



Class Diagrams & Design Patterns

CE-1103 Algorithms and Data Structures I

ВЕДОМОСТЬ ПРЕСЛОВ		NN	размер послово ВхН, мм
		позначит	размер послово ВхН, мм
1	5	9	1510 x 2070
	10	9	910 x 2070
	2	10	910 x 1510
2	5	10	2310 x 2110
	2	10	2310 x 2110

Disclaimer / Descargo de Responsabilidad

Esta presentación corresponde a una guía usada por el profesor durante las clases. La misma ha sido modificada para ser utilizado en el modelo de cursos asistidos por tecnología. No es una versión final, por lo que la misma podría requerir todavía hacer algunos ajustes. Para aspectos de evaluación esta presentación es solo una guía, por lo que el estudiante debe profundizar con el material de lectura asignado y lo discutido en clases para aspectos de evaluación.

This presentation corresponds to a guide material used by the professor during classes. It has been modified to be used in the model of technology-assisted courses. It is not a final version, so it may still require some adjustments. For evaluation aspects, this presentation is only a guide, so the student should delve with the assigned reading material and what has been discussed in class.

Let's be honest...

- When you receive a project specification, what do you do?
- ◆ Do you start coding right away?
 - ◆ Do you carefully plan everything and then code?
 - ◆ Do you make a balance between planning and coding?



Let's be honest...

- Most people goes direct to code!
 - ◆ Too much design and thinking is for cowards! - they say
 - ◆ Kids code apps in their bedrooms!
- Do you see any risk in this approach?



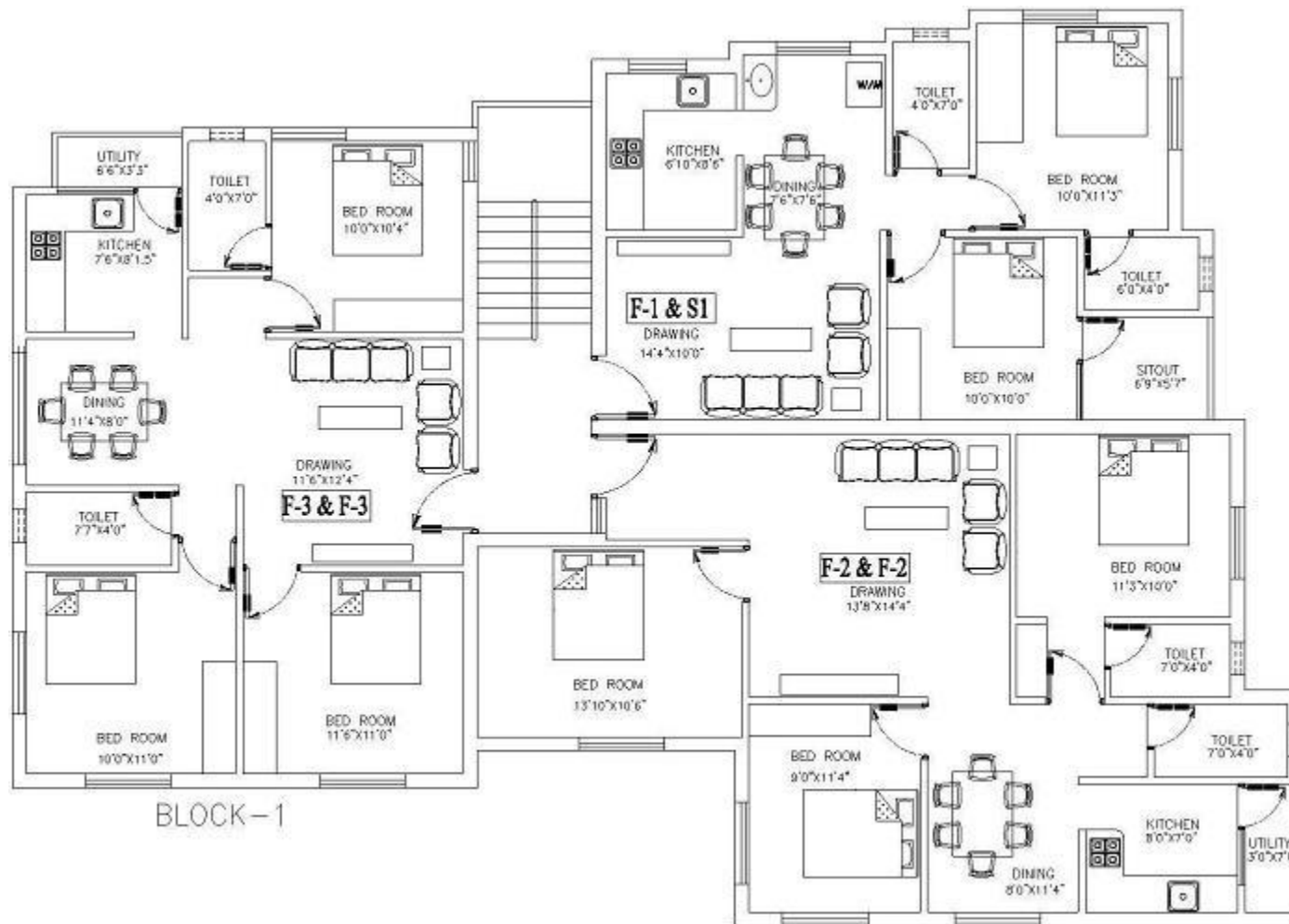
Professional vs empiric



Professional vs empiric



Professional vs empiric



For computer engineers...

- There are formal ways to do our jobs
 - ◆ Project management methodologies
 - ◆ Development methodologies
 - ◆ Standards and frameworks
 - ◆ Best practices
 - ◆ Tools

- These are some of the things that separates an engineer from the kid coding in his room..

In this topic

- We will focus just on:
 - ◆ Class diagrams
 - ◆ Design Patterns
- Take a guess, can you explain each of these items?

Before going into details...

- We need to have some notion of what is UML
- We will revisit this topic in the next course
- For now, we just need to know what it is





Unified Modeling Language

What is UML?

- UML stands for **U**nified **M**odeling **L**anguage
 - ◆ Standard language for writing software blueprints
- The usage of UML give you a professional and standard approach to create artifacts during the SDLC

What is UML?

- UML stands for **U**nified **M**odeling **L**anguage
 - ◆ Standard language for writing software blueprints
- The usage of UML give you a professional and standard approach to create artifacts during the SDLC



Software
Development Life
Cycle

What is UML?

- Think of UML as the **formal notation for computer engineers**
- Just like the drawings of a house blueprint can be understood by **any** architect, UML can be understood by **any** computer engineer

Brief history of UML

- After the arrival of the OOP, many object modeling techniques started to emerge
- In the years between 1989 and 1994 at least 50 object oriented methods with different modeling languages

Brief history of UML

- In the middle of this chaos, there were three prominent methods:
 - ◆ Booch
 - ◆ OOSE (Object Oriented Software Engineering)
 - ◆ OMT (Object Modeling Technique)
- Each of these had strengths and weaknesses

Brief history of UML

- The authors of these three methods joined forces to unify them in one single modeling language
- Grady Booch, Ivar Jacobson and James Rumbaugh created a consortium for UML
- Many companies joined this consortium:
 - ◆ IBM, HP, Oracle, Unisys, Texas Instruments, Microsoft, among many others

Brief history of UML

- In 1996 the first version of UML was released
- Currently the Object Management Group maintains the UML standard



Why modeling is necessary?

- We all want to build good software, so:
 - ◆ We build models to **communicate** the desired structure and behavior of our system
 - ◆ We build models to **visualize and control** the system's architecture
 - ◆ We build models to better **understand** the system we are building
 - ◆ We build models to **manage risk**

- Is not the same to build the house of a dog to build a skyscraper

Why modeling is necessary?

→ We model because is a **well-accepted engineering technique!**

- ◆ Constructions & Architecture
- ◆ Electronics
- ◆ Sociology
- ◆ Economics

→ A model is:

- ◆ A simplification of reality (**Abstraction**)

Why modeling is necessary?

- UML is composed of three types of models:
 - ◆ Functional model
 - ◆ Object model
 - ◆ Dynamic model
- Each model is composed by a **set of diagrams**
- One of the most common type of diagrams is **class diagrams**

Class diagram

- Also known as **static view**
- This diagram captures the object structure. Includes all the data structures and the operations on the data
- It does not contains any information related to the dynamic behavior

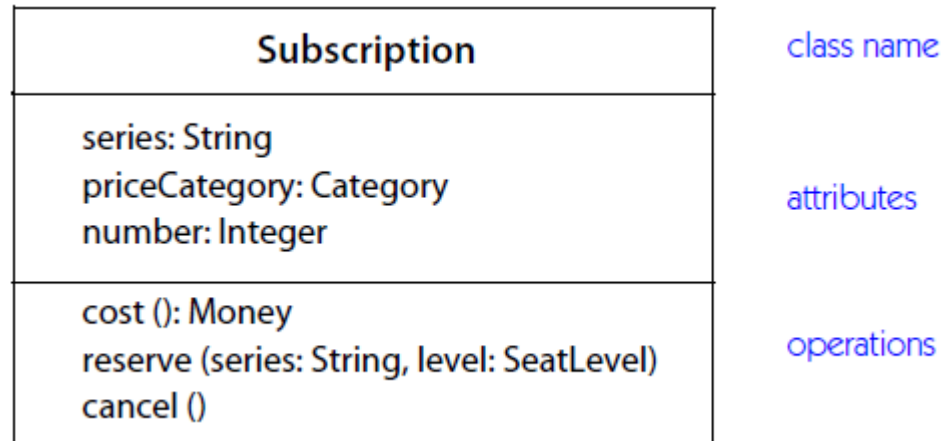


Run time

Unified Modeling Language:



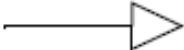
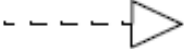
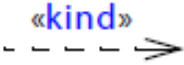
Class diagram

→ A class is represented as:

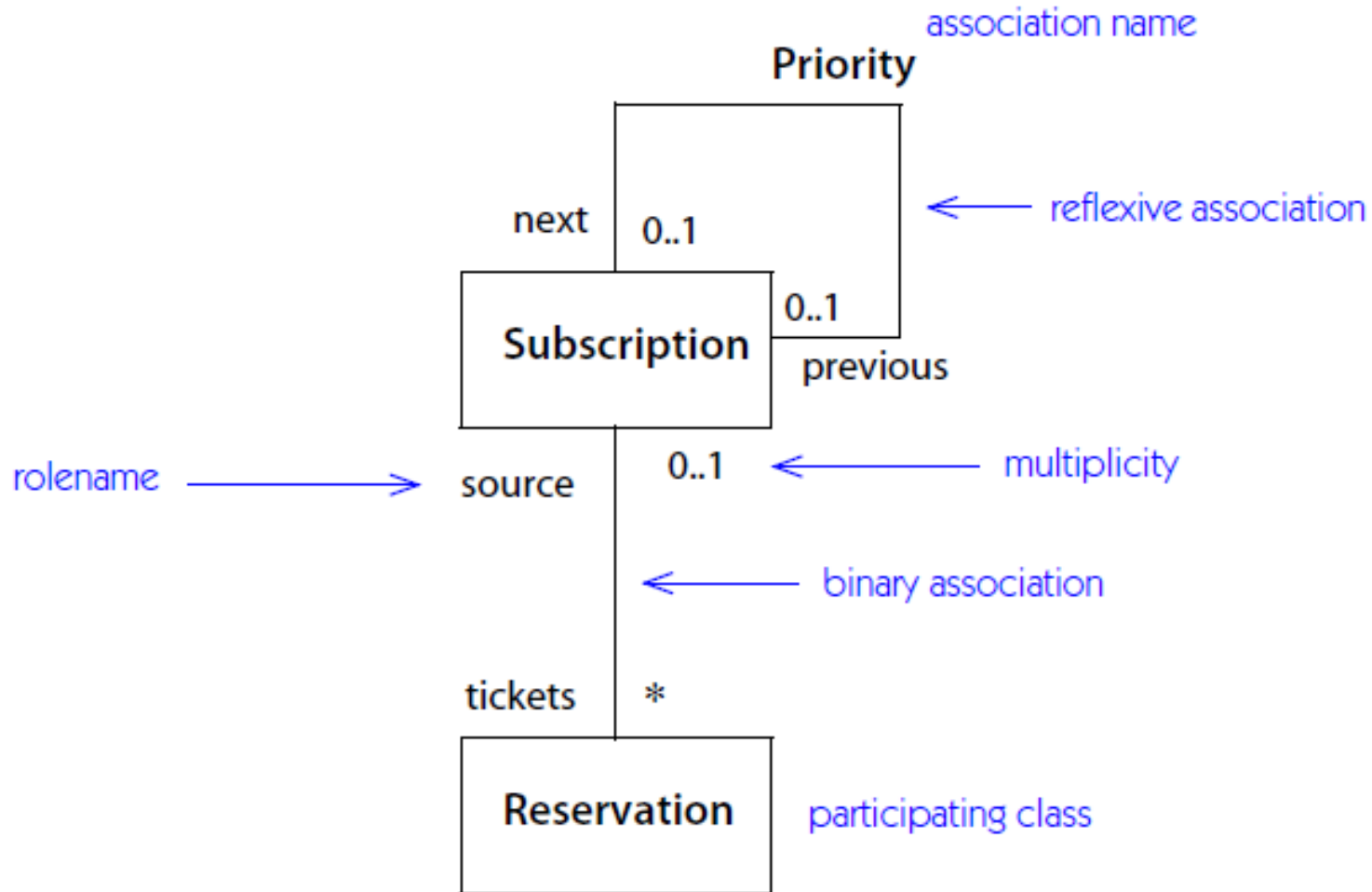


Unified Modeling Language:

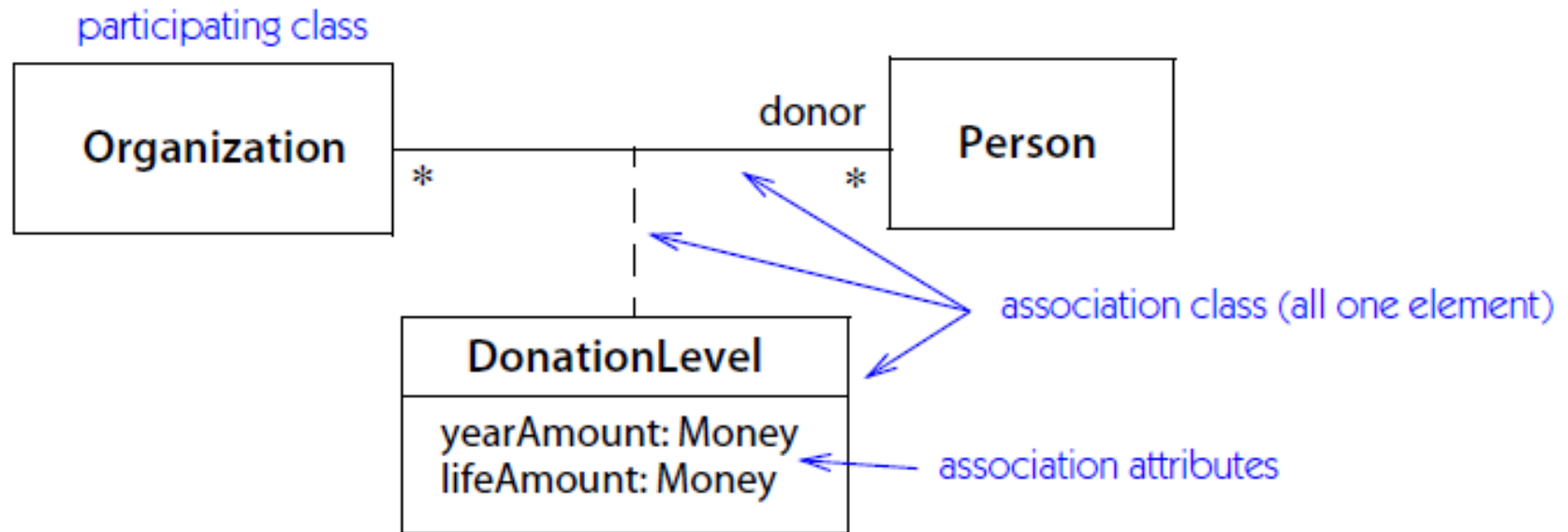
Class diagram

<i>Relationship</i>	<i>Function</i>	<i>Notation</i>
association	A description of a connection among instances of classes	
dependency	A relationship between two model elements	
generalization	A relationship between a more specific and a more general description, used for inheritance and polymorphic type declarations	
realization	Relationship between a specification and its implementation	
usage	A situation in which one element requires another for its correct functioning	

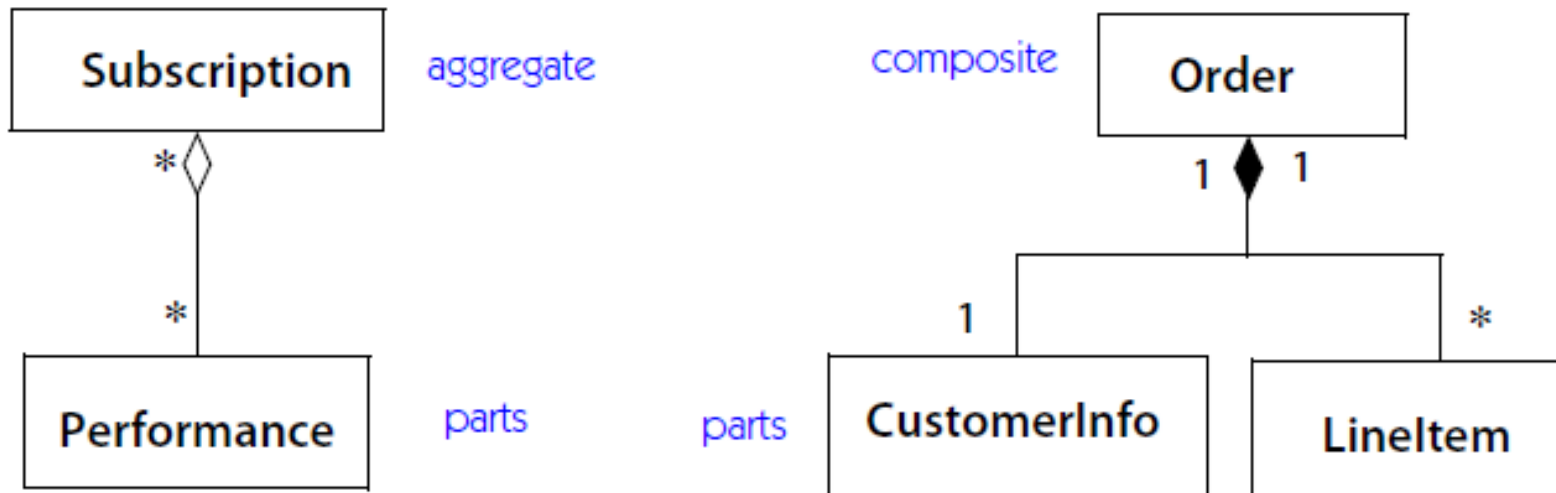
Class diagram: Association



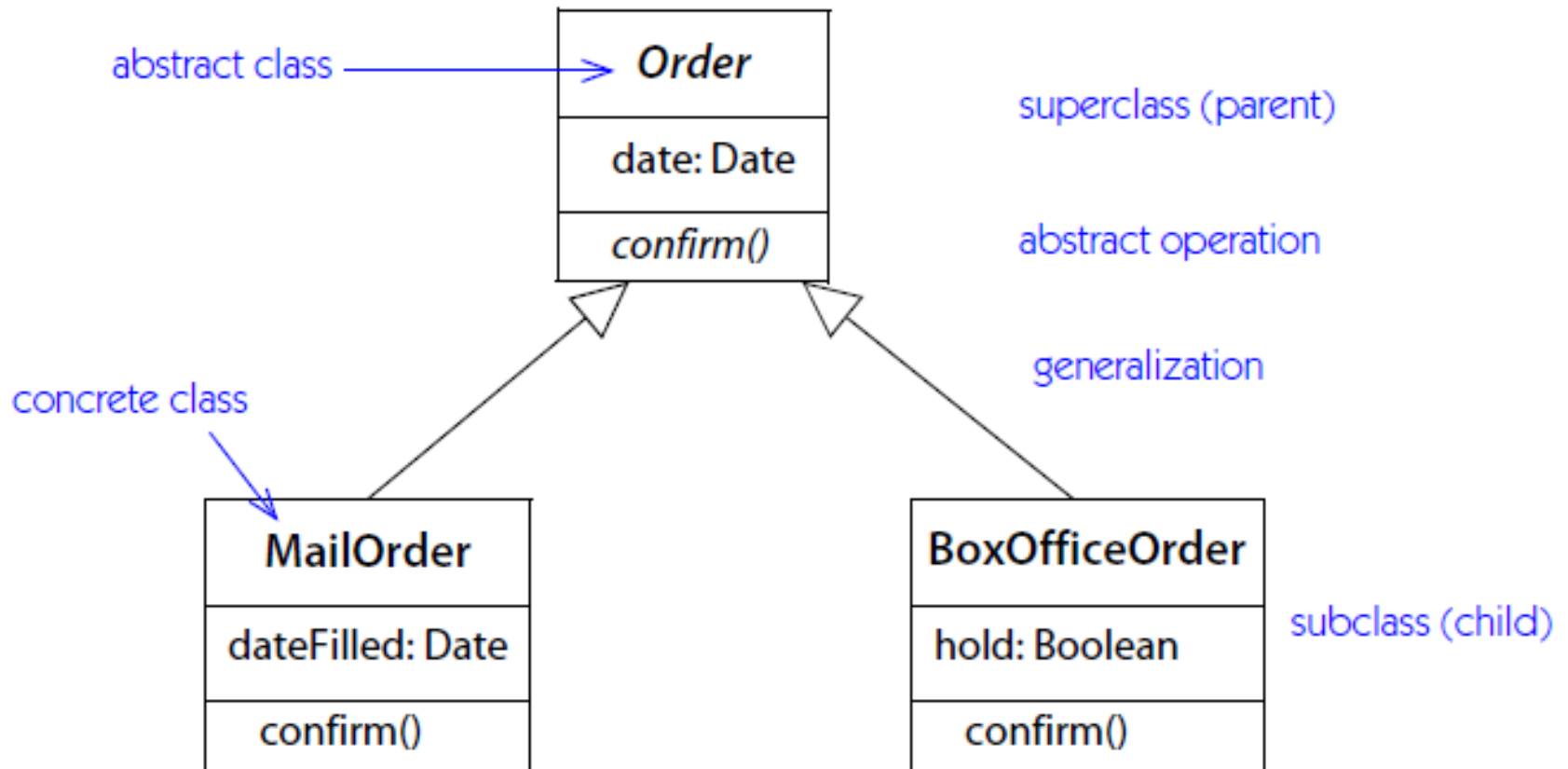
Class diagram: Association



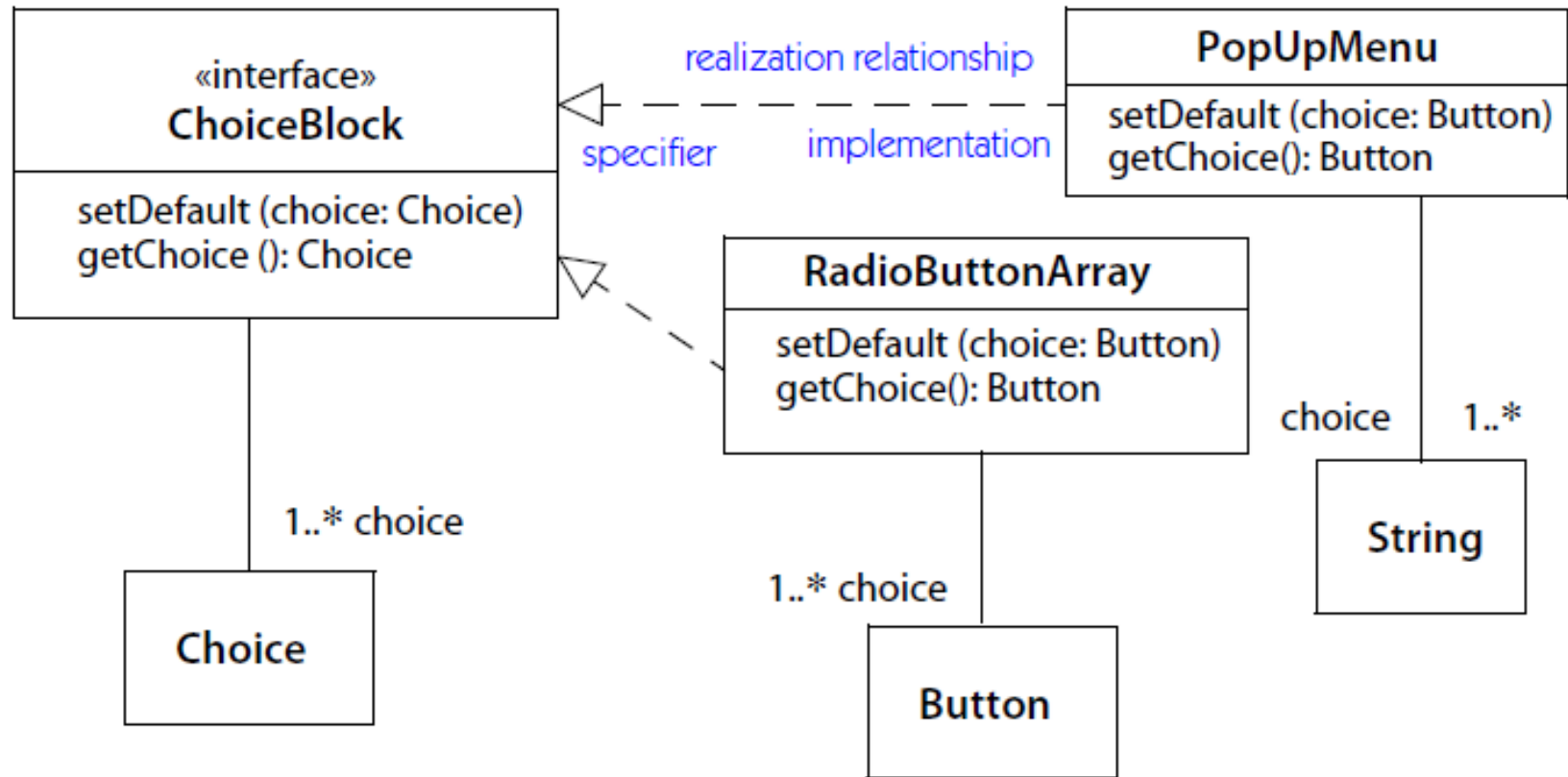
Class diagram: Association



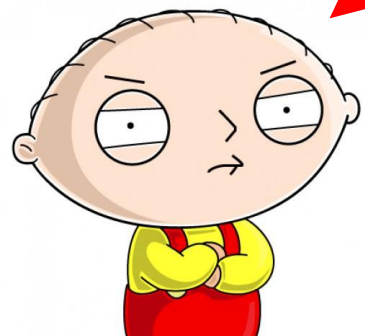
Class diagram: Generalization



Class diagram: Realization



**Where can I create
class diagrams?**



Class diagram: Tools

- There are many commercial and open source tools that you can use to create class diagrams.
- Some even generate code from the diagram (**forward engineering**) or generate diagrams from existing code (**reverse engineering**)



Unified Modeling Language:

Class diagram: Tools

→ Some useful tools are:

- ◆ Microsoft Visio
- ◆ Lucid Chart (Cloud-based solution)
- ◆ IBM Rational Software Architect
- ◆ Eclipse plugins
- ◆ DIA





Object-Oriented Design

Let's pause for a moment...

- Like most things in life, there is more than one way of doing things
- Solving one problem may have many different solutions



Let's pause for a moment...

- What is a **good** design of a solution?
- Now, in terms of software, what is a **good** design?
- It depends of the "eye of the beholder!"



Let's pause for a moment...

- Although there is some level of subjectivity in the term "good", there are some principles of what is good
- So now the question is: how can I achieve a good design in OOP?



Object-oriented design

- Object-oriented design is a process that can help you create “good” designs for OOP
- It also involves a set of principles that can give you a **sense** of correctness in your work



Object-oriented design

- This OOD process will start after the project has received "green light"
- You will learn more about development projects and project management in time...
- For now OOD will be enough



Object-oriented design

→ OK...let's assume you have a problem you need to solve using OOP...how do you tackle it? Where do you start?

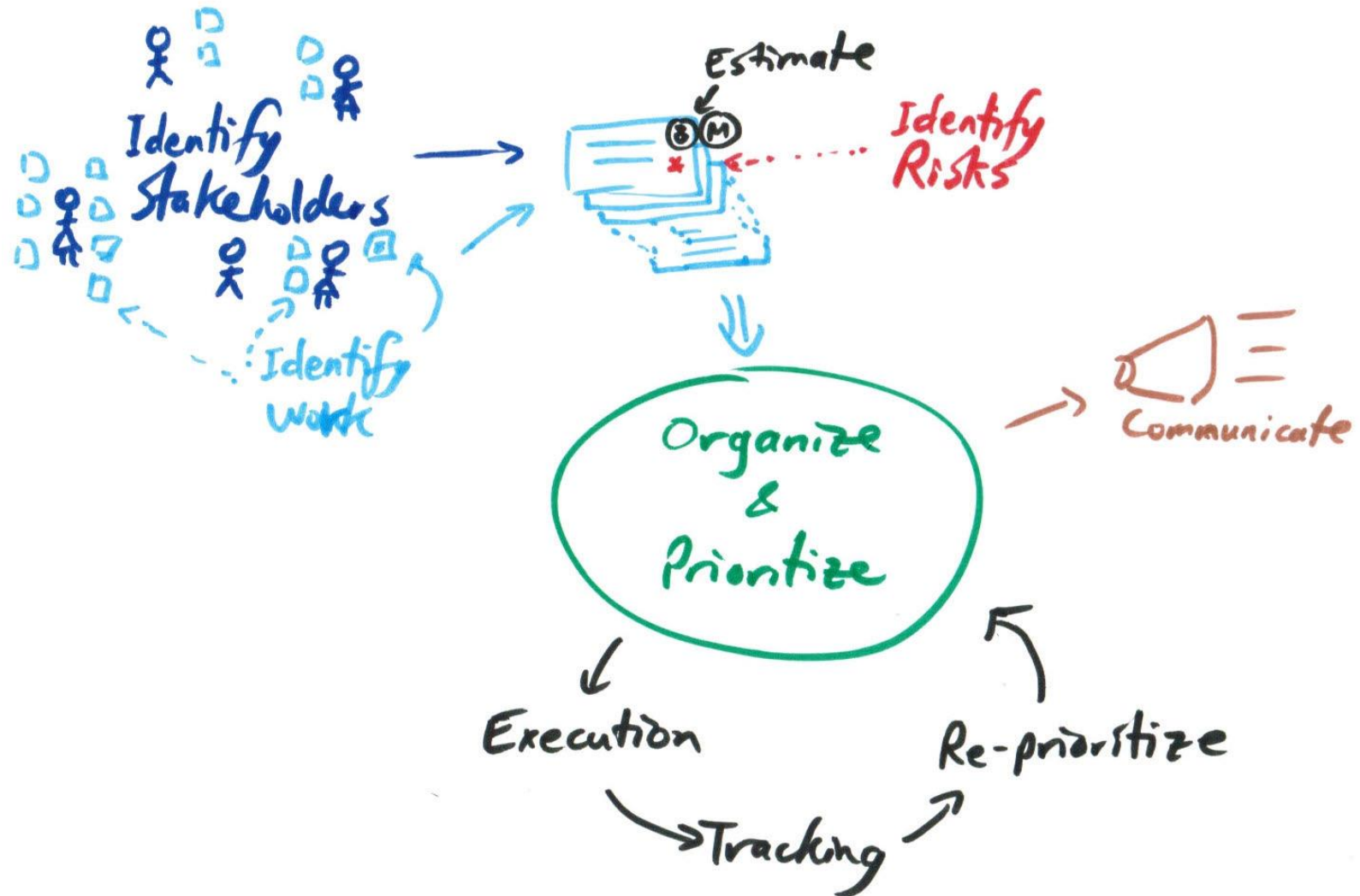


Conceptual modeling

- Start by understanding the domain of the problem and create a conceptual model
- This conceptual model can be done however you feel better, it doesn't have to be elaborated
- Make sure you understand!!

Object-oriented design

Conceptual modeling



Object-oriented design

Conceptual modeling

- You want to gain insights of how the current problem fits in the bigger picture
- So an overall diagram will do the trick...

Object-oriented design

Analysis phase

- In this phase you need to do a deeper research to understand what you need to do
- Most of the time is not clear of what he/she wants...
- You can use **user stories**

Object-oriented design

Analysis phase

→ User stories are a simple way to collect **requirements**. For example:

As a user, I want to upload photos so that I can share photos with others.

As an administrator, I want to approve photos before they are posted so that I can make sure they are appropriate.

Object-oriented design

Analysis phase

- Another useful tool to help you understanding the problem is creating prototypes
- A **prototype** is a very basic version of the system or solution you want to build. The main objective is show users how the final solution might look

Object-oriented design

Design phase

- Is the final step of the OOD process. With the data you have recovered before, you can now begin creating your design
- The output of this phase is the class diagram
- This phase can be done in steps

Object-oriented design

Design phase

→ **Step #1: Identify classes**

Pay special attention on the substantives in the user stories

Never try to find all classes at once or think that everything will be do at first attempt

In software things tend to evolve iteratively

Object-oriented design

Design phase

→ **Step #1: Identify classes**

You may even identify candidate classes that will be remove later...

The final solution can look totally different from the first sketches

Sometime very useful to do is to assign the responsibilities to each class. **A class should have only one responsibility**

Object-oriented design

Design phase

→ Step #2: Identify Associations

Identify how the classes interact between each other. For example, you may find that the *Car* class needs the class *Engine* or the class *Break*

Finding associations may result in removing classes or adding new ones

Classes should not be isolated

→ **Step #3: Identify Attributes and Methods**

Add the attributes and methods to each identified class.

Try to think from an overall point of view and seeing how each class will play its role. This will help you identifying its behavior and attributes

Object-oriented design

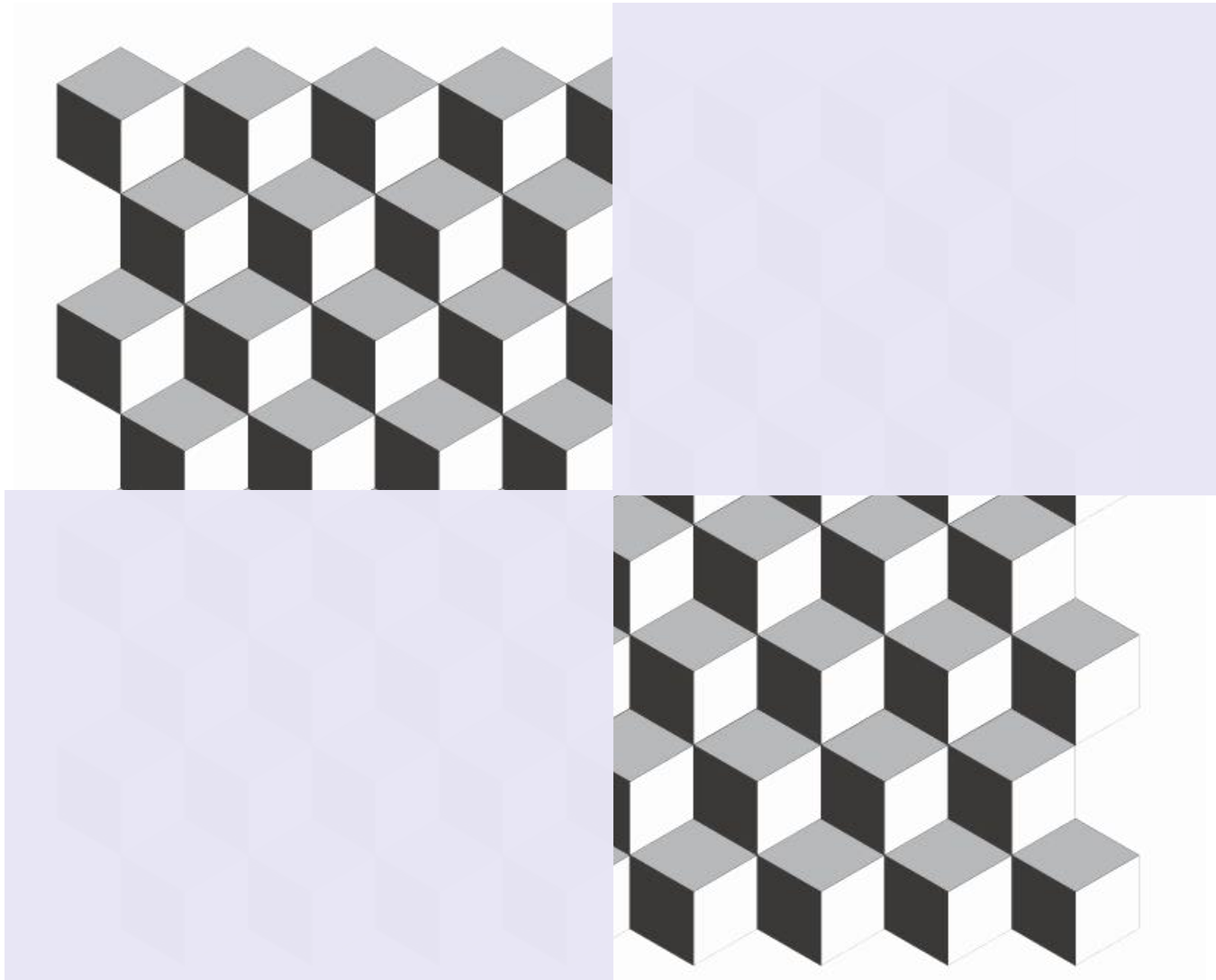
- At first creating an OOP design is a hard task
- As time goes by, you'll become more and more experienced and the process will become more natural





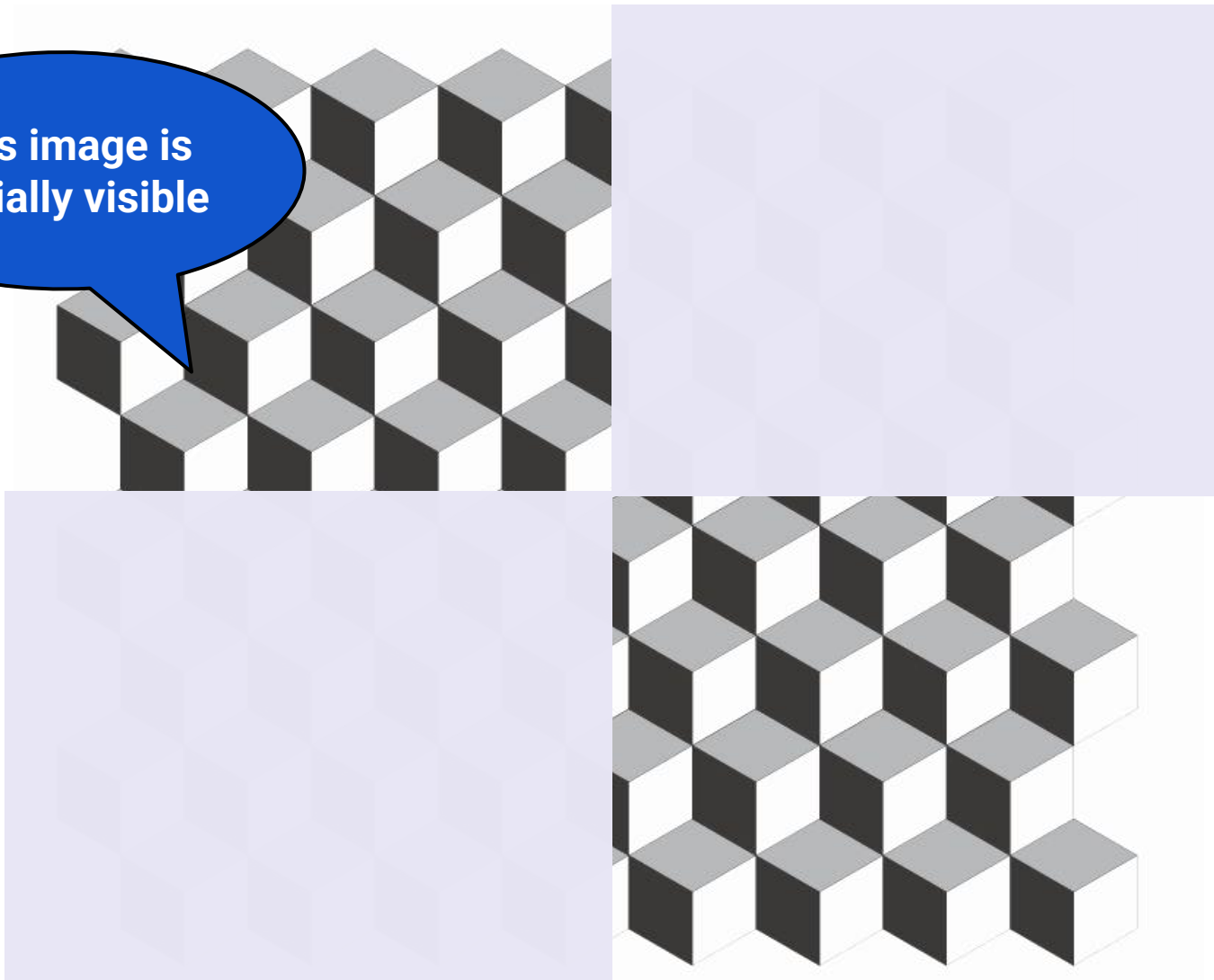
Design Patterns

What is a pattern?



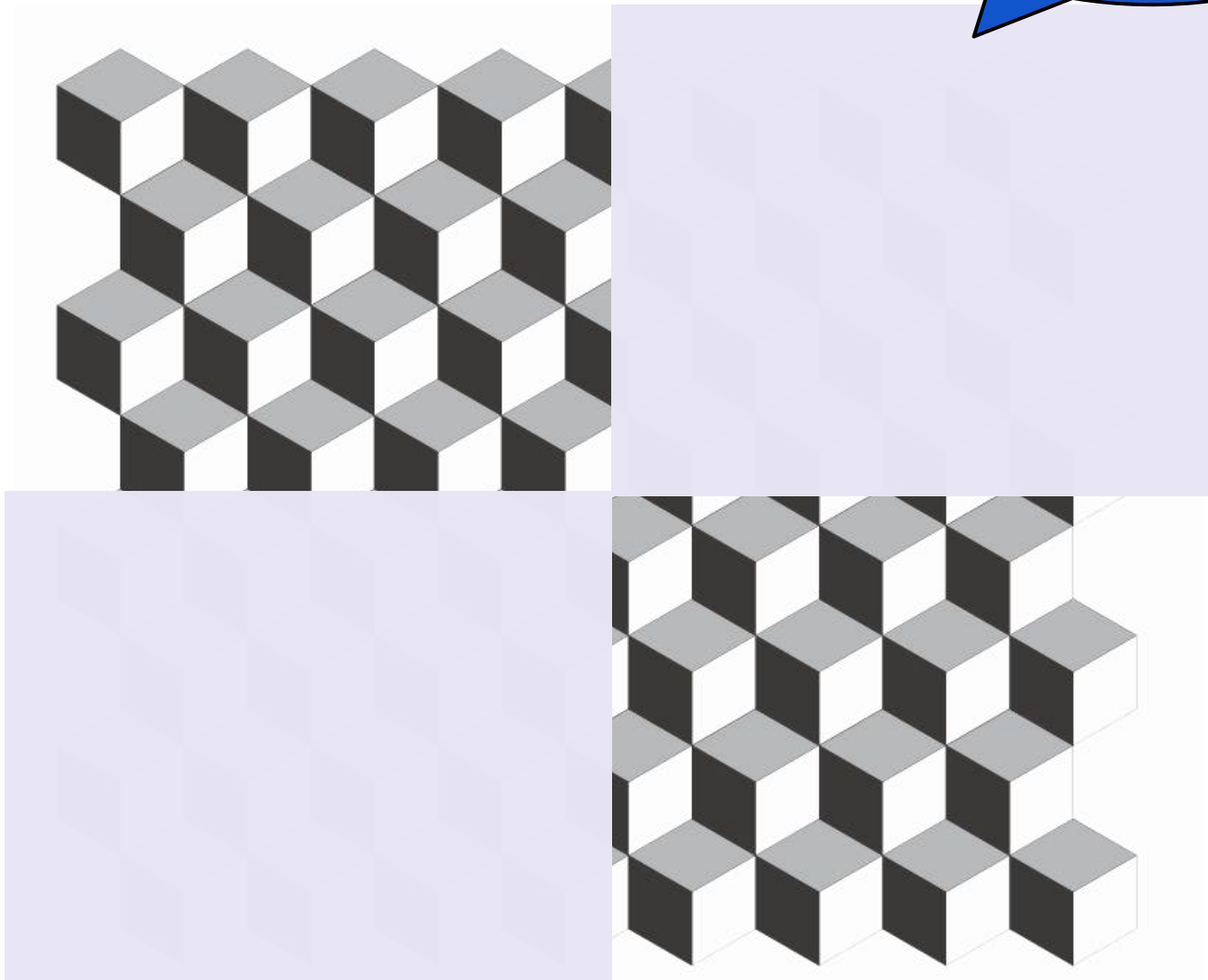
What is a pattern?

This image is
partially visible

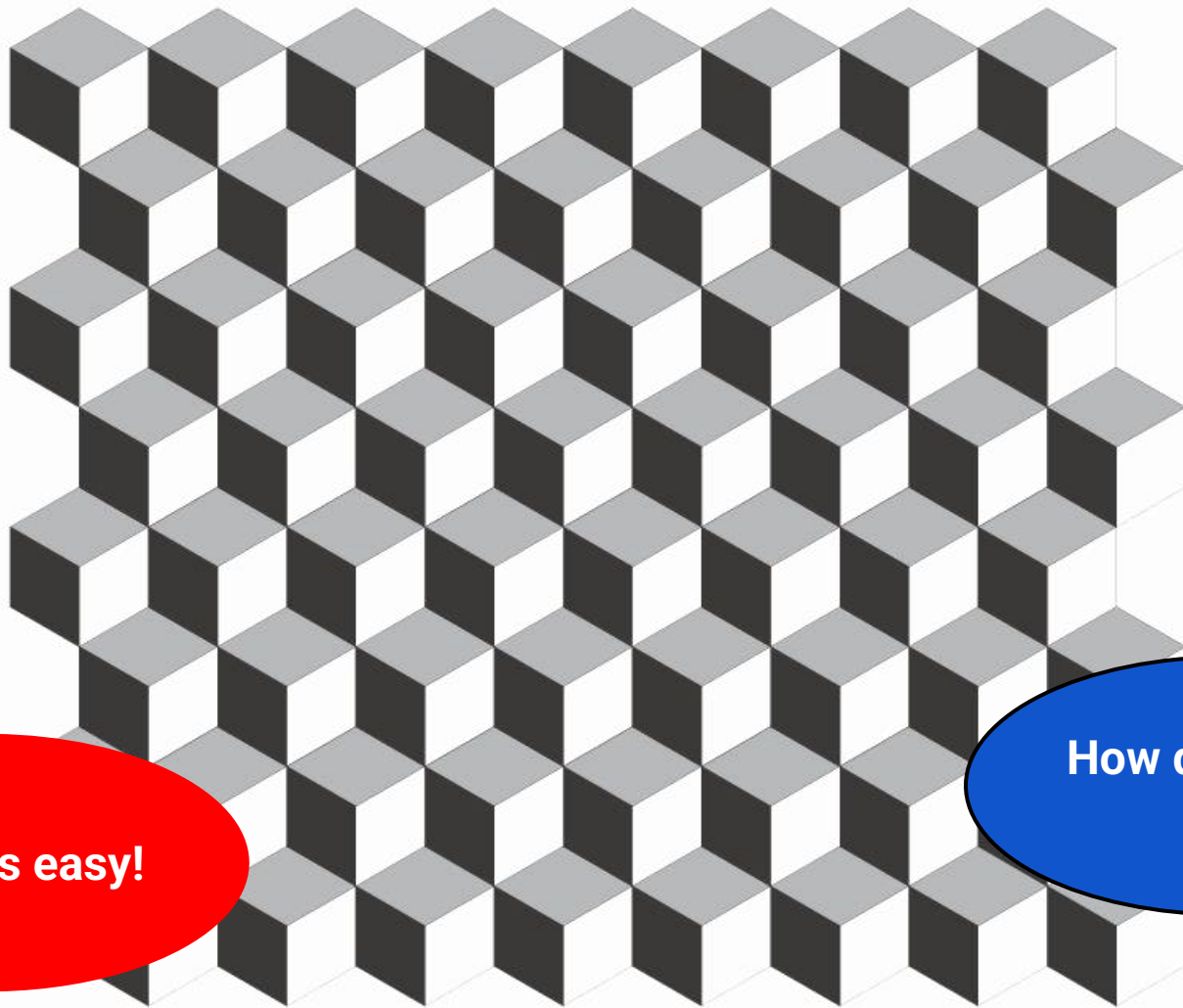


What is a pattern?

Can you tell
what is behind
the squares?



What is a pattern?



That was easy!

**How did you do
it?**

What is a pattern?

- Is a **discernible regularity** in the world or in a manmade design - Wikipedia
- One thing an expert designer know not to do is solve every problem from first principles
 - ◆ Why?
 - ◆ Does this make sense to you?
 - ◆ Have you heard the old saying: don't reinvent the wheel?

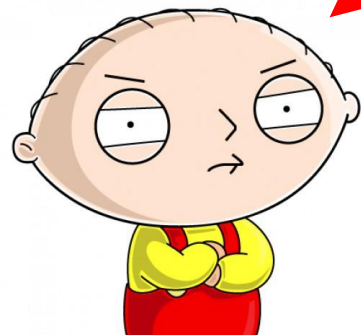
What is a pattern?

- Experts designers reuse solutions that have worked for them in the past
 - ◆ Base new designs on prior experience

What is a pattern?

- Experts designers reuse solutions that have worked for them in the past
 - ◆ Base new designs on prior experience

**But every problem
is unique!!**



What is a pattern?

- Experts designers reuse solutions that have worked for them in the past
 - ◆ Base new designs on prior experience



**But every problem
is unique!!**

**Experience will let
you know when a
solution applies or
not**

**Not every
known solution
applies for all
problems**

Design patterns in the OOP context

- Is a **general reusable solution** to a commonly occurring problem with a given context in software design
- A pattern contains a description of communicating objects and classes that are customized to solve a general design problem in a **particular context**

Design patterns in the OOP context

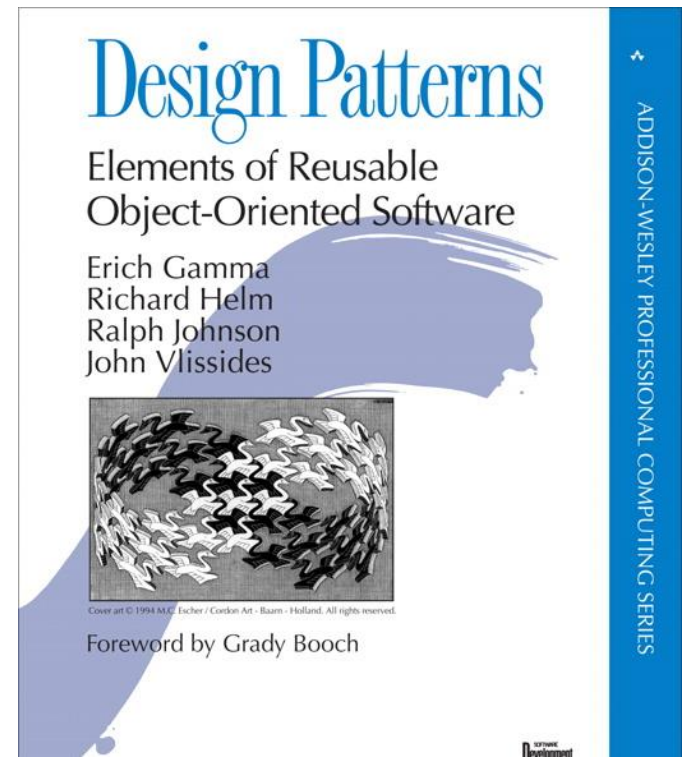
- A pattern can be thought as a good practice, a **proven solution** for a problem
- Design patterns are usually known by many software professionals across the world.

Design patterns:

History background

→ The first catalog of design patterns was recompiled in 1994 by “the gang of four”:

- ◆ Erich Gamma
- ◆ Richard Helm
- ◆ Ralph Johnson
- ◆ John Vlissides



Design patterns:

History background

- They recompile best practices known informally by many engineers
- Also they proposed patterns designed by themselves
- The book has become a classic and a reference for all software engineers

Design patterns:

Categories

- Patterns can be classified by two criteria: **purpose** and **scope**.
- Purpose reflect what a pattern does
- Scope specifies if the pattern applies to classes or objects

Design patterns:

Categories

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method (107)	Adapter (class) (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

Design patterns:

Categories

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method (107) Abstract Factory (87) Builder (127)	Adapter (class) (139) Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Interpreter (243) Template Method (325) Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

Concerns the
process of object
creation

Design patterns:

Categories

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method (107)	Adapter (class) (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (139) Facade (147) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

Deals with the
composition of
classes or objects

Design patterns:

Categories

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method (107)	Adapter (class) (139)	Interpreter (213) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (223) Strategy (315) Visitor (331)

Characterize the ways
in which classes or
objects interact and
distribute
responsibility

Design patterns:

Categories

Scope		Purpose		
		Structural		Behavioral
		Class	Object	
	Class	Abstract Factory (139)	Bridge (151)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (97) Builder (117) Prototype (127) Singleton (127)	Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

Deal with
relationship
between classes
and subclasses

Design patterns:

Categories

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method (135) Abstract Factory (139)	Adapter (143) Bridge (147) Composite (153) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Interpreter (243) Template Method (325)
	Object	Builder (125) Prototype (129) Singleton (127)		Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

Deal with
relationship between
objects which can be
changed in run-time

Creational patterns

- Abstracts the instantiation process
- Help make a system independent of how its objects are created, composed and represented.



Creational patterns

→ Two main focuses:

- ◆ Encapsulates the knowledge about **which concrete classes** the system uses
- ◆ Hides how instances of the classes are created and put together



Creational patterns

Singleton

→ Purpose:

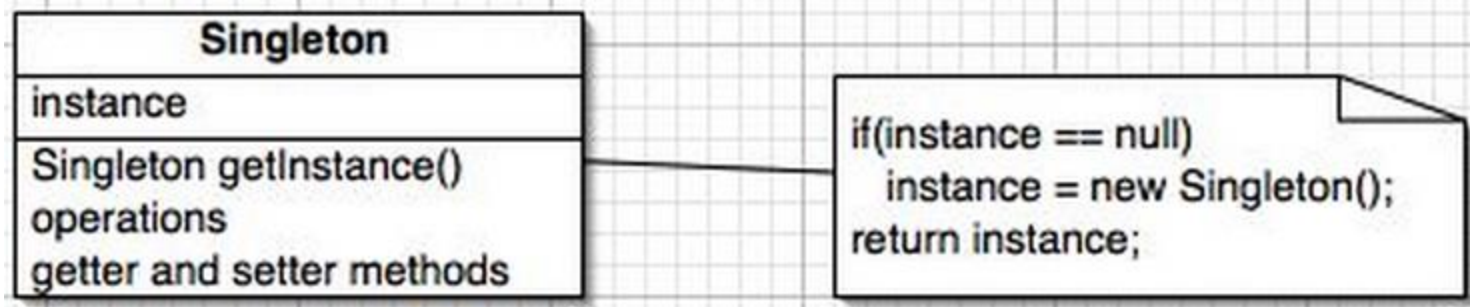
- ◆ Ensure that a class has one instance and provide a global point of access to it
- ◆ Singleton ensures that no other instance of a class can be created
- ◆ Singleton provides a way to access the instance



Creational patterns

Singleton

→ Structure:



Creational patterns

Singleton

→ Structure:

```
public class ClassicSingleton {  
    private static ClassicSingleton instance = null;  
    protected ClassicSingleton() {  
        // Exists only to defeat instantiation.  
    }  
    public static ClassicSingleton getInstance() {  
        if(instance == null) {  
            instance = new ClassicSingleton();  
        }  
        return instance;  
    }  
}
```

Singleton

→ Collaborations:

- ◆ Singleton class defines an *Instance* static method that let clients access its unique instance.
- ◆ Singleton is in charge of creating its own instance
- ◆ Clients access the singleton only through the *Instance* method

Creational patterns

Singleton

→ Example:

```
public class LogManager
{
    private java.io.PrintStream m_out;

    private LogManager( PrintStream out )
    {
        m_out = out;
    }

    public void log( String msg )
    {
        m_out.println( msg );
    }
    static private LogManager sm_instance;
    static public LogManager getInstance()
    {
        if ( sm_instance == null )
            sm_instance = new LogManager( System.out );
        return sm_instance;
    }
}
```

Usage:

```
LogManager.getInstance().log( "some message" );
```

Singleton

→ Apply when:

- ◆ There must be exactly one instance of a class and it must be accessible from a well-known access point

Abstract Factory

→ Purpose:

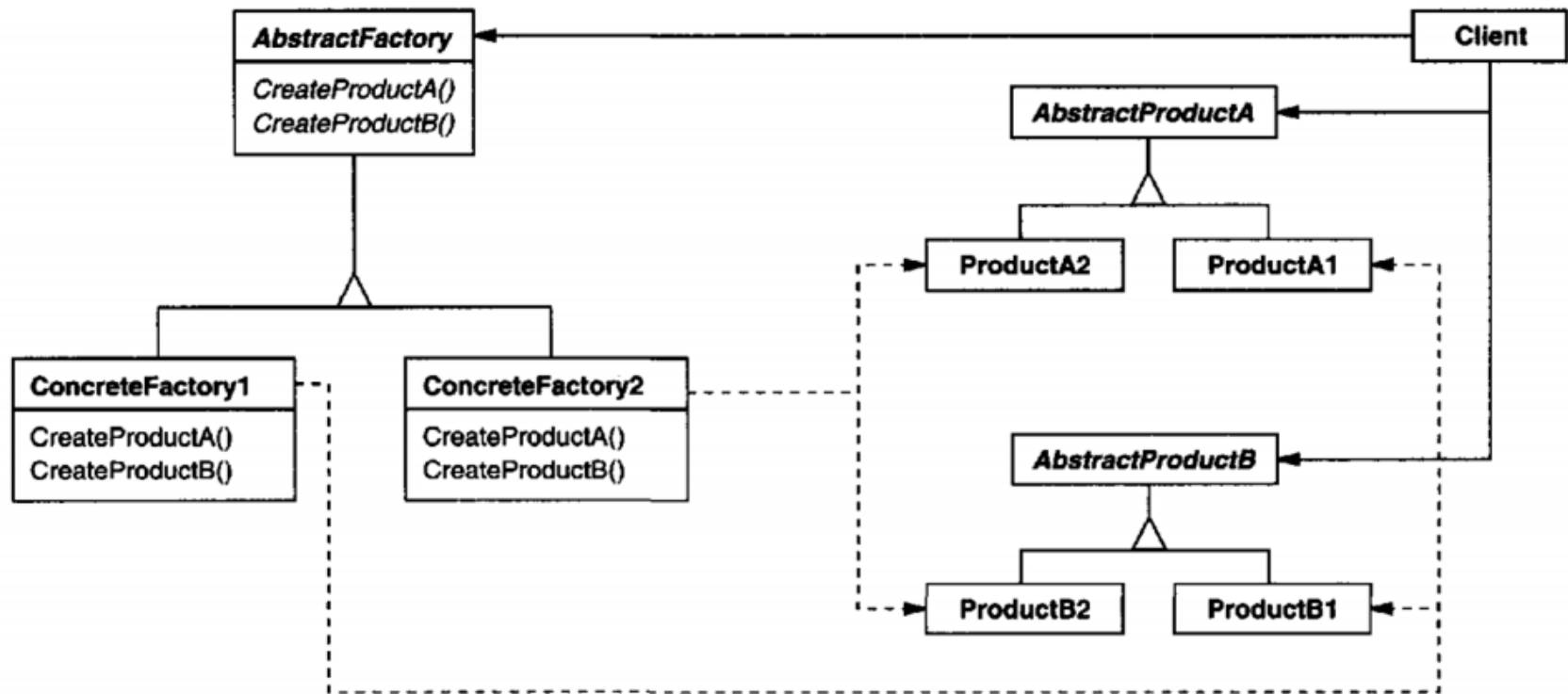
- ◆ Provide an interface for creating families of related or dependent objects without specifying their concrete class



Creational patterns

Abstract Factory

→ Structure:



Abstract Factory

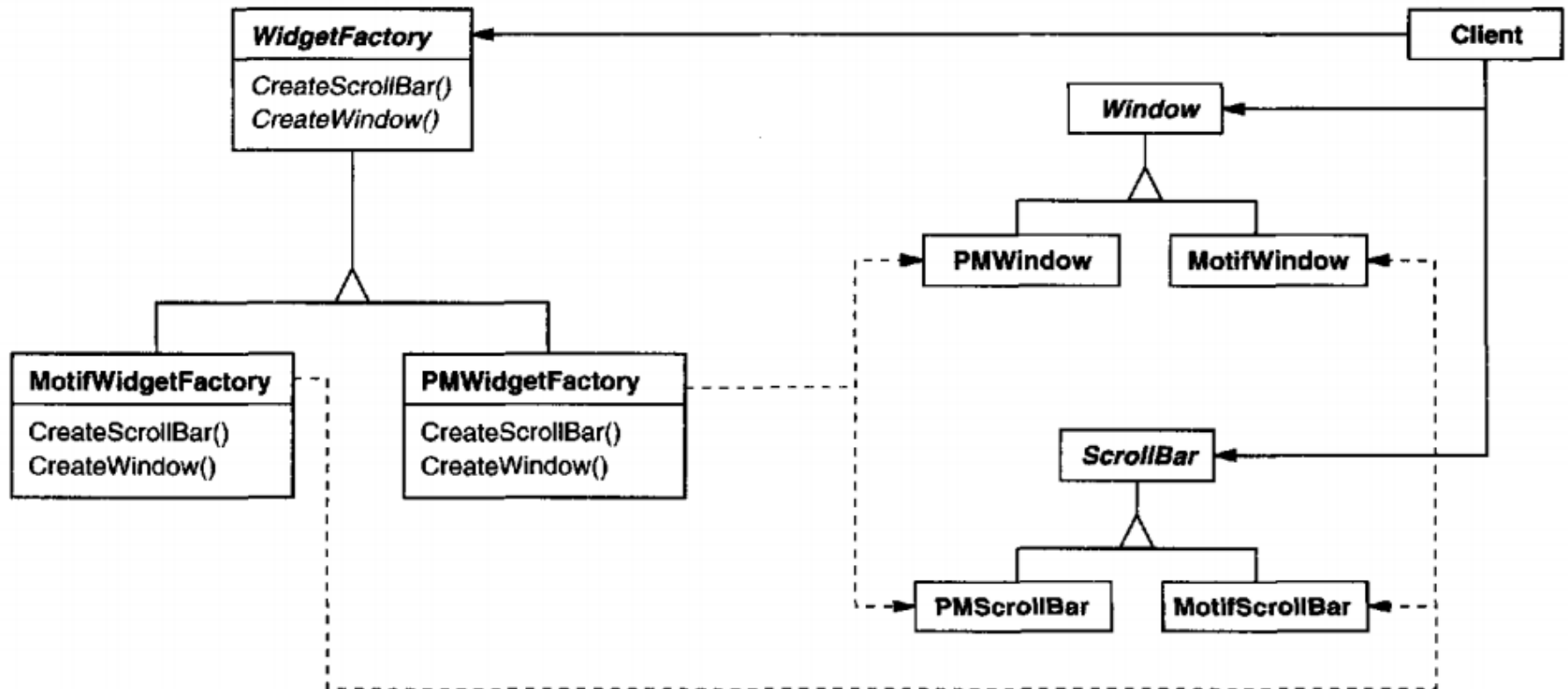
→ Collaborations:

- ◆ Normally a single instance of a ConcreteFactory is created at run-time
- ◆ The concrete factory creates product object having a particular implementation
- ◆ To create different product objects, clients should use a different concrete factory
- ◆ Abstract Factory defers creation to its ConcreteFactory

Creational patterns

Abstract Factory

→ Example:



Creational patterns

Abstract Factory

→ Example:

```
//Abstract Product
interface Button {
    void paint();
}

//Abstract Product
interface Label {
    void paint();
}

//Abstract Factory
interface GUIFactory {
    Button createButton();
    Label createLabel();
}

//Concrete Factory
class WinFactory implements GUIFactory {
    public Button createButton() {
        return new WinButton();
    }

    public Label createLabel() {
        return new WinLabel();
    }
}
```

Creational patterns

Abstract Factory

→ Example:

```
//Concrete Factory
class OSXFactory implements GUIFactory {
    public Button createButton() {
        return new OSXButton();
    }

    public Label createLabel() {
        return new OSXLabel();
    }
}

//Concrete Product
class OSXButton implements Button {
    public void paint() {
        System.out.println("I'm an OSXButton");
    }
}

//Concrete Product
class WinButton implements Button {
    public void paint() {
        System.out.println("I'm a WinButton");
    }
}
```

Creational patterns

Abstract Factory

→ Example:

```
//Concrete Product
class OSXLabel implements Label {
    public void paint() {
        System.out.println("I'm an OSXLabel");
    }
}

//Concrete Product
class WinLabel implements Label {
    public void paint() {
        System.out.println("I'm a WinLabel");
    }
}

//Client application is not aware about the how the product is created. Its only responsible to give a name of
//concrete factory
class Application {
    public Application(GUIFactory factory) {
        Button button = factory.createButton();
        Label label = factory.createLabel();
        button.paint();
        label.paint();
    }
}
```

Creational patterns

Abstract Factory

→ Example:

```
public class ApplicationRunner {  
    public static void main(String[] args) {  
        new Application(createOsSpecificFactory());  
    }  
  
    public static GUIFactory createOsSpecificFactory() {  
        String osname = System.getProperty("os.name").toLowerCase();  
        if(osname != null && osname.contains("windows"))  
            return new WinFactory();  
        else  
            return new OSXFactory();  
    }  
}
```

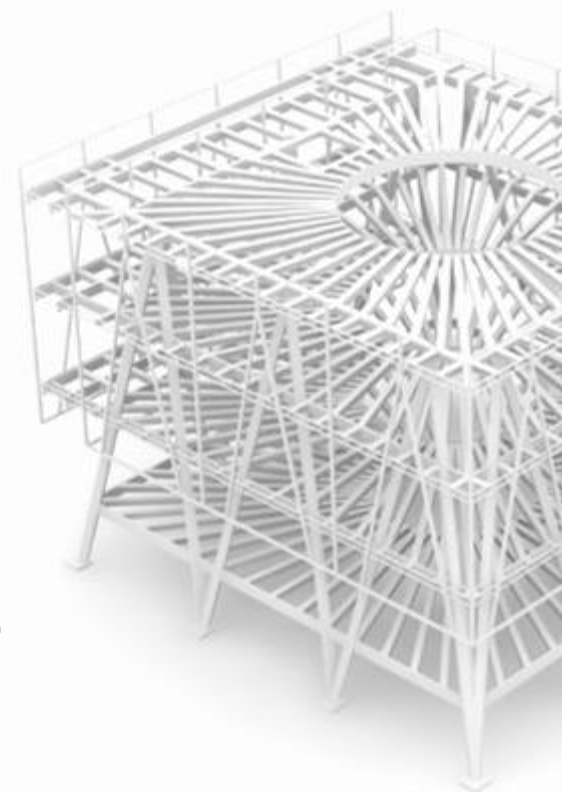
Abstract Factory

→ Apply 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 implementation

Structural patterns

- Are concerned with **how classes and objects are composed** to form larger structures
- Describe ways to compose objects to realize **new functionalities**
- Change composition at run-time



Structural Patterns:

Facade

→ Purpose:

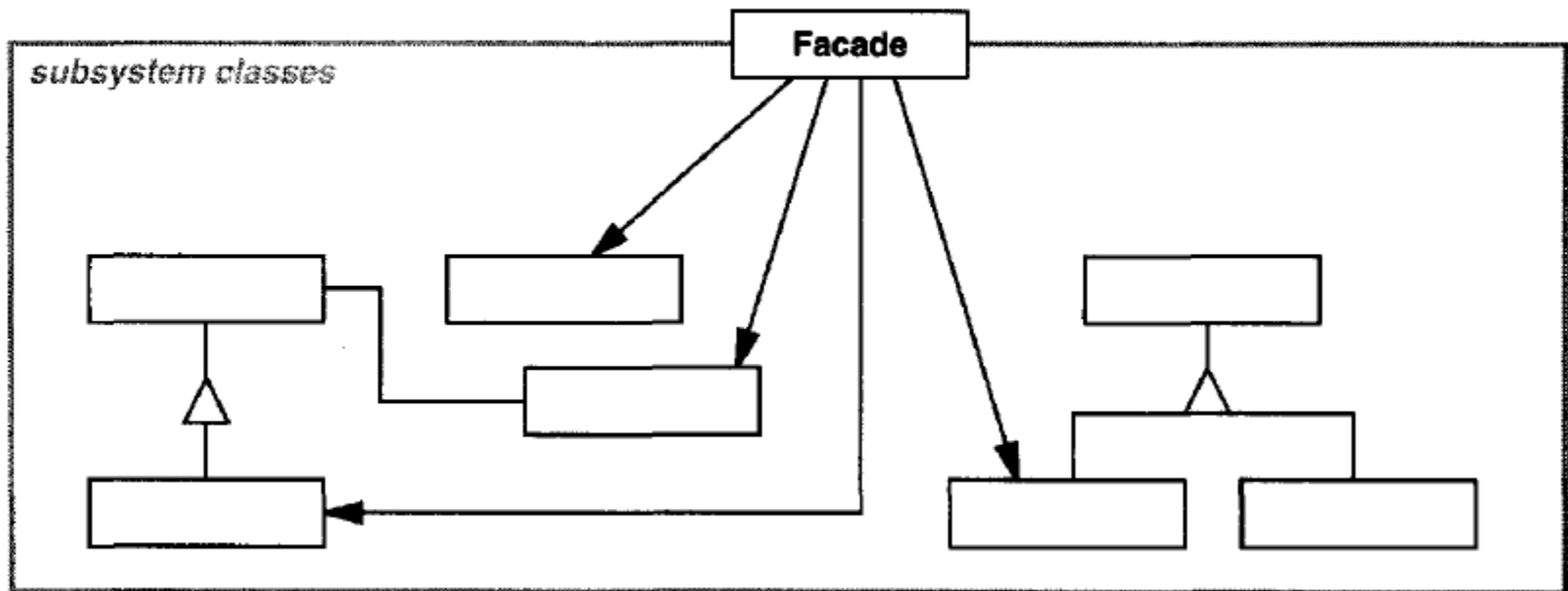
- ◆ "Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher level interface that makes subsystem easier to use"



Structural Patterns:

Facade

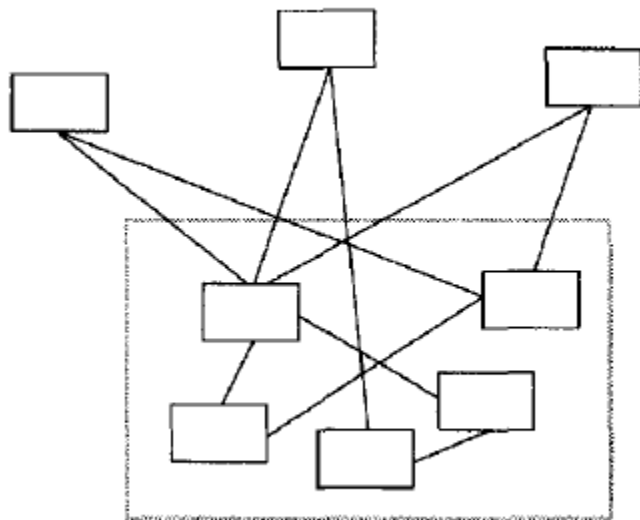
→ Structure:



Structural Patterns:

Facade

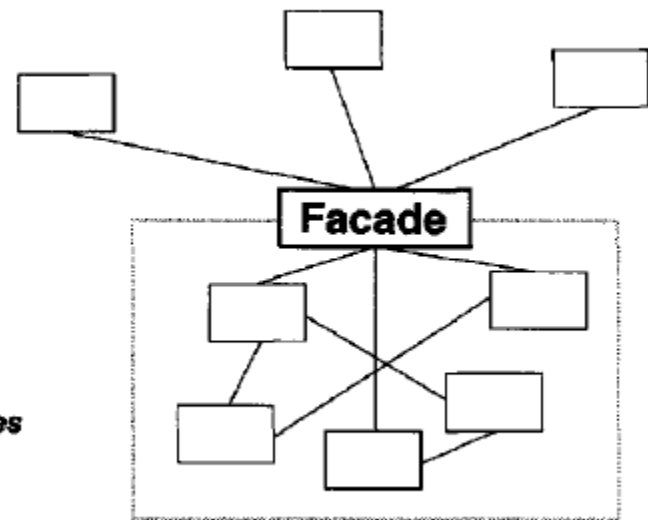
→ Structure:



client classes



subsystem classes



Structural Patterns:

Facade

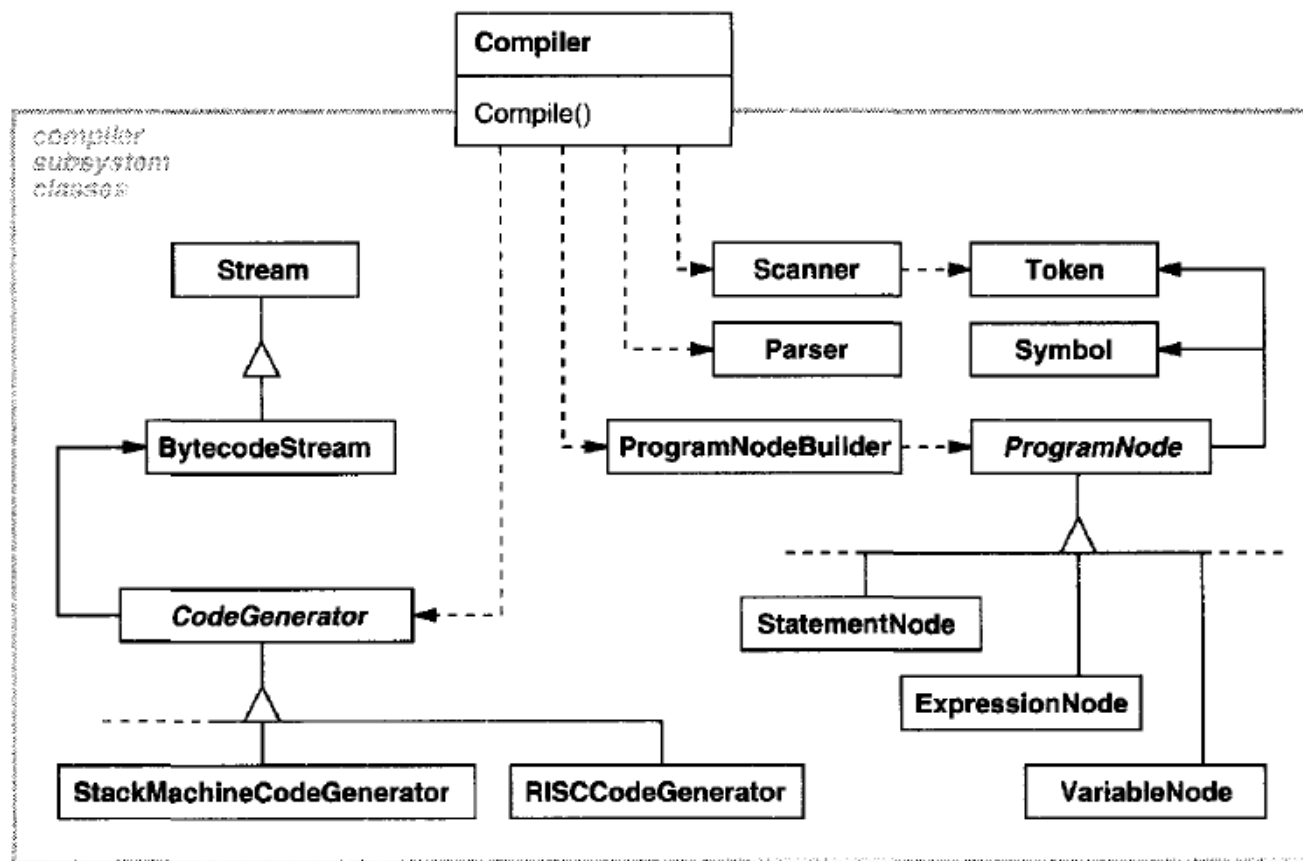
→ Collaborations:

- ◆ Clients communicate with the subsystem by sending request to Facade, which forwards to the appropriate subsystem objects
- ◆ Clients that use the facade don't have access to its subsystem objects directly

Structural Patterns:

Facade

→ Example:



Structural Patterns:

Facade

→ Apply it when:

- ◆ You want to provide a simple interface to a complex subsystem
- ◆ You want to give a simple default view of the subsystem that is good enough for the clients
- ◆ There are many dependencies between clients and the implementation classes. Facade **decouples** the subsystem from clients
- ◆ You want to define an entry point to a subsystem while hiding complexity

Structural Patterns:

Facade

→ Consequences:

- ◆ Shields client from the subsystem components, making the subsystem easier to use
- ◆ Promotes weak coupling between clients and subsystems
- ◆ Don't prevent clients from using subsystem classes directly

Structural Patterns:

Adapter

→ Purpose:

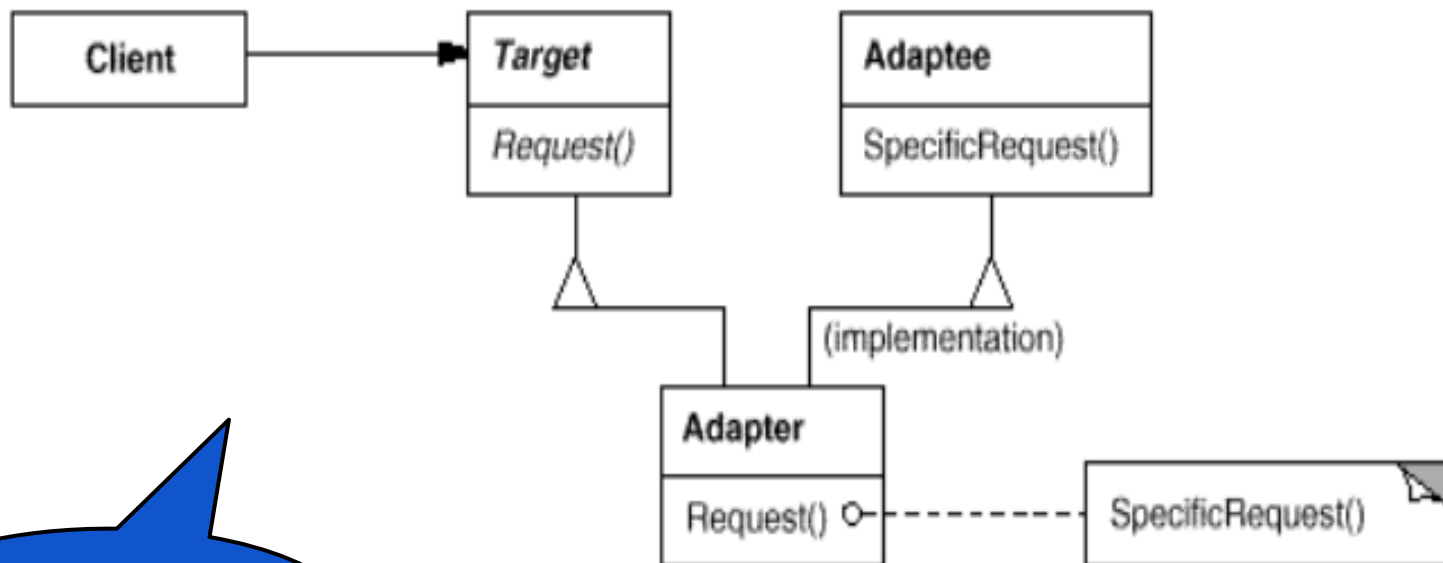
- ◆ "Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces."



Structural Patterns:

Adapter

→ Structure:

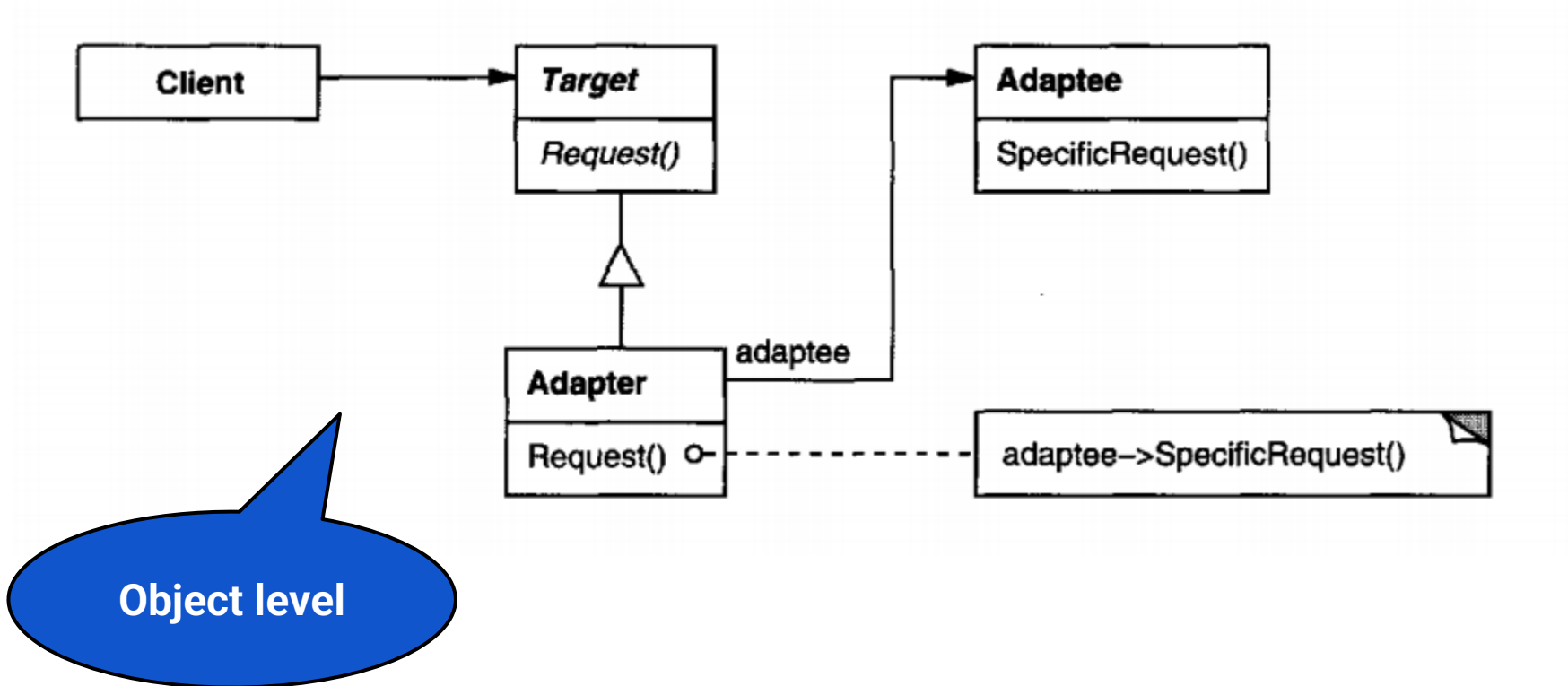


Class level

Structural Patterns:

Adapter

→ Structure:



Structural Patterns:

Adapter

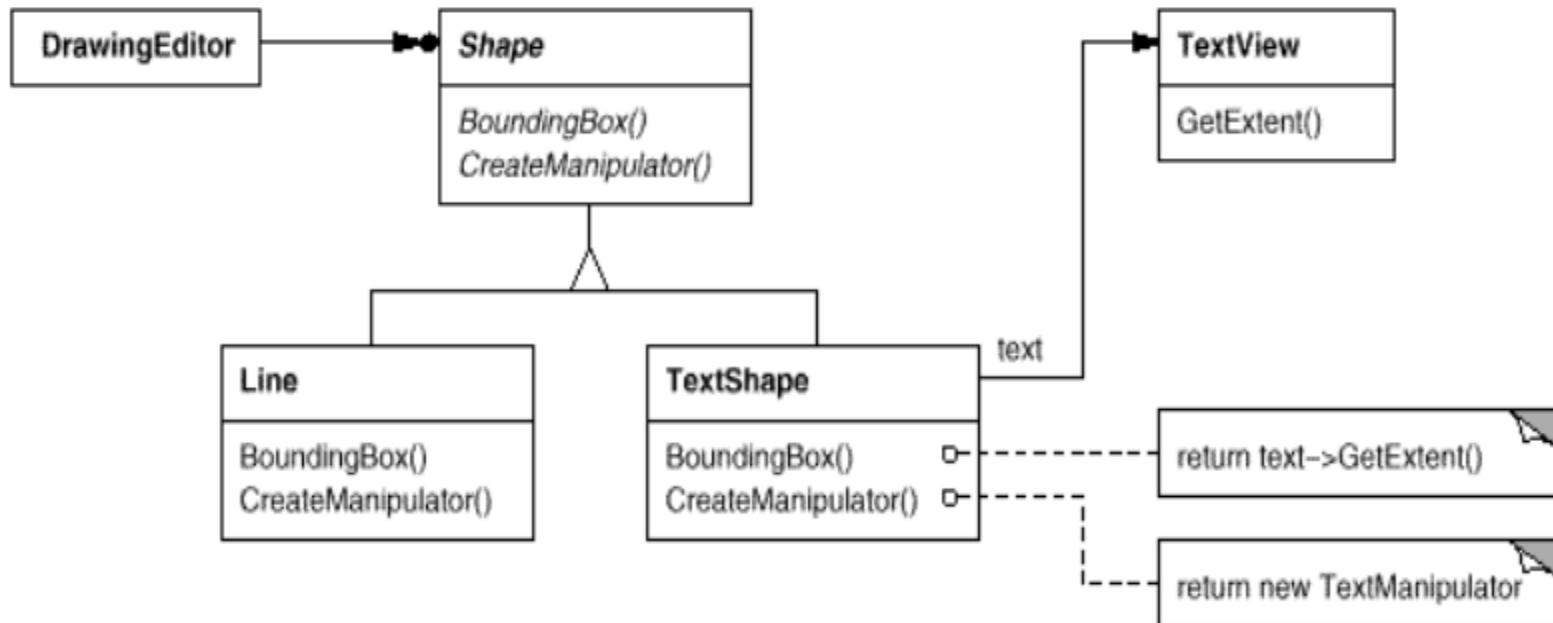
→ Collaborations:

- ◆ Clients call operations on an Adapter instance. In turn the adapter calls adaptee operations that carry out the request

Structural Patterns:

Adapter

→ Example:



Structural Patterns:

Adapter

→ Apply it when:

- ◆ You want to use an existing class, and its interface does not match the one you need.
- ◆ You want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.
- ◆ (object adapter only) You need to use several existing subclasses, but it's impractical to adapt their interface by subclassing everyone.

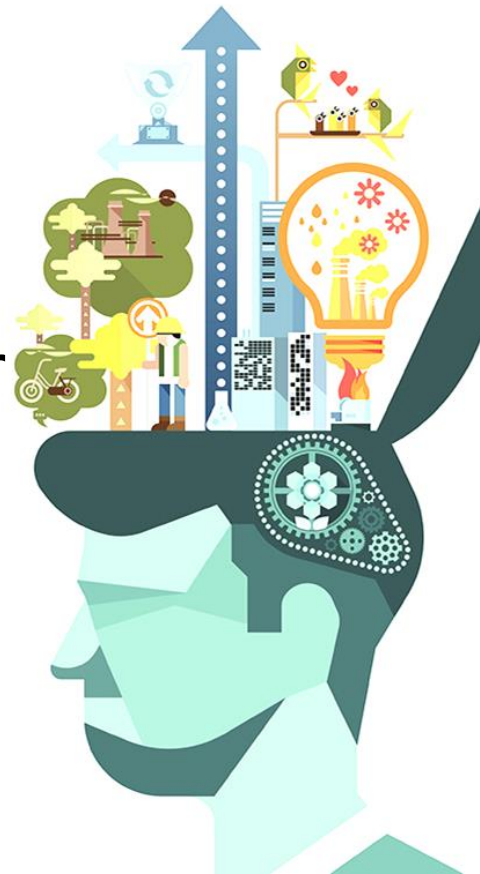
Structural Patterns:

Adapter

- Consequences on class adapter:
 - ◆ Won't work when we want to adapt a class and all its subclasses
 - ◆ Lets override some of Adaptee's behavior
- Consequences on object adapter:
 - ◆ The Adapter can also add functionality to all adaptees (all subclasses) at once.
 - ◆ Harder to override adaptee behavior

Behavioral Patterns

- Are concerned with algorithms and the assignment of responsibilities between objects
- Describe the patterns of communication between classes or objects
- Let you concentrate on the interconnection between objects



Behavioral Patterns:

Observer

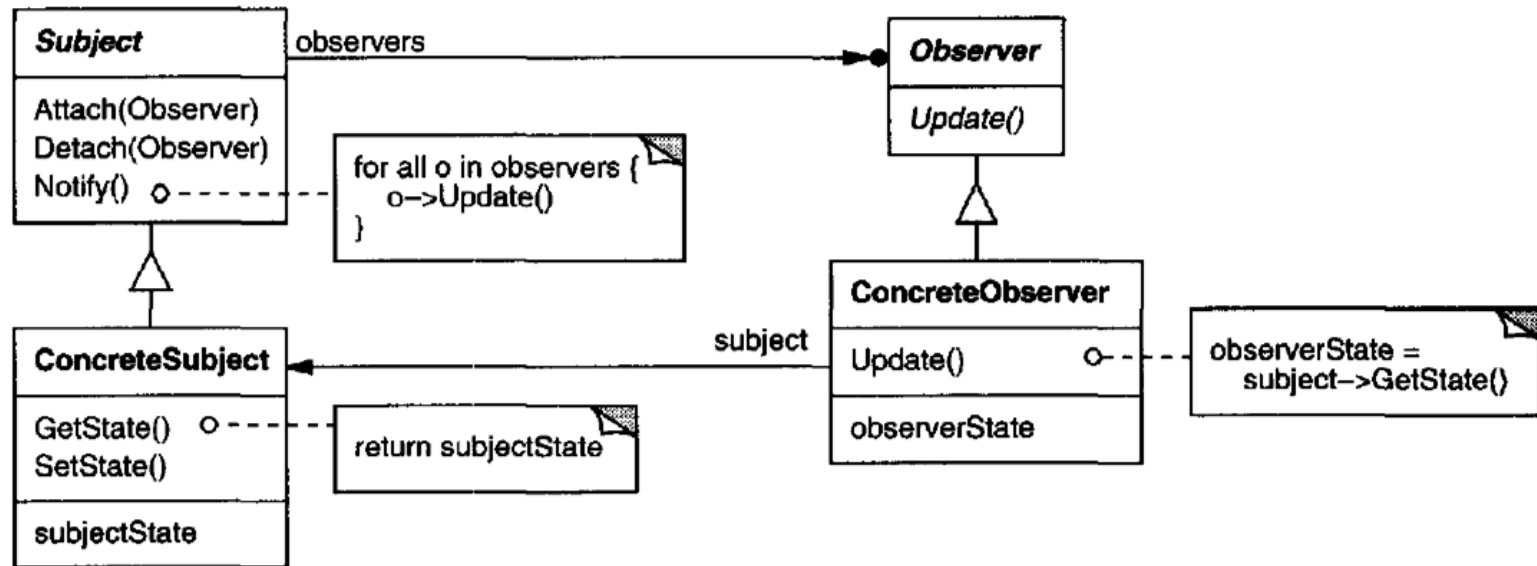
→ Purpose:

- ◆ "Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically"

Behavioral Patterns:

Observer

→ Structure:



Behavioral Patterns:

Observer

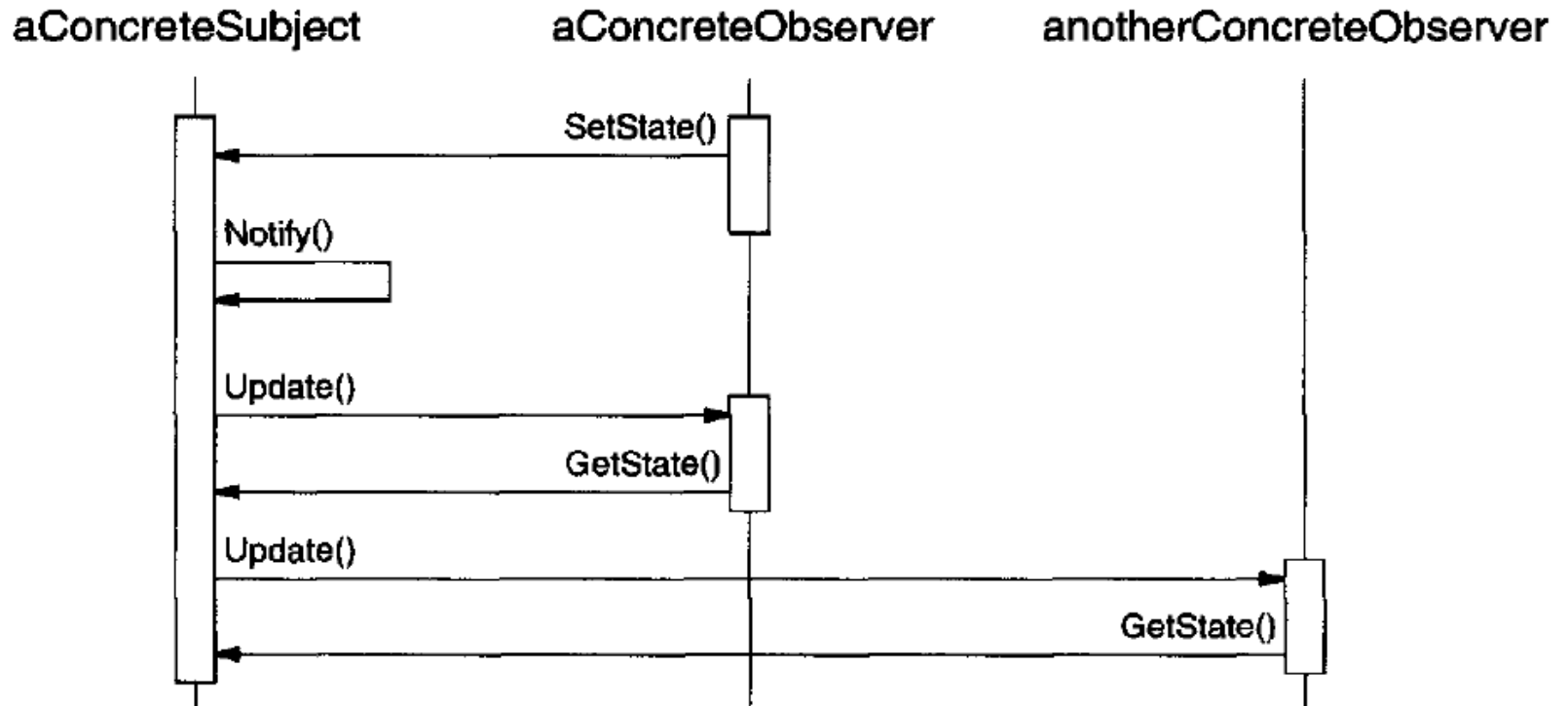
→ Collaborations:

- ◆ ConcreteSubject notifies its observers whenever a change occurs that could make its observers' state inconsistent with its own
- ◆ After being informed of a change in the concrete subject, a ConcreteObserver may query the subject for information

Behavioral Patterns:

Observer

→ Collaborations:



Behavioral Patterns:

Observer

→ Example:

```
public class ObservDemo extends Object {
    MyView view;

    MyModel model;

    public ObservDemo() {
        view = new MyView();

        model = new MyModel();
        model.addObserver(view);
    }

    public static void main(String[] av) {
        ObservDemo me = new ObservDemo();
        me.demo();
    }

    public void demo() {
        model.changeSomething();
    }

    /** The Observer normally maintains a view on the data */
    class MyView implements Observer {
        /** For now, we just print the fact that we got notified. */
        public void update(Observable obs, Object x) {
            System.out.println("update(" + obs + "," + x + ");");
        }
    }

    /** The Observable normally maintains the data */
    class MyModel extends Observable {
        public void changeSomething() {
            // Notify observers of change
            setChanged();
            notifyObservers();
        }
    }
}
```

Behavioral Patterns:

Observer

→ Apply when:

- ◆ An abstraction has two aspects: one dependent on the other
- ◆ A change to one object requires changing others and you don't know how many others are
- ◆ An object should be able to notify other objects without making assumptions who they are

Behavioral Patterns:

Observer

→ Consequences:

- ◆ Decouples subject and observer
- ◆ Supports broadcast communication
- ◆ Unexpected updates: observers have no knowledge of each other's presence they don't know the cost of each change for the overall system
- ◆ Is hard for observers know what changed

Design patterns

→ Why do we need design patterns?





Architectural Patterns:

Model-View-Controller

Architectural patterns

- We learned that design patterns are good practices of how to structure software at the **class level**
- Design patterns are applied at the design phase of a system



Architectural patterns

- Before design starts, software architects define the **architecture** of the system
- Architecture involves defining the components of the system. Components are the pieces of the system



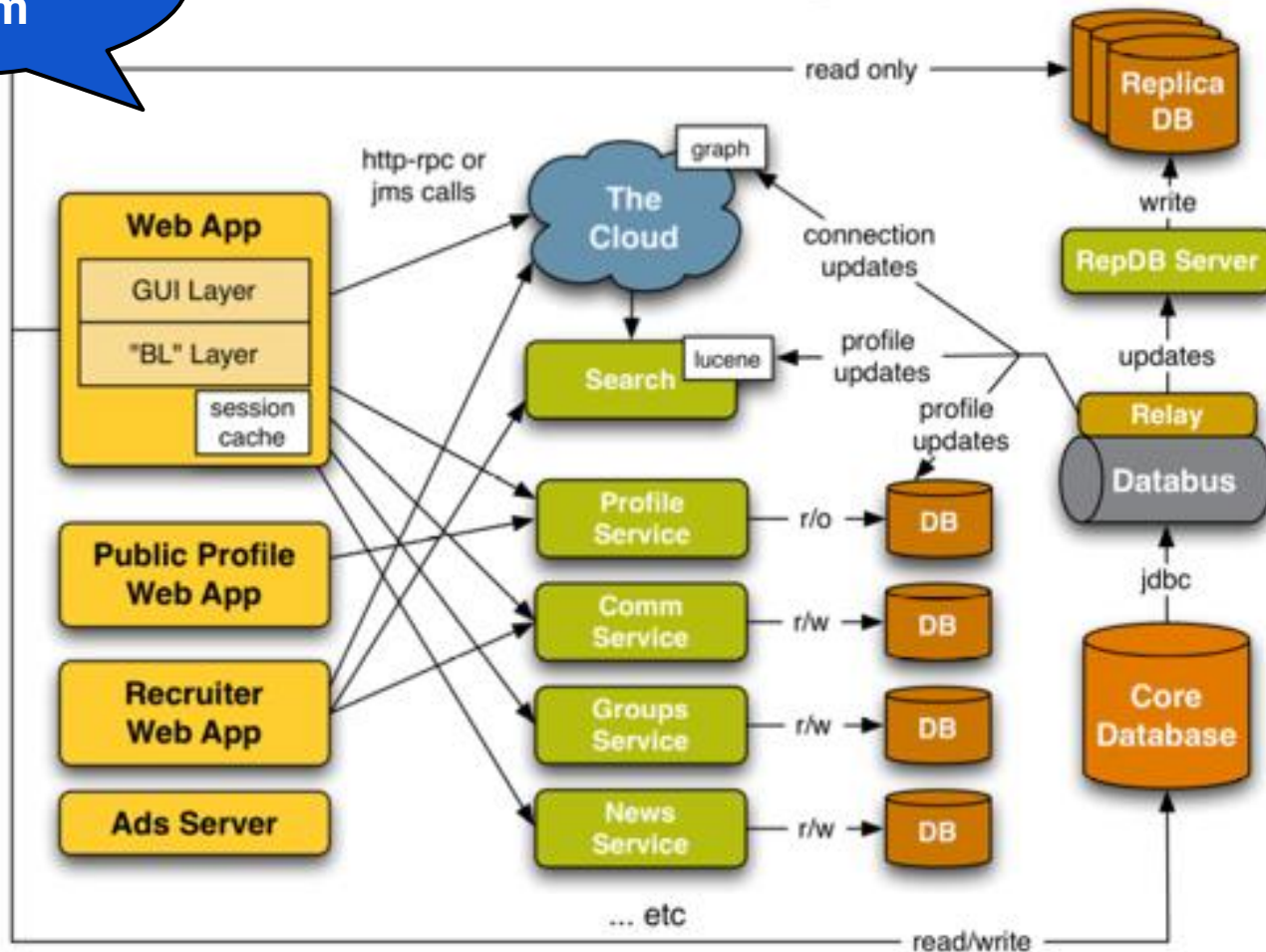
Architectural patterns

- The architecture is at a higher level than the design. So design patterns are not considered in this phase
- Analogous to the design patterns, there are **architectural patterns**



Architectural patterns

Architectural diagram

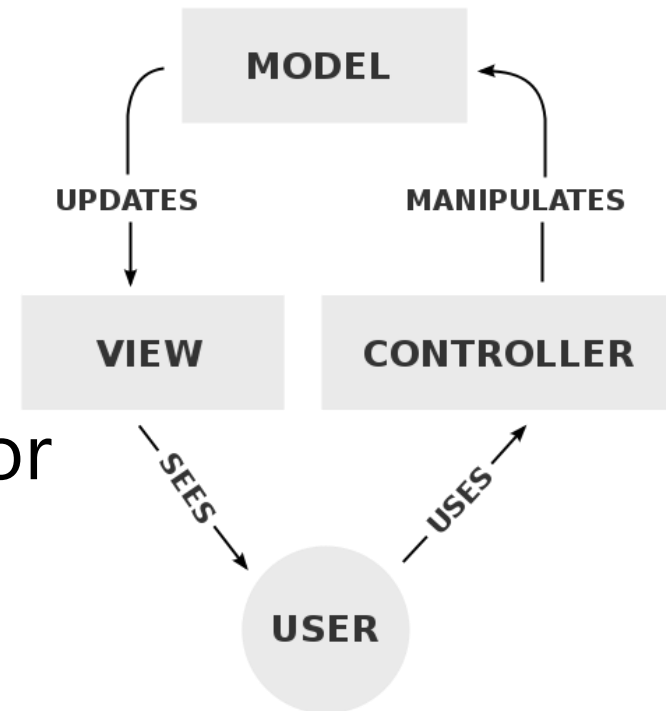


Model-View-Controller (MVC)

→ MVC is one of the most common architectural patterns for implementing user interfaces

→ Divides the system in three separated parts

→ It has become an standard for web development



Model-View-Controller (MVC)

→ **Controller** sends commands to the model to update the model's state. It can also send commands to the view to update the representation of the model

Contains the main logic of the application and the flow of the user interaction

Model-View-Controller (MVC)

→ **Model** stores data that is retrieved to the controller and displayed in the view. When a change in the data occurs, the controller updates the model

Model usually represent the database tables or any other source of persistent data

Model-View-Controller (MVC)

→ **View** request information from the model that it uses to generate an output representation to the user

For example, a view in a web application is an HTML file that the user interacts with



Class Diagrams & Design Patterns

CE-1103 Algorithms and Data Structures I

ВЕДОМОСТЬ ПРЕСЛОВ		№	размер послово в.ч. мм
		позначит	1510 x 2070
		③	910 x 2070
		⑩	
		⑤	910 x 1510
		⑤	2310 x 2110
		②	2170