# UML, Patterns and Software Quality

Algorithms and Data Structures

## 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.

## Think about it...

- ➔ 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?

## Think about it...

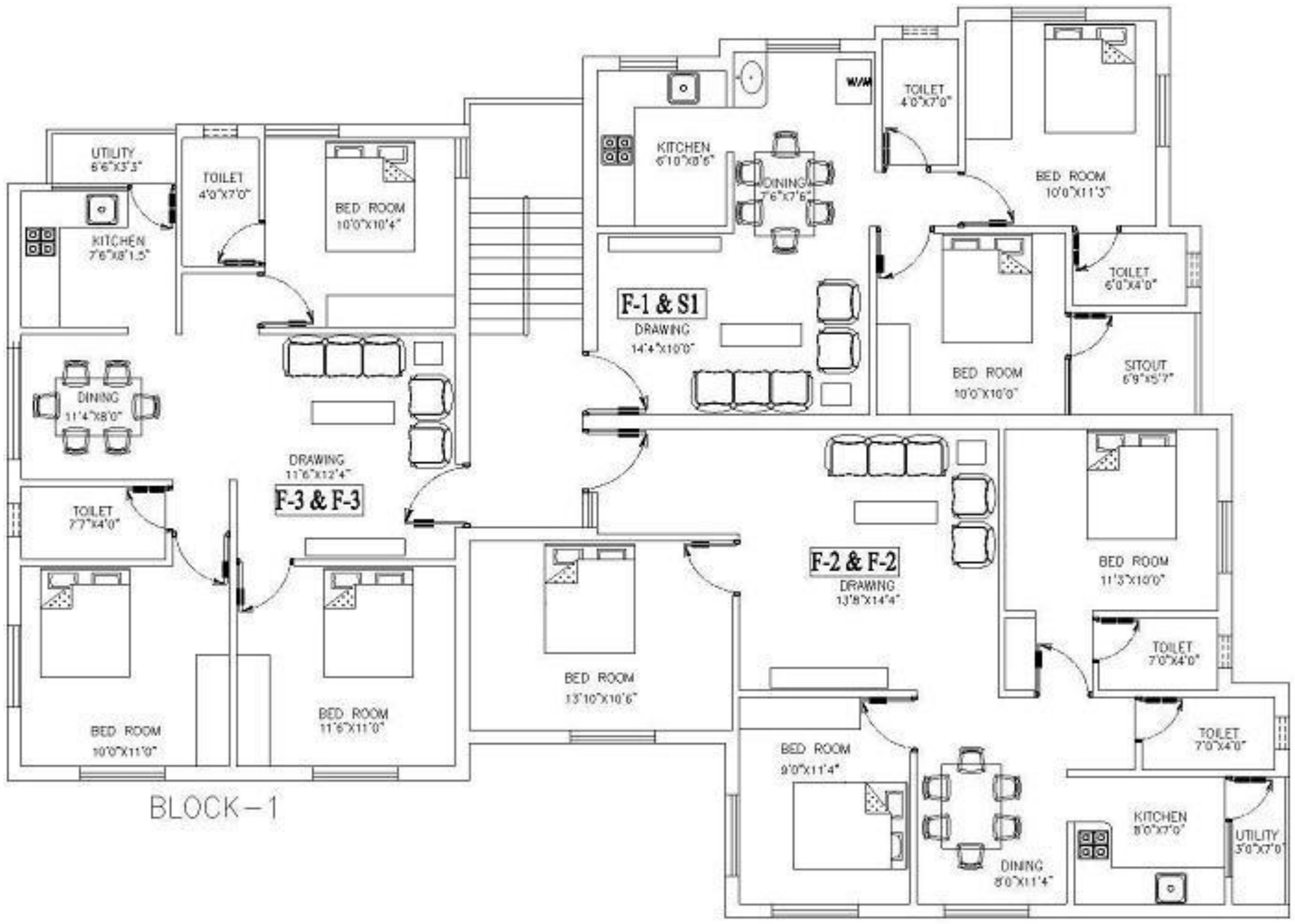
- ➔ Most people goes direct to code!
  - ◆ Too much design and thinking is for cowards! - they say
  - ◆ Kids code apps in their bedrooms!

- ➔ 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 ours 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:
  - ◆ UML
  - ◆ OOP
  - ◆ Design Patterns
  - ◆ Software quality

- ➔ Take a guess, can you explain each of these items?

## Before going into details...

➔ UML has a tight relationship with the object-oriented paradigm

➔ Let's review some OOP concepts first

# Object-Oriented Paradigm

Revisiting some definitions

## What is an object?

➔ An object packages **data** and procedures that operate on the data

➔ Procedures are called **methods** and operates on the data of the object. Data is called **attributes**.

➔ The only way to interact with an object is through a request or **message** from a client.

## What is the interface of an object?

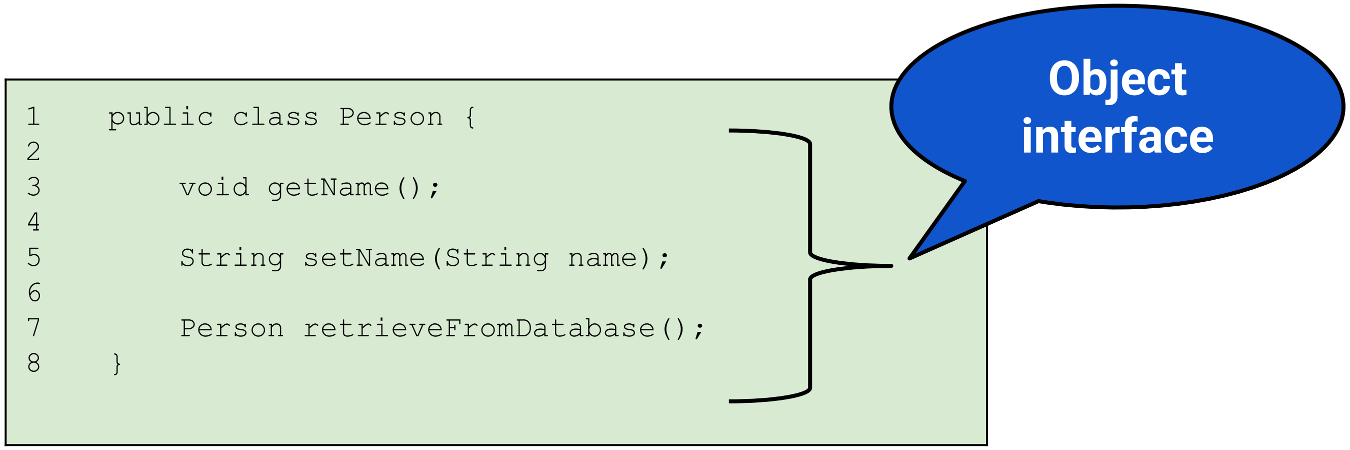
- ➔ The internal state of an object is **encapsulated**: it cannot be accessed directly
- ➔ A method name, parameters and return value is called the **signature** of the method.
- ➔ The set of all signatures is the **interface of the object**

## What is the interface of an object?

```
public class Person {

    void getName();

    String setName(String name);

    Person retrieveFromDatabase();
}
```

Signature

# What is the interface of an object?



```
public class Person {

    void getName();

    String setName(String name);

    Person retrieveFromDatabase();
}
```

## What is the class of an object?

- ➔ An object has a **class**. A class defines the implementation of an object
  - ◆ Specifies the attributes and interface of an object
- ➔ Objects are created by **instantiating** a class.
- ➔ When an **instance** is created, it allocates the memory of the object and associates the methods to this memory.

## What is inheritance?

➔ Is a way of create a new class **based on an existing class**.

➔ A subclass extends a parent class, including all the definitions of the data and operations defined by the parent.

➔ A subclass can access **most** of the data and operations defined by its parent

## What is inheritance?

➔ Is a way of create a new class **based on an existing class**.

➔ A subclass extends a parent class, including all the definitions of the data and operations defined by the parten

**Except private data / operations**

➔ A subclass can access **most** of the data and operations defined by its parent

## What is an abstract class?

➔ Is a class whose main purpose is to d**efine a common interface** for its subclasses

➔ It cannot be instantiated, only extended. The subclass of an abstract is called **concrete class**

➔ An abstract class can implement some operations or not. The non-implemented are called **abstract methods**

## Overriding and overloading

- ➔ A subclass can **override a method** of the parent class. This means, replace the original method of its parent
  - ◆ Intercept requests instead of relying on its parent

- ➔ **Overloading** means that a class can have more than one method with the same name but with different parameters or return value.

## Polymorphism

- ➔ Extending a class allows to override a method of the parent class

- ➔ When a request for a method is issued, the association between the object and the method is done at **run-time**
  - ◆ This is called dynamic binding

## Polymorphism

➔ Dynamic binding allows to substitute objects that have **identical interfaces** for each other at run time

➔ This substitutability is called **polymorphism**

## Polymorphism

➔ Dynamic binding allows to substitute objects that have **identical interfaces** for each other at run time

**Key concept in OOP**

➔ This substitutability is called **polymorphism**

## What is object composition?

➔ You can reuse an object with two different approaches:
  - ◆ Inheritance
  - ◆ Composition

➔ In inheritance you can define an object in terms of another

➔ In composition, you can reuse functionality by **assembling** objects together

## What is object composition?

➔ You can reuse an object with two different approaches:
- ◆ Inheritance
- ◆ Composition

➔ In inheritance you can define an object in terms of an

Objects instances as attributes of another

➔ In composition, you can reuse functionality by **assembling** objects together

## Coupling and Cohesion

- ➔ **Coupling** is the degree of interdependence between two modules / classes of a system
  - ◆ If one class uses many methods of another **concrete** class, both classes are tightly coupled
  - ◆ If one class uses just a few methods of another **concrete** class, coupling lows
  - ◆ If the class uses methods of an interface and not uses the concrete object directly, coupling is loose.

## Coupling and Cohesion

- **Cohesion** is the degree to which elements of a module/class belong together
  - Methods inside a class have a lot in common, functionally and conceptually speaking
  - Methods are small and carry specific tasks, and are used to the inside of the class
- Coupling should be low and cohesion should be high
  - Signs of a good design

## 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**

➔ You will learn more about this in future courses. We will see just an overview

Functional model:

## Use cases diagram

- ➔ An use case diagram captures the **behavior of a system, subsystem, class or component** as it appears to an outside user

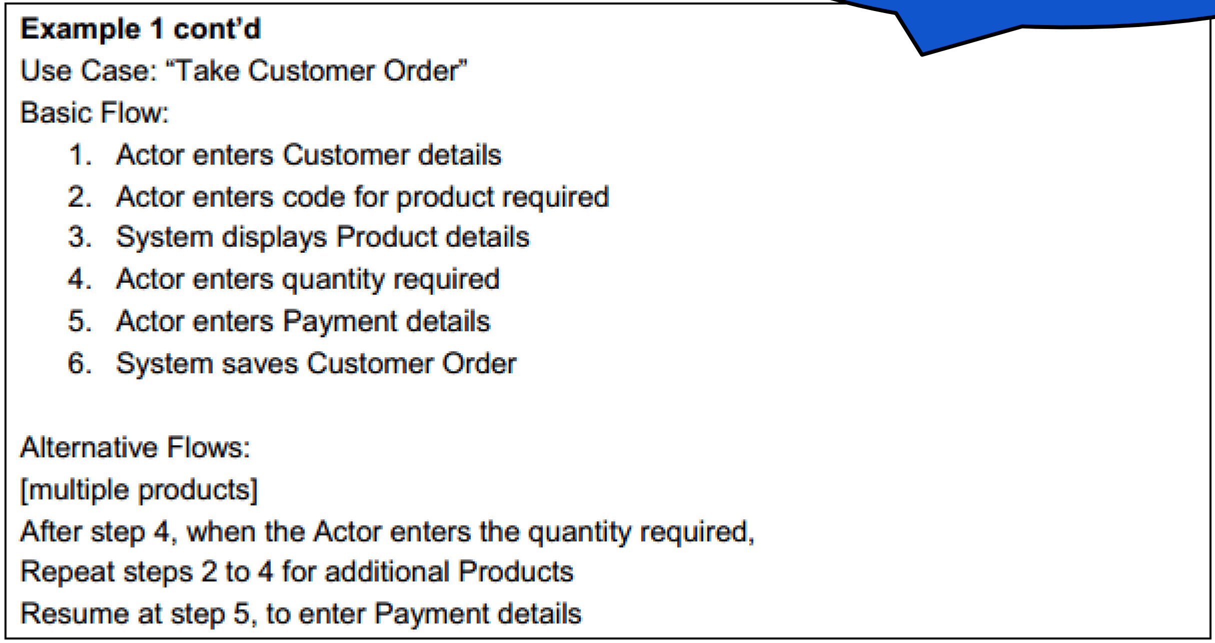
- ➔ It partitions the system into **transactions** meaningful to actors

- ➔ An **use case** is a coherent unit of functionality expressed in terms of messages exchange by the system and the actors
  - ◆ Logical description of a slice of functionality

Functional model:

## Use cases diagram

This is a use case specification



**Example 1 cont'd**

Use Case: "Take Customer Order"

Basic Flow:

1. Actor enters Customer details
2. Actor enters code for product required
3. System displays Product details
4. Actor enters quantity required
5. Actor enters Payment details
6. System saves Customer Order

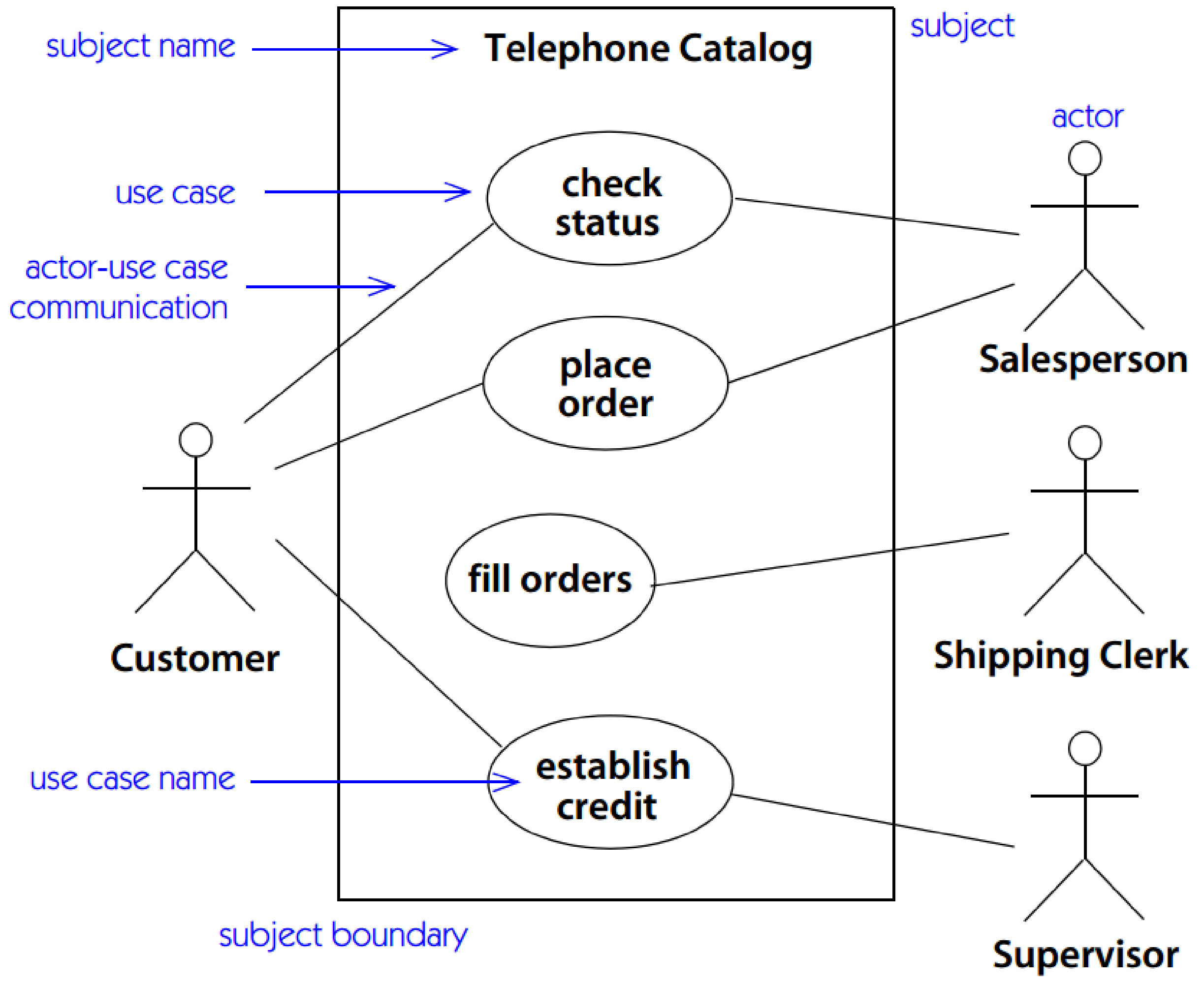
Alternative Flows:

[multiple products]

After step 4, when the Actor enters the quantity required,
Repeat steps 2 to 4 for additional Products
Resume at step 5, to enter Payment details

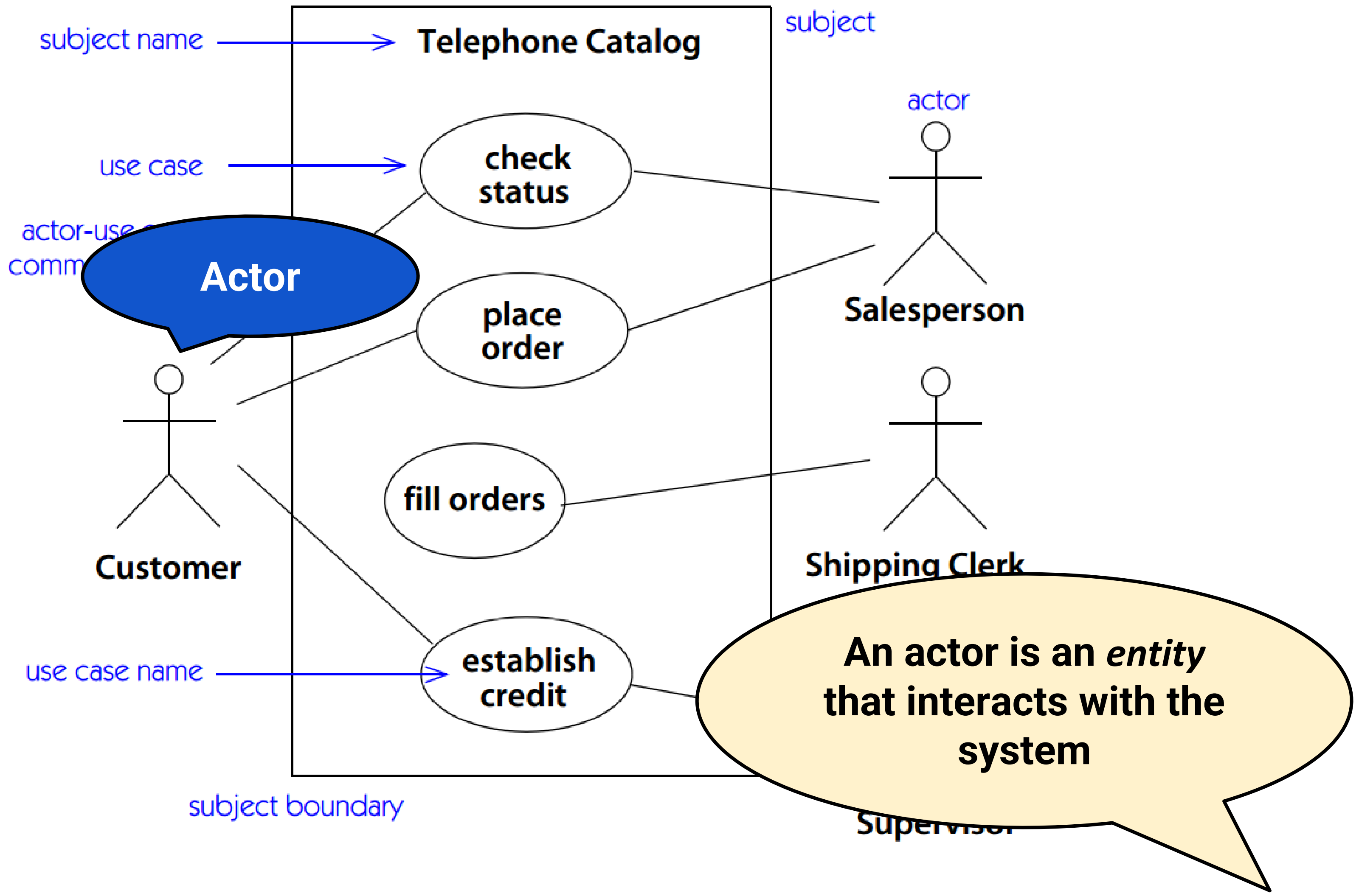
Functional model:

## Use cases diagram



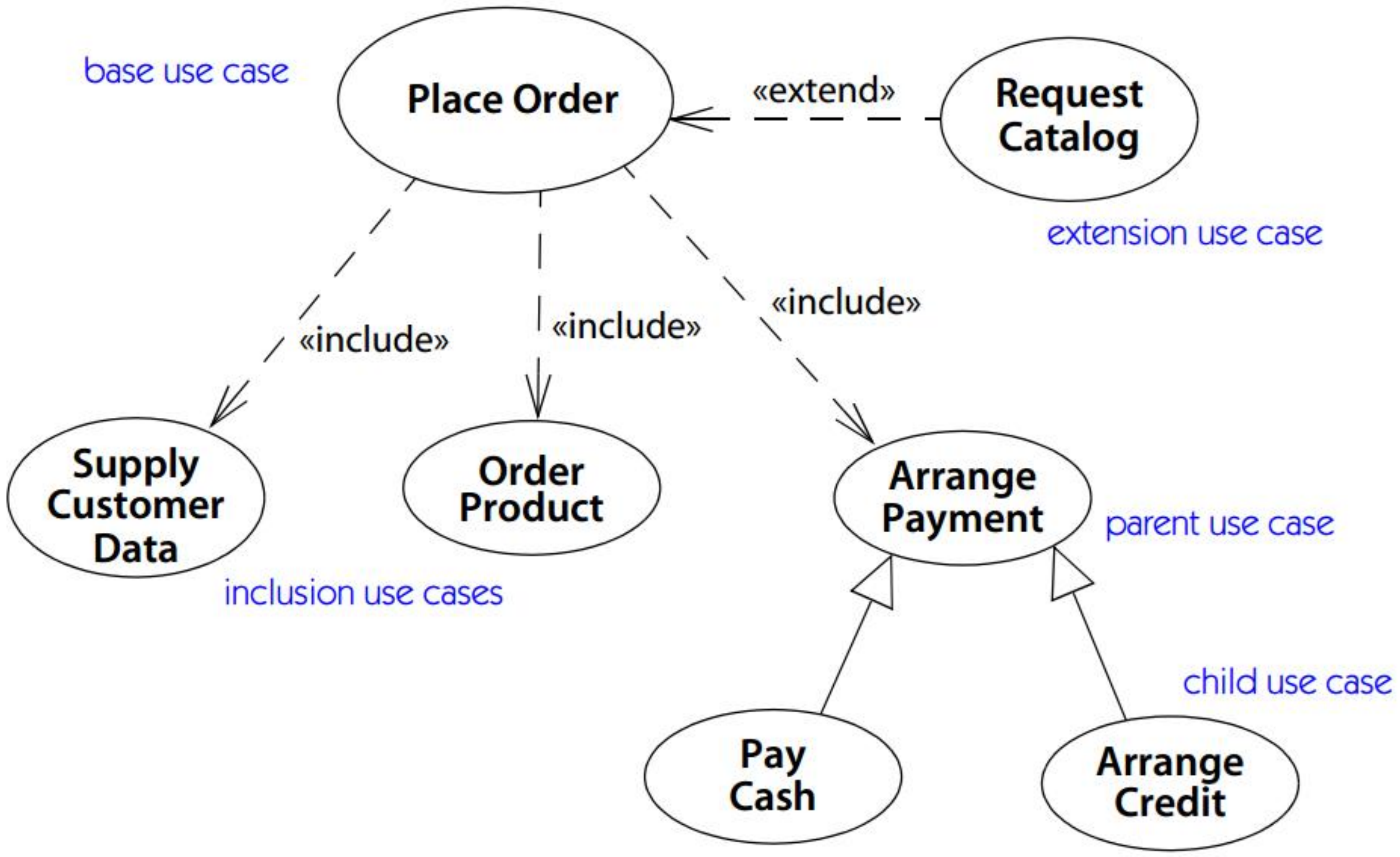
Functional model:

## Use cases diagram



Functional model:

## Use cases diagram



Functional model:

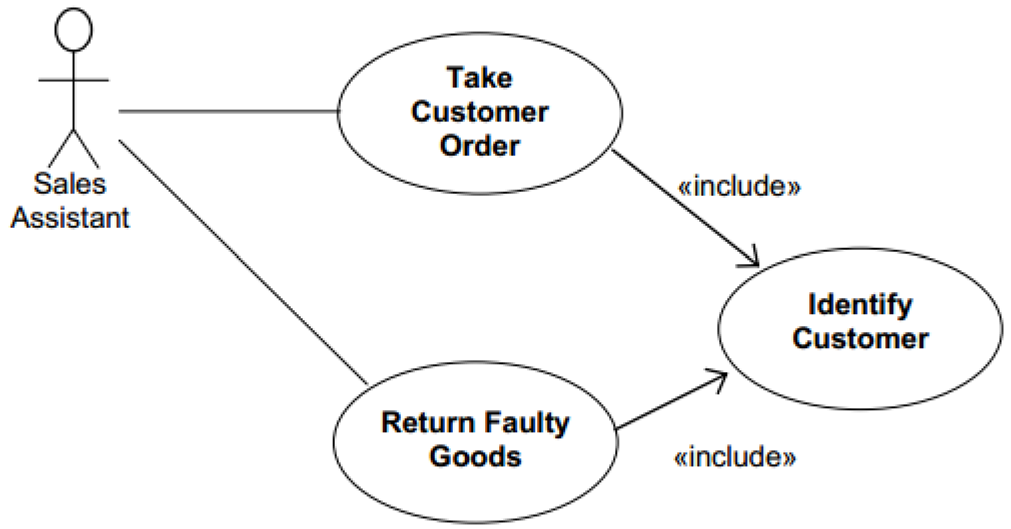
## Use cases diagram

| *Relationship* | *Function* | *Notation* |
|---|---|---|
| association | The communication path between an actor and a use case that it participates in | ——— |
| extend | The insertion of additional behavior into a base use case that does not know about it | «extend» - - - -> |
| include | The insertion of additional behavior into a base use case that explicitly describes the insertion | «include» - - - -> |
| use case generali-zation | A relationship between a general use case and a more specific use case that inherits and adds features to it | ——▷ |

Functional model:

## Use cases diagram: include

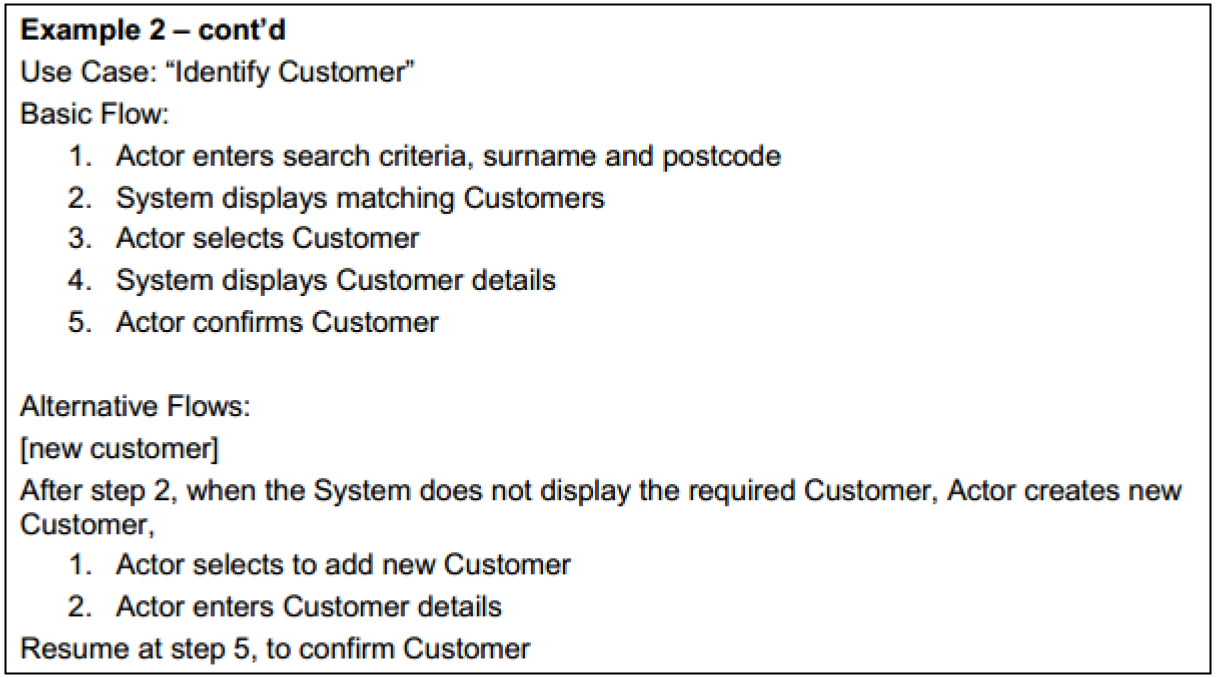
➔ Includes the steps from one use case into another



Functional model:

## Use cases diagram: include

→ Includes the steps from one use case into another



**Example 2 – cont'd**

Use Case: "Identify Customer"

Basic Flow:

1. Actor enters search criteria, surname and postcode
2. System displays matching Customers
3. Actor selects Customer
4. System displays Customer details
5. Actor confirms Customer

Alternative Flows:

[new customer]

After step 2, when the System does not display the required Customer, Actor creates new Customer,

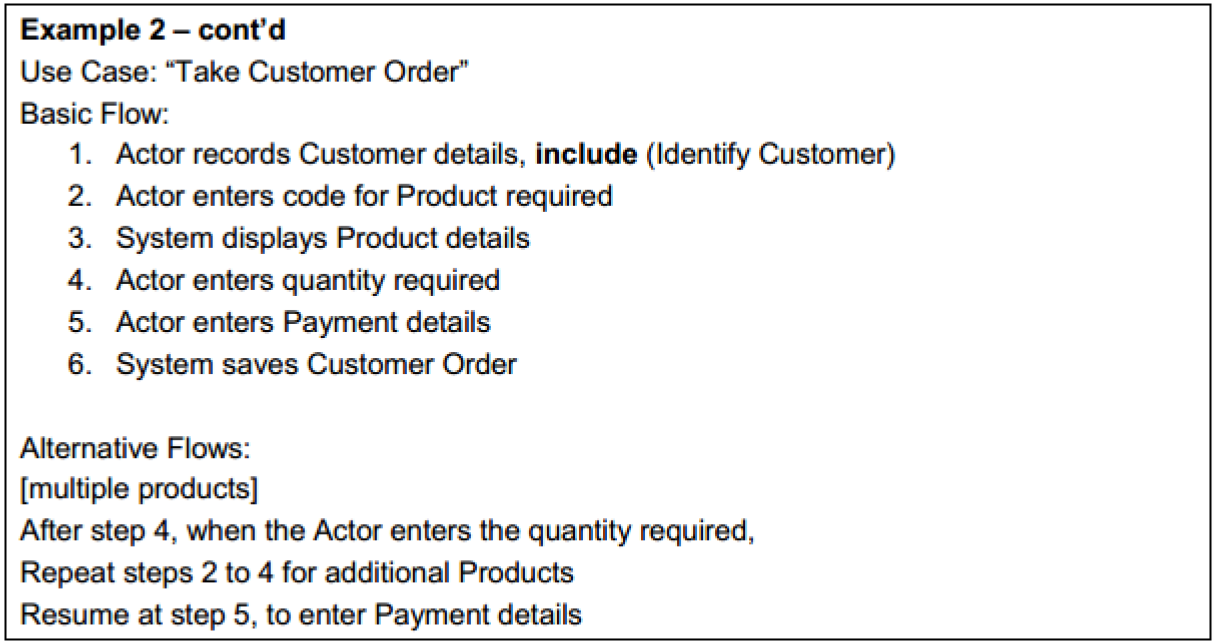
1. Actor selects to add new Customer
2. Actor enters Customer details

Resume at step 5, to confirm Customer

Functional model:

## Use cases diagram: include

➔ Includes the steps from one use case into another



**Example 2 – cont'd**
Use Case: "Take Customer Order"
Basic Flow:

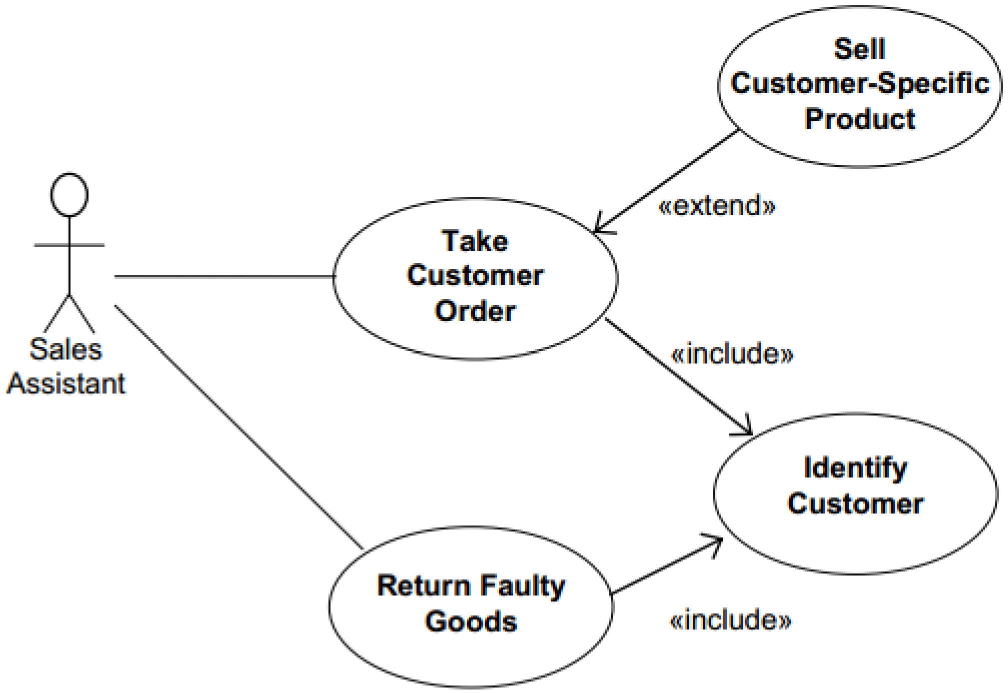
1. Actor records Customer details, **include** (Identify Customer)
2. Actor enters code for Product required
3. System displays Product details
4. Actor enters quantity required
5. Actor enters Payment details
6. System saves Customer Order

Alternative Flows:
[multiple products]
After step 4, when the Actor enters the quantity required,
Repeat steps 2 to 4 for additional Products
Resume at step 5, to enter Payment details

Functional model:

## Use cases diagram: extends

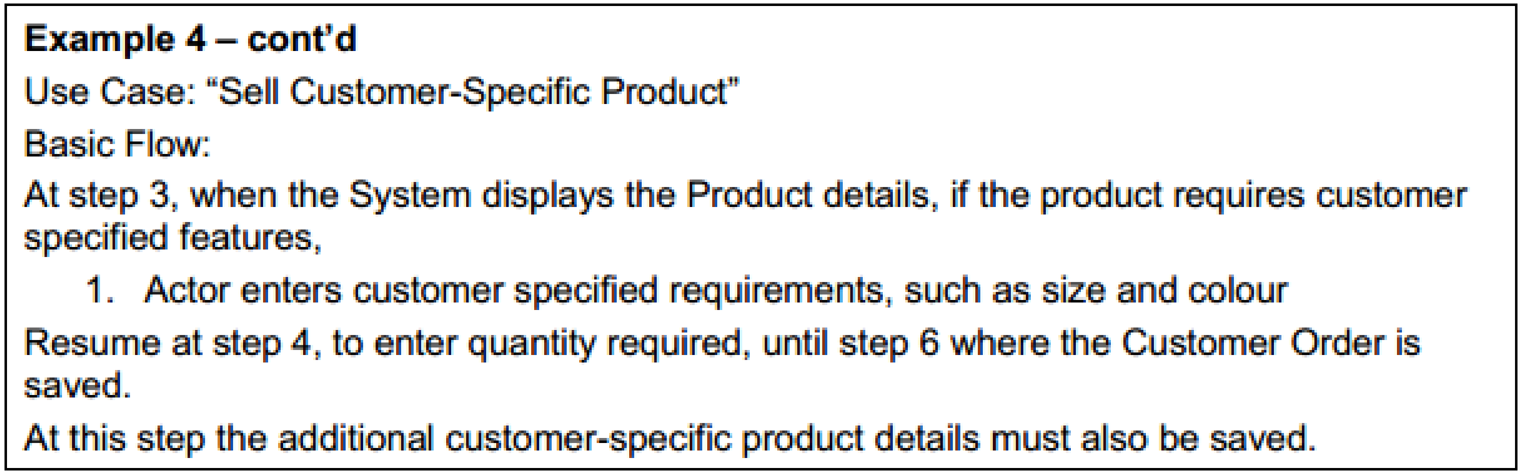
➔ Allows to modify the behavior of the base use case



Functional model:

## Use cases diagram: extends

➔ Allows to modify the behavior of the base use case



**Example 4 – cont'd**

Use Case: "Sell Customer-Specific Product"

Basic Flow:

At step 3, when the System displays the Product details, if the product requires customer specified features,

1. Actor enters customer specified requirements, such as size and colour

Resume at step 4, to enter quantity required, until step 6 where the Customer Order is saved.

At this step the additional customer-specific product details must also be saved.

Functional model:

## Use cases diagram: generalization

- ➔ Allows to replace the behavior of the base use case.
- ➔ Is more a theoretical relationship
- ➔ Many people finds hard to understand
- ➔ For now keep the include and extends in mind

Object Model:

## 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

Object Model:

## Class diagram

➔ Also known as **static view**

➔ This diagram captures the object structure. Includes all the data structures and the operations on the data

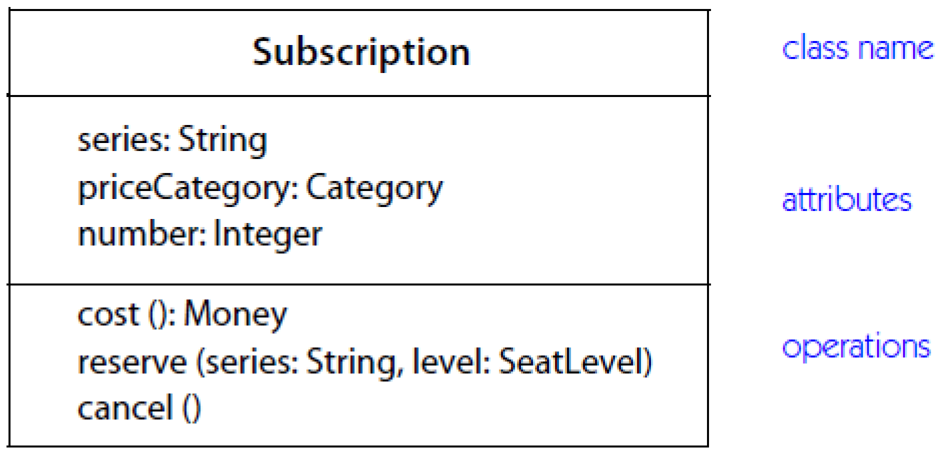
➔ It does not contains any information related to the dynamic behavior

Run time

Object Model:

## Class diagram

- ➔ A class is represented as:



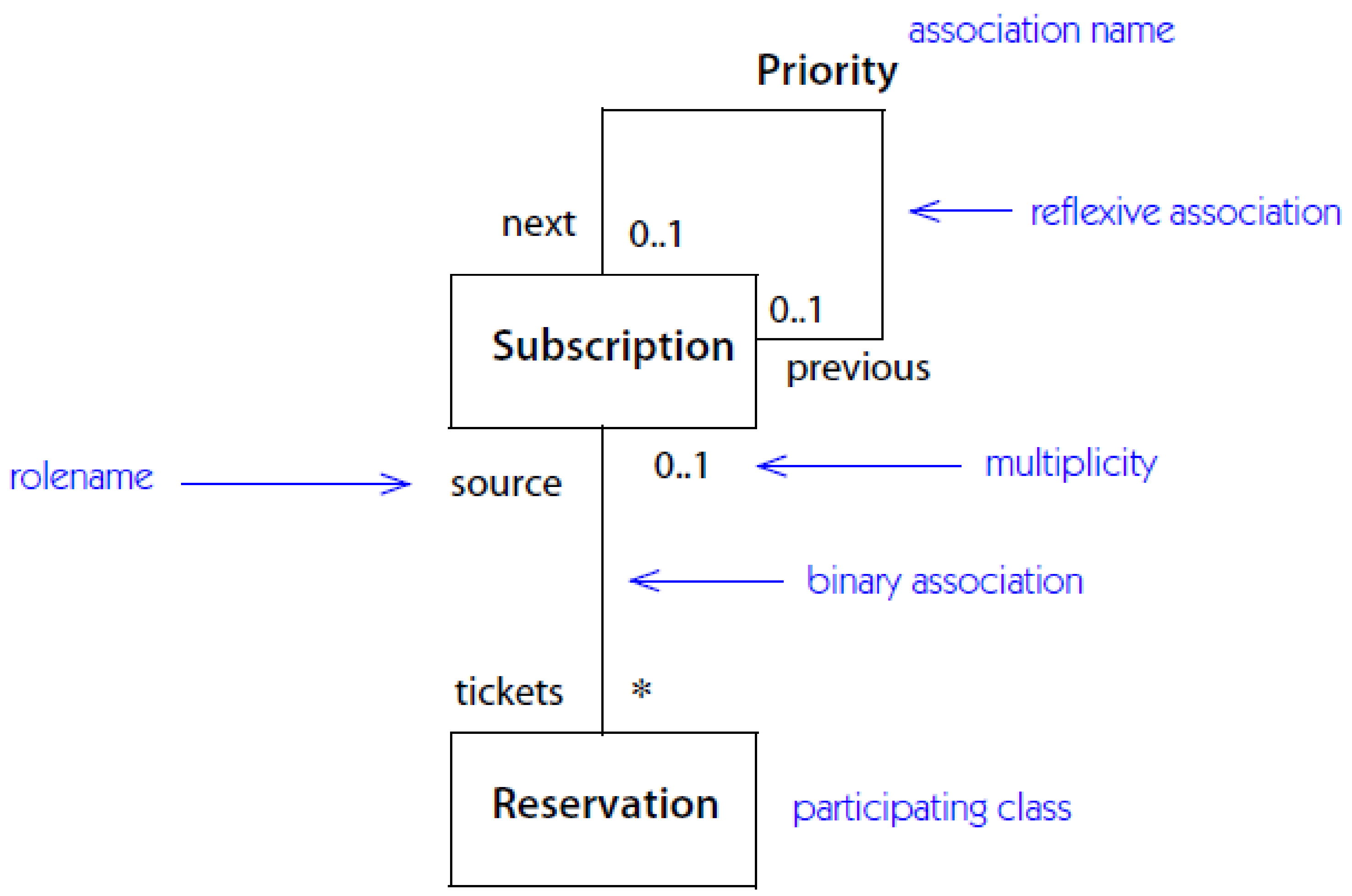
Object Model:

## Class diagram

| *Relationship* | *Function* | *Notation* |
|---|---|---|
| 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 | «kind» - - - -> |

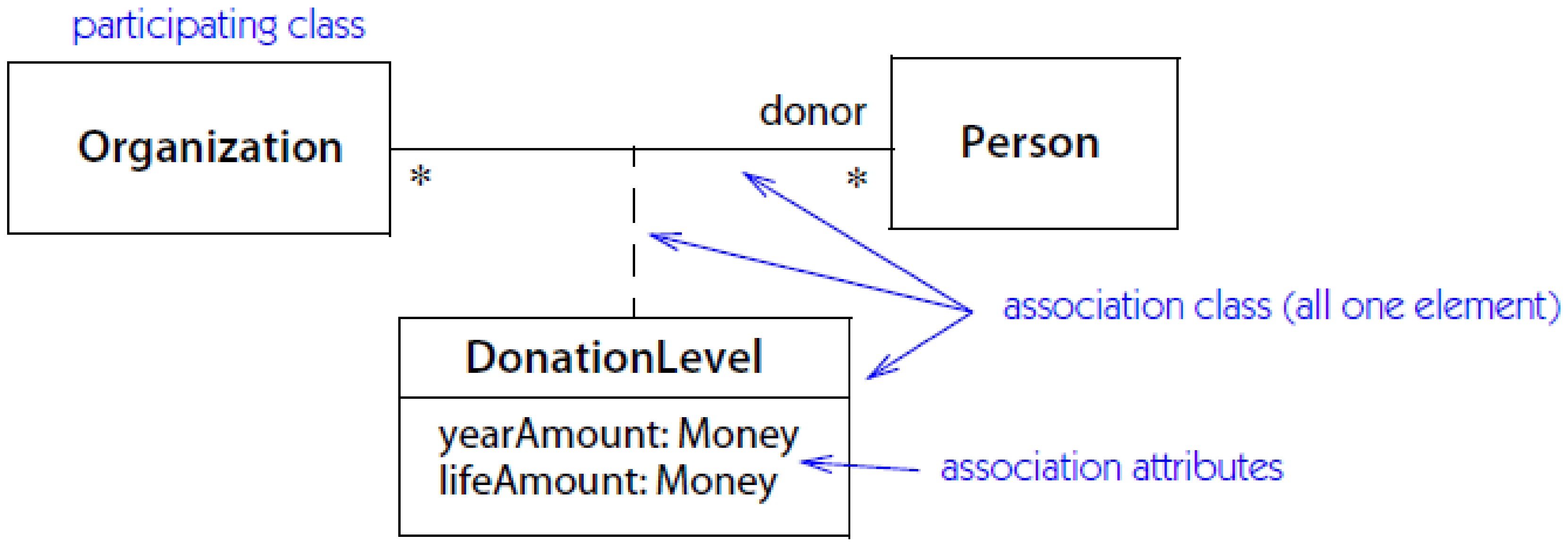
Object Model:

## Class diagram: Association



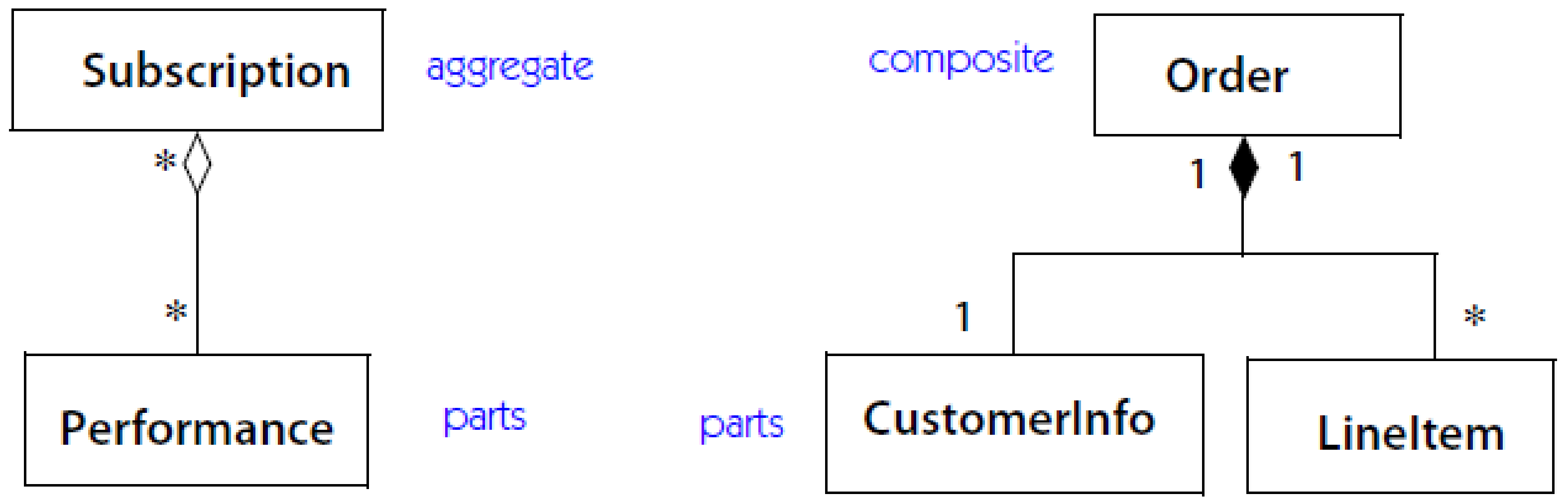
Object Model:

## Class diagram: Association



Object Model:

## Class diagram: Association



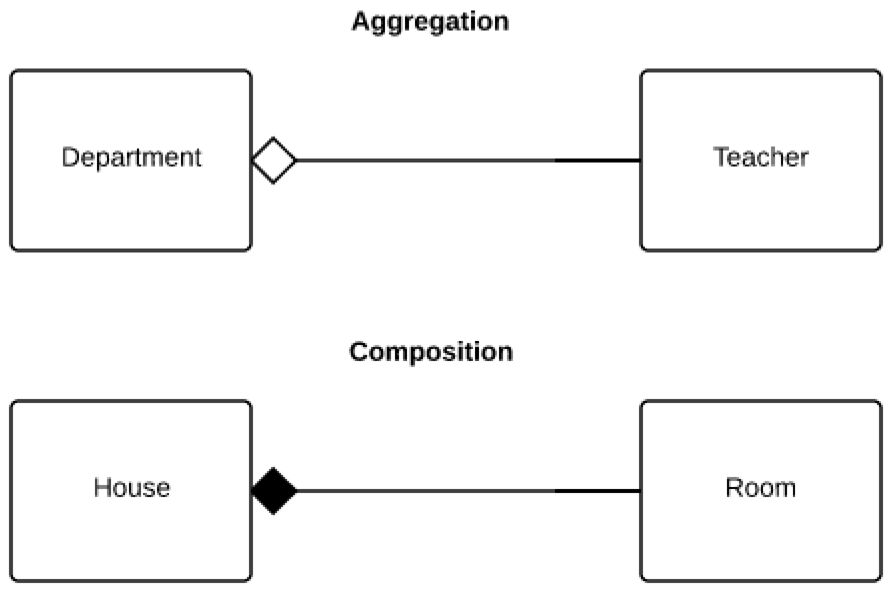
Object Model:

## Class diagram: Association

- ➔ What is the difference between aggregate and composite?
- ➔ Depends if the enclosed object is stand alone or not.
  - ◆ If the object is stand alone, it is an aggregate
  - ◆ If the object depends on the container to live, is composite

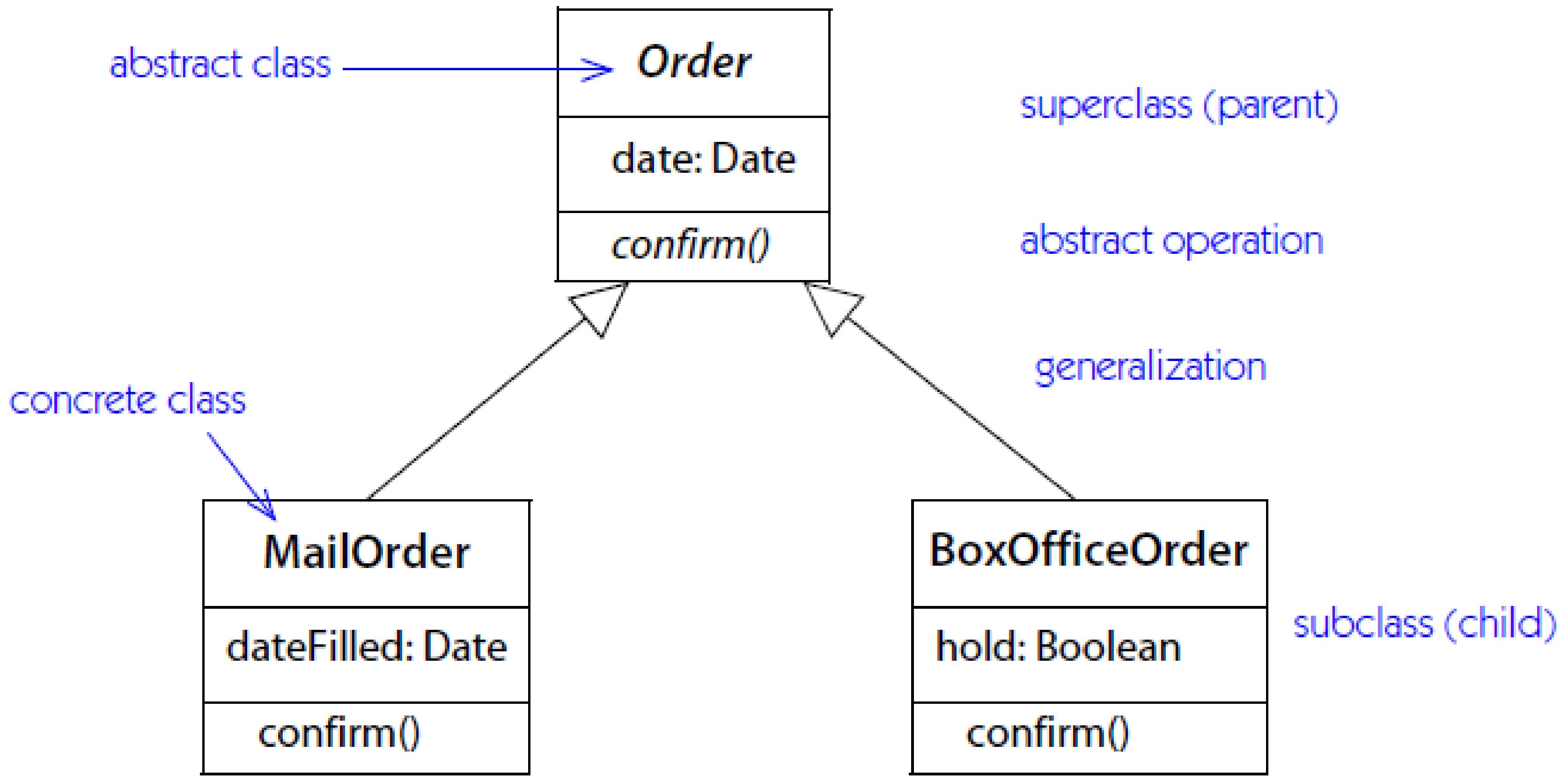
Object Model:

## Class diagram: Association



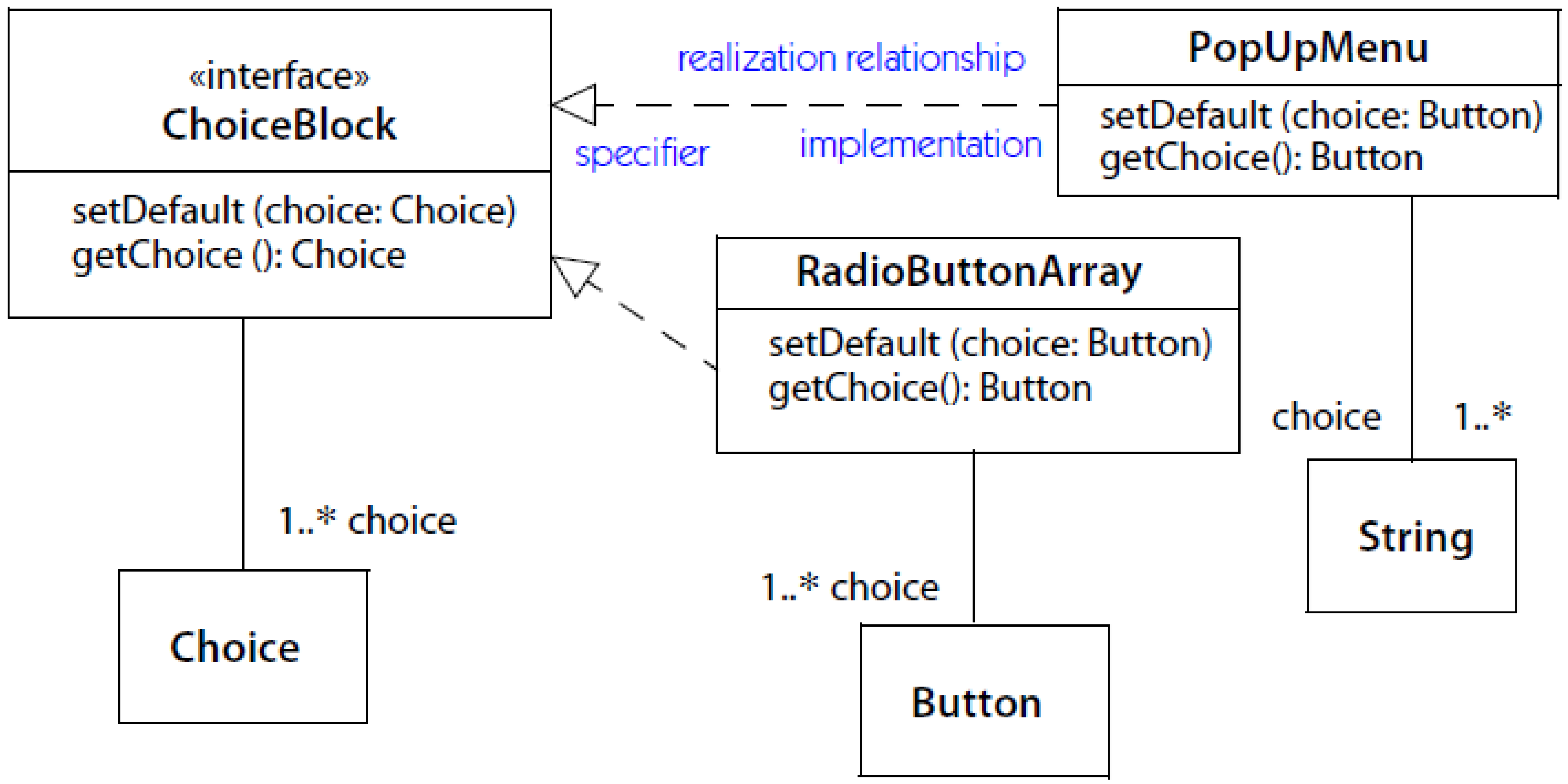
Object Model:

## Class diagram: Generalization



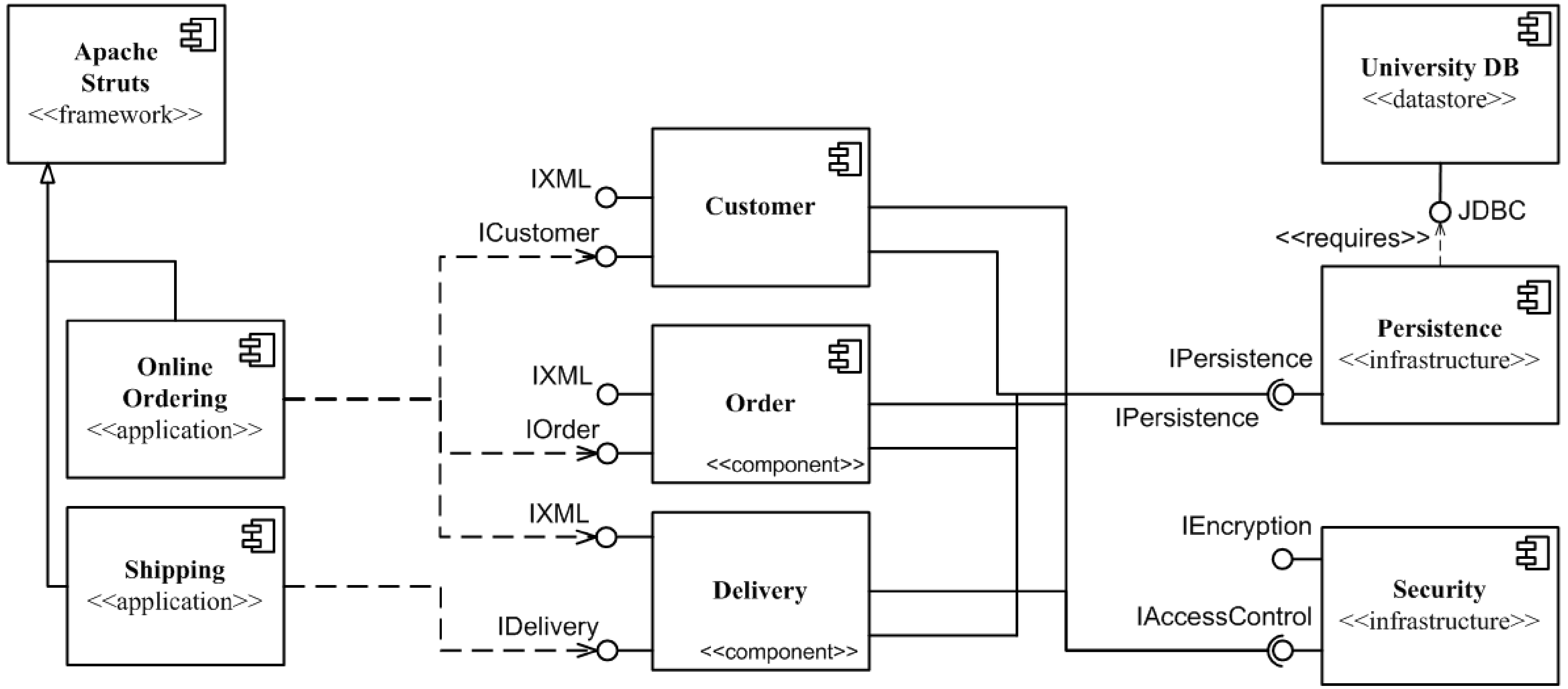
Object Model:

# Class diagram: Realization



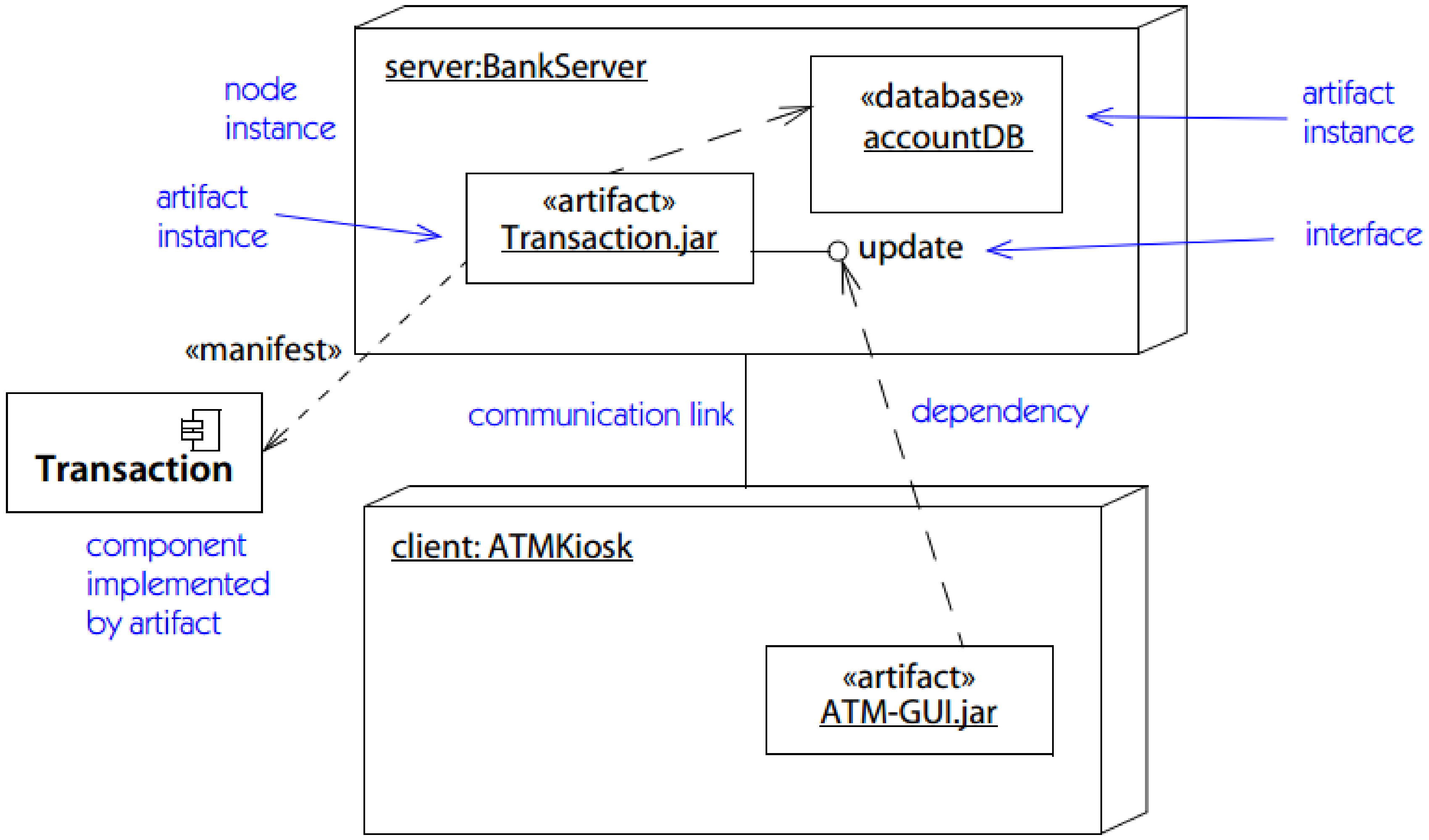
Object Model:

## Component diagram



Object Model:

## Deployment diagram



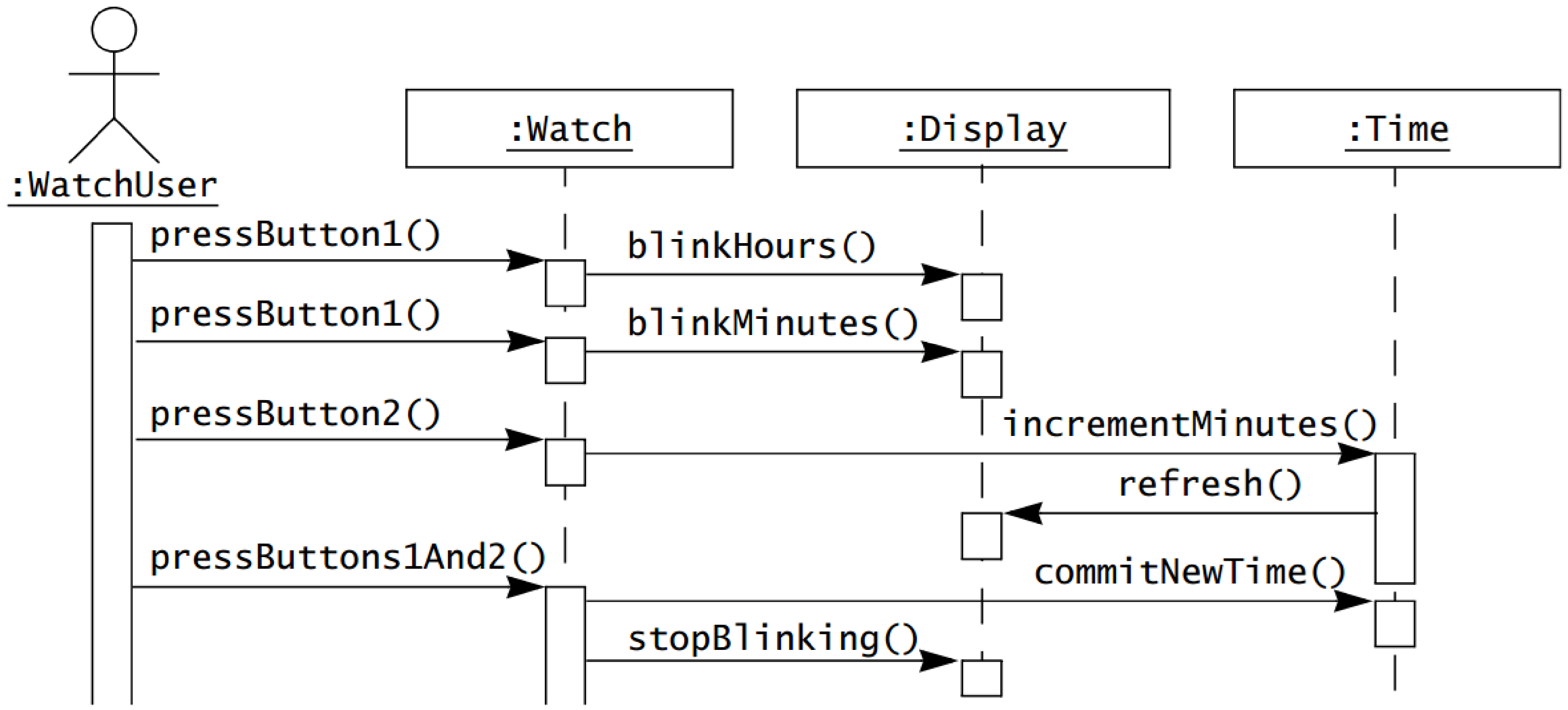
Dynamic Model:

## Sequence diagram

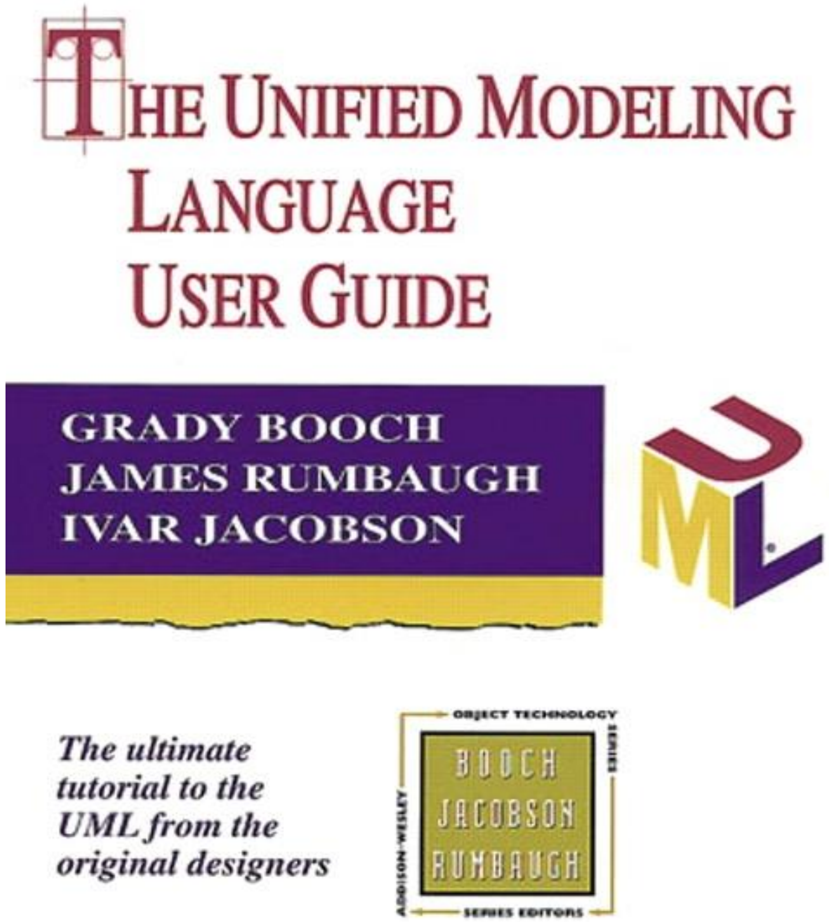
➔ Sequence diagrams are used to formalize the **dynamic behavior of the system** and to visualize the object communication

Dynamic Model:

## Sequence diagram



## Further readings…


THE UNIFIED MODELING LANGUAGE USER GUIDE

GRADY BOOCH
JAMES RUMBAUGH
IVAR JACOBSON

*The ultimate tutorial to the UML from the original designers*

## The message is...

➔ Be an engineer! Avoid staying in the empirical side of things

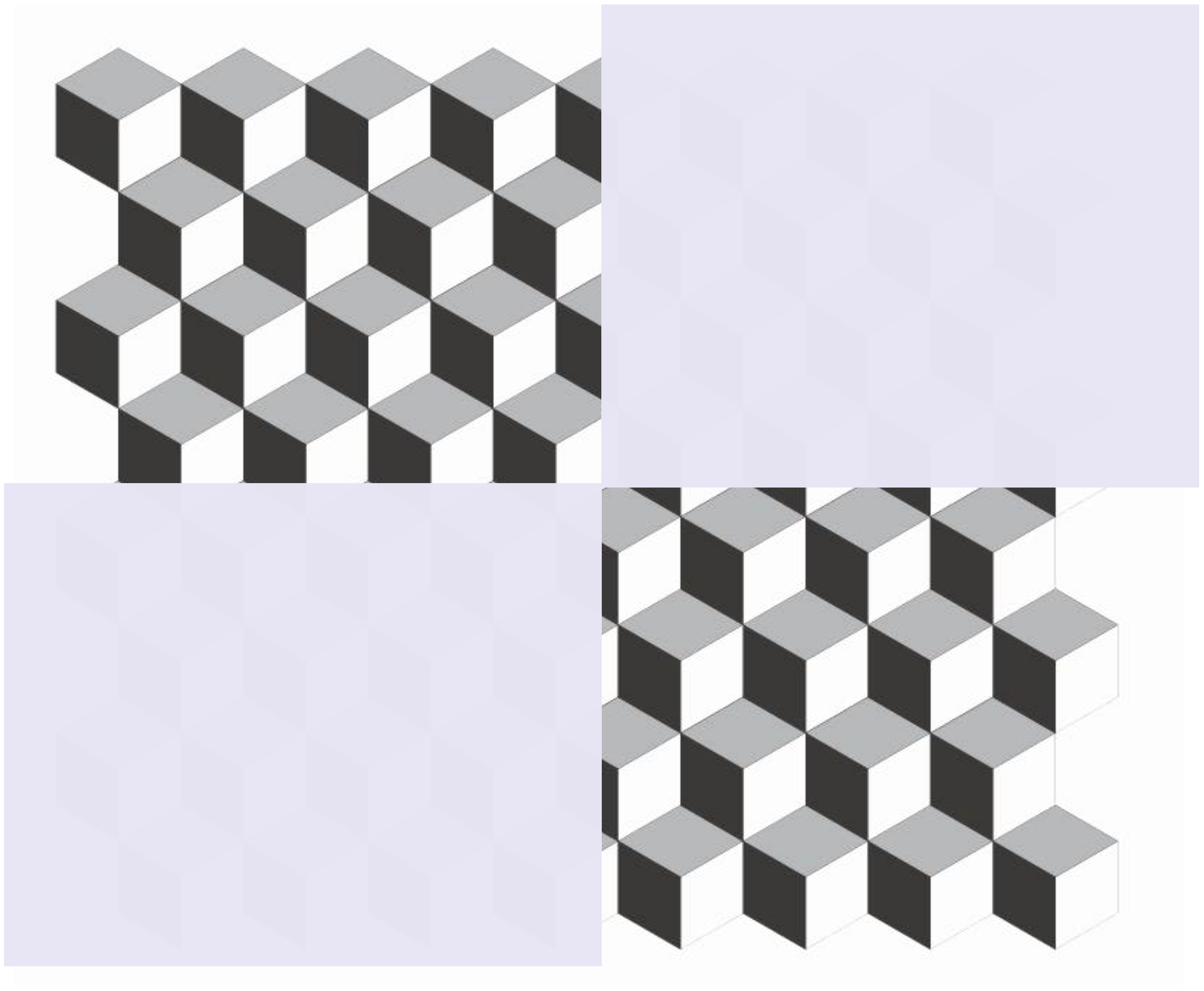
➔ Model the problem before starting to code!

➔ Think!

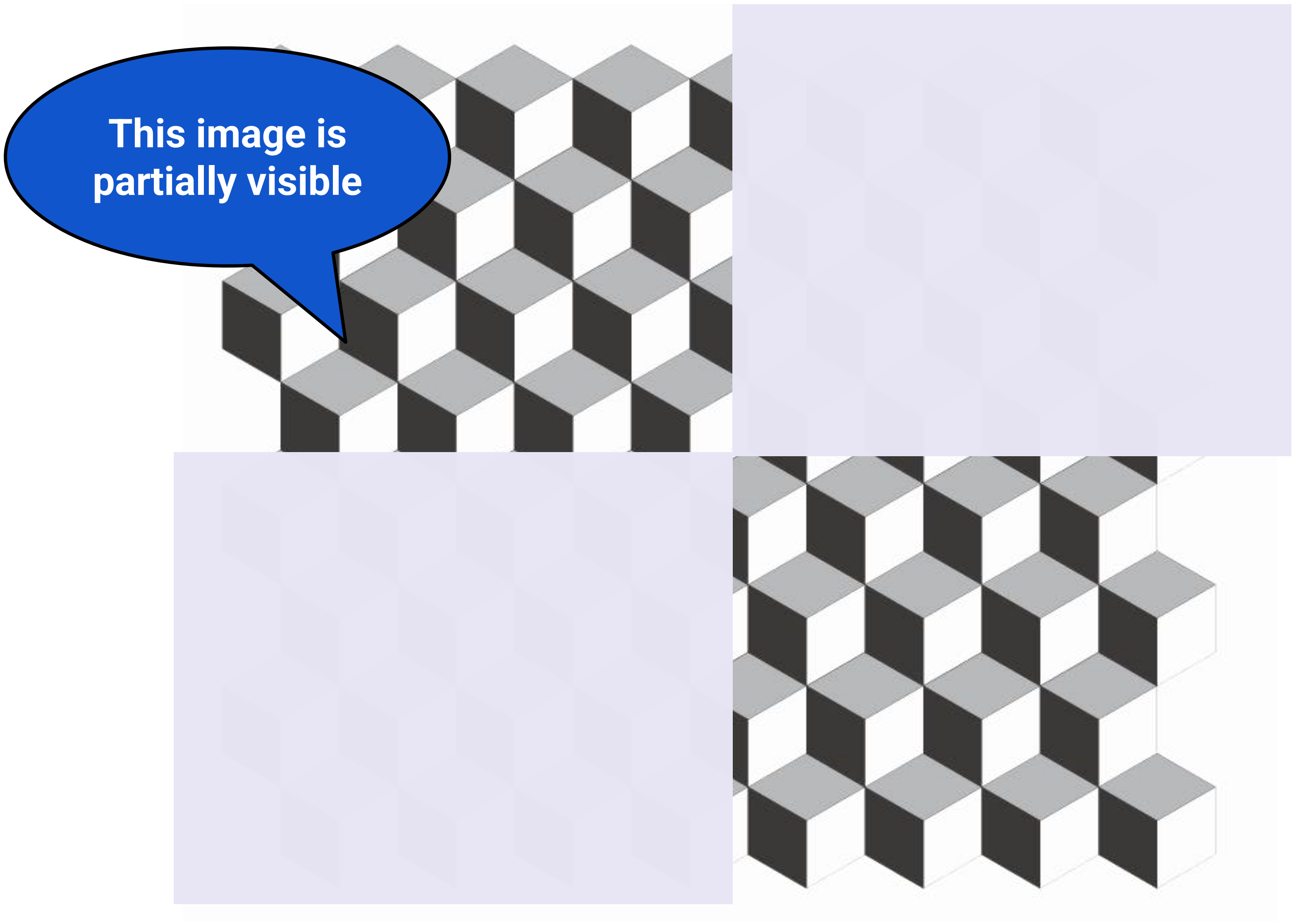
➔ Be formal!

## Design Patterns

## What is a pattern?

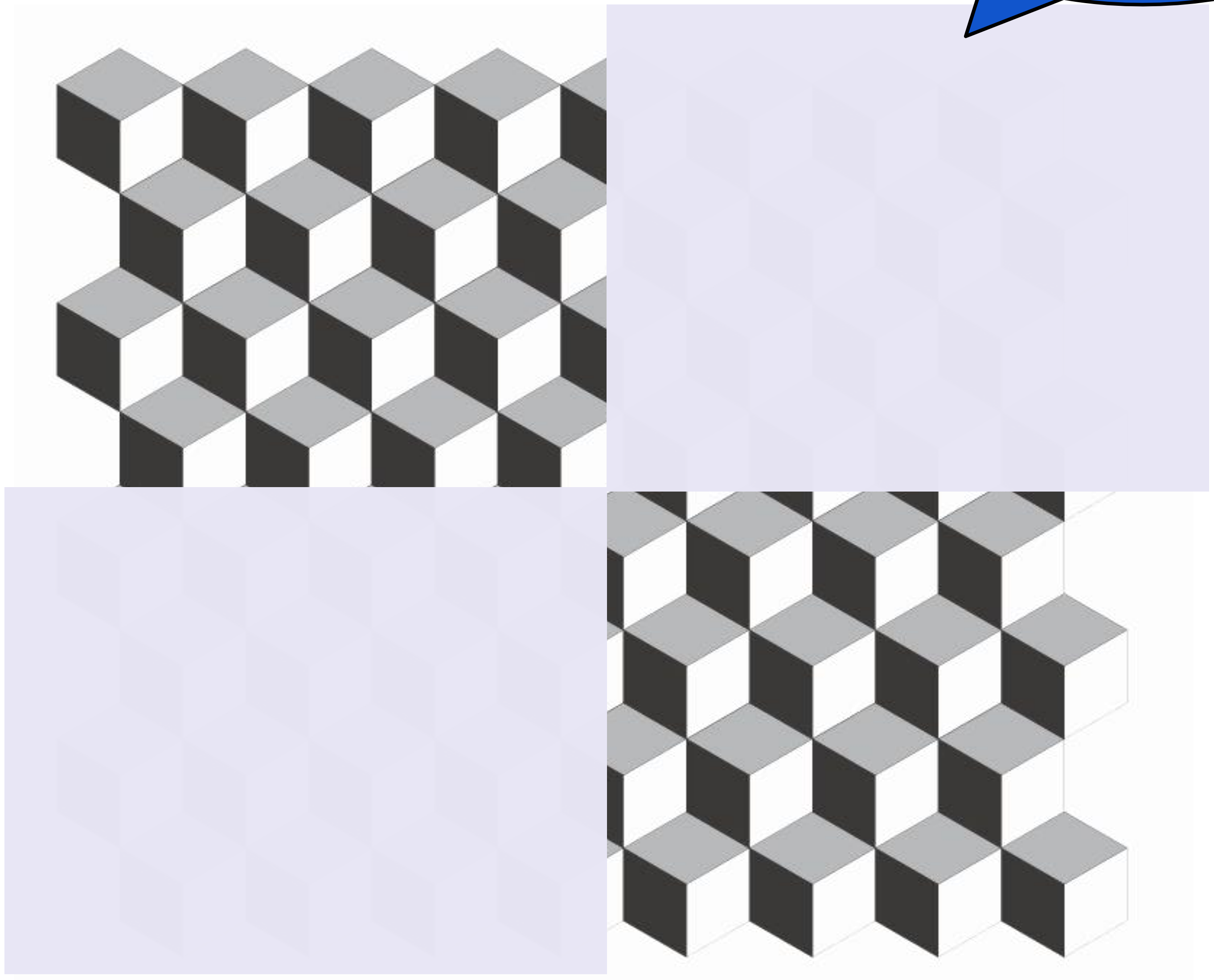


## What is a pattern?

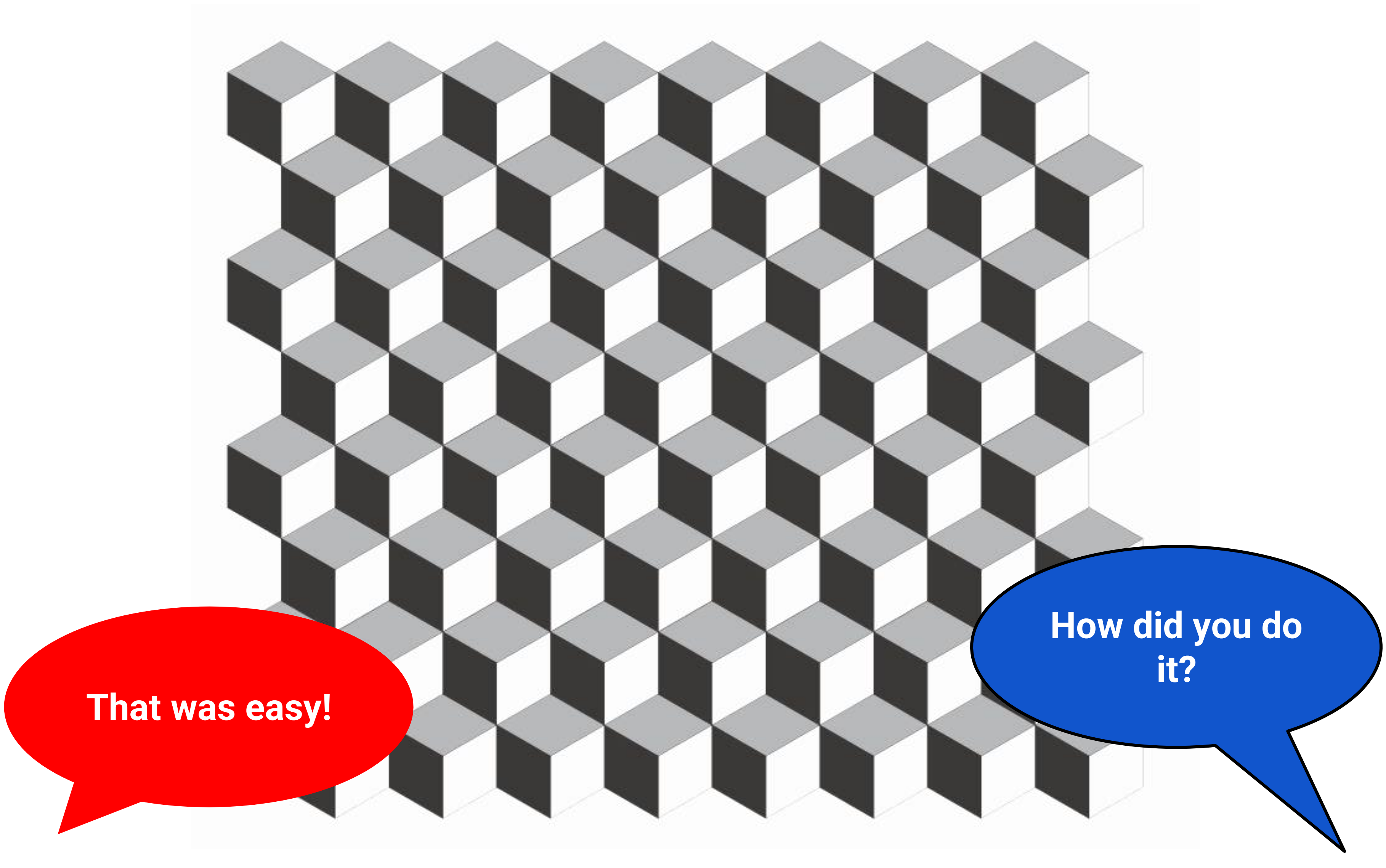


## What is a pattern?

Can you tell what is behind the squares?



## What is a pattern?



## 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?

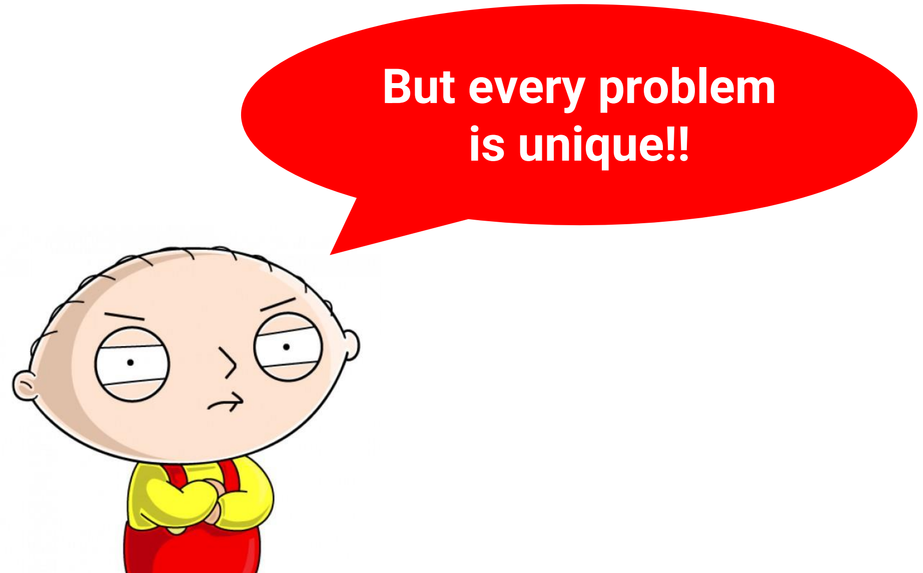
## What is a pattern?

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

## What is a pattern?

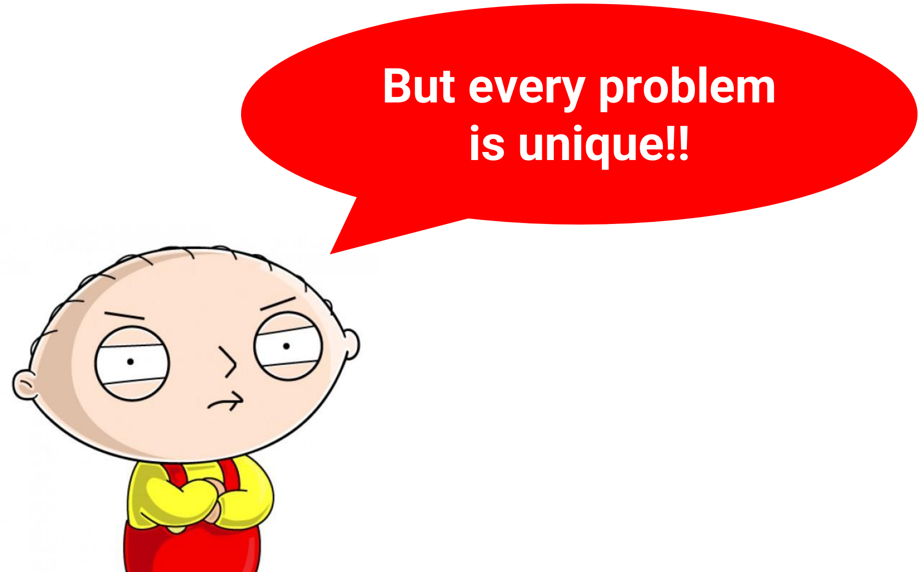
➔ Experts designers reuse solutions that have worked for them in the past

- ◆ Base new designs on prior experience



# What is a pattern?

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



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 in the OOP context

➔ Each pattern has the following elements:
- ◆ Name
- ◆ Problem
- ◆ Solution
- ◆ Consequences

➔ **Name**: is a shortcut to describe the pattern in a word or two
- ◆ Increases our vocabulary
- ◆ Allows to communicate in common terms with others

## Design patterns in the OOP context

- ➔ **Problem**: describes when to apply the pattern. Explains the problem and its context
  - ◆ Describe the symptoms of an inflexible design

- ➔ **Solution**: describes the elements that make up the design, their relationship, responsibilities and collaborations

## Design patterns in the OOP context

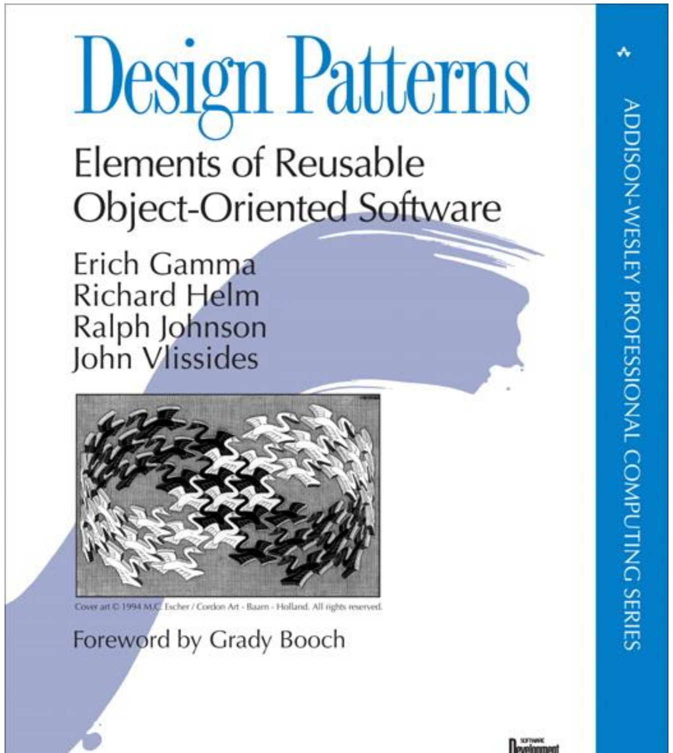
➔ **Consequences:** results and trade-off of applying a pattern
- ◆ Critical for evaluating design alternatives
- ◆ Understand costs and benefits

## 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

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

## Design patterns:
## History background

- ➔ They recompile best practices known informally by many engineers
- ➔ Also they proposed patterns designed by themselfs
- ➔ 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

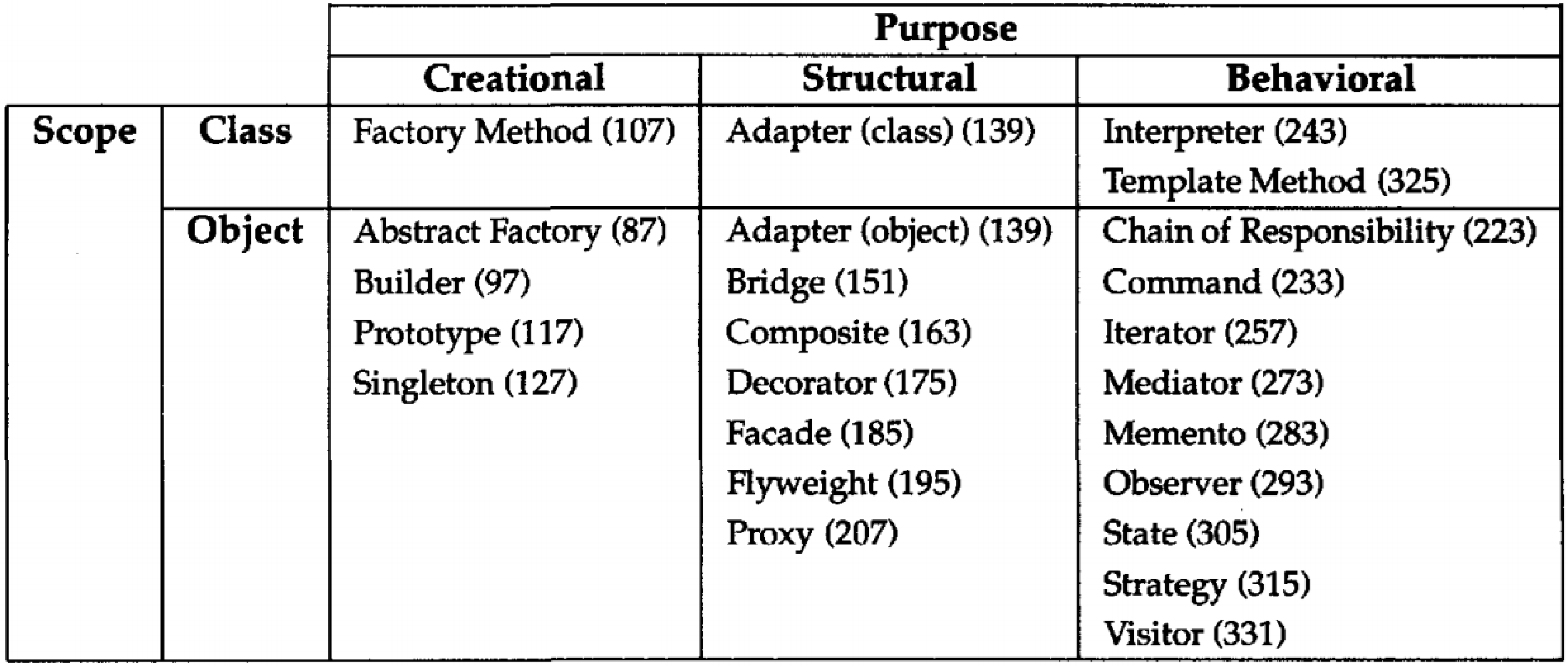
## Design patterns:
## Categories

- ➔ 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

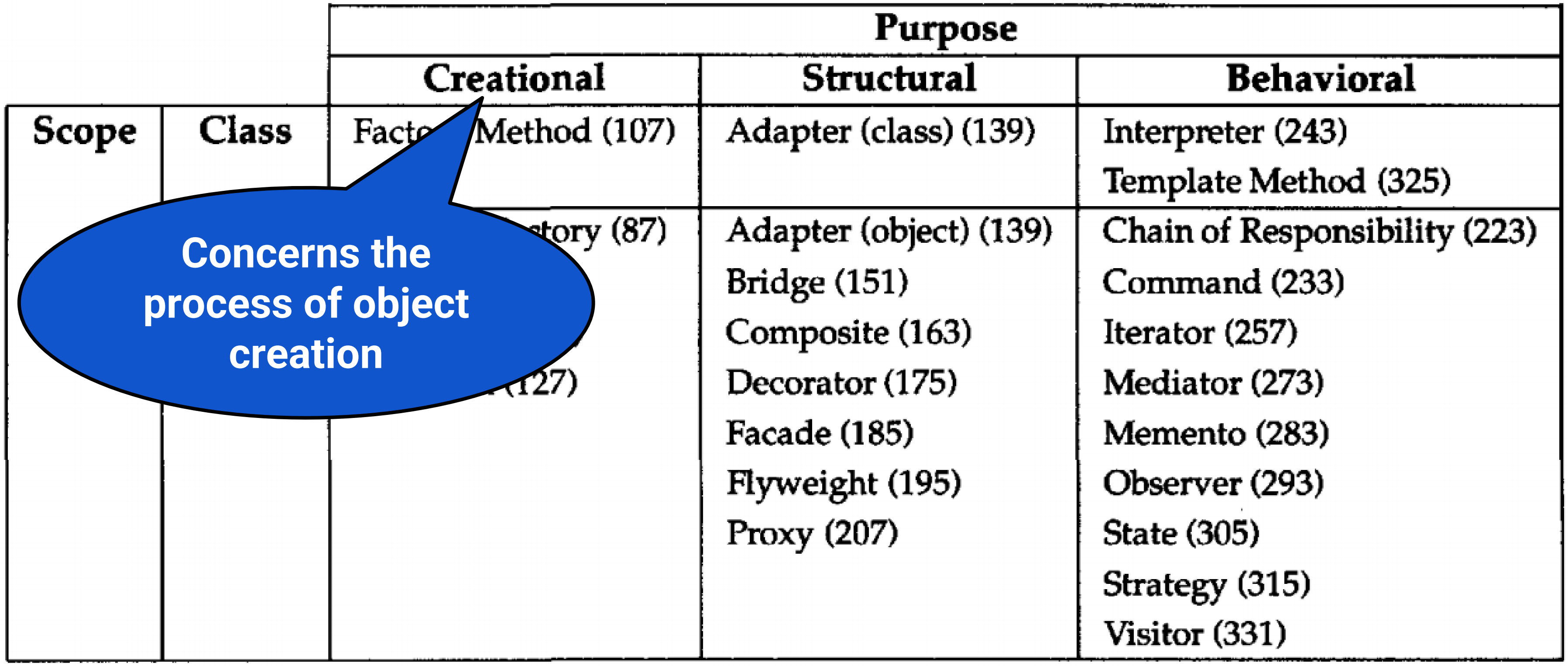
## Design patterns:
## Categories



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

# Design patterns:

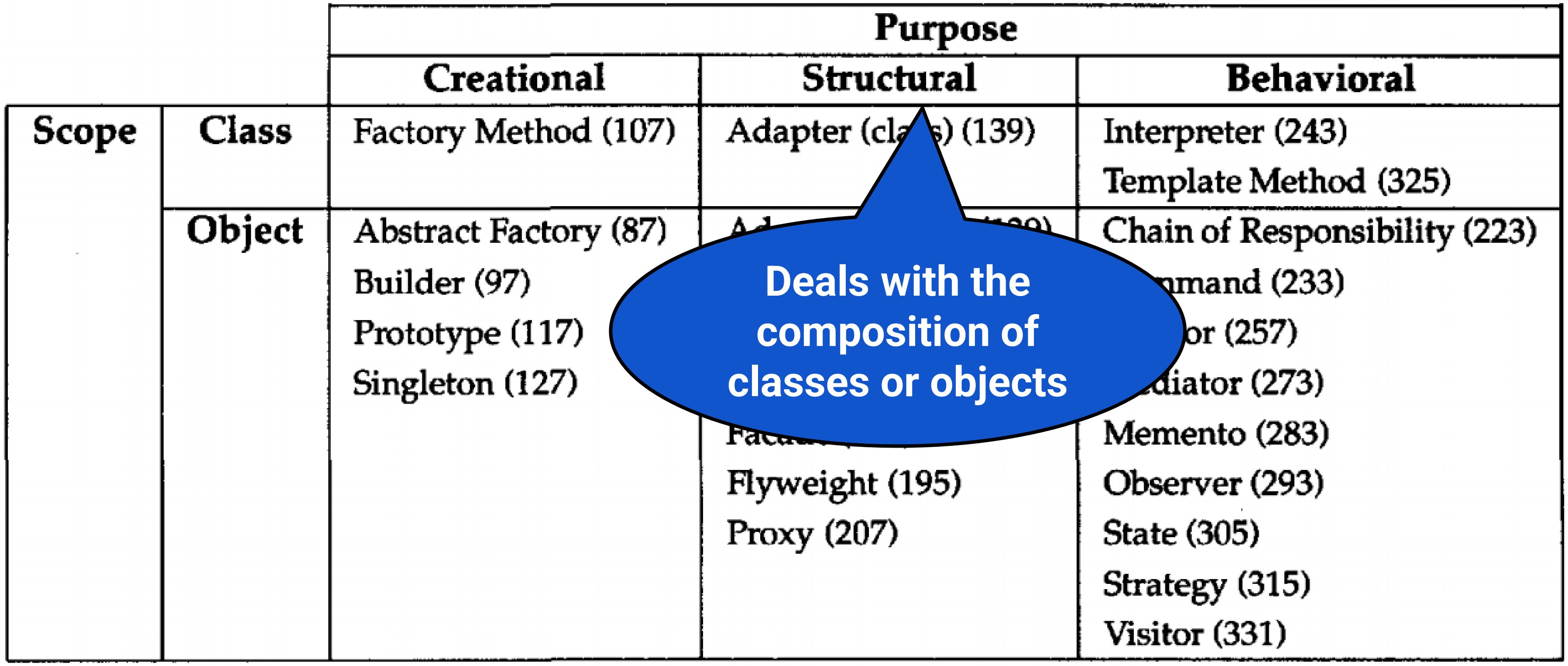
## Categories



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

Concerns the process of object creation

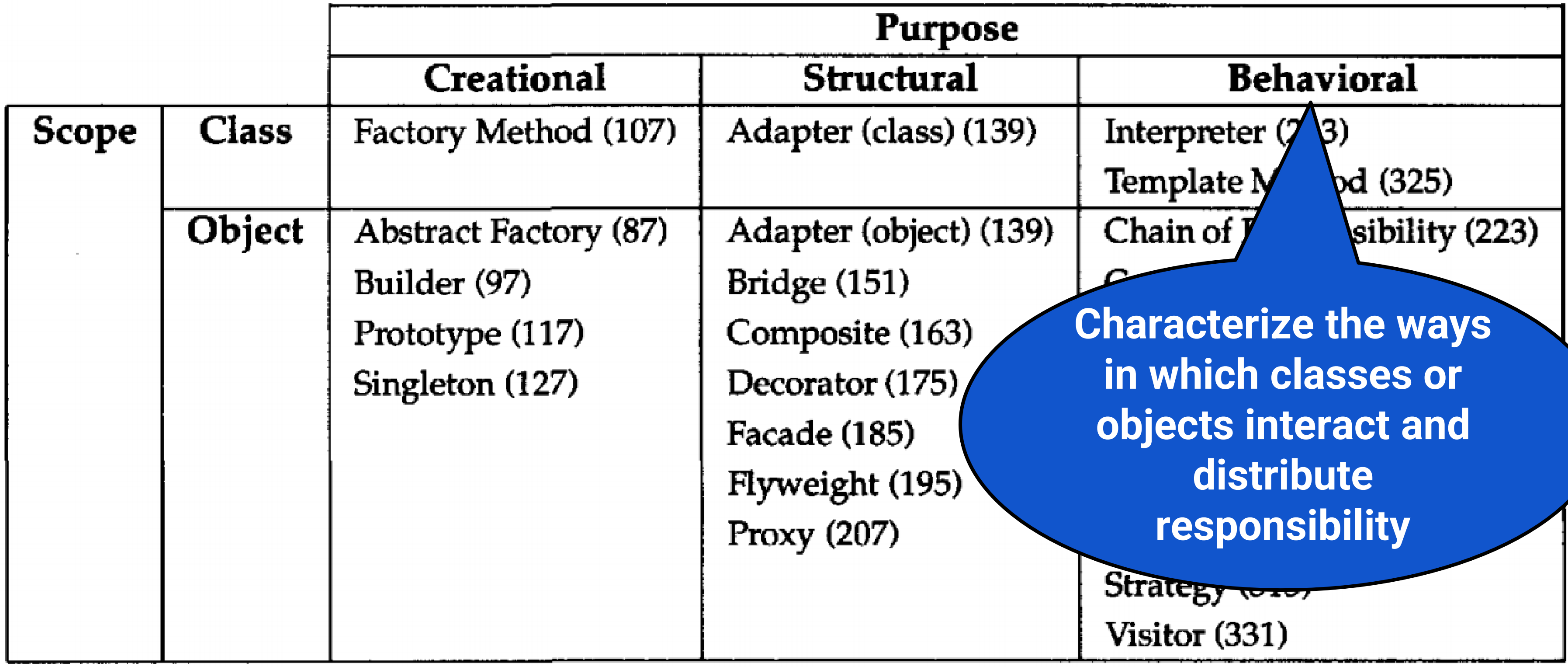
## Design patterns:
## Categories



|  |  | Purpose |  |  |
|---|---|---|---|---|
|  |  | Creational | Structural | Behavioral |
| Scope | Class | Factory Method (107) | Adapter (cla s) (139) | Interpreter (243)<br>Template Method (325) |
|  | Object | Abstract Factory (87)<br>Builder (97)<br>Prototype (117)<br>Singleton (127) | Ad (139)<br>Fac<br>Flyweight (195)<br>Proxy (207) | Chain of Responsibility (223)<br>mmand (233)<br>or (257)<br>diator (273)<br>Memento (283)<br>Observer (293)<br>State (305)<br>Strategy (315)<br>Visitor (331) |

Deals with the composition of classes or objects

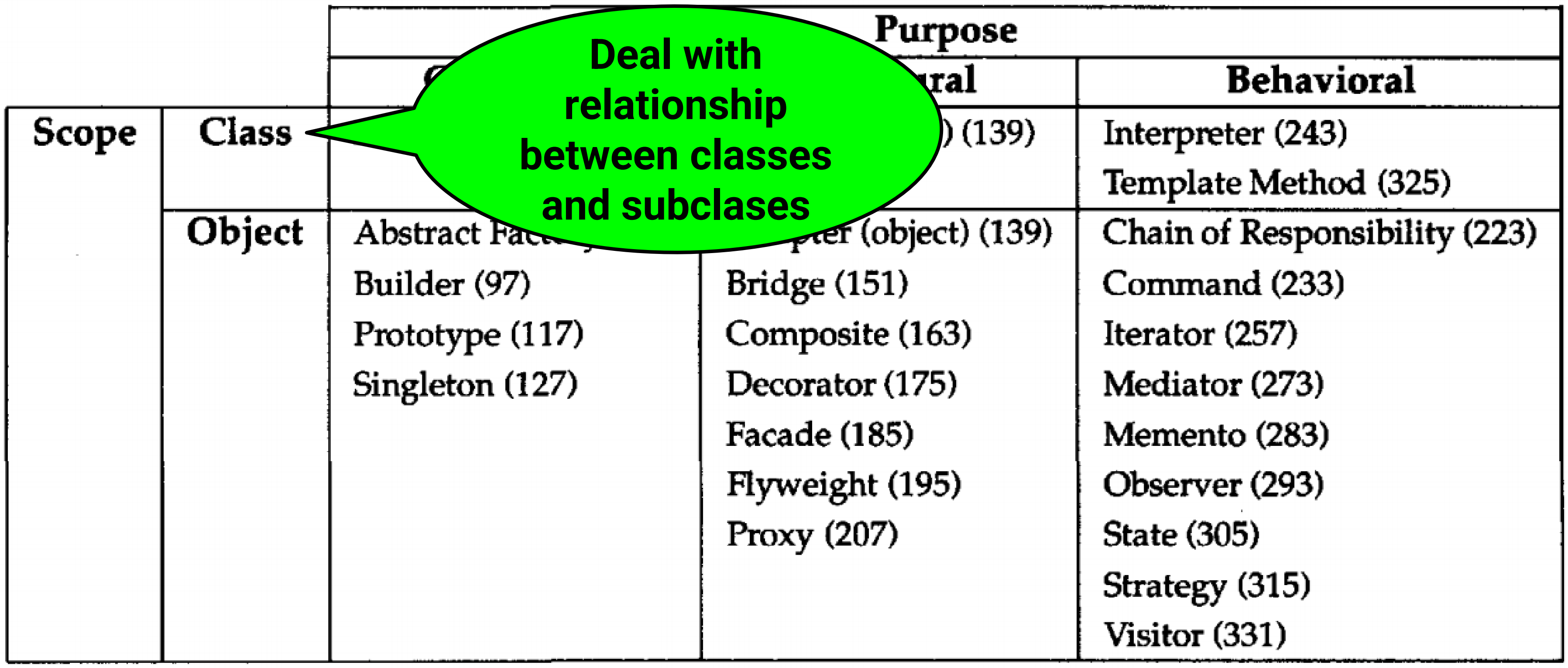
# Design patterns:
## Categories



| | | Purpose | | |
|---|---|---|---|---|
| | | Creational | Structural | Behavioral |
| Scope | Class | Factory Method (107) | Adapter (class) (139) | Interpreter (2 3) |
| | | | | Template M od (325) |
| | Object | Abstract Factory (87) | Adapter (object) (139) | Chain of R sibility (223) |
| | | Builder (97) | Bridge (151) | C |
| | | Prototype (117) | Composite (163) | |
| | | Singleton (127) | Decorator (175) | |
| | | | Facade (185) | |
| | | | Flyweight (195) | |
| | | | Proxy (207) | |
| | | | | Strategy |
| | | | | Visitor (331) |

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

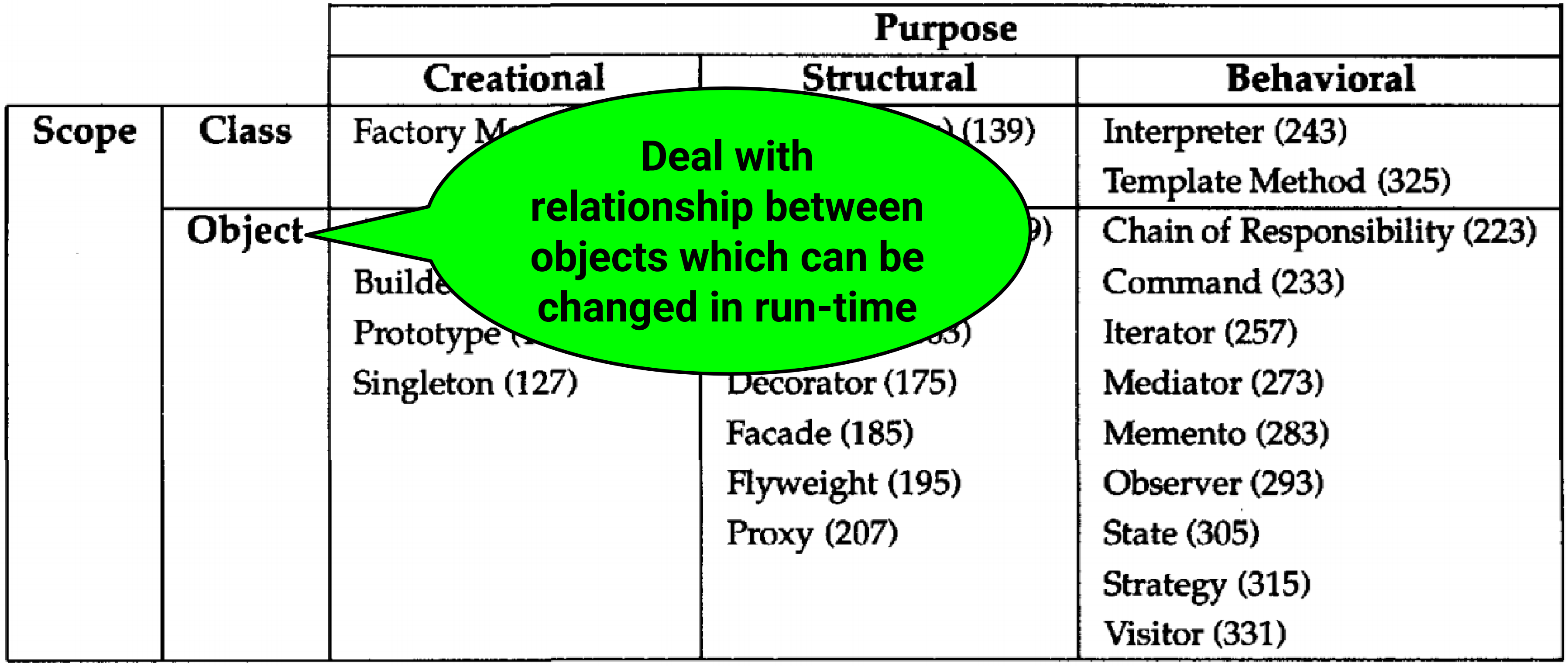
## Design patterns:
## Categories



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

Deal with relationship between classes and subclases

## Design patterns:
## Categories



| | | Purpose | | |
|---|---|---|---|---|
| | | Creational | Structural | Behavioral |
| Scope | Class | Factory M | ) (139) | Interpreter (243)<br>Template Method (325) |
| | Object | Builde<br>Prototype (<br>Singleton (127) | 9)<br>3)<br>Decorator (175)<br>Facade (185)<br>Flyweight (195)<br>Proxy (207) | Chain of Responsibility (223)<br>Command (233)<br>Iterator (257)<br>Mediator (273)<br>Memento (283)<br>Observer (293)<br>State (305)<br>Strategy (315)<br>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

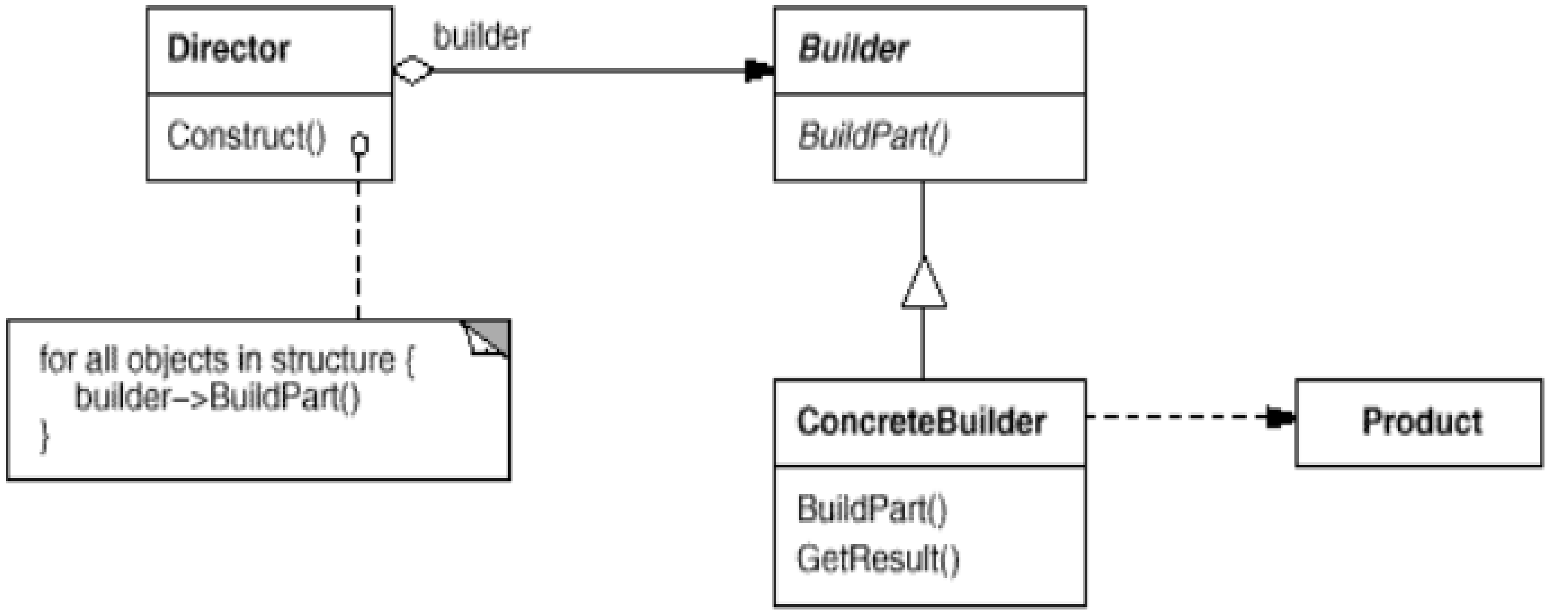
➔ Two main focuses:
- ◆ Encapsulates the knowledge about **which concrete classes** the system uses
- ◆ Hides how instances of the classes are created and put together

## Creational Pattern:
## Builder

- ➔ Purpose:
  - ◆ "Separate the construction of a complex object from its representation so that the same construction process can create different representations."

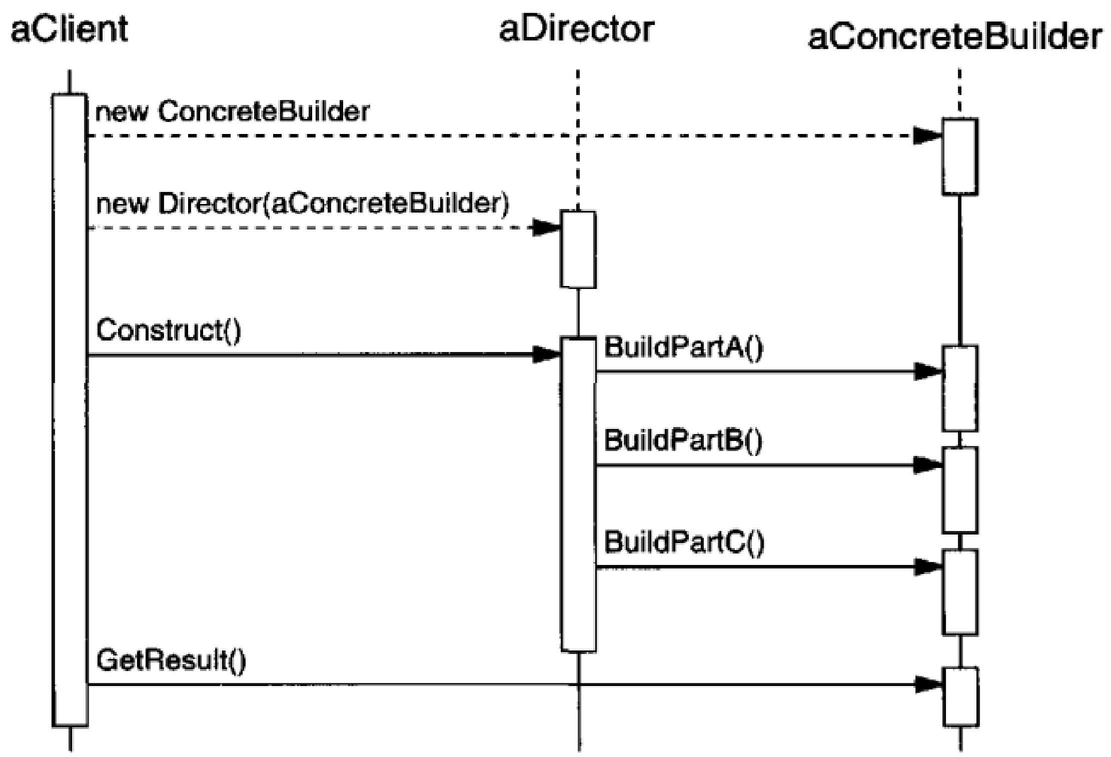
# Creational Pattern:
# Builder

➔ Structure:



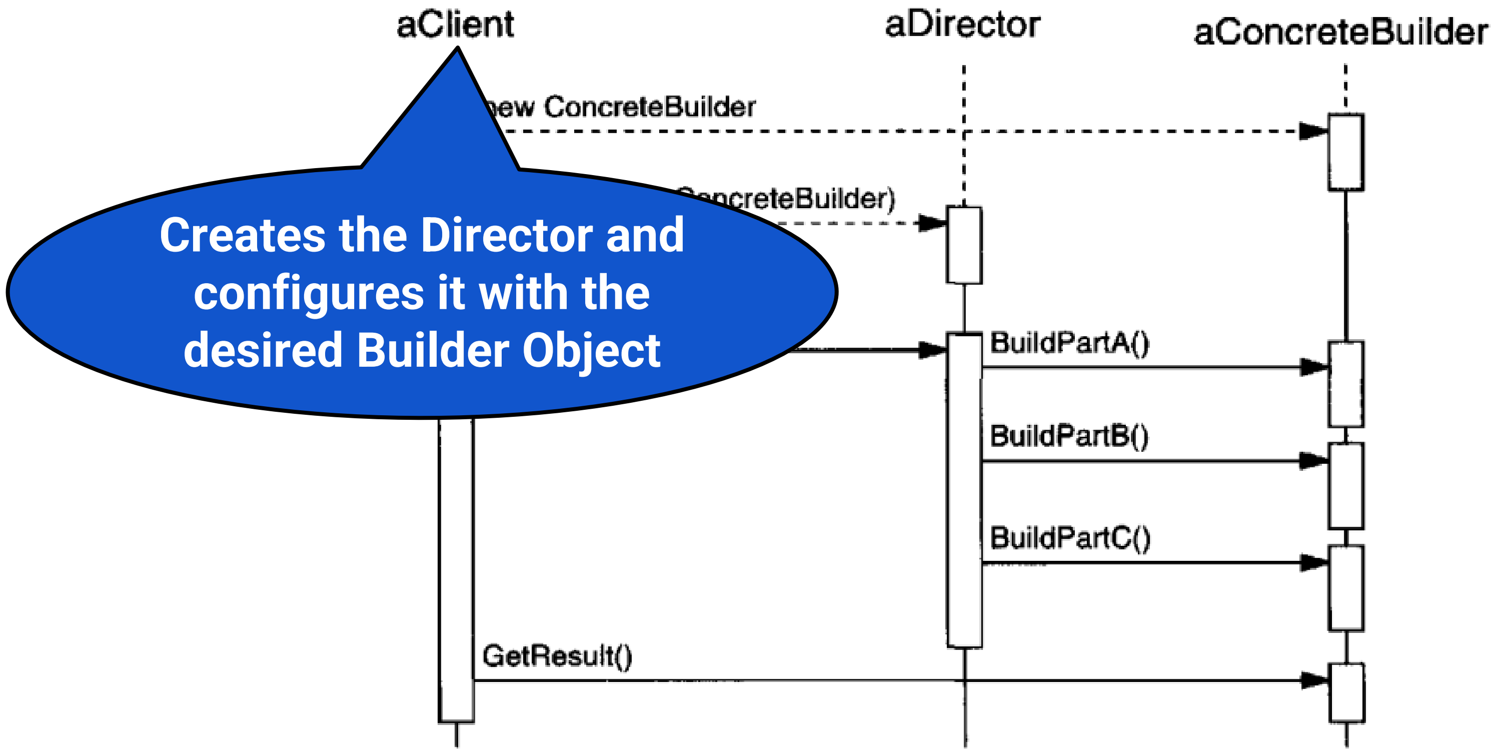
## Creational Pattern:
# Builder

➔Collaborations:



## Creational Pattern:
## Builder

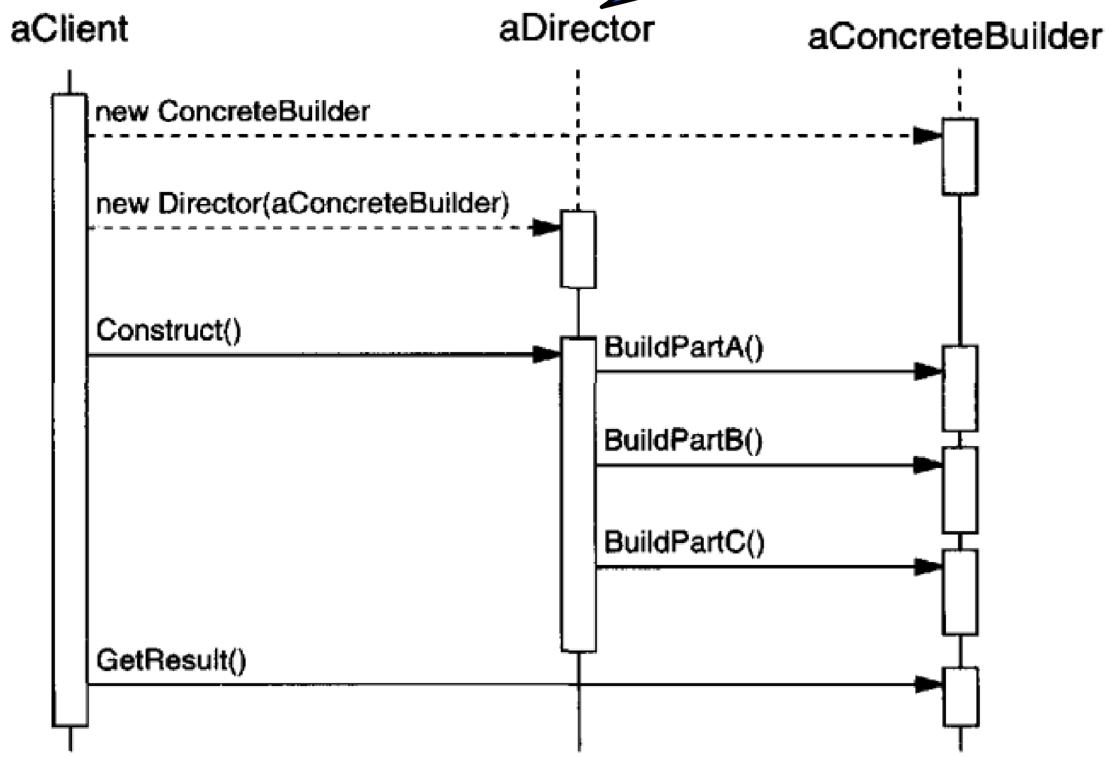
➔Collaborations:



## Creational Pattern:
## Builder

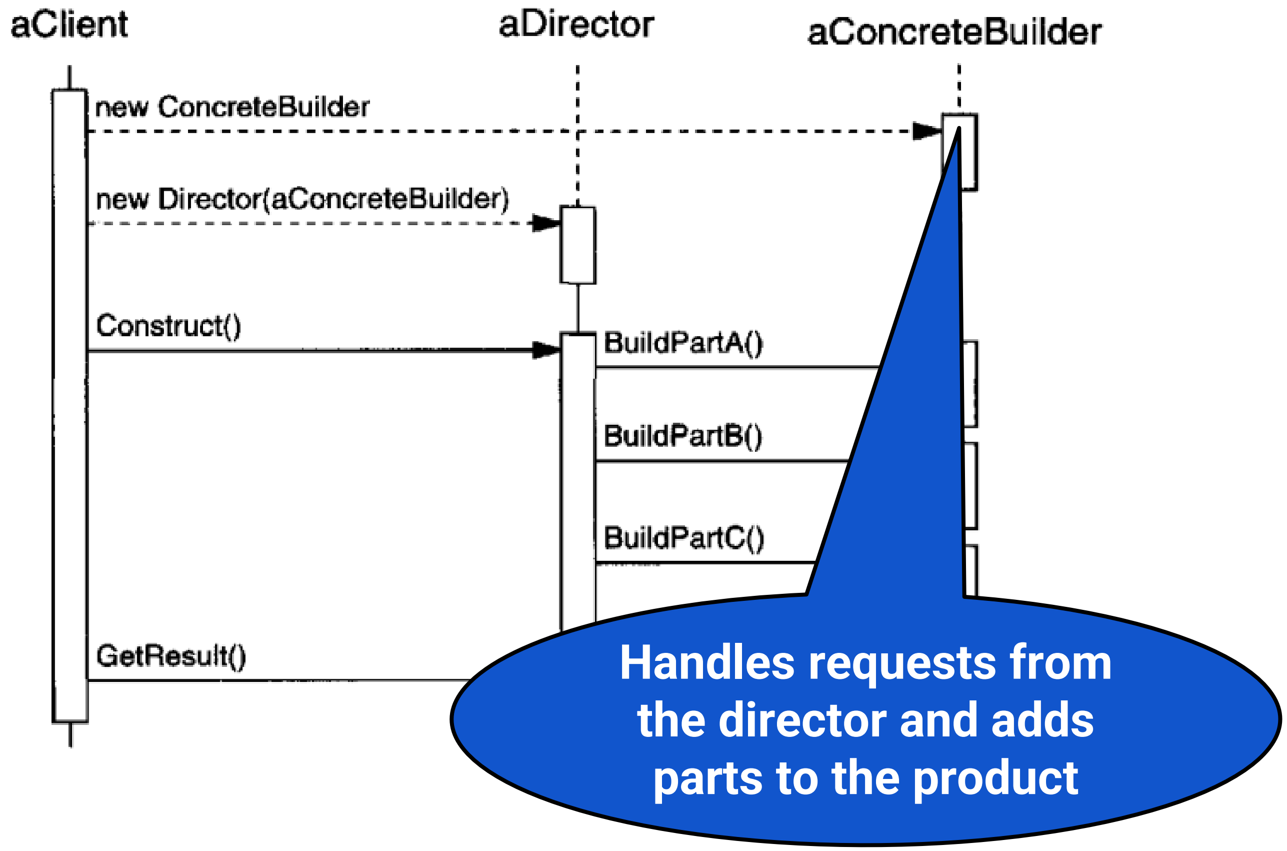
➔Collaborations:

Directs the assembly process of the whole. Relies on the builder



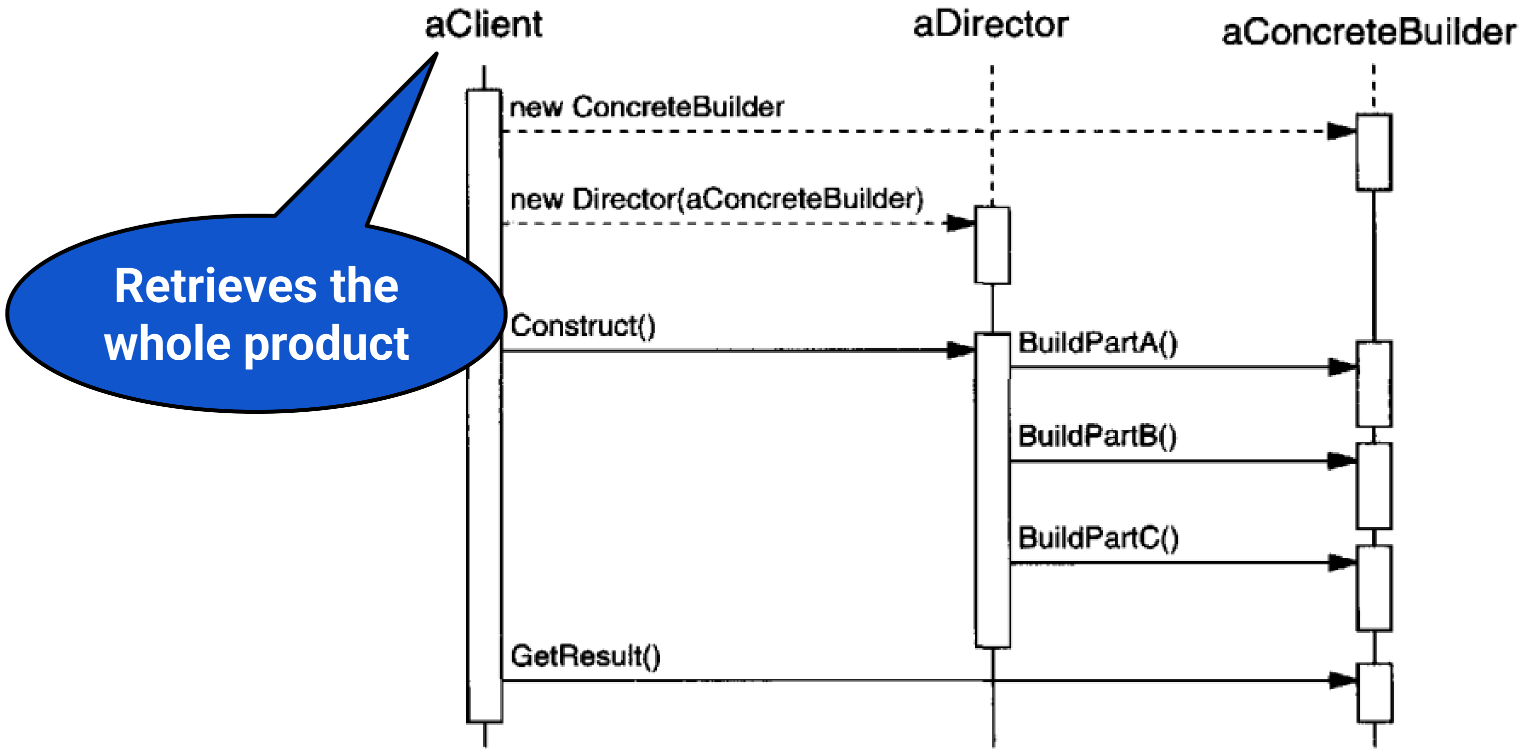
## Creational Pattern:
## Builder

➔ Collaborations:



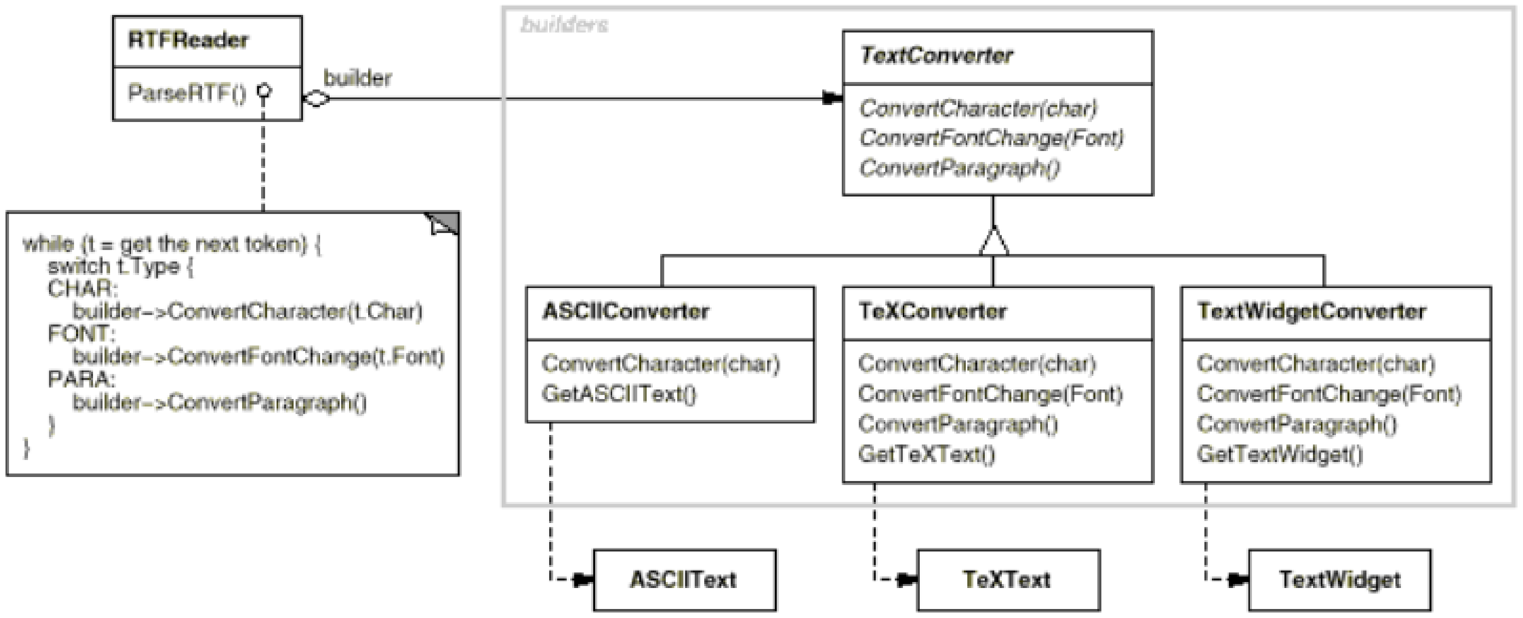
## Creational Pattern:
## Builder

➔Collaborations:



# Creational Pattern:
# Builder

➔ Example:



```
while (t = get the next token) {
    switch t.Type {
    CHAR:
        builder->ConvertCharacter(t.Char)
    FONT:
        builder->ConvertFontChange(t.Font)
    PARA:
        builder->ConvertParagraph()
    }
}
```

## Creational Pattern:
## Builder

➔ Apply it when:

- ◆ The algorithm for creating a complex object **should be independent of the parts** that make up the object and how they're assembled.
- ◆ The construction process must allow **different representations for the object** that's constructed.

## Creational Pattern:
## Builder

➔ Consequences:

It lets you vary a product's internal representation

- ◆ The builder can hide the internal structure of the product
- ◆ Hides how the product gets assembled
- ◆ A new builder is all that is need to chance the internals of the product

## Creational Pattern:
## Builder

➔ Consequences:

It isolates code for construction and representation

- ◆ Hides the way a complex object is constructed and represented
- ◆ Clients don’t need to know anything about the classes that define the product’s internal structure
- ◆ Different directors can reuse builders

## Creational Pattern:
## Builder

➔ Consequences:

It gives you finer control over the **construction process**

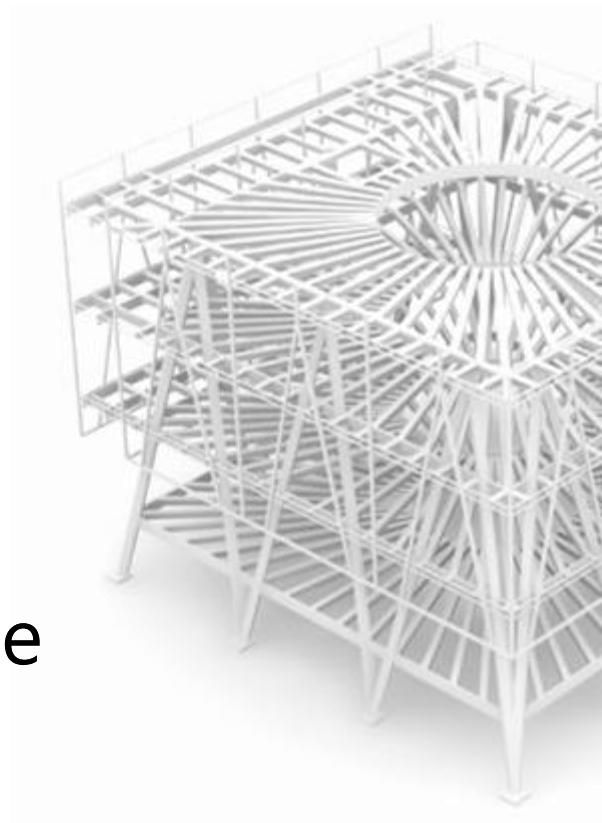
- ◆ Creates the product step by step under the control of the director
- ◆ The director retrieves the product only when it is finished

## 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

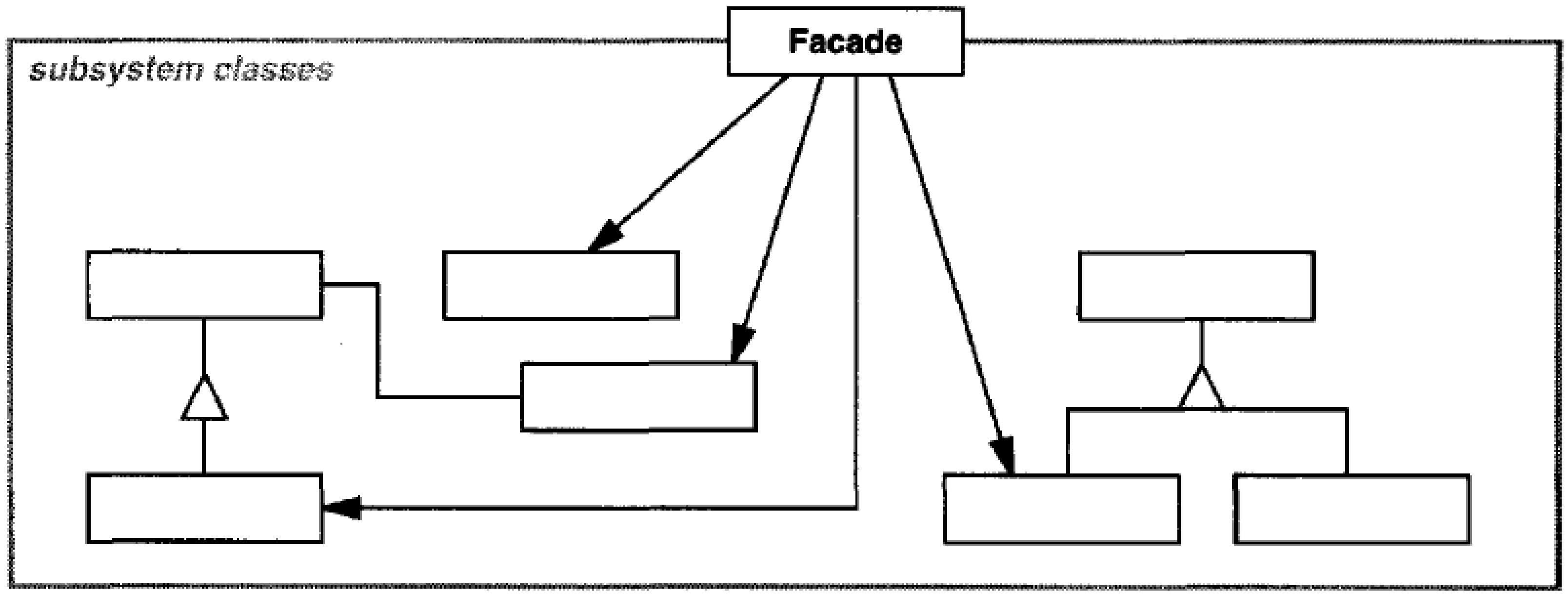


## Structural Patterns:
## Facade

- 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

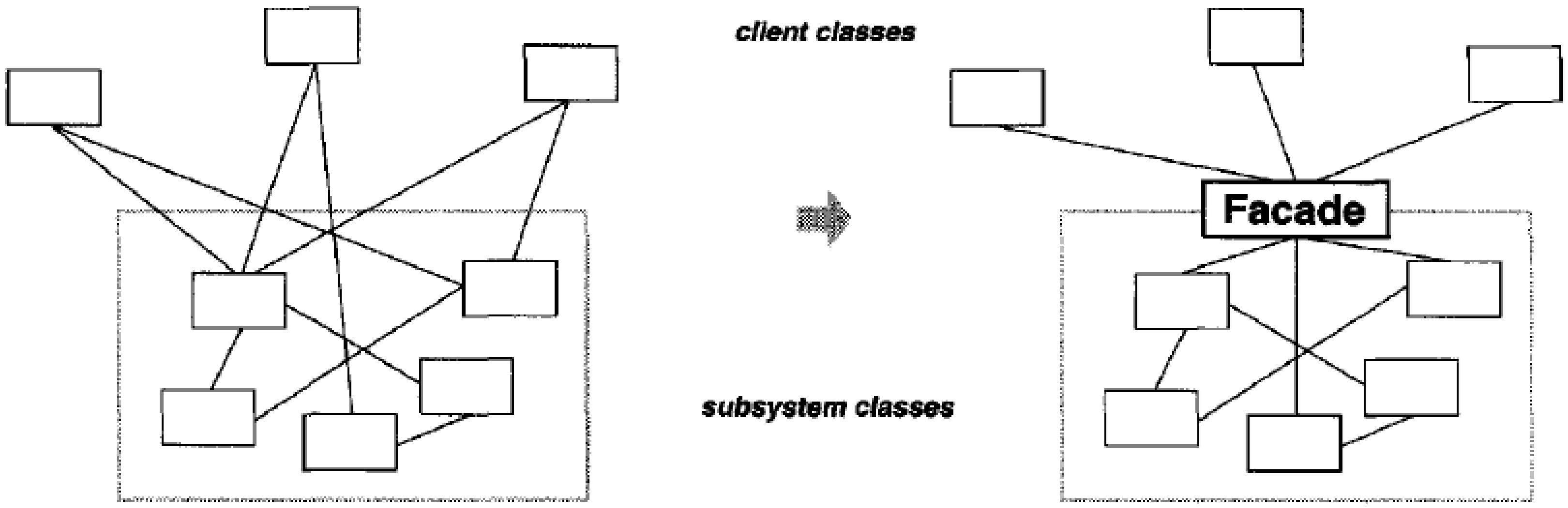
➔ Structure:



## Structural Patterns:

# Facade

➔ Structure:



## Structural Patterns:
## Facade

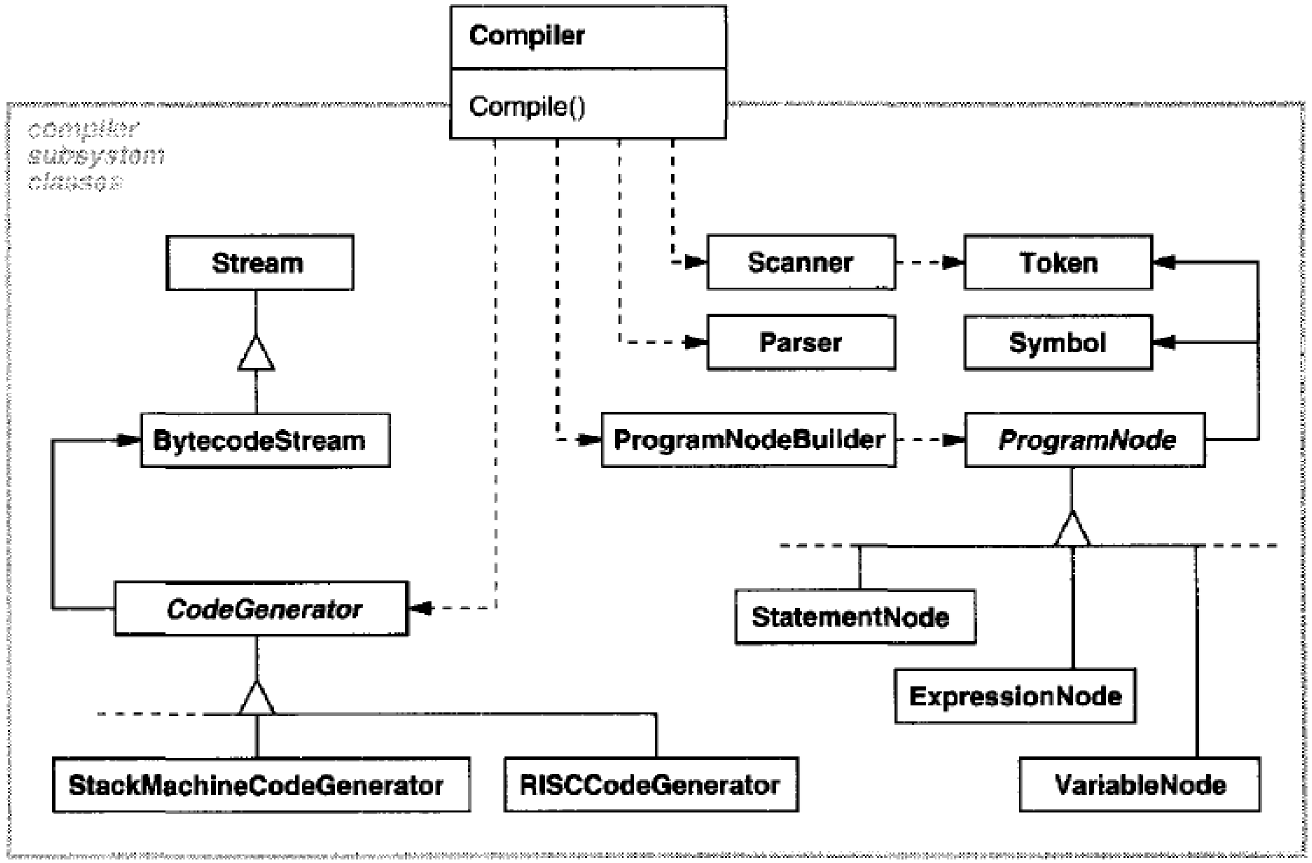
➔ Collaborations:

- ◆ 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:

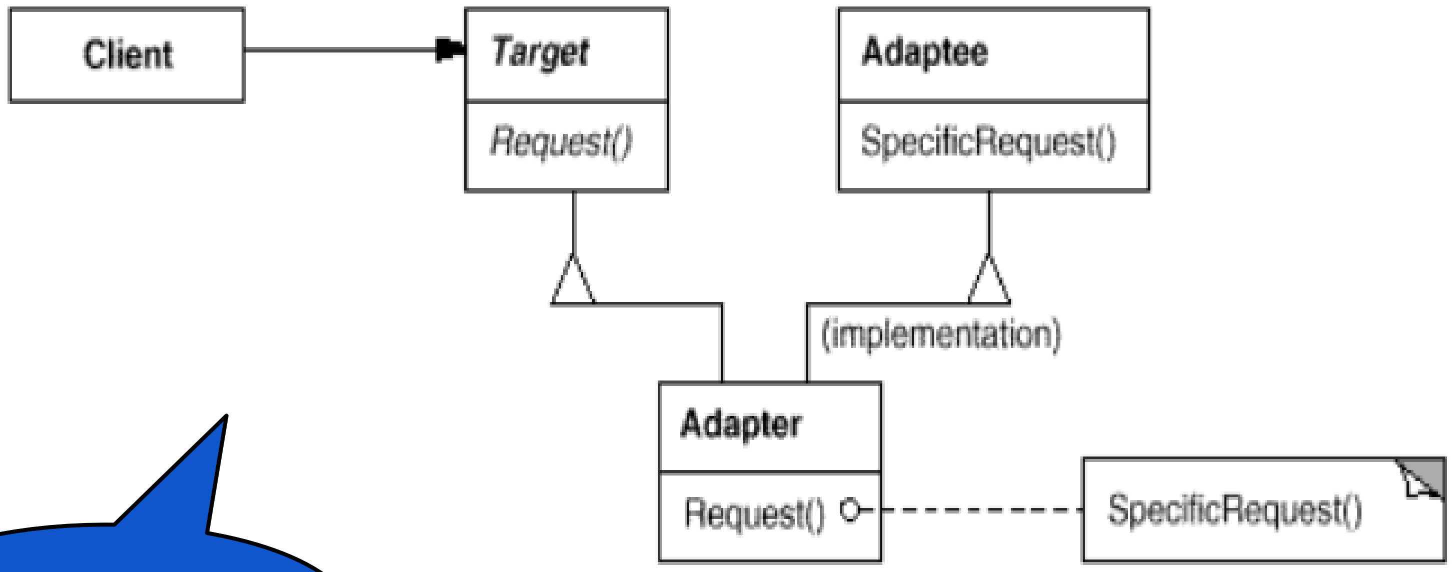
## Adapter

➔ 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

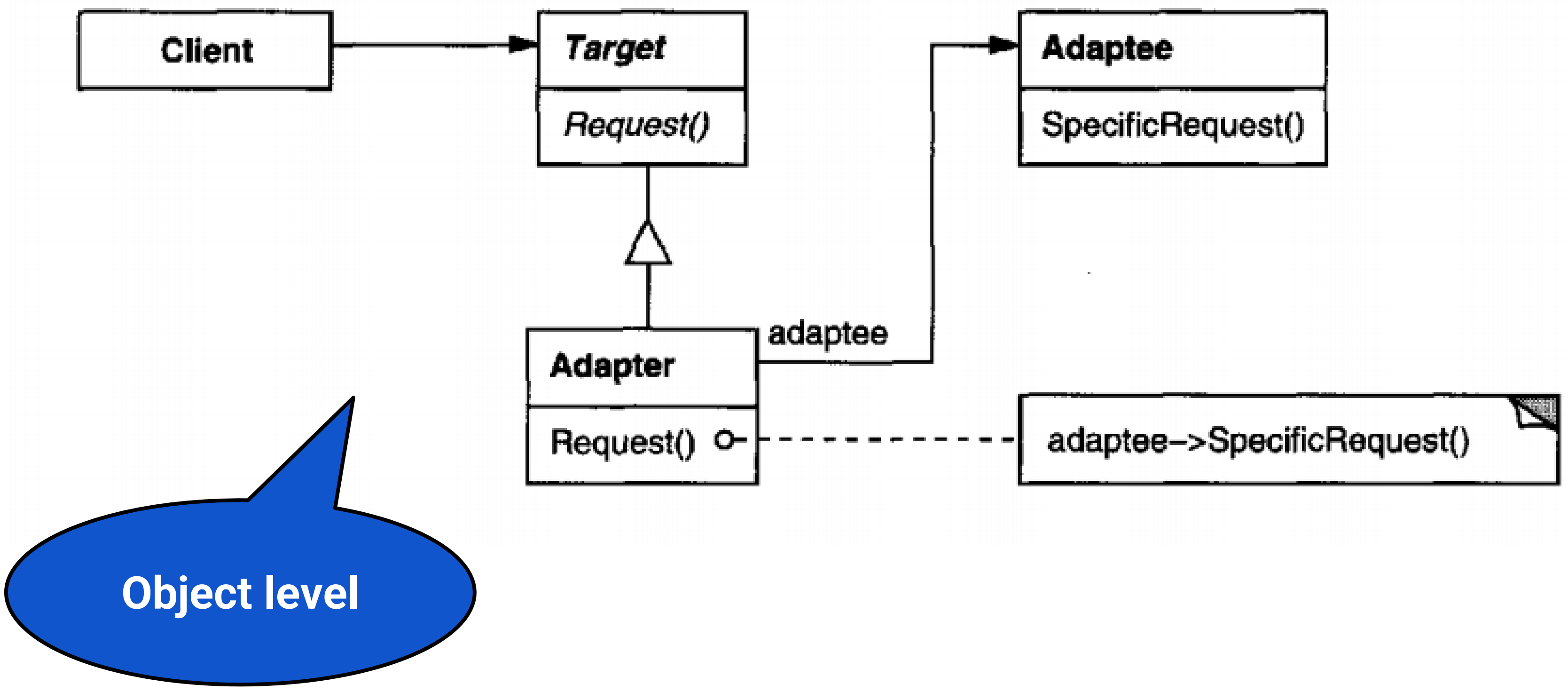
➔Structure:



Class level

## Structural Patterns:
## Adapter

➔ Structure:

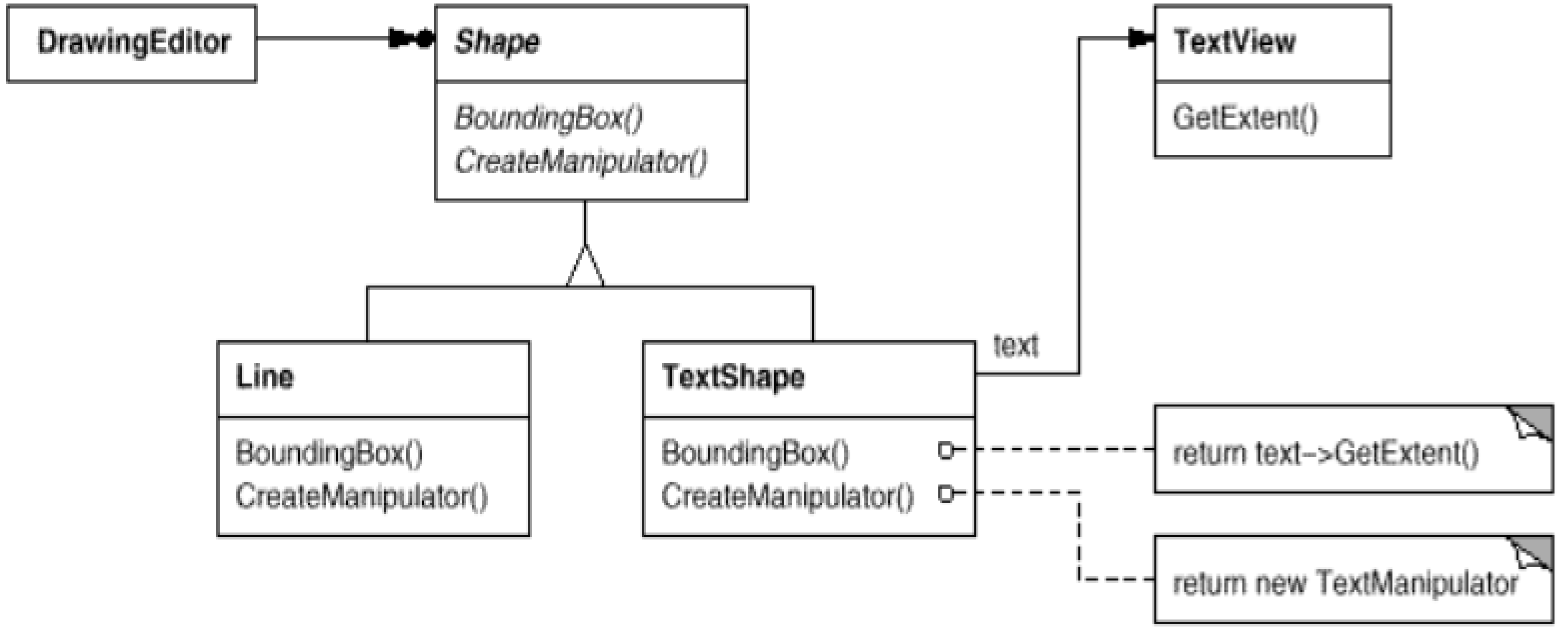


## Structural Patterns:
## Adapter

- ➔ 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

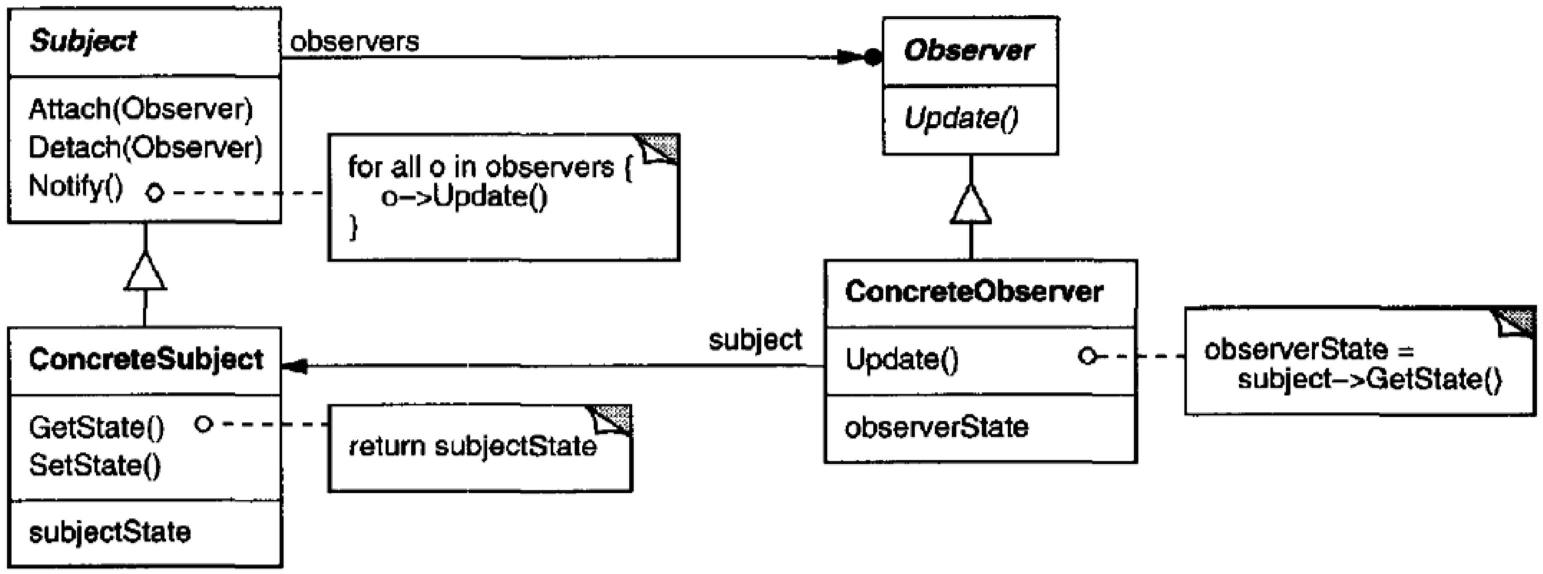
- ➔ Are concerned with algorithms and the assignment of responsibilities between objects
- ➔ Describe the patterns of communication between classes or objects
- ➔ Let you concentrate on the interconnection between objects

## 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:

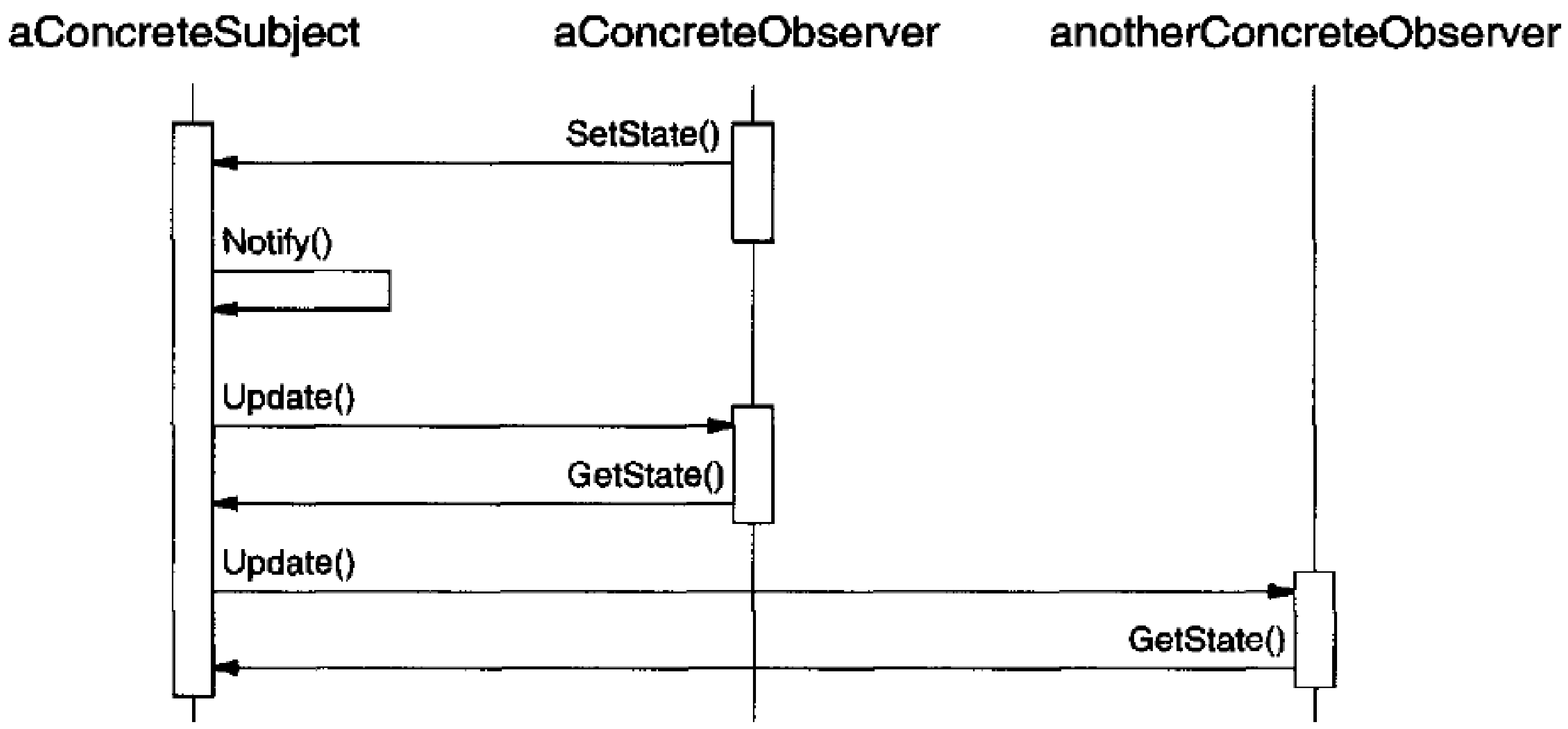


## Behavioral Patterns:
## Observer

- ➔ 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 abstractions 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:

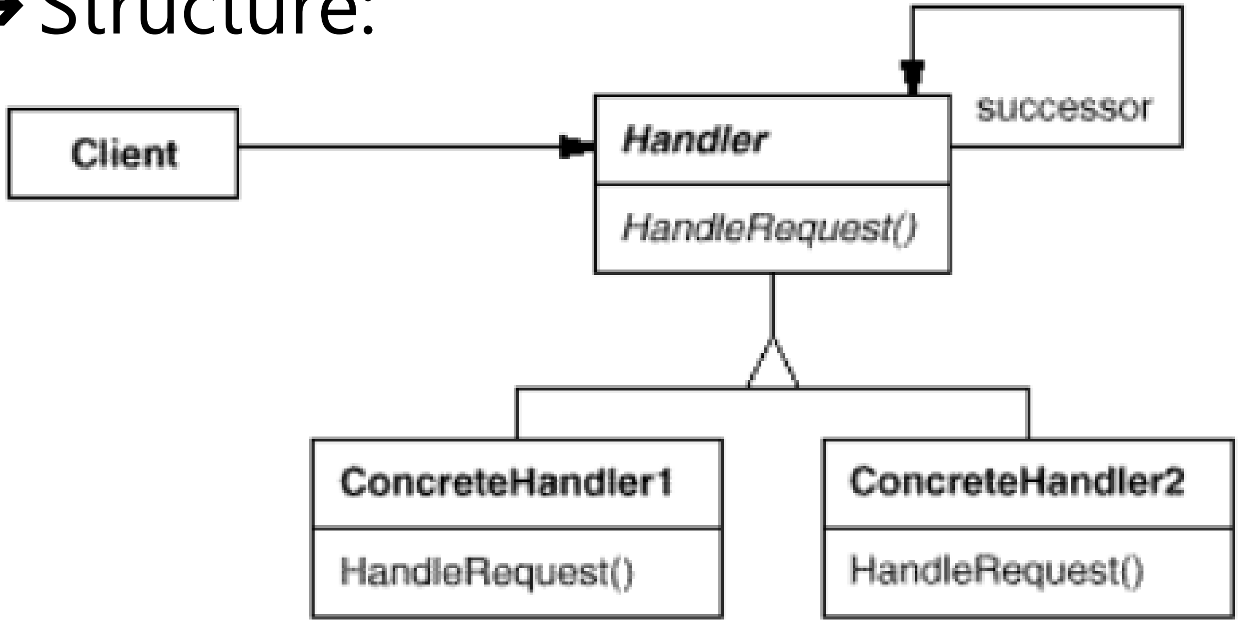
- ◆ Decouples subject and observer
- ◆ Supports broadcast communication
- ◆ Unexpected updates: observers have no knowledge of each other's presence they don't know the cost of each change for the overall system
- ◆ Is hard for observers know what changed

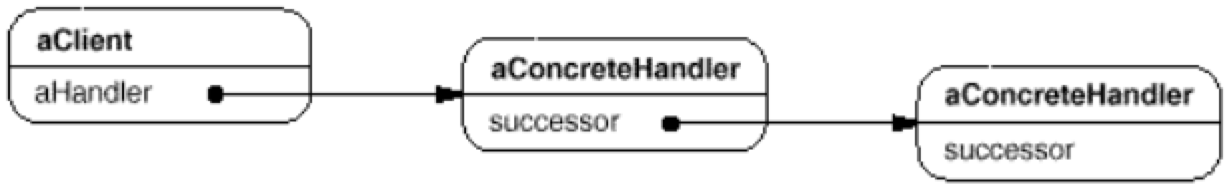
## Behavioral Patterns:
## Chain of Responsibility

- ➔ Purpose:
  - ◆ "Avoid coupling the sender of a request to its receiver by giving  more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it."

## Behavioral Patterns:
# Chain of Responsibility

➔ Structure:





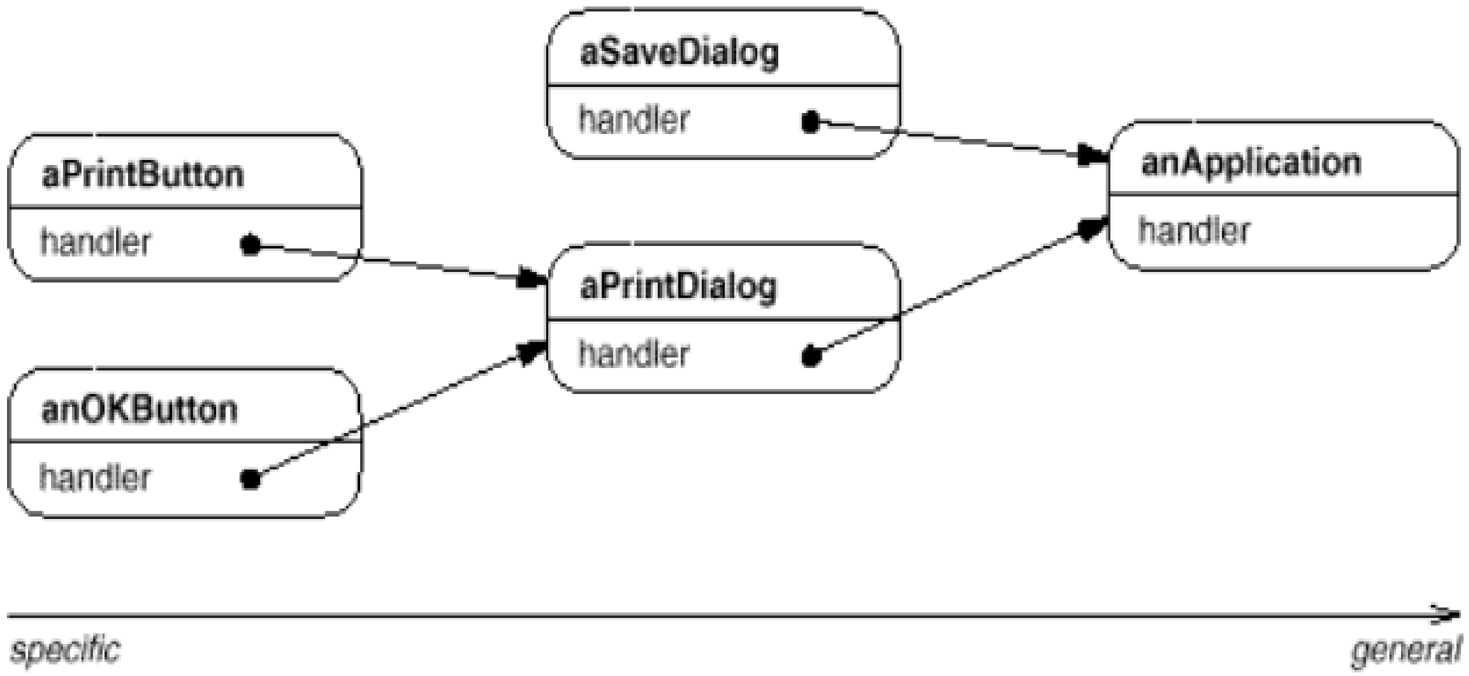
## Behavioral Patterns:
## Chain of Responsibility

➔ Collaborations:

- ◆ When a client issues a request, the request propagates along the chain until a concrete handler object takes responsibility for handling it

## Behavioral Patterns:
## Chain of Responsibility

➔ Example:



## Behavioral Patterns:
## Chain of Responsibility

➔ Apply when:

- ◆ More than one object may handle a request, and the handler isn't known a priori. The handler should be ascertained automatically.
- ◆ You want to issue a request to one of several objects without specifying the receiver explicitly.
- ◆ The set of objects that can handle a request should be specified dynamically.

## Behavioral Patterns:
## Chain of Responsibility

➔ Consequences

Reduced coupling:

- ◆ Frees an object from knowing which other object handles a request.
- ◆ The sender and receiver have no explicit knowledge of each other, and the object in the chain doesn't know about the chain structure

**Behavioral Patterns:**

## Chain of Responsibility

➔ Consequences

Added flexibility in assigning responsibilities to objects

- ◆ Distributes responsibilities among objects
- ◆ You can add/remove responsibilities for handling a request by adding or changing the chain at run-time

## Behavioral Patterns:
## Chain of Responsibility

- ➔ Consequences

  Receipt isn't guaranteed

  - ◆ Since a request has no explicit receiver, there is no guarantee it’ll be handled

## How to select a design pattern?

- ➔ Consider how the pattern solve a design problem
  - ◆ What do you want to solve? Granularity? Coupling? Flexibility? Design for change?

- ➔ Think on what you may want to change later without having to redesign

- ➔ Encapsulate the concept that varies

## Design patterns

➔ Why do we need design patterns?

## What about the other patterns?

➔ Homework:

- ◆ Select at least three patterns, do a recording explaining each of them that includes:
  - Purpose
  - Structure
  - Consequences
  - Scenarios where the pattern is applicable
  - Code sample in Java and C++

- ◆ More details will be sent to the group

# Software Product Quality

An introduction

## Before we start…

- ➔ Why is this topic software **product** quality?

- ➔ Software quality may refer to two different things:
  - ◆ Quality in the **process** of creating software
  - ◆ Quality in the software **product**

## Before we start...

➔ What is QA?

➔ What is QC?

➔ What is Testing?

## Before we start…

➔ Many times QA, QC and testing are terms used interchangeable

➔ But they are not the same!

## Before we start…

➔ QA is quality assurance

➔ Quality assurance is a set of activities designed to ensure that the **development or maintenance process** is adequate to ensure a system will meet its objectives

## Before we start…

➔ QC is quality control

➔ Quality control is a set of activities designed to **evaluate a developed work product**

➔ Testing is one of the quality control activities. Is the **process of executing a system** with the intent of finding defects

## Before we start…

- ➔ You will see many job descriptions marked as QA when the role really is only testing
- ➔ Many people say they are “QA Engineers” when they should be saying they are “Testers”
- ➔ “QA Engineer” sounds better than “Tester”

## Before we start...

➔ QA & QC are both essential achieve software quality

➔ Applying only QA we may have optimized processes to create quality software, but we will never check if the product really have the quality we want

## Before we start...

- ➔ Likewise, applying on QC is simply conducting tests without any vision on how to avoid defects during the process

- ➔ So remember:
  - ◆ QA is the process of managing for quality
  - ◆ QC is used to verify quality of the output
  - ◆ Testing is one of the many techniques involves in QC

## What is quality?

- ➔ ISO defines quality as the degree to which a set of inherent characteristics fulfills requirements
  - ◆ Functional requirements
  - ◆ Non-functional requirements
- ➔ There are many formal specifications/standards related to software quality

## ISO/IEC 9126-1 Software Quality

- ➔ Generally accepted software quality standard
- ➔ The fundamental objective is to address some of the **well known human biases** that can adversely affect the delivery and perception of a software development project

## ISO/IEC 9126-1 Software Quality

- ➔ Divided in four parts:
  - ◆ Quality model
  - ◆ Internal metrics
  - ◆ External metrics
  - ◆ Quality in use metrics

- ➔ Focused on the product not in the process
  - ◆ But this doesn’t mean is not important for QA

## ISO/IEC 9126-1 Software Quality

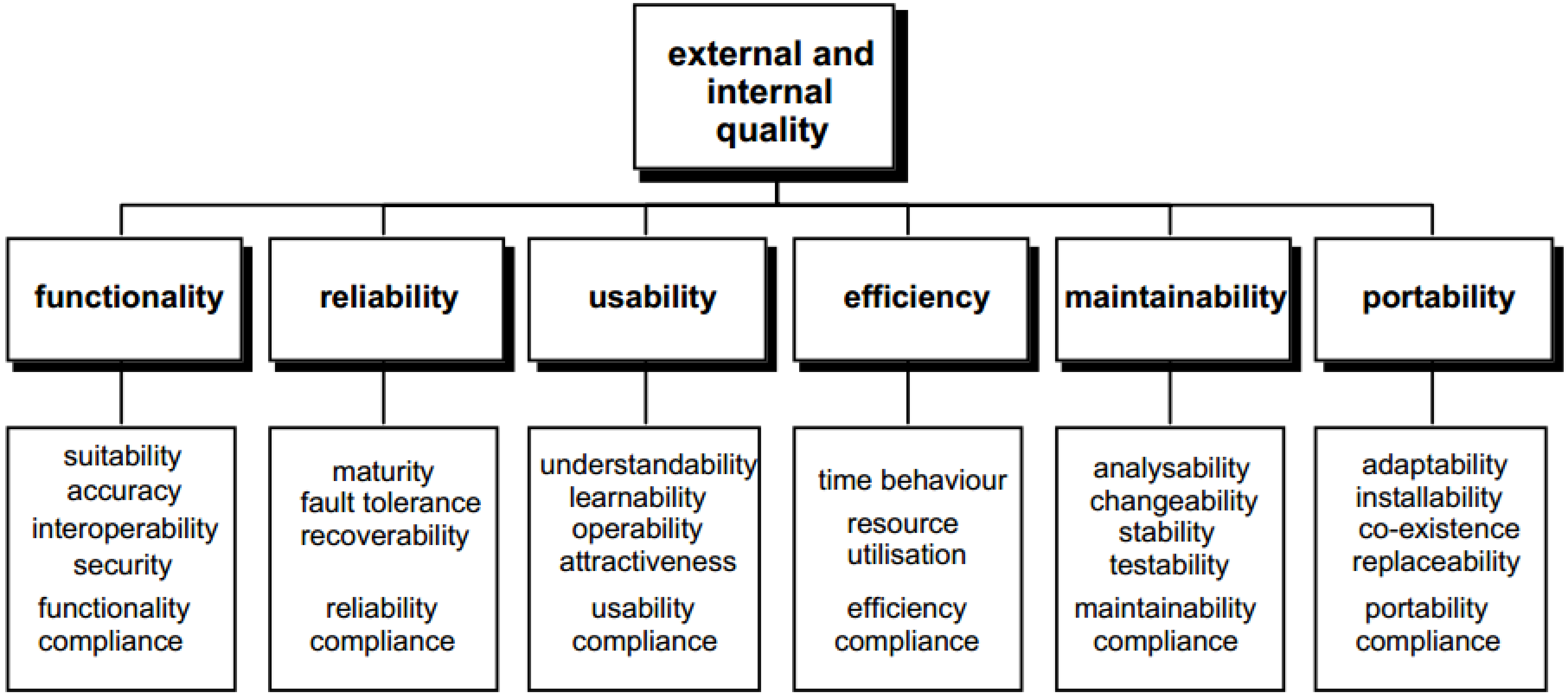
➔ The quality model defines internal and external quality attributes

➔ Internal is related to the software structure, architecture, code: **static attributes**

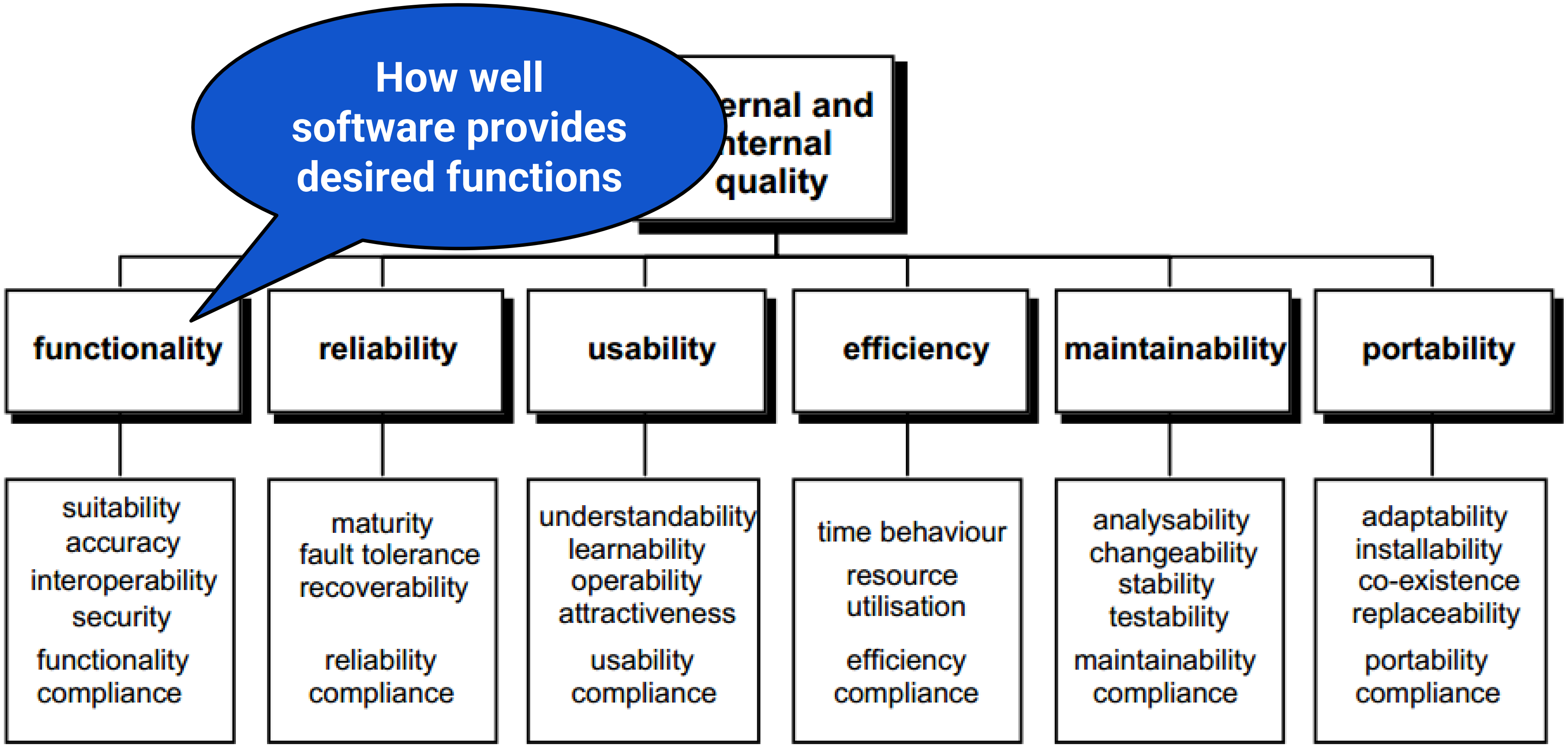
## ISO/IEC 9126-1 Software Quality

➔ External is related to how the software behaves under certain circumstances

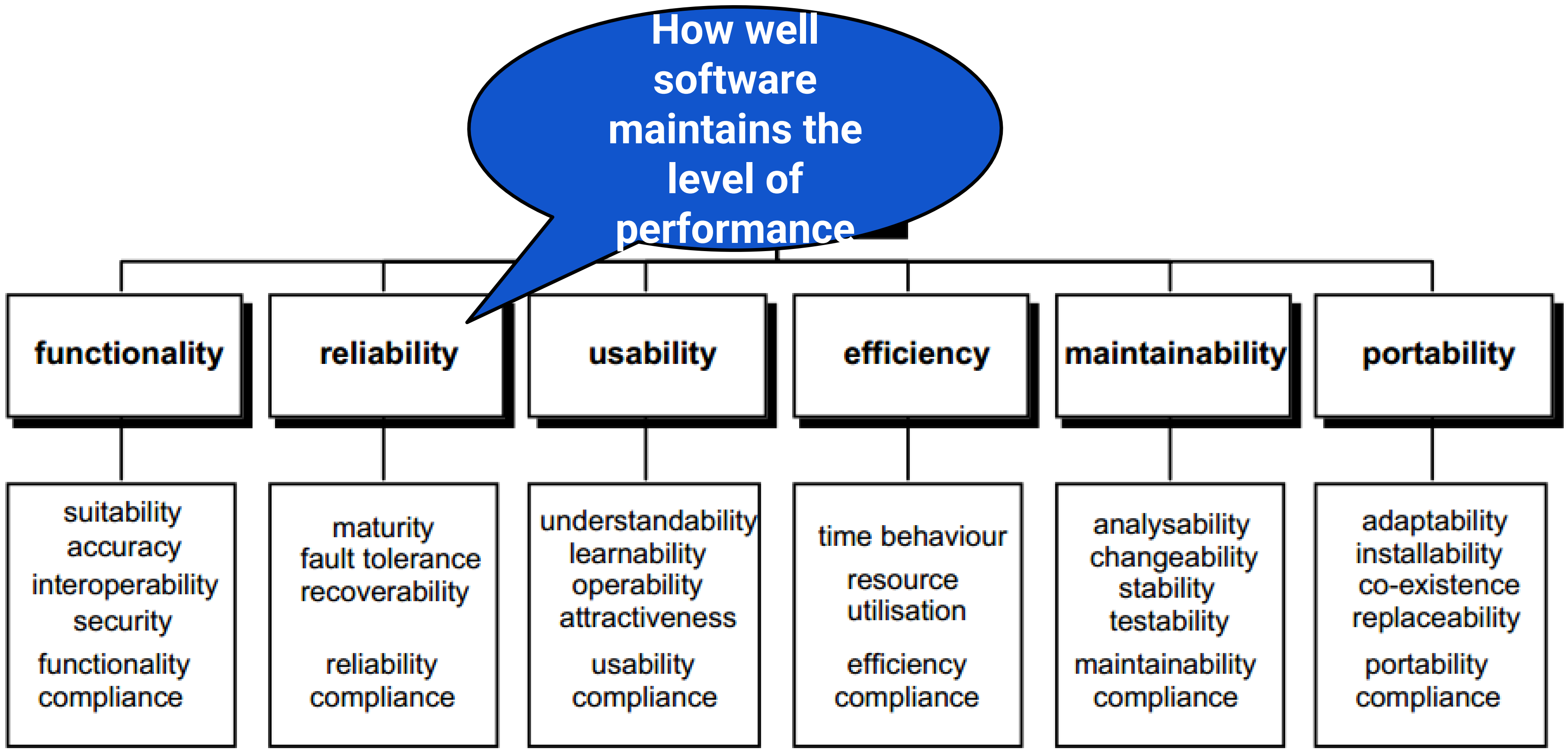
# ISO/IEC 9126-1 Software Quality



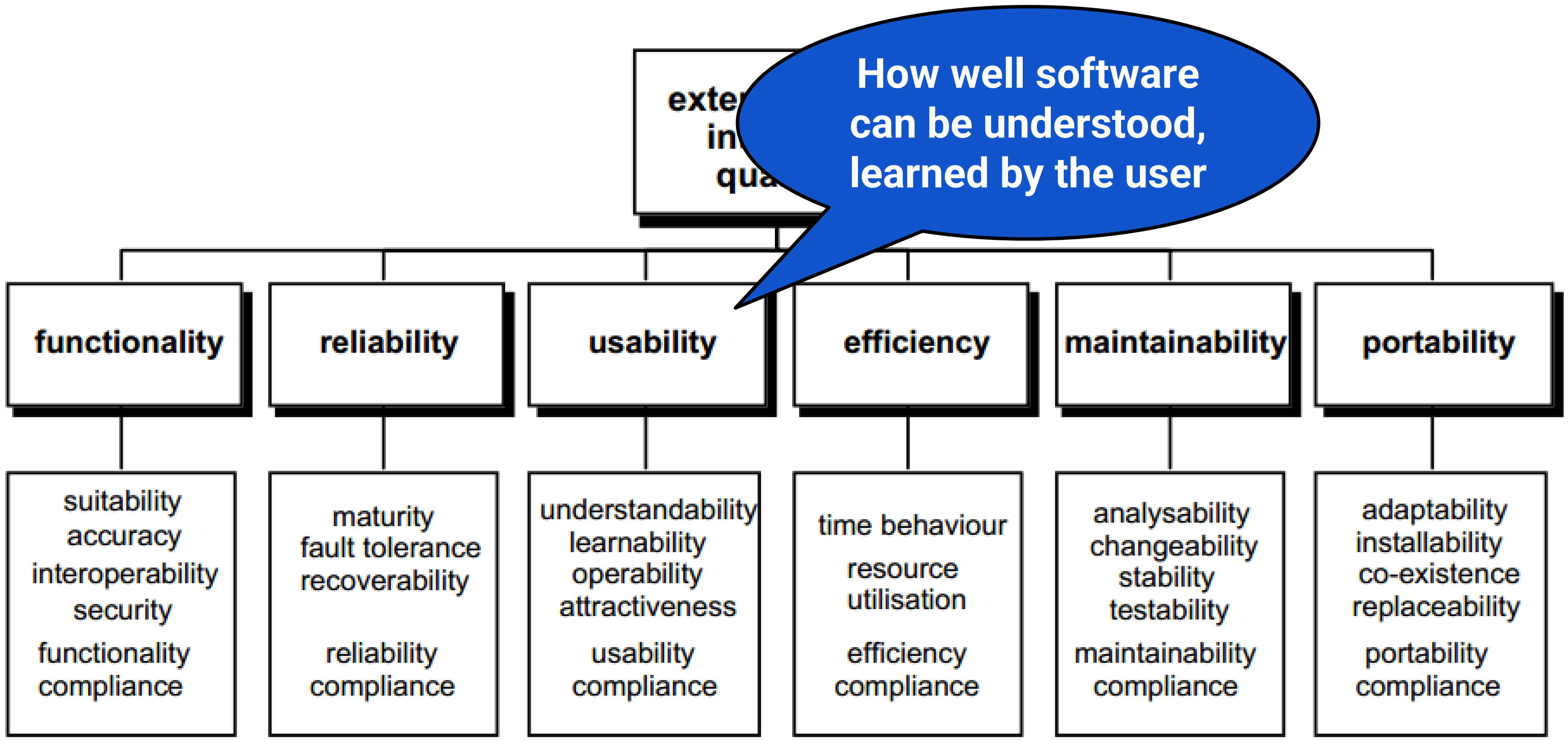
# ISO/IEC 9126-1 Software Quality


How well software provides desired functions
ernal and
nternal
quality
functionality
reliability
usability
efficiency
maintainability
portability
suitability accuracy interoperability security functionality compliance
maturity fault tolerance recoverability reliability compliance
understandability learnability operability attractiveness usability compliance
time behaviour resource utilisation efficiency compliance
analysability changeability stability testability maintainability compliance
adaptability installability co-existence replaceability portability compliance

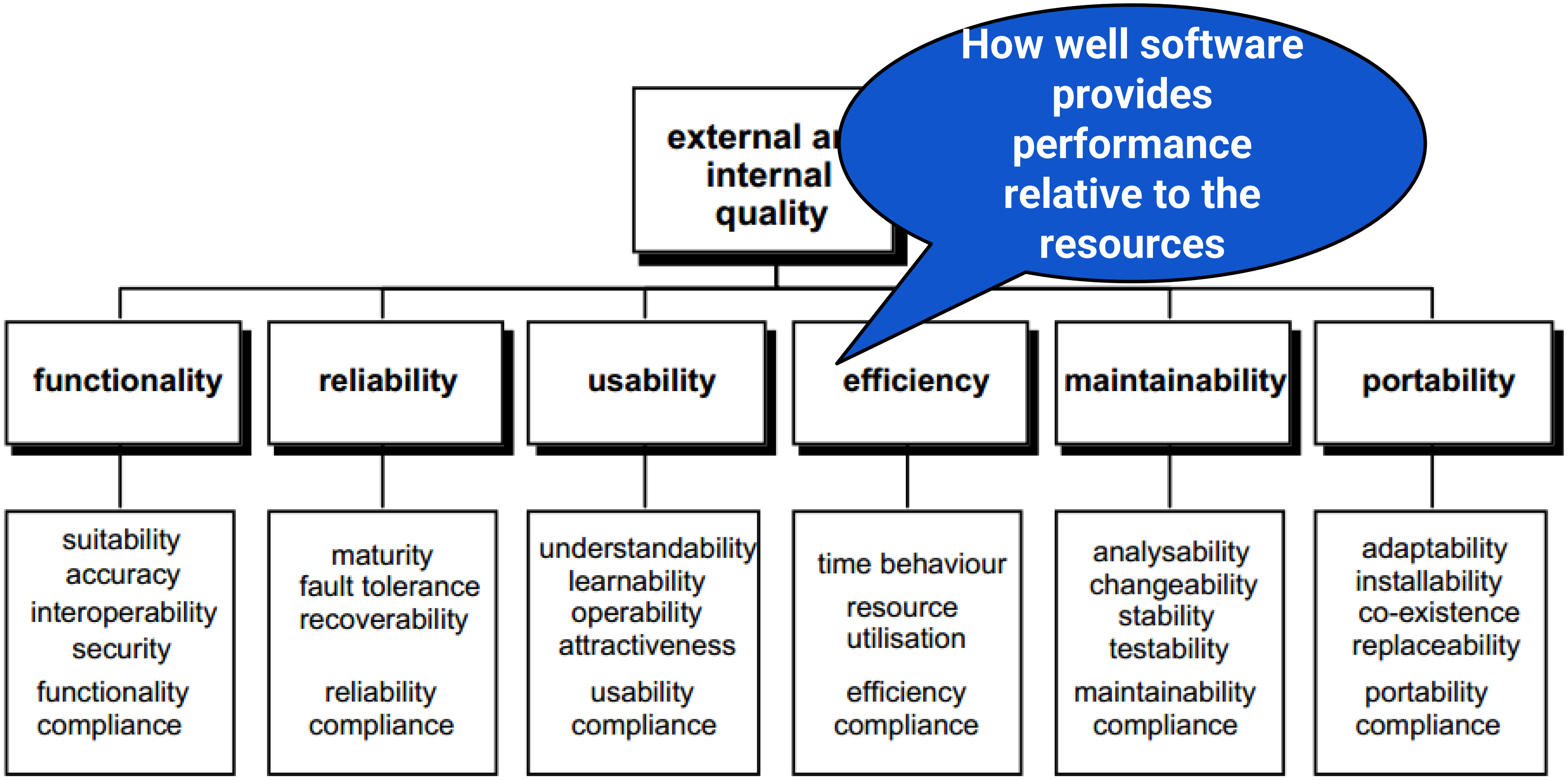
# ISO/IEC 9126-1 Software Quality


How well software maintains the level of performance
functionality
reliability
usability
efficiency
maintainability
portability
suitability accuracy interoperability security functionality compliance
maturity fault tolerance recoverability reliability compliance
understandability learnability operability attractiveness usability compliance
time behaviour resource utilisation efficiency compliance
analysability changeability stability testability maintainability compliance
adaptability installability co-existence replaceability portability compliance

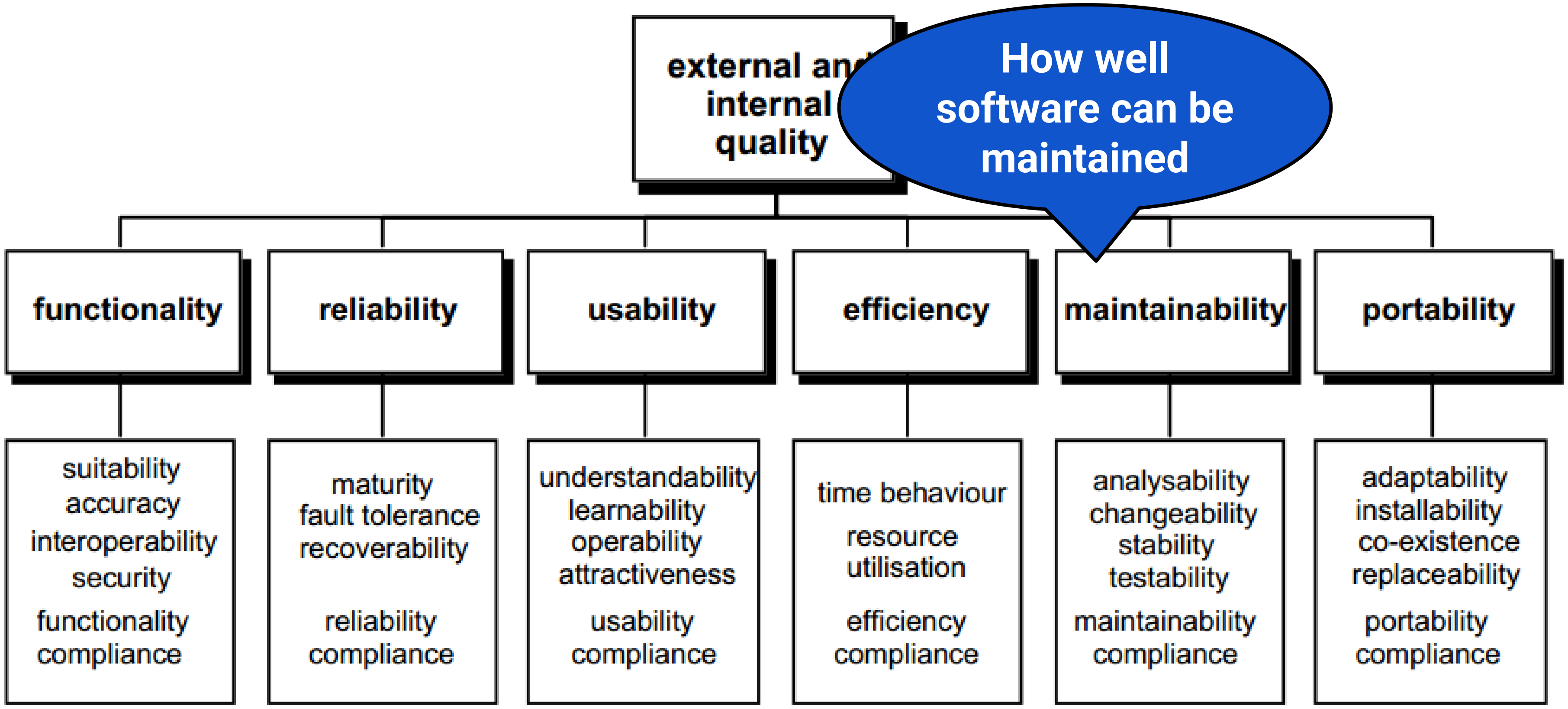
# ISO/IEC 9126-1 Software Quality



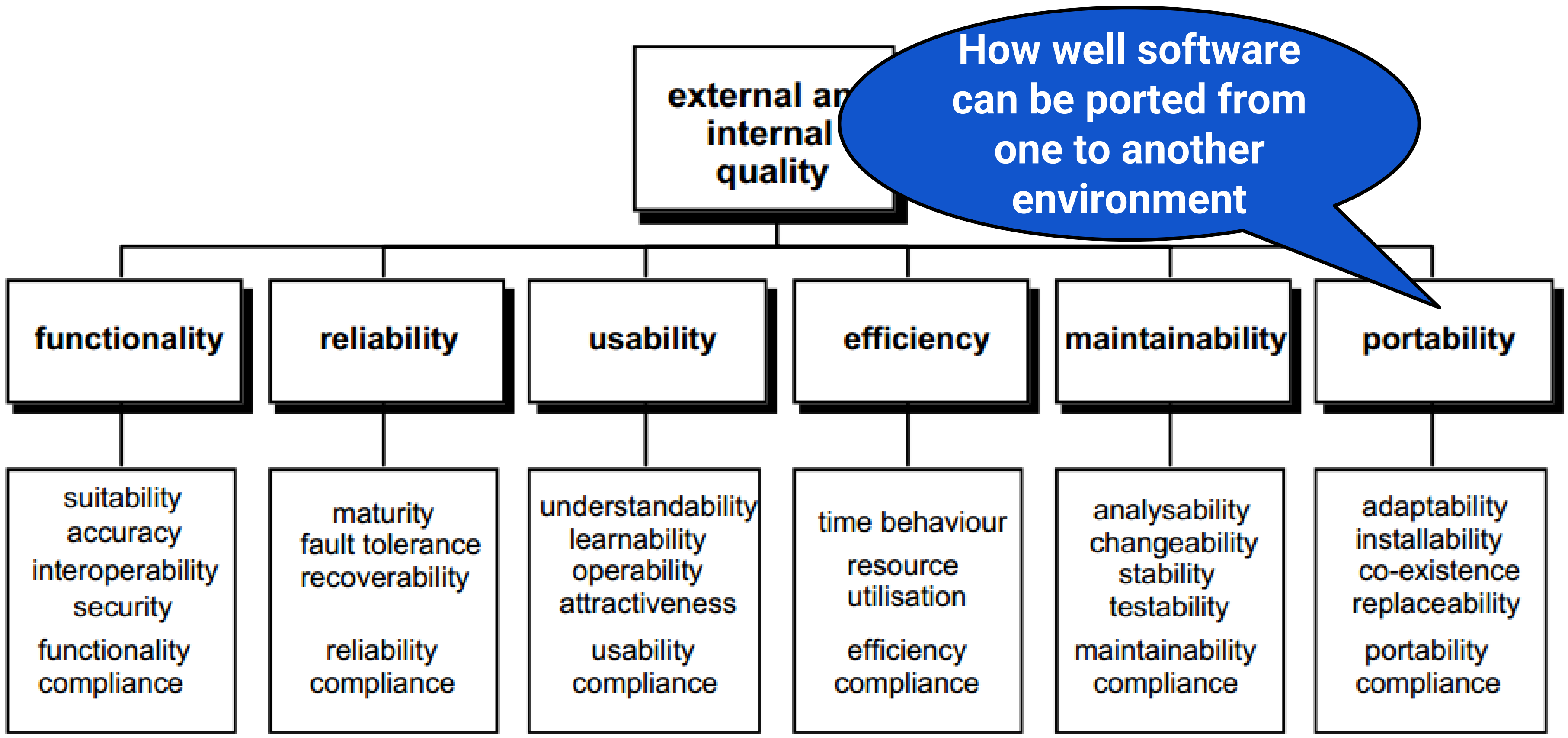
# ISO/IEC 9126-1 Software Quality



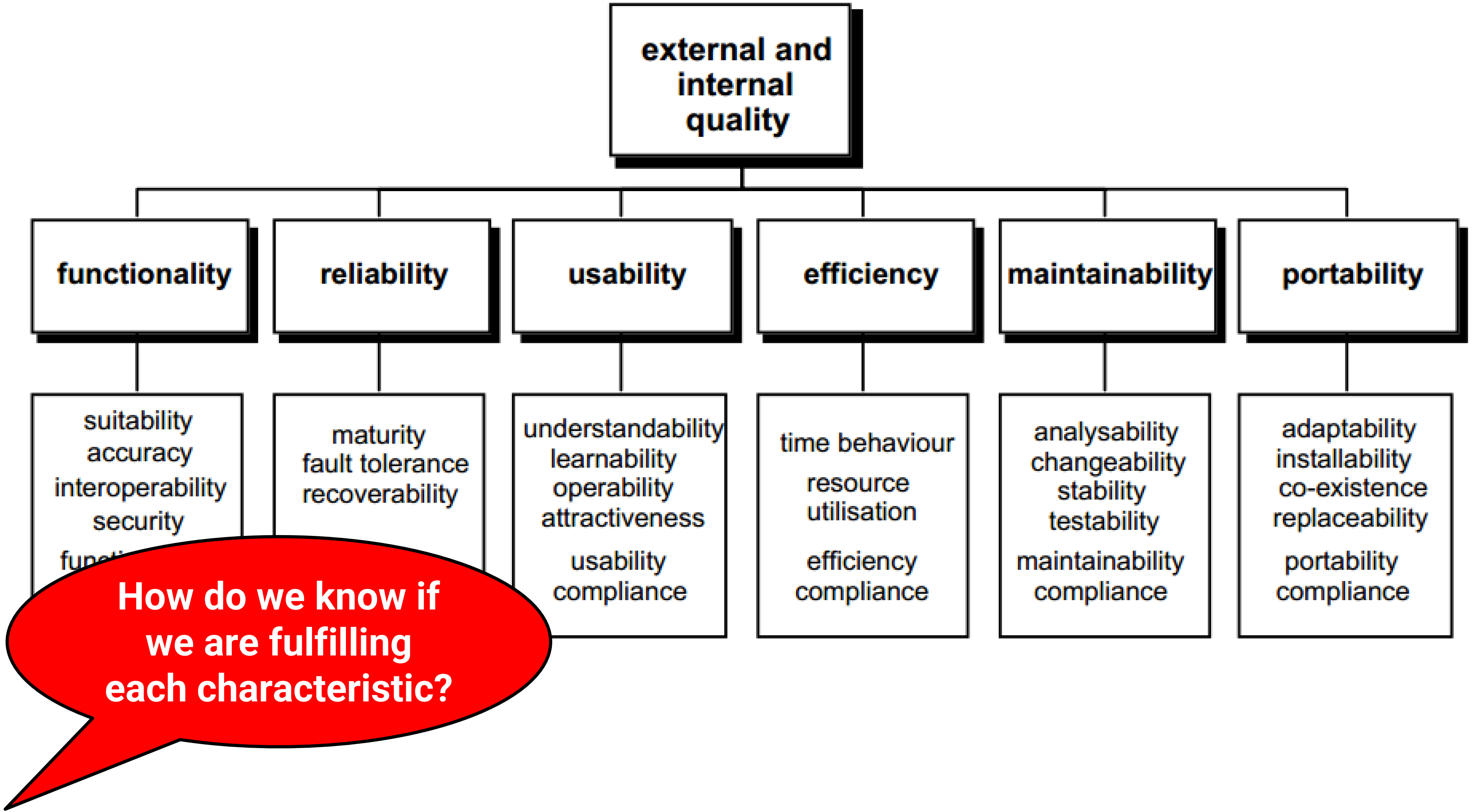
# ISO/IEC 9126-1 Software Quality



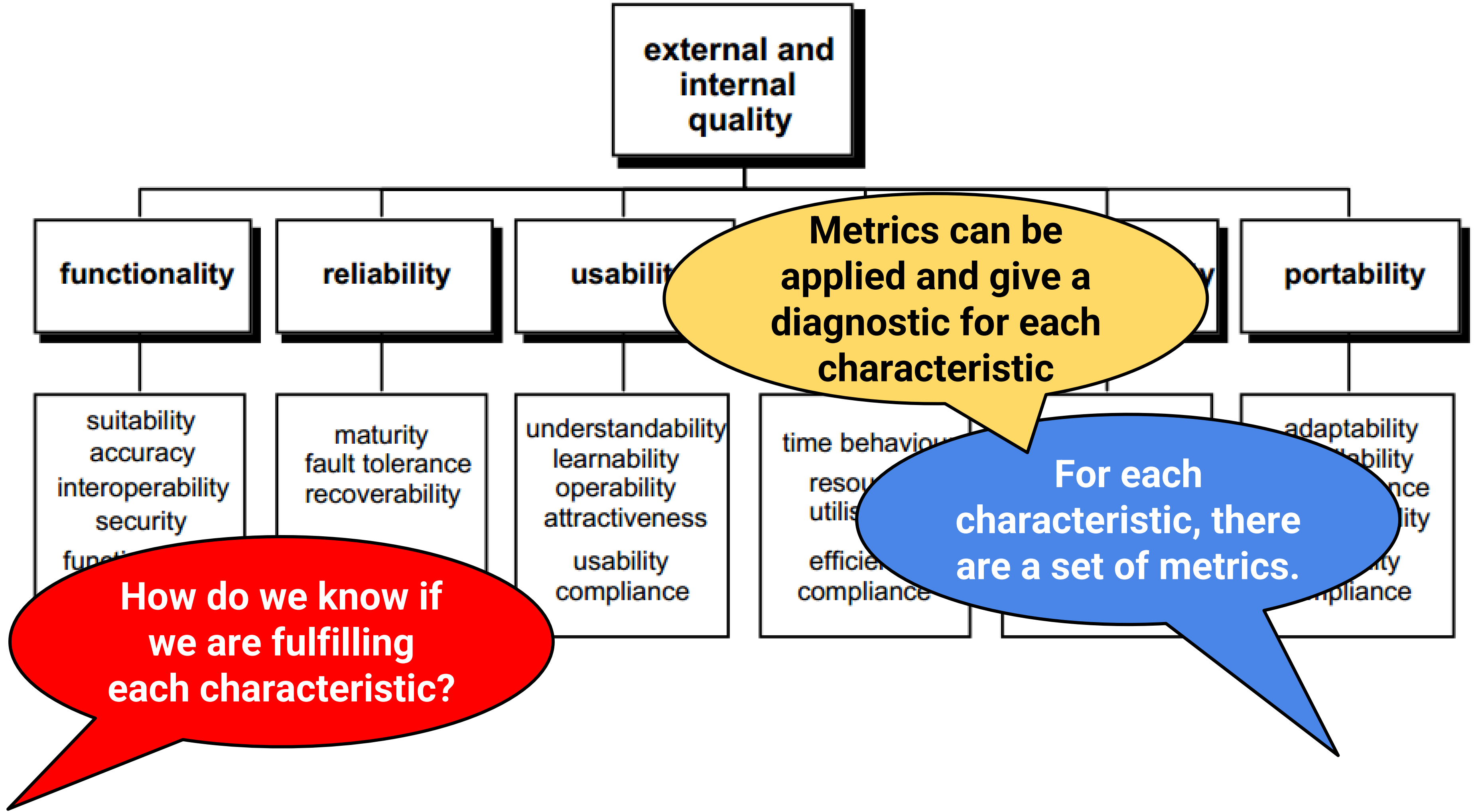
# ISO/IEC 9126-1 Software Quality



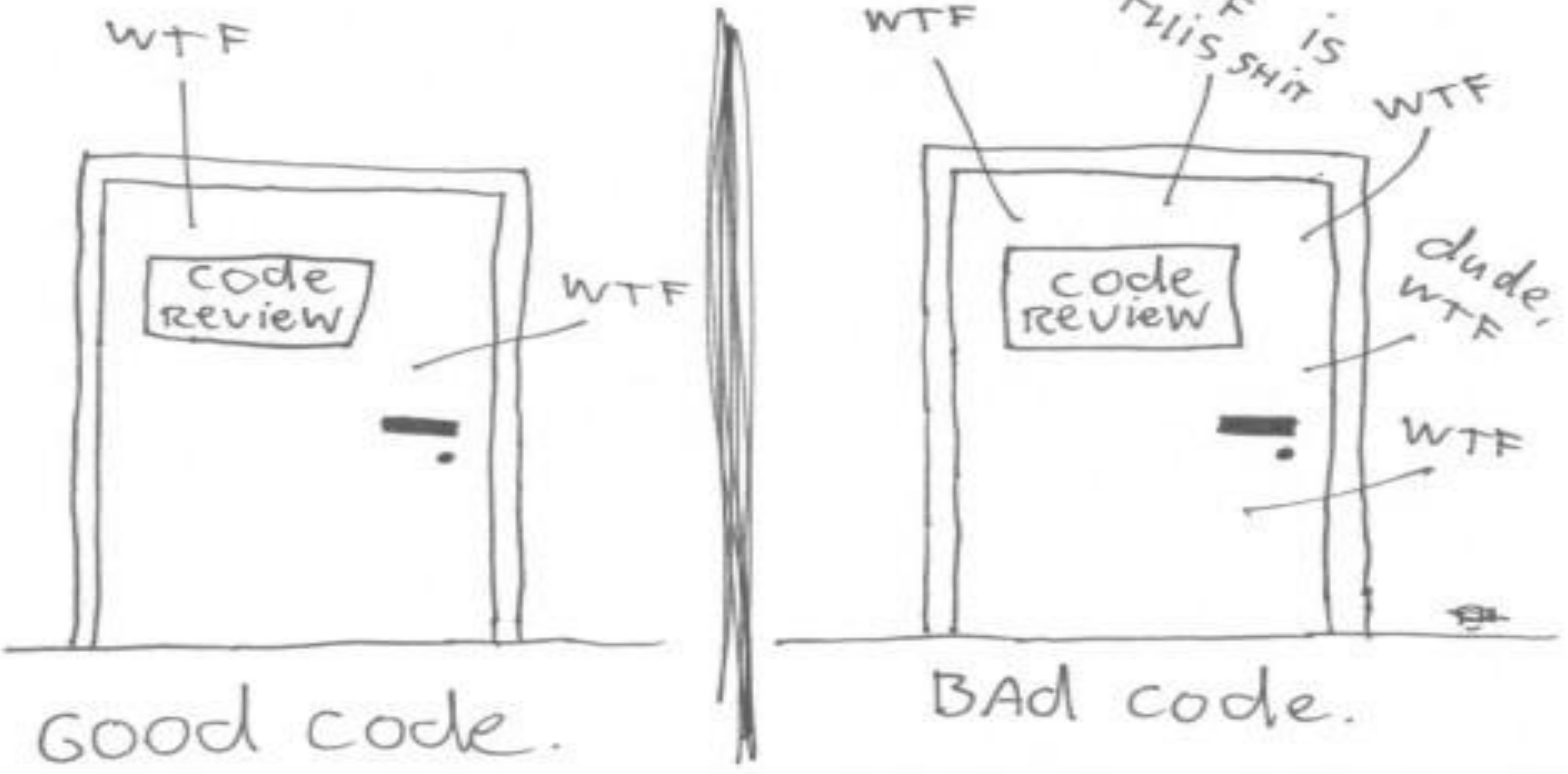
# ISO/IEC 9126-1 Software Quality

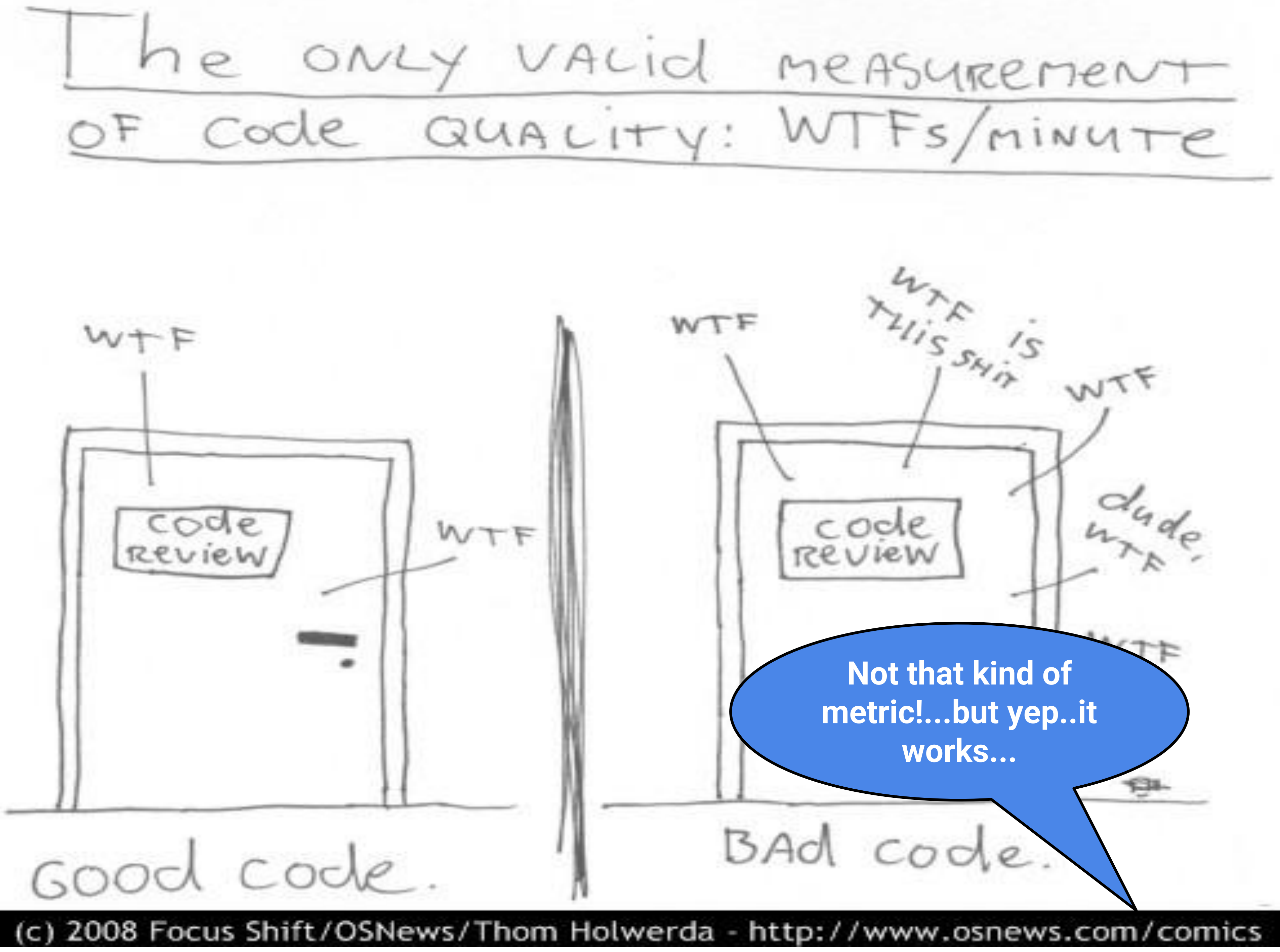


# ISO/IEC 9126-1 Software Quality



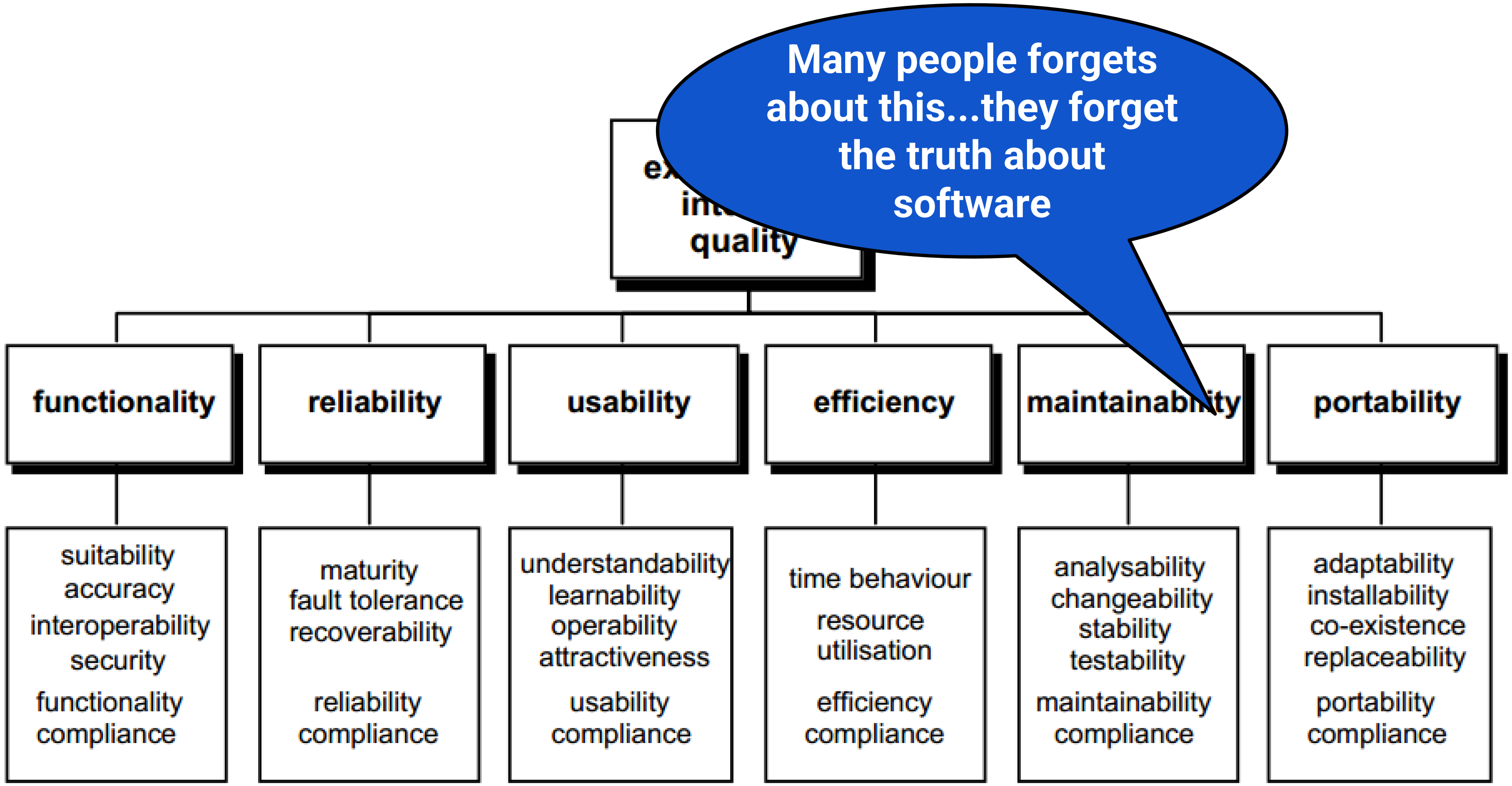
# The only valid measurement of code quality: WTFs/minute




The only valid measurement of code quality: WTFs/minute
WTF
code review
WTF
Good code.
WTF
WTF is this shit
WTF
code review
dude, WTF
WTF
BAd code.
(c) 2008 Focus Shift/OSNews/Thom Holwerda - http://www.osnews.com/comics

Not that kind of metric!...but yep..it works...

# ISO/IEC 9126-1 Software Quality



## The truth about software...

➔ A new system is most of the time:
- Easy to control
- Have a clean and neat design
- Most of the people involved have a clear understanding of the code, components and the whole system
- Defects are solved very quick
- Nobody is afraid of the system!

## The truth about software...

➔ As time goes by:
- ◆ A lot of hot fixes
- ◆ Running to meet unrealistic schedules
- ◆ Bad design decisions are made
- ◆ Programmers code just to finish their work ASAP
- ◆ Software starts degrading

## The truth about software...

➔ Things change:

- ◆ High complexity
- ◆ Low maintainability
- ◆ Changes are expensive, risky and slow
- ◆ The system is a mystery to the programmers
- ◆ Nobody wants to touch them, only a few brave ones

## The truth about software...

➔ What can we do to avoid software to become a terrorific monster?

➔ Software is a living being. Don't forget that!

➔ **Aiming for maintainability** can be a good shield against evil software!

## The truth about software...

- ➔ Software easily "rots"
- ➔ Avoiding the rotting is very hard, but creating software that is less prone to rotting is not that hard.
- ➔ We'll talk more about this later...now testing!

## Testing

- ➔ Is the process of executing a program with the **intent** of finding errors

- ➔ Sometimes testing is the last part of a large development process, and sometimes the time for testing is took to finish coding instead

## Testing

➔ People often avoid testing thinking that is too expensive in time and money

➔ If you don’t test your code, you are leaving potential errors that are going to be even more expensive

➔ The mindset should be: invest money and time now, instead of investing **even more money and time later**!

## Testing

- ➔ Test-lazy people can find a great help following Test-Driven Development
  - ◆ Code tests first even before the real code

- ➔ There are two main types of tests:
  - ◆ Black box
  - ◆ White box

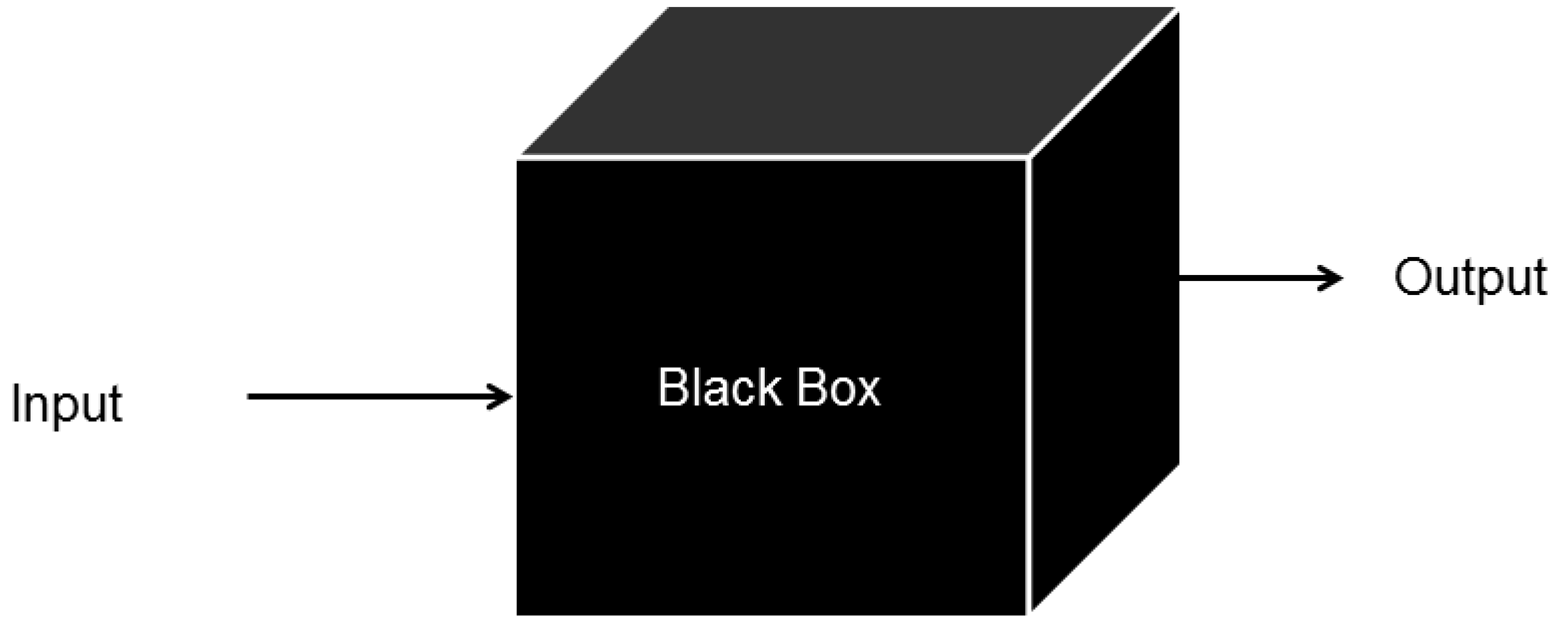
## Black box testing

➔ View the program as a black box: you don’t know what is inside

➔ Your goal is be c**ompletely unconcerned about the internal** behavior and structure of the program

➔ Concentrate on finding circumstances in which the program does not behave according to the specifications

## Black box testing



*Internal behavior of the code is unknown*

## Black box testing

➔ If I don’t know what is inside, how do I know how to test it?

## Black box testing

- ➔ Test data is designed based on the specification of the program

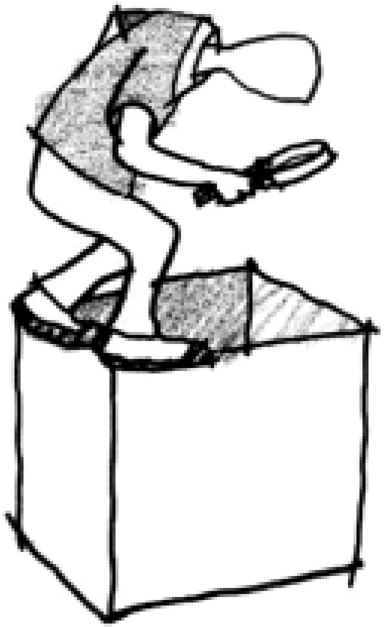
- ➔ To find errors in the program the criterion is **exhaustive input testing**
  - ◆ Make use of every possible input
  - ◆ This is almost impossible
  - ◆ Focus on maximizing the number of errors found with a finite set of test cases

## White box testing

➔ Also called logic-driven testing

➔ Permits you to examine the **internal structure** of the program

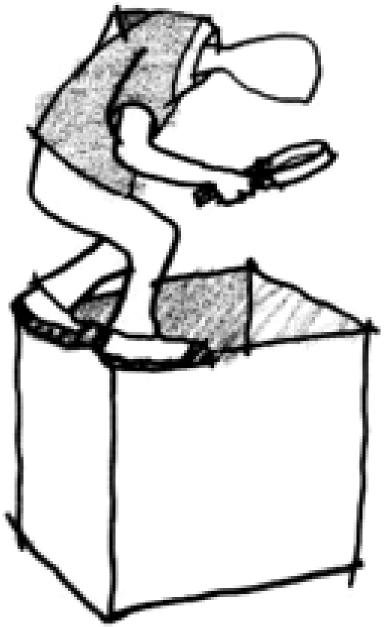
➔ Derives the test data from an examination of the program's logic and structure



## White box testing

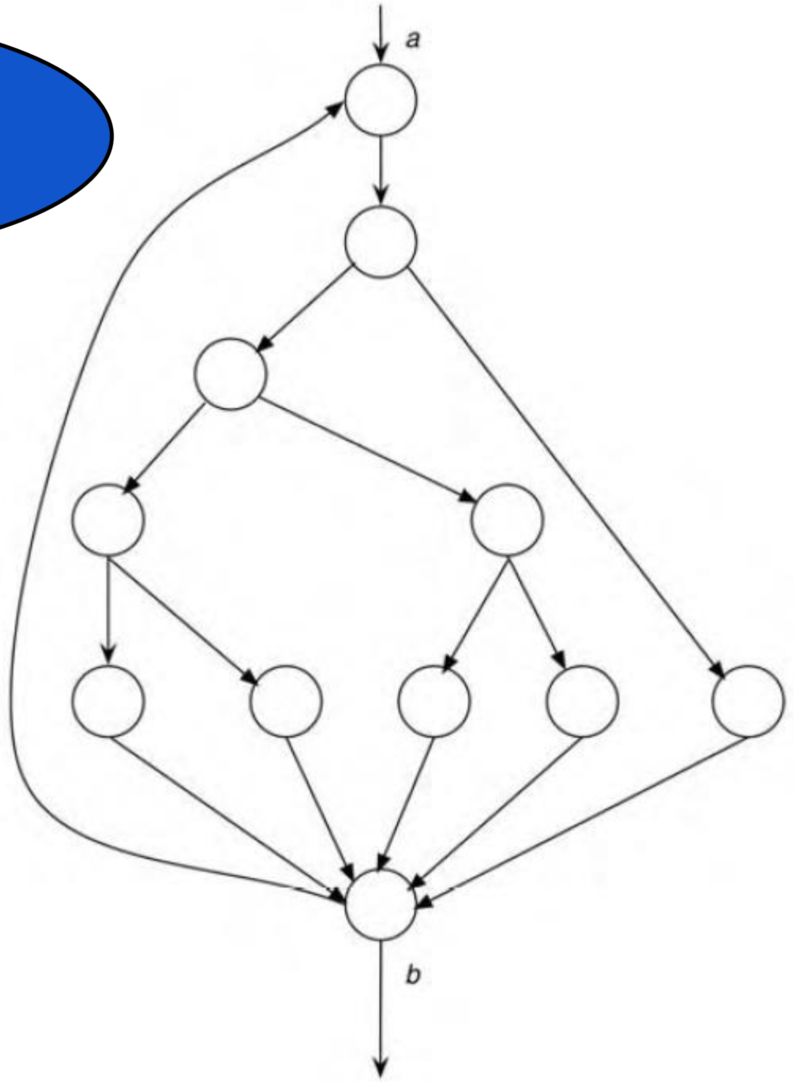
- ➔ Focus on **executing most** (if not all) **of the code** at least once

- ➔ Test all possible paths of control flow of the program



## White box testing

High complexity can make testing virtually impossible



## Testing

➔ Inside this white/black box categories, there are many types of tests:

- Installation
- Compatibility
- Regression
- Acceptance
- Alpha
- Beta
- Functional
- Accessibility
- Security

## Testing

- ➔ Also there are different levels to which the tests are applied:
  - ◆ Unit testing
  - ◆ Integration testing
  - ◆ Component interface testing
  - ◆ System testing
- ➔ Sometimes the line between these definitions becomes blurry

## Unit Testing

- ➔ Unit testing is all about white box. Is focused on testing a module of the software
- ➔ Isn't a new concept, is around since the 70s
- ➔ Is a proven technique to improve the quality of the code while gaining deeper knowledge of the functional requirements

## Unit Testing

➔ An unit test is:

- **A piece of code** (usually a method) that **invokes** another piece and checks the correctness of some assumptions afterward
- If the **assumptions** turn out to be wrong the unit test has failed
- A "unit" is a method or function

## Unit Testing

➔ A good unit test should be:

- ◆ Automated and repeatable
- ◆ Easy to implement
- ◆ Once it is written, it should remain for future use
- ◆ **Anyone** should be able to run it
- ◆ It should run at the push of a button
- ◆ It should run quickly

**If lacks one of these, then it is not an unit test**

## Unit Testing

```
public class SimpleParser
    {
        public int ParseAndSum(string numbers)
        {
            if(numbers.Length==0)
            {
                return 0;
            }
            if(!numbers.Contains(","))
            {
                return int.Parse(numbers);
            }
            else
            {
                throw new InvalidOperationException(
"I can only handle 0 or 1 numbers for now!");
            }
        }
    }
```

## Unit Testing

- ➔ There is a better way to code unit tests
- ➔ xUnit is a collection of unit testing frameworks that derives Smalltalk's SUnit
- ➔ For C++ there are many xUnit frameworks. For now we will use CppUnit

## Going back to maintainability

➔ We talk before how the software "rots" and that it happen very easily and without even notice it

➔ But there are some "smells" that can let you know when the software is starting to "rot"

## Design "smells"

- ➔ **Rigidity**: tendency of software to be difficult to change, even in simple ways

- ➔ **Fragility**: tendency of a program to break in many places when a single change is made

## Design "smells"

- ➔ **Immobility**: parts of the system could be reused, but is very hard and risky to separate those parts
- ➔ **Viscosity**: doing the right thing is harder than doing things wrong. Feel more like hacking than coding

## Design "smells"

- ➔ **Needless complexity**: contains elements that are not currently useful. A lot of nice features that nobody uses
- ➔ **Needless repetition**: a lot of repeating structures that could be in just one place

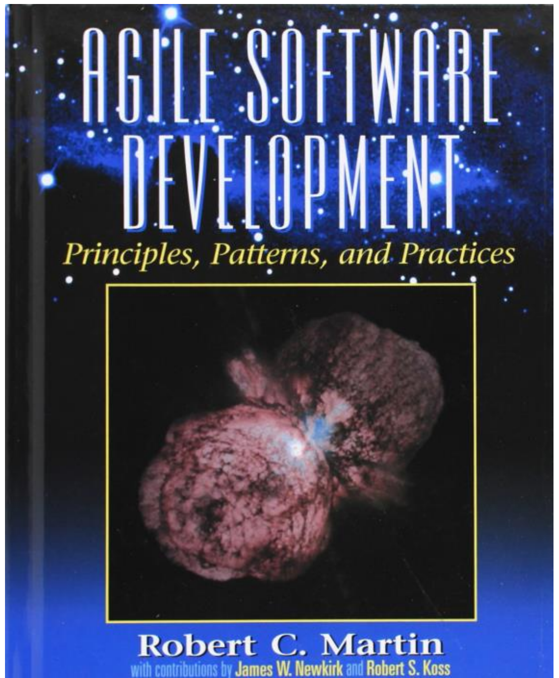
## Design "smells"

➔ **Opacity:** code is really hard to understand and don't express its real intention.

## How to avoid design "smells"

- ➔ There are a set of design principles that sets the bases prolong the life of the software without "rotting"

- ➔ That will be your homework...


AGILE SOFTWARE DEVELOPMENT
Principles, Patterns, and Practices
Robert C. Martin
with contributions by James W. Newkirk and Robert S. Koss

## Geek Corner

"Fun" facts for geeks

Today's fact:

## Framework vs Toolkit

➔ Those two terms are used a lot in the software development environment

➔ Sometimes people used them interchangeable, or sometimes people have a hard time understanding the difference

➔ Actually is kind of simple

Geek Corner

Today's fact:

## Framework vs Toolkit

➔ Let's put it in a simple way:

When you use a toolkit, **you write the main body** of the application **and call the code of the toolkit** that you want to use

When you use a framework, **you reuse the main body and write the code it calls**. You'll have to follow some code/naming conventions and the framework will call your code.

Geek Corner

Today's fact:

## Framework vs Toolkit

- ➔ A toolkit is a set or related and reusable classes designed to provide useful, general-purpose functionality
  - ◆ Don't impose a particular design to your application
  - ◆ They just provide the functionalities and you decide if you want to use it

- ➔ Examples of toolkits:
  - ◆ QT
  - ◆ GTK
  - ◆ Java Swing, AWT
  - ◆ Dojo

Geek Corner

Today's fact:

## Framework vs Toolkit

- ➔ A framework is a set of cooperating classes that make up a reusable design for a specific class of software
- ➔ They dictate the architecture of your application and defines all the design patterns that you have to follow so you can concentrate on the specifics of your application
- ➔ Design reuse over code reuse.
- ➔ Examples: AngularJS, .NET, Eclipse RPC, xUnit, Spring, Struts, Rails, Django...

# UML, Patterns and Software Quality

CE-2101 Algorithms and Data Structures