Ingineria Programării

Cursul 6 – 27 Martie 2017

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
- GRASP

R – GRASP

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

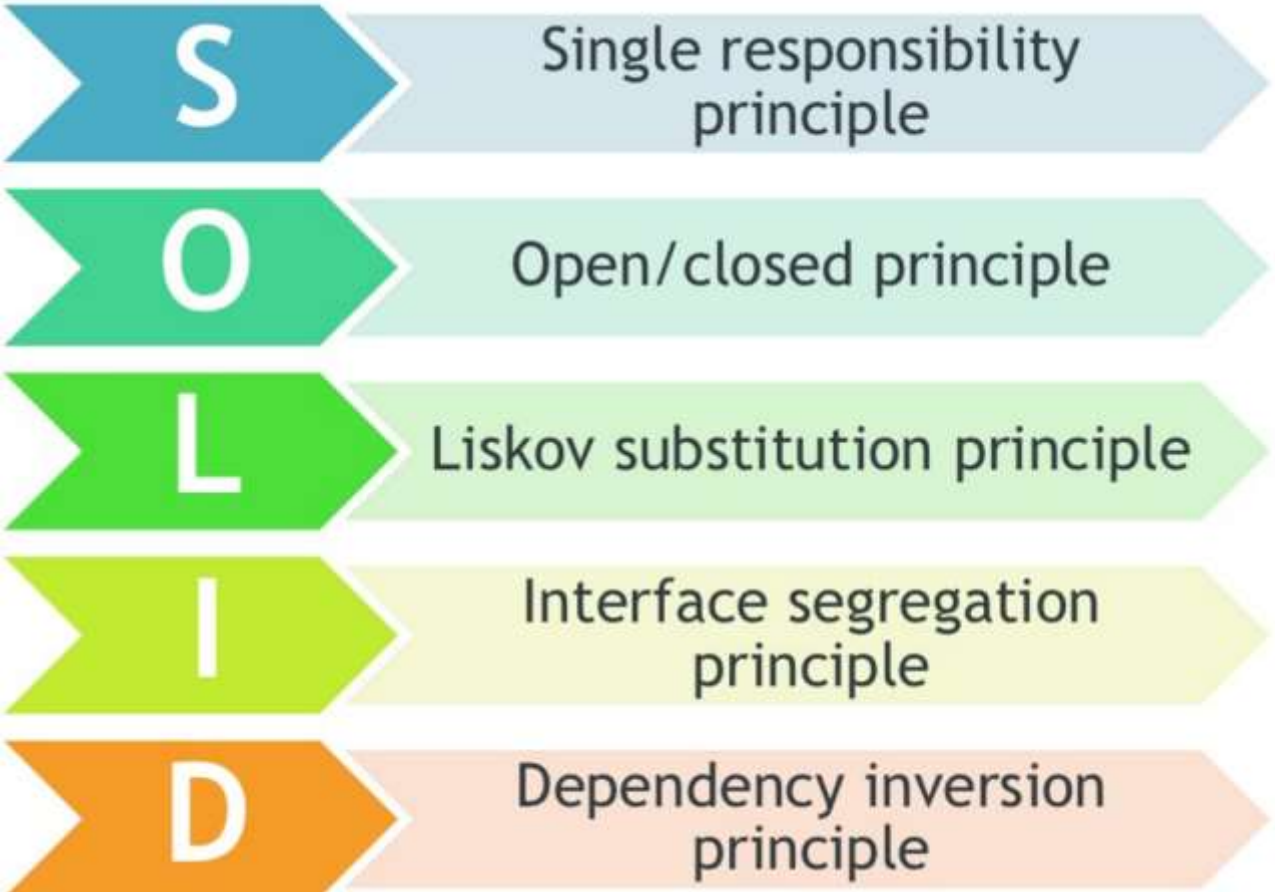
SOLID and Other Principles

- 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

S.O.L.I.D.
Principles

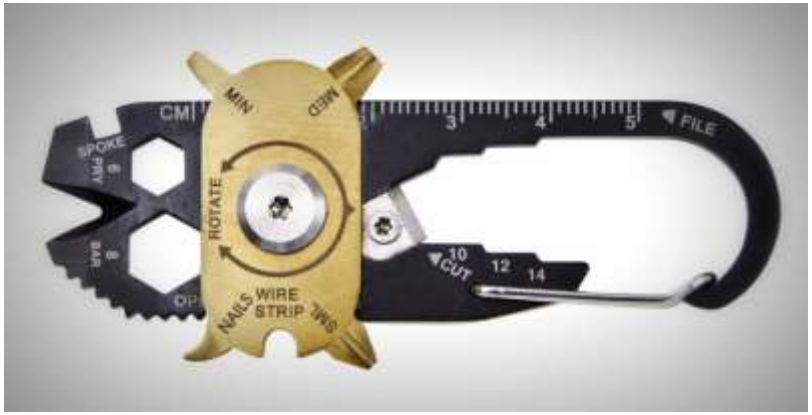
SOLID

- SOLID was introduced by Robert C. Martin in the an article called the “Principles of Object Oriented Design” in the early 2000s



SOLID – Single Responsibility Principle

- Every object should have a single responsibility, and all its services should be narrowly aligned with that responsibility



SOLID – SRP – Definitions

- “The Single Responsibility Principle states that every object should have a single responsibility, and that responsibility should be entirely encapsulated by the class.” – Wikipedia
- “There should never be more than one reason for a class to change.” – Robert Martin
- Low coupling & strong cohesion

SOLID – SRP – Problems & Solutions

- Classic violations
  - Objects that can print/draw themselves
  - Objects that can save/restore themselves
- Classic solution
  - Separate printer & Separate saver
- Solution
  - Multiple small interfaces (ISP)
  - Many small classes
  - Distinct responsibilities
- Result
  - Flexible design
  - Lower coupling & Higher cohesion

SOLID – SRP – Example

- Two responsabilities

```
interface Modem {
 public void dial(String pno);
 public void hangup();

 public void send(char c);
 public char recv();
}
```

- Separated interfaces

```
interface DataChannel {
 public void send(char c);
 public char recv();
}
```

```
interface Connection {
 public void dial(String phn);
 public char hangup();
}
```

SOLID – Open/Closed Principle

- *Open chest surgery is not needed when putting on a coat*
- Bertrand Meyer originated the OCP term in his 1988 book, *Object Oriented Software Construction*

SOLID – OCP – Definitions

- “The Open / Closed Principle states that software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.” – Wikipedia
- “All systems change during their life cycles. This must be borne in mind when developing systems expected to last longer than the first version.” – Ivar Jacobson
- **Open** to Extension – New behavior can be added in the future
- **Closed** to Modification – Changes to source or binary code are not required

SOLID – OCP – How?

- Change behavior without changing code?!
  - Rely on abstractions, not implementations
  - Do not limit the variety of implementations
- In .NET – Interfaces, Abstract Classes
- In procedural code – Use parameters
- Approaches to achieve OCP
  - Parameters – Pass delegates / callbacks
  - Inheritance / Template Method pattern – Child types override behavior of a base class
  - Composition / Strategy pattern – Client code depends on abstraction, "Plug in" model

SOLID – OCP – Problems & Solutions

- Classic violations
  - Each change requires re-testing (possible bugs)
  - Cascading changes through modules
  - Logic depends on conditional statements
- Classic solution
  - New classes (nothing depends on them yet)
  - New classes (no legacy coupling)
- When to apply OCP?
  - Experience tell you
- OCP add complexity to design (TANSTAAFL)
- No design can be closed against all changes

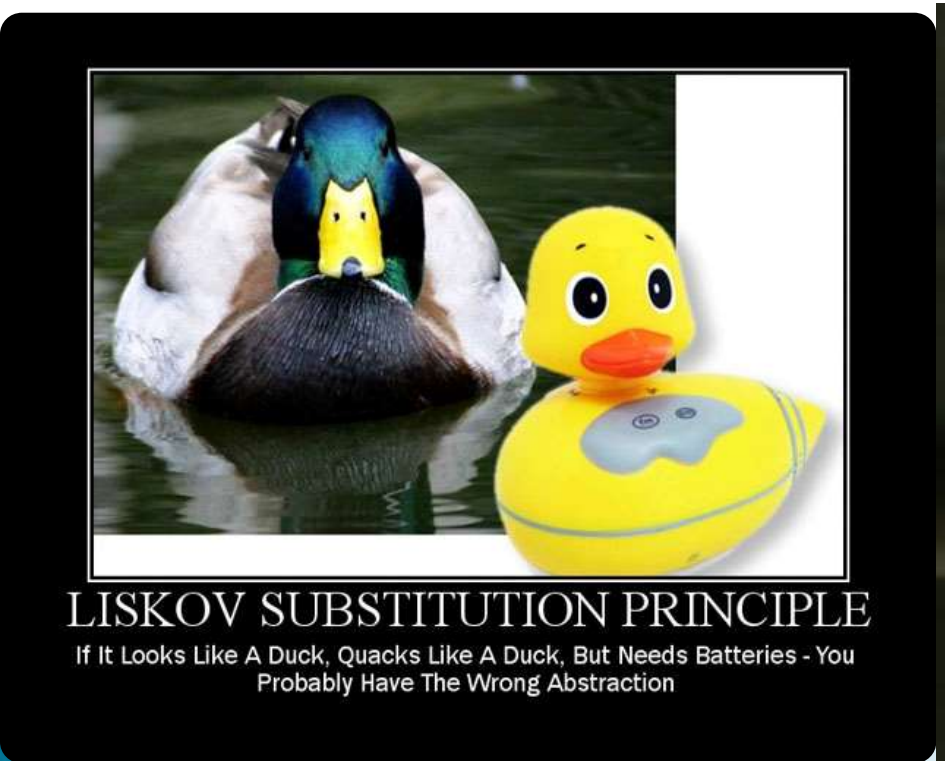
SOLID – OCP – Example

```
// Open-Close Principle - Bad example
class GraphicEditor {
 public void drawShape(Shape s) {
  if (s.m_type==1)
        drawRectangle(s);
  else if (s.m_type==2)
        drawCircle(s);
 }
 public void drawCircle(Circle r)
{....}
 public void drawRectangle(Rectangle r)
{....}
}

class Shape {
 int m_type;
}

class Rectangle extends Shape {
 Rectangle() {super.m_type=1;}
}

class Circle extends Shape {
 Circle() {super.m_type=2;}
}
```

```
// Open-Close Principle - Good
example
class GraphicEditor {
 public void drawShape(Shape s) {
        s.draw();
 }
}

class Shape {
        abstract void draw();
}

class Rectangle extends Shape {
 public void draw() {
        // draw the rectangle
 }
}
```

SOLID – Liskov Substitution

- If it looks like a duck, quacks like a duck, but needs batteries – you probably have the wrong abstraction
- Barbara Liskov described the principle in 1988



SOLID – LSP – Definitions

- "The Liskov Substitution Principle states that Subtypes must be substitutable for their base types.“ – Agile Principles, Patterns, and Practices in C#
- Substitutability – child classes must not
  - Remove base class behavior
  - Violate base class invariants
- Normal OOP inheritance
  - IS-A relationship
- Liskov Substitution inheritance
  - IS-SUBSTITUTABLE-FOR

SOLID – LSP – Problems & Solutions

- The problem
  - Polymorphism break Client code expectations
  - "Fixing" by adding if-then – nightmare (OCP)
- Classic violations
  - Type checking for different methods
  - Not implemented overridden methods
  - Virtual methods in constructor
- Solutions
  - “Tell, Don’t Ask” – Don’t ask for types and Tell the object what to do
  - Refactoring to base class- Common functionality and Introduce third class

SOLID – LSP – Example (1)

```
// Violation of Liskov's Substitution Principle
 class Rectangle{
   int m_width;
   int m_height;

   public void setWidth(int width){
         m_width = width;
   }

   public void setHeight(int h){
         m_height = ht;
   }

   public int getWidth(){
         return m_width;
   }

   public int getHeight(){
         return m_height;
   }

   public int getArea(){
         return m_width * m_height;
   }
 }
```

```
class Square extends Rectangle {
  public void setWidth(int width){
        m_width = width;
        m_height = width;
  }

  public void setHeight(int height){
        m_width = height;
        m_height = height;
  }
 }
```

SOLID – LSP – Example (2)

```
class LspTest
{
private static Rectangle getNewRectangle()
{
        // it can be an object returned by some factory ...
        return new Square();
}

public static void main (String args[])
{
        Rectangle r = LspTest.getNewRectangle();
        r.setWidth(5);
        r.setHeight(10);
// user knows that r it's a rectangle. It assumes that he's able to set the width
 and height as for the base class

        System.out.println(r.getArea());
        // now he's surprised to see that the area is 100 instead of 50.
}
}
```

SOLID – Interface Segregation

- You want me to plug this in. Where?

SOLID – ISP – Definitions

- “The Interface Segregation Principle states that Clients should not be forced to depend on methods they do not use.” – Agile Principles, Patterns, and Practices in C#
- Prefer small, cohesive interfaces – Interface is the interface type + All public members of a class
- Divide "fat" interfaces into smaller ones
  - “fat” interfaces means classes with useless methods, increased coupling, reduced flexibility and maintainability

SOLID – ISP – Problems & Solutions

- Classic violations
  - Unimplemented methods (also in LSP)
  - Use of only small portion of a class
- When to fix?
  - Once there is pain! Do not fix, if is not broken!
  - If the "fat" interface is yours, separate it to smaller ones
  - If the "fat" interface is not yours, use "Adapter" pattern
- Solutions
  - Small interfaces
  - Cohesive interfaces
  - Focused interfaces
  - Let the client define interfaces
  - Package interfaces with their implementation

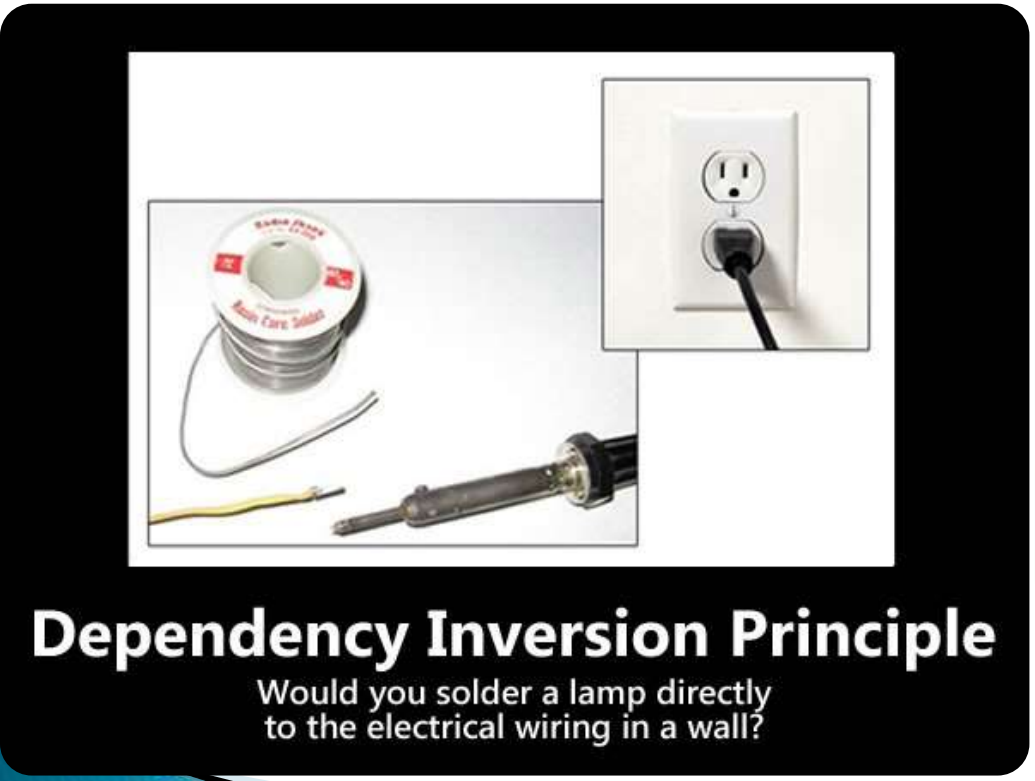
SOLID – ISP – Example

```
//Bad example (polluted interface)


interface Worker {
 void work();
 void eat();
}


ManWorker implements Worker {
        void work() {…};
        void eat() {30 min break;};
}


RobotWorker implements Worker {
        void work() {…};
        void eat() {//Not Appliciable
                for a RobotWorker};
}
```

```
//Solution: split into two interfaces


interface Workable {
        public void work();
 }

 interface Feedable{
        public void eat();
 }
```

SOLID – Dependency Inversion

- Would you solder a lamp directly to the electrical wiring in a wall?



SOLID – DIP – Definitions

- “High-level modules should not depend on low-level modules. Both should depend on abstractions.”

- “Abstractions should not depend on details. Details should depend on abstractions.” – Agile Principles, Patterns, and Practices in C#

SOLID – DIP – Dependency

- Framework
- Third Party Libraries
- Database
- File System
- Email
- Web Services
- System Resources (Clock)
- Configuration

- The new Keyword
- Static methods
- Thread.Sleep
- Random

SOLID – DIP – Problems & Solutions

- How it should be
  - Classes should declare what they need
  - Constructors should require dependencies
  - Dependencies should be abstractions and be shown
- How to do it
  - Dependency Injection
  - The Hollywood principle "Don't call us, we'll call you!"
- Classic violations
  - Using of the new keyword, static methods/properties
- How to fix?
  - Default constructor, main method/starting point
  - Inversion of Control container

SOLID – DIP – Example

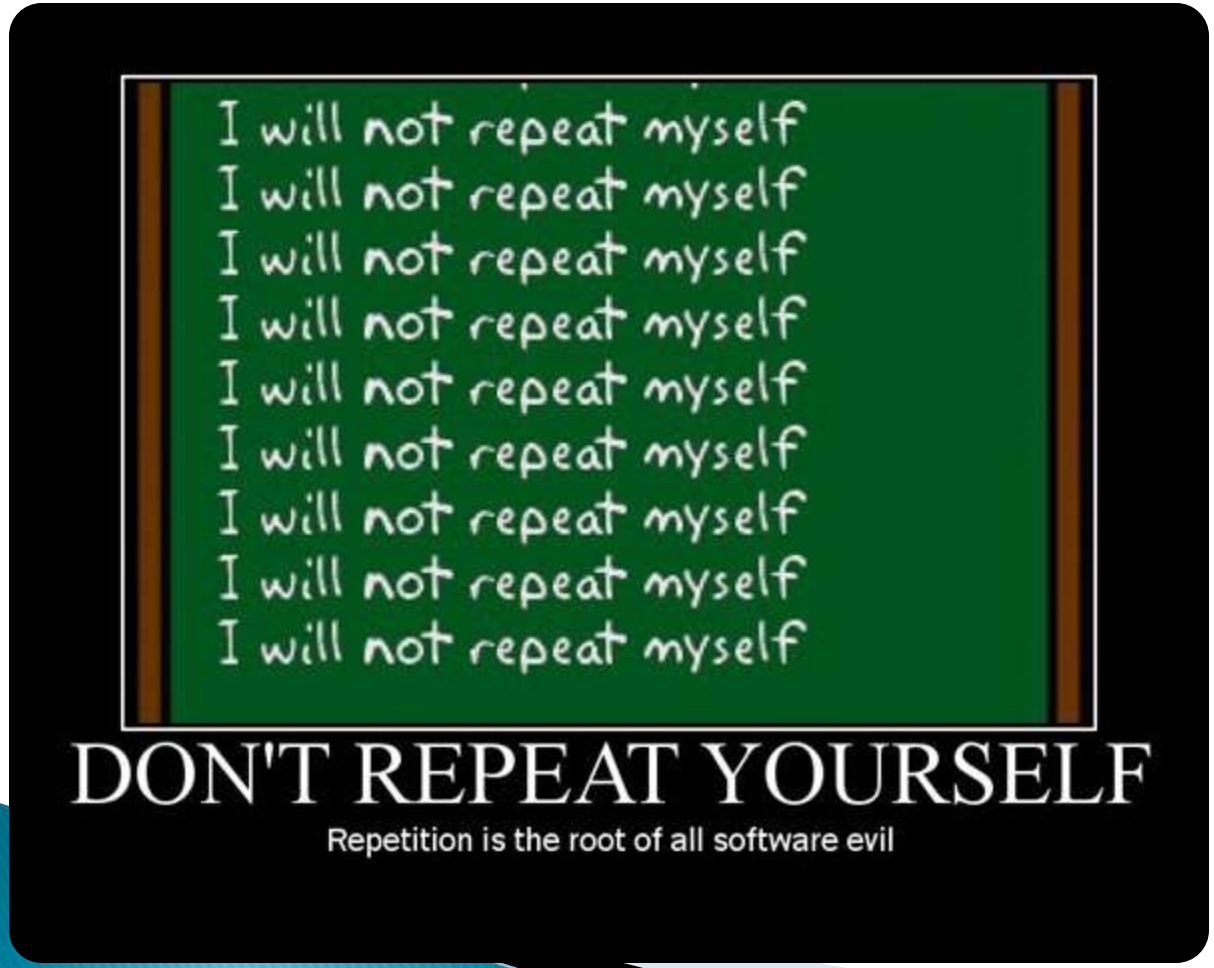
```
//DIP - bad example
public class EmployeeService {
        private EmployeeFinder emFinder //concrete class, not abstract. Can access a SQL DB for instance
        public Employee findEmployee(…) {
                emFinder.findEmployee(…)
        }
}

//DIP - fixed
public class EmployeeService {
        private IEmployeeFinder emFinder //depends on an abstraction, no an implementation
        public Employee findEmployee(…) {
                emFinder.findEmployee(…)
        }
}
//Now its possible to change the finder to be a XmlEmployeeFinder, DBEmployeeFinder, FlatFileEmployeeFinder,
MockEmployeeFinder….
```

Other Principles

- Don't Repeat Yourself (DRY)
- You Ain't Gonna Need It (YAGNI)
- Keep It Simple, Stupid (KISS)

OP – Don't Repeat Yourself

- Repetition is the root of all software evil



OP – DRY – Definitions

- "Every piece of knowledge must have a single, unambiguous representation in the system.“ – The Pragmatic Programmer

- "Repetition in logic calls for abstraction. Repetition in process calls for automation.“ – 97 Things Every Programmer Should Know

- Variations include:
  - Once and Only Once
  - Duplication Is Evil (DIE)

OP – DRY – Problems

- Magic Strings/Values
- Duplicate logic in multiple locations
- Repeated if-then logic
- Conditionals instead of polymorphism
- Repeated Execution Patterns
- Lots of duplicate, probably copy-pasted, code
- Only manual tests
- Static methods everywhere

OP – You Ain't Gonna Need It

- Don’t waste resources on what you might need

OP – YAGNI – Definitions

- "A programmer should not add functionality until deemed necessary.“ – Wikipedia

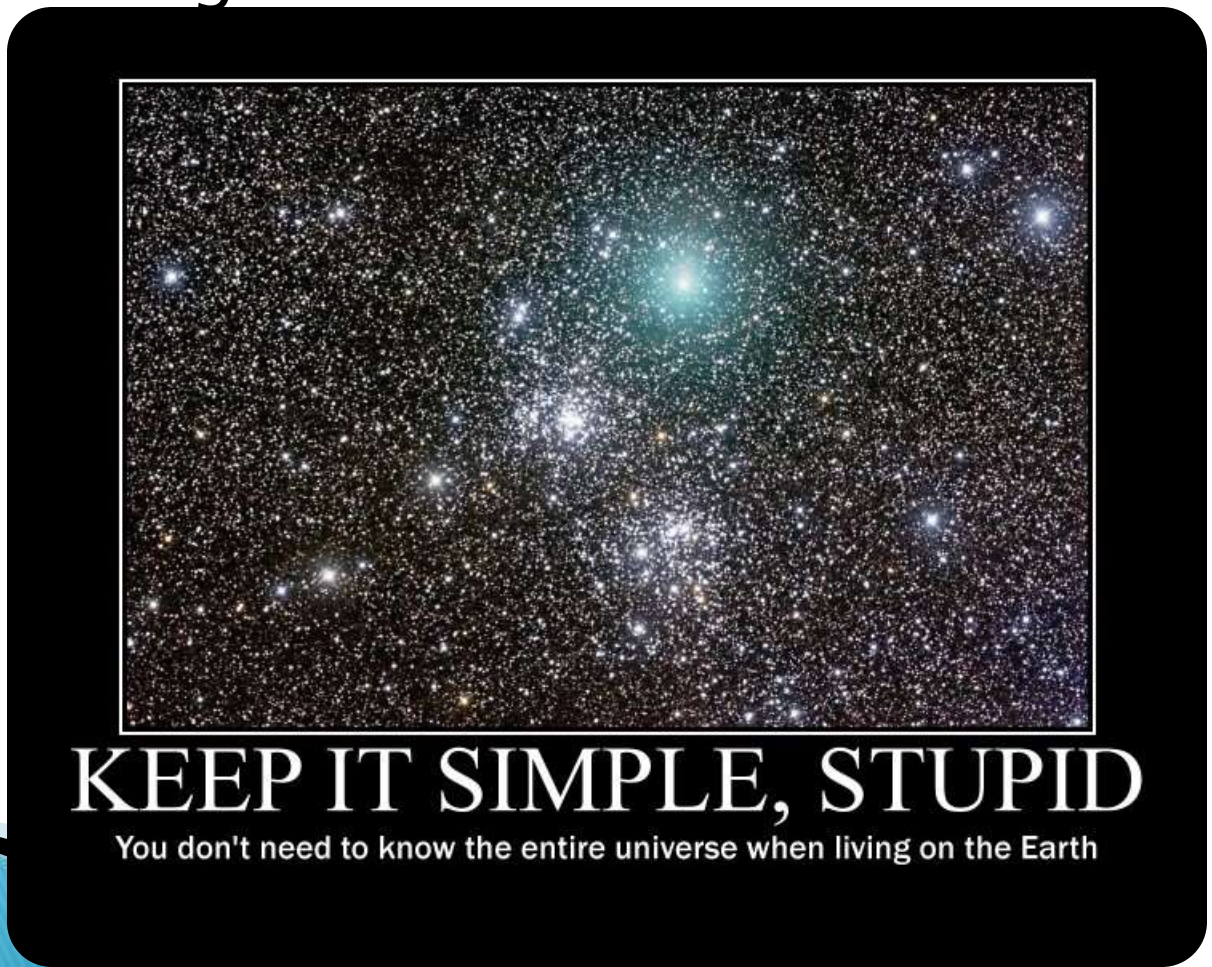
- "Always implement things when you actually need them, never when you just foresee that you need them.“ – Ron Jeffries, XP co-founder

OP – YAGNI – Problems

- Time for adding, testing, improving
- Debugging, documented, supported
- Difficult for requirements
- Larger and complicate software
- May lead to adding even more features
- May be not know to clients

OP – Keep It Simple, Stupid

- You don't need to know the entire universe when living on the Earth



OP – KISS – Definitions

- "Most systems work best if they are kept simple.“ – U.S. Navy
- "Simplicity should be a key goal in design and unnecessary complexity should be avoided.“ – Wikipedia

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

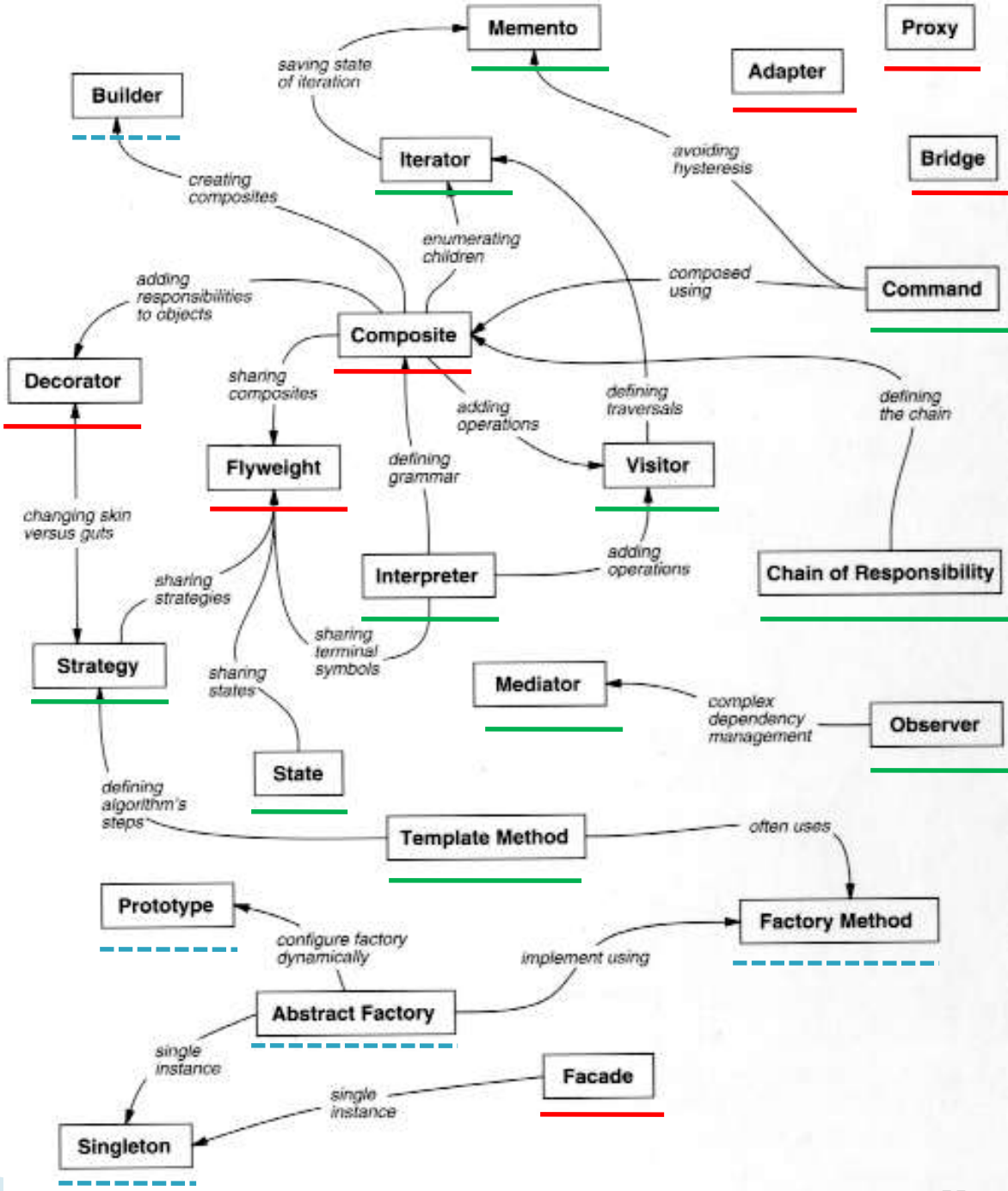
Behavioral patterns 1

- **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*

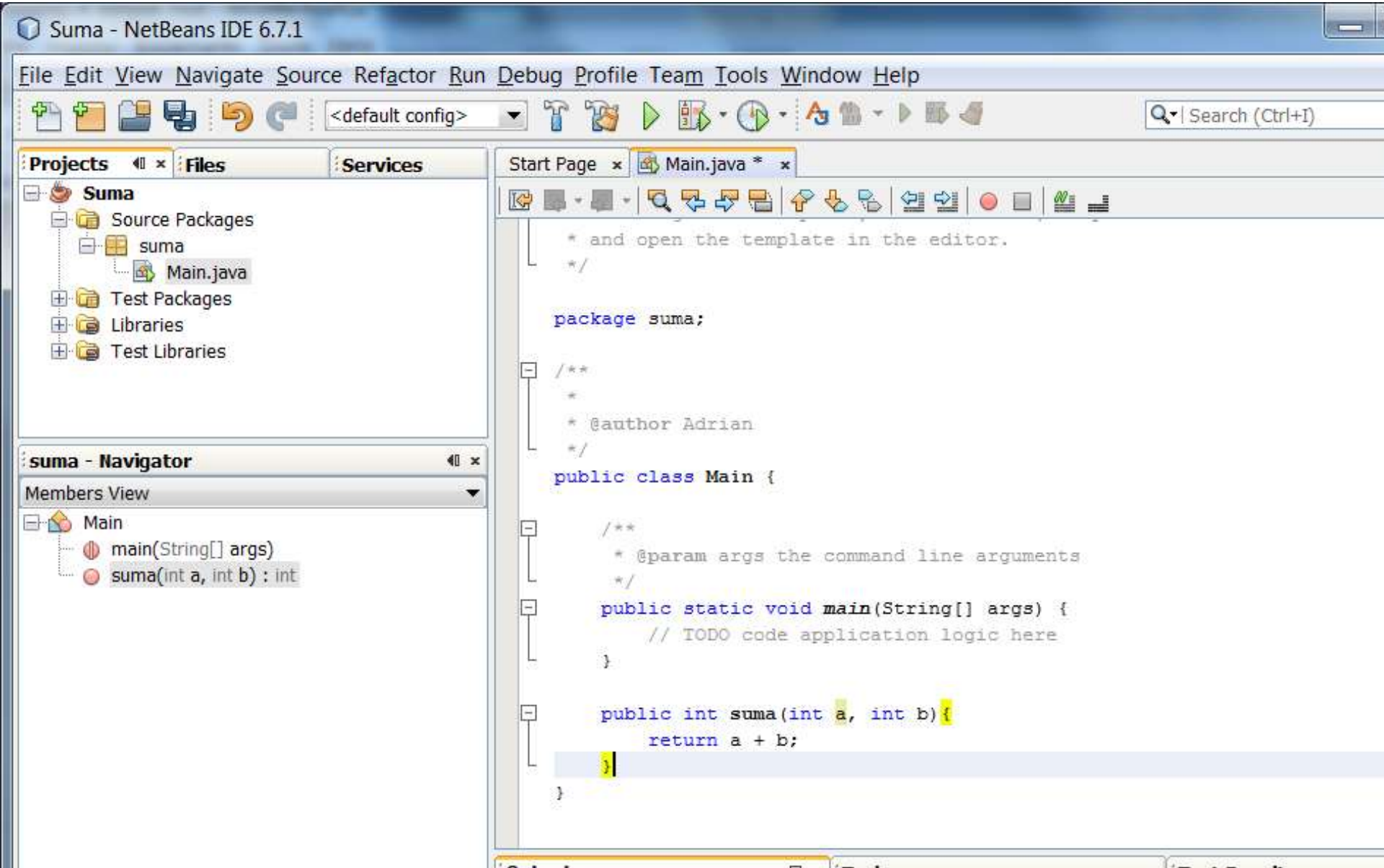
How to Use a Design Pattern?

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*

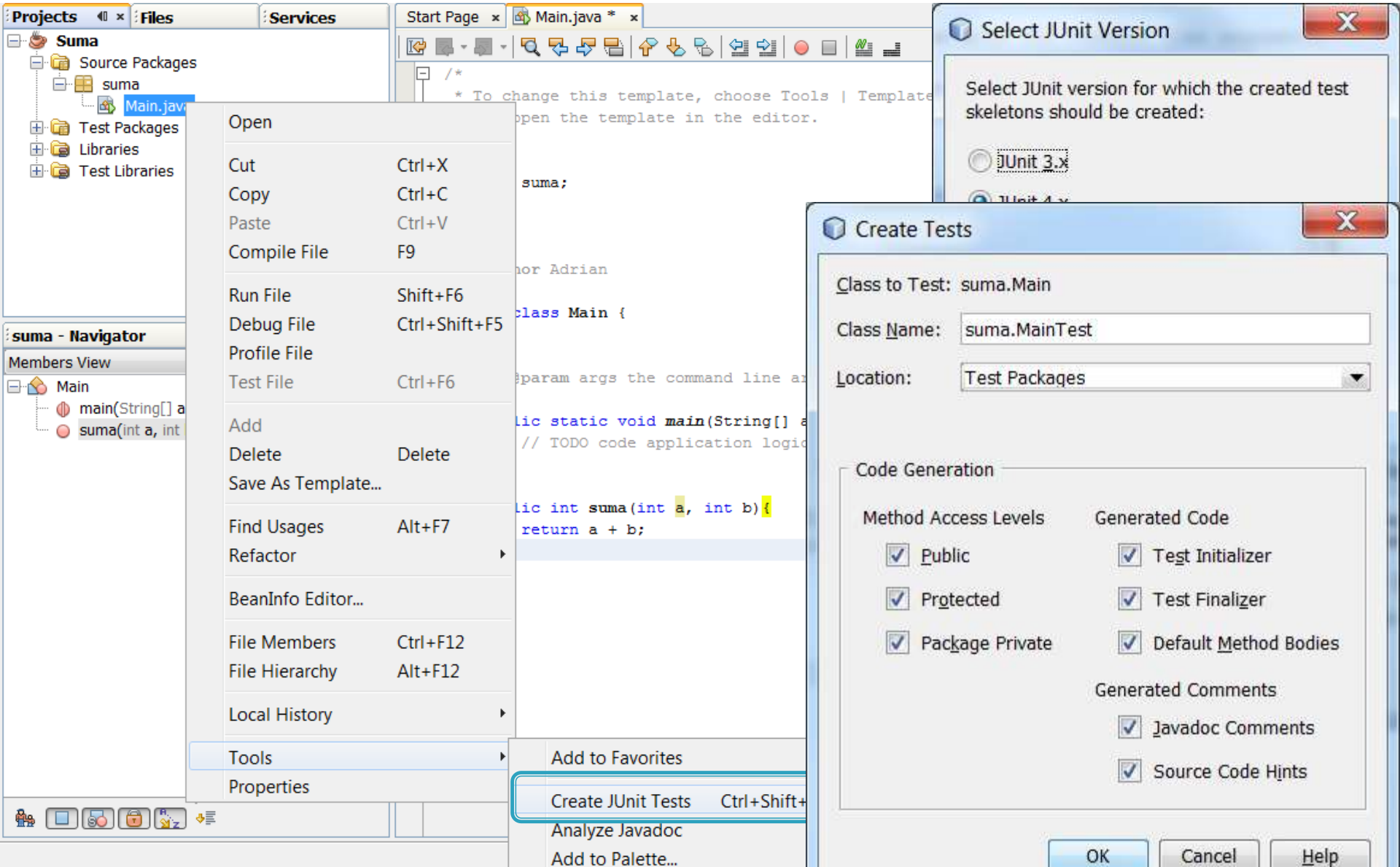
Unit Testing

- 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

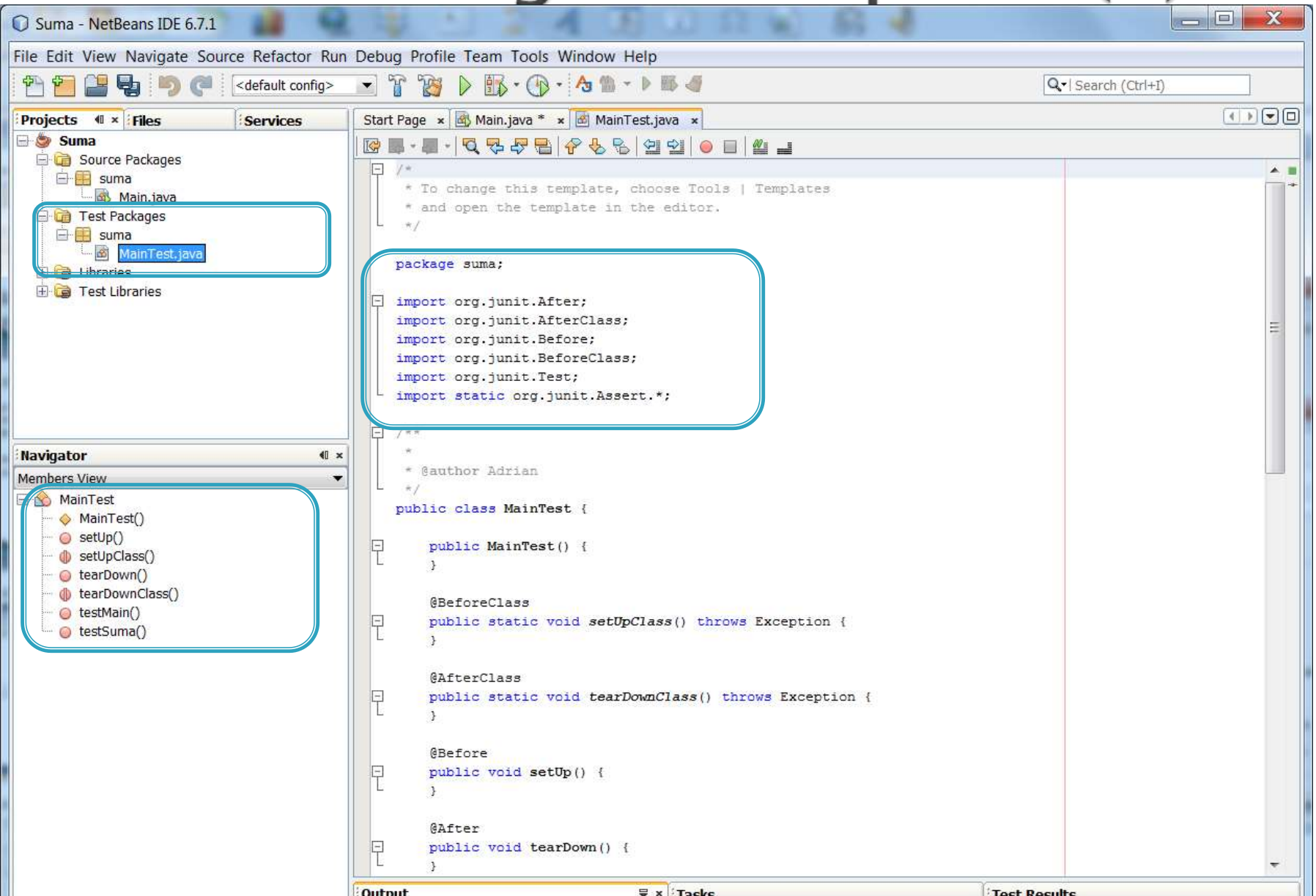
Unit Testing – Exemplu 1 (1)



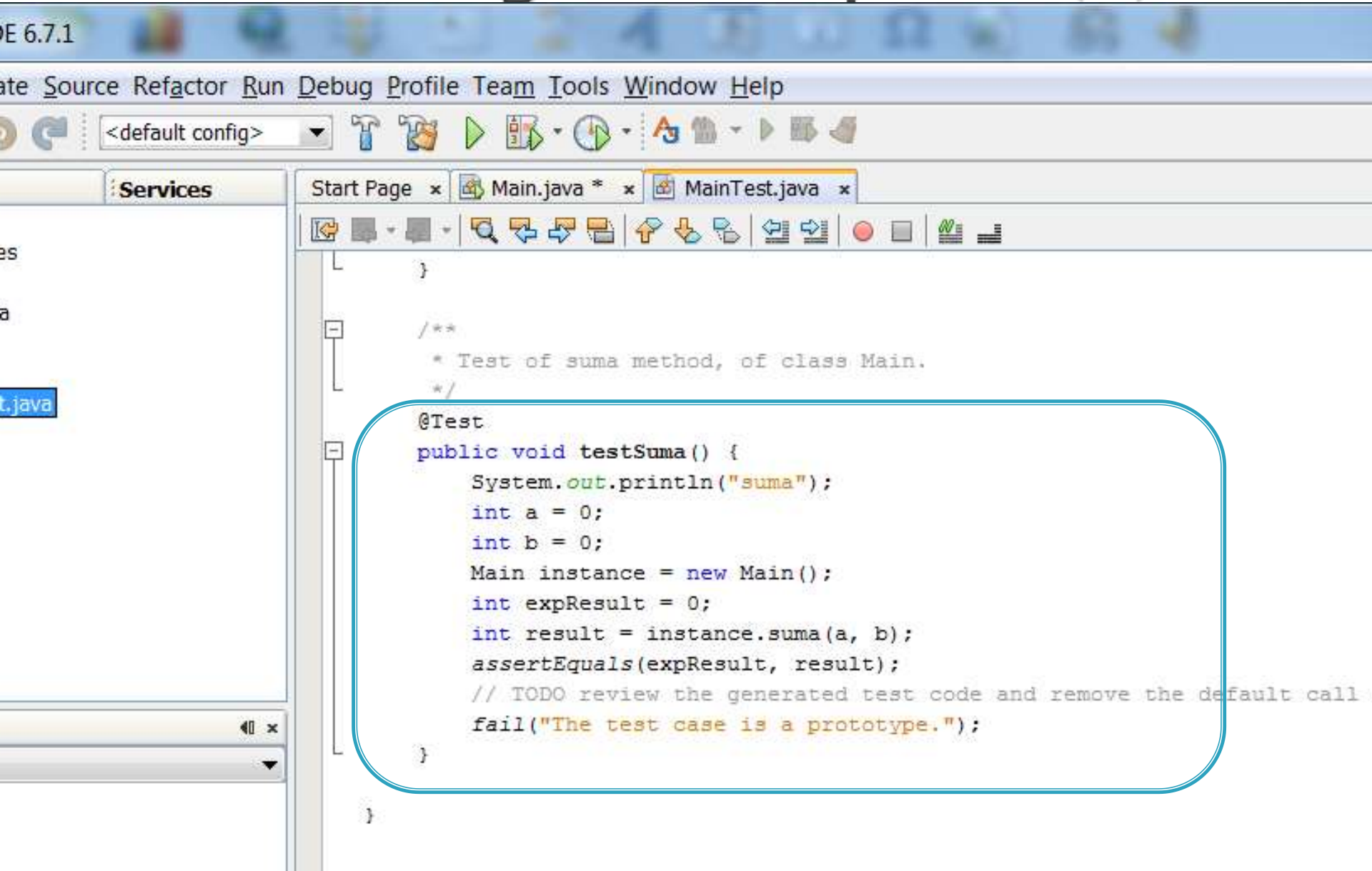
Unit Testing – Exemplu 1 (2)



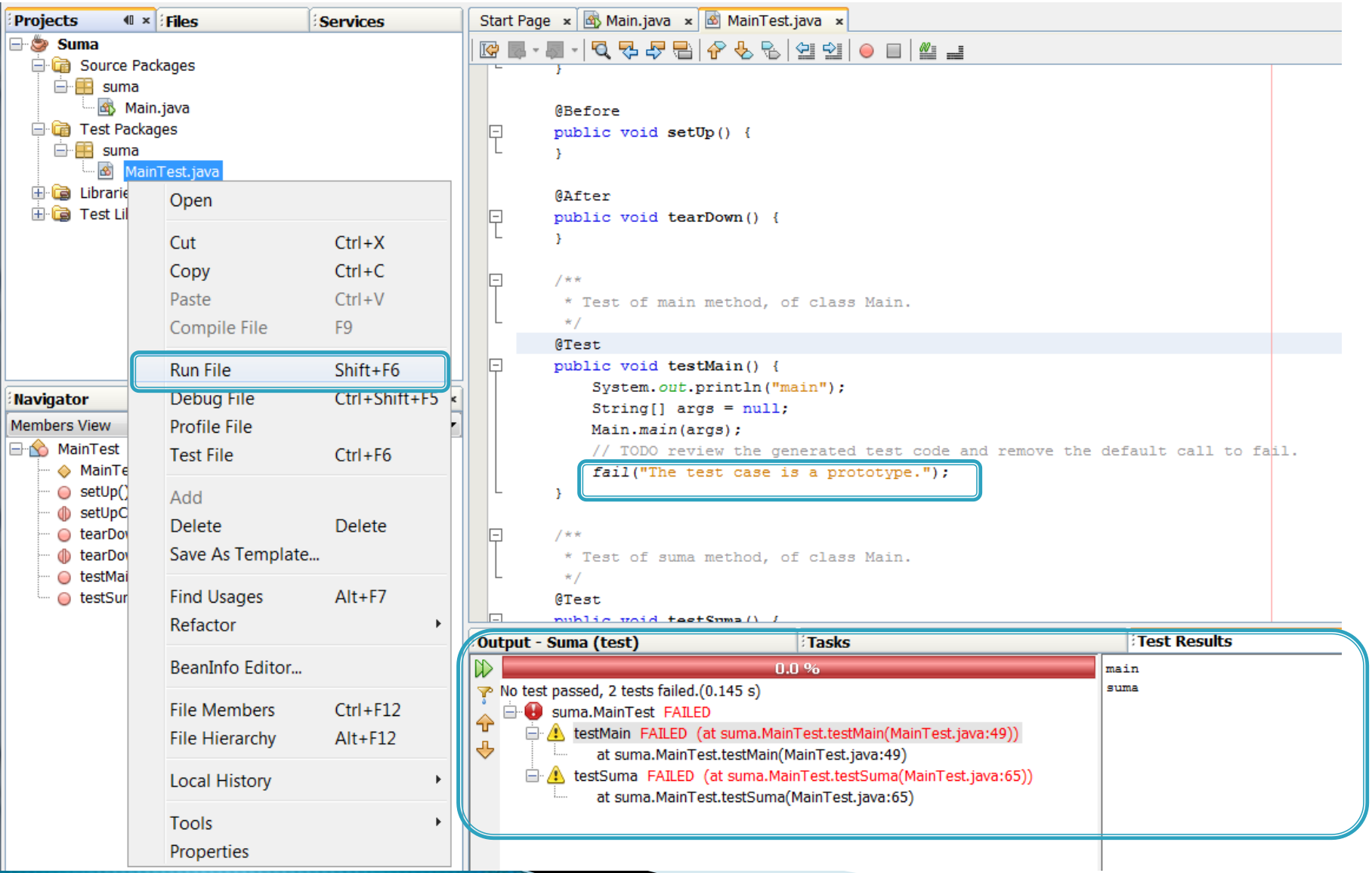
Unit Testing – Exemplu 1 (3)



Unit Testing – Exemplu 1 (4)

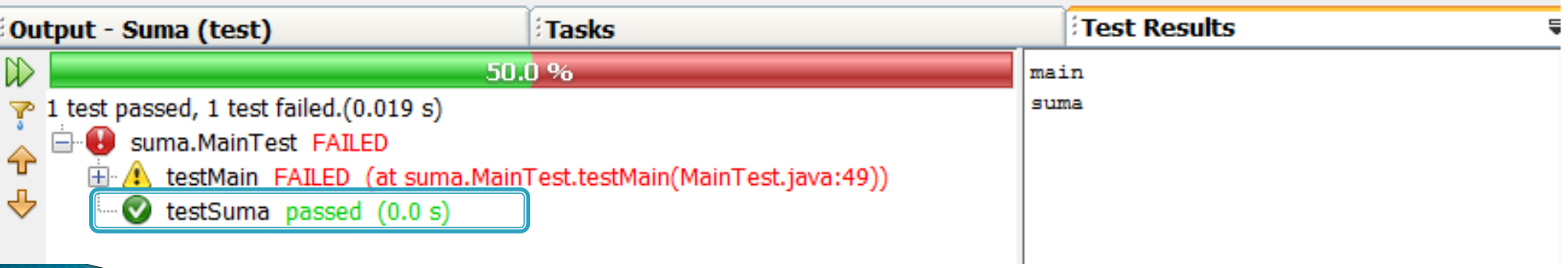


Unit Testing – Exemplu 1 (5)



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.");
    }
}
```

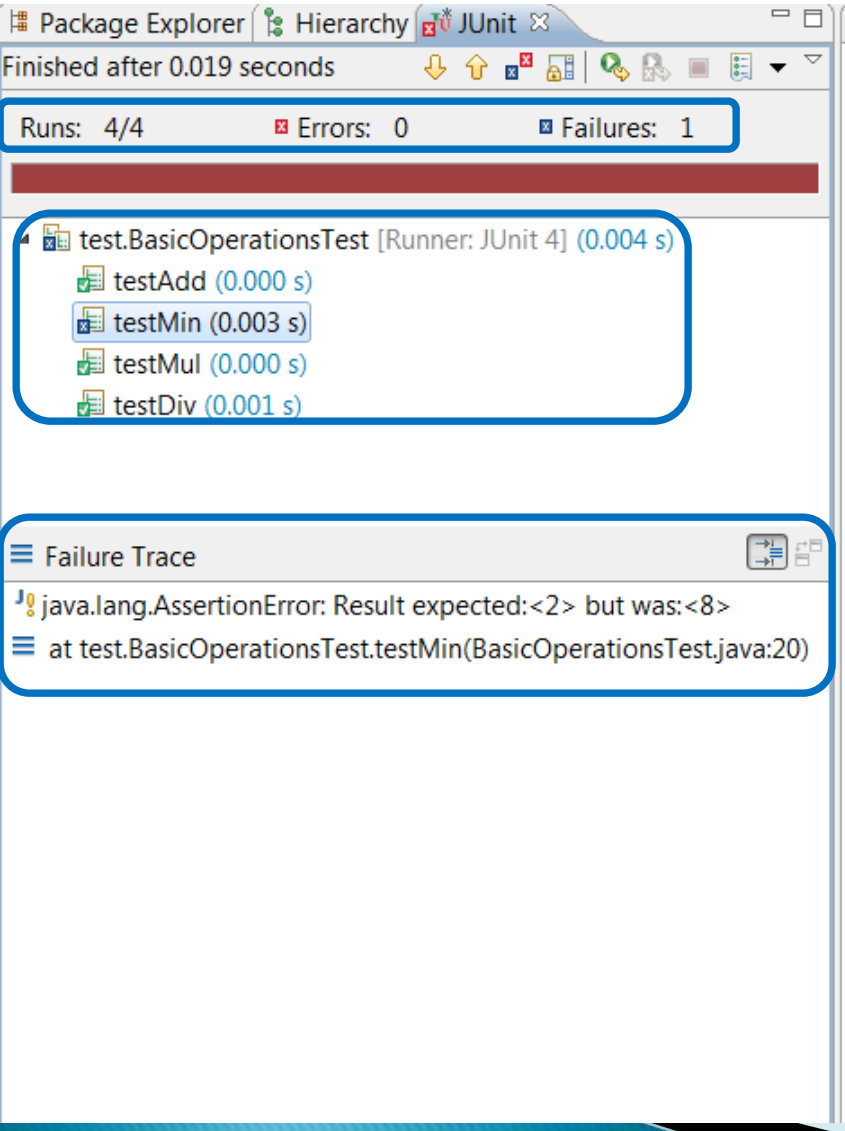


Unit Testing – Example 2 (1)

BasicOperations.java | BasicOperationsTest.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));
    }

}
```

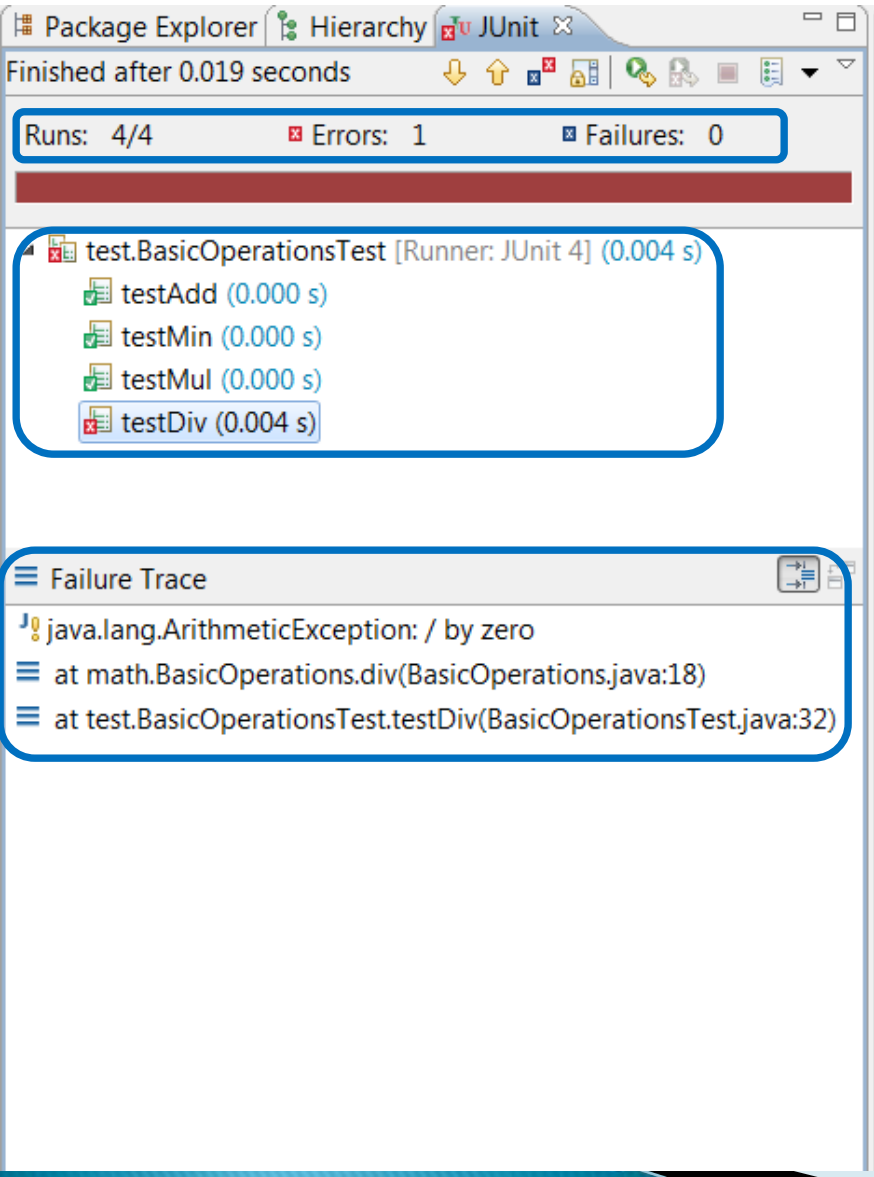
Unit Testing – Example 2 (2)



BasicOperations.java | BasicOperationsTest.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));
    }
}
```

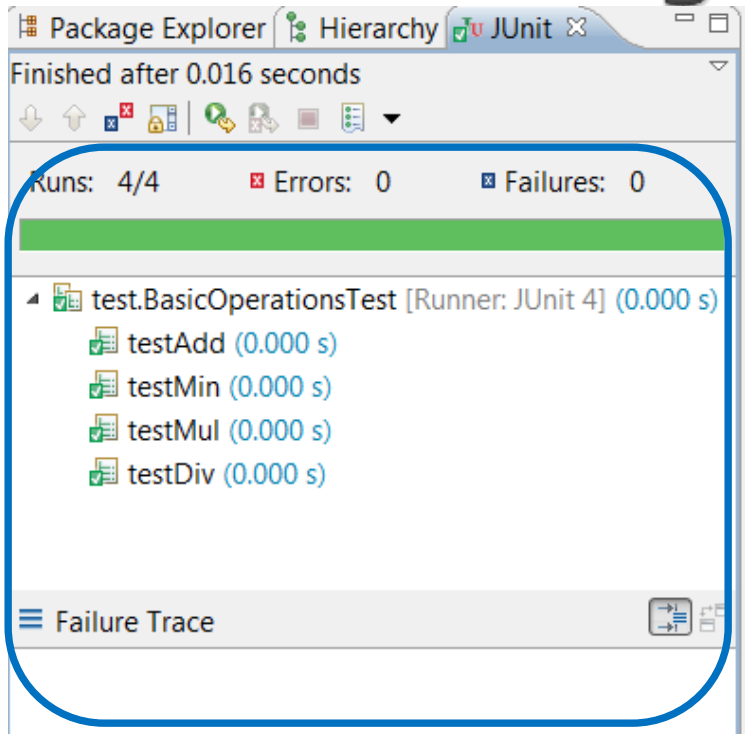
Unit Testing – Example 2 (3)



BasicOperations.java | BasicOperationsTest.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)



BasicOperations.java | BasicOperationsTest.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!");
    }
}
```

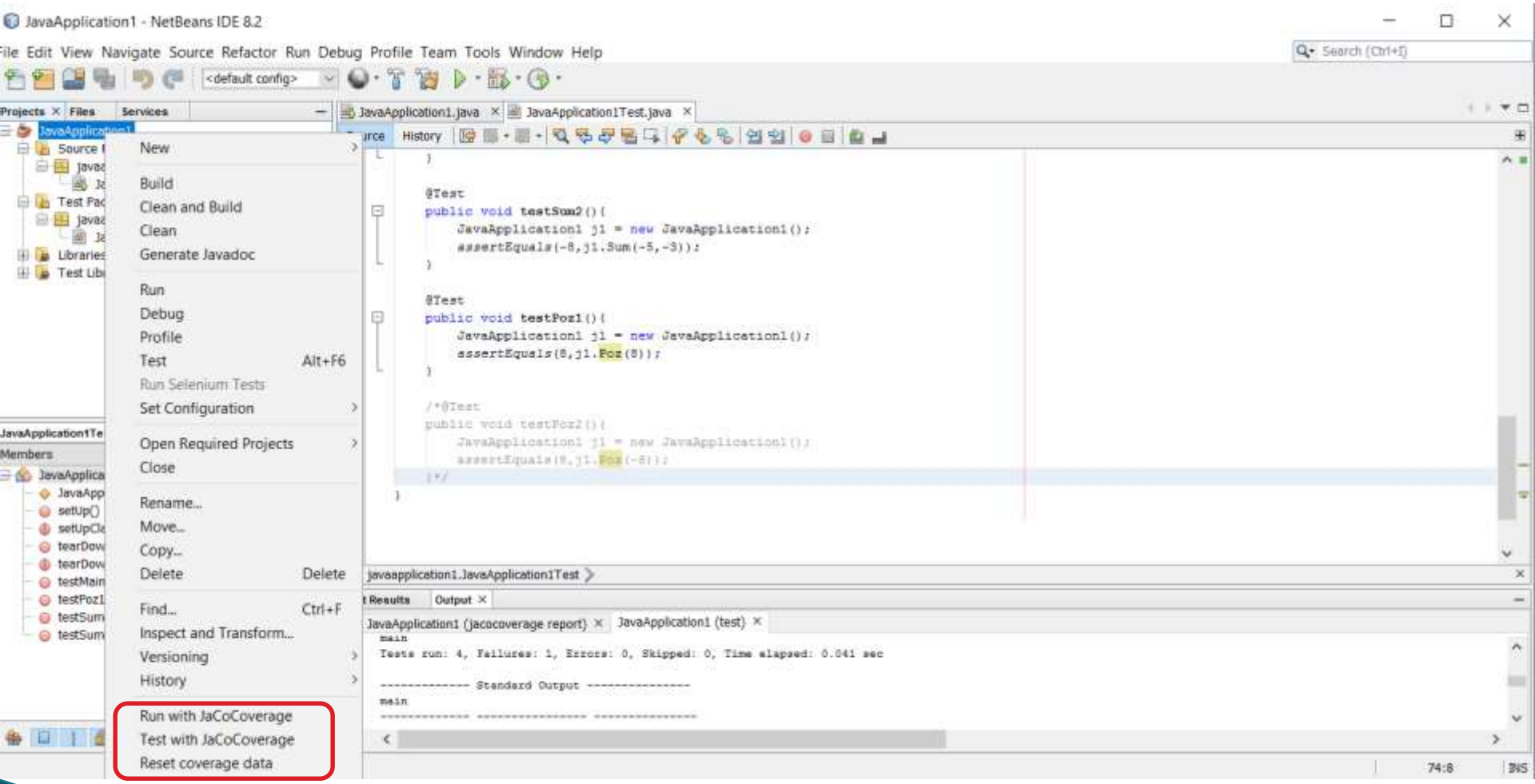
Code Coverage

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

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

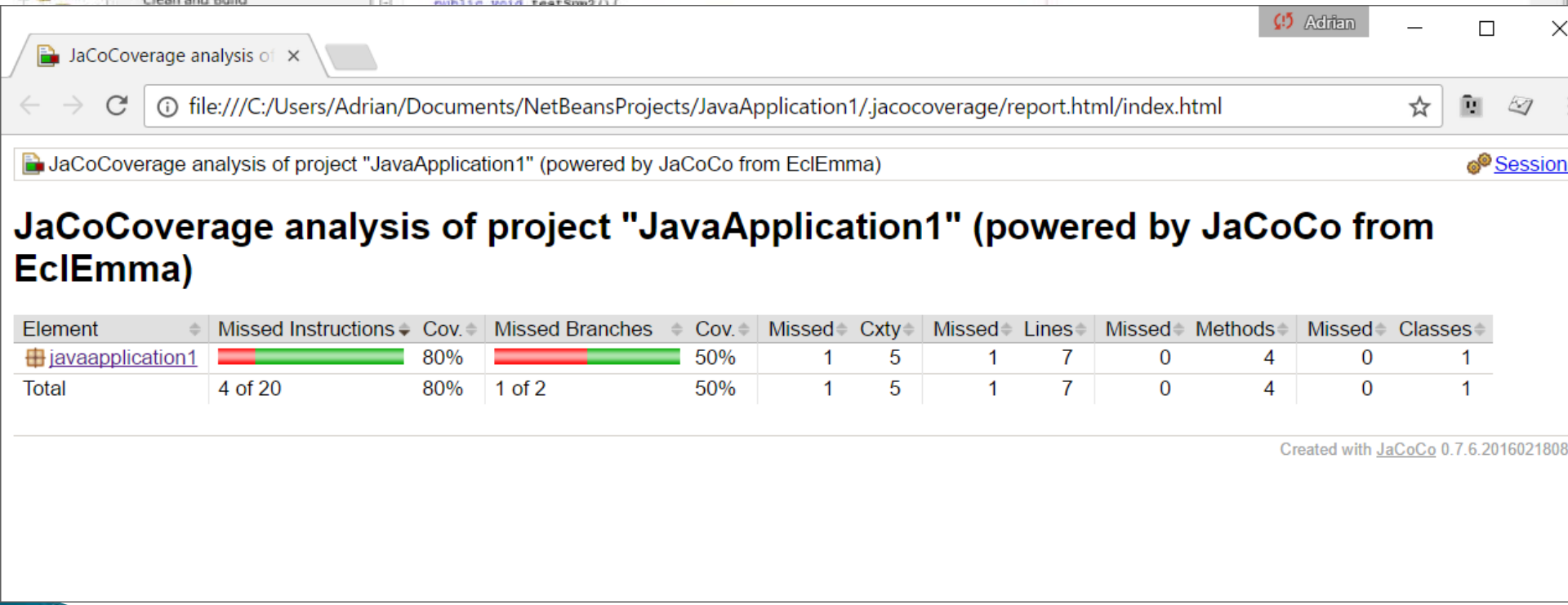
- IntelliJ – Running with coverage:
- https://www.jetbrains.com/help/idea/2016.3/running-with-coverage.html

NetBeans – 1

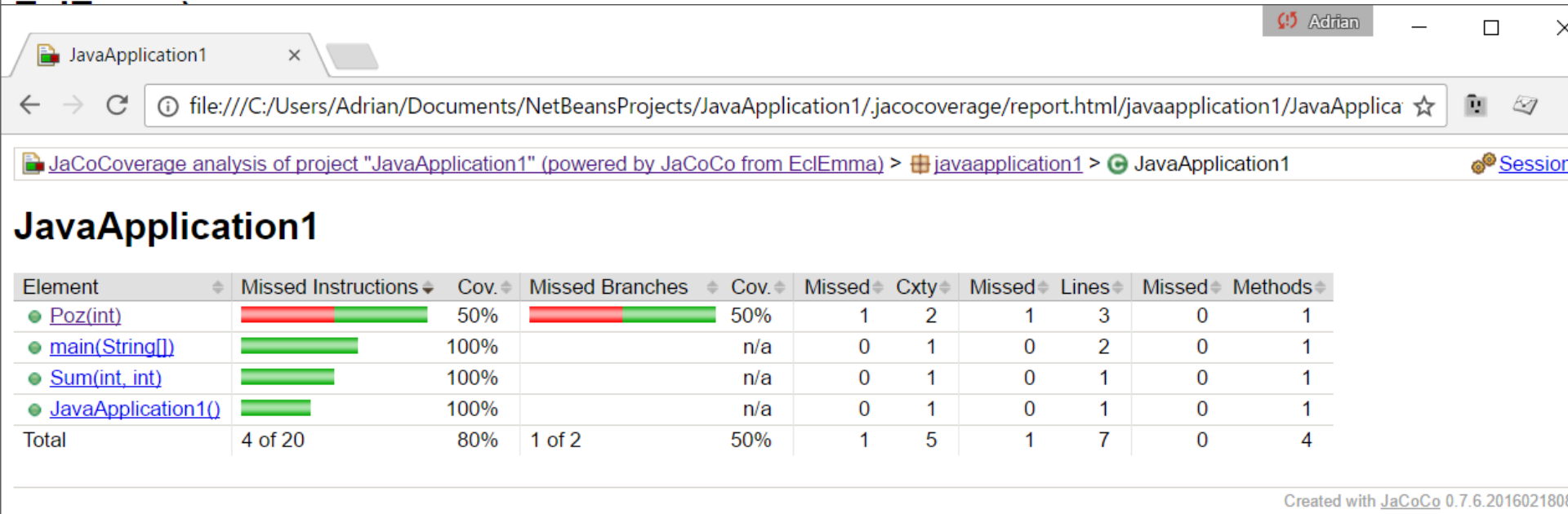

JavaApplication1 - NetBeans IDE 8.2
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Search (Ctrl+I)
<default config>
Projects
Files
Services
JavaApplication1.java
JavaApplication1Test.java
History
New
Build
Clean and Build
Clean
Generate Javadoc
Run
Debug
Profile
Test Alt+F6
Run Selenium Tests
Set Configuration
Open Required Projects
Close
Rename...
Move...
Copy...
Delete Delete
Find... Ctrl+F
Inspect and Transform...
Versioning
History
Run with JaCoCoverage
Test with JaCoCoverage
Reset coverage data
@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));
}
javaapplication1.JavaApplication1Test
Output
JavaApplication1 (jacocoverage report)
JavaApplication1 (test)
Tests run: 4, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.041 sec
Standard Output
74:8

NetBeans – 2


JavaApplication1 - NetBeans IDE 8.2
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
<default config>
Search (Ctrl+I)
Projects
Files
Services
JavaApplication1.java
JavaApplication1Test.java
History
New
Build
Clean and Build
@Test


Adrian
JaCoCoCoverage analysis o
file:///C:/Users/Adrian/Documents/NetBeansProjects/JavaApplication1/.jacocoverage/report.html/index.html
JaCoCoCoverage analysis of project "JavaApplication1" (powered by JaCoCo from EclEmma)
Session
JaCoCoCoverage 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 0.7.6.201602180812

NetBeans – 3



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 0.7.6.201602180

Concluzii

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

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