National University
Of Computer and Emerging Sciences

Software Design and Architecture

Object Oriented Design
GoF Patterns

Sajid Anwer

Department of Software Engineering,
FAST-NUCES, CFD Campus

Lecture Outline

- Design Patterns Fundamentals
- GoF Patterns
  - » Creational
  - » Structural
  - » Behavioral

Design Patterns Fundamentals

- Gamma, Helm, Johnson, Vlissides: (Gang of Four, 1995) presented 23 design patterns in 3 categories:
- Provides a *scheme* for refining the subsystems or components of a software system, or the relationships between them.
- Designers *come up with various solutions* to deal with problems.
- Eventually, best known solutions are collected and documented as *Design Patterns*.
- A design pattern is a *generalized solution* to a *commonly occurring problem*".
- There are recurring/characteristic problems during a design activity.
  - » E.g., A certain class should have only one instance.
  - » Known as *Pattern.*

Design Patterns Fundamentals

- A Design Pattern is the outline of a *reusable solution* to a general problem encountered in a particular context:
  - » describes a recurring problem
  - » describes the core of the solution to that problem.
  - » identifies classes and their roles in the solution to a problem

Design Patterns Fundamentals -- Benefits

- Patterns give a design *common vocabulary* for software design:
  - » Allows engineers to abstract a problem and talk about that abstraction in isolation from its implementation.
  - » A culture; domain-specific patterns increase design speed.
- *Capture expertise* and allow it to be communicated:
  - » Promotes design reuse and avoid mistakes.
  - » Makes it easier for other developers to understand a system.
- *Improve documentation* (less is needed):
  - » Improve understandability (patterns are described well, once).

GoF Patterns Categories

- *Creational: Creation* of objects. Separate the operations of an application from how its objects are created.

- *Structural:* Concern about the *composition* of objects into larger structures. To provide the possibility of future extension in structure.

- *Behavioral:* Define *how objects interact* and how responsibility is distributed among them.
  - » Use inheritance to spread behavior across the subclasses, or aggregation and composition to build complex behavior from simpler components.

GoF Patterns

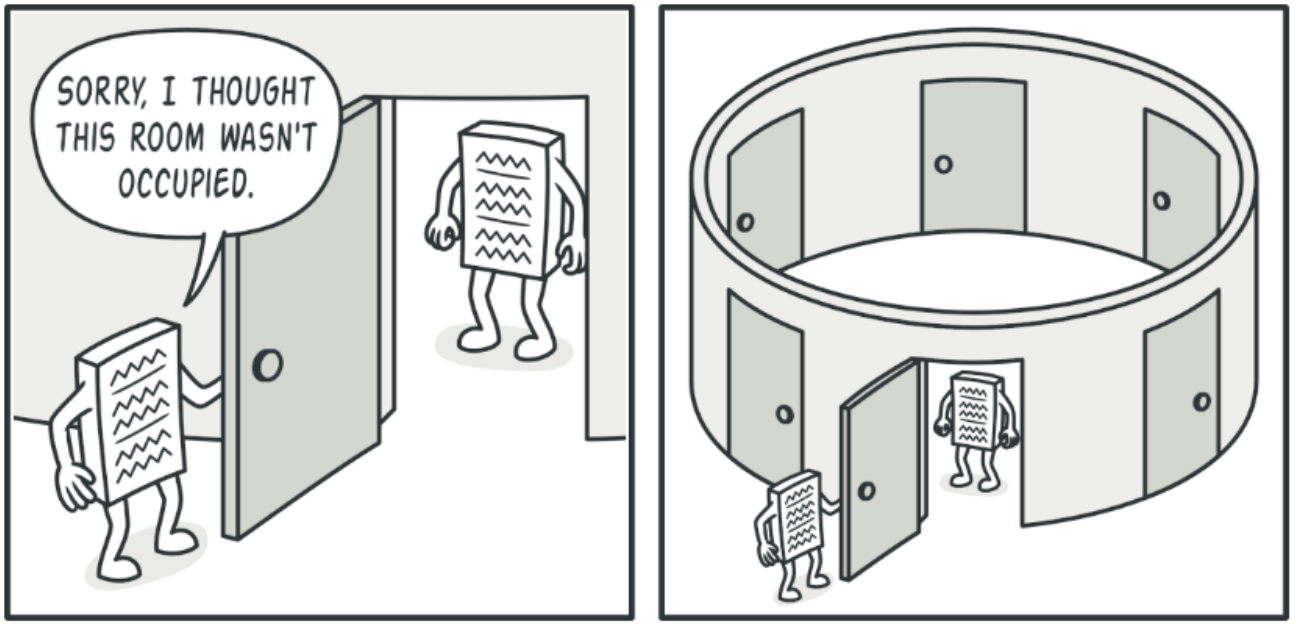
- **Creational Patterns** *(abstracting the object-instantiation process)*
  - » Factory Method
  - » Abstract Factory
  - » Singleton
  - » Builder
  - » Prototype
- **Structural Patterns** *(how objects/classes can be combined)*
  - » Adapter Bridge
  - » Composite
  - » Decorator
  - » Facade
  - » Flyweight
  - » Proxy
- **Behavioral Patterns** *(communication between objects)*
  - » Command
  - » Interpreter
  - » Iterator
  - » Mediator
  - » Observer
  - » State
  - » Strategy Chain of Responsibility
  - » Visitor
  - » Template Method
  - » Memento

GoF Patterns – Description Format

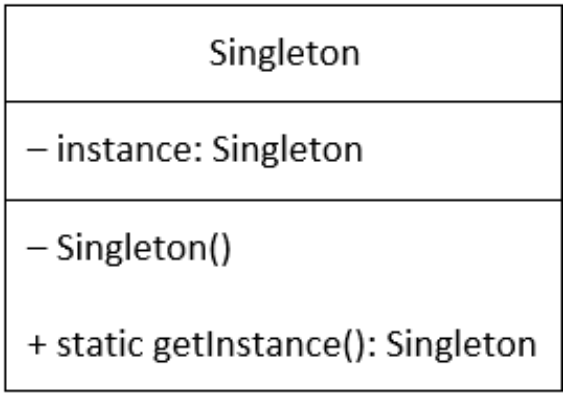
- Context / Name:
  - » The general situation in which the pattern applies.
- Problem:
  - » The main difficulty to be tackled.
  - » Criteria for a good solution.
- Solution:
  - » Recommended way to solve the problem.
- Antipattern (optional):
  - » Erroneous or inferior solution.

GoF Patterns – Creational -- Singleton

- Objective
  - » The objective is to create *only one instance* of a class and to provide only one global access point to that object



GoF Patterns – Creational -- Singleton

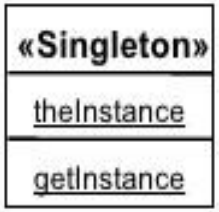


- A private static variable, holding the only instance of the class.
- A private constructor, so it cannot be instantiated anywhere else.
- A public static method, to return the single instance of the class.

GoF Patterns – Creational -- Singleton

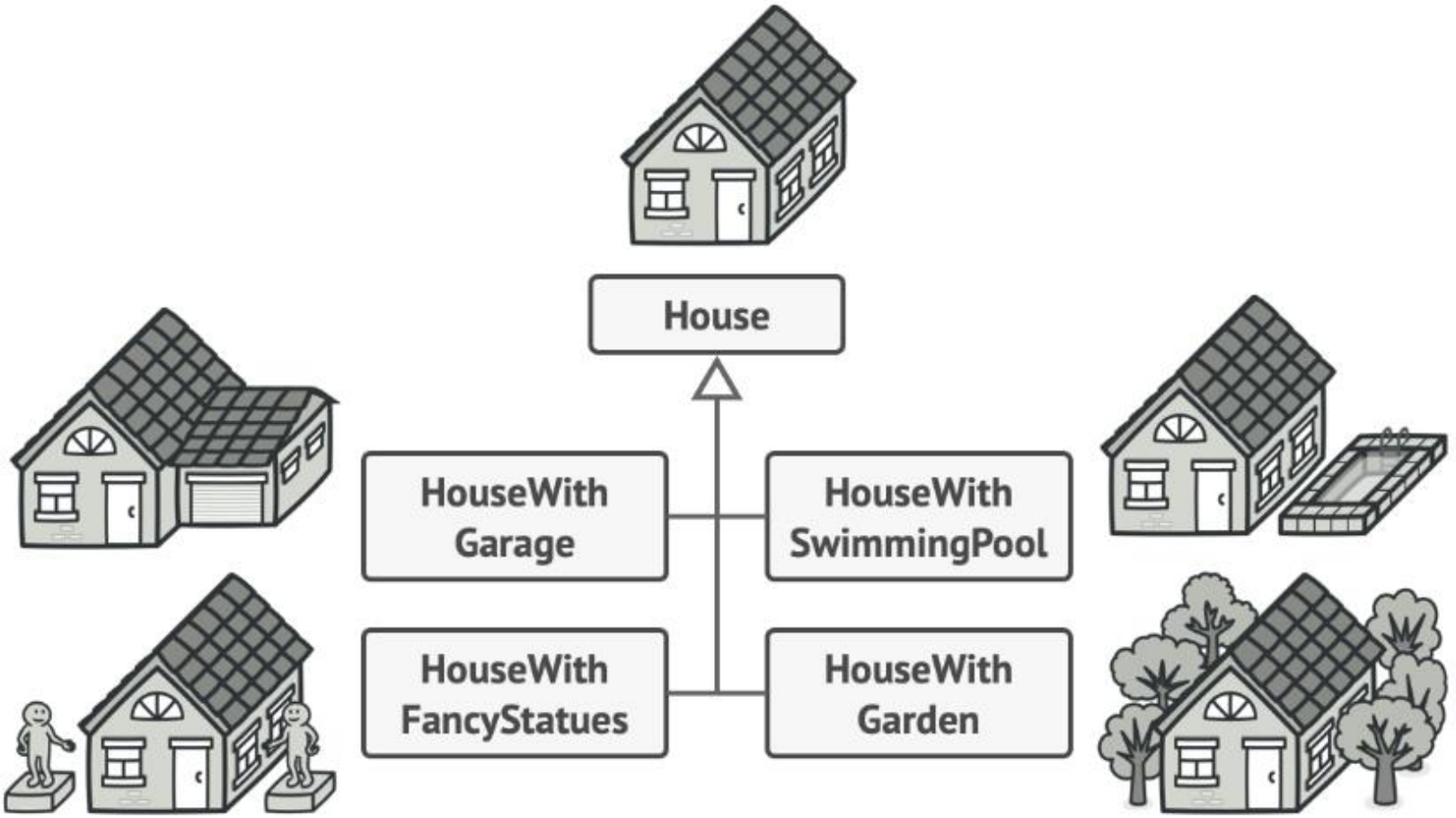
Singleton

- ***Solution:***

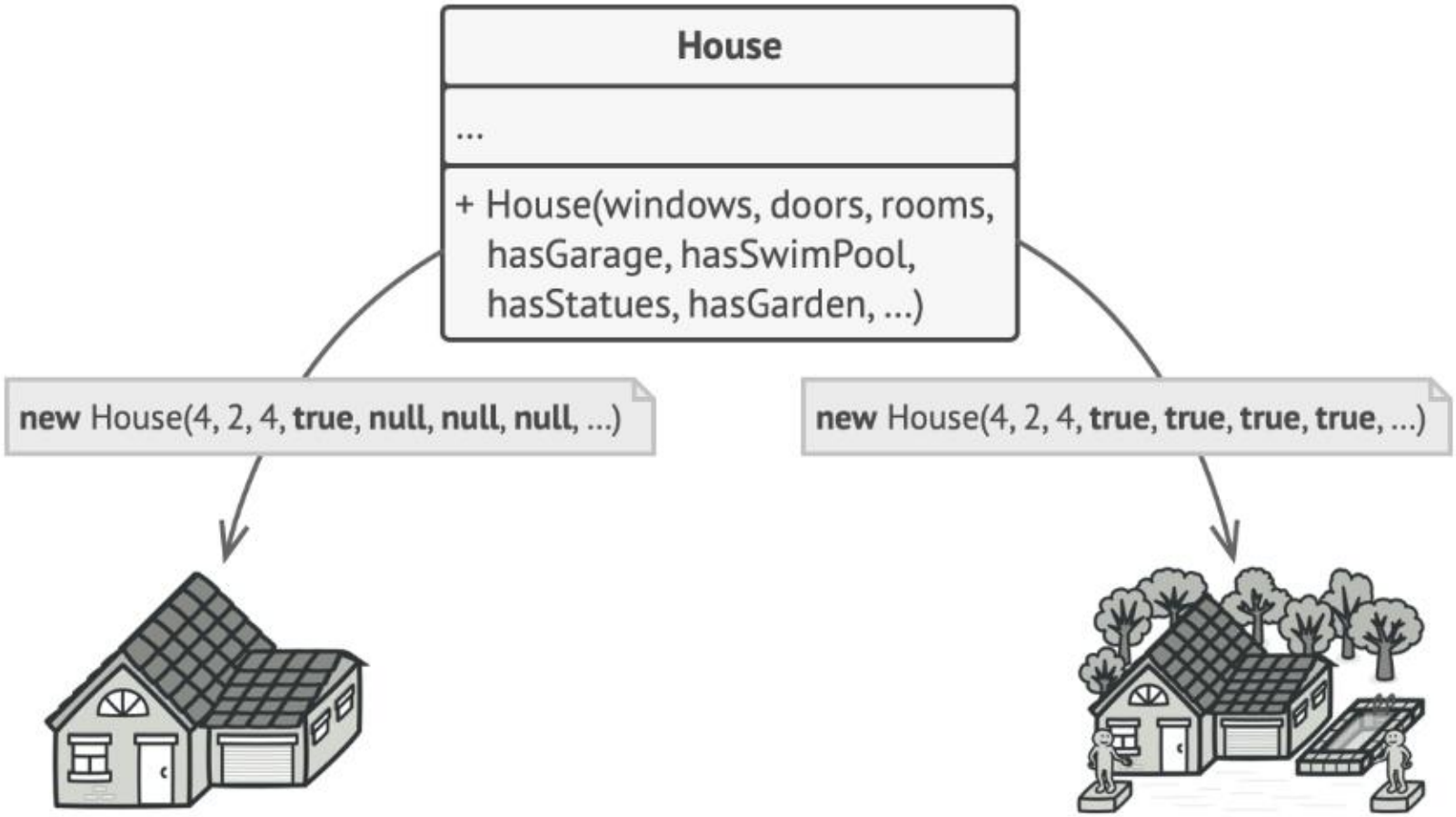



```
public class Singleton {
        static Singleton theInstance
                                = new Singleton();
        private Singleton() { }
        public static Singleton getInstance()  {
                return theInstance;
        }
}
```

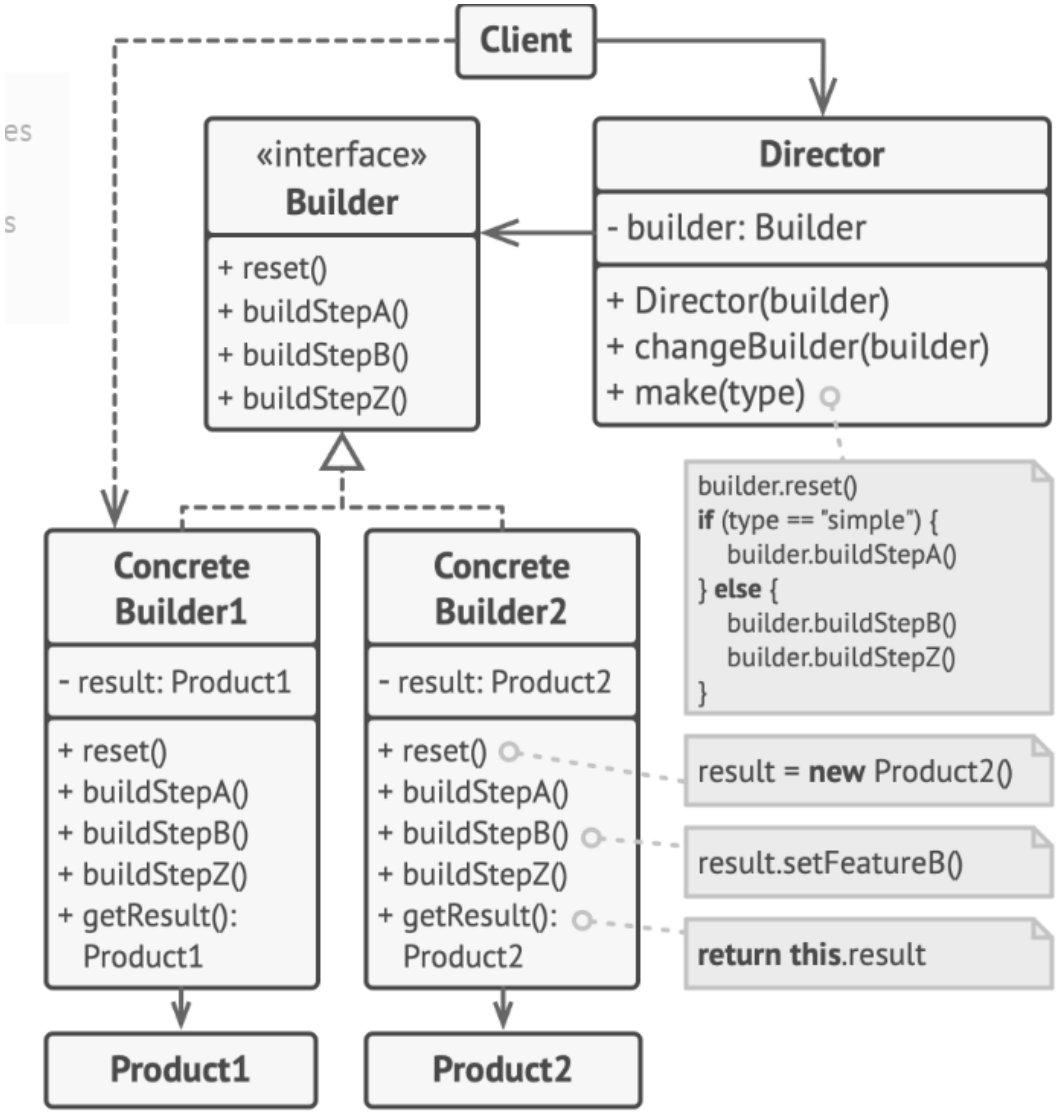
GoF Patterns – Creational -- Builder



GoF Patterns – Creational -- Builder

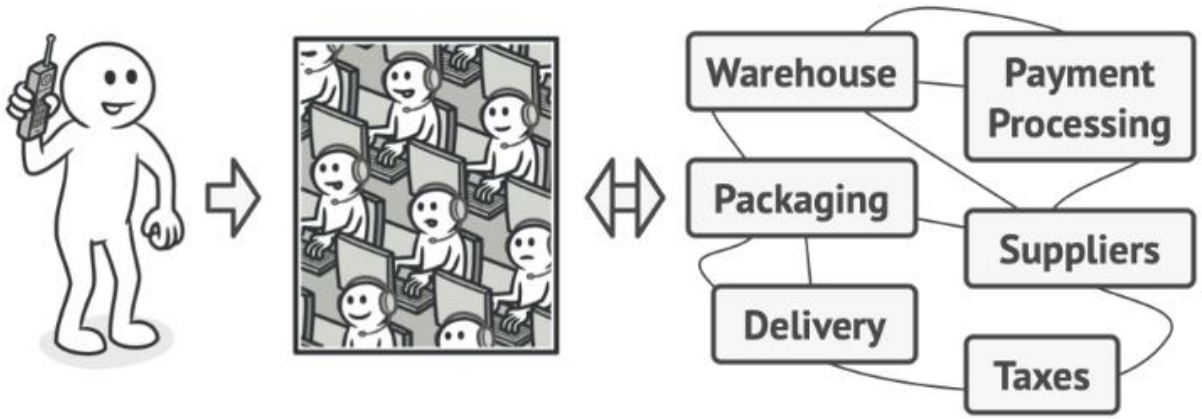


GoF Patterns – Creational -- Builder



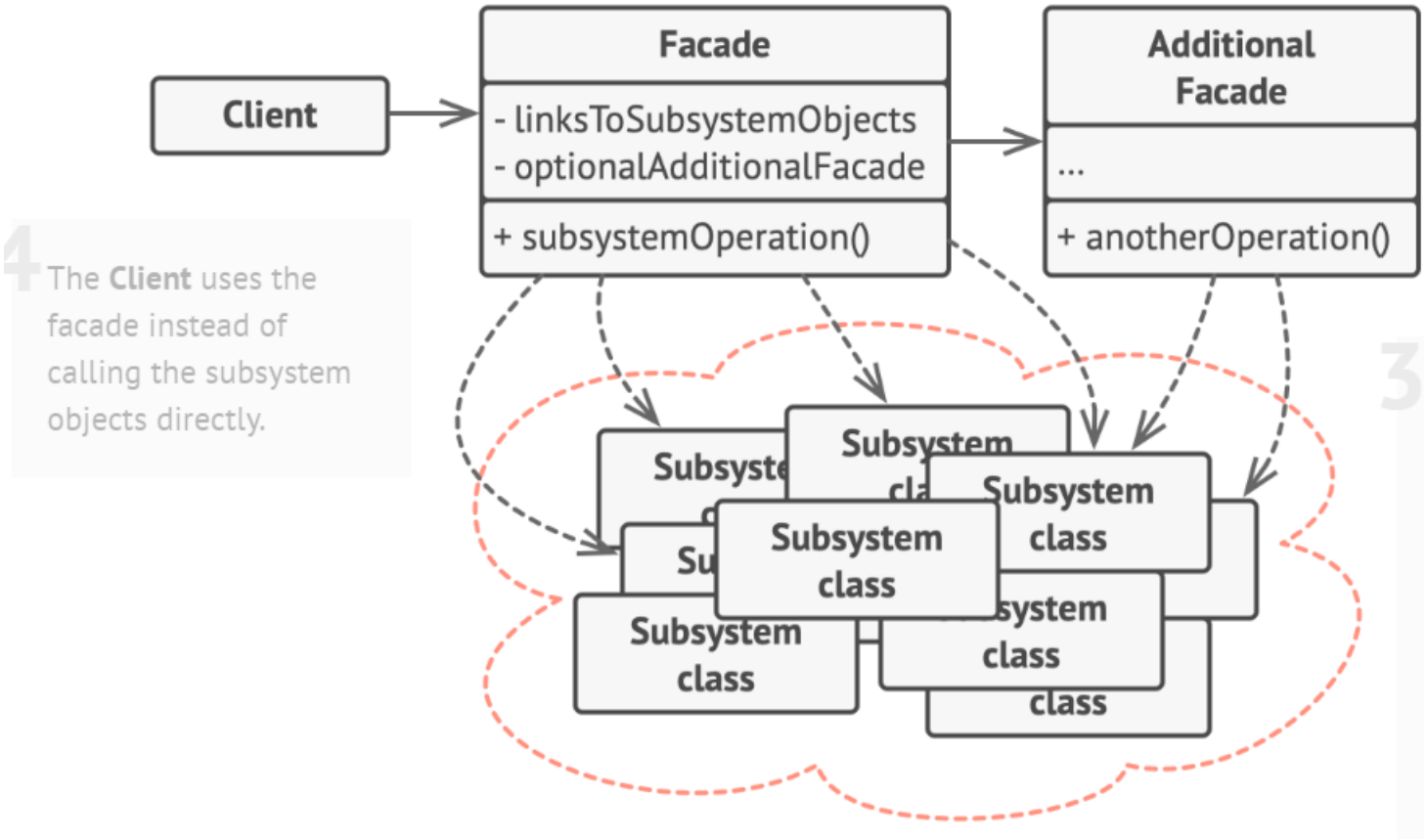
GoF Patterns – Structural -- Facade

- Facade is a structural design pattern that provides a *simplified interface* to a library, a framework, or any other complex set of classes.



Placing orders by phone.

GoF Patterns – Structural -- Facade

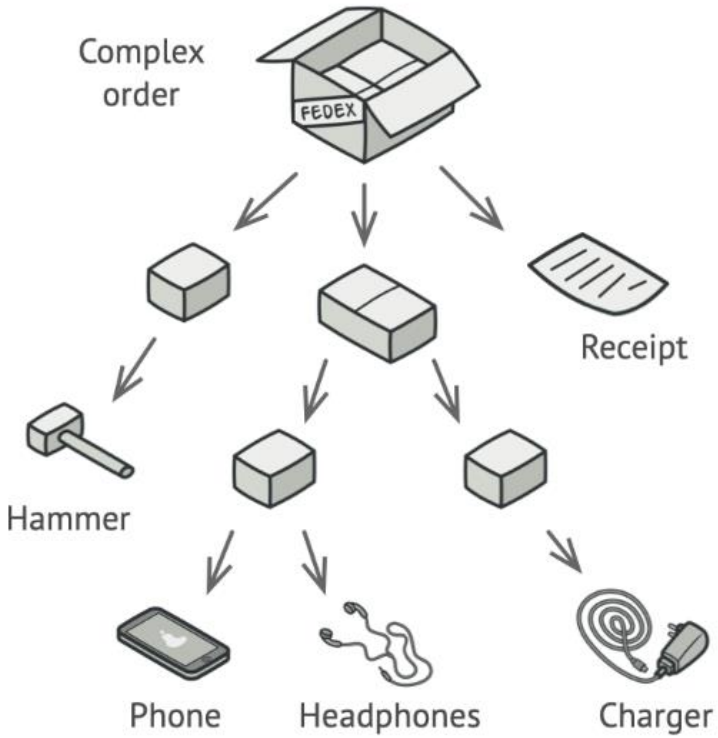


GoF Patterns – Structural -- Facade

- Usually, subsystems *gets complicated* over time.
- The Facade attempts to fix this problem by providing a shortcut to the *most-used features* of the subsystem which fit most client requirements.
- Façade divides the code into layers.
- However, a facade *can* become *a god object* coupled to all classes of an app.

GoF Patterns – Structural -- Composite

- Composite is a structural design pattern that lets you compose objects into *tree structures* and then work with these structures *as if they were individual objects*.

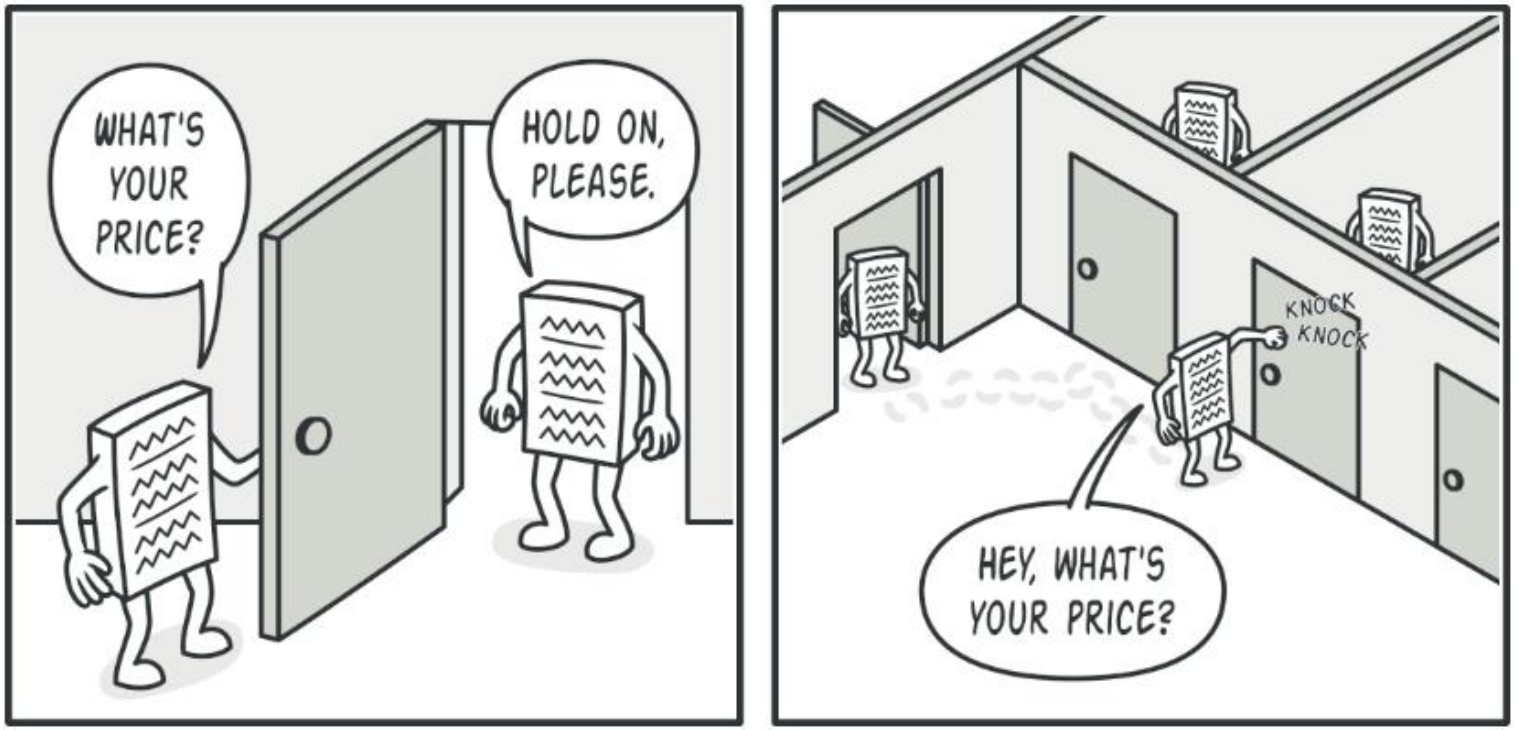


GoF Patterns – Structural -- Composite

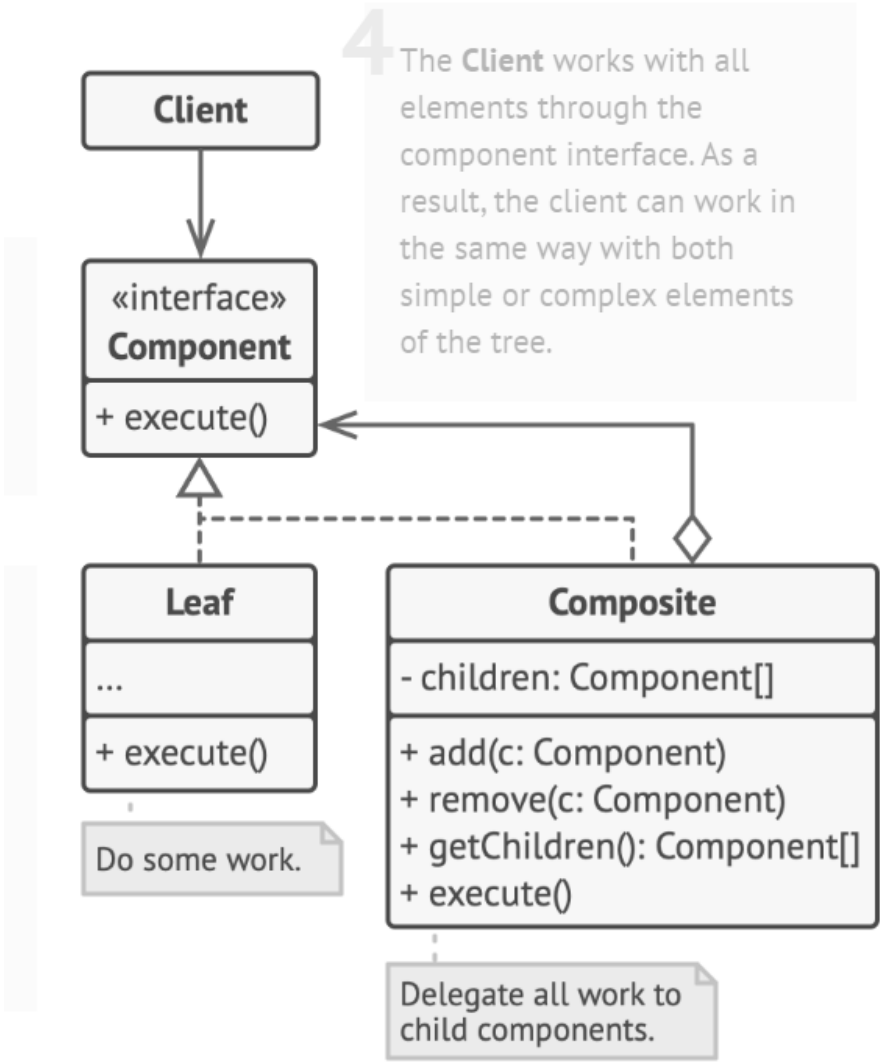
- Direct Approach
  - » unwrap all the boxes, go over all the products and then calculate the total. That would be doable in the real world;
  - » However, but in a program, it's not as simple as running a loop. You have to know the classes of *products and boxes* you're going through, the nesting level of the boxes

- The Composite pattern suggests that you work with products and boxes through a *common interface* which declares a method for calculating the total price.

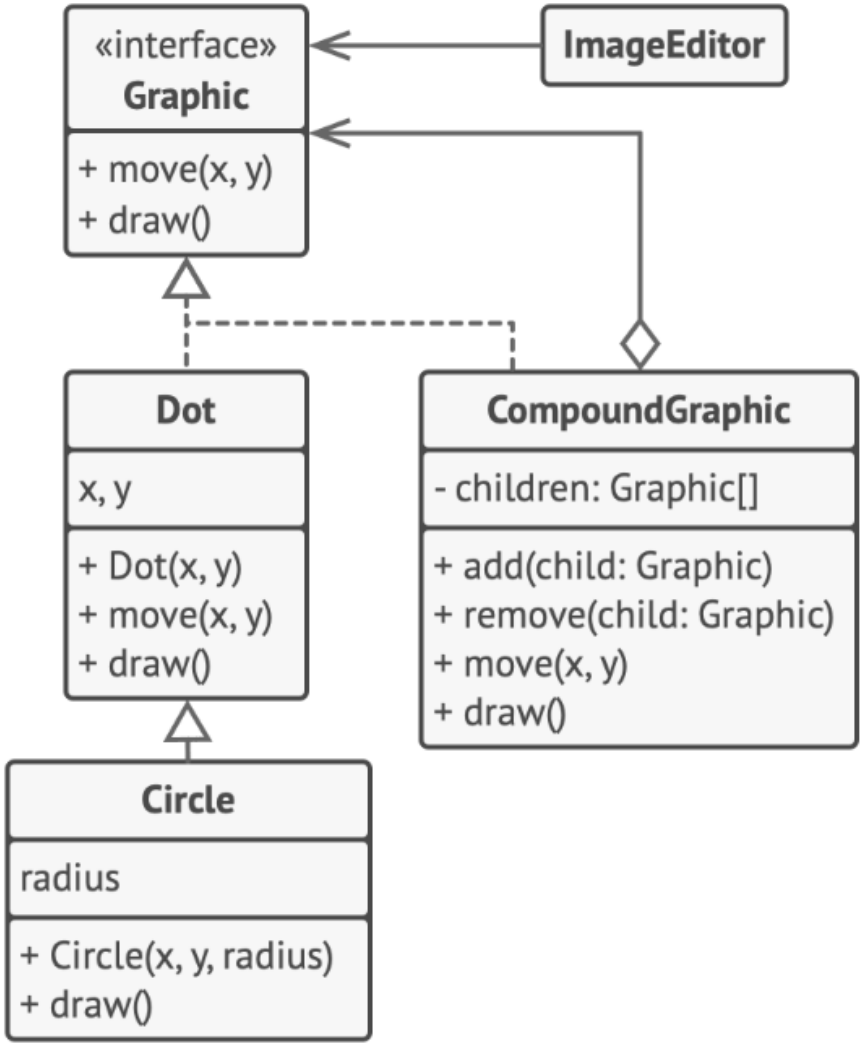
GoF Patterns – Structural -- Composite



GoF Patterns – Structural -- Composite



GoF Patterns – Structural -- Composite



GoF Patterns – Structural -- Composite

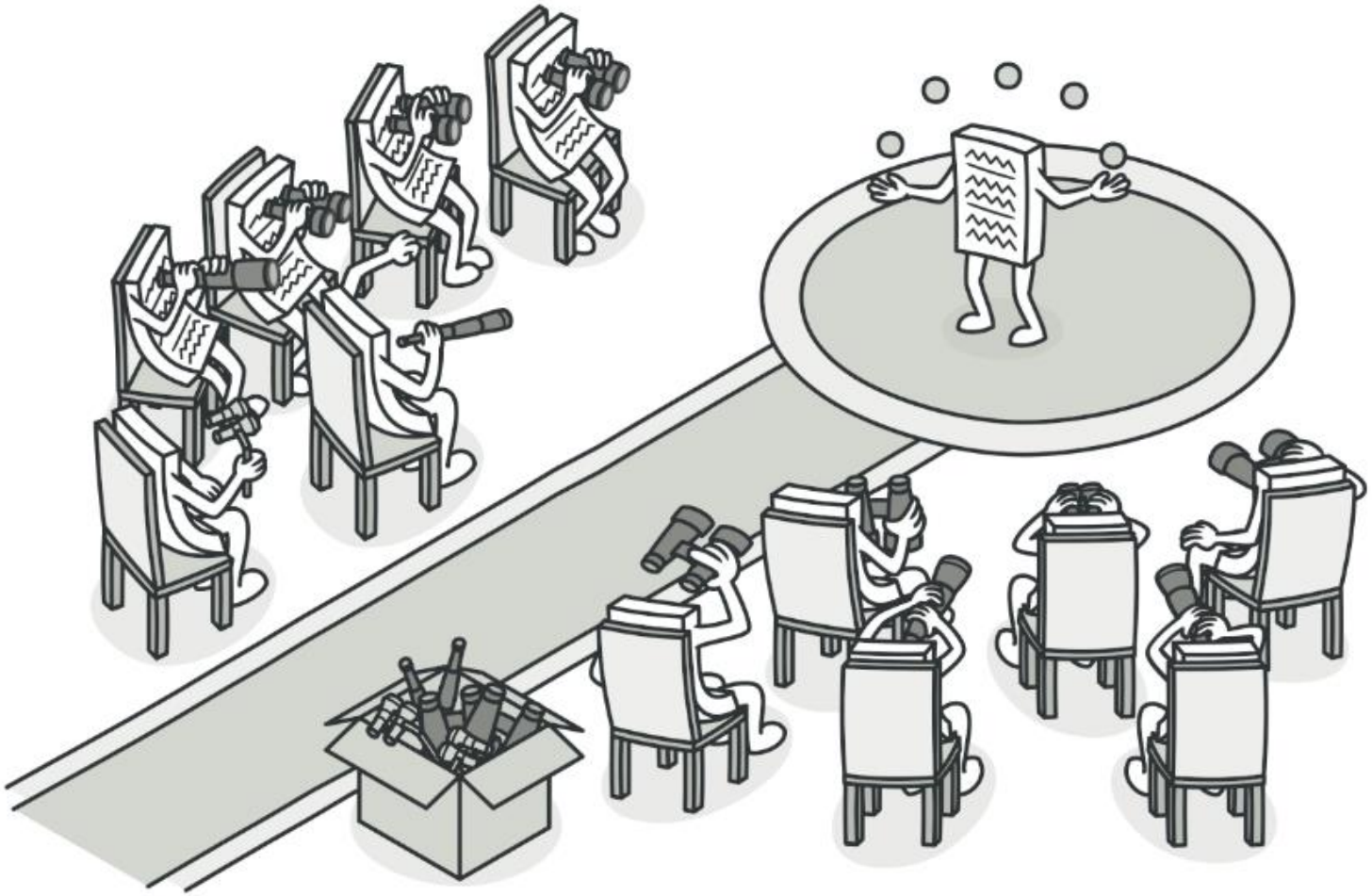
- You can work with complex tree structures more conveniently: use *polymorphism* and *recursion* to your advantage.

- *Open/Closed Principle*. You can introduce new element types into the app without breaking the existing code, which now works with the object tree.
-

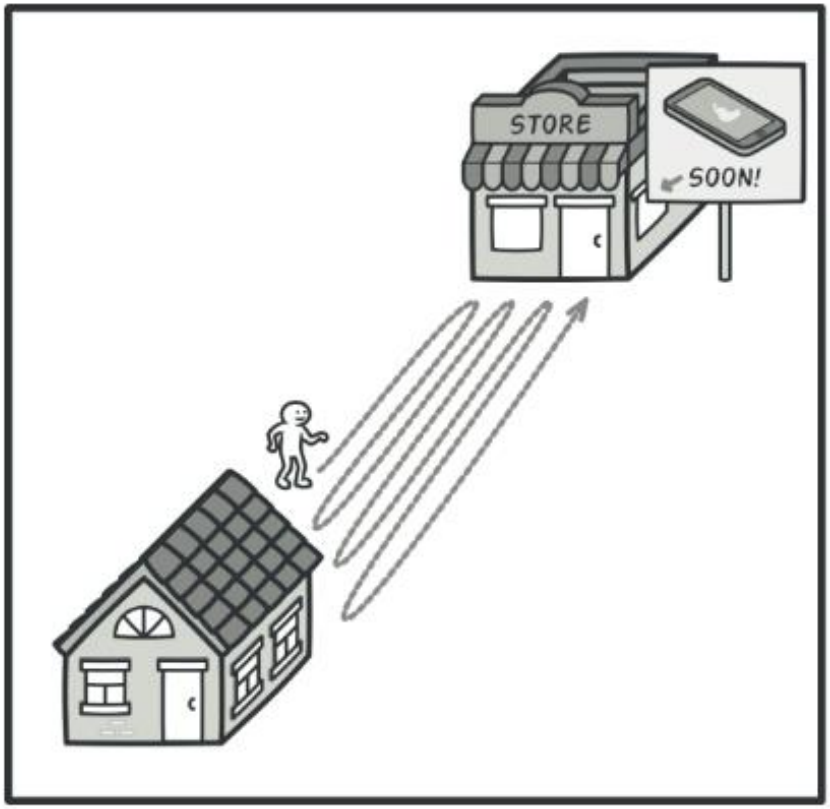
GoF Patterns – Behavioral -- Observer

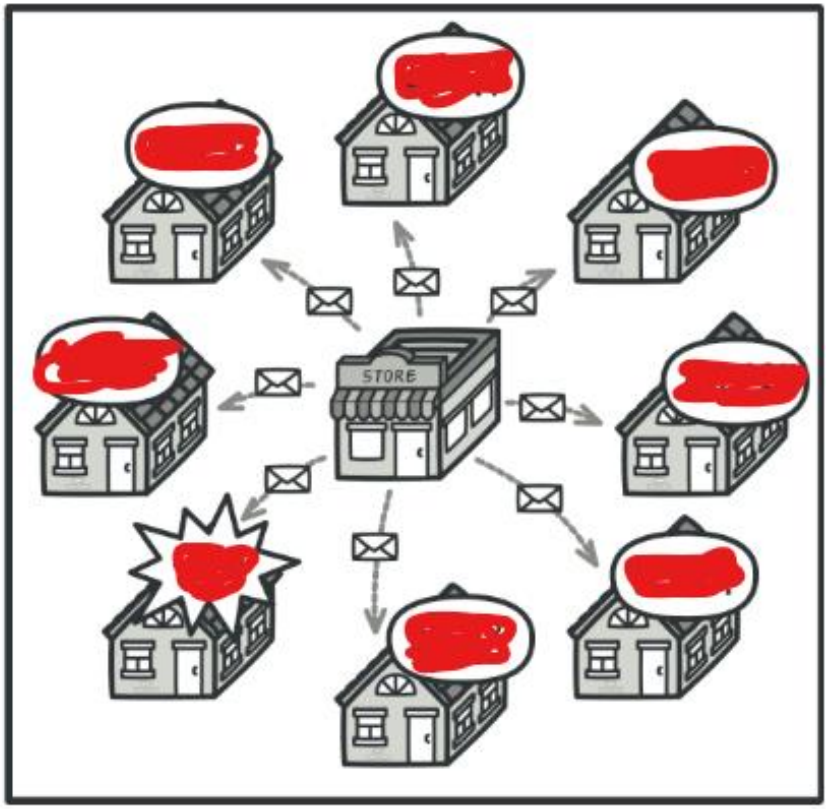
- A two-way association creates tight coupling between the two classes.
- Reusing or changing either of the classes will have to involve the other (when one object changes state, all its dependents are notified and updated automatically).
- On the other hand, we want to maximize the flexibility of the system to the greatest extent possible.
- Observer intent is:
  - » Define a *one-to-many* dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

GoF Patterns – Behavioral -- Observer

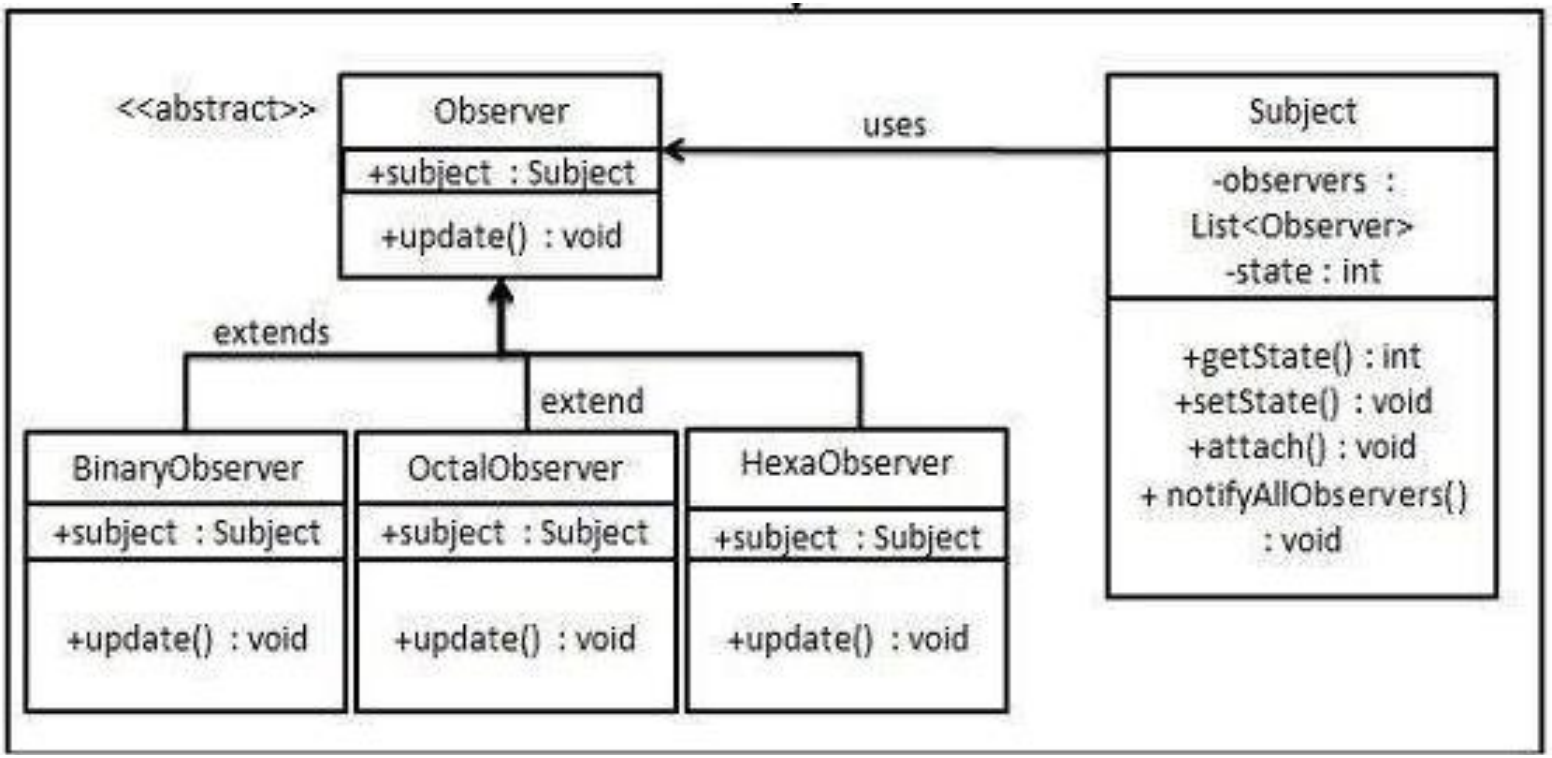


GoF Patterns – Behavioral -- Observer

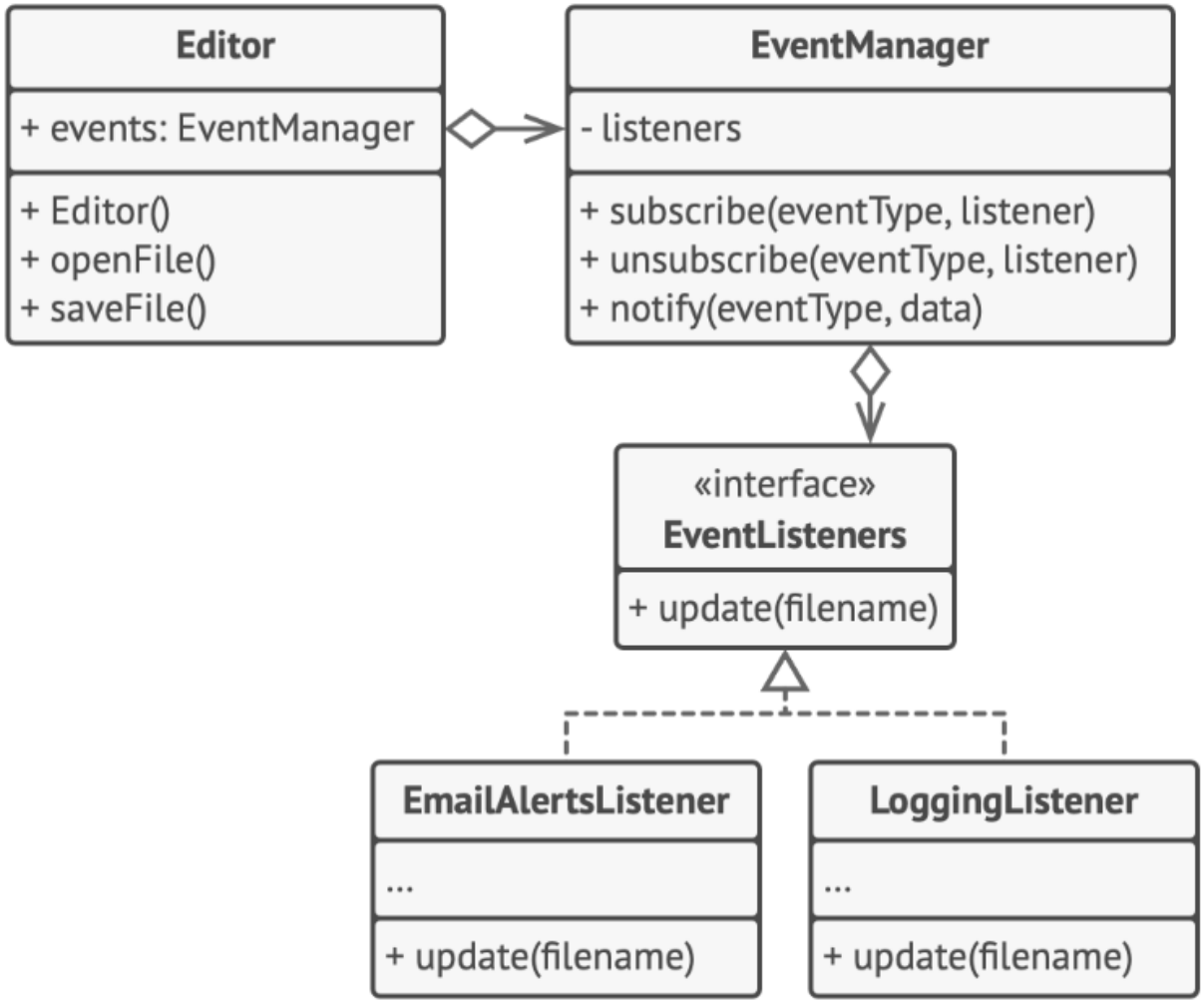




GoF Patterns – Behavioral -- Observer

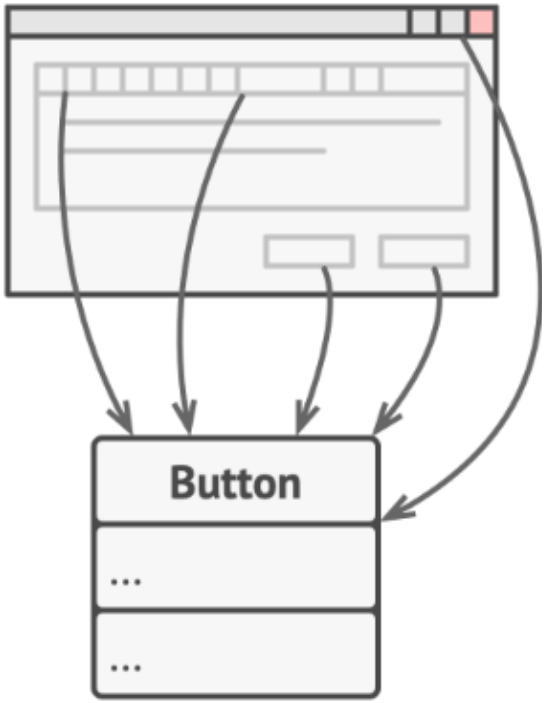


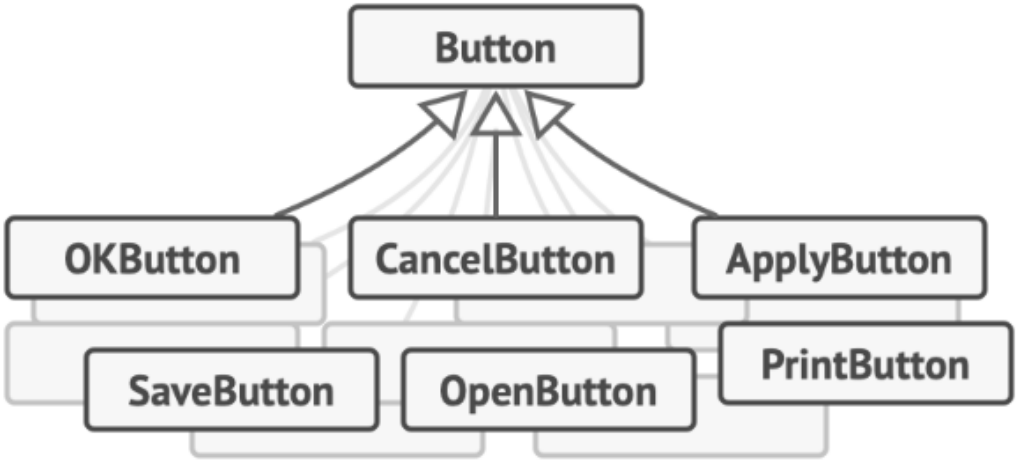
GoF Patterns – Behavioral -- Observer



GoF Patterns – Behavioral -- Command

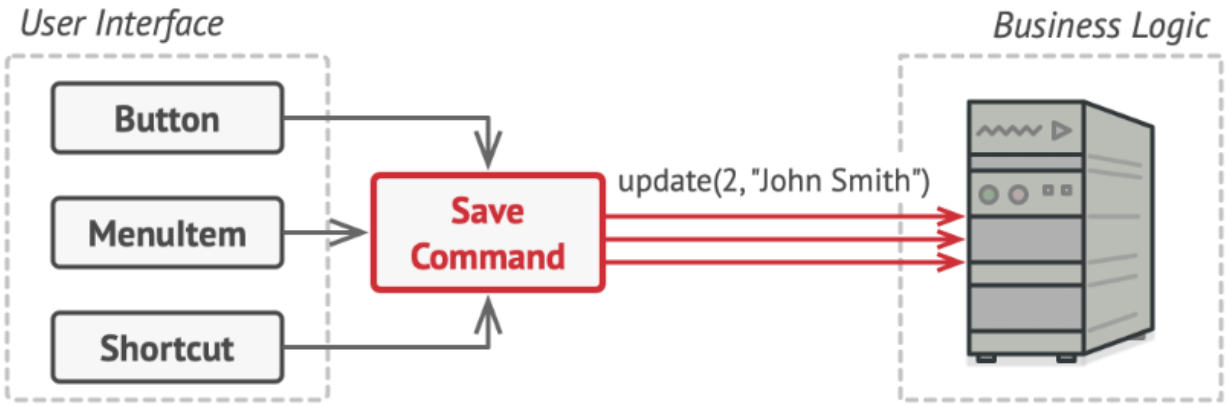
- Command is a behavioral design pattern that turns a request into a *stand-alone object* that contains all information about the request.





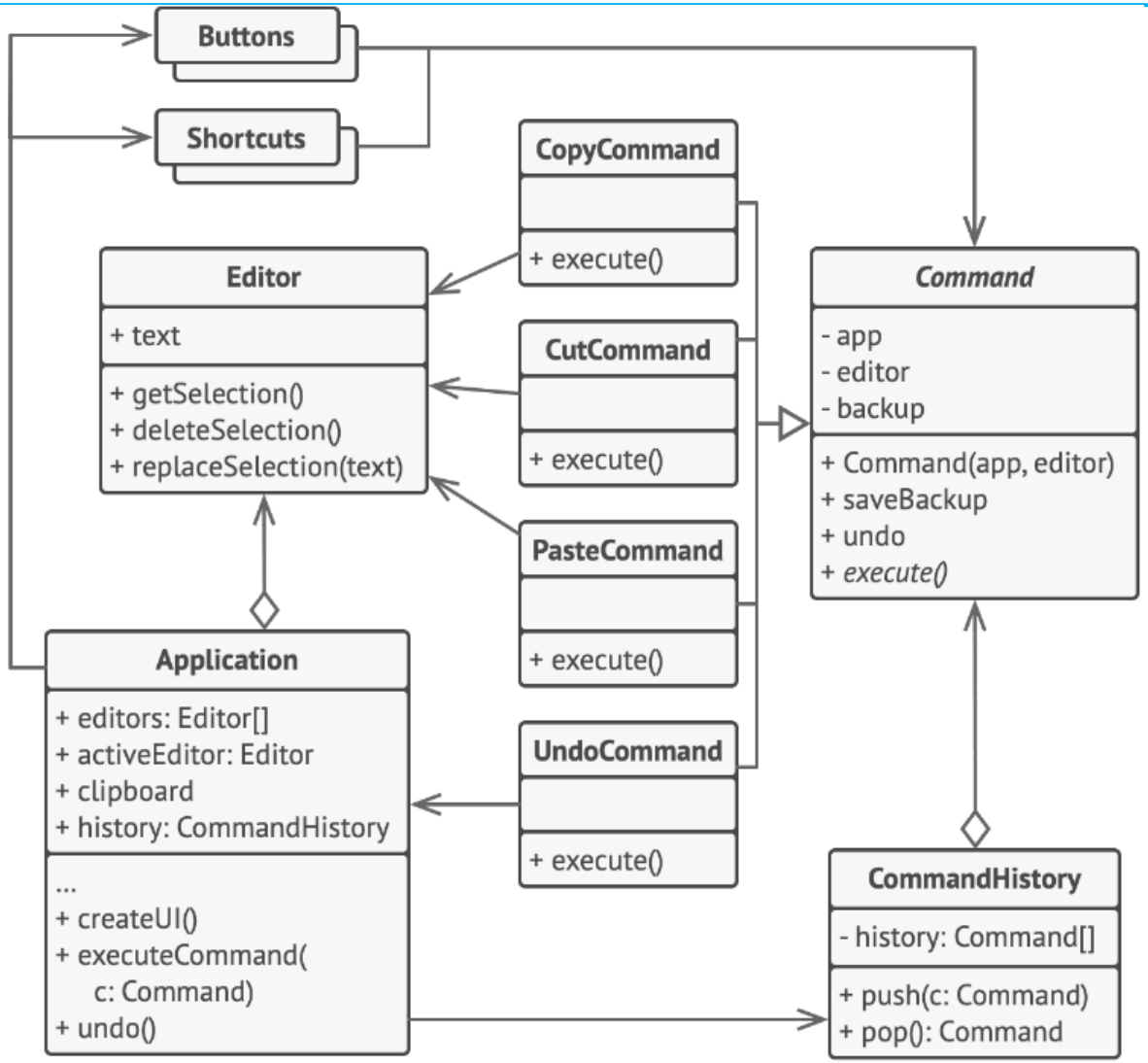
GoF Patterns – Behavioral -- Command

The GUI objects may access the business logic objects directly.



Accessing the business logic layer via a command.

GoF Patterns – Behavioral -- Command



GoF Patterns – Behavioral -- Command

- *Single Responsibility Principle*. You can *decouple* classes that invoke operations from classes that perform these operations.

- *Open/Closed Principle*. You can introduce *new commands* into the app without breaking existing client code.

- However, The code may become more *complicated* since you're introducing a whole *new layer* between senders and receivers.