Ingineria Programării

Cursul 6 – 24 Martie

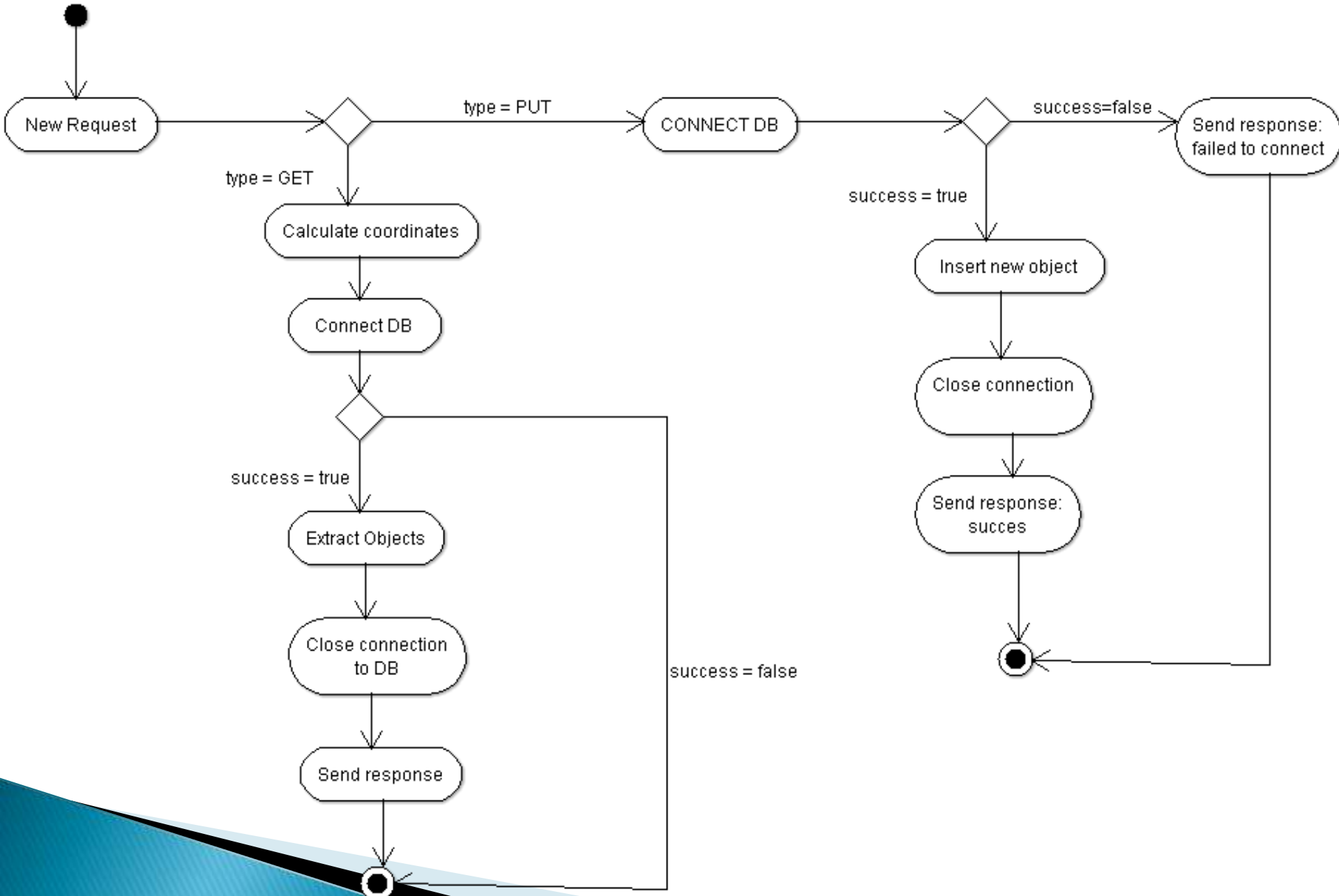
adiftene@infoiasi.ro

Cuprins

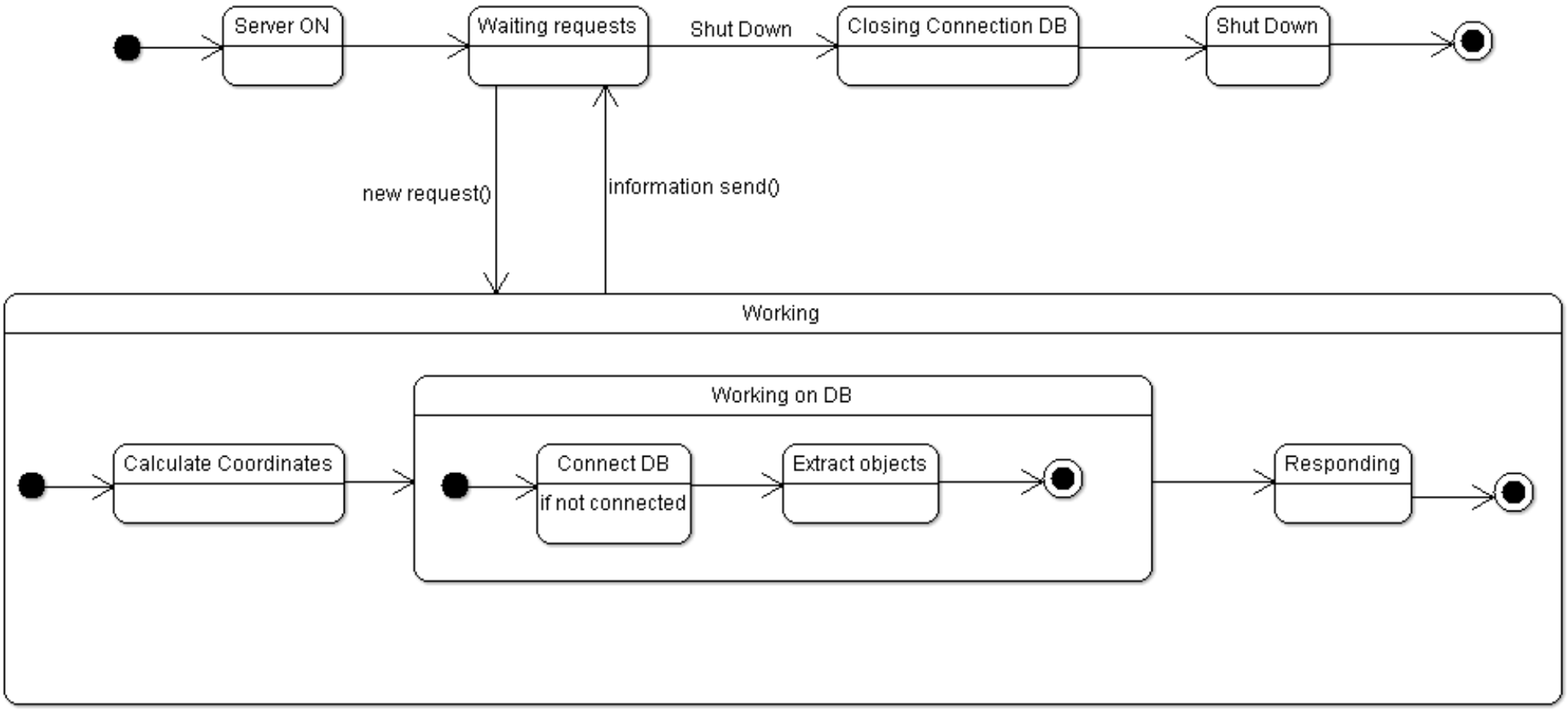
- Din Cursurile trecute…
- Design Patterns
  - Definitions
  - Elements
  - Example
  - Classification
- Creational Patterns
  - Abstract Factory
  - Builder
  - Factory Method
  - Prototype
  - Singleton
- JUnit Testing
  - Netbeans (Exemplu 1)
  - Eclipse (Exemplu 2)

Din Cursurile Trecute

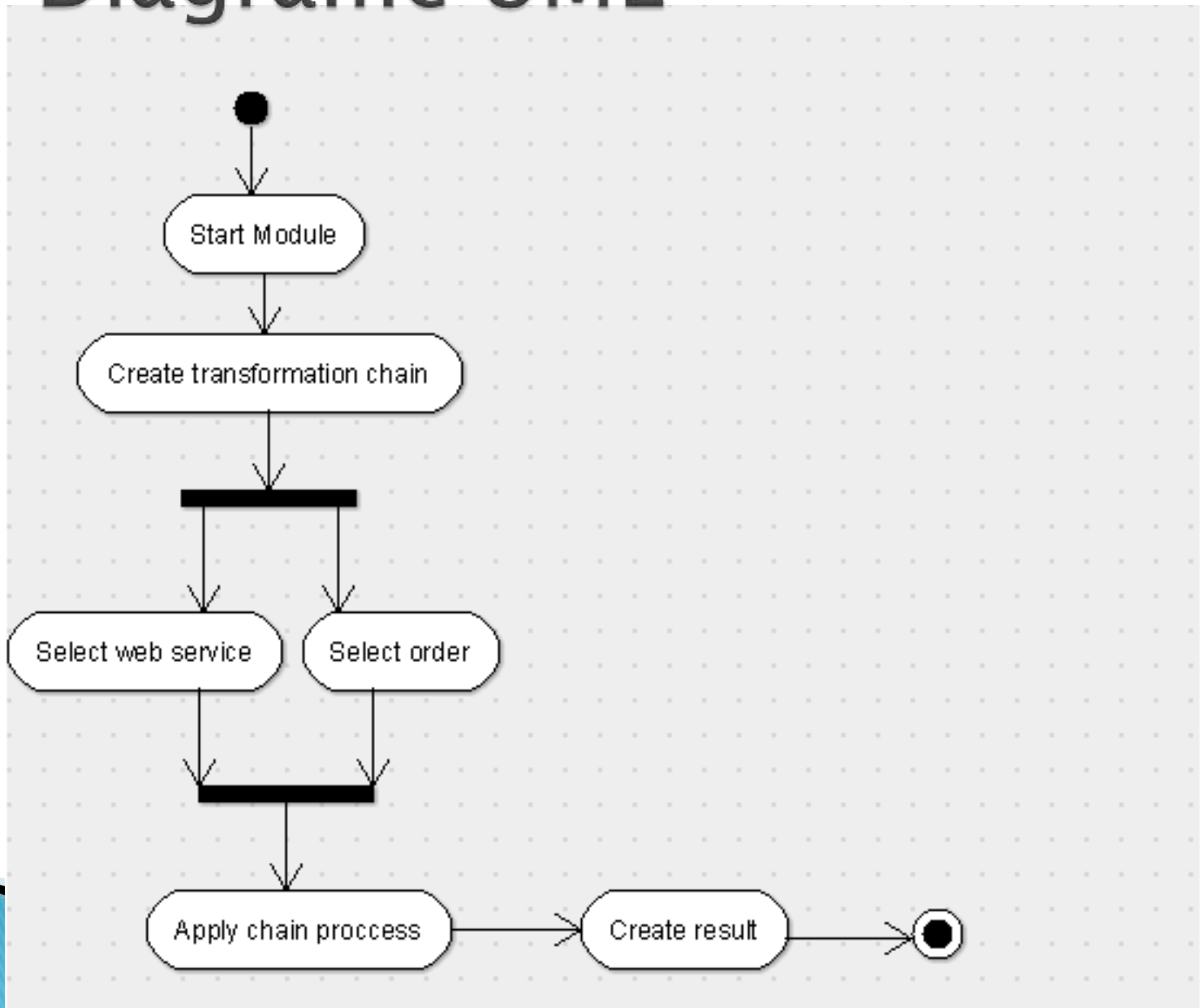
- Etapele Dezvoltării Programelor
- Ingineria Cerinţelor
- Diagrame UML
- GRASP


New Request
type = PUT
CONNECT DB
success=false
Send response: failed to connect
type = GET
success = true
Calculate coordinates
Insert new object
Connect DB
Close connection
success = true
Send response: succes
Extract Objects
Close connection to DB
success = false
Send response

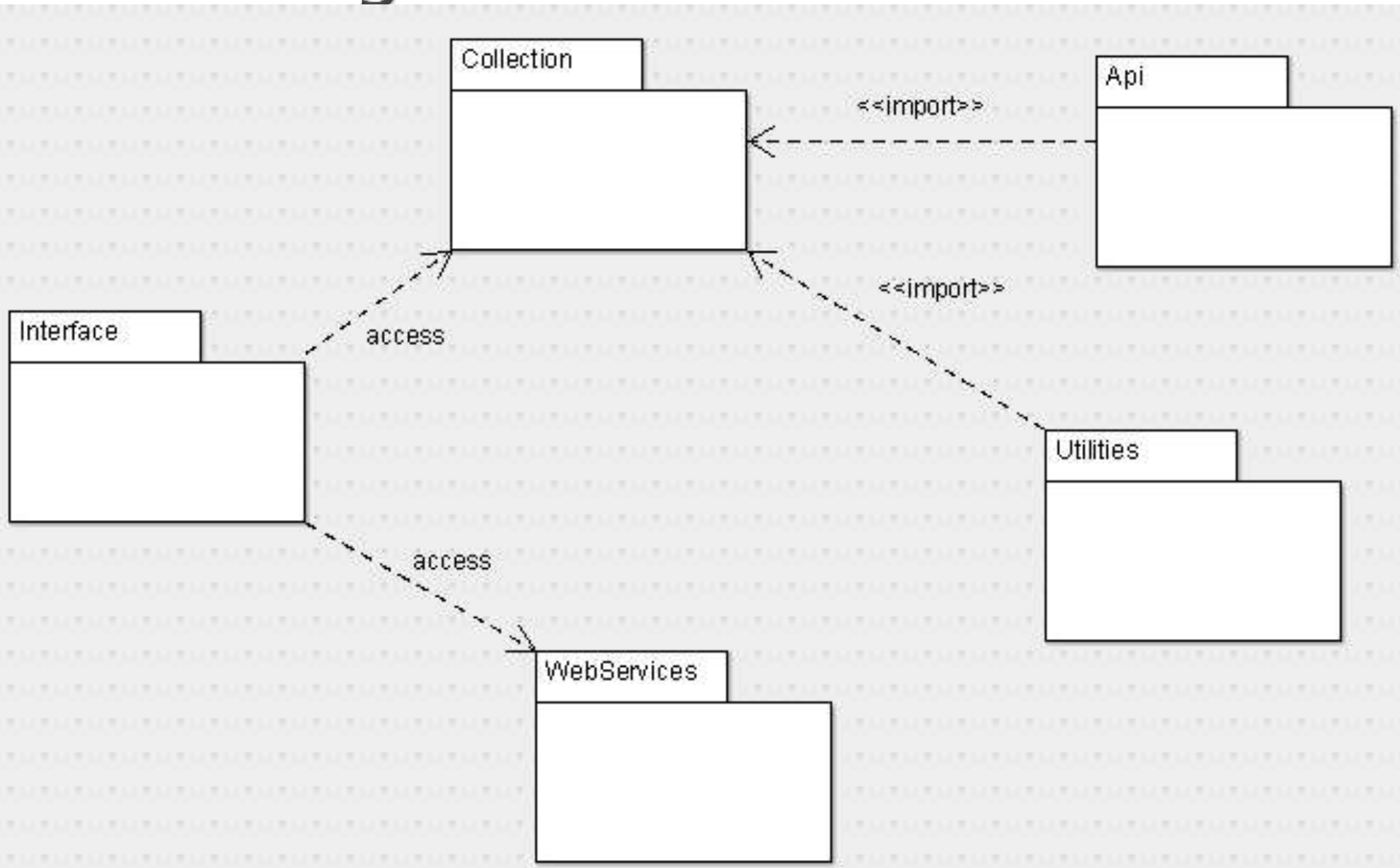
R – Diagrame UML



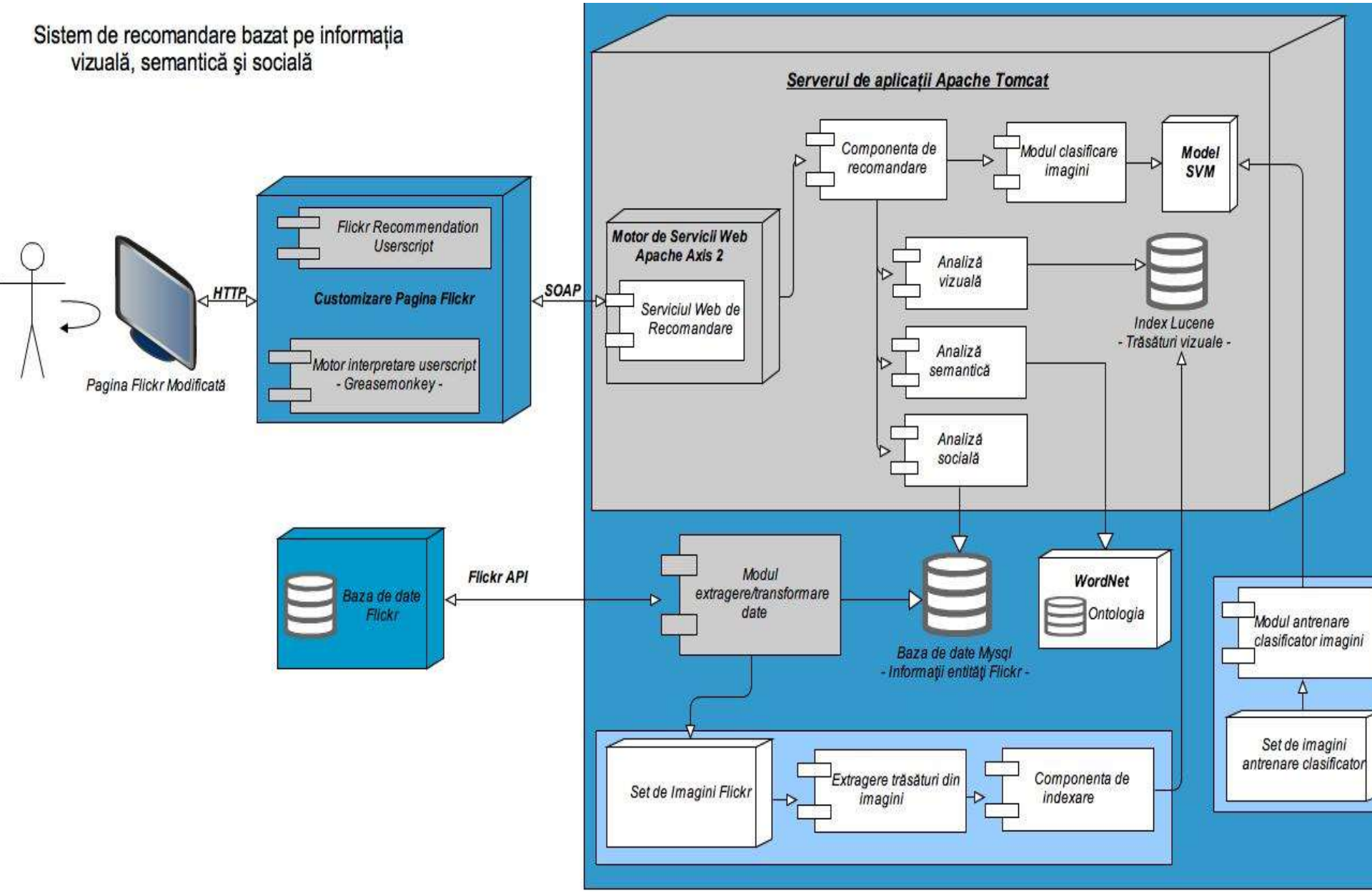
R – Diagrame UML



R – Diagrame UML



Sistem de recomandare bazat pe informația vizuală, semantică şi socială



R – GRASP

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

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

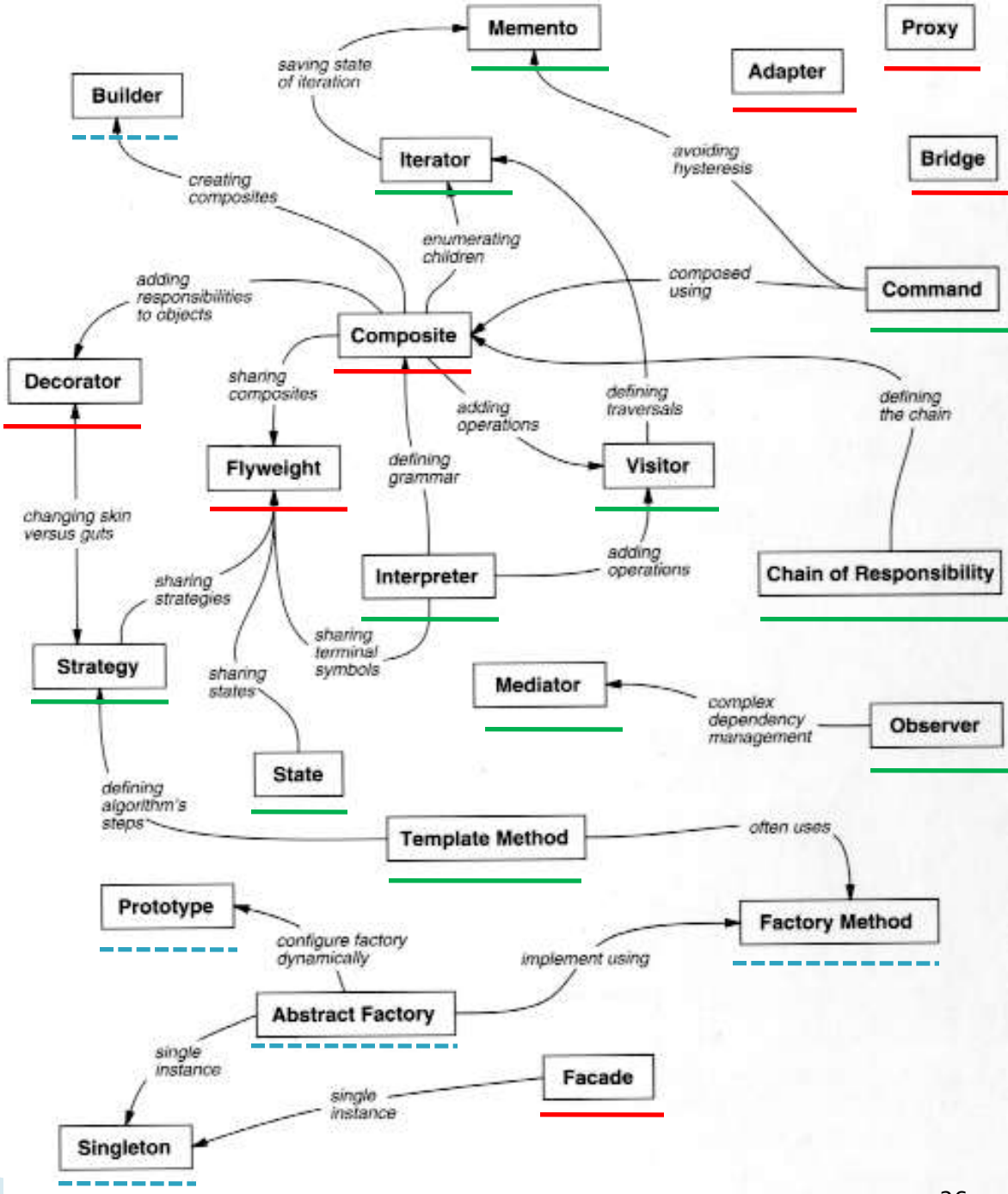
Behavioral patterns 1

- **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*

Creational Patterns

- **Abstract Factory**
- **Builder**
- **Factory Method**
- **Prototype**
- **Singleton**

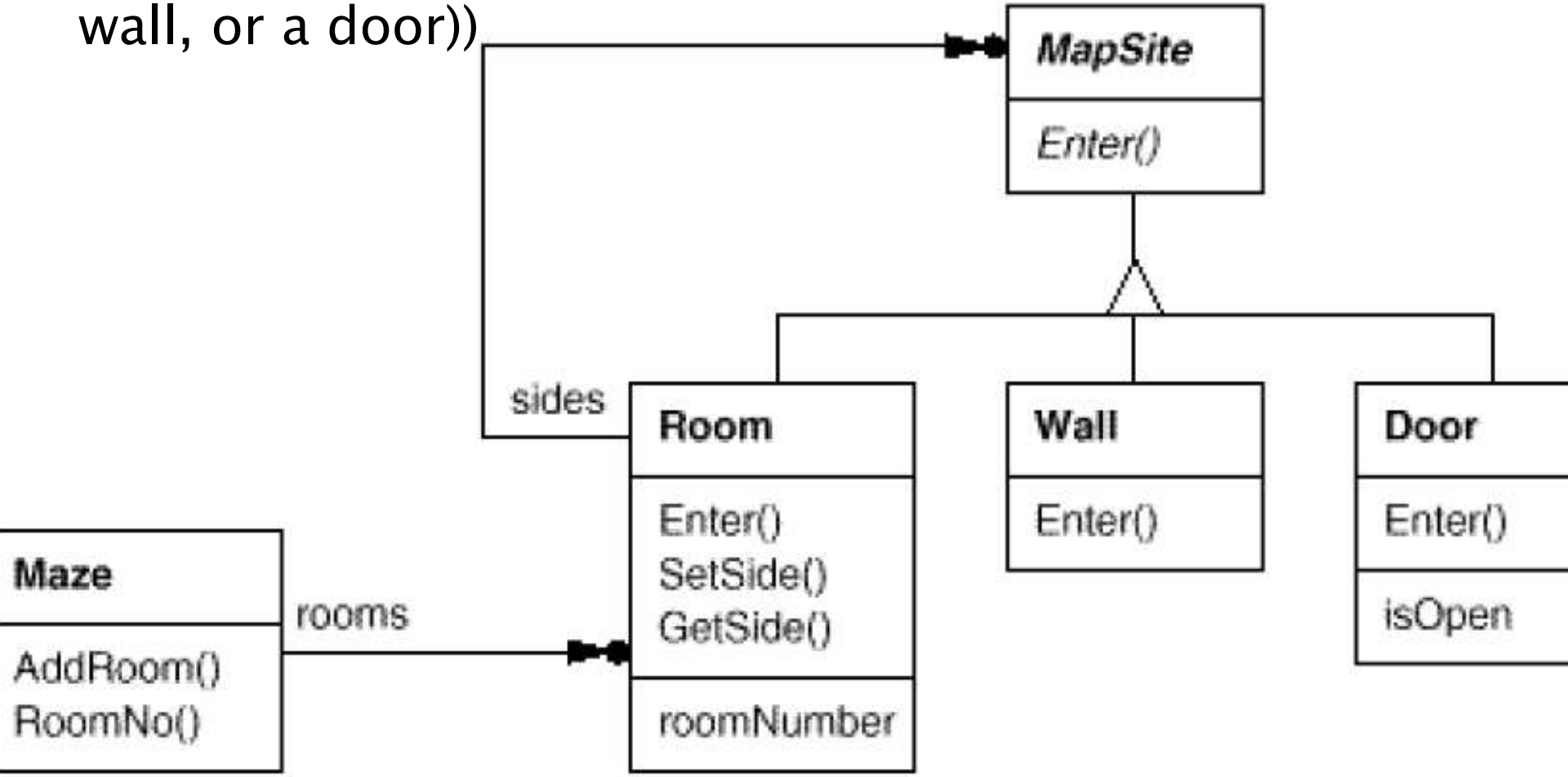
- Not in GOF book: Lazy initialization, Object pool, Multiton, Resource acquisition

Creational Patterns – Introduction

- Abstract the instantiation process
- Help **make a system independent of how its objects are created, composed, and represented**
- There are two recurring themes in these patterns:
  - First, they all encapsulate knowledge about which concrete classes the system uses
  - Second, they hide how instances of these classes are created and put together
- The creational patterns give a lot of flexibility in *what* gets created, *who creates it, how it gets created, and when*

Creational Patterns – Example 1

- Game – find your way out of a maze (maze = set of rooms. A room knows its neighbors (another room, a wall, or a door))



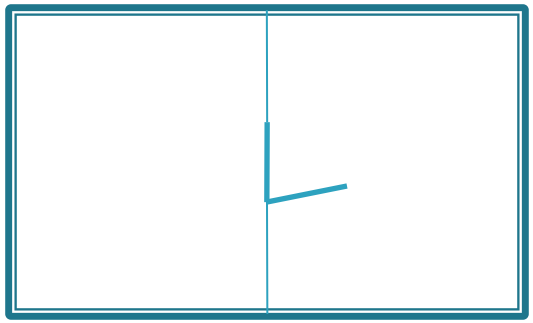
Creational Patterns – Example 1'

- MapSite is the common abstract class for all the components of a maze and defines *Enter*
- **If you enter a room**, then your location changes
- **If you try to enter a door**, then one of two things happen: If the door is open, you go into the next room. If the door is closed, then you hurt your nose

```
class MapSite {
  public:
  virtual void Enter() = 0;
};
```

Creational Patterns – Example 1”

```
Maze* MazeGame::CreateMaze () {
    Maze* aMaze = new Maze;
    Room* r1 = new Room(1);
    Room* r2 = new Room(2);
    Door* theDoor = new Door(r1, r2);
    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);

    r1->SetSide(North, new Wall);
    r1->SetSide(East, theDoor);
    r1->SetSide(South, new Wall);
    r1->SetSide(West, new Wall);

    r2->SetSide(North, new Wall);
    r2->SetSide(East, new Wall);
    r2->SetSide(South, new Wall);
    r2->SetSide(West, theDoor);
    return aMaze;
}
```

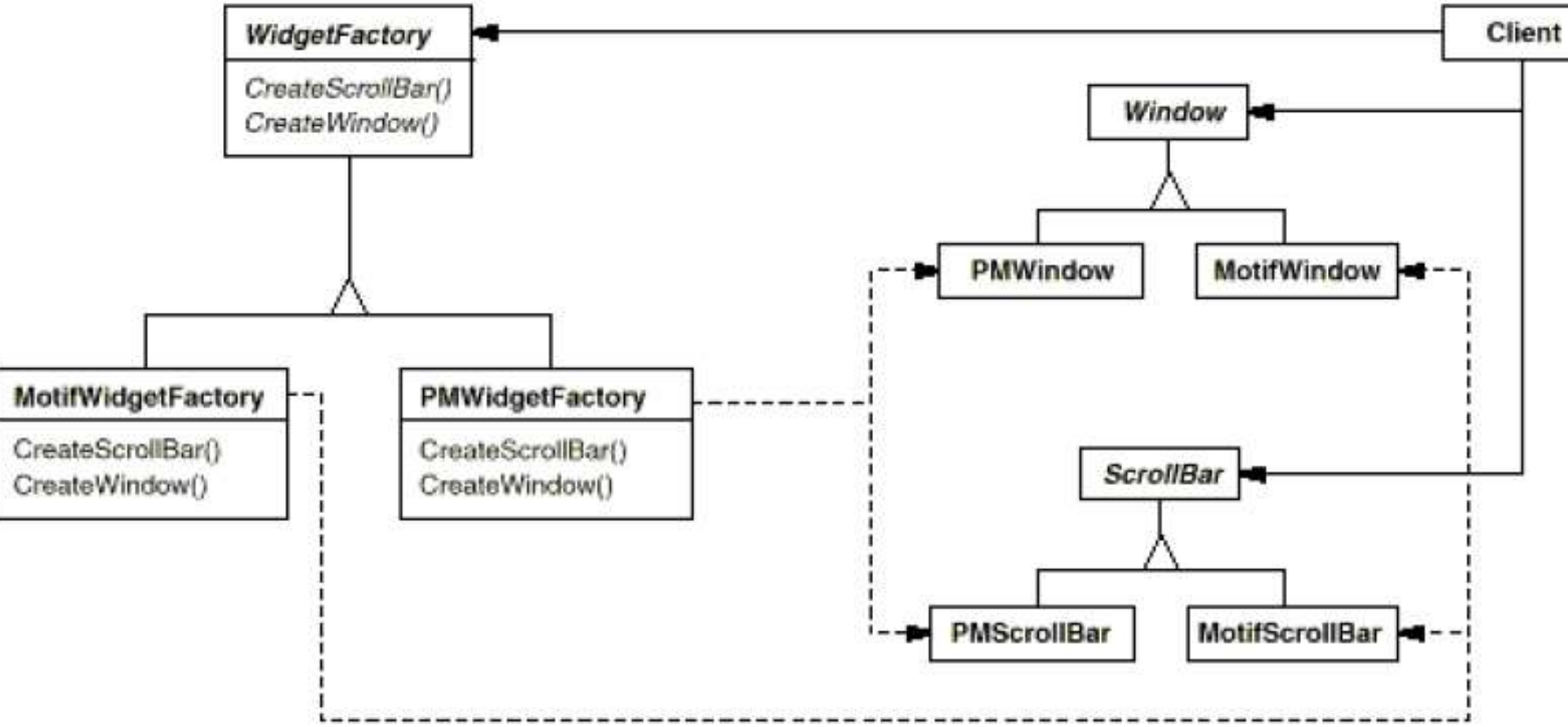


Creational Patterns – Abstract Factory

- **Intent** – Provide an interface for creating families of related or dependent objects without specifying their concrete classes
- **Also Known As** – Kit
- **Motivation**
  - Consider a user interface toolkit that supports multiple look-and-feel standards, defines different appearances and behaviors for user interface "widgets" like scroll bars, windows, and buttons
  - To be portable across look-and-feel standards, an application should not hard-code its widgets for a particular look and feel

Abstract Factory 2

- Each subclass implements the operations to create the appropriate widget for the look and feel

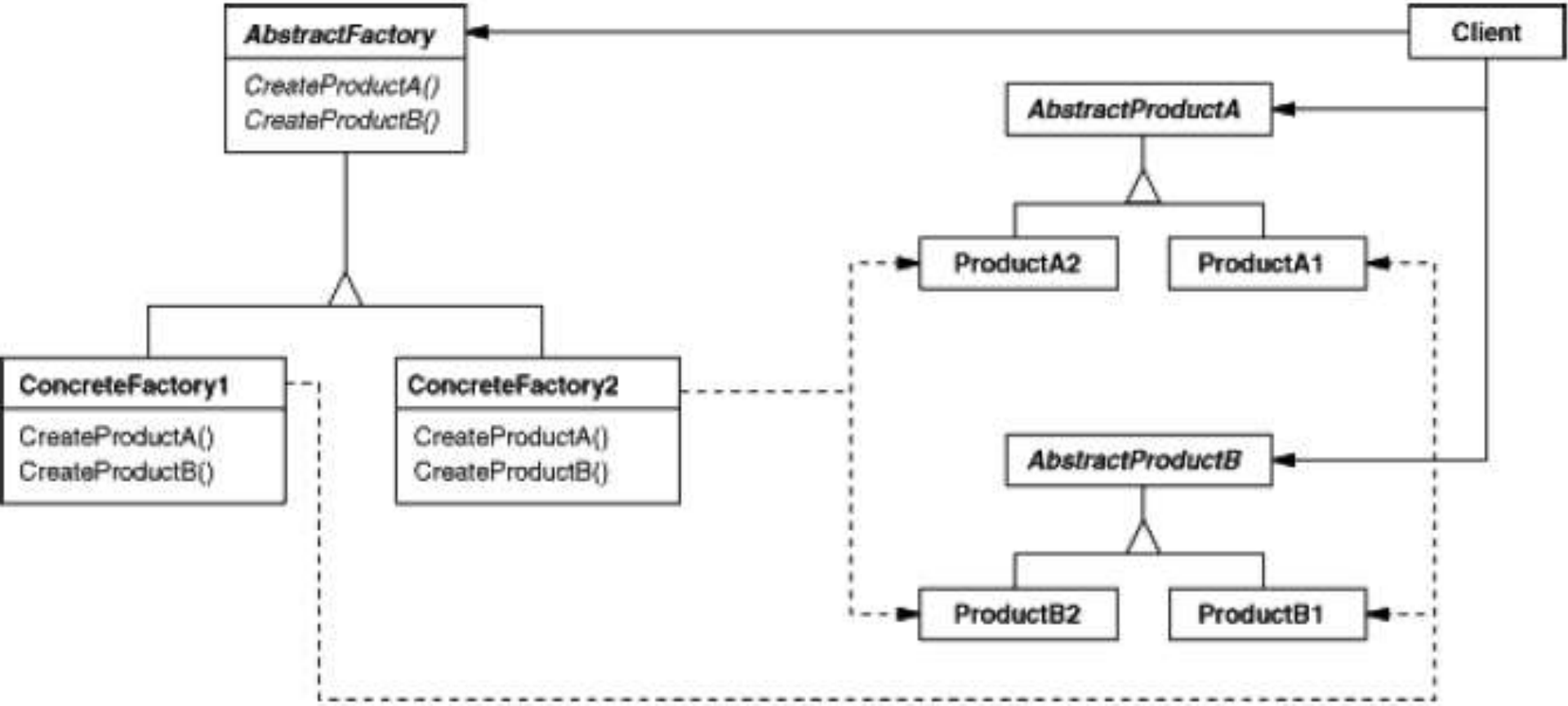


Abstract Factory 3

- **Applicability** – Use this pattern when:
  - a system should be independent of how its products are created, composed, and represented
  - a system should be configured with one of multiple families of products
  - a family of related product objects is designed to be used together, and you need to enforce this constraint
  - you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations

Abstract Factory 4

- Structure



Abstract Factory – Example

- Suppose we need to get the specification of various parts of a computer based on which work the computer will be used for
- The different parts of computer are, say **Monitor, RAM** and **Processor.** The different types of computers are **PC, Workstation** and **Server**
- So, here we have an abstract base class **Computer**

Abstract Factory – Java 1

```
public abstract class Computer {
    public abstract Parts getRAM();
    public abstract Parts getProcessor();
    public abstract Parts getMonitor();
 }

public class Parts {
    public String specification;

    public Parts(String specification) {
            this.specification = specification;}

    public String getSpecification() {
            return specification;}
}
```

Abstract Factory – Java 2

```
public class PC extends Computer {
        public Parts getRAM() {return new Parts("512 MB");}
        public Parts getProcessor() {return new Parts("Celeron");}
        public Parts getMonitor() {return new Parts("15 inches");}
}

public class Workstation extends Computer {
        public Parts getRAM() {return new Parts("1 GB");}
        public Parts getProcessor() {return new Parts("Intel P 3");}
        public Parts getMonitor() {return new Parts("19 inches");}
}

public class Server extends Computer{
        public Parts getRAM() {return new Parts("4 GB");}
        public Parts getProcessor() {return new Parts("Intel P 4");}
        public Parts getMonitor() {return new Parts("17 inches");}
}
```

Abstract Factory – Java 3

```
public class ComputerType {
    private Computer comp;
    public static void main(String[] args) {
            ComputerType type = new ComputerType();
            Computer c = type.getComputer("Server");
            System.out.print("Monitor: "+ c.getMonitor().getSpecification());
            System.out.print("RAM: "+ c.getRAM().getSpecification());
            System.out.print("Processor: "+ c.getProcessor().getSpecification());
    }

    public Computer getComputer(String computerType) {
            if (computerType.equals("PC"))
                    comp = new PC();
            else if(computerType.equals("Workstation"))
                    comp = new Workstation();
            else if(computerType.equals("Server"))
                    comp = new Server();
            return comp;
    }}
```

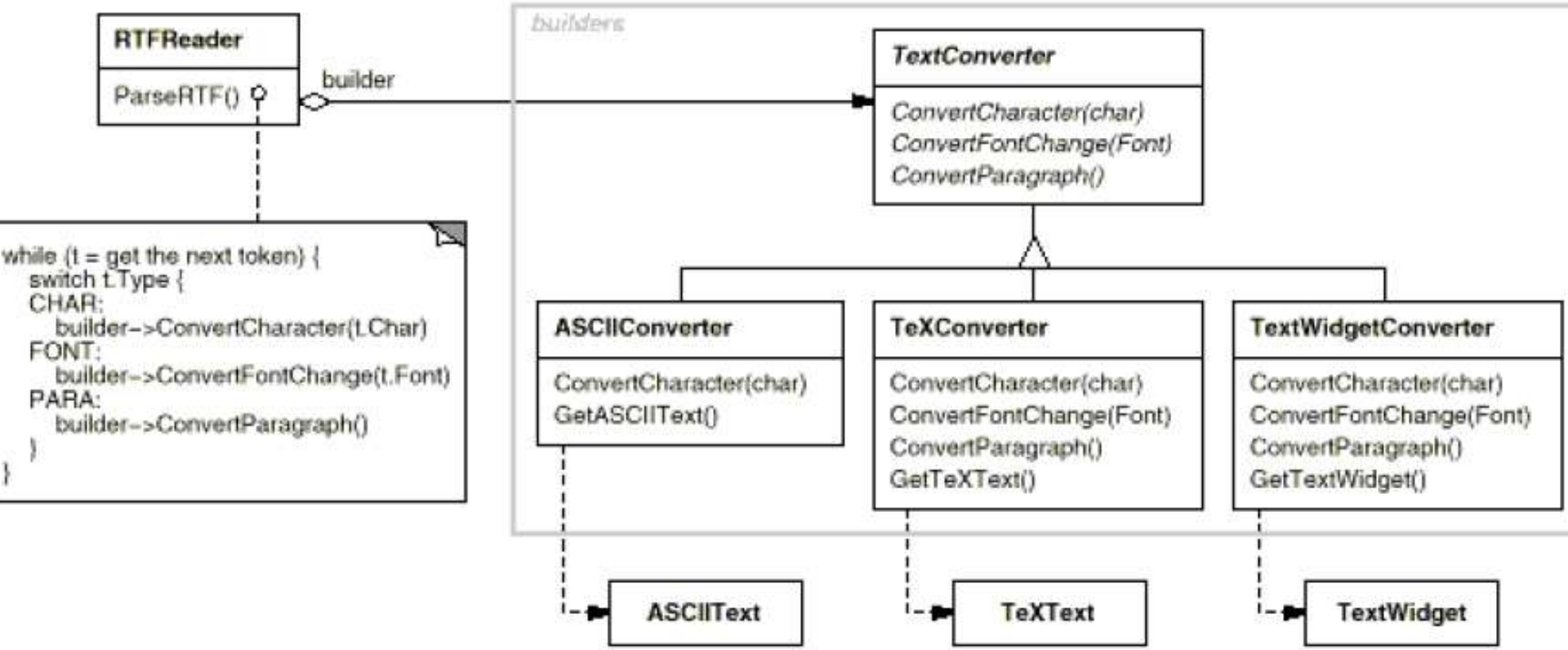
- Result: Monitor: 17 inch RAM: 4 GB, Processor: Intel P 4.

Creational Patterns – Builder

- **Intent – Separate the construction of a complex object from its representation** so that the same construction process can create different representations
- **Motivation**
  - A reader for the RTF (Rich Text Format) document exchange format should be able to convert RTF to many text formats.
  - The reader might convert RTF documents into plain ASCII text or into a text widget that can be edited interactively. The problem, however, is that the number of possible conversions is open-ended. So it should be easy to add a new conversion without modifying the reader.

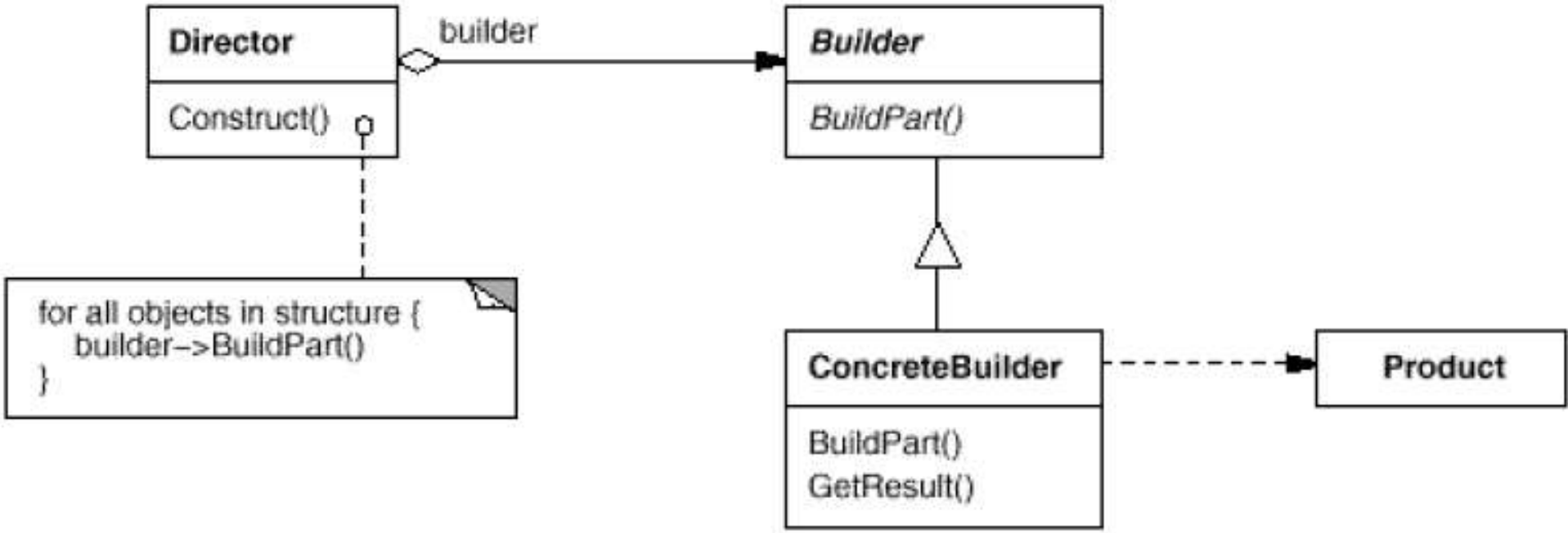
Builder 2

- Each kind of converter class takes the mechanism for creating and assembling
- The converter is separate from the reader



Builder 3

- **Structure**



Builder – Example

- What is it comprised of a children meal at a fast food restaurant? Well, **a burger, a cold drink, a medium fries and a toy**
- **Here, what is important**? Every time a children's meal is ordered, the service boy will take a burger, a fries, a cold drink and a toy
- Now suppose, there are **3 types of burgers**: *Vegetable, Fish and Chicken*, **2 types of cold drinks**: Cola and Orange and **2 types of toys**: a car and a doll
- So, the order might be a combination of one of these, but the process will be the same

Builder – Java 1

```
public interface Item {
  public Packing pack();
  public int price();
}
public abstract class Burger implements Item {
  public Packing pack() {
       return new Wrapper();
  }
  public abstract int price();
}
```

Builder – Java 2

```
public class VegBurger extends Burger {
  public int price() {
      return 39;}
}
public class Fries implements Item {
  public Packing pack() {
      return new Envelop();
  }
  public int price() {
      return 25;
  }
}
```

Builder – Java 3

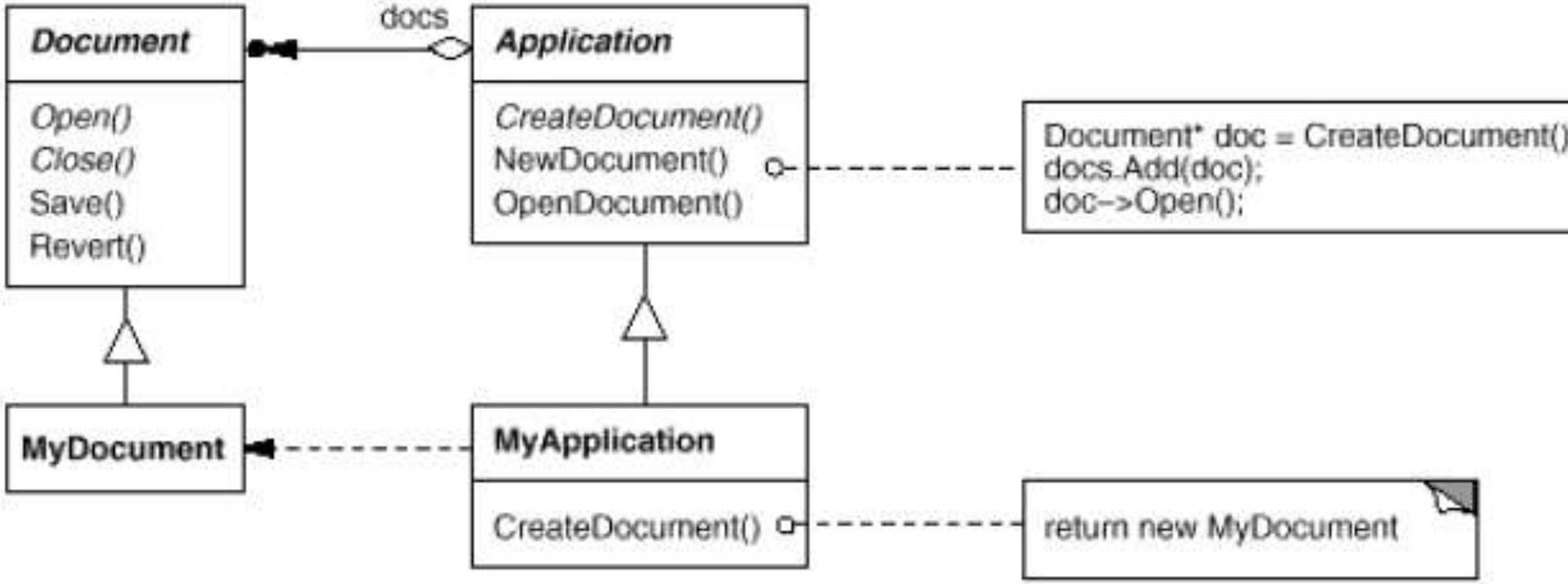
```
public class MealBuilder {

  public Packing additems() {
   Item[] items = {new VegBurger(), new Fries(), new
   Cola(), new Doll()}
   return new MealBox().addItems(items);
   }

public int calculatePrice() {
   int totalPrice = new VegBurger().price() + new
   Cola().price() + new Fries().price() + new Doll().price();
   return totalPrice;
   }
}
```

Creational Patterns – Factory Method

- **Intent** – Define an interface for creating an object, but let subclasses decide which class to instantiate
- **Also Known As** – Virtual Constructor
- **Motivation** – To create a drawing application, for example, we define the classes DrawingApplication and DrawingDocument. The Application class is responsible for managing Documents and will create them as required (at Open or New from a menu, for example)

Factory Method 2

- Subclasses redefine an abstract CreateDocument
- It can then instantiate application-specific Documents without knowing their class
- CreateDocument is a **factory method because it's** responsible for "manufacturing" an object

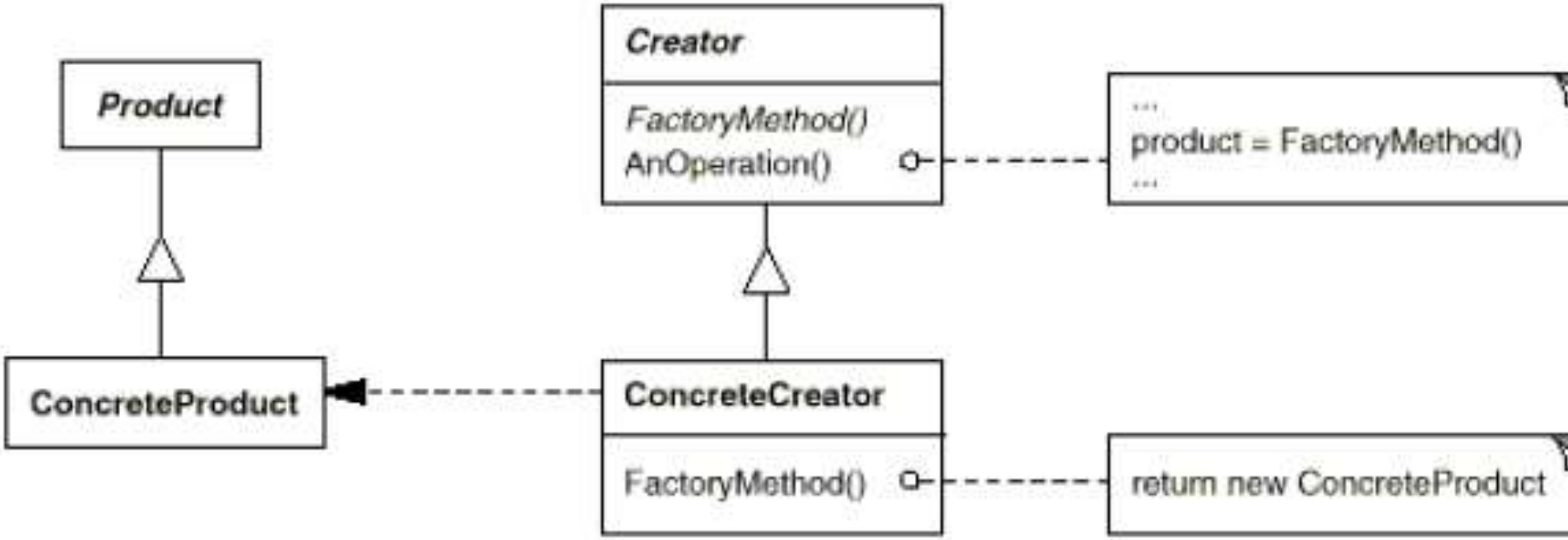


Factory Method 3

- **Applicability** – Use this pattern when
  - a class can't anticipate the class of objects it must create
  - a class wants its subclasses to specify the objects it creates
  - classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate

Factory Method 4

- Structure



Factory Method 5

- **Consequences**

1. *Provides interconnections for subclasses*

2. *Connects parallel class hierarchies*

Factory Method – Example

- Let's suppose an application asks for entering the name and sex of a person
- If the sex is Male (M), it displays welcome message saying **Hello Mr. <Name>** and
- If the sex is Female (F), it displays message saying **Hello Ms. <Name>**

Factory Method – Java 1

```
public class Person {
   public String name;
   private String gender;

   public String getName() {
        return name;
   }

   public String getGender() {
        return gender;
   }
}
```

Factory Method – Java 2

```
public class Male extends Person {
  public Male(String fullName) {
      System.out.println("Hello Mr. "+fullName);
  } }

public class Female extends Person {
  public Female(String fullName) {
      System.out.println("Hello Ms. "+fullName);
  } }
```

Factory Method – Java 3

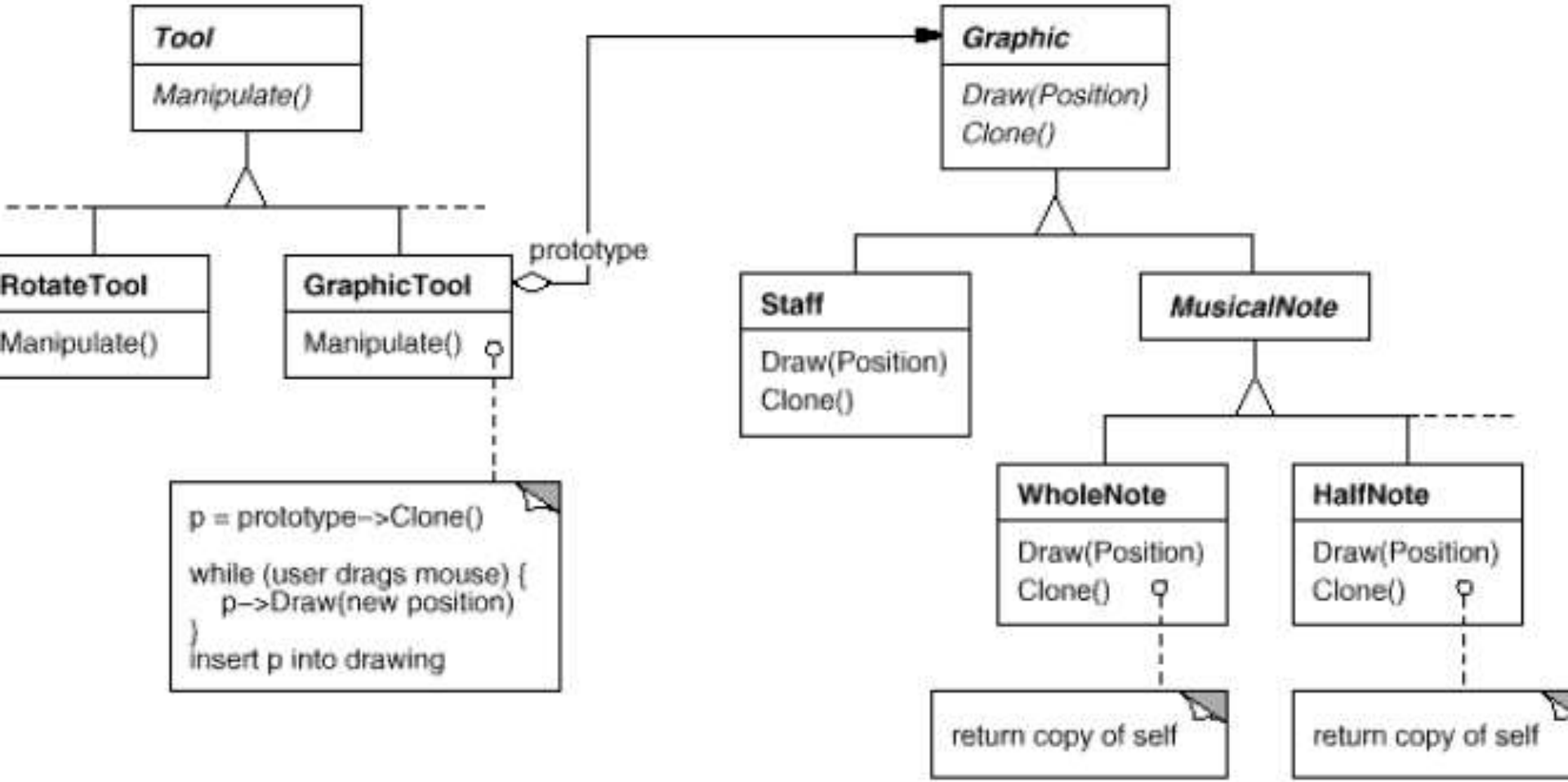
```
public class SalutationFactory {
   public static void main(String args[]) {
        SalutationFactory factory = new SalutationFactory();
        factory.getPerson(args[0], args[1]);
   }

   public Person getPerson(String name, String gender) {
        if (gender.equals("M"))
                return new Male(name);
        else if(gender.equals("F"))
                return new Female(name);
        else
                return null;
   }
}
```

Creational Patterns – Prototype

- **Intent** – Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype
- **Motivation** – You could build an editor for music scores by customizing a general framework for graphical editors and adding new objects that represent notes, rests, and staves
- The editor framework may have a palette of tools for adding these music objects to the score. The palette would also include tools for selecting, moving, and otherwise manipulating music objects

Prototype 2

- We can use the Prototype pattern to reduce the number of classes

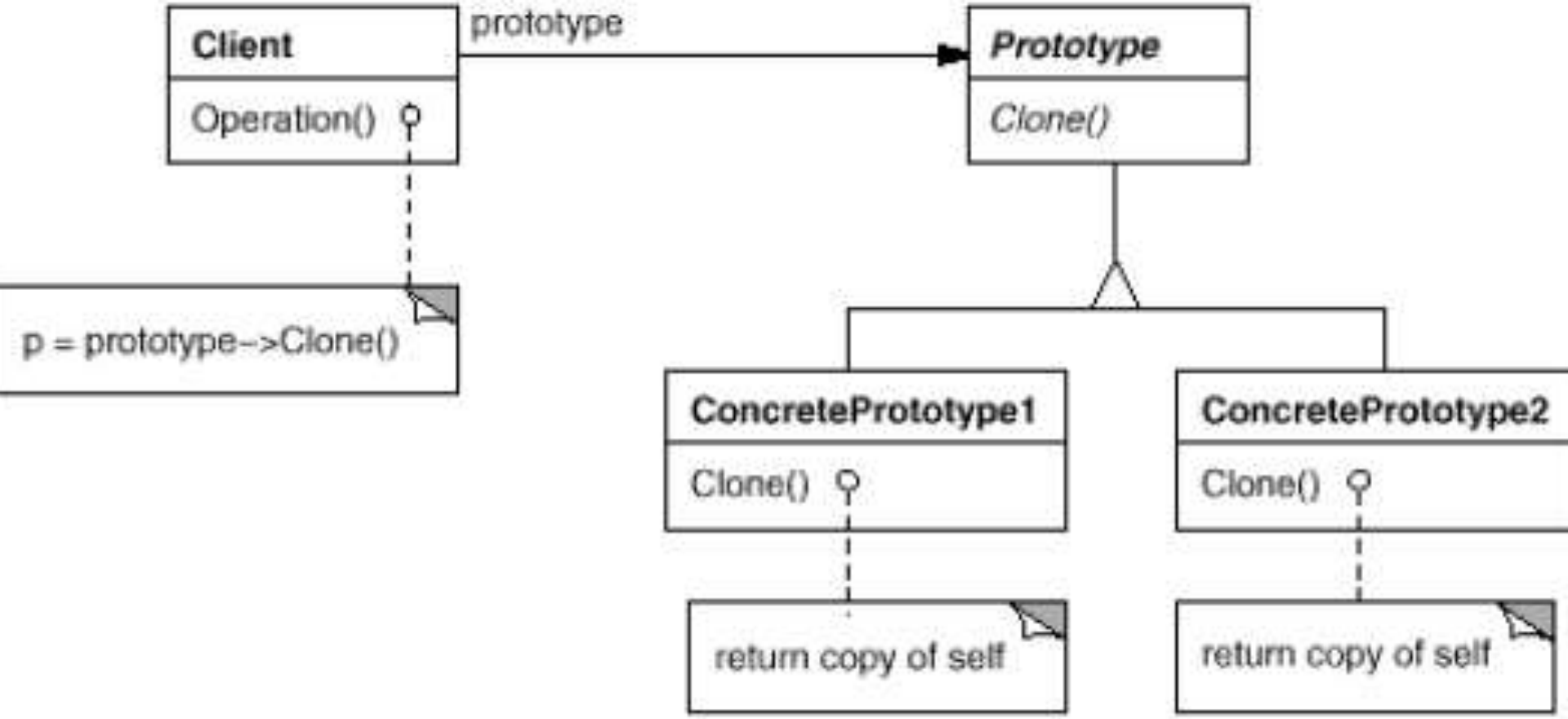


Prototype 3

- **Applicability** – Use the Prototype pattern when a system should be independent of how its products are created, composed, and represented; *and*
  - when the classes to instantiate are specified at run-time, for example, by dynamic loading; *or*
  - to avoid building a class hierarchy of factories that parallels the class hierarchy of products; *or*
  - when instances of a class can have one of only a few different combinations of state

Prototype 4

- Structure



Prototype 5

- **Consequences**

1. *Adding and removing products at run-time*
2. *Specifying new objects by varying values*
3. *Specifying new objects by varying structure.*
4. *Reduced subclassing*
5. *Configuring an application with classes dynamically*

Prototype – Example

- The prototype means making a clone. We use the interface **Cloneable** and call its method **clone()** to clone the object
- In plant cell, we break the cell in two and make two copies and the original one does not exist. But, this example will serve the purpose
- Let's say the Mitotic Cell Division in plant cells. Let's take a class **PlantCell** having a method **split().** The plant cell will implement the interface **Cloneable**.

Prototype – Java 1

```
public class PlantCell implements Cloneable {
    public Object split() {
        try {
                return super.clone();
        }catch(Exception e) {
                System.out.println("Exception occured:
                        "+e.getMessage());
                return null;}
    }
}
```

Prototype – Java 2

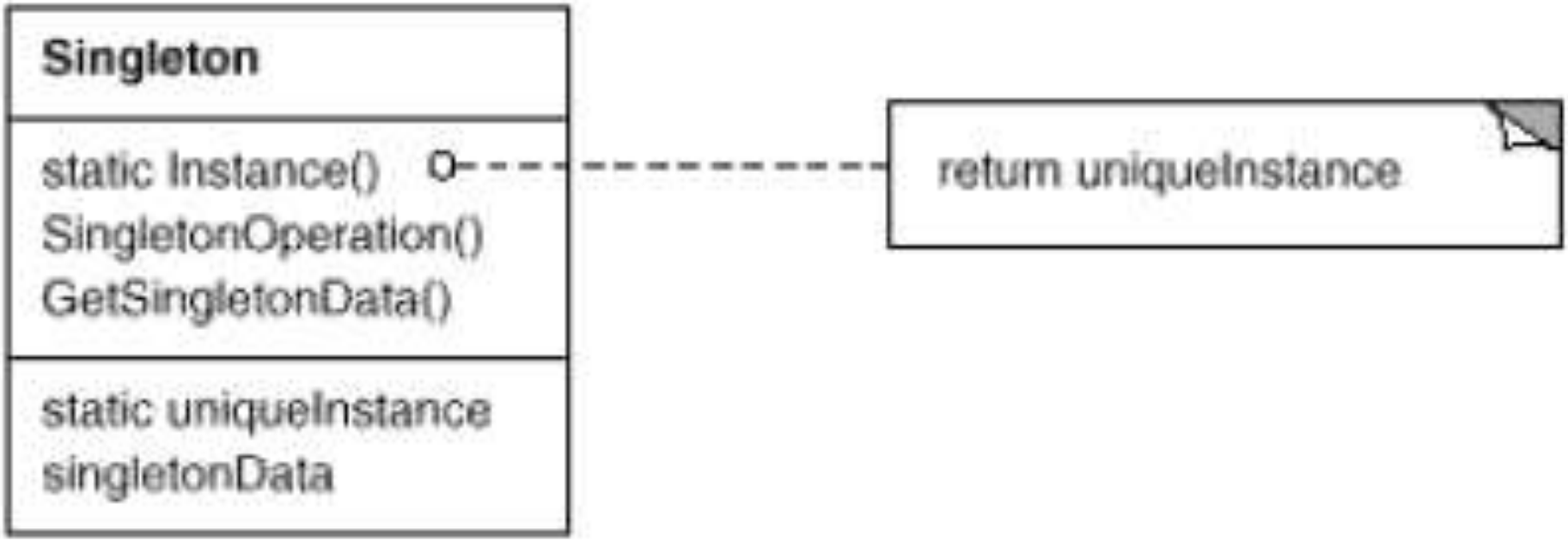
```
public class CellDivision {
    public static void main(String[] args) {
        PlantCell cell = new PlantCell();
        PlantCell newPlantCell = (PlantCell)cell.split();
    }
}
```

Creational Patterns – Singleton

- **Intent** – Ensure a class only has one instance, and provide a global point of access to it
- **Motivation** – It's important for some classes to have exactly one instance. There should be only one file system and one window manager. An accounting system will be dedicated to serving one company.
- How do we ensure that a class has only one instance and that the instance is easily accessible? A global variable makes an object accessible, but it doesn't keep you from instantiating multiple objects.

Singleton 2

- **Applicability** – Use the Singleton pattern when there must be exactly one instance of a class**Structure**



Singleton – Example

- A very simple example is say **Logger**, suppose we need to implement the logger and log it to some file according to date time. In this case, we cannot have more than one instances of Logger in the application otherwise the file in which we need to log will be created with every instance.

- We use Singleton pattern for this and instantiate the logger when the first request hits or when the server is started.

Singleton – Java 1

```
public class Logger {
   private String fileName;
   private Properties properties;
   private Priority priority;
   private Logger() {
         logger = this;}


   public int getRegisteredLevel() {
         int i = 0;

         InputStream inputstream =
   getClass().getResourceAsStream("Logger.properties");
         properties.load(inputstream);
         inputstream.close();
         i = Integer.parseInt(properties.getProperty("**logger.registeredlevel**"));
         if(i < 0 || i > 3)
                  i = 0;

         return i;
   }
```

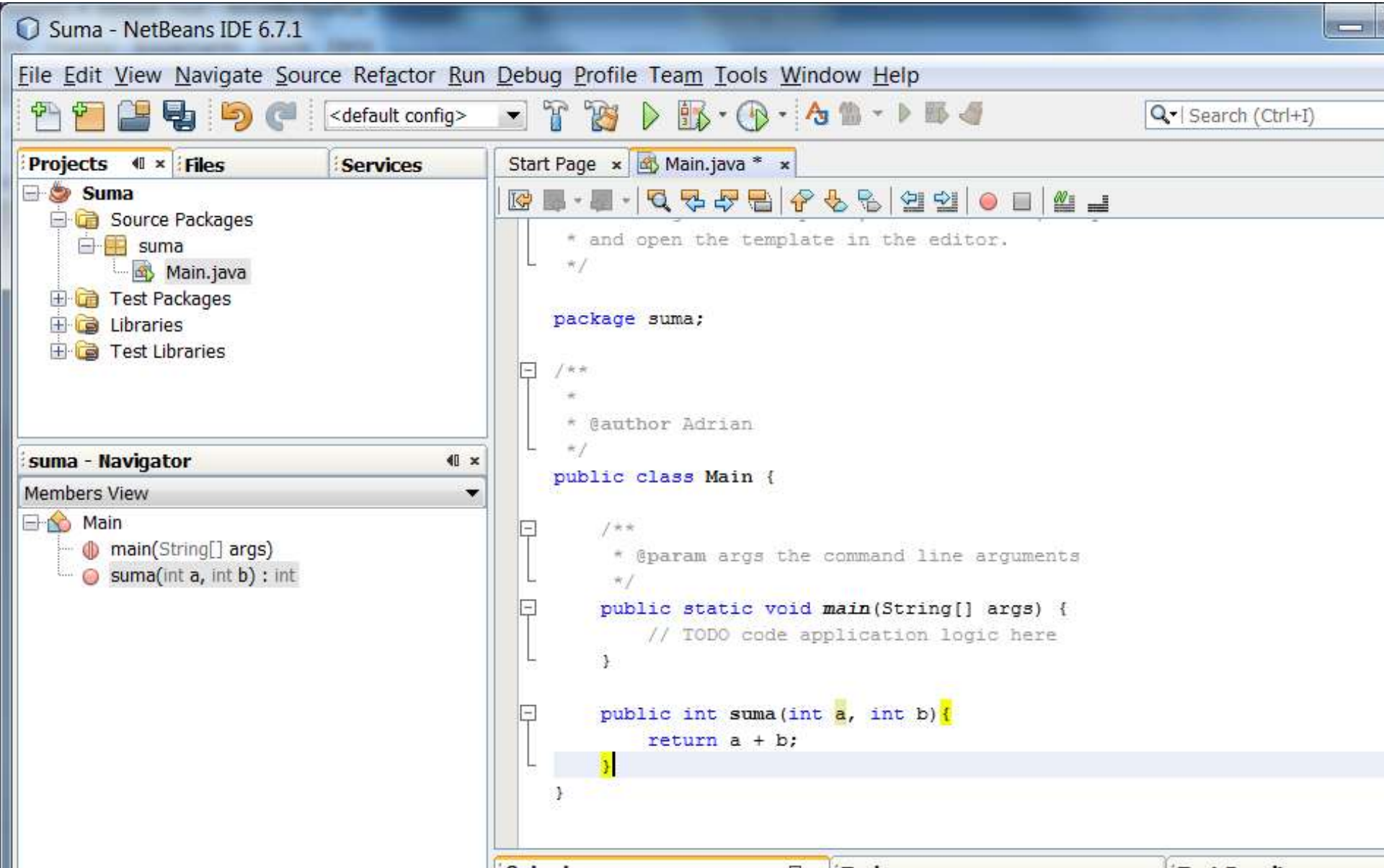
Singleton – Java 2

```
private String getFileName(GregorianCalendar gc) {
        SimpleDateFormat dateFormat1 = new SimpleDateFormat("dd-MMM-yyyy");
        String dateString = dateFormat1.format(gc.getTime());String fileName =
   "C:\\log\\PatternsExceptionLog-" + dateString + ".txt";
        return fileName;
   }

   public void logMsg(Priority p, String message) {
        GregorianCalendar gc = new GregorianCalendar();
        String fileName = getFileName(gc);
        FileOutputStream fos = new FileOutputStream(fileName, true);
        PrintStream ps = new PrintStream(fos);
        SimpleDateFormat dateFormat2 = new SimpleDateFormat("EEE, MMM d, yyyy 'at'
   hh:mm:ss a");ps.println("<"+dateFormat2.format(gc.getTime())+">["+message+"]");
        ps.close();
   }

   public static void initialize() {
        logger = new Logger();
   }

   private static Logger logger;
   public static Logger getLogger() {
        return logger;
   }
}
```

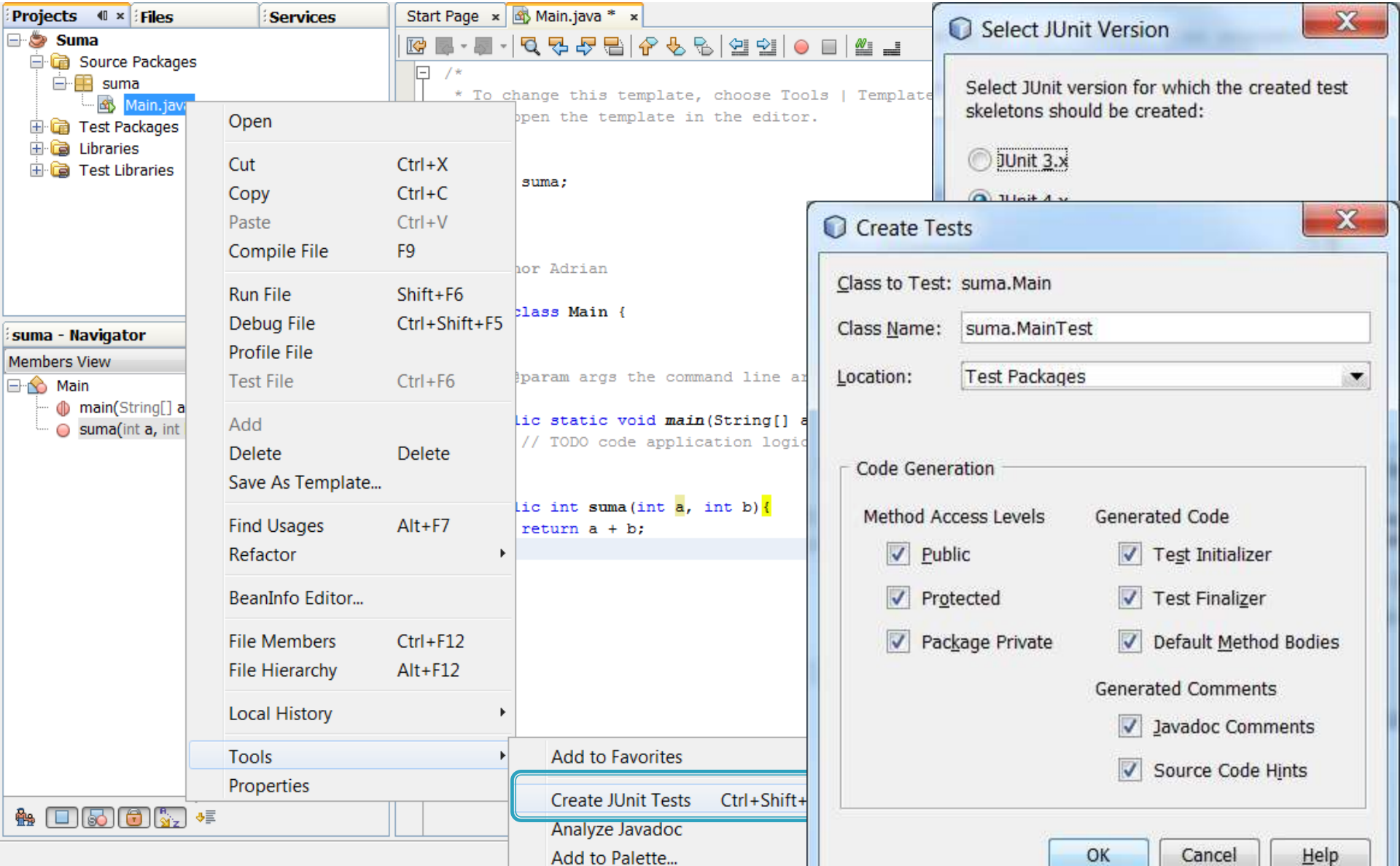
Unit Testing

- 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

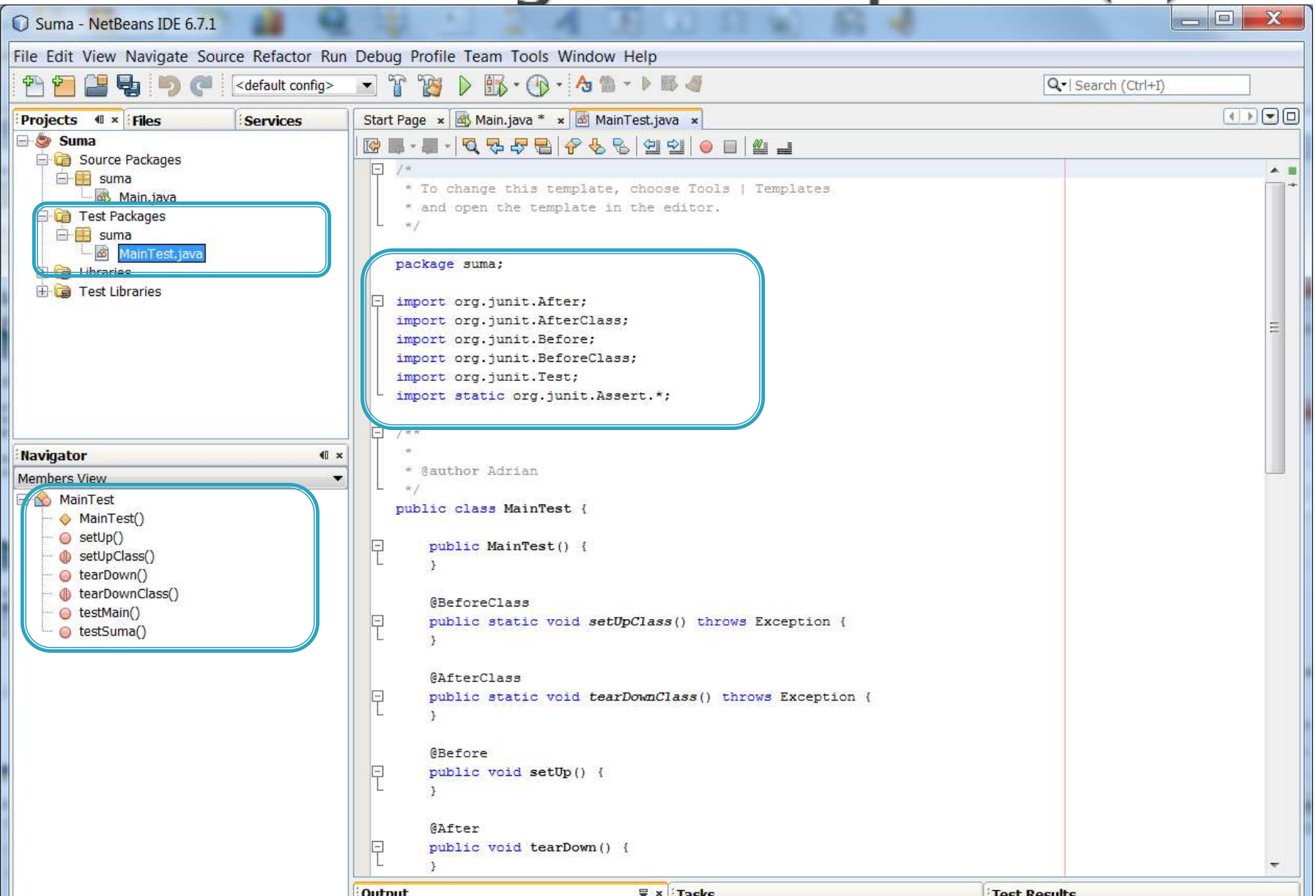
Unit Testing – Exemplu 1 (1)



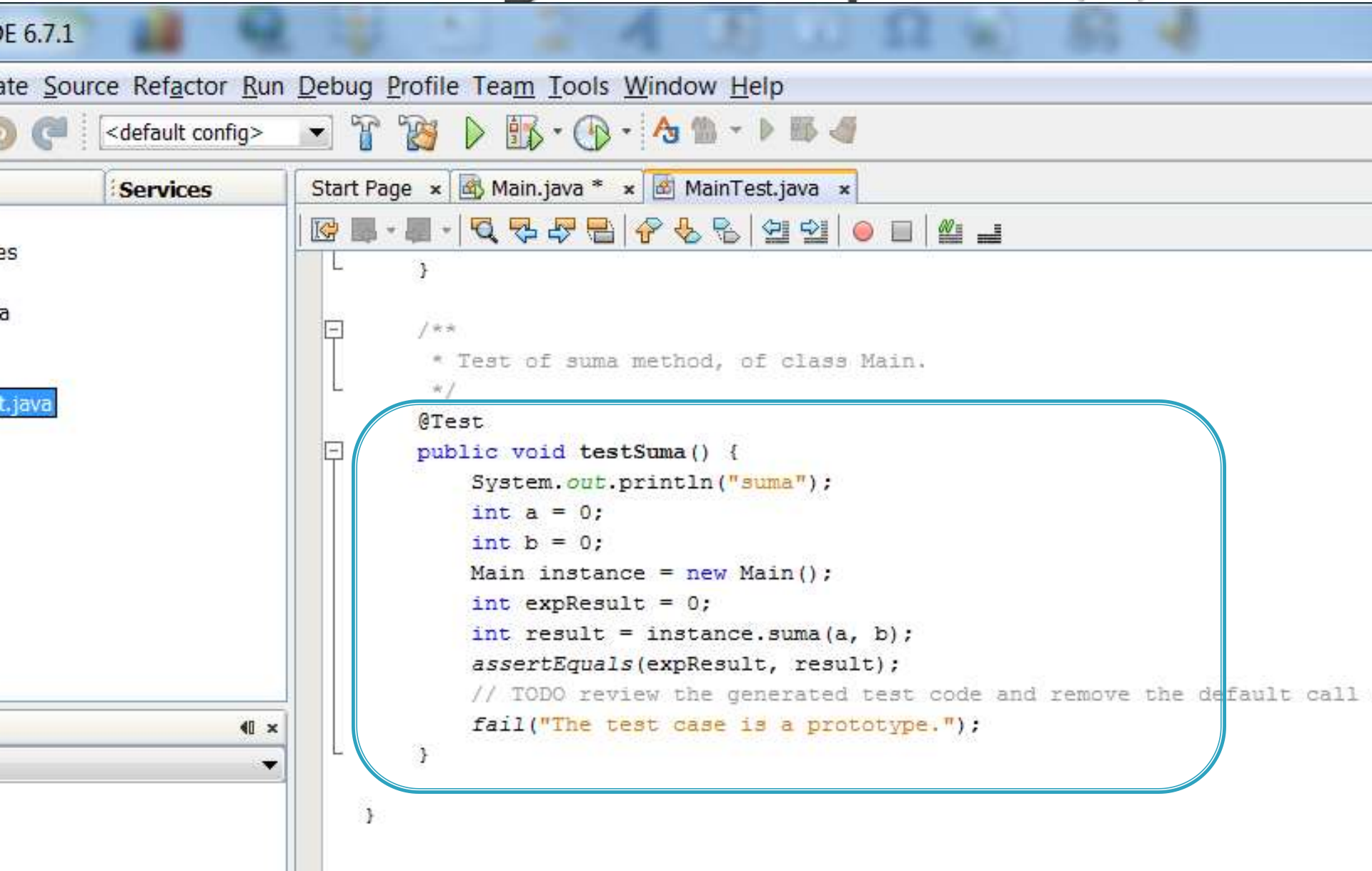
Unit Testing – Exemplu 1 (2)



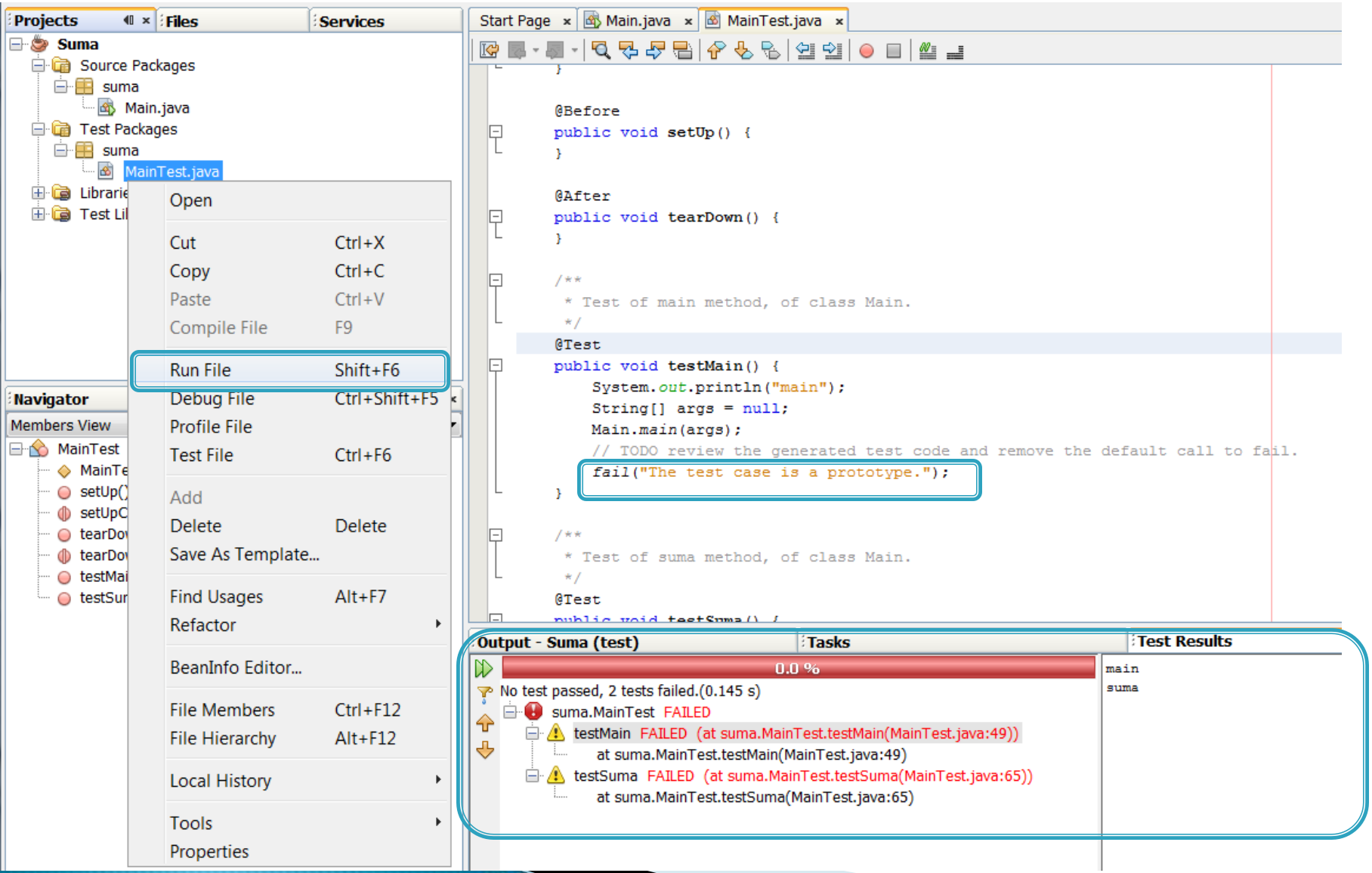
Unit Testing – Exemplu 1 (3)



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.");
    }
}
```

Output - Suma (test) | Tasks | Test Results

50.0 %

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

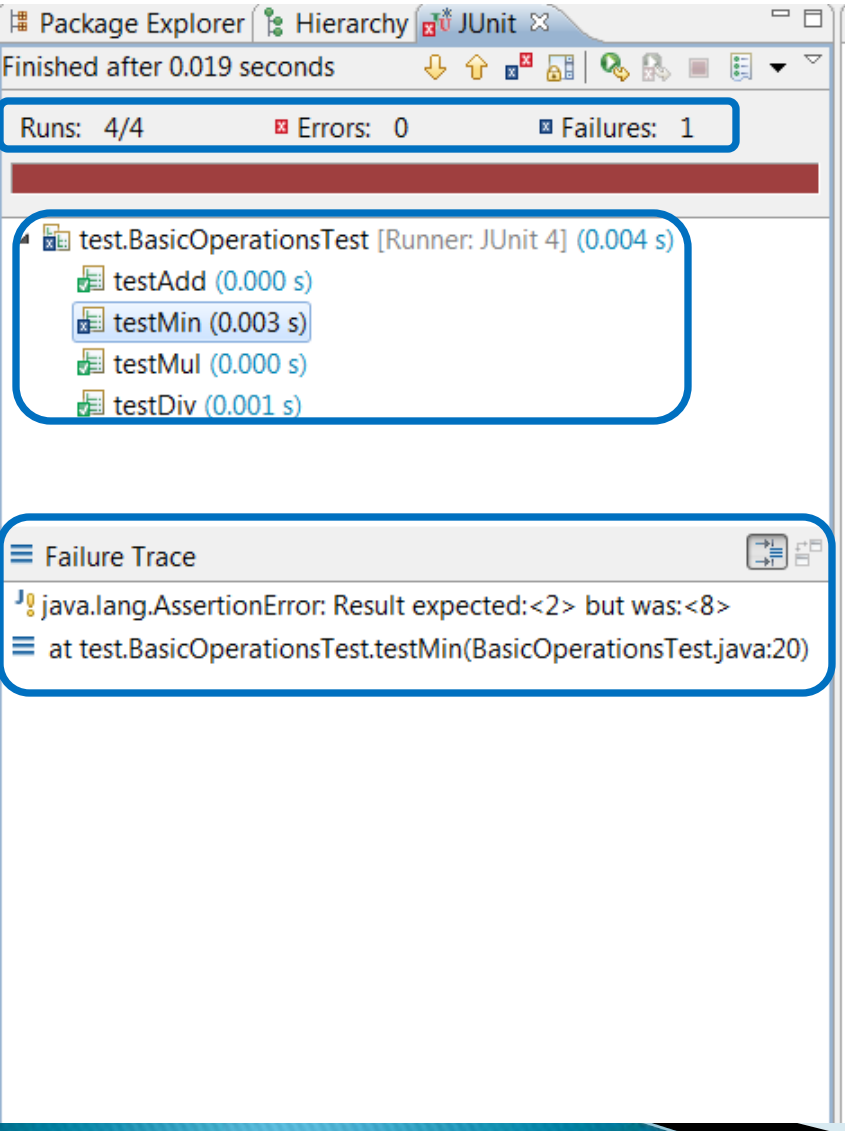
suma

Unit Testing – Example 2 (1)

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));
    }

}
```

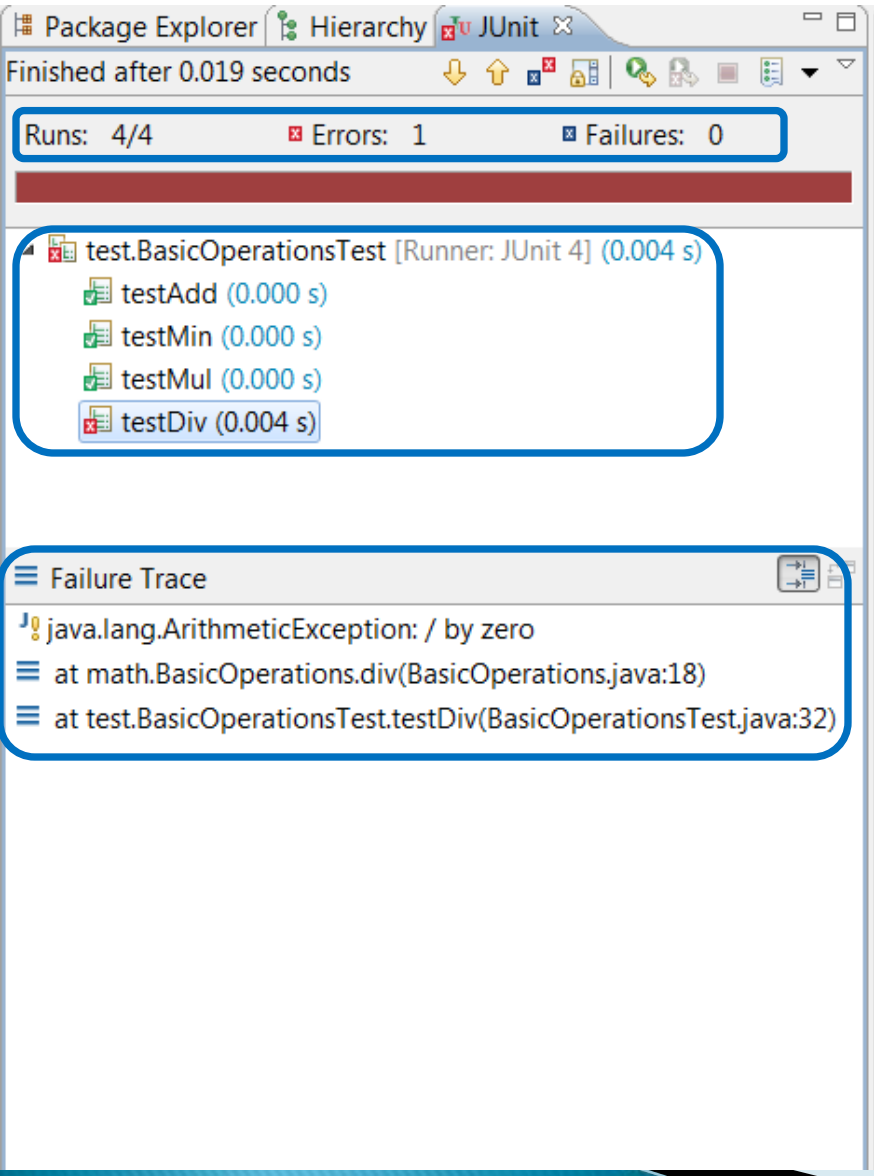
Unit Testing – Example 2 (2)



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));
    }
}
```

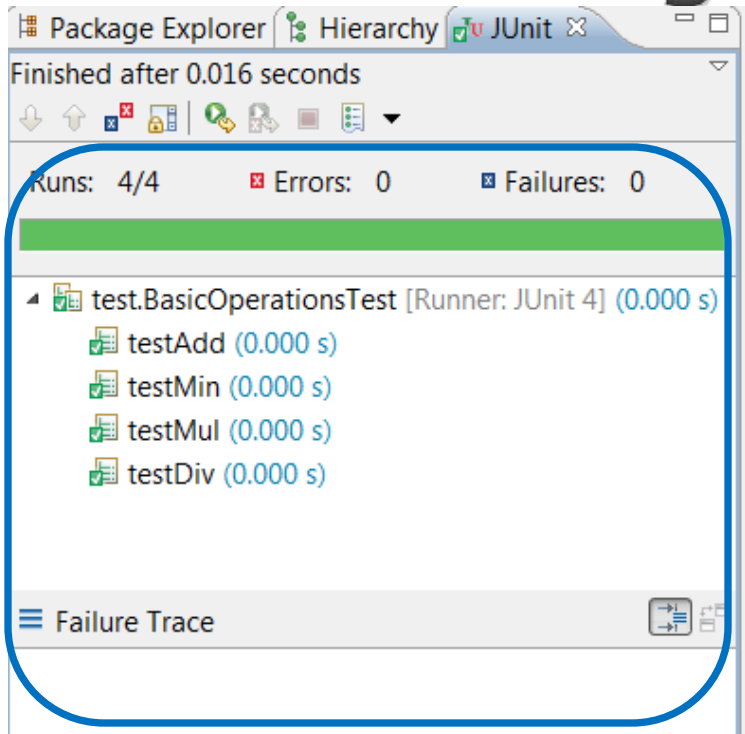
Unit Testing – Example 2 (3)



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!");
    }
}
```

Concluzii

- Design Patterns
  - Definitions, Elements, Example, Classification
- Creational Patterns
  - Abstract Factory, Builder, Factory Method, Prototype, Singleton
- JUnit Testing

Design Patterns

- 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