Ingineria Programării

Cursul 6 – 30 Martie 2022

adiftene@info.uaic.ro

Cuprins

- Din Cursurile trecute…
- SOLID Principles
- Design Patterns
  - Definitions
  - Elements
  - Example
  - Classification
- JUnit Testing
  - Netbeans (Exemplu 1)
  - Eclipse (Exemplu 2)

Din Cursurile Trecute

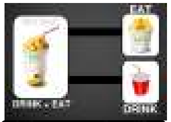
- Etapele Dezvoltării Programelor
- Ingineria Cerinţelor
- Diagrame UML
- SOLID
- GRASP

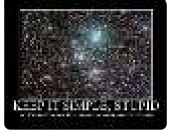
R – GRASP

- Principii, responsabilități
- Information Expert
- Creator
- Low Coupling
- High Cohesion
- Controller

R – SOLID and Other Principles

- SOLID Principles
  - SRP – Single Responsibility Principle
  - OCP – Open/Closed Principle
  - LSP – Liskov Substitution Principle
  - ISP – Interface Segregation Principle
  - DIP – Dependency Inversion Principle
- DRY – Don't Repeat Yourself
- YAGNI – You Aren't Gonna Need It
- KISS – Keep It Simple, Stupid





Cuprins

- Din Cursurile trecute…
- Design Patterns
  - Definitions
  - Elements
  - Example
  - Classification
- JUnit Testing
  - Netbeans (Exemplu 1)
  - Eclipse (Exemplu 2)

Design Patterns – Why?

- **If a problem occurs over and over again**, a solution to that problem has been used effectively (solution = pattern)

- When you make a design, you **should know the names of some common solutions.** Learning design patterns is good for people to **communicate each other effectively**

Design Patterns – Definitions

- “Design patterns capture solutions that have developed and evolved over time” (GOF - ***Gang-Of-Four*** (because of the four authors who wrote it), *Design Patterns: Elements of Reusable Object-Oriented Software)*

- **In software engineering (or computer science), a design pattern is a general repeatable solution to a commonly occurring problem in software design**

- The **design patterns** are language-independent strategies for solving common object-oriented design problems

Gang of Four

- Initial was the name given to a leftist political faction composed of four Chinese Communist party officials
- The name of the book (“Design Patterns: Elements of Reusable Object-Oriented Software”) is too long for e-mail, so “book by the gang of four” became a shorthand name for it
- That got shortened to "**GOF book**“. Authors are: *Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides*
- The **design patterns** in their book are *descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context*

Design Patterns – Elements

1. Pattern name
2. Problem
3. Solution
4. Consequences

Design Patterns – Pattern name

- A handle **used to describe a design problem**, its solutions, and consequences in a word or two
- Naming a pattern immediately increases our **design vocabulary**. It lets us design at a higher level of abstraction
- Having a **vocabulary** for patterns lets us talk about them with our colleagues, in our documentation
- Finding good names has been one of the hardest parts of developing our catalog

Design Patterns – Problem

- Describes **when** to apply the pattern. It explains the problem and its **context**
- It might describe specific design problems such as how to represent **algorithms** as objects
- It might describe **class** or **object** structures that are symptomatic of an inflexible design
- Sometimes the problem will include a **list of conditions** that must be met before it makes sense to apply the pattern

Design Patterns – Solution

- Describes the elements that make up the **design, their relationships, responsibilities,** and **collaborations**
- The solution **doesn’t describe a particular concrete design or implementation**, because a pattern is like a template that can be applied in many different situations
- Instead, the pattern provides an **abstract description of a design problem** and **how** a general arrangement of elements (classes and objects in our case) **solves it**

Design Patterns – Consequences

- Are the results and trade-offs of applying the pattern
- They are critical for **evaluating design alternatives** and for **understanding the costs and benefits** of applying the pattern
- The consequences for software often concern **space** and **time trade-offs**, they can address **language and implementation issues** as well
- Include its impact on a system's **flexibility, extensibility, or portability**
- Listing these consequences explicitly helps you **understand and evaluate** them

Example of (Micro) pattern

- **Pattern name**: Initialization
- **Problem**: It is important for some code sequence to be executed only once at the beginning of the execution of the program.
- **Solution**: The solution is to use a static variable that holds information on whether or not the code sequence has been executed.
- **Consequences**: The solution requires the language to have a static variable that can be allocated storage at the beginning of the execution, initialized prior to the execution and remain allocated until the program termination.

Describing Design Patterns 1

- **Pattern Name and Classification**
- **Intent** – the answer to question: *What does the design pattern do*?
- **Also Known As**
- **Motivation** – A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem
- **Applicability** – *What are the situations in which the design pattern can be applied*? *How can you recognize these situations*?
- **Related Patterns**

Describing Design Patterns 2

- **Structure** – A graphical representation of the classes in the pattern
- **Participants** – The classes and/or objects participating in the design pattern and their responsibilities
- **Collaborations** – How the participants collaborate to carry out their responsibilities
- **Consequences** – *How does the pattern support its objectives*?
- **Implementation** – *What techniques should you be aware of when implementing the pattern*?
- **Sample Code**
- **Known Uses** – Examples of the pattern found in real systems

Design Patterns – Classification

- **Creational patterns**
- **Structural patterns**
- **Behavioral patterns**
- NOT in GOF: Fundamental, Partitioning, GRASP, GUI, Organizational Coding, Optimization Coding, Robustness Coding, Testing, Transactions, Distributed Architecture, Distributed Computing, Temporal, Database, Concurrency patterns

Creational Patterns

- **Abstract Factory** groups object factories that have a common theme
- **Builder** constructs complex objects by separating construction and representation
- **Factory Method** creates objects without specifying the exact class to create
- **Prototype** creates objects by cloning an existing object
- **Singleton** restricts object creation for a class to only one instance
- Not in GOF book: Lazy initialization, Object pool, Multiton, Resource acquisition (is initialization)

Structural Patterns

- **Adapter** allows classes with incompatible interfaces to work together
- **Bridge** decouples an abstraction from its implementation so that the two can vary independently
- **Composite** composes zero-or-more similar objects so that they can be manipulated as one object.
- **Decorator** dynamically adds/overrides behavior in an existing method of an object
- **Facade** provides a simplified interface to a large body of code
- **Flyweight** reduces the cost of creating and manipulating a large number of similar objects
- **Proxy** provides a placeholder for another object to control access, reduce cost, and reduce complexity

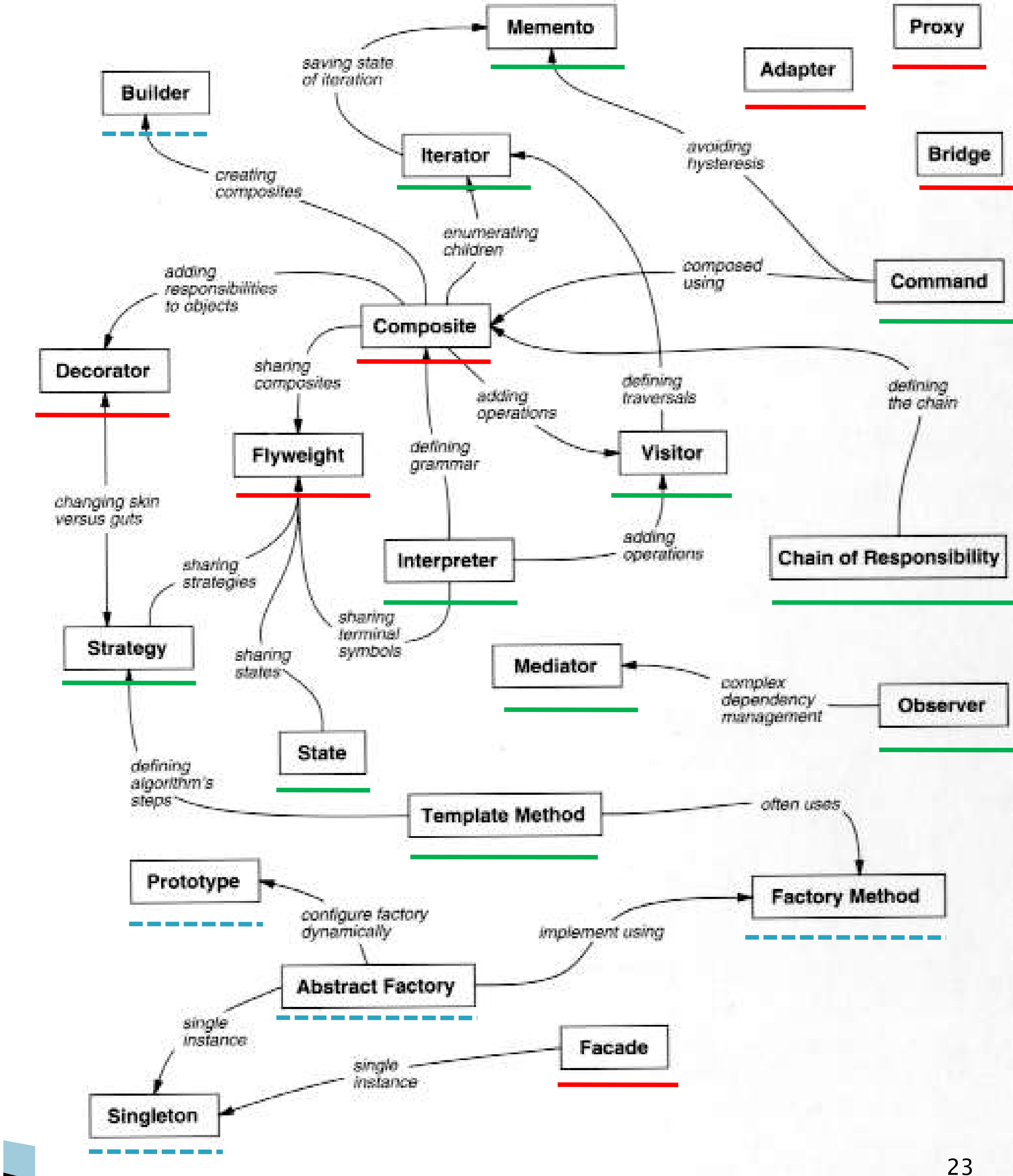
Behavioral patterns 1

- **Chain of responsibility** delegates commands to a chain of processing objects
- **Command** creates objects which encapsulate actions and parameters
- **Interpreter** implements a specialized language
- **Iterator** accesses the elements sequentially
- **Mediator** allows loose coupling between classes by being the only class that has detailed knowledge of their methods
- **Memento** provides the ability to restore an object to its previous state

Behavioral patterns 2

- **Observer** allows to observer objects to see an event
- **State** allows an object to alter its behavior when its internal state changes
- **Strategy** allows one of a family of algorithms to be selected on-the-fly at runtime
- **Template** defines an algorithm as an abstract class, allowing its subclasses to provide concrete behavior
- **Visitor** separates an algorithm from an object structure
- Not in GOF book: Null Object, Specification

- Patterns
  - Creational
  - Structural
  - Behavioral



Design Pattern Relationships

How to Select a Design Pattern?

- With more than 20 design patterns to choose from, it might be hard to find the one that addresses a particular design problem

- Approaches to finding the design pattern that's right for your problem:
  1. *Consider how design patterns solve design problems*
  2. *Scan Intent sections*
  3. *Study relationships between patterns*
  4. *Study patterns of like purpose (comparison)*
  5. *Examine a cause of redesign*
  6. *Consider what should be variable in your design*

How to Use a Design Pattern?

1. *Read the pattern once through for an overview*
2. *Go back and study the Structure*, Participants, and Collaborations sections
3. *Look at the Sample Code section to see a concrete example*
4. *Choose names for pattern participants that are meaningful in the application context*
5. *Define the classes*
6. *Define application-specific names for operations in the pattern*
7. *Implement the operations to carry out the responsibilities and collaborations in the pattern*

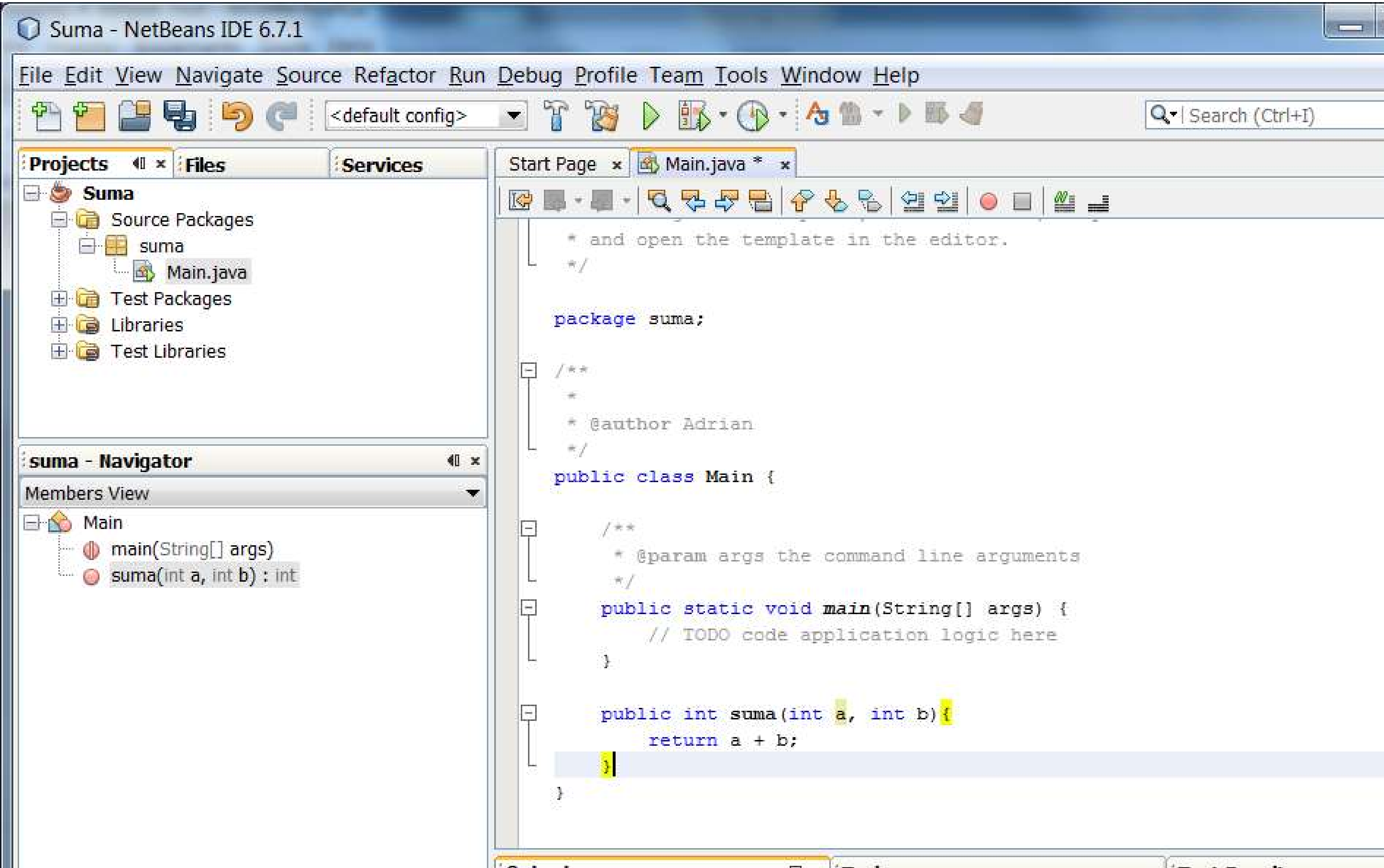
Cuprins

- Din Cursurile trecute…
- Design Patterns
  - Definitions
  - Elements
  - Example
  - Classification
- JUnit Testing
  - Netbeans (Exemplu 1)
  - Eclipse (Exemplu 2)

Unit Testing

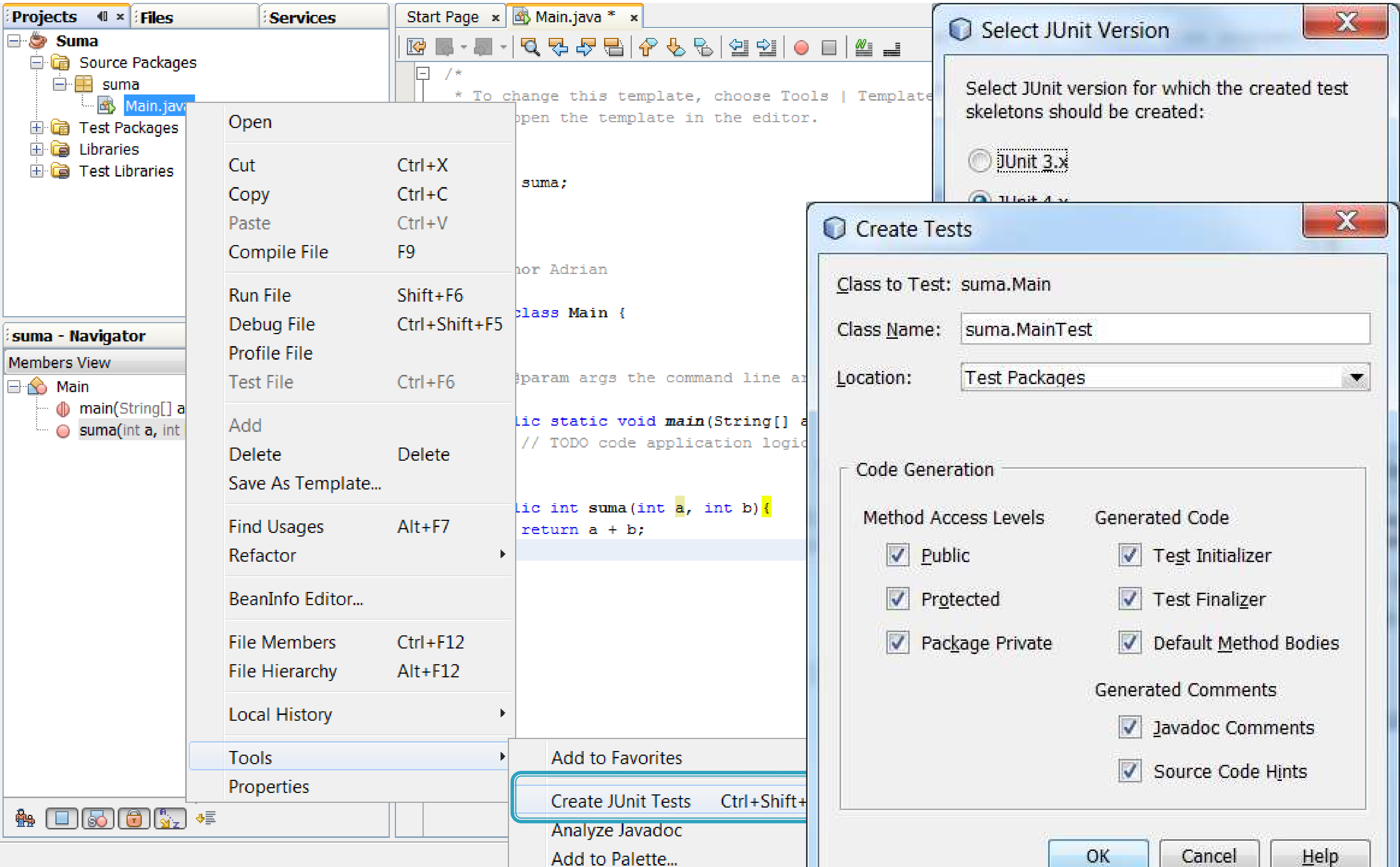
- Testarea unei funcţii, a unui program, a unui ecran, a unei funcţionalităţi
- Se face de către programatori
- Predefinită
- Rezultatele trebuie documentate
- Se folosesc simulatoare pentru Input şi Output

Unit Testing – Exemplu 1 (1)



```
 * and open the template in the editor.
 */

package suma;

/**
 *
 * @author Adrian
 */
public class Main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
    }

    public int suma(int a, int b){
        return a + b;
    }
}
```

Unit Testing – Exemplu 1 (2)



```
/*
 * To change this template, choose Tools | Template
 pen the template in the editor.

 suma;

 hor Adrian

 class Main {

 @param args the command line ar

 lic static void main(String[] a
 // TODO code application logic

 lic int suma(int a, int b){
 return a + b;
```

Select JUnit version for which the created test skeletons should be created:

○ JUnit 3.x

● JUnit 4.x

Class to Test: suma.Main

Class Name: suma.MainTest

Location: Test Packages

Code Generation

Method Access Levels

☑ Public

☑ Protected

☑ Package Private

Generated Code

☑ Test Initializer

☑ Test Finalizer

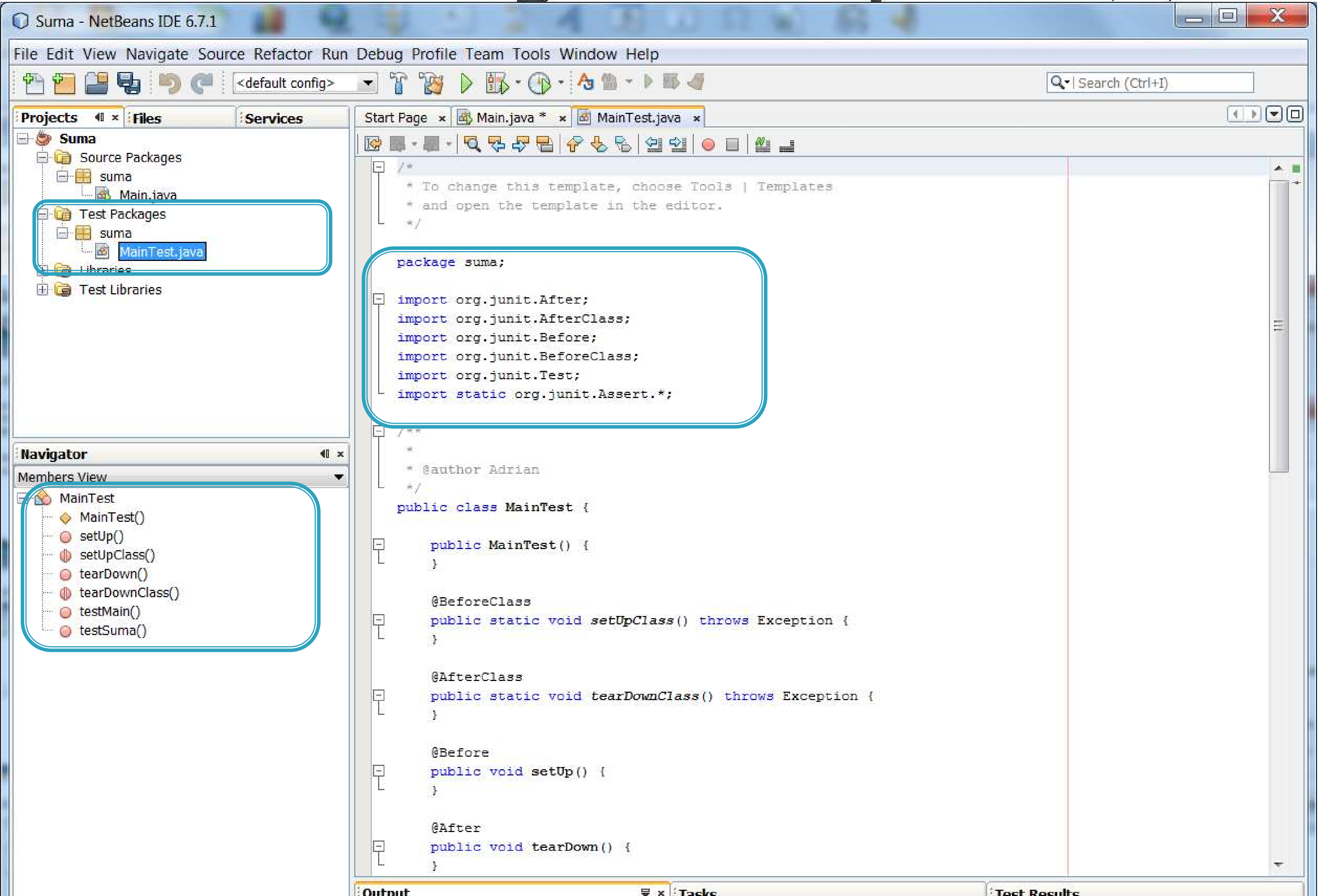
☑ Default Method Bodies

Generated Comments

☑ Javadoc Comments

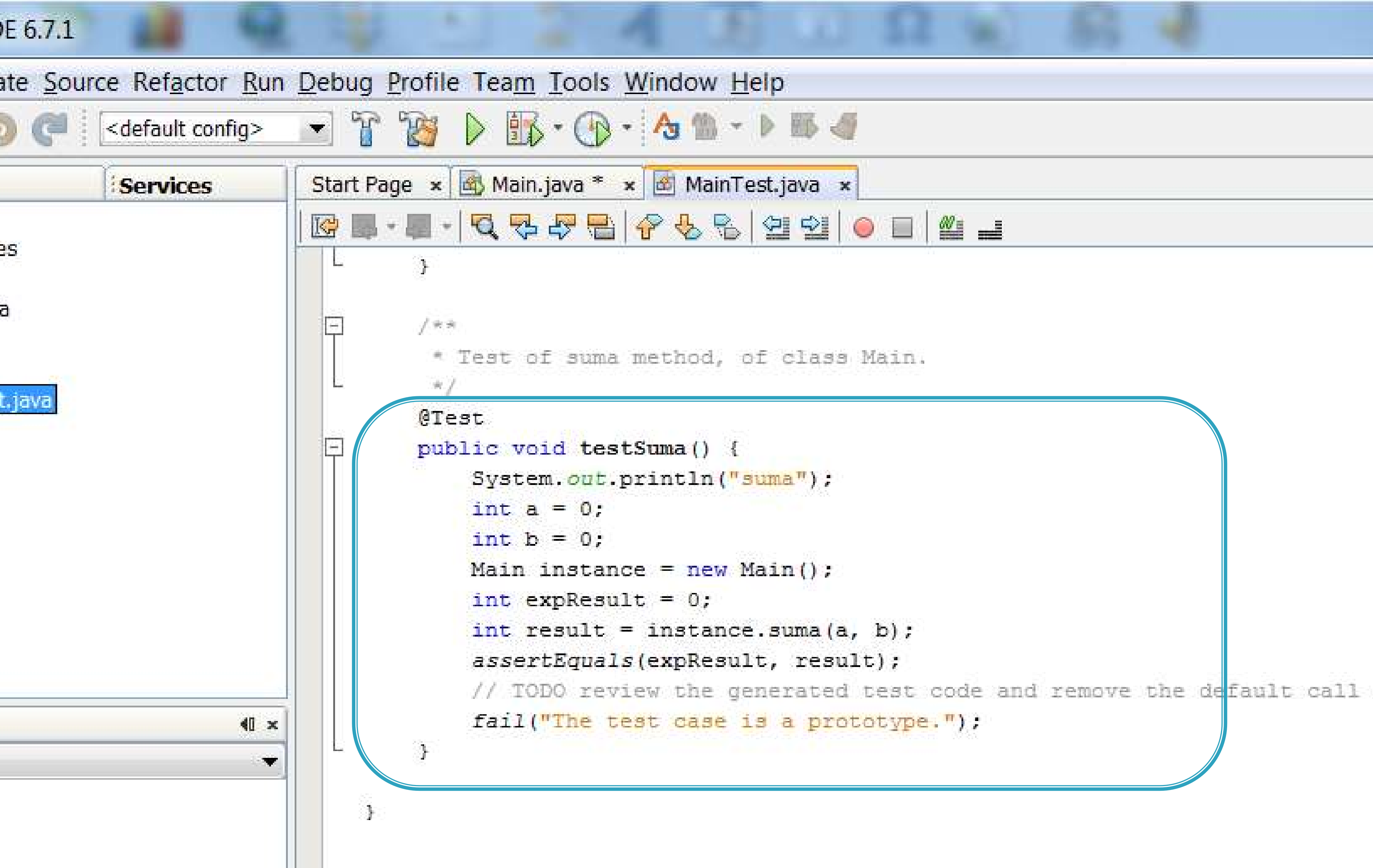
☑ Source Code Hints

Unit Testing – Exemplu 1 (3)



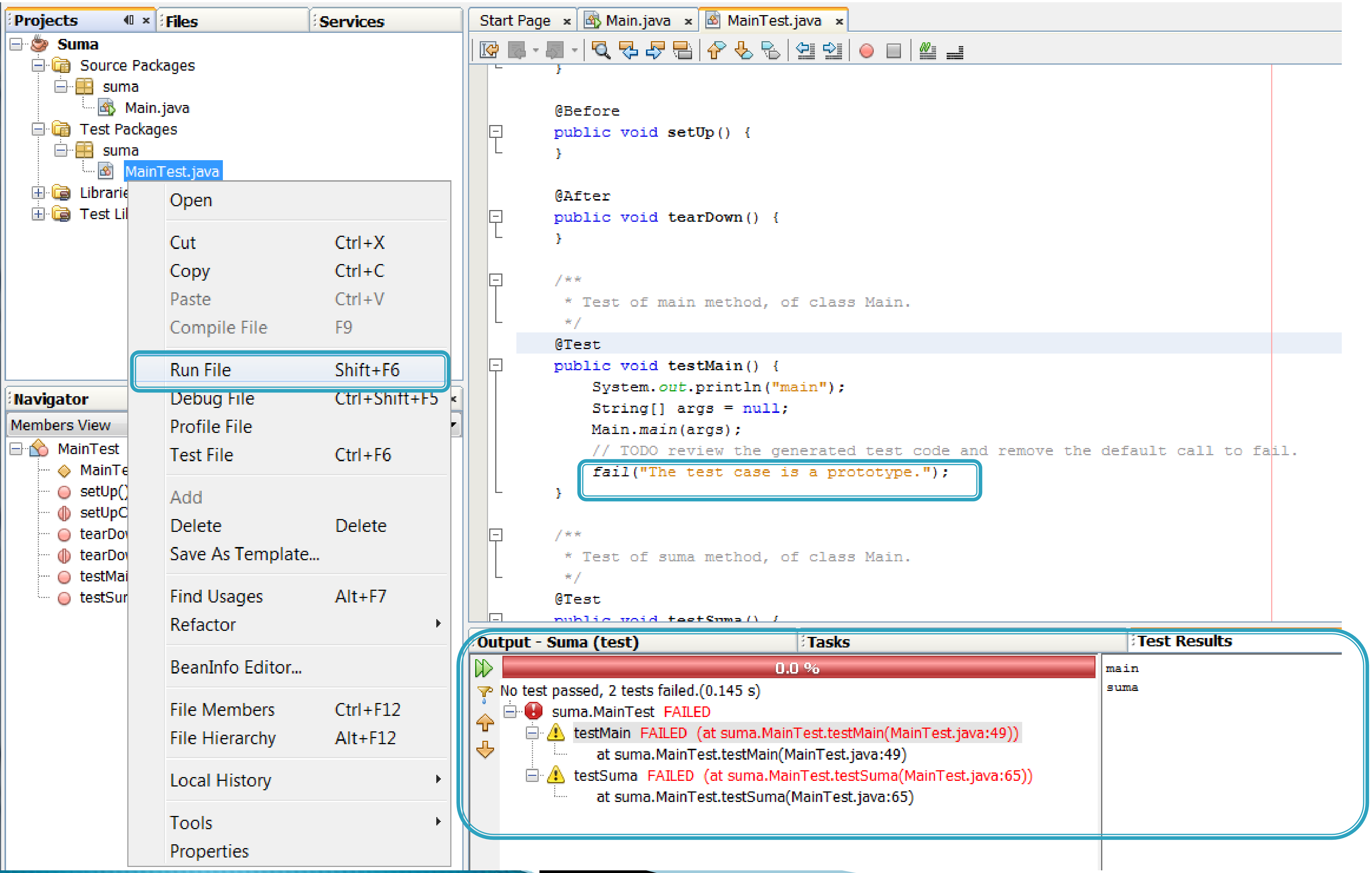
```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package suma;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;

/**
 *
 * @author Adrian
 */
public class MainTest {

    public MainTest() {
    }

    @BeforeClass
    public static void setUpClass() throws Exception {
    }

    @AfterClass
    public static void tearDownClass() throws Exception {
    }

    @Before
    public void setUp() {
    }

    @After
    public void tearDown() {
    }
```

Unit Testing – Exemplu 1 (4)



```
    }

    /**
     * Test of suma method, of class Main.
     */
    @Test
    public void testSuma() {
        System.out.println("suma");
        int a = 0;
        int b = 0;
        Main instance = new Main();
        int expResult = 0;
        int result = instance.suma(a, b);
        assertEquals(expResult, result);
        // TODO review the generated test code and remove the default call
        fail("The test case is a prototype.");
    }

}
```

Unit Testing – Exemplu 1 (5)

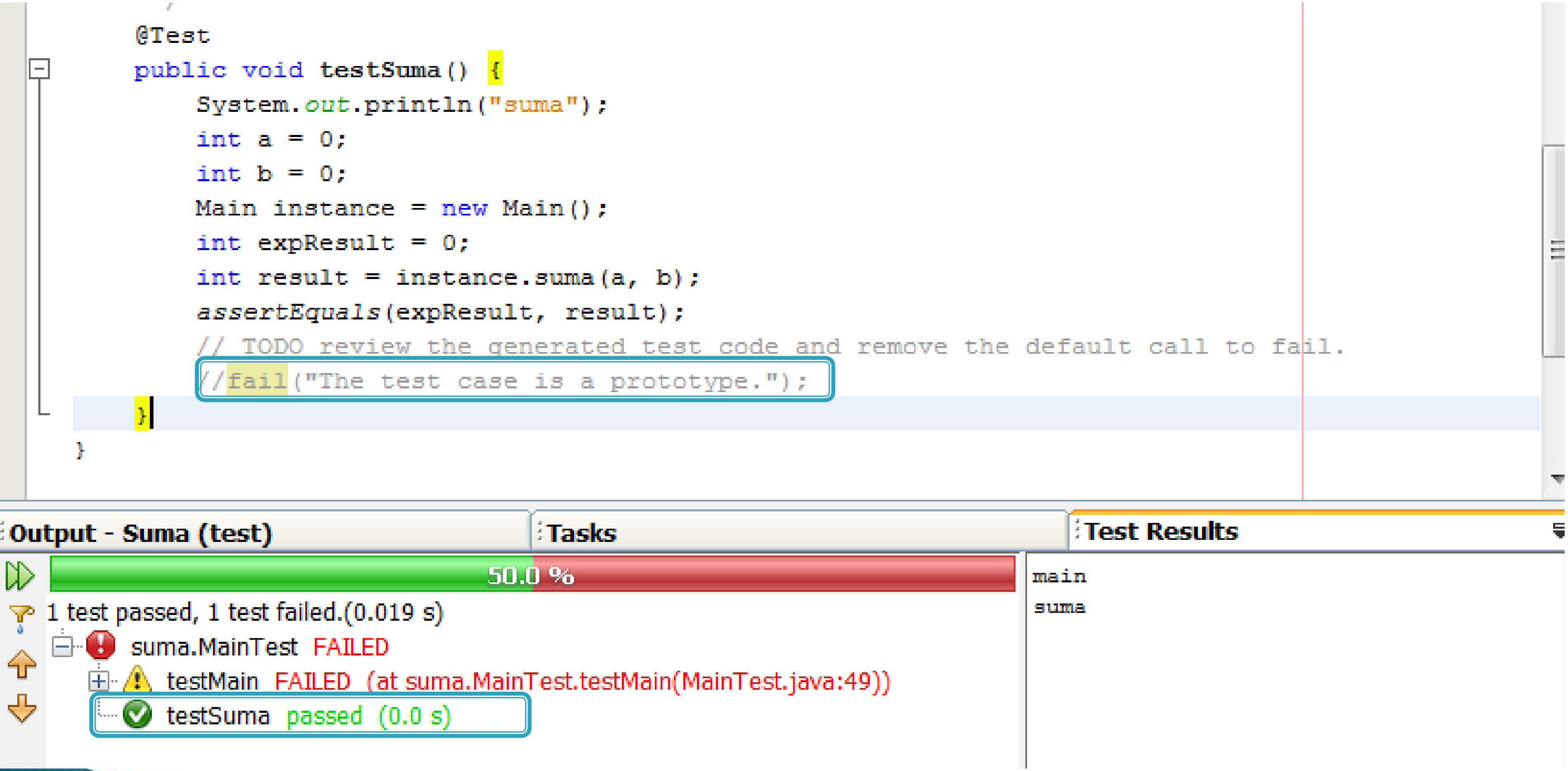


```
    }

    @Before
    public void setUp() {
    }

    @After
    public void tearDown() {
    }

    /**
     * Test of main method, of class Main.
     */
    @Test
    public void testMain() {
        System.out.println("main");
        String[] args = null;
        Main.main(args);
        // TODO review the generated test code and remove the default call to fail.
        fail("The test case is a prototype.");
    }

    /**
     * Test of suma method, of class Main.
     */
    @Test
    public void testSuma() {
```

```
0.0 %
No test passed, 2 tests failed.(0.145 s)
suma.MainTest FAILED
  testMain FAILED (at suma.MainTest.testMain(MainTest.java:49))
    at suma.MainTest.testMain(MainTest.java:49)
  testSuma FAILED (at suma.MainTest.testSuma(MainTest.java:65))
    at suma.MainTest.testSuma(MainTest.java:65)
```

```
main
suma
```

Unit Testing – Exemplu 1 (6)

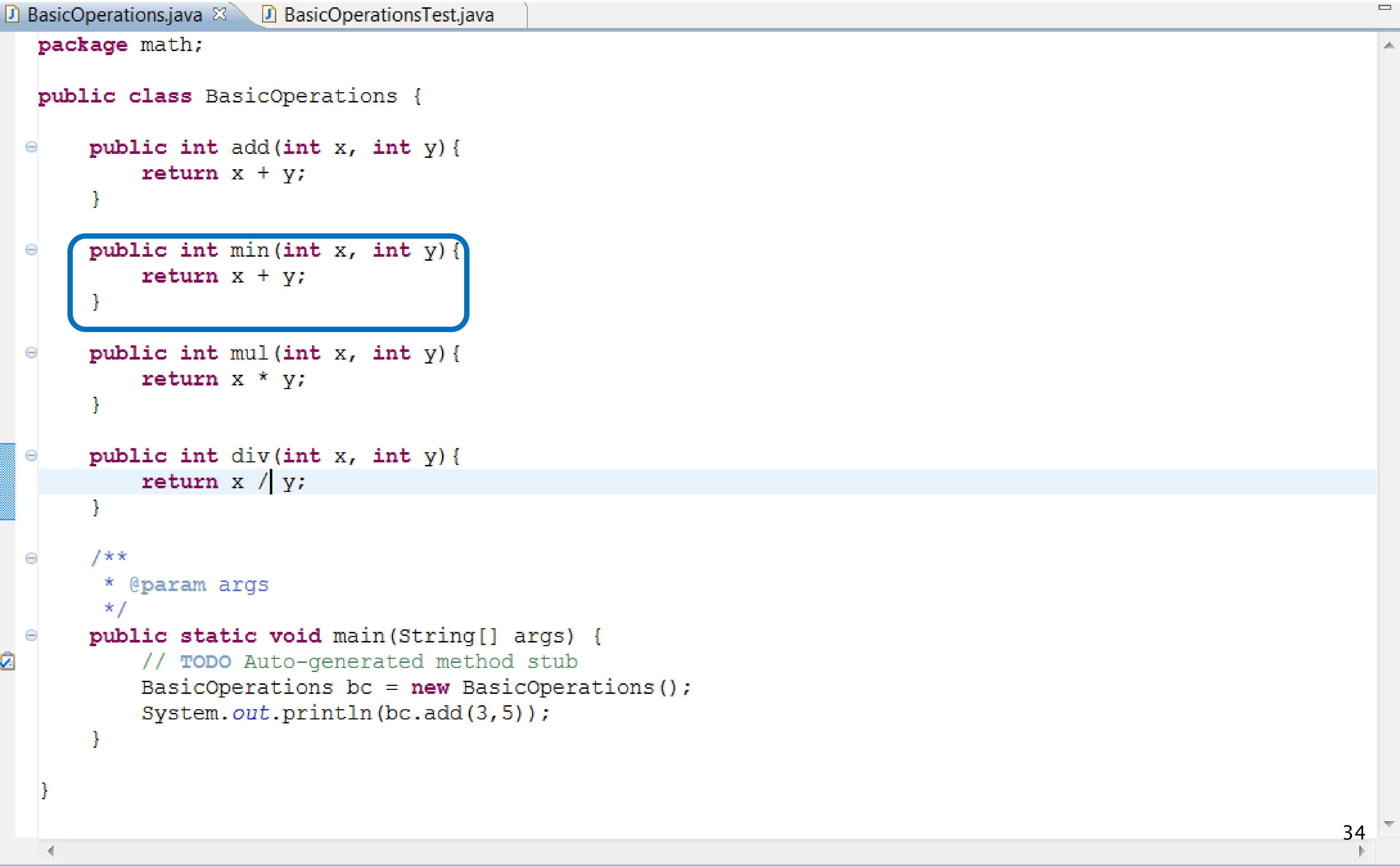


```
    @Test
    public void testSuma() {
        System.out.println("suma");
        int a = 0;
        int b = 0;
        Main instance = new Main();
        int expResult = 0;
        int result = instance.suma(a, b);
        assertEquals(expResult, result);
        // TODO review the generated test code and remove the default call to fail.
        //fail("The test case is a prototype.");
    }
}
```

```
1 test passed, 1 test failed.(0.019 s)
suma.MainTest FAILED
  testMain FAILED (at suma.MainTest.testMain(MainTest.java:49))
  testSuma passed (0.0 s)
```

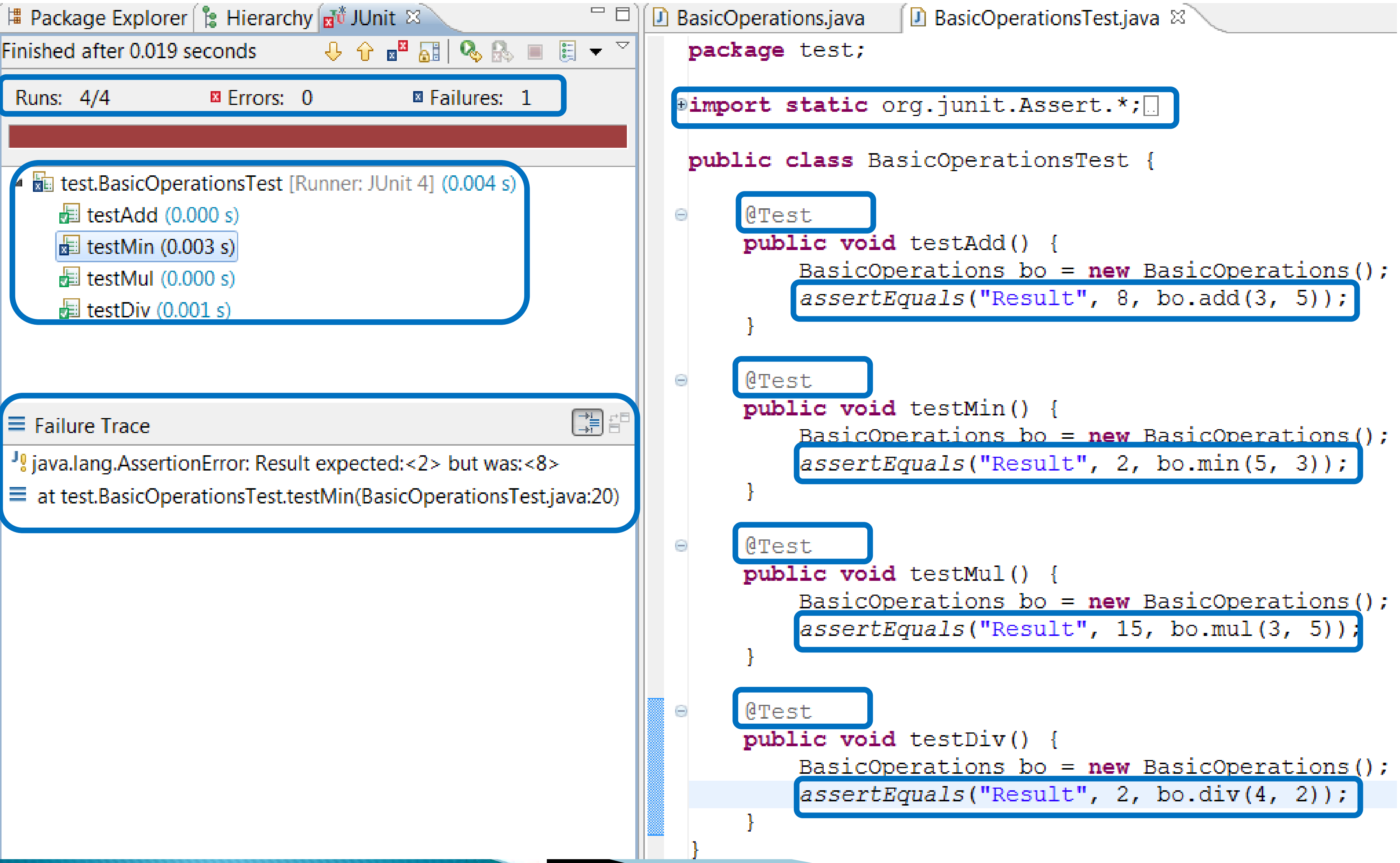
```
main
suma
```

Unit Testing – Example 2 (1)



```java
package math;

public class BasicOperations {

    public int add(int x, int y){
        return x + y;
    }

    public int min(int x, int y){
        return x + y;
    }

    public int mul(int x, int y){
        return x * y;
    }

    public int div(int x, int y){
        return x / y;
    }

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        BasicOperations bc = new BasicOperations();
        System.out.println(bc.add(3,5));
    }

}
```

Unit Testing – Example 2 (2)

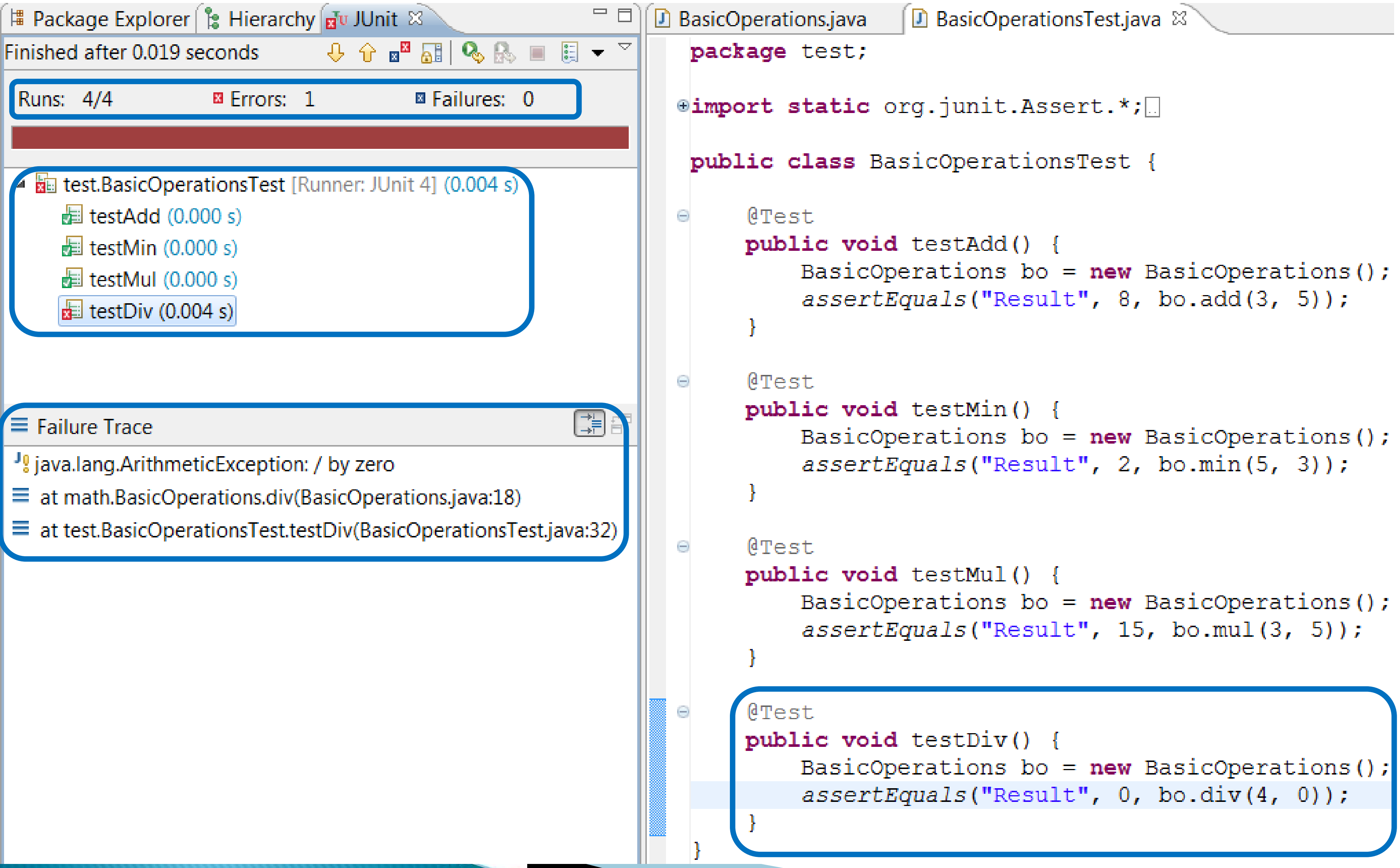


```
Finished after 0.019 seconds

Runs:   4/4          Errors:   0          Failures:   1

test.BasicOperationsTest [Runner: JUnit 4] (0.004 s)
    testAdd (0.000 s)
    testMin (0.003 s)
    testMul (0.000 s)
    testDiv (0.001 s)
```

```
java.lang.AssertionError: Result expected:<2> but was:<8>
at test.BasicOperationsTest.testMin(BasicOperationsTest.java:20)
```

```java
package test;

import static org.junit.Assert.*;

public class BasicOperationsTest {

    @Test
    public void testAdd() {
        BasicOperations bo = new BasicOperations();
        assertEquals("Result", 8, bo.add(3, 5));
    }

    @Test
    public void testMin() {
        BasicOperations bo = new BasicOperations();
        assertEquals("Result", 2, bo.min(5, 3));
    }

    @Test
    public void testMul() {
        BasicOperations bo = new BasicOperations();
        assertEquals("Result", 15, bo.mul(3, 5));
    }

    @Test
    public void testDiv() {
        BasicOperations bo = new BasicOperations();
        assertEquals("Result", 2, bo.div(4, 2));
    }
}
```

Unit Testing – Example 2 (3)



Finished after 0.019 seconds

Runs: 4/4 ☒ Errors: 1 ☒ Failures: 0

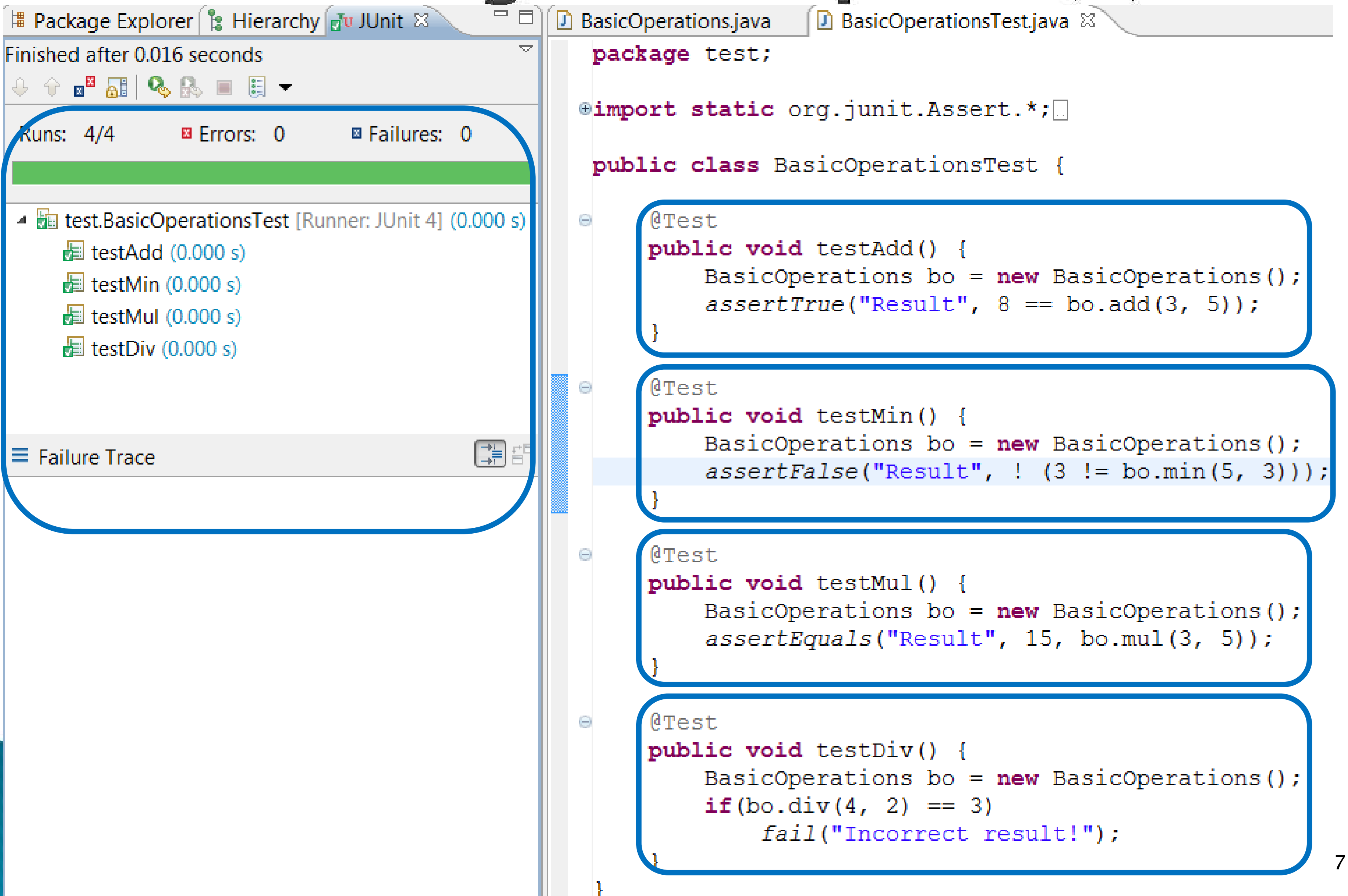
```
test.BasicOperationsTest [Runner: JUnit 4] (0.004 s)
    testAdd (0.000 s)
    testMin (0.000 s)
    testMul (0.000 s)
    testDiv (0.004 s)
```

Failure Trace

```
java.lang.ArithmeticException: / by zero
at math.BasicOperations.div(BasicOperations.java:18)
at test.BasicOperationsTest.testDiv(BasicOperationsTest.java:32)
```

```java
package test;

import static org.junit.Assert.*;

public class BasicOperationsTest {

	@Test
	public void testAdd() {
		BasicOperations bo = new BasicOperations();
		assertEquals("Result", 8, bo.add(3, 5));
	}

	@Test
	public void testMin() {
		BasicOperations bo = new BasicOperations();
		assertEquals("Result", 2, bo.min(5, 3));
	}

	@Test
	public void testMul() {
		BasicOperations bo = new BasicOperations();
		assertEquals("Result", 15, bo.mul(3, 5));
	}

	@Test
	public void testDiv() {
		BasicOperations bo = new BasicOperations();
		assertEquals("Result", 0, bo.div(4, 0));
	}
}
```

Unit Testing – Example 2 (4)



```
Finished after 0.016 seconds

Runs:  4/4        Errors:  0        Failures:  0

test.BasicOperationsTest [Runner: JUnit 4] (0.000 s)
    testAdd (0.000 s)
    testMin (0.000 s)
    testMul (0.000 s)
    testDiv (0.000 s)
```

```java
package test;

import static org.junit.Assert.*;

public class BasicOperationsTest {

    @Test
    public void testAdd() {
        BasicOperations bo = new BasicOperations();
        assertTrue("Result", 8 == bo.add(3, 5));
    }

    @Test
    public void testMin() {
        BasicOperations bo = new BasicOperations();
        assertFalse("Result", ! (3 != bo.min(5, 3)));
    }

    @Test
    public void testMul() {
        BasicOperations bo = new BasicOperations();
        assertEquals("Result", 15, bo.mul(3, 5));
    }

    @Test
    public void testDiv() {
        BasicOperations bo = new BasicOperations();
        if(bo.div(4, 2) == 3)
            fail("Incorrect result!");
    }
}
```

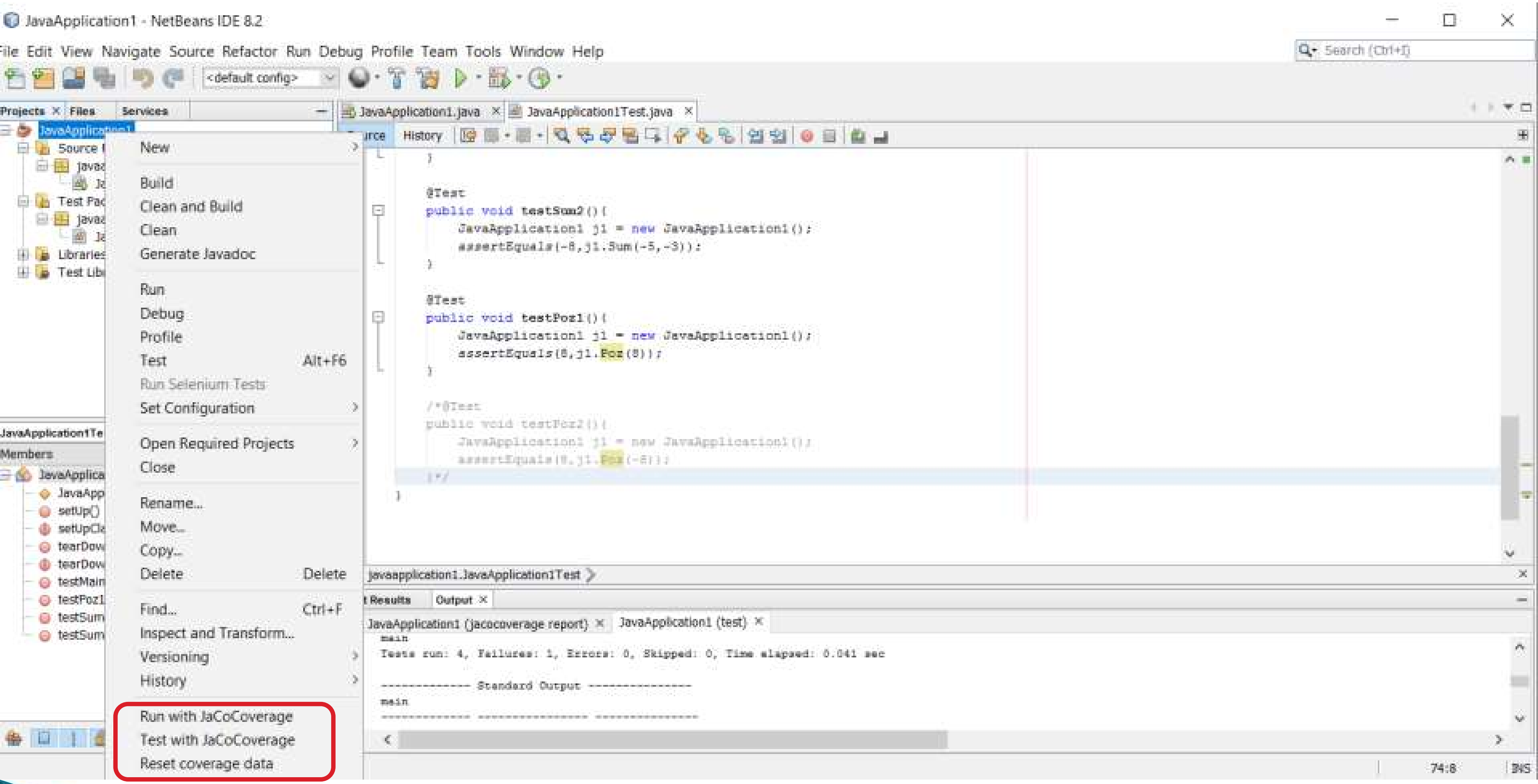
Code Coverage

- NetBeans – TikiOne JaCoCoverage:
- http://plugins.netbeans.org/plugin/48570/tikione-jacocoverage

- Java Code Coverage for Eclipse:
- http://www.eclemma.org/

- IntelliJ – Running with coverage:
- https://www.jetbrains.com/help/idea/2016.3/running-with-coverage.html

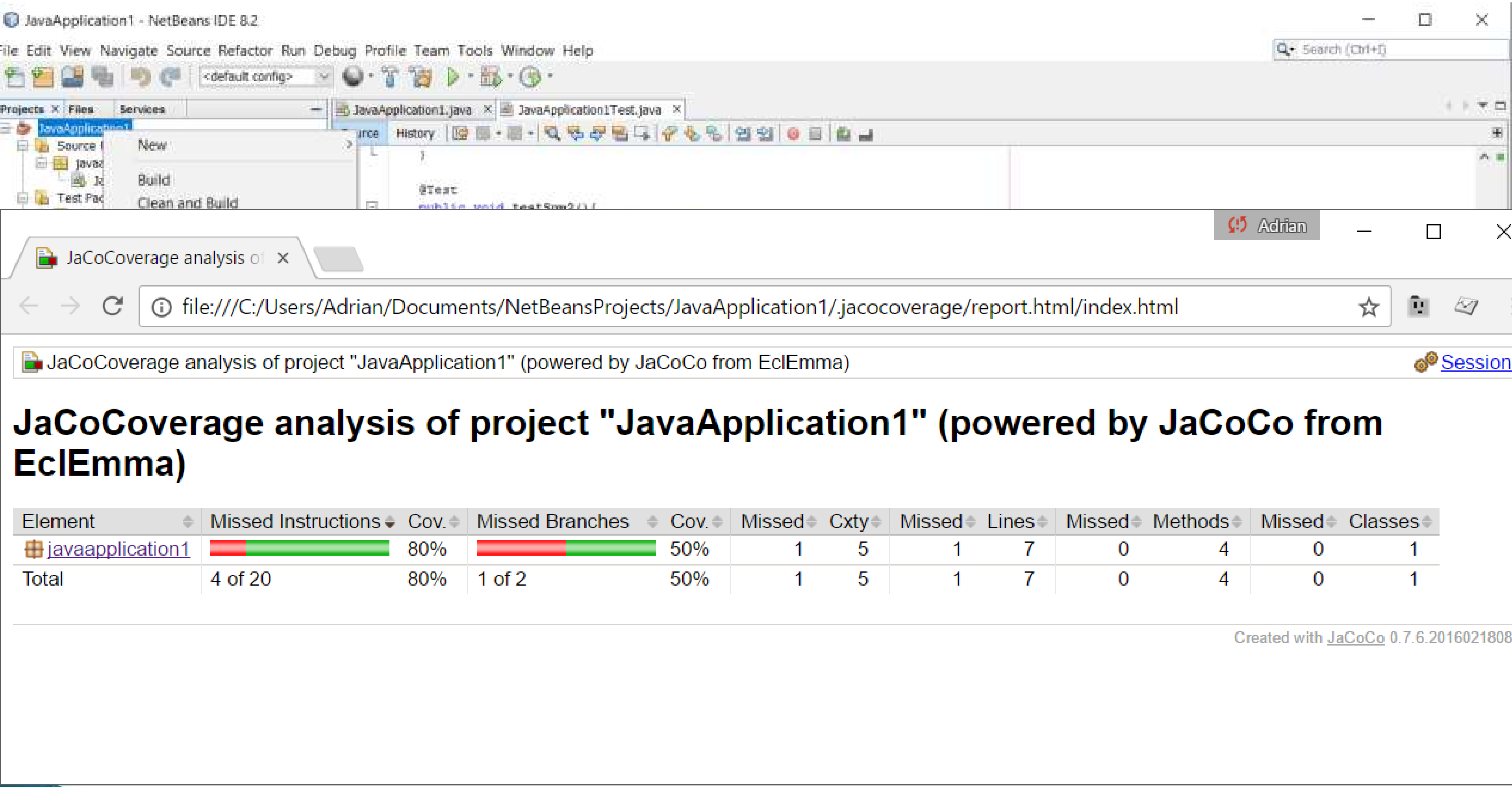
NetBeans – 1



```
    }

    @Test
    public void testSum2(){
        JavaApplication1 j1 = new JavaApplication1();
        assertEquals(-8,j1.Sum(-5,-3));
    }

    @Test
    public void testPoz1(){
        JavaApplication1 j1 = new JavaApplication1();
        assertEquals(8,j1.Poz(8));
    }

    /*@Test
    public void testPoz2(){
        JavaApplication1 j1 = new JavaApplication1();
        assertEquals(8,j1.Poz(-8));
    }*/
}
```

```
main
Tests run: 4, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.041 sec

------------- Standard Output ---------------
main
------------- ---------------- ---------------
```

NetBeans – 2



```
}

@Test
public void testSum2() {
```

file:///C:/Users/Adrian/Documents/NetBeansProjects/JavaApplication1/.jacocoverage/report.html/index.html

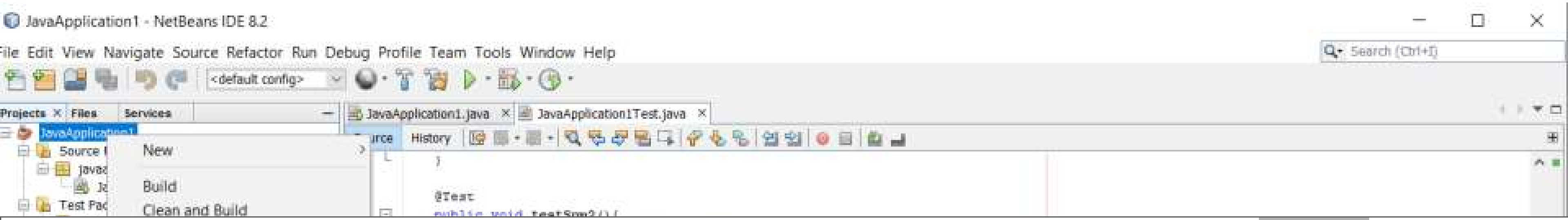
JaCoCoverage analysis of project "JavaApplication1" (powered by JaCoCo from EclEmma)

JaCoCoverage analysis of project "JavaApplication1" (powered by JaCoCo from EclEmma)

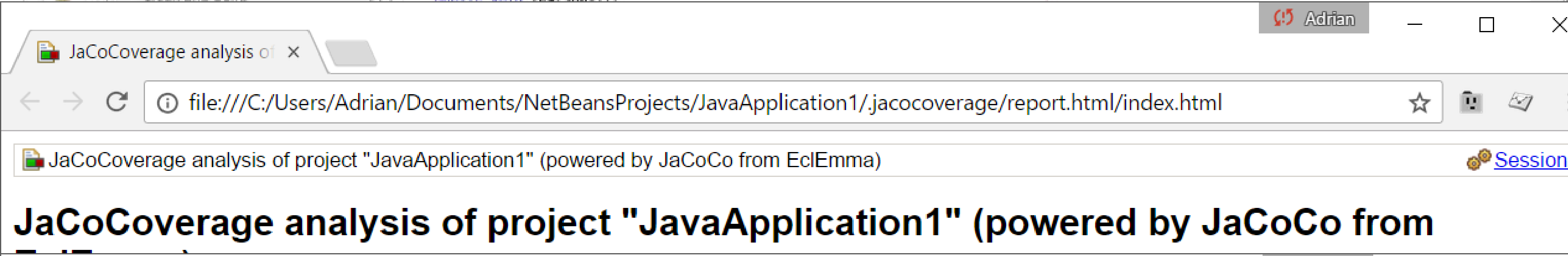
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
javaapplication1	🟥🟩	80%	🟥🟩	50%	1	5	1	7	0	4	0	1
Total	4 of 20	80%	1 of 2	50%	1	5	1	7	0	4	0	1

Created with JaCoCo 0.7.6.2016021808

NetBeans – 3



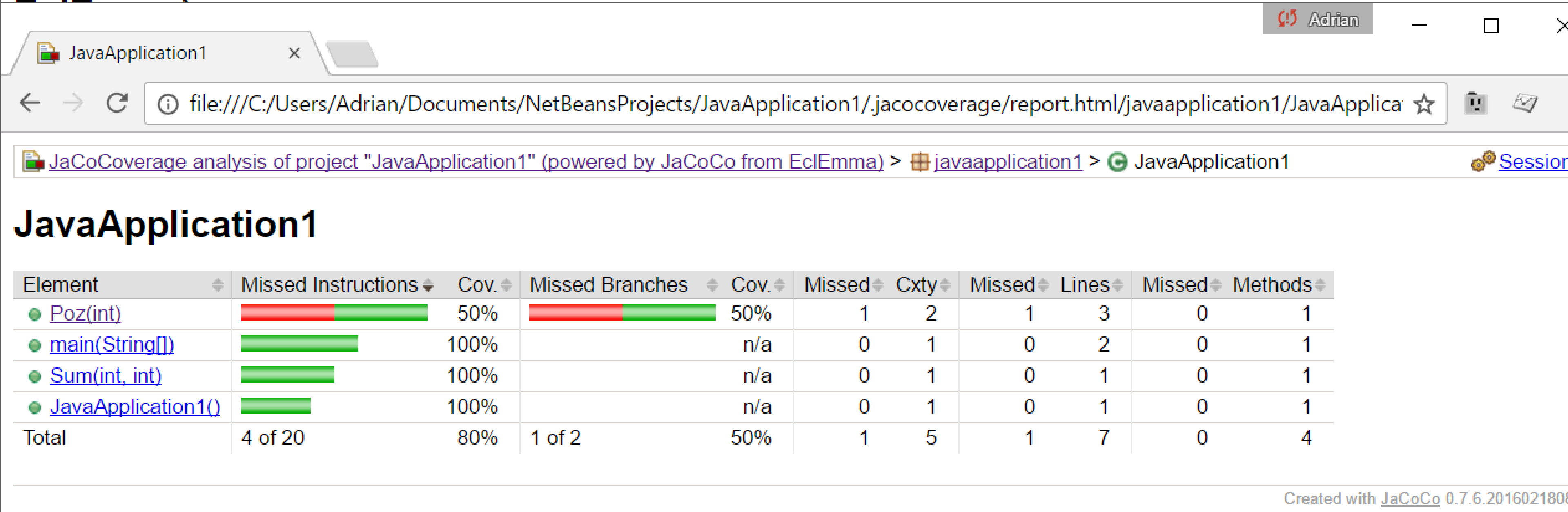
```
}

@Test
public void testSum2() {
```



file:///C:/Users/Adrian/Documents/NetBeansProjects/JavaApplication1/.jacocoverage/report.html/index.html

JaCoCoverage analysis of project "JavaApplication1" (powered by JaCoCo from EclEmma)

JaCoCoverage analysis of project "JavaApplication1" (powered by JaCoCo from EclEmma)



file:///C:/Users/Adrian/Documents/NetBeansProjects/JavaApplication1/.jacocoverage/report.html/javaapplication1/JavaApplica

JaCoCoverage analysis of project "JavaApplication1" (powered by JaCoCo from EclEmma) > javaapplication1 > JavaApplication1

JavaApplication1

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
● Poz(int)	🟥🟩	50%	🟥🟩	50%	1	2	1	3	0	1
● main(String[])	🟩	100%		n/a	0	1	0	2	0	1
● Sum(int, int)	🟩	100%		n/a	0	1	0	1	0	1
● JavaApplication1()	🟩	100%		n/a	0	1	0	1	0	1
Total	4 of 20	80%	1 of 2	50%	1	5	1	7	0	4

Created with JaCoCo 0.7.6.20160218

Concluzii

- SOLID
- Design Patterns
  - Definitions, Elements, Example, Classification
- JUnit Testing

Myths

- Clients
  - A general description of the objectives is sufficient to begin writing program
  - Requirements are constantly changing, but the software is flexible and can easy adapts
- Developers
  - Once the program is written and it is functional, our role has ended
  - Until the program doesn't work, we can not assess the quality
  - The only good product is the functional program
  - Software Engineering will create voluminous and unnecessary documentation and will cause delays

Design Patterns – Întrebări

- 1) Argumentați pentru folosirea DP.
- 2) Veniți cu argumente pentru a nu folosi DP.
- Criticism: http://sourcemaking.com/design_patterns

Bibliografie

- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software* (GangOfFour)
- Ovidiu Gheorghieş, Curs 7 IP
- Adrian Iftene, Curs 9 TAIP: http://thor.info.uaic.ro/~adiftene/Scoala/2011/TAIP/Courses/TAIP09.pdf

Links

- Gang-Of-Four: http://c2.com/cgi/wiki?GangOfFour, http://www.uml.org.cn/c%2B%2B/pdf/DesignPatterns.pdf
- Design Patterns Book: http://c2.com/cgi/wiki?DesignPatternsBook
- About Design Patterns: http://www.javacamp.org/designPattern/
- Design Patterns – Java companion: http://www.patterndepot.com/put/8/JavaPatterns.htm
- Java Design patterns: http://www.allapplabs.com/java_design_patterns/java_design_patterns.htm
- Overview of Design Patterns: http://www.mindspring.com/~mgrand/pattern_synopses.htm
- Gang of Four: http://en.wikipedia.org/wiki/Gang_of_four
- JUnit in Eclipse: http://www.vogella.de/articles/JUnit/article.html
- JUnit in NetBeans: http://netbeans.org/kb/docs/java/junit-intro.html

Vă Mulţumesc!

Pentru prezenţă,
răbdare,
colaborare...