UML

- Unified Modeling Language - UML (Unified Modeling Language)
- UML is a standardized language and graphical notation for
  - visualization,
  - specification,
  - modeling and
  - documentation

  parts of the software system being designed
- UML is a common "dictionary" for communication between people involved in the design and development of a software system

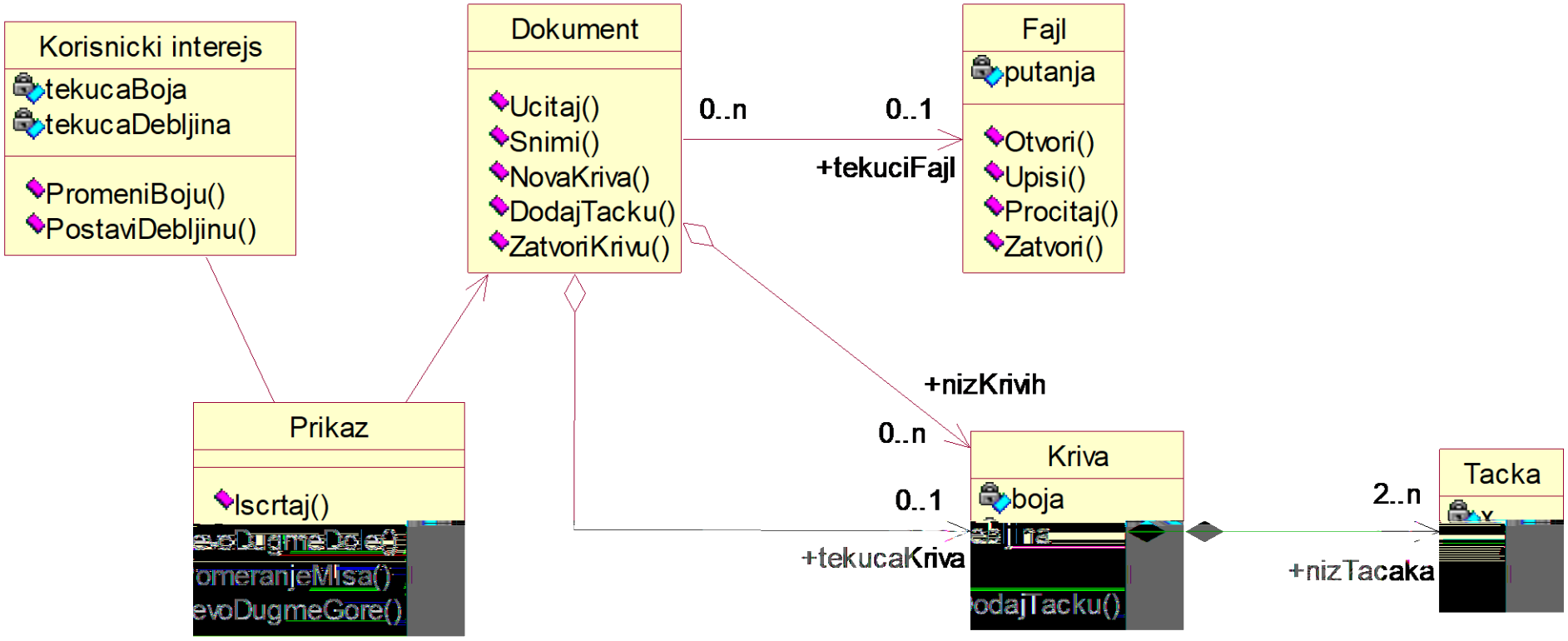
Types of UML diagrams

- Class diagrams (engl. *Class Diagram)*
- Use case diagrams *Use-Case Diagram)*
- Sequence diagrams (engl. *Sequence Diagram)*
- Cooperation diagrams (engl. *Collaboration Diagram)*
- State diagrams (engl. *Statechart Diagram)*
- Activity diagrams *Activity Diagram)*
- Component diagrams *Component Diagram)*
- Layout diagrams (engl. *Deployment Diagram)*

Class diagrams

- They are used to represent classes and their organizations in packages
- Class diagrams are used for modeling
  - system domain
  - applications
- The elements of the diagram are:
  - class
  - connections between classes
    - inheritance
    - association
    - aggregation
    - composition
  - packages
  - dependency relationships between packages

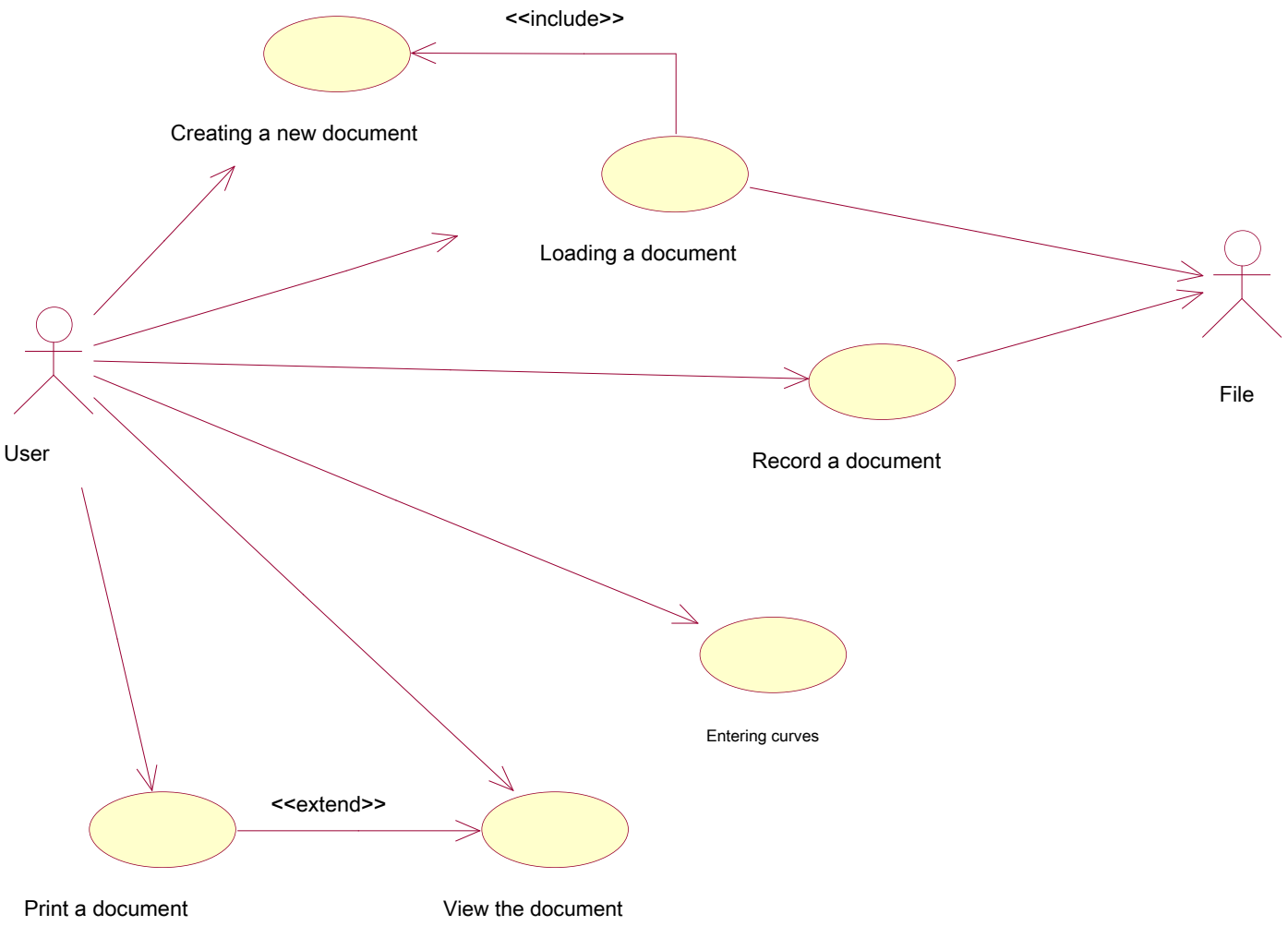
Class diagram - example



Usage case diagrams

- They are used in the process of collecting and documenting user requests
- The elements of the diagram are:
  - actors
    - system users
    - other systems from the environment
  - cases of using the system
  - links between actors and use cases
    - association
    - generalization
  - packages
  - dependency relationships between packages

Usage case diagram - example

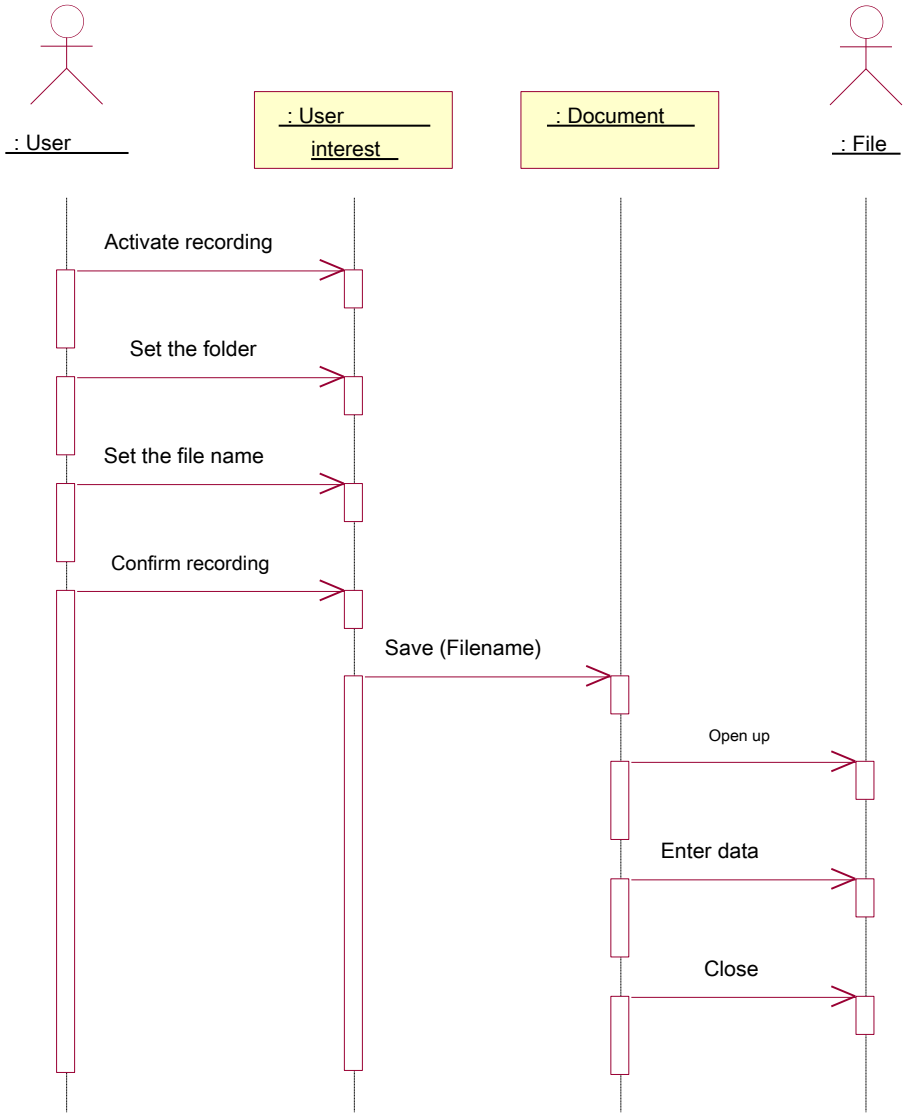


Sequence diagrams

- They are used to represent scenarios of interaction between objects in the system
  - Most often, these scenarios refer to cases of using the system

- The elements of the diagram are:
  - objects
  - timeline
  - messages between objects

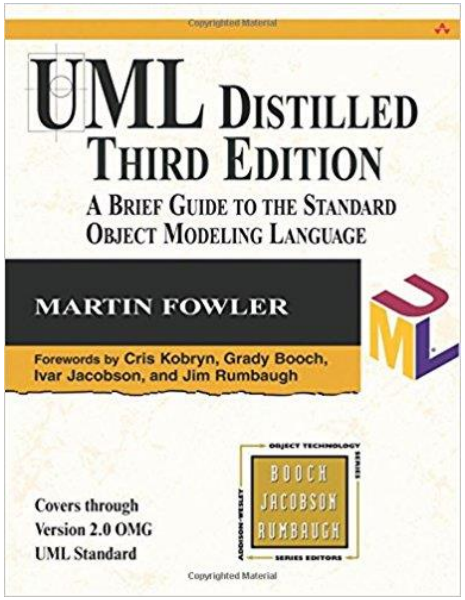
Sequence diagram - example



Literature for UML

- Martin Fowler, UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3rd edition, Addison Wesley, 2003.

- Grady Booch, James Rumbaugh, Ivar Jacobson, UML User Guide, CET, 2001.



Exercise tasks

- Draw UML class diagrams and usage case diagrams for classes previously implemented within the package employees, mathematics i institution. Identify typical usage cases for applications that use previous packages.

OBJECT-ORIENTED SOFTWARE DESIGN

OO program design

“ *There are two ways to design and develop a program. One is to make them so simple that they obviously have no flaws. The second is to do so many of them*
complicated not to have ***obvious*** *shortcomings.* ”

CAR Hoare

- The key to **successful OO program design** is **a well-designed abstraction of the problem** through object concepts:

  “ *The essence of abstraction is to store information that is relevant in a given context and to ignore information that is irrelevant to a given context.* ”

  John Guttag

Input data for OO design

- **Conceptual model** is the result of object-oriented analysis and describes concepts in the problem domain. It is explicitly created to be independent of implementation details, such as competitiveness or data retention

- **Usage cases (** use cases) are descriptions of a sequence of events that together lead to the system realizing some useful activity

- **Sequence diagrams** graphically display, for a particular scenario within a use case, events generated by external actors, their order and possible events within the system

Input data for OO design

- **User interface documentation (** engl. user interface - UI) displays and describes the appearance and flow of work with the program via the user interface of the final software product

- **Relational data model** - a data model is an abstract model that describes how data is presented
  and use. If an object database is not used, then it is necessary to create a relational data model in advance, since the chosen strategy for object-relational mapping is one of the outputs of the object-oriented design process.

OO program design

- **Define objects (** create class diagrams based on conceptual diagrams). On this occasion, entities are typically mapped to classes
- **Define class elements (** attributes and methods)
- **Define the number of objects, the moment** their **emergence** i **disappearances,** as well as **way of interacting with each other** over time
- **Define responsibilities** every part of the system

OO program design

- **Use project forms (** if applicable) - the main advantage of applying project forms is the possibility of their reuse in several applications. Object-oriented design patterns typically show relationships and interactions between classes and objects, without specifying finite application classes and objects

- **Choose a work environment (** if applicable) - work environments include a large number of libraries and classes that can be used to implement standard structures in the application. This way you can save a lot of time on software development as it avoids rewriting much of the code when developing new applications

- **Identify persistent objects / data (** if applicable) - it is necessary to identify objects that should exist longer than the duration of one application execution. If the application uses a relational database, design an object-relational mapping

Project forms

- **Project forms** are **general, reusable solutions to frequently encountered problems** in a particular software design context

- They **are not a complete project that can be directly transformed into source or machine code,** they are already a description or template for solving a problem that can be used in many different situations

- Project forms are **formalized best practice procedures (** engl. best practices) that developers can use to effectively solve typical problems that arise when designing an application or system

Project forms

- Project forms were originally (according to GoF) divided into: **creative (** engl. creational), **structural (** engl. structural) i **behavioral (** engl. behavioral), and are also used today **competitive (** e.g. blockchain) i **architec**

  (eg Model-View-Controller - MVC)

- **Creative:** Singleton (ensures that the class has only one instance for which there is a global approach), Builder, Factory ...

- **Structural:** Adapter, Facade, Decorator ...

- **Behavioral:** Iterator, Interpreter, Visitor ...

Example 5.1 - form Singleton in Java

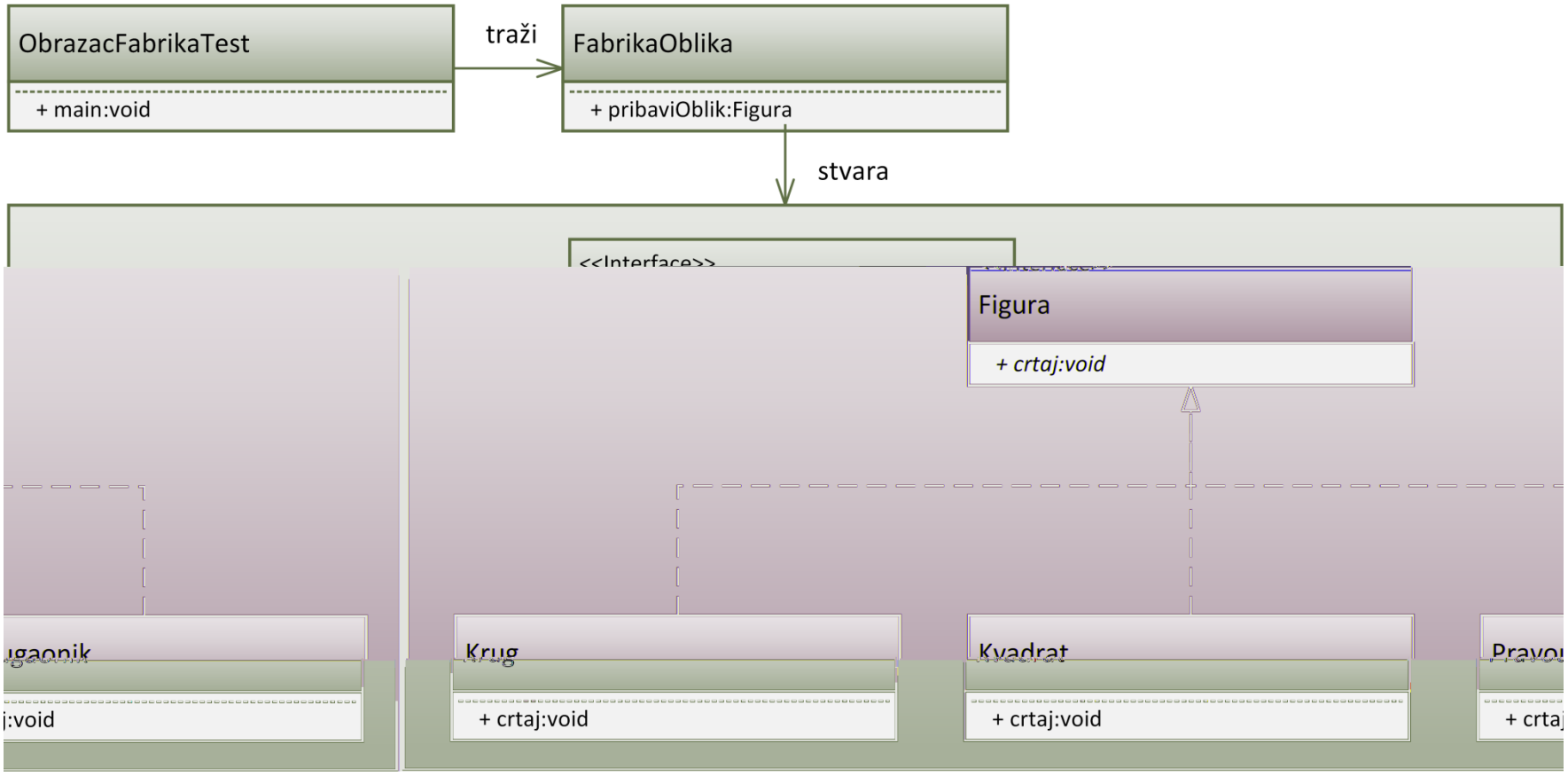
```
public class Singleton {

        private static Singleton instance = null ;

        protected Singleton () {
                // Exists only to prevent instantiation
        }

        public static Singleton obtainInstance () {
                if ( instance == null ) {
                        instance = new Singleton ();
                }
                return instance ;
        }
}
```

Typical applications: saving application settings (configuration files), data logging, etc.

Example 5.2 - Form Factory in Java

- With the Factory form, we create an object without having to set out the logic of creating the object. *creation logic)*
  client and then address the newly created object using a common interface
- We will create an interface Figure and the specific classes that implement this interface. Then in the next step we will define the class "factory" FactoryShape
- We will also do a test class FormFactoryTest which will use the class FactorySh to obtain a suitable object of some shape. The test class will only pass information to the shape factory whether it is a circle, square or rectangle, and the class FactoryShape will then "deliver" the required form to the test class

Example 5.2 - Form Factory in Java



Example 5.2 - interface Shape, class Circle, Rectangle i Square

```
public interface Figure {                                          //Figura.java
        void draw ();
}

public class Rectangle implements Figure {                         //Pravougaonik.java
        @Override public void draw () {
                System. out . println ( "Inside Rectangle :: draw () methods!" );
        }
}

public class Square implements Figure {                            //Kvadrat.java
        @Override public void draw () {
                System. out . println ( "Inside the Square :: draw () methods!" );
        }
}

public class Circle implements Figure {                            //Krug.java
        @Override public void draw () {
                System. out . println ( "Inside the Circle :: draw () method!" );
        }
}
```

Example 5.2 - class FactoryShape

```
public class FabrikaOblika {
        // get method The shape supplies the shape of the desired type
        public Figure getShape (String typeForm ) {
                if ( typeForm == null ) {
                        return null ;
                }
                if ( typeForm .equalsIgnoreCase ( "CIRCLE" )) {
                        return new Circle ();
                } else if ( typeForm .equalsIgnoreCase ( "RECTANGULAR" )) {
                        return new Rectangle ();
                } else if ( typeForm .equalsIgnoreCase ( "SQUARE" )) {
                        return new Square ();
                }
                return null ;
        }
}
```

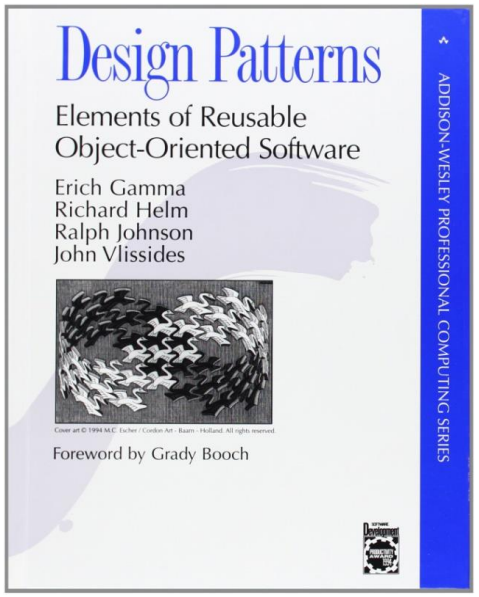
Example 5.2 - class FormFactoryTest

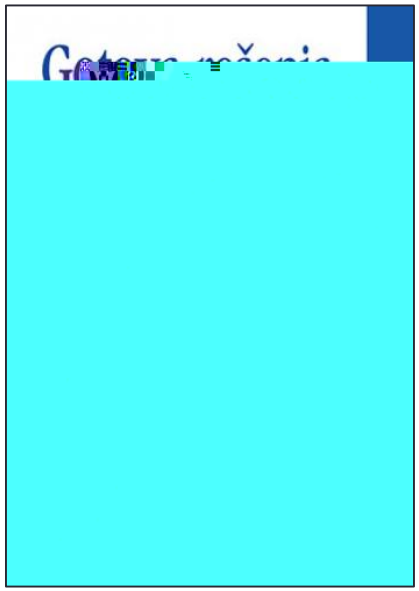
```
public class FormFactoryTest {

        public static void main (String [] args ) {FactoryShape factory Shape = new FactoryShape ();


                // get the shape Circle and call its drawing method
                Figure shape1 = factory Shape .adgetShape ( "CIRCLE" );
                // call the circle drawing method
                shape1 .draw ();

                // get the Rectangle shape and call its drawing method
                Figure shape2 = factory Shape .adgetShape ( "RECTANGULAR" );
                // call the draw method for rectangle
                shape2 .draw ();

                // get the shape Square and call its drawing method
                Figure shape3 = factory Shape .adgetShape ( "SQUARE" );
                // call the draw method for a square
                shape3 .draw ();
        }
}
```

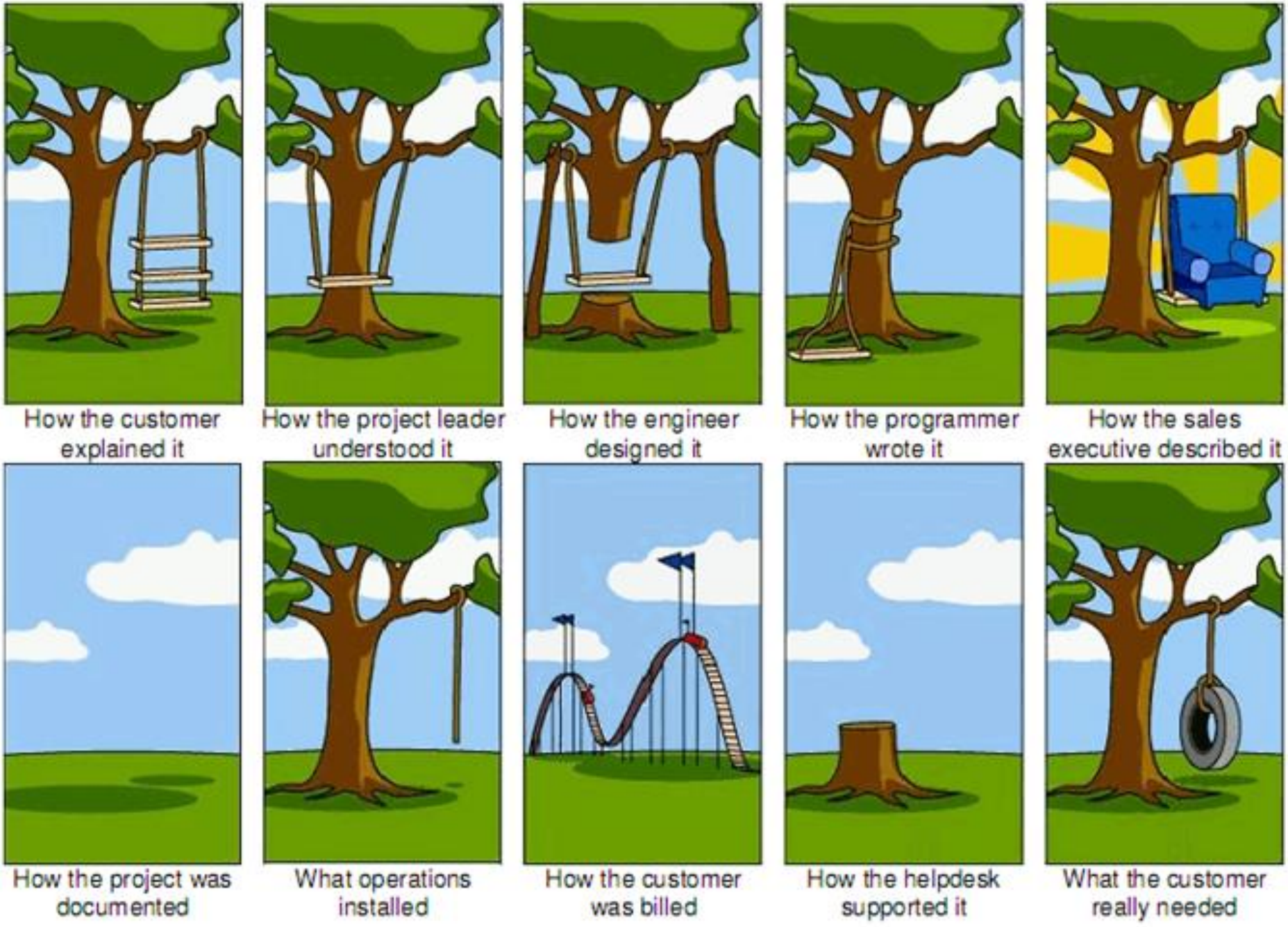
Project forms

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns: Elements of Reusable ObjectOriented Software, Addison Wesley 1994 - known as “Gang of Four” - “GoF”

- Serbian edition: Ready solutions, CET Belgrade, 2002.






How the customer explained it
How the project leader understood it
How the engineer designed it
How the programmer wrote it
How the sales executive described it
How the project was documented
What operations installed
How the customer was billed
How the helpdesk supported it
What the customer really needed

Task for class work

- Create a "vehicle factory" using the Factory project form.

- An interface needs to be created Product and the specific classes that implement this interface. It is also necessary to define the class "factory" - FabrikaVozila

- A test class should also be created FormFactoryTest which will use the class FabrikaVozila in order to obtain a suitable facility of some type of vehicle. The test class will only pass information to the vehicle factory whether it is a car, truck or motorcycle, and the class

  FabrikaVozila will then "deliver" the requested vehicle to the test class

Exercise tasks

- Study the other most important project patterns (in addition to Singleton i Factory, these are e.g. Builder, Adapter, Facade, Iterator, Visitor, MVC) and modify the classes in the package employed so that they use project forms in situations where it is adequate.

- Modify the remaining classes developed during the OOP course so that project templates are applied in situations where appropriate.