Unit Testing

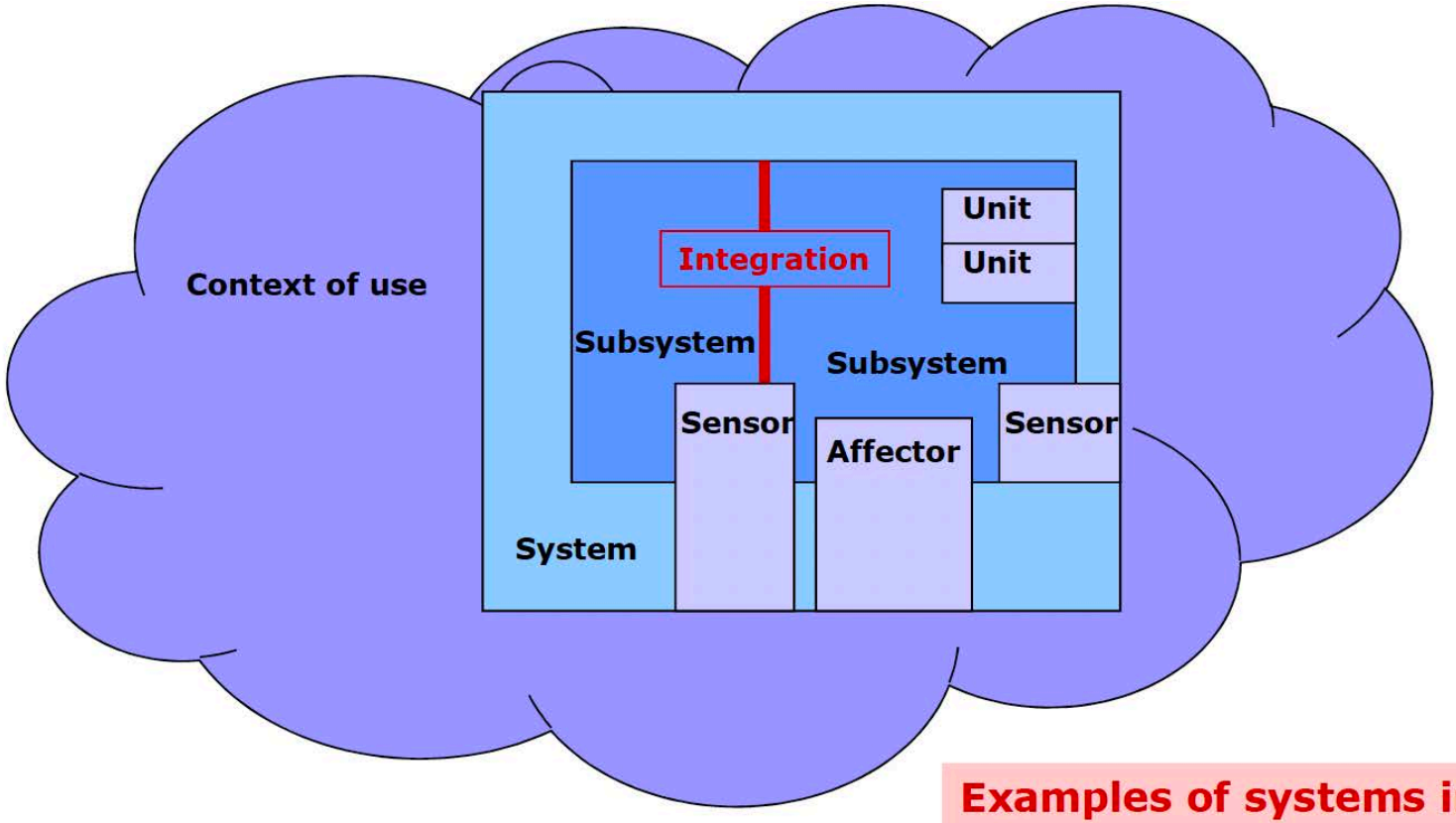
Spring, 2023

Yi Xiang xiangyi@scut.edu.cn

Contents

- What do we test?
- Unit Testing
- Test-Driven Development (TDD)
- Automated Unit Testing

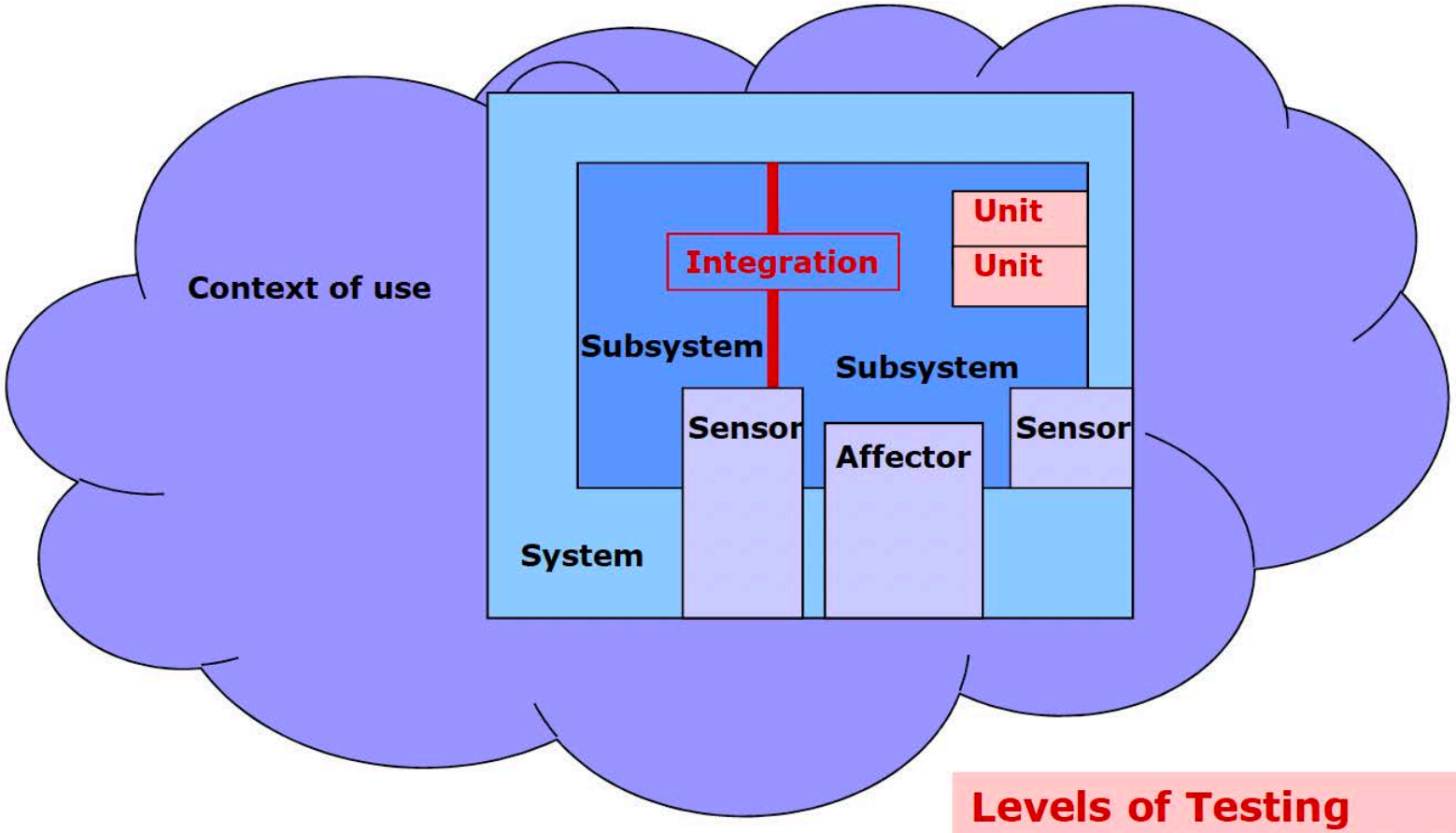
1. What do we test? - – the Focus of Concern



Examples of systems in context

- Mars rover
- Cell phone
- Clothes washing machine
- Point of sale system
- Telecom switch
- Software development tool

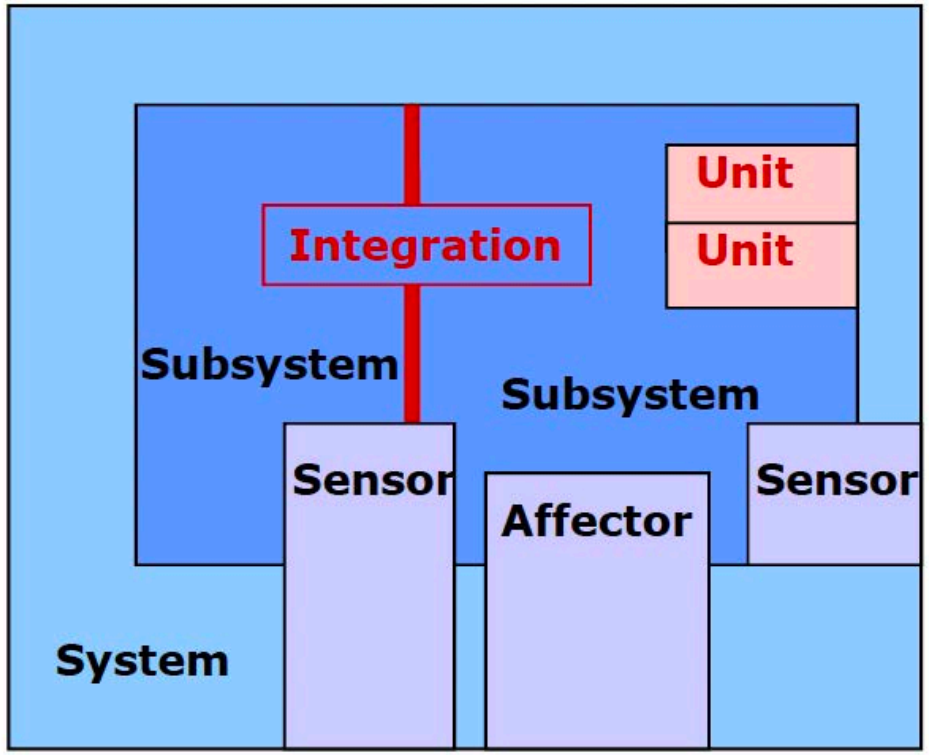
What do we test? - – the Focus of Concern



Levels of Testing

- User testing, field testing

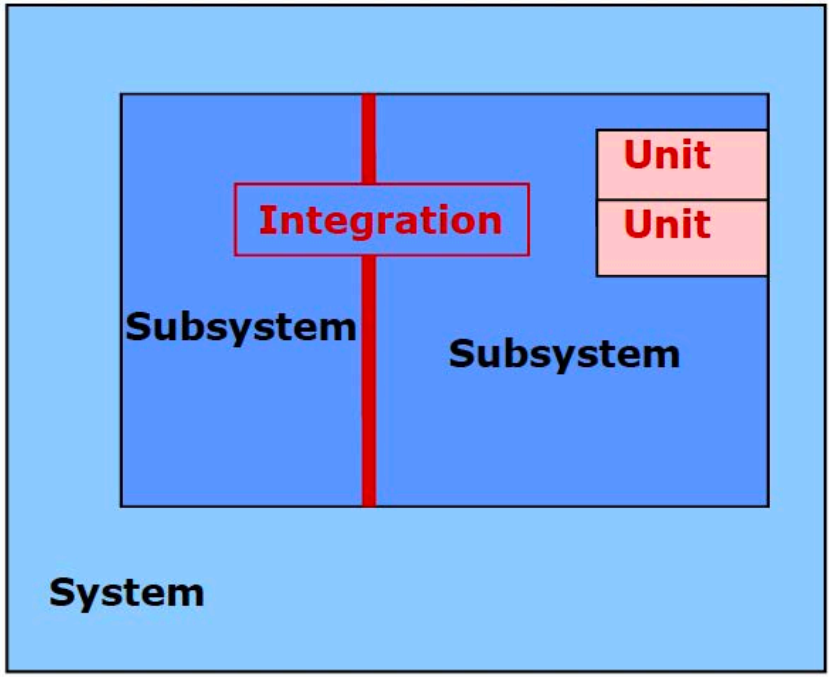
What do we test? - – the Focus of Concern



Levels of Testing

- User testing, field testing
- System testing

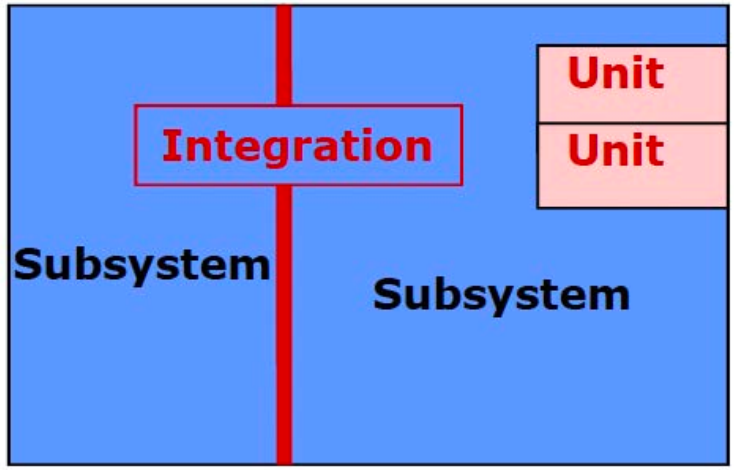
What do we test? - – the Focus of Concern



Levels of Testing

- User testing, field testing
- System testing
  - With or without hardware

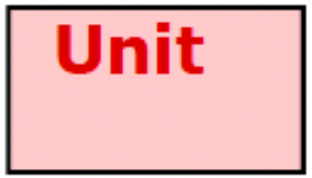
What do we test? - – the Focus of Concern



Levels of Testing

- User testing, field testing
- System testing
  - With or without hardware
- Integration testing

What do we test? - – the Focus of Concern



Levels of Testing

- User testing, field testing
- System testing
  - With or without hardware
- Integration testing
- Unit testing

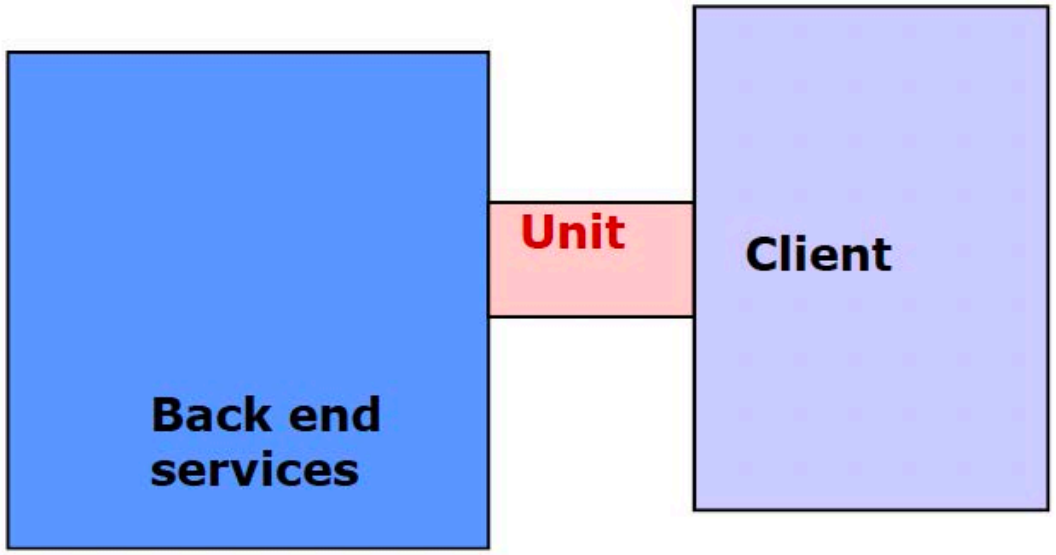
2. Unit Testing

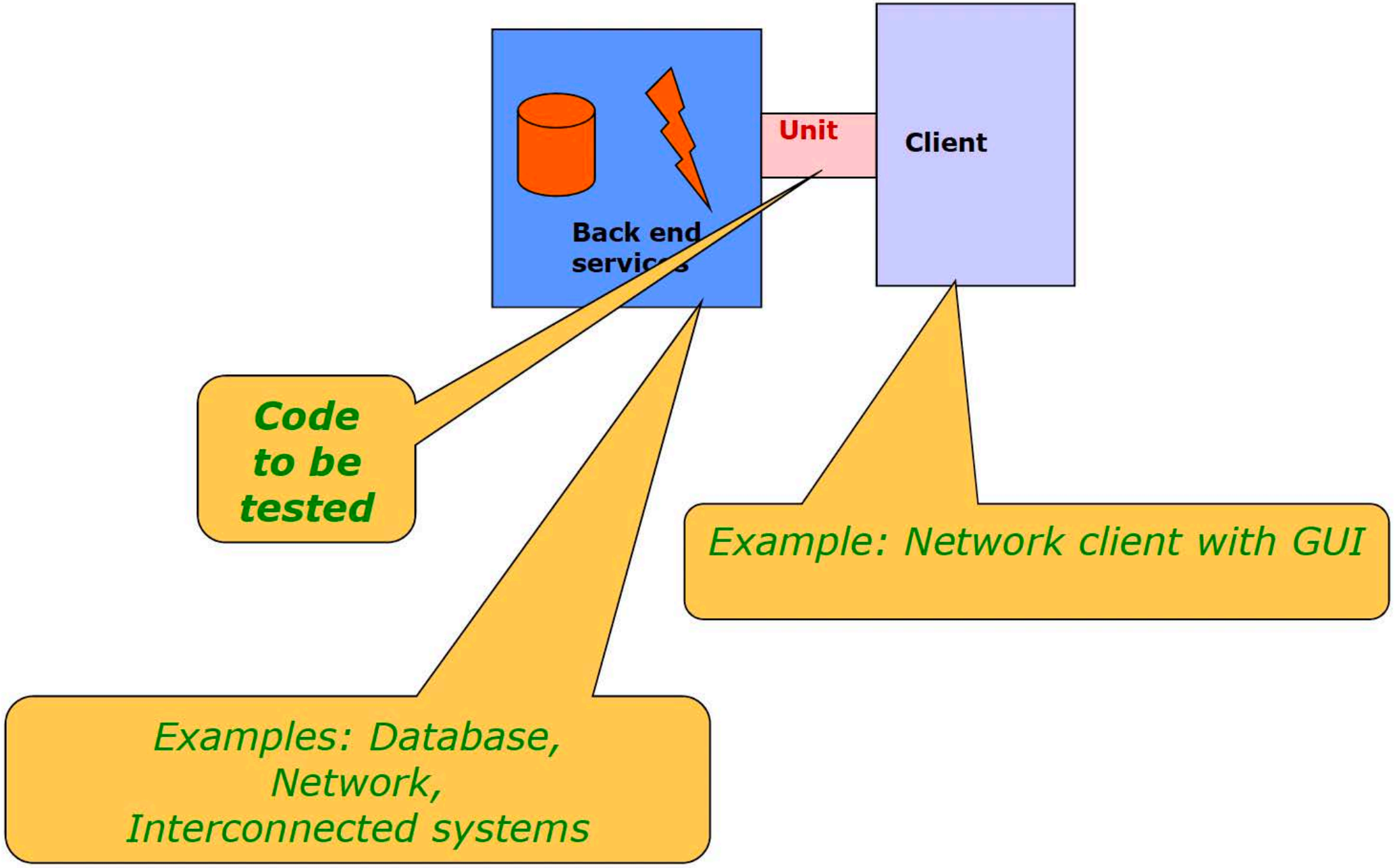
- Unit tests are mainly whitebox tests written by developers, and designed to verify small units of program functionality.
  - Key Metaphor: I.C. Testing
    Integrated Circuits are tested individually for functionality before the whole circuit is tested.
  - Definitions
    Whitebox – Unit tests are written with full knowledge of implementation details.
    Developers – Unit tests are written by you, the developer, concurrently with implementation.
    Small Units – Unit tests should isolate one piece of software at a time.
  - Individual methods and classes
    Verify – Make sure you built 'the software right.' Testing against the contract.
  - Contrast this with validation

Role of Unit Testing

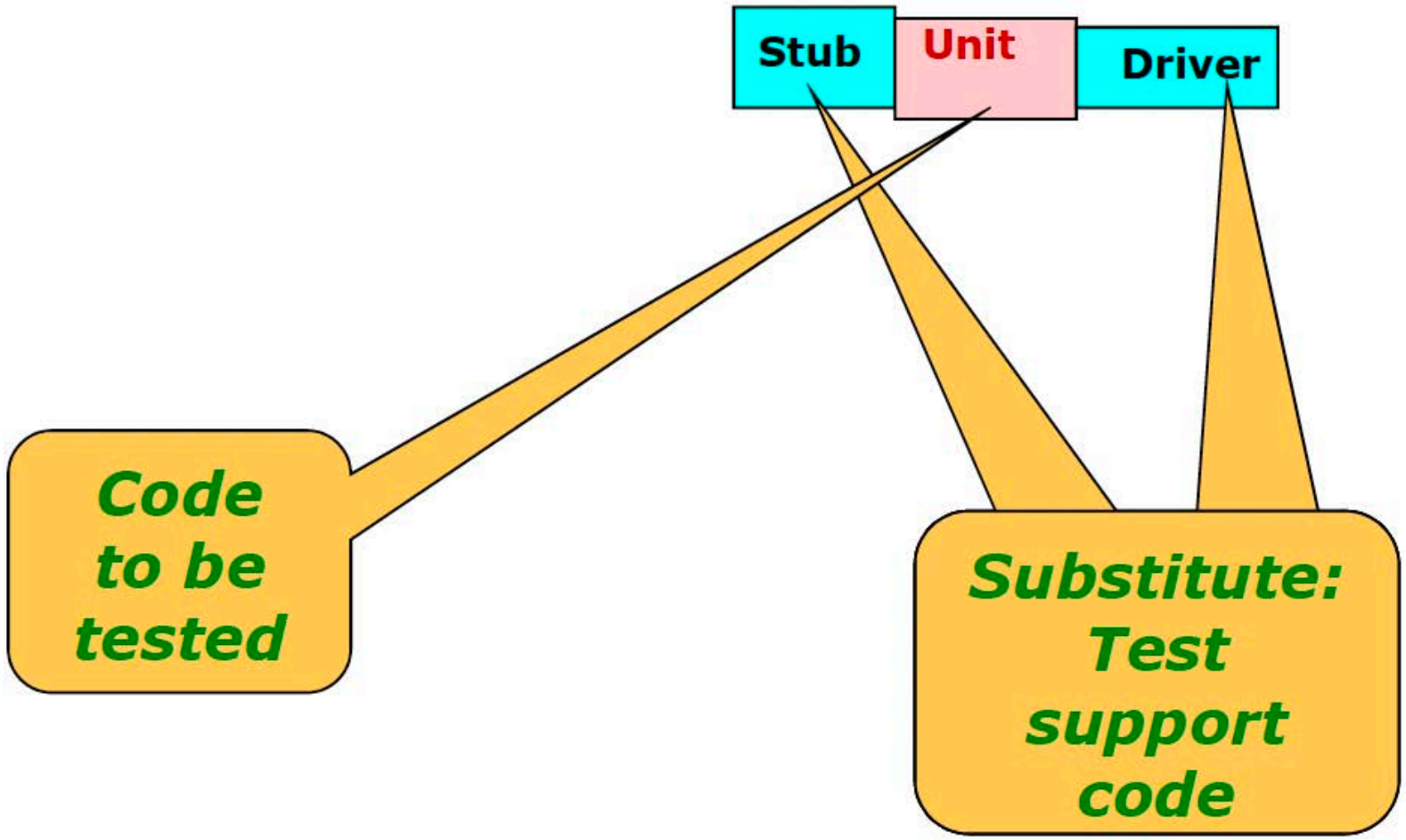
- Helps localize errors
  - Failure indicates problem in the unit under test
- Find errors early
  - Unit tests are written during development, usually by developer
  - More expensive to fix defects found later by another team
- Avoid unnecessary functionality
  - Write test first, only write enough code to get it working
- Improve code quality code
  - Helps developer deliver working code
  - Assure minimum quality of units before integration into system

Unit Test and Scaffolding




Unit
Client
Back end
Code
to be
tested
Example: Network client with GUI
Examples: Database,
Network,
Interconnected systems

Unit Test and Scaffolding



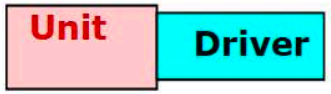
Techniques for Unit Testing : Scaffolding

- Use "scaffold" to simulate external code

- External code – scaffold points
  1. Client code
  2. Underlying service code

1. Client API
   - Model the software client for the service being tested
   - Create a **test driver**
   - Object-oriented approach:
     - Test individual calls and sequences of calls

Testers write driver code



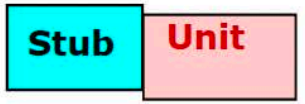
Techniques for Unit Testing : Scaffolding

- Use "scaffold" to simulate external code

- External code – scaffold points
  1. Client code
  2. Underlying service code

2. Service code
   - Underlying services
     - Communication services
       - Model behavior through a communications interface
       - Database queries and transactions
     - Network/web transactions
     - Device interfaces
       - Simulate device behavior and failure modes
     - File system
       - Create file data sets
       - Simulate file system corruption
     - *Etc*
   - Create a set of **stub** services or **mock** objects
     - *Minimal* representations of APIs for these services

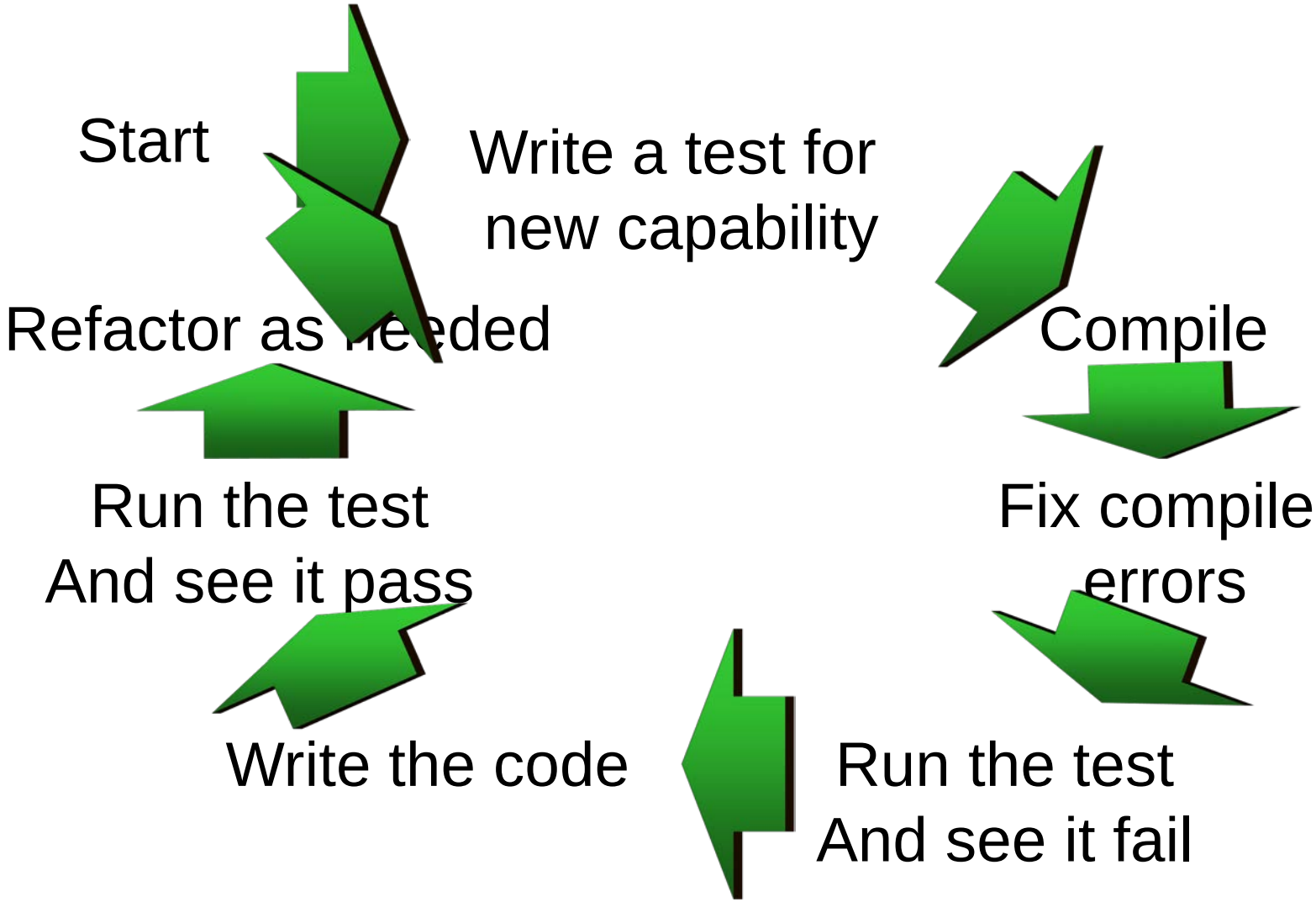
Testers write stub code



3. Test-Driven Development

- Write the tests before the code
  - Helps you think about corner cases when writing
  - Helps you think about interface design
- Write code only when an automated test fails
- If you find a bug through other means, first write a test that fails, then fix the bug
  - Bug won't resurface later
- Run tests as often as possible, ideally every time the code is changed
  - Having comprehensive unit tests allows you to refactor code with confidence
  - Without unit tests, code is fragile – changes might break clients!

TDD - *Test-Driven Development*



Test-Driven Development

- An excellent practice promoted by the iterative and agile XP method , also known as test-first development

- Advantages
  - The unit tests actually get written
  - Programmer satisfaction leading to more consistent test writing
  - Clarification of detailed interface and behavior
  - Provable, repeatable, automated verification
  - The confidence to change things

Test-Driven Development

- The most popular unit testing framework is the xUnit family
  JUnit for java, NUnit for .NET, and so forth.

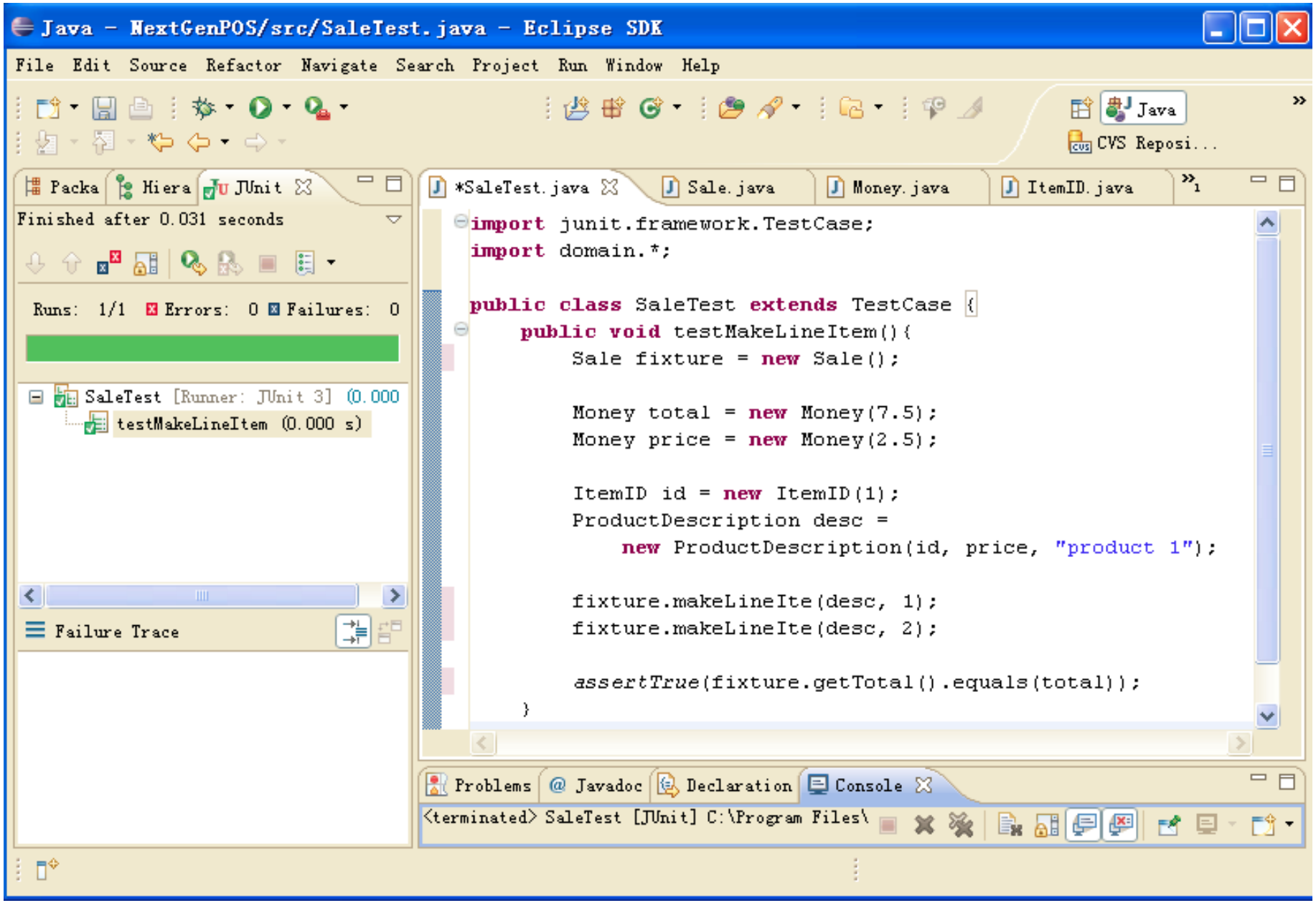
- Example: using JUnit and TDD to create the Sale class.

Before programming the Sale class, we write a unit testing method in a SaleTest class that does the following

- Create a Sale
- Add some line items to it with the makeLineItem method
- Ask for the total and verify that it is the expected value
- Each testing method follows this pattern
- Create the fixture.
- Do something to it (some operation that you want to test).
- Evaluate that the results are as expected.

Test-Driven Development - Example

```
import junit.framework.TestCase;
import domain.*;

public class SaleTest extends TestCase {
    public void testMakeLineItem(){
        Sale fixture = new Sale();

        Money total = new Money(7.5);
        Money price = new Money(2.5);

        ItemID id = new ItemID(1);
        ProductDescription desc =
                new ProductDescription(id, price, "product 1");

         fixture.makeLineIte(desc, 1);
         fixture.makeLineIte(desc, 2);

        assertTrue(fixture .getTotal().equals(total));
    }
}
```



IDE Support for TDD and xUnit

Refactoring

- Continuously refactoring code is another XP practice and applicable to all iterative methods
  - A structured, disciplined method to rewrite or restructure existing code without changing its external behavior .
  - Applying small transformation steps combined with re-executing tests each step.

- The essence of refactoring is applying small behavior preserving transformations (each called a ‘refactoring’), one at a time .

- After each transformation, the unit tests are re-executed to prove that the refactoring did not cause a regression （failure).

The Activities and Goals of Refactoring

- They are simply the activities and goals of good programming
  - Remove duplicate code
  - Improve clarity
  - Make long methods shorter
  - Remove the use of hand-coded literal constants
  - And more …
- Some code smells include:
  - Duplicated code
  - Big method
  - Class with many instance variables
  - Class with lots of code
  - Strikingly similar subclasses
  - high coupling between many objects
  - And so many other ways bad code is written …

Refactorings

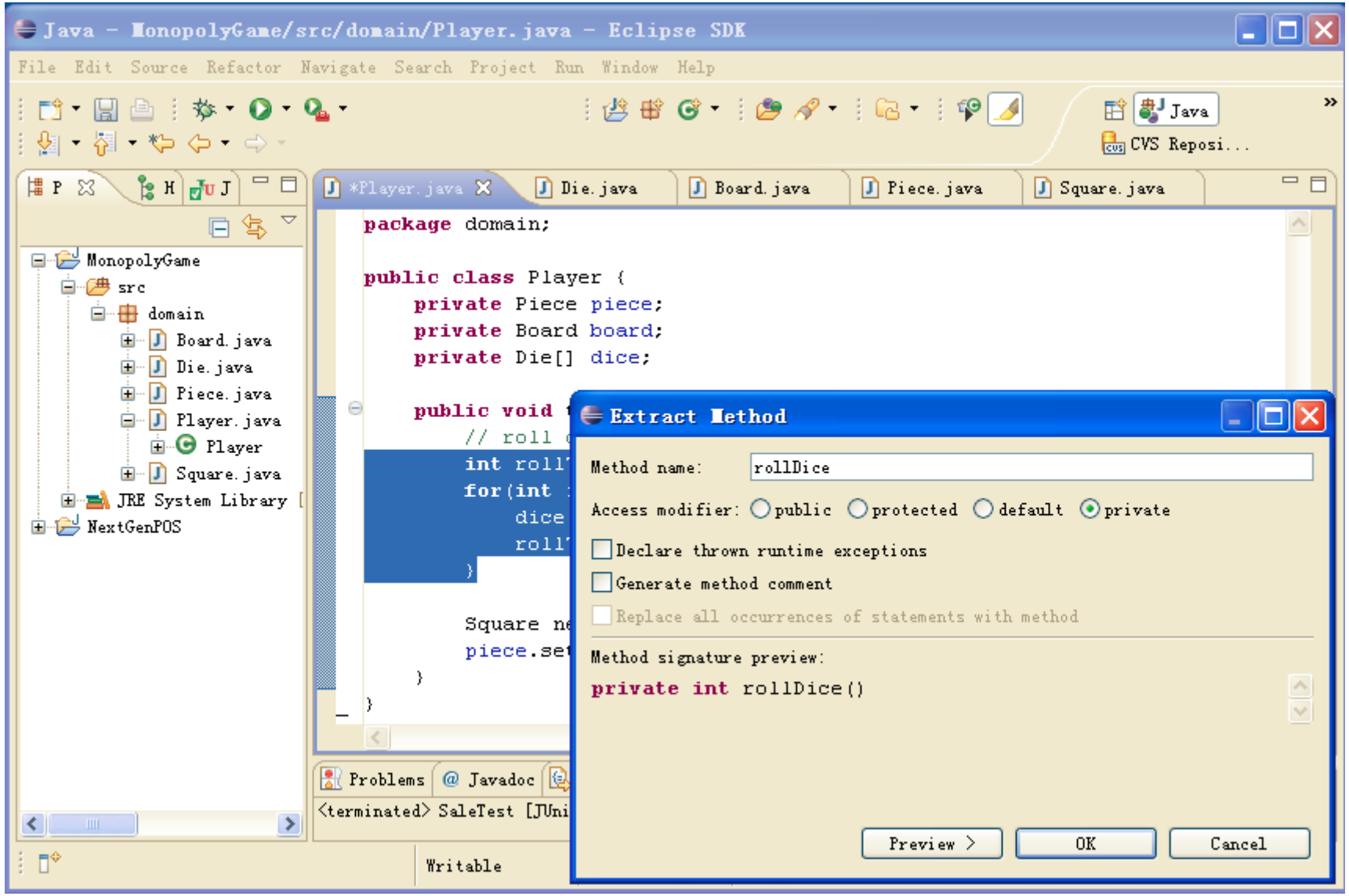
Refactoring	Description
Extract Method	Transform a long method into a shorter one by factoring out a portion into a private helper method.
Extract Constant	Replace a literal constant with a constant variable.
Introduce Explaining Variable (specialization of Extract Local Variable)	Put the result of the expression, or parts of the expression, in a temporary variable with a name that explains the purpose.
Replace Constructor Call with Factory Method	In Java, for example, replace using the *new* operator and constructor call with invoking a helper method that creates the object (hiding the details).

Martin Fowler《Refactoring: Improving the Design of Existing Code》

Example 1: Monopoly --The takeTurn method before refactoring

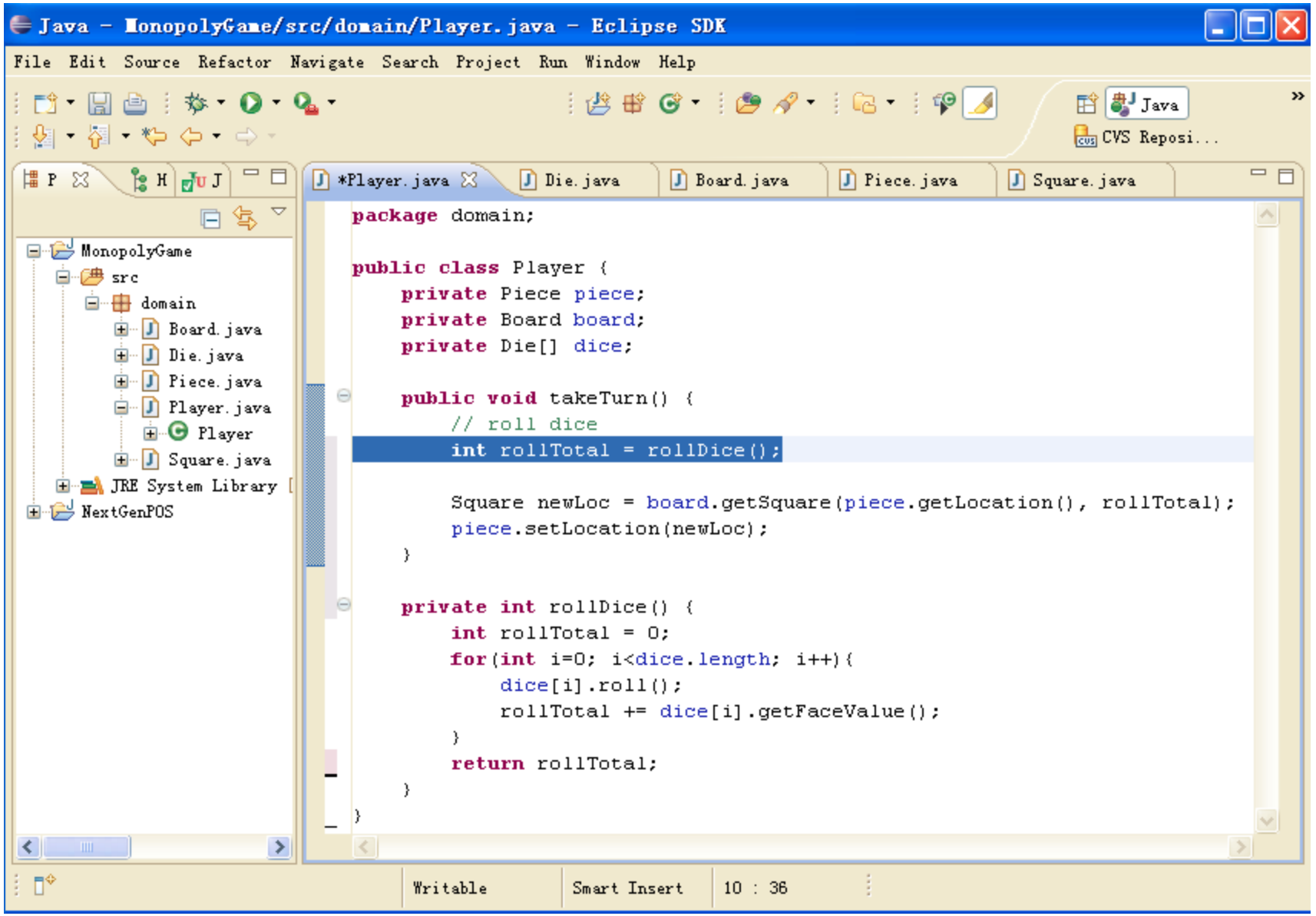
```
public class Player
{
   private Piece  piece;
   private Board  board;
   private Die[]  dice;
   // …

public void takeTurn()
{
        // roll dice
   int rollTotal = 0;
   for (int i = 0; i < dice.length; i++)
   {
      dice[i].roll();
      rollTotal += dice[i].getFaceValue();
   }

   Square newLoc = board.getSquare(piece.getLocation(), rollTotal);
   piece.setLocation(newLoc);

}

} // end of class
```

Refactor -->Extract Method...



IDE Support for Refactoring

The takeTurn method after refactoring



IDE Support for Refactoring

Example 2: Introduce Explaining Variable

- Before

```
   // good method name, but the logic of the body is not clear
boolean isLeapYear( int year )
{
   return( ( ( year % 400 ) == 0 ) ||
         ( ( ( year % 4 ) == 0 ) && ( ( year % 100 ) != 0 ) ) );
}
```

- After

```
      // that's better!
   boolean isLeapYear( int year )
   {
      boolean isFourthYear = ( ( year % 4 ) == 0 );
      boolean isHundrethYear = ( ( year % 100 ) == 0);
      boolean is4HundrethYear = ( ( year % 400 ) == 0);
      return (
         is4HundrethYear
         || ( isFourthYear && ! isHundrethYear ) );
   }
```

4. Automated Unit Test

- Testing Framework are tools that help manage and run your unit tests.
- xUnit Framework： JUnit(java), CppUnit(c++), NUnit(.Net)

- Help achieve three properties of good unit tests:
  - Automatic

    Tests should be easy to run and check for correct completion.
- This allows developers to quickly confirm their code is working after a change.
  - Repeatable

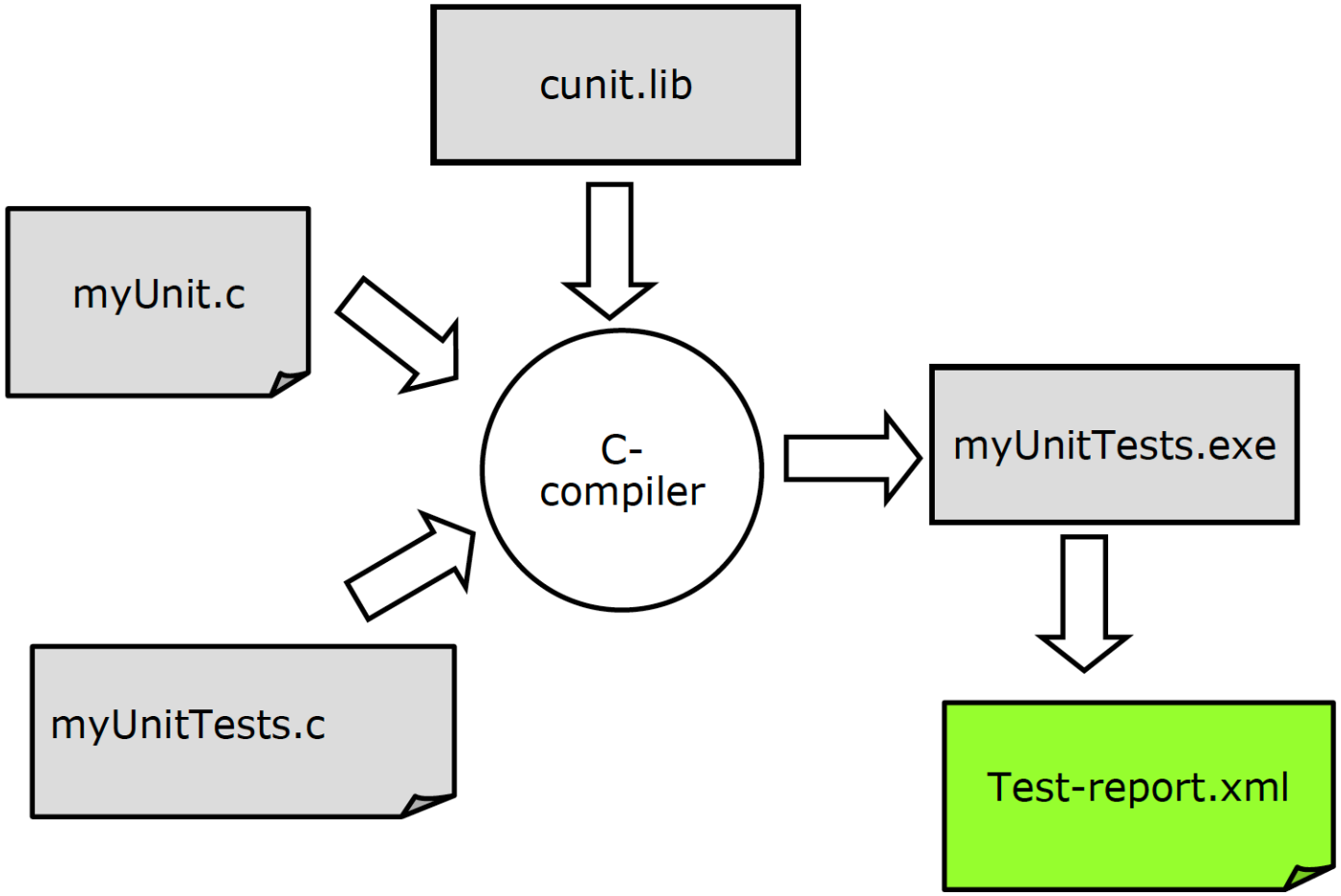
    Any developer can run the tests and they will work right away.
  - Independent

    Tests can be run in any order and they will still work.

What is xUnit?

- A set of “Frameworks” for programming and automated execution of test-cases
- X stands for programming language
  - Most Famous is J-UNIT for Java
  - But exists for almost all programming languages
  - C-unit, Cpp-Unit, DUnit, JUnit NUnit, …
- A framework is a collection of classes, procedures, and macros

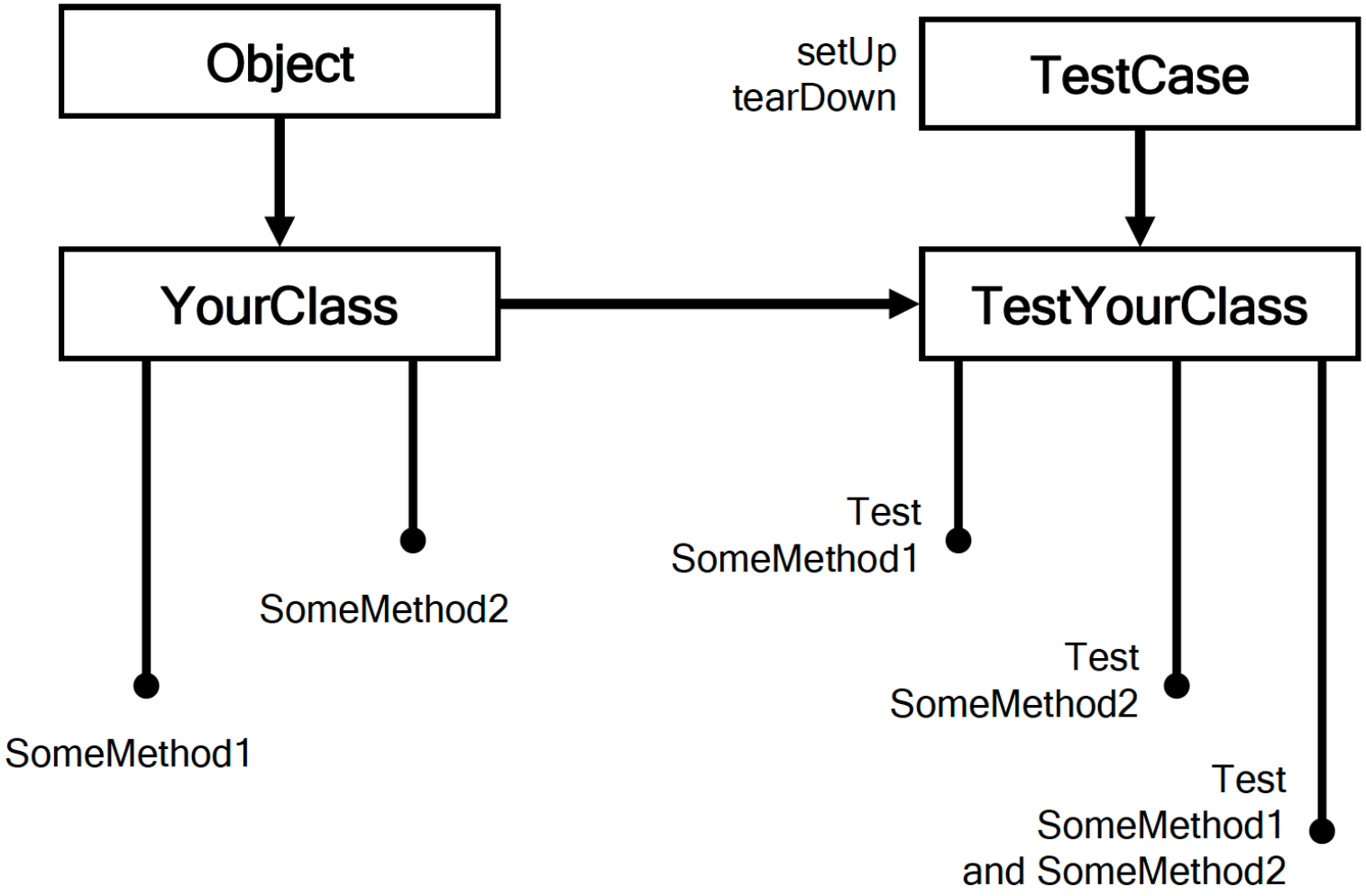
Basic Use of FrameWork



Concepts

- Assertions
  - Boolean expression that compares expected and actual results
  - The basic and smallest building-block
  - General: **ASSERT** (expected, actual)
- Test Case
  - A class that extends "TestCase"s
  - A composition of concrete test procedures
  - May contain several assertions and test for several test objectives
  - E.g all test of a particular function
- Test Suite
  - Collection of related test cases
  - Can be executed automatically in a single command

xUnit

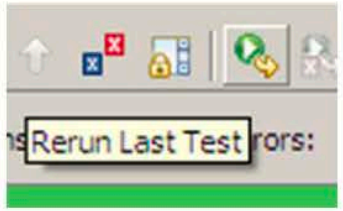

Object
YourClass
SomeMethod1
SomeMethod2
setUp
tearDown
TestCase
TestYourClass
Test
SomeMethod1
Test
SomeMethod2
Test
SomeMethod1
and SomeMethod2

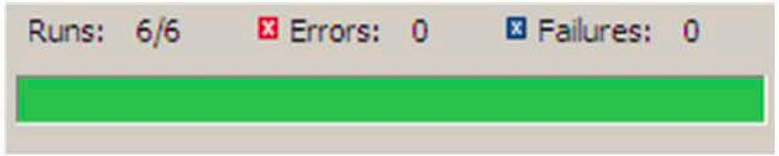
JUnit: A Java Unit Testing Framework

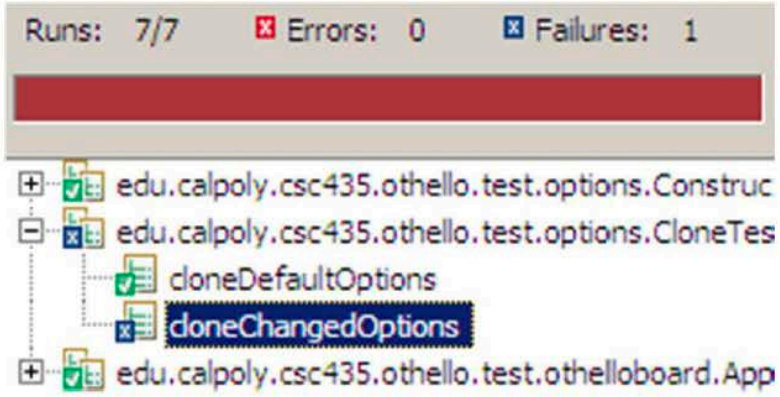
- Features
  - One click runs all tests
  - Visual confirmation of success or failure.
  - Source of failure is immediately obvious.

- JUnit framework interface
  - `@Test` annotation marks a test for the harness
  - `org.junit.Assert` contains functions to check results.

http://www.junit.org/







A JUnit Test Case

```
public class SampleTest  {
    private List<String> emptyList;

    @Before
    public void setUp() {
        emptyList = new ArrayList<String>();
    }

    @After
    public void tearDown() {
        emptyList = null;
    }

    @Test
    public void testEmptyList() {
        assertEquals("Empty list should have 0 elements",
                0, emptyList.size());
    }
}
```

Helpful JUnit Assert Statements

- `assertTrue(boolean condition)`
- `assertFalse(boolean condition)`
  - Assert some condition is true (or false)

- `assertEquals(Object expected, Object actual)`
  - Check that some value is equal to another

- `assertEquals(float expected, float actual, float delta)`
  - Used for so that floating point equality is unnecessary.

- `assertSame(Object expected, Object actual)`
  - Tests for two objects are the same reference (identical) in memory.

- `assertNull(java.lang.Object object)`
  - Asserts that a reference is null.

- `assertNotNull(String message, Object object)`
  - Many 'not' asserts exists.
  - Most asserts have an optional message that can be printed.

AssertionsTest.java

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

public class AssertionsTest {

        @Test
        public void test() {
                String obj1 = "junit";
                String obj2 = "junit";
                String obj3 = "test";
                String obj4 = "test";
                String obj5 = null;
                int var1 = 1;
                int var2 = 2;
                int[] arithmetic1 = { 1, 2, 3 };
                int[] arithmetic2 = { 1, 2, 3 };

                assertEquals(obj1, obj2);

                assertSame(obj3, obj4);

                assertNotSame(obj2, obj4);

                assertNotNull(obj1);

                assertNull(obj5);

                assertTrue(var1 < var2);

                assertArrayEquals(arithmetic1, arithmetic2);
        }

}
```

Other Helpful JUnit Features

- @BeforeClass
  - Run once before all test methods in class.
- @AfterClass
  - Run once after all test methods in class.

- Together, these methods are used for setting up computationally expensive test elements.
  - E.g., database, file on disk, network…

- @Before
  - Run before each test method.
- @After
  - Run after each test method.

- Make tests independent by setting and resetting your testing environment.
  - E.g., creating a fresh object

- @Test(expected=ParseException.**class**)
  - When you expect an exception

@BeforeClass –>

@Before –> @Test –> @After

–> @AfterClass

AnnotationsTest.java

```
import static org.junit.Assert.*;
import java.util.*;
import org.junit.*;

public class AnnotationsTest {

        private ArrayList testList;

        @BeforeClass
        public static void onceExecutedBeforeAll() {
                System.out.println("@BeforeClass: onceExecutedBeforeAll");
        }

        @Before
        public void executedBeforeEach() {
                testList = new ArrayList();
                System.out.println("@Before: executedBeforeEach");
        }

        @AfterClass
        public static void onceExecutedAfterAll() {
                System.out.println("@AfterClass: onceExecutedAfterAll");
        }

        @After
        public void executedAfterEach() {
                testList.clear();
                System.out.println("@After: executedAfterEach");
        }
```

```
        @Test
        public void EmptyCollection() {
                assertTrue(testList.isEmpty());
                System.out.println("@Test: EmptyArrayList");

        }

        @Test
        public void OneItemCollection() {
                testList.add("oneItem");
                assertEquals(1, testList.size());
                System.out.println("@Test: OneItemArrayList");
        }

        @Ignore
        public void executionIgnored() {

                System.out.println("@Ignore: This execution is ignored");
        }
}
```

```
@BeforeClass: onceExecutedBeforeAll
@Before: executedBeforeEach
@Test: EmptyArrayList
@After: executedAfterEach
@Before: executedBeforeEach
@Test: OneItemArrayList
@After: executedAfterEach
@AfterClass: onceExecutedAfterAll
```

Junit Suite Testing

PrepareMyBagTest.java

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

public class PrepareMyBagTest {

        FirstDayAtSchool school = new FirstDayAtSchool();

        String[] bag = { "Books", "Notebooks", "Pens" };

        @Test
        public void testPrepareMyBag() {

                System.out.println("Inside testPrepareMyBag()");
                assertArrayEquals(bag, school.prepareMyBag());

        }

}
```

Junit Suite Testing

AddPencilsTest.java

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

public class AddPencilsTest {

        FirstDayAtSchool school = new FirstDayAtSchool();

        String[] bag = { "Books", "Notebooks", "Pens", "Pencils" };

        @Test
        public void testAddPencils() {

                System.out.println("Inside testAddPencils()");
                assertArrayEquals(bag, school.addPencils());

        }

}
```

Junit Suite Testing

SuiteTest.java

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({ PrepareMyBagTest.class, AddPencilsTest.class })
public class SuitTest {

}
```

- @ Suite.SuiteClass;
- Run As -> JUnit Test

Junit Parameterized Testing

CalculateTest.java

```
import static org.junit.Assert.assertEquals;
import java.util.Arrays;
import java.util.Collection;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;

@RunWith(Parameterized.class)
public class CalculateTest {

        private int expected;
        private int first;
        private int second;

        public CalculateTest(int expectedResult, int firstNumber, int secondNumber) {
                this.expected = expectedResult;
                this.first = firstNumber;
                this.second = secondNumber;
        }

        @Parameters
        public static Collection addedNumbers() {
                return Arrays.asList(new Integer[][] { { 3, 1, 2 }, { 5, 2, 3 },
                                { 7, 3, 4 }, { 9, 4, 5 }, });
        }

        @Test
        public void sum() {
                Calculate add = new Calculate();
                System.out.println("Addition with parameters : " + first + " and "
                                + second);
                assertEquals(expected, add.sum(first, second));
        }
}
```

```
Addition with parameters : 1 and 2
Adding values: 1 + 2
Addition with parameters : 2 and 3
Adding values: 2 + 3
Addition with parameters : 3 and 4
Adding values: 3 + 4
Addition with parameters : 4 and 5
Adding values: 4 + 5
```