, 1, 1}, // expected, valueOne, valueTwo
        {3, 2, 1}, // expected, valueOne, valueTwo
        {4, 3, 1}, // expected, valueOne, valueTwo
    });
}

    public ParameterizedTest(double expected, double
valueOne, double valueTwo) {
        this.expected = expected;
        this.valueOne = valueOne;
        this.valueTwo = valueTwo;
    }
```

# Running Parameterized Tests (Cont)

```
@Test
public void sum() {
    Calculator calc = new Calculator();
    assertEquals(expected, calc.add(valueOne, valueTwo), 0);
}
}
```

# Composing Tests With a Suite



JUnit designed the Suite to run one or more test class. The Suite is a container used to gather tests for the purpose of grouping and invocation.



# Composing Tests With a Suite

## Composing a Suite from test classes

```
[...]  
@RunWith(value=org.junit.internal.runners.Suite.class)  
@SuiteClasses(value={FolderConfigurationTest.class, FileConfigurationTest.class})  
public class FileSystemConfigurationTestSuite {  
}
```

We specify the appropriate runner with the `@RunWith` annotation and list the tests we want to include in this test by specifying the test classes in the `@SuiteClasses` annotation. All the `@Test` methods from these classes will be included in the Suite.

# Composing a Suite of Suite

It's possible to create a suite of test suites.

```
[...]
public class TestCaseA {
    @Test
    public void testA1() {
    }
}

[...]
public class TestCaseB {
    @Test
    public void testB1() {
    }
}
```

[...]

@RunWith(value=Suite.class)

@SuteClasses(value={TestCaseA.class})

```
public class TestSuiteA {  
}
```

[...]

@RunWith(value=Suite.class)

@SuteClasses(value={TestCaseB.class})

```
public class TestSuiteB {  
}
```

[...]

@RunWith(value=Suite.class)

@SuteClasses(value={TestSuiteA.class, TestSuiteB.class})

```
public class MasterTestSuite {  
}
```

# Software Testing Principles

## The need for unit tests

The main goal of unit testing is to verify that your application works as expected and to catch bugs early.



Allow greater test coverage

```
11110100010100100111111001110  
0111110000001101111100001100  
011011101000110100011101110  
100110000101011101100000110  
1101010011011111000100010111  
111100101100000111011100011  
100110011011001110110101000  
110110000110000101101010010  
111010000100110100010101111  
0000100110100100010000011010  
0110011010011101110110011000  
110110110111001001111100100  
000010011001100000110101010  
00110001110010000111101110  
010000110100111011100001010  
0001001110011001100111011
```

Detect regressions and limiting debugging



Increasing team productivity



Refactoring with confidence

# Software Testing Principles (Const)

## The need for unit tests (Cont)



Improving Implementation



Documenting expected  
behaviour



Enabling code coverage  
and other metrics

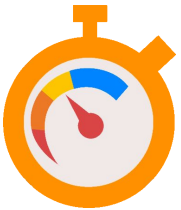
# The Four Type of Software Tests



Integration tests



Functional tests



Stress and load tests



Acceptance tests

# Integration Tests

Individual unit tests are essential to quality control, but what happens when different units of works are combined into workflow?



Just a more traffic collisions occur at intersections, the point where objects interact are major contributors of bugs.

Ideally, You should defined integration tests before.

# Functional Tests

Functional tests examine the code at the boundary of its public API. In general, this corresponds to testing application use cases.



A web application contains a secure web page that only authorized client can access. If the client doesn't log in, then redirect to the login page.



# Stress and Load Tests

How well will the application perform when many people are using it at once?



Most stress tests examine whether the application can process a large number of requests within a given good period.

# Acceptance Tests

It's important that an application perform well, but the application must also meet the customer's needs. The customer or a proxy usually conducts acceptance tests to ensure that application has met whatever goals the customer or stakeholder defined.

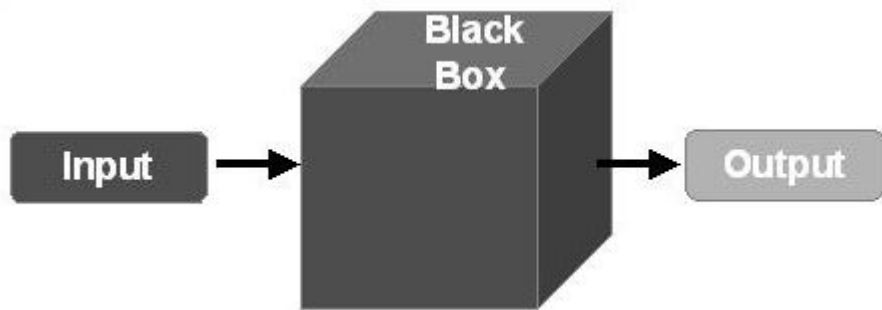


Acceptance Test are our final level of Testing

# The Three Type of Unit Tests

Test type	Description
Logic unit test	A test that exercises code by focusing on a single method. You can control the boundaries of a given test method by using mock object or stub.
Integration unit test	A test the focuses on the interaction between component in their real environment. For example, code that accesses a database.
Functional unit test	A test that extends the boundaries of integration unit testing to conform a stimulus response. For example, a web application contains a secure web page that only authorized clients can access. If client doesn't login, then redirect to the login page.

# Black Box versus White Box Testing

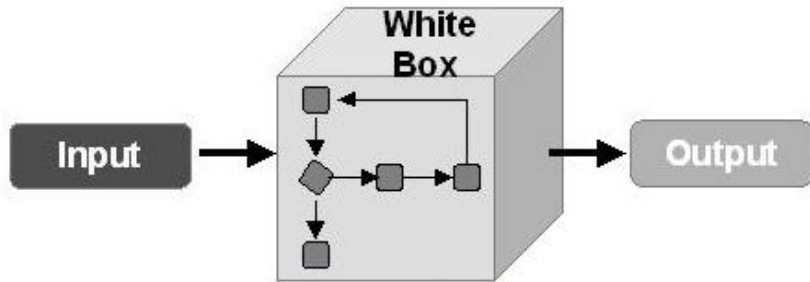


A black box test has no knowledge of the internal state or behavior of the system.

All we know is that by providing correct input, the system produces the desired output.

All we need to know in order to test the system is system's functional specification.

# Black Box versus White Box Testing

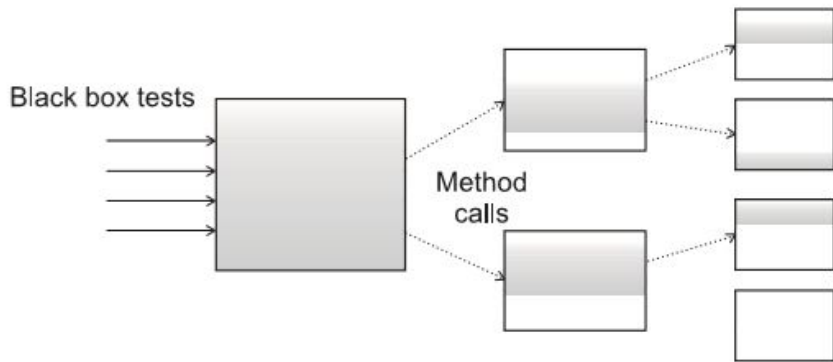


A white box test, In contrast to black box testing, We use detailed knowledge of the implementation to create tests and drive the testing process.

Not only is knowledge of a component's implementation required, but also of how it interacts with other components.

# Test Coverage and Development

Writing unit tests gives you the confidence to change and refactor an application. As you make changes, you run tests, which gives you immediate feedback on new features under test and whether your changes break existing tests. The issue is that these changes may still break existing untested functionality.



Partial test coverage with black box tests.

In order to resolve this issue, we need to know precisely code runs when you or the build invokes tests.

**Ideally, our tests should cover 100 percent of your application code.**

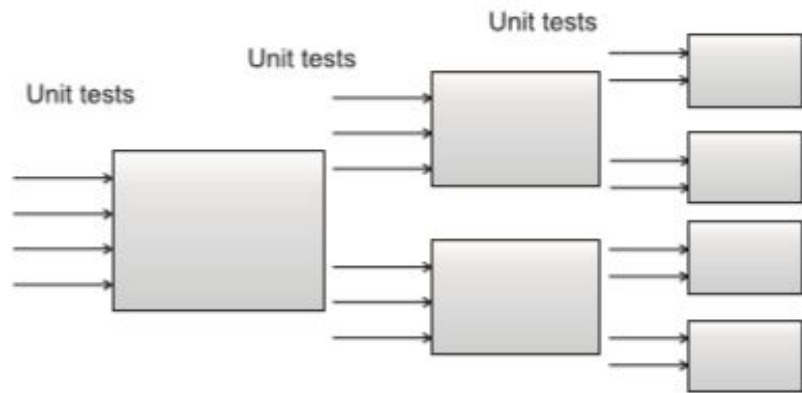
# Introduction to test coverage

Using black box testing, we can create tests that cover the public API of an application. Because we're using documentation as our guide and not knowledge of the implementation, we don't create tests.

One metric of test coverage would be to track which methods the tests call. This doesn't tell you whether the tests are complete, but it does tell you if you have a test for a method.

# Introduction to test coverage (Cont)

You can write a unit test with intimate knowledge of a method's implementation. If a method contains branch, you can write two unit tests, once for each branch. Because you need to see into the method to create such a test, this falls under white box testing.



Complete test coverage using white box tests.


You can achieve higher test coverage using white box unit tests because you have access to more methods and because you can control both the input to each method



# Introduction to Cobertura

Cobertura is a code coverage tool that integrates with JUnit.

## Coverage Report - com.stream.Calculator

Classes in this File	Line Coverage	Branch Coverage	Complexity
<a href="#">Calculator</a>	66%  2/3	N/A  N/A	1

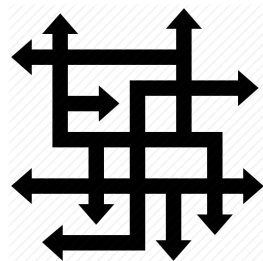
1	package com.stream;
2	
3	5 public class Calculator {
4	
5	public int squareRoot(int i) {
6	5 return (int) Math.sqrt(i);
7	}
8	
9	public int sum(int a, int b){
10	0 return a + b;
11	}
12	
13	}

# Coarse-Grained Testing with Stubs

Stubs are a mechanism for faking the behavior of real code or code that isn't ready yet. Stubs allow you to test a portion of a system even if the other part isn't available.



Integration testing between different subsystem.



When you can't modify an existing system because it's too complex and fragile.

# Testing with Mock Objects

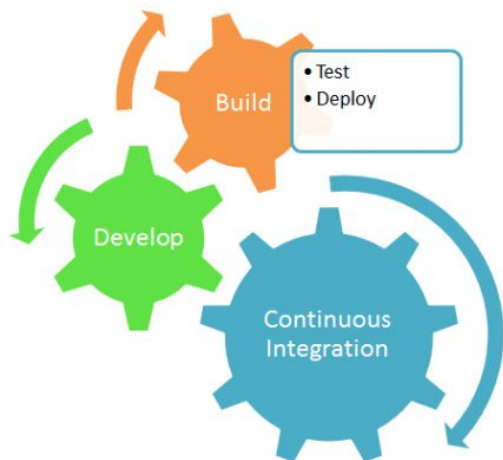
Testing in isolation offers strong benefits, such as the ability to test that has not yet been written (as long as you at least have an interface to work with)

The biggest advantage is ability to write focused tests that test only a single method, without side effects resulting from other objects being call from the method under test.

Mock objects are perfectly suited for testing a portion of code login in isolation from the rest of the code. Mocks replace the objects with which your methods under test collaborate.

# Continuous Integration Tools

Integration tests should be executed independently from the development process. The best way is to execute them at a regular interval (say, 15 minutes). This way, if something gets broken, you'll hear about it within 15 minutes, and there's a better chance for you to fix it.



**Continuous Integration is all about  
Team software development**