

# Génie Logiciel et Gestion de Projets

## Testing

# A golden rule...

- Make it Work
- Make it Right
- Make it Fast

# How does this work?

- First make sure the software does what you want
  - use unit tests
- Then rework the code until it speaks for itself
  - use refactorings
- Then optimize the performance, if needed
  - use profiling

# Roadmap

- What is testing?
- What are tests?
- What is unit testing?
  - Designing unit tests?
- Testing Frameworks
  - Let's go for an example!!

# Testing

# What is testing?

- Testing is the activity of executing a program with the intent of finding a defect
- a successful test is one that finds a defect!

# Golden Rules of Testing

- Goal of testing:
  - maximize the number and severity of defects.
- Limits of testing
  - testing can only determine the presence of defects, never their absence.

# Testing Techniques

Input  
determined  
by...

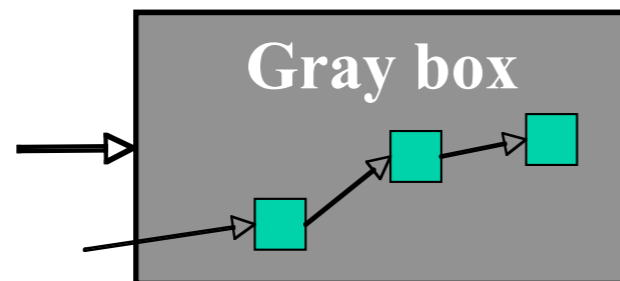
*... requirements*



Result  
*Actual output compared with required output*

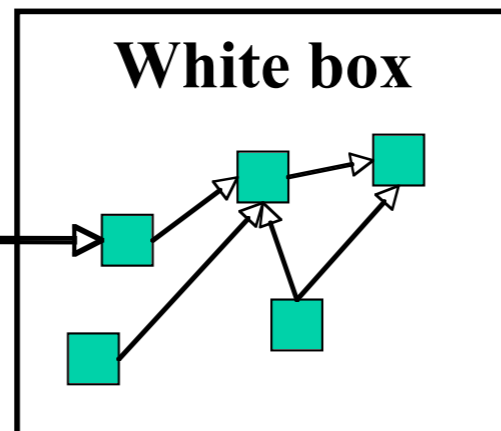
techniques with high probability of finding an as yet undiscovered mistake

*... requirements & key design elements*



*As for black- and white box testing*

*... design elements*



**Confirmation of expected behavior**

Adapted from *Software Engineering: An Object-Oriented Perspective* by Eric J. Braude (Wiley 2001).

# Testing Strategies

- Testing strategies plans that tell you when you should perform what testing technique.

# Testing Strategies

Unit Testing	test individual components
Module Testing	test a collection of related components
Sub-System Testing	test sub-system interface mismatches
System Testing	test interactions between sub-systems tests that the complete system fulfills requirements
Acceptance Testing	test system with real rather than simulated data

# Tests

# Tests

- Tests represent your trust in the system
- Build them incrementally
  - Do not need to focus on everything
  - When a new bug shows up: write a test
- Even better: test first!
  - Act as your first client
  - Helps finding proper interfaces
- Tests are active documentation: they are always in sync

# Testing Style

- “The style here is to write a few lines of code, then a test that should run, or even better, to write a test that won't run, then write the code that will make it run.”
  - write unit tests that thoroughly test a single class
  - write tests as you develop (even before you implement)
  - write tests for every new piece of functionality
- “Developers should spend 25-50% of their time developing tests.”

# But I can't cover anything!

- Sure! Nobody can but:
  - When someone discovers a defect in your code, first write a test that demonstrates the defect.
  - Then debug until the test succeeds.

“Whenever you are tempted to type something into a print statement or a debugger expression, write it as a test instead.” Martin Fowler

# Unit tests

# Unit Testing?

- Why?
  - locate small errors (within a unit) fast.
- Who?
  - Person developing the unit writes the tests
- When?
  - At the latest when a unit is delivered to the rest of the team
    - no test => no unit

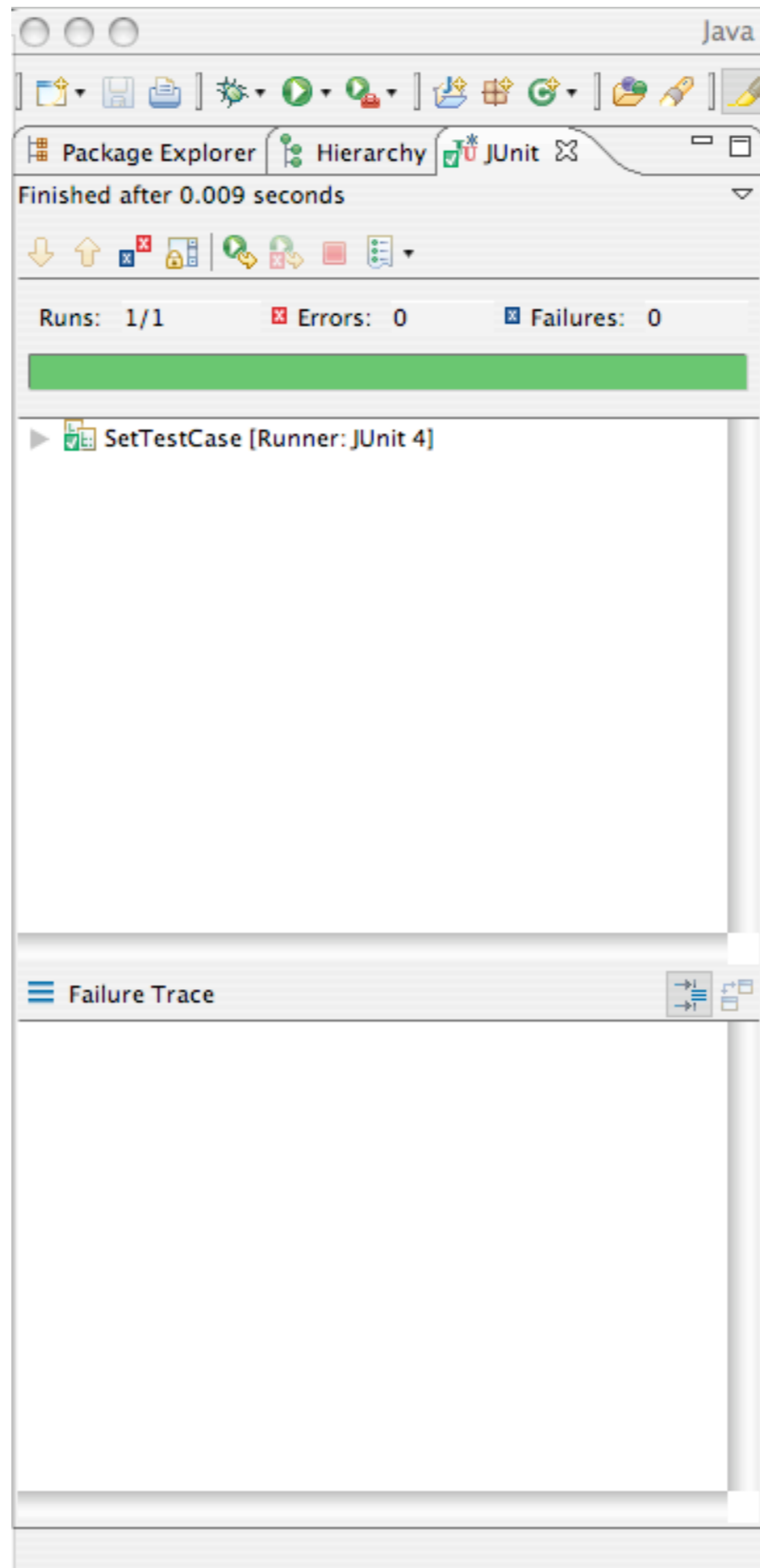
# Example Unit Test

```
public class SetTestCase extends TestCase{

    public SetTestCase(String name) {
        super(name);
    }

    public void testAdd() {
        Set<Integer> empty = new HashSet<Integer>();
        empty.add(5);
        assertTrue(empty.contains(5));
    }

    public static void main(String[] args) {
        junit.textui.TestRunner.run(SetTestCase.class);
    }
}
```



# Designing tests

- Build simple tests
- Check that failures are caught
- Run tests frequently (every couple of minutes)
- Test Infrastructure code first, then application-specific code
- Reuse as much test code as you can (tests are code!)
- Write small tests that test one particular aspect
- Make sure the tests are deterministic

# Designing tests

- Checklists for method and class testing in [Braude].
- Cookbook by [Beck].

# Checklists by [Braude]

# One way to... perform Method Testing (1/2)

1. Verify operation at *normal parameter* values.
2. Verify operation at *limit parameter* values.
3. Verify operation *outside parameter* values.
4. Ensure that *all instructions* execute.
5. Check all *paths*, including both sides of all branches.
6. Check the *use of all called objects*.
7. Verify the handling of all *data structures*.
8. Verify the handling of all *files*.

# One way to... perform Method Testing (2/2)

9. Check *normal* termination of all *loops*.
10. Check *abnormal* termination of all *loops*.
11. Check *normal* termination of all *recursion*.
12. Check *abnormal* termination of all *recursions*.
13. Verify the *handling* of all *error* conditions.
14. Check *timing* and *synchronization*.
15. Verify all *hardware dependencies*.

# One way to... perform Class Unit Testing (1/2)

## 1. Exercise methods in combination

- 2-5, usually
- choose most common sequences first
- include sequences likely to cause defects
- requires hand-computing the resulting attribute values

## 2. Focus unit tests on each attribute

- initialize, then execute method sequences that affect it

## 3. Verify that each class invariant is unchanged

1. verify that the invariant is true with initial values
2. execute a sequence (e.g., the same as in 1.)
3. verify that the invariant still true

# One way to... perform Class Unit Testing

## 4. Verify that objects transition among expected states

- plan the state / transition event sequence
- set up the object in the initial state by setting variables
- provide first event & check that transition occurred etc.

# Cookbook by [Beck]

# Unit Testing Cookbook

- Developers write their own tests, one per class.
- Ensure that you get the specified behaviour of the public interface of a class.
- General setup of a test:
  - Create a context,
  - Send a stimulus,
  - Check the results

# Example Unit Test

```
public class SetTestCase extends TestCase{

    public SetTestCase(String name) {
        super(name);
    }

    public void testAdd() {
        Set<Integer> empty = new HashSet<Integer>(); Context
        empty.add(5); Stimulus
        assertTrue(empty.contains(5)); Check
    }

    public static void main(String[] args) {
        junit.textui.TestRunner.run(SetTestCase.class);
    }

}
```

# Testing Frameworks

# Testing Frameworks

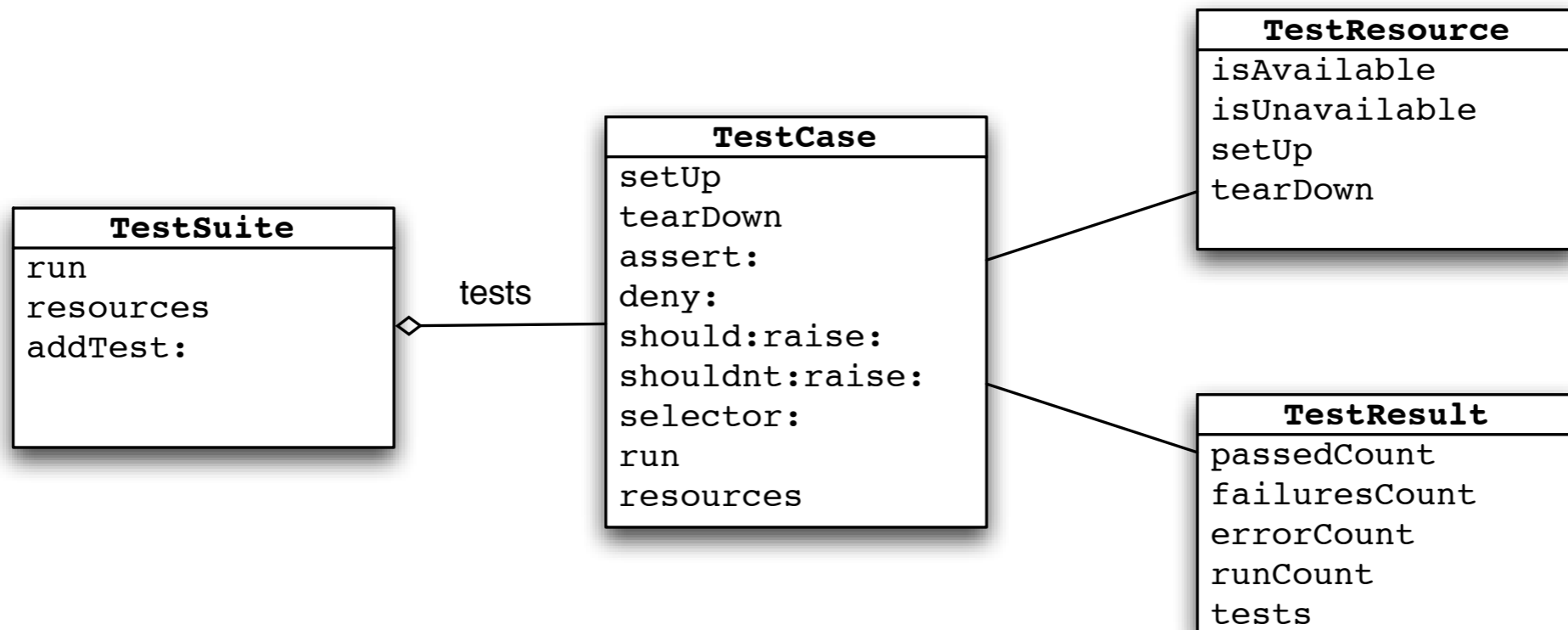
- Tests have to be repeatable.
- Unit Testing Frameworks implement necessary infrastructure so that you can set up your tests, run them frequently, and see the results.
- SUnit is “the mother of all unit test frameworks”.
  - started in Smalltalk,
  - fanned out to all kinds of other languages
    - JUnit, CppUnit, ...

# SUnit

- Simple unit testing framework for Smalltalk
- Smalltalk-dialect independent
  - you find the same thing for different Smalltalk environments (VisualWorks, Squeak, VisualAge, Dolphin, ...)
- Simple GUI
  - Again, dialect-independent
  - More refined GUI's exist for most platforms

# SUnit core classes

- a TestCase represents one test
- A testSuite is a group of tests
- SUnit automatically builds a suite from the methods starting with 'test\*'
- TestResult represents a test execution result



# Basic usage

- Make a subclass from class TestCase
- Create methods where the name starts with 'test'
- if not, they are not run as tests
  - means that you can add auxiliary methods that do not start with 'test' !

# Example

```
Class: SetTestCase
```

```
    superclass: TestCase
```

```
SetTestCase>>testAdd                                "one test"
```

```
    | empty |  
    empty := Set new.  
    empty add: 5.  
    self assert: (empty includes: 5).
```

```
SetTestCase>>testOccurrences                          "other test"
```

```
    | empty |  
    empty := Set new.  
    self assert: (empty occurrencesOf: 0) = 0.  
    empty add: 5; add:5.  
    self assert: (empty occurrencesOf: 5) = 1
```

# Example: Let's take a closer look

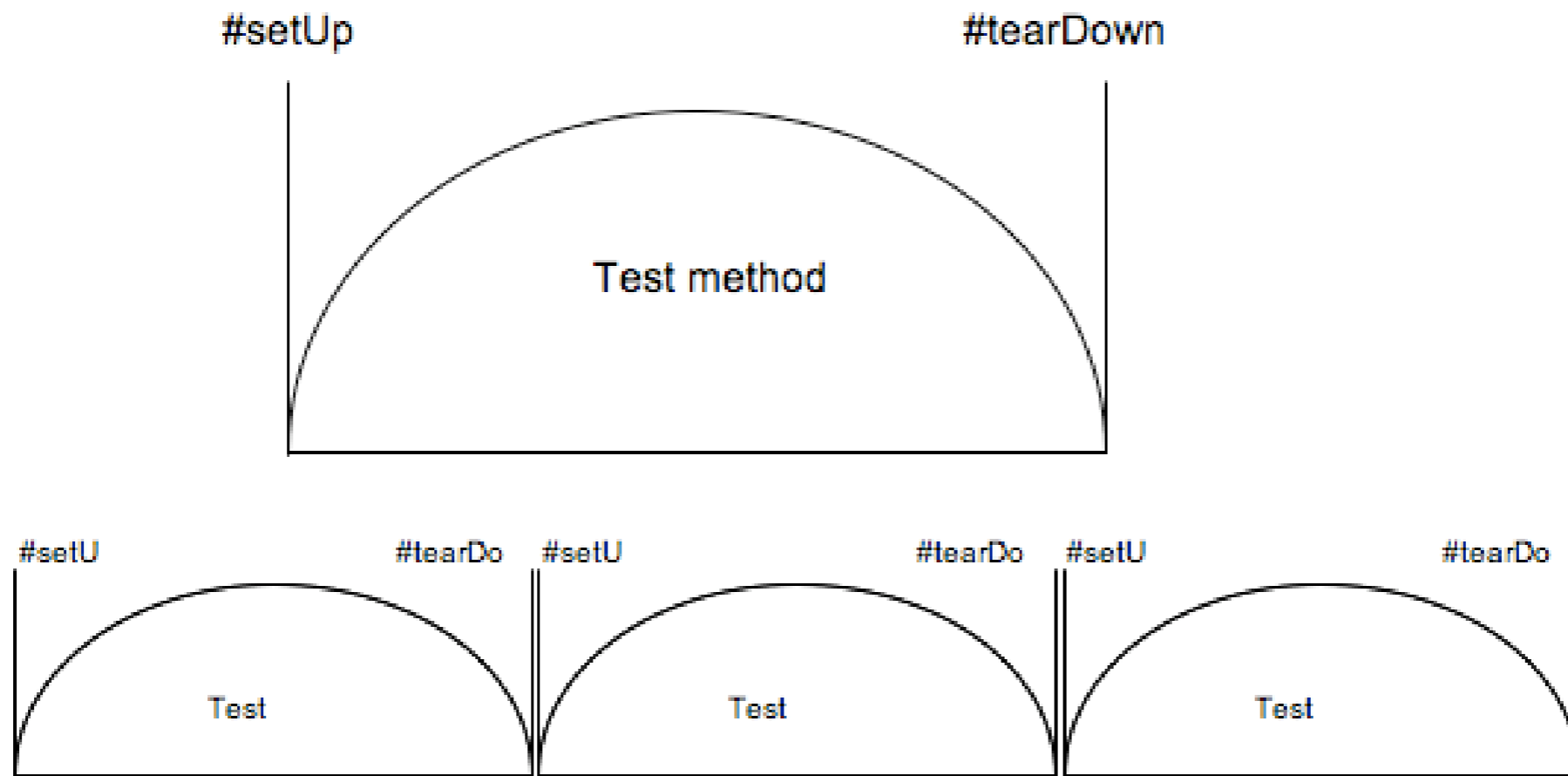
- What bothers you if you inspect these two methods?

```
SetTestCase>>testAdd                "one test"  
  | empty |  
  empty := Set new.  
  empty add: 5.  
  self assert: (empty includes: 5).
```

```
SetTestCase>>testOccurrences        "other test"  
  | empty |  
  empty := Set new.  
  self assert: (empty occurrencesOf: 0) = 0.  
  empty add: 5; add:5.  
  self assert: (empty occurrencesOf: 5) = 1
```

# Setup and TearDown

- Executed before and after each test
  - *setUp* allows us to specify and reuse the context
  - *tearDown* makes us clean-up afterwards



# Set Example Revisited

```
Class: SetTestCase
```

```
  superclass: TestCase
```

```
  instance variable: 'empty full'
```

```
SetTestCase>>setUp
```

```
  empty := Set new.
```

```
  full := Set with: #abc with: 5
```

```
SetTestCase>>testAdd
```

```
    "one test method"
```

```
  empty add: 5.
```

```
  self assert: (empty includes: 5).
```

```
SetTestCase>>testOccurrences
```

```
    "other test method"
```

```
  self assert: (empty occurrenceOf: 0) = 0.
```

```
  self assert: (full occurrencesOf: 5) = 1.
```

```
  full add: 5. self assert: (full occurrencesOf: 5) = 1
```

```
SetTestCase>>testRemove
```

```
    "yet another test method"
```

```
  full remove: 5.
```

```
  self assert: (full includes: #abc).
```

```
  self deny: (full includes: 5)
```

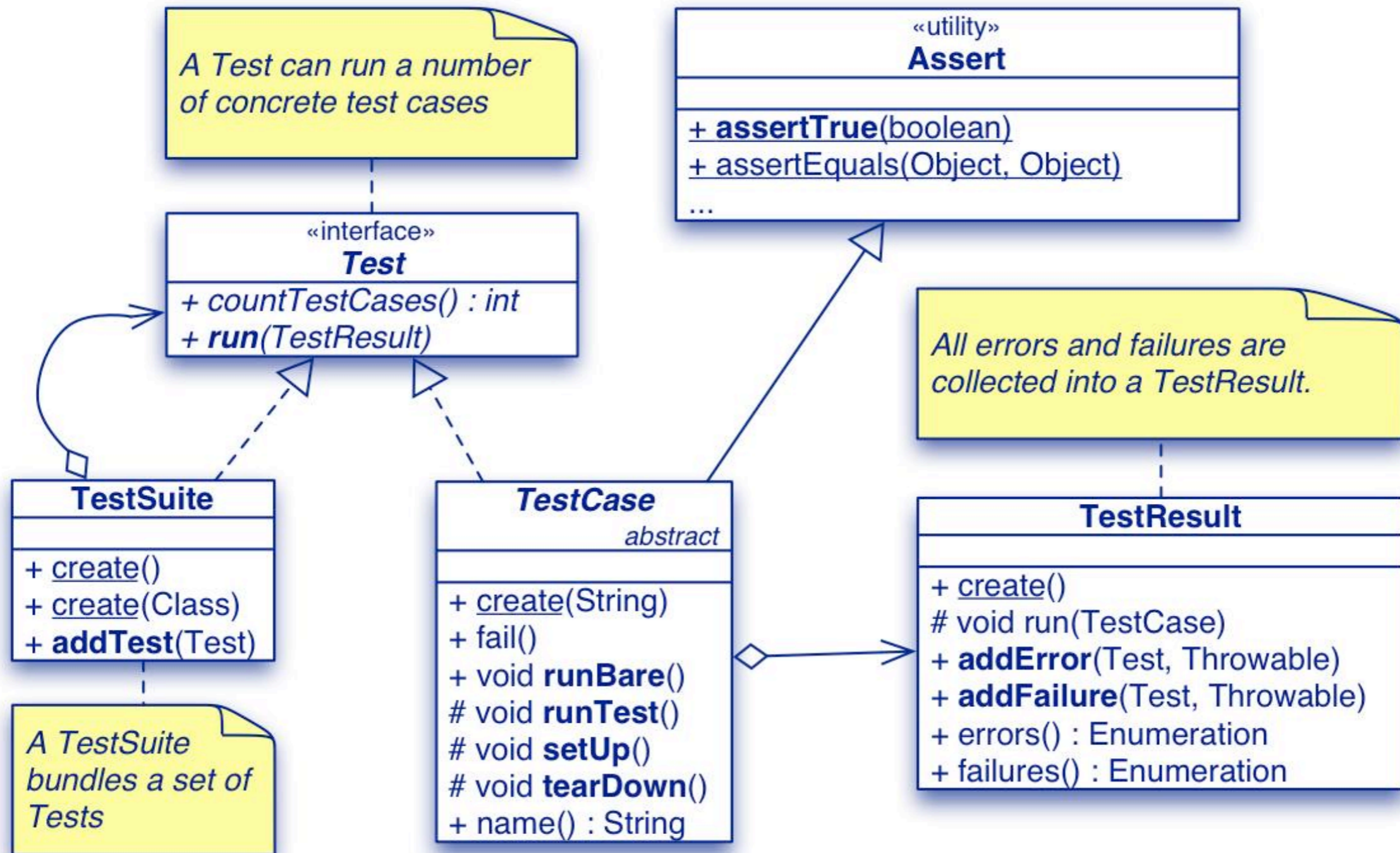
# cppUnit

- testClass derived from TestFixture
- macro's for initiating and finishing a test suite (CPPUNIT\_TEST\_SUITE and CPPUNIT\_TEST\_SUITE\_END)
- method setUp and tearDown (cf. SUnit)
- actual tests are defined with the help of CppUnit macros (e.g. CPPUNIT\_ASSERT\_EQUAL, CPPUNIT\_ASSERT,...)

# JUnit overview

- Junit (inspired by Sunit) is a simple “testing framework” that provides:
  - classes for writing Test Cases and Test Suites
  - methods for setting up and cleaning up test data (“fixtures”)
  - methods for making assertions
  - textual and graphical tools for running tests

# JUnit(<4) Framework



# JUnit 4 Annotations

- **@BeforeClass:**  
Methods that are called before any test runs. Test parameters can be set and objects can be instantiated here if they do not change in later tests.
- **@AfterClass:**  
Methods that are called at the end of a testsuite after all tests have been run. Resources that have been used during the tests (like network connections or streams) can be freed here.
- **@Before:**  
Methods that are executed before every test.
- **@After:**  
Methods that are executed after every test.
- **@Test:**  
The actual test methods.
- **@Ignore:**  
Methods, testing features that are still to be implemented, can be disabled temporarily.

# JUnit 4 Assertions

- `assertEquals(a, b)`
- `assertTrue(a, b)`
- `assertFalse(a, b)`
- `assertNull(a, b)`
- `assertNotNull(a, b)`
- `assertSame(a, b)`
- `assertNotSame(a, b)`

# JUnit 4 Example

```

public class SimpleFraction {

    private int numerator,
    private int denominator;

    public SimpleFraction(int num, int den) {
        numerator = num;
        denominator = den;
    }

    public void simplify() {
        long gcd = gcd(denominator, numerator);
        denominator /= gcd;
        numerator /= gcd;
        if(denominator < 0) {
            denominator = -denominator;
            numerator = -numerator;
        }
    }

    private static int gcd(int a, int b) {
        return b == 0 ? a : gcd(b, a % b);
    }
}

```

```
public int getDenominator() {
    return denominator;
}

public int getNumerator() {
    return numerator;
}

public void setDenominator(int i) {
    denominator = i;
}

public void setNumerator(int i) {
    numerator = i;
}

public static void main(String[] args) {
    SimpleFraction fract = new SimpleFraction(10, 3);
    System.out.println(fract.getNumerator() + " "
+fract.getDenominator());
}
}
```

```
import junit.framework.TestCase;  
import org.junit.Test;  
  
public class SimpleFractionTest extends TestCase{  
  
    private SimpleFraction f1, f2;  
  
    public SimpleFractionTest(String arg0){  
super(arg0);  
}
```

**@Before**

```
protected void setUp(){  
    super.setUp();  
    f1 = new SimpleFraction(15, 25);  
    f2 = new SimpleFraction(-27, 6);  
}
```

**@Test**

```
public void testGetDenominator() {  
    int result = f1.getDenominator();  
    assertTrue("getDenominator() returned" +result +" instead of 25.",  
               result==25);  
    result = f2.getDenominator();  
    assertEquals(6, result);}  
}
```

```
@Test
public void testSimplify() {
    f1.simplify();
    assertEquals(3, f1.getNumerator());
    assertEquals(5, f1.getDenominator());

    f2.simplify();
    assertEquals(-9, f2.getNumerator());
    assertEquals(2, f2.getDenominator());
}
}
```

```

import junit.framework.Test;
import junit.framework.TestSuite;

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

@RunWith(value=Suite.class)
@SuiteClasses(value={SimpleFractionTest.class})
public class AllTests {

    public static Test suite() {
        TestSuite suite = new TestSuite("Test for default package");
        //$JUnit-BEGIN$
        suite.addTest(new TestSuite(SimpleFractionTest.class));
        //$JUnit-END$
        return suite;
    }
}

```

# Conclusion: Why spending time testing?

- Find problems soon.
  - in context of what you were doing!
- Serve as documentation.
- Ease maintenance and evolution.
  - new developers jump in anytime.
- Have something to show all the time.

# References

- [Sommerville] Ian Sommerville. Software Engineering. Eighth Edition. 2007. ISBN: 9780321313799
- [Braude] Eric J. Braude. Software Engineering. An Object-Oriented Perspective. 2001. ISBN: 0471322083
- [Beck] Simple Smalltalk Testing: With Patterns. Kent Beck. <http://www.xprogramming.com/testfram.htm>
- Using JUnit in Eclipse. Christopher Batty. 2003. <http://www.cs.umanitoba.ca/~eclipse/10-JUnit.pdf>
- cppUnit project page. <http://cppunit.sourceforge.net>