SOLID Principle and Design Patterns

Dr. Youna Jung

Northeastern University

yo.jung@northeastern.edu

Design Principles

- **Principles**
  - ✓ used to **diagnose** problems with designs

SOLID Principle

- **Patterns**
  - ✓ used to **address** the problems

Design Patterns

SOLID Design Principles

- **Single responsibility** principle (SRP)
  - ✓ Every class should have only one reason to be changed
    - – ➔ If class "**A**" has two responsibilities, create new classes "**B**" and "**C**" to **handle each responsibility** in **isolation**, and then compose "**A**" out of "B" and "C"
- **Open/closed** principle (OCP)
  - ✓ Every class should be **open** for **extension** (derivative classes), but **closed** for **modification** (fixed interfaces)
    - – ➔ Put the system parts that are **likely** to **change** into **implementations** (i.e. concrete classes) and define **interfaces** around the **parts** that are **unlikely** to **change** (e.g. abstract base classes)

SOLID Design Principles

- **Liskov substitution** principle (LSP)
  - ✓ If class **A** is a subtype of class **B**, we should be able to replace **B** with **A without disrupting** the behavior of our program
    - ➔ Any **algorithm** that works on the **interface**, **should** continue to **work for any** substitute **implementation**
- **Interface segregation** principle (ISP)
  - ✓ Keep **interfaces** as **small** as possible, to avoid **unnecessary** dependencies
    - ➔ **Ideally**, it **should** be possible to **understand** any part of the **code in isolation**, **without** needing to **look up the rest** of the system code

SOLID Design Principles

- **Dependency inversion** principle (DIP)
  - ✓ Instead of having concrete implementations communicate directly (and depend on each other), **decouple** them by **formalizing** their communication interface as an **abstract interface** based on the needs of the higher-level class

Single Responsibility Principle (SRP)

- Every class should have only one reason to be changed
  - ✓ Ex) **Book** class to store the *name*, *author* and *text* associated with an instance of a Book

```java
public class Book {

    private String name;
    private String author;
    private String text;

    //constructor, getters and setters

    // methods that directly relate to the book properties
    public String replaceWordInText(String word, String replacementWord){
        return text.replaceAll(word, replacementWord);
    }

    public boolean isWordInText(String word){
        return text.contains(word);
    }
}
```

Single Responsibility Principle (SRP)

- ✓ Need a print method to display text
  - 1)

```java
public class BadBook {
    //...

    void printTextToConsole(){
        // our code for formatting and printing the text
    }
}
```

➔ **Violation** of **SRP**

```java
public class BookPrinter {

    // methods for outputting text
    void printTextToConsole(String text){
        //our code for formatting and printing the text
    }

    void printTextToAnotherMedium(String text){
        // code for writing to any other location..
    }
}
```

Open/Closed Principle (OCP)

- Every class should be **open** for **extension**, but **closed** for **modification**
  - ✓ Ex) **`Guitar`** class to represent an instance of a guitar

```
public class Guitar {

    private String make;
    private String model;
    private int volume;
```

  - ✓ Want to use a cool flame pattern
    - 1) modify **`Guitar`** class? ➔ **Violation** of **OCP**
    - 2)

```
public class SuperCoolGuitarWithFlames extends Guitar {

    private String flameColor;

    //constructor, getters + setters
}
```

Liskov Substitution Principle (LSP)

- If class **A** is a subtype of class **B**, we should be able to replace **B** with **A without disrupting** the behavior of our program

```java
public interface Car {

    void turnOnEngine();
    void accelerate();
}
```

Redesign the interface

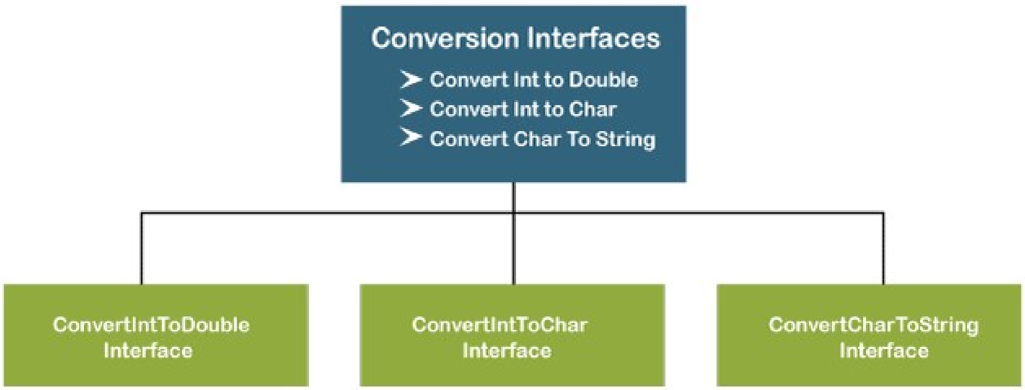
```java
public class ElectricCar implements Car {

    public void turnOnEngine() {
        throw new AssertionError("I don't have an engine!");
    }

    public void accelerate() {
        //this acceleration is crazy!
    }
}
```

Violation of **LSP**

```java
public class MotorCar implements Car {

    private Engine engine;

    //Constructors, getters + setters

    public void turnOnEngine() {
        //turn on the engine!
        engine.on();
    }

    public void accelerate() {
        //move forward!
        engine.powerOn(1000);
    }
}
```

Interface Segregation Principle (ISP)

- Keep **interfaces** as **small** as possible, to avoid **unnecessary** dependencies
  - ✓ Ex) **`Conversion`** interface



  - ✓ Ex) **`BearKeeper`** interface

```
public interface BearKeeper {
    void washTheBear();
    void feedTheBear();
    void petTheBear();
}
```

  – Some zookeepers take a part of the responsibilities ➔ **Violation** of **ISP**

Interface Segregation Principle (ISP)

```java
public interface BearCleaner {
    void washTheBear();
}

public interface BearFeeder {
    void feedTheBear();
}

public interface BearPetter {
    void petTheBear();
}
```

```java
public class BearCarer implements BearCleaner, BearFeeder {

    public void washTheBear() {
        //I think we missed a spot...
    }

    public void feedTheBear() {
        //Tuna Tuesdays...
    }
}
```

```java
public class CrazyPerson implements BearPetter {

    public void petTheBear() {
        //Good luck with that!
    }
}
```

Dependency Inversion Principle (DIP)

- **Decouple** implementation by **formalizing** their communication interface as an **abstract interface**
  - ✓ Ex) **`Windows98Machine`** interface

```
public class Windows98Machine {

    private final StandardKeyboard keyboard;
    private final Monitor monitor;

    public Windows98Machine() {
        monitor = new Monitor();
        keyboard = new StandardKeyboard();
    }

}
```

  - ✓ **`Windows98Machine`** class, **`StandardKeyboard`** class, and **`Monitor`** classes are tightly coupled → **Violation** of **DIP**

Dependency Inversion Principle (DIP)

```java
public interface Keyboard { }
```

```java
public class Windows98Machine{

    private final Keyboard keyboard;
    private final Monitor monitor;

    public Windows98Machine(Keyboard keyboard, Monitor monitor) {
        this.keyboard = keyboard;
        this.monitor = monitor;
    }
}
```

```java
public class StandardKeyboard implements Keyboard { }
```

Engineering Knowledge

- Engineering Knowledge is not only a set of **algorithms**

- It also contains a catalog of **patterns** describing generic solutions for recurring problems
  - ✓ Not described in ~~a programming language.~~
    - – Description usually in natural language
  - ✓ A pattern is presented in form of a schema consisting of sections of text and pictures (Drawings, UML diagrams, etc.)

Algorithm vs Pattern

❑ Algorithm

- ✓ A method for solving a problem using a finite sequence of well-defined instructions for solving a problem
- ✓ Starting from an initial state, the algorithm proceeds through a series of successive states, eventually terminating in a final state

❑ Pattern

- ✓ “*A pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem in such a way that you can use this solution a million times over, without ever doing it the same way twice*”

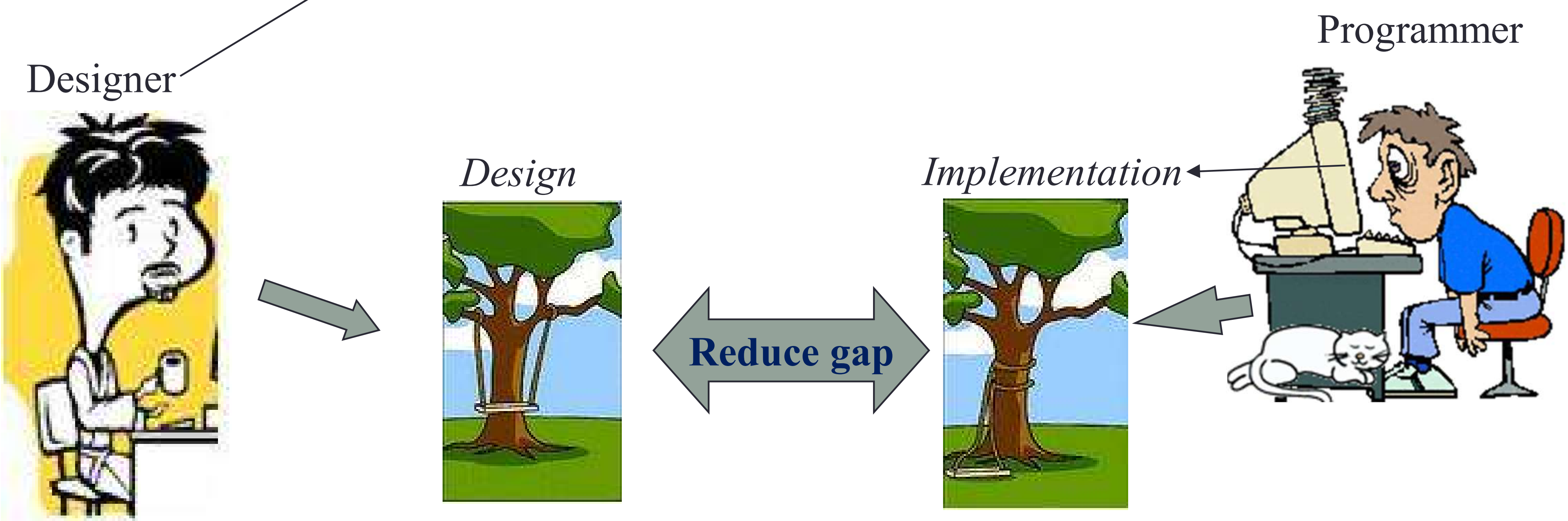
– *Christopher Alexander*, A Pattern language.

Pattern

- **Definition** (Christopher Alexander)
  - A pattern is a three-part rule, which expresses a relation between a certain **context**, a **problem**, and a **solution** for conflicting forces
  - Conflicting Forces (Design Tradeoffs)
    - Ex) The conflicting forces between a **sunny room** and **a room that does not overheat on on a sunny summer afternoon**
    - Ex) The conflicting forces between a **portable** and an **effective software** system.

Pattern

1. Design Problem
2. Solution
3. Implementation details



Benefits of using Patterns

- Patterns are a **common design vocabulary**
  - ✓ allows engineers to **abstract** a **problem** and talk about that abstraction in isolation **from** its **implementation**
  - ✓ embodies a **culture**
    - – **Domain-specific patterns** increase **design speed**
- Patterns capture **design expertise** and allow that expertise to be communicated
  - ✓ promotes design **reuse** and avoid mistakes
- Patterns improve **documentation** (less is needed) and understandability (patterns are described well once)

Design patterns you have already seen

- **Encapsulation** (Data Hiding)
- **Subclassing** (Inheritance)
- **Iteration**
- **Exceptions**

Design patterns you have already seen

- **Encapsulation** pattern
  - **Problem**
    - Exposed fields are directly manipulated from outside, leading to undesirable dependences that prevent changing the implementation.
  - **Solution**
    - **Hide** some components, **permitting** only stylized access to the object.
- **Subclassing** pattern
  - **Problem**
    - Similar abstractions have similar members (fields and methods). Repeating these is tedious, error-prone, and a maintenance headache.
  - **Solution**
    - **Inherit** default members **from** a **superclass** → **select** the **correct implementation** via run-time dispatching.

Design patterns you have already seen

- **Iteration** pattern
  - **Problem**
    - Clients that wish to access all members of a collection must perform a specialized traversal for each data structure.
  - **Solution**
    - **Implementations** perform **traversals**. The results are communicated to clients via a standard interface.
- **Exception** pattern
  - **Problem**
    - Code is cluttered with error-handling code.
  - **Solution**
    - Errors occurring in one part of the code should often be handled elsewhere. Use language structures for **throwing** and **catching** **exceptions**.

Pattern Language and Pattern Catalogs

Pattern Language

- A **collection** of patterns that forms a **vocabulary** for understanding and communicating ideas and the **rules** to combine them into an **architectural style**
- Pattern languages describe **software frameworks** or families of related systems.

Pattern Catalog

- A collection of **related patterns**
- It typically subdivides the patterns into at least a small number of broad **categories** and may include some amount of **cross referencing** between patterns.

Schemata for Describing Patterns

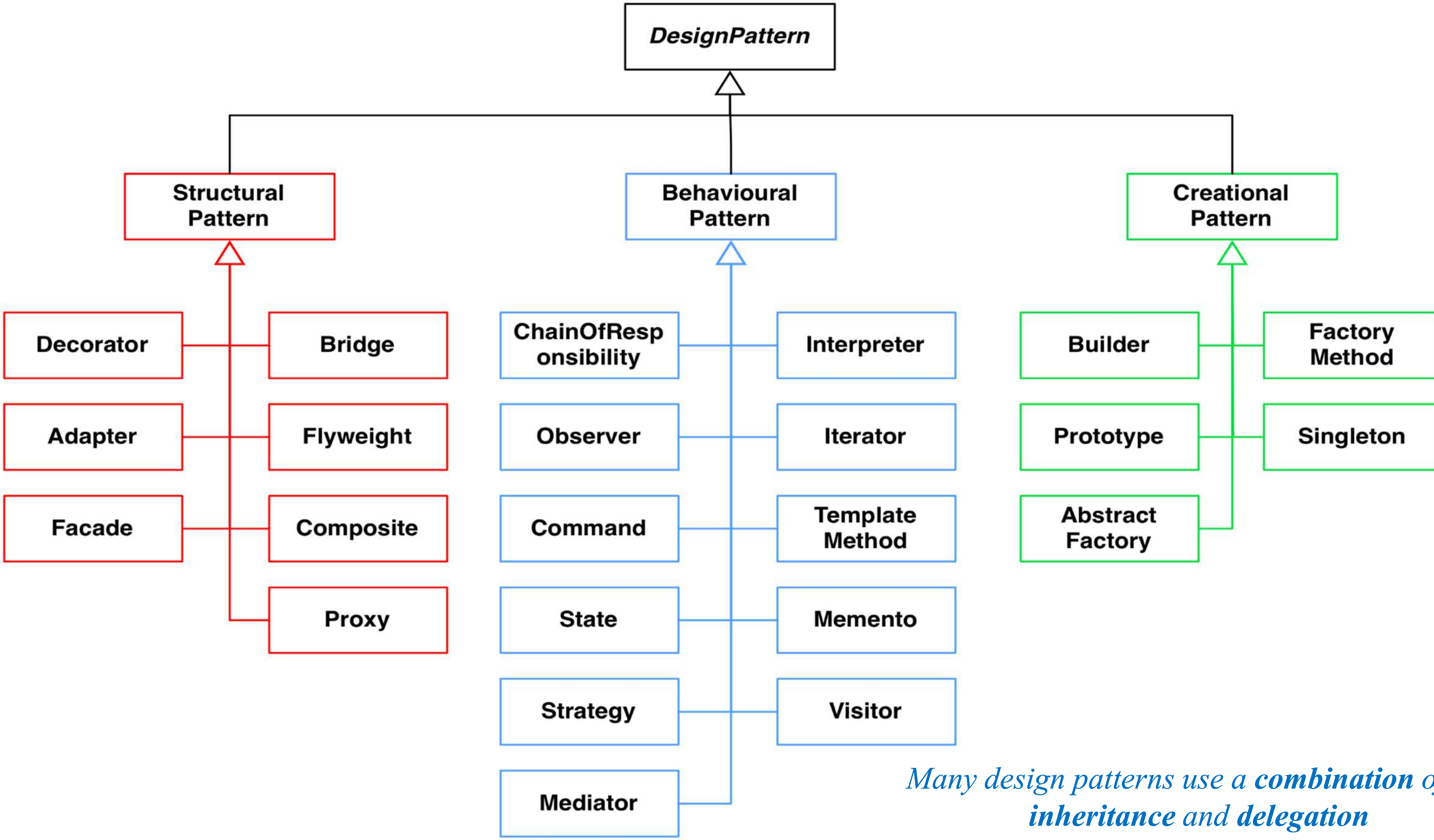
- **Alexander's Schema ("Alexandrian Form")**
  - ✓ *A Pattern Language – Towns Buildings Construction,* Christopher Alexander, Sara Ishikawa, Murray Silverstein, Vol. 2, Oxford University Press, New York, 1977
- **Gang of Four Schema**
  - ✓ *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison Wesley, October 1994
- **Gang of Five Schema**
  - ✓ *Pattern-Oriented Software Architecture - A System of Patterns*, Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, Wiley and Sons Ltd., 1996.

GoF: 3 Types of Design Patterns

- **Structural** Patterns
  - ✓ Reduce ~~coupling~~ between two or more classes
  - ✓ Introduce an **abstract class** to enable future extensions
  - ✓ **Encapsulate** complex structures
- **Behavioral** Patterns
  - ✓ Allow a choice between **algorithms** and the assignment of **responsibilities** to objects (“Who does what?”)
  - ✓ Characterize complex **control flows** that are difficult to follow at runtime
- **Creational** Patterns
  - ✓ Allow to abstract from complex **instantiation** processes
  - ✓ Make the system **independent** from the way its objects are **created**, **composed** and **represented**.

PATTERNS

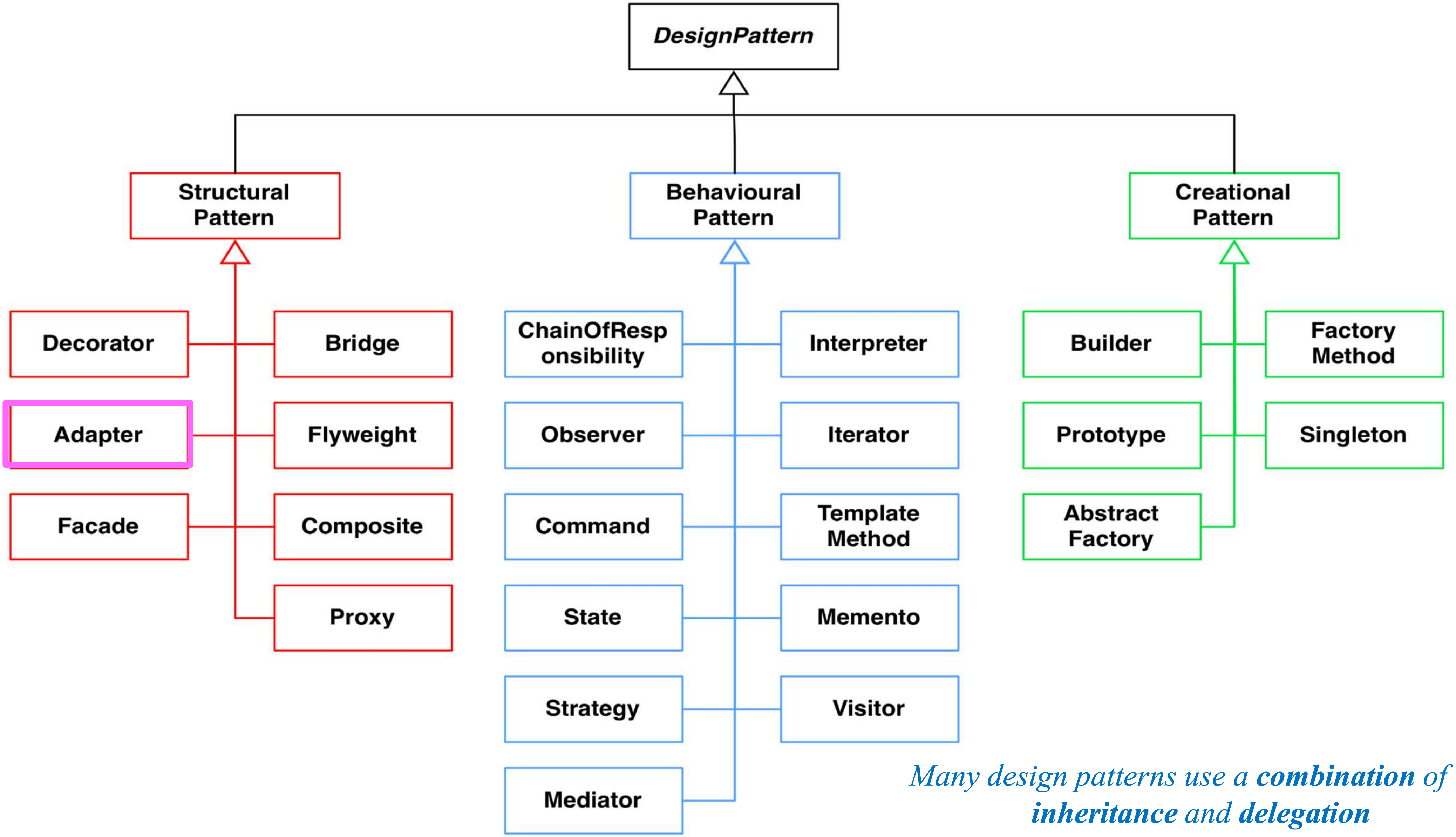
Taxonomy of Design Patterns (23 Patterns)



*Many design patterns use a **combination** of **inheritance** and **delegation***

ADAPTER PATTERN

Taxonomy of Design Patterns (23 Patterns)



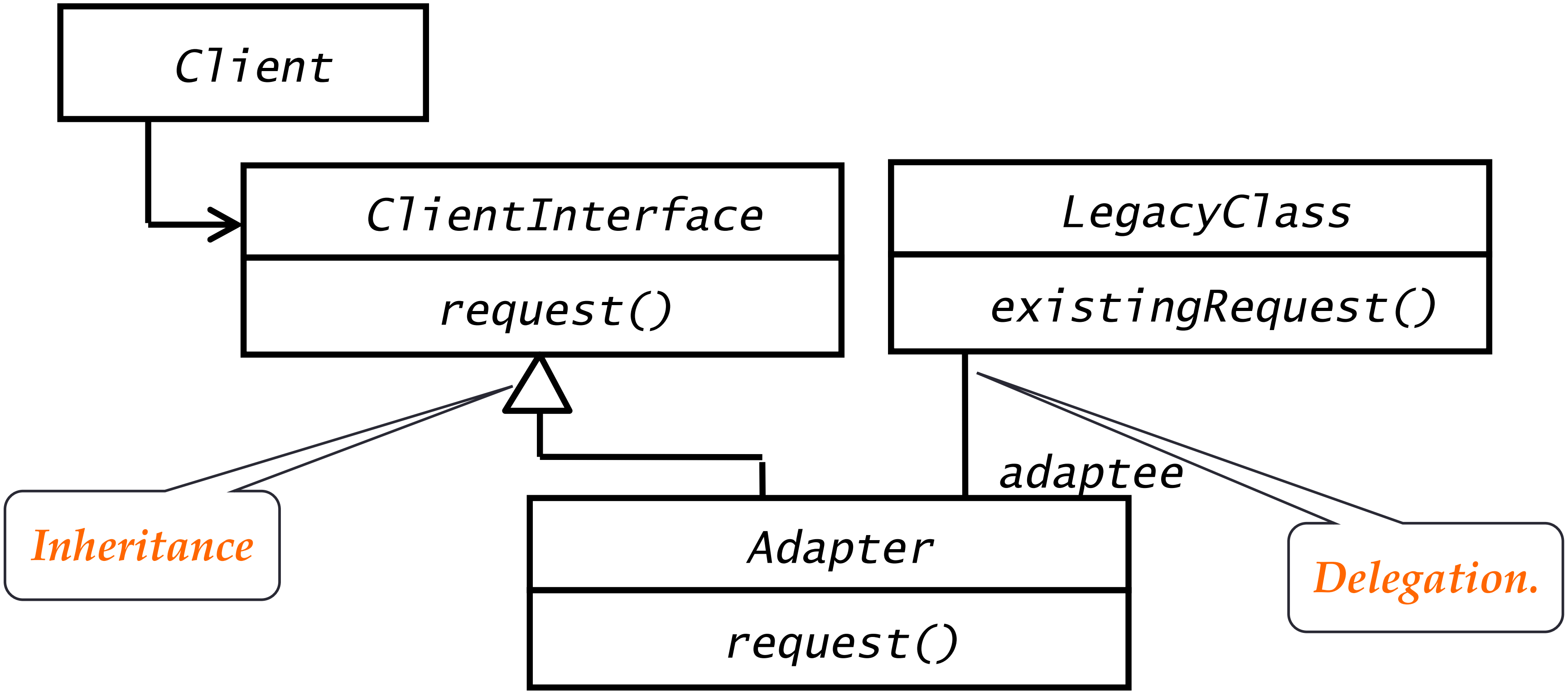
Many design patterns use a ***combination*** *of* ***inheritance*** *and* ***delegation***

Adapter Pattern

- The adapter pattern lets **classes** work together that couldn't otherwise because of **incompatible interfaces**
  - ✓ acts as a connector between two incompatible interfaces that otherwise cannot be connected directly
    - – "***Convert*** *the* ***interface*** *of a class into* ***another*** *interface expected by a client class*"
      - ➢ Used to provide a **new interface** to existing legacy components (Interface engineering, reengineering)

- Two adapter patterns
  - ✓ **Class** adapter (Out of the scope)
    - – Uses **multiple** inheritance to adapt one interface to another
  - ✓ **Object** adapter
    - – Uses **single** inheritance and delegation
    - – Object adapters are much more **frequent**.

Adapter Pattern

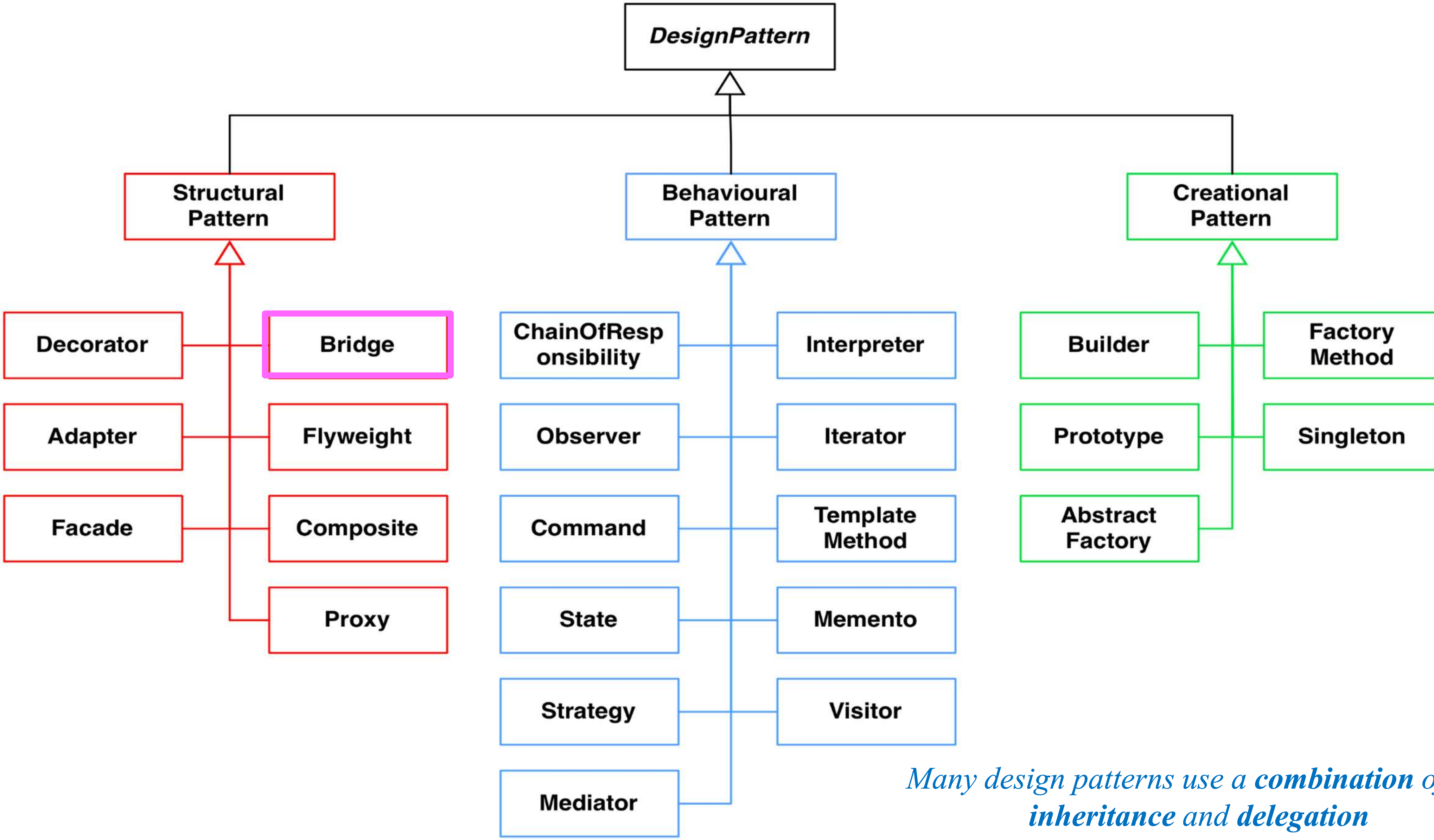


The adapter pattern uses **inheritance** as well as **delegation**:

- **Interface inheritance**: **Adapter** inherits **Request()** from **ClientInterface**
- **Delegation**: Binds **LegacyClass** to the **Adapter**.
  - **LegacyClass** delegates the responsibility to **Adapter** by **invoking** **Adapter.request()** whenever **existingRequest**() is **called**

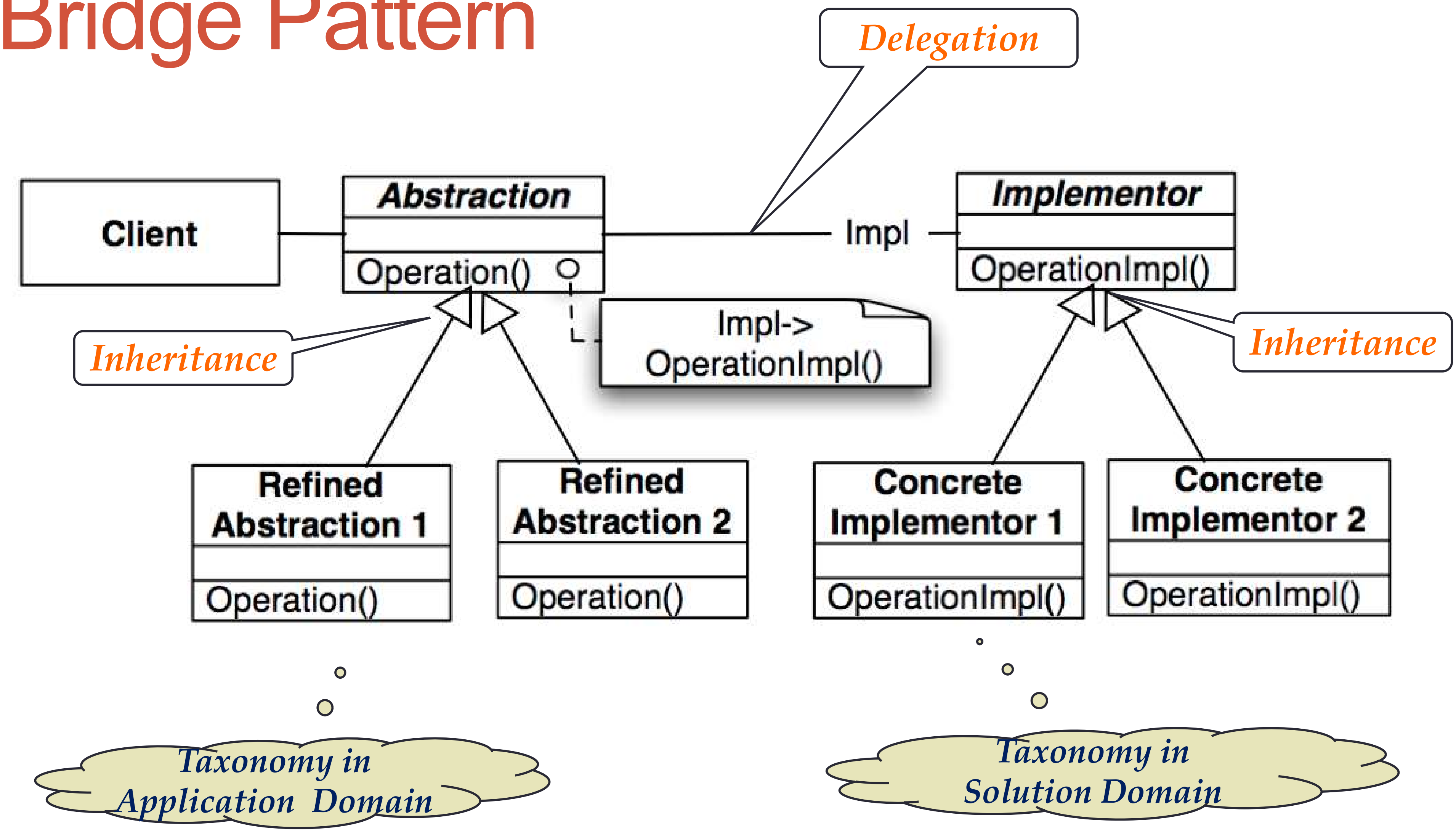
BRIDGE PATTERN

Taxonomy of Design Patterns (23 Patterns)



*Many design patterns use a **combination** of **inheritance** and **delegation***

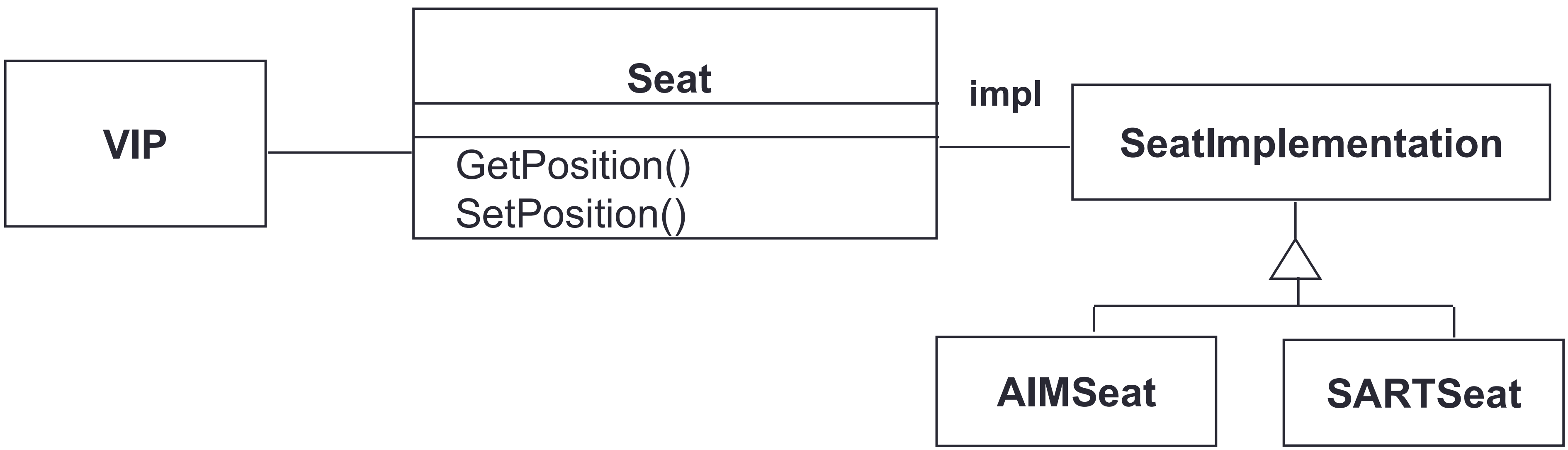
Bridge Pattern



It provides a **bridge between** the **Abstraction** (in the **application** domain) and the **Implementor** (in the **solution** domain)

Using a Bridge

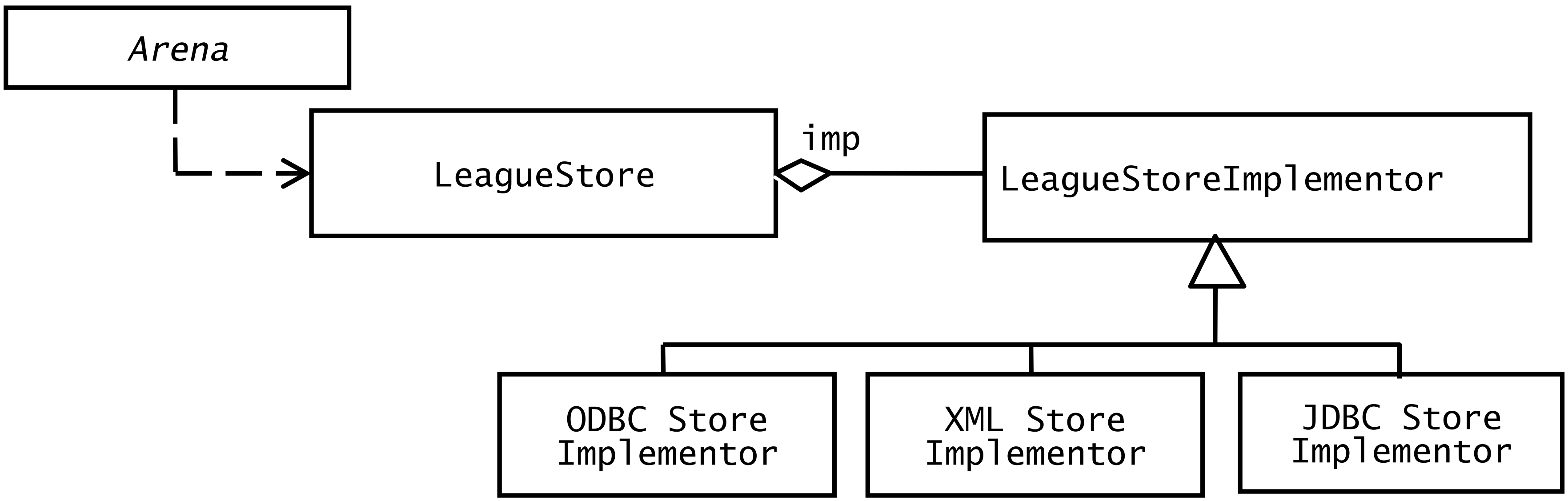
- The bridge pattern can be used to provide **multiple implementations** under the **same interface**



SeatImplementation

```
public interface SeatImplementation {
  public int GetPosition();
  public void SetPosition(int newPosition);
}
public class AimSeat implements SeatImplementation {
  public int GetPosition() {
    // actual call to the AIM simulation system
  }
  ….
}
public class SARTSeat implements SeatImplementation {
  public int GetPosition() {
    // actual call to the SART seat simulator
  }
  ...
}
```

Use of the Bridge Pattern: Supporting multiple Database Vendors



Advantage

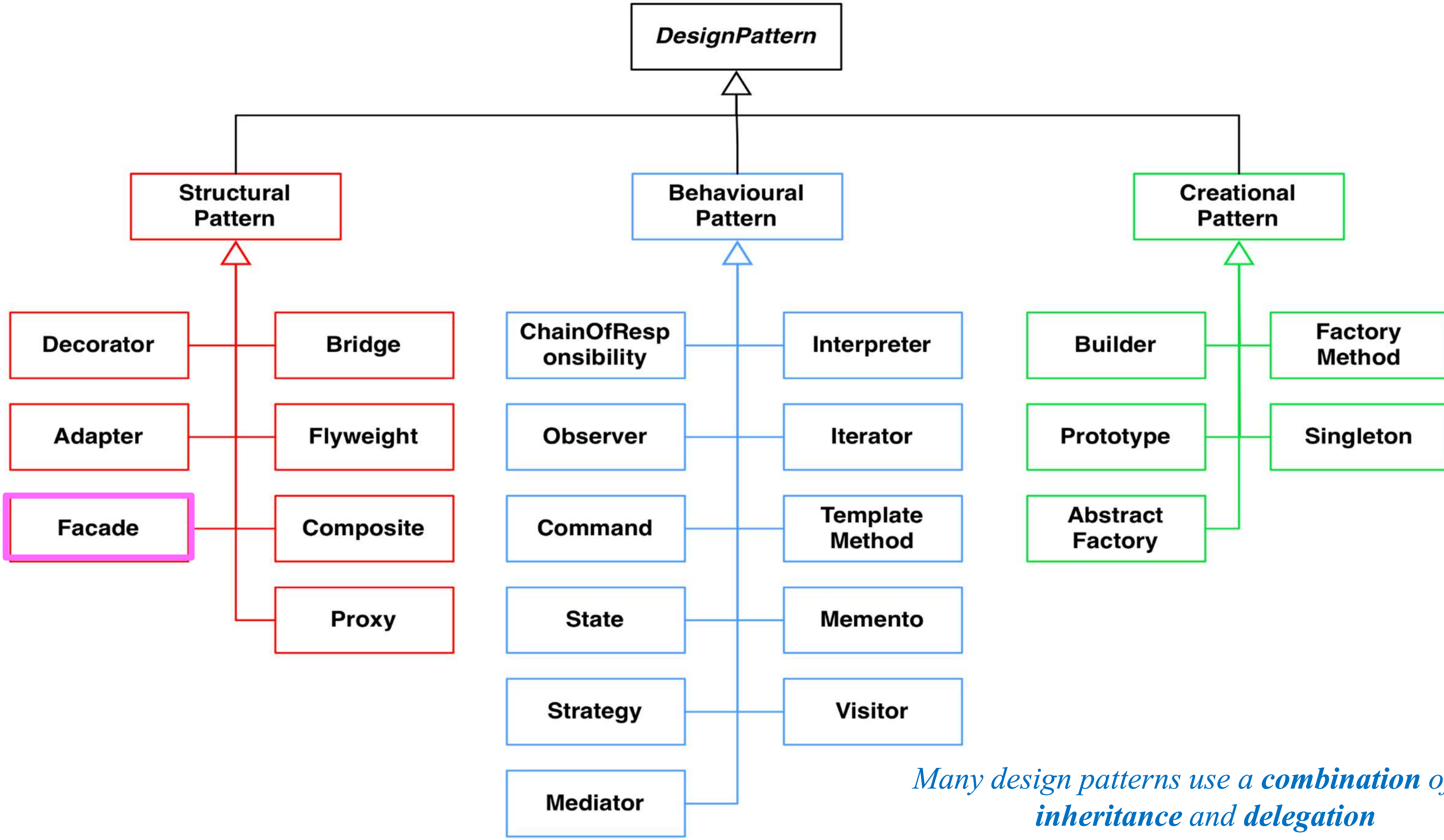
- The Bridge Pattern allows to postpone **Design Decisions** to the startup time of a system
  - ✓ Many design **decisions** are **made** at **design time** (Design Window), or at the latest, at **compile time**
    - → Bind a **client** to **one** of many **implementation classes** of an interface
  - ✓ The bridge pattern is useful to delay this binding between client and interface implementation until run time
    - Usually the binding occurs at the start up of the system (e.g. in the constructor of the interface class).

Adapter vs Bridge

- **Similarities**
  - ✓ Both hide the details of the underlying implementation
- **Difference**
  - ✓ The **adapter** pattern is geared towards making unrelated components work together
    - – Applied to systems that are **already designed** (reengineering, interface engineering projects)
      - ➢ "*Inheritance followed by delegation*"
  - ✓ A **bridge**, on the other hand, is used up-front in a design to let abstractions and implementations vary independently
    - – New "beasts" can be added to the "zoo" ("application and solution domain zoo", even if these are not known at analysis or system design time
      - ➢ *"Delegation followed by inheritance".*

FAÇADE PATTERN

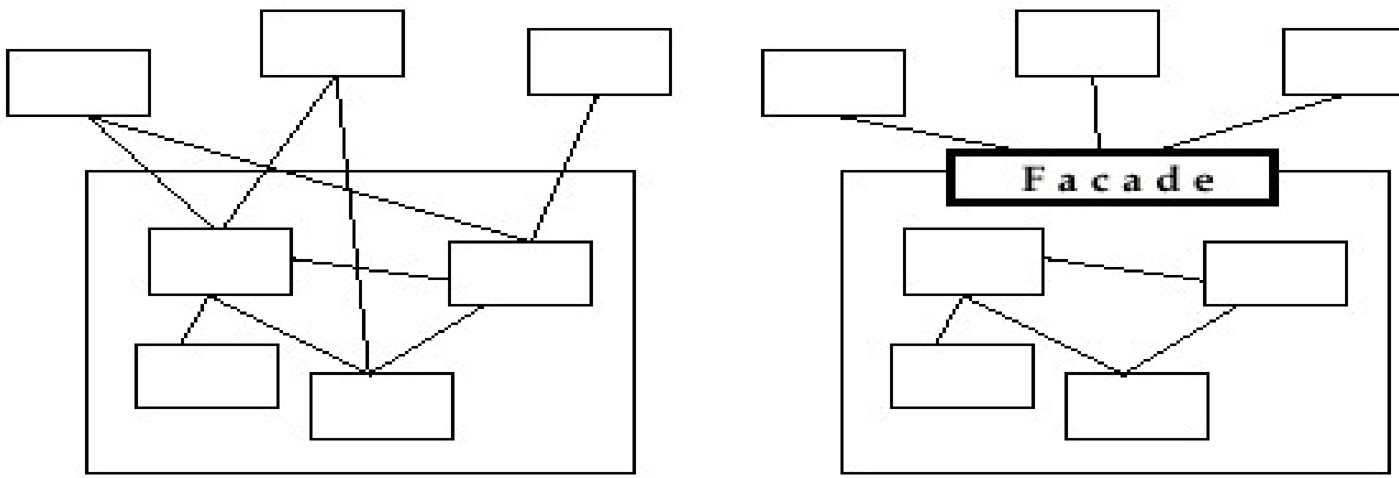
Taxonomy of Design Patterns (23 Patterns)



Many design patterns use a ***combination*** *of* ***inheritance*** *and* ***delegation***

Facade Pattern

- Provides a **unified interface** to a set of **classes** in a **subsystem**
  - A **façade** consists of a **set** of **public operations**
    - **Each** public **operation** is **delegated** to one or more **operations** in the **classes behind** the **facade**
- A facade defines a **higher-level interface** that makes the **subsystem easier** to **use**



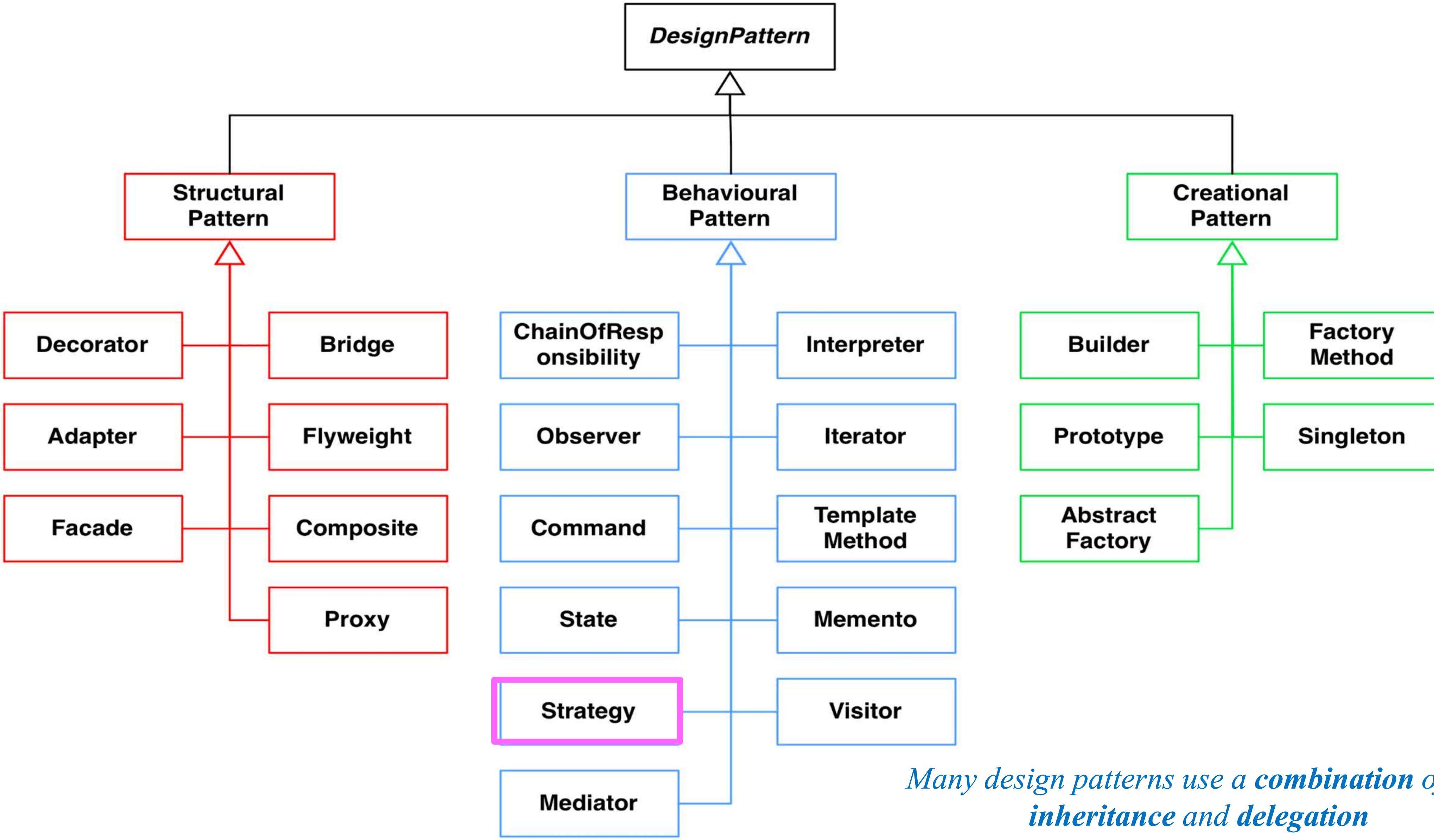
Subsystem Design

- The **ideal structure** of a **subsystem consists** of
  - ✓ an **interface** object
  - ✓ a set of **entity** objects (application domain objects) modeling real entities or existing systems
    - – Some of these entity objects are interfaces to existing systems
  - ✓ one or more **control** objects

- **Realization** of the **interface** object: **Facade**
  - ✓ Provides the **interface** to the **subsystem**
- **Interface** to the **entity** objects: **Adapter** or **Bridge**
  - ✓ Provides the **interface** to an **existing system** (legacy system)
    - – The existing system is not necessarily object-oriented!

STRATEGY PATTERN

Taxonomy of Design Patterns (23 Patterns)

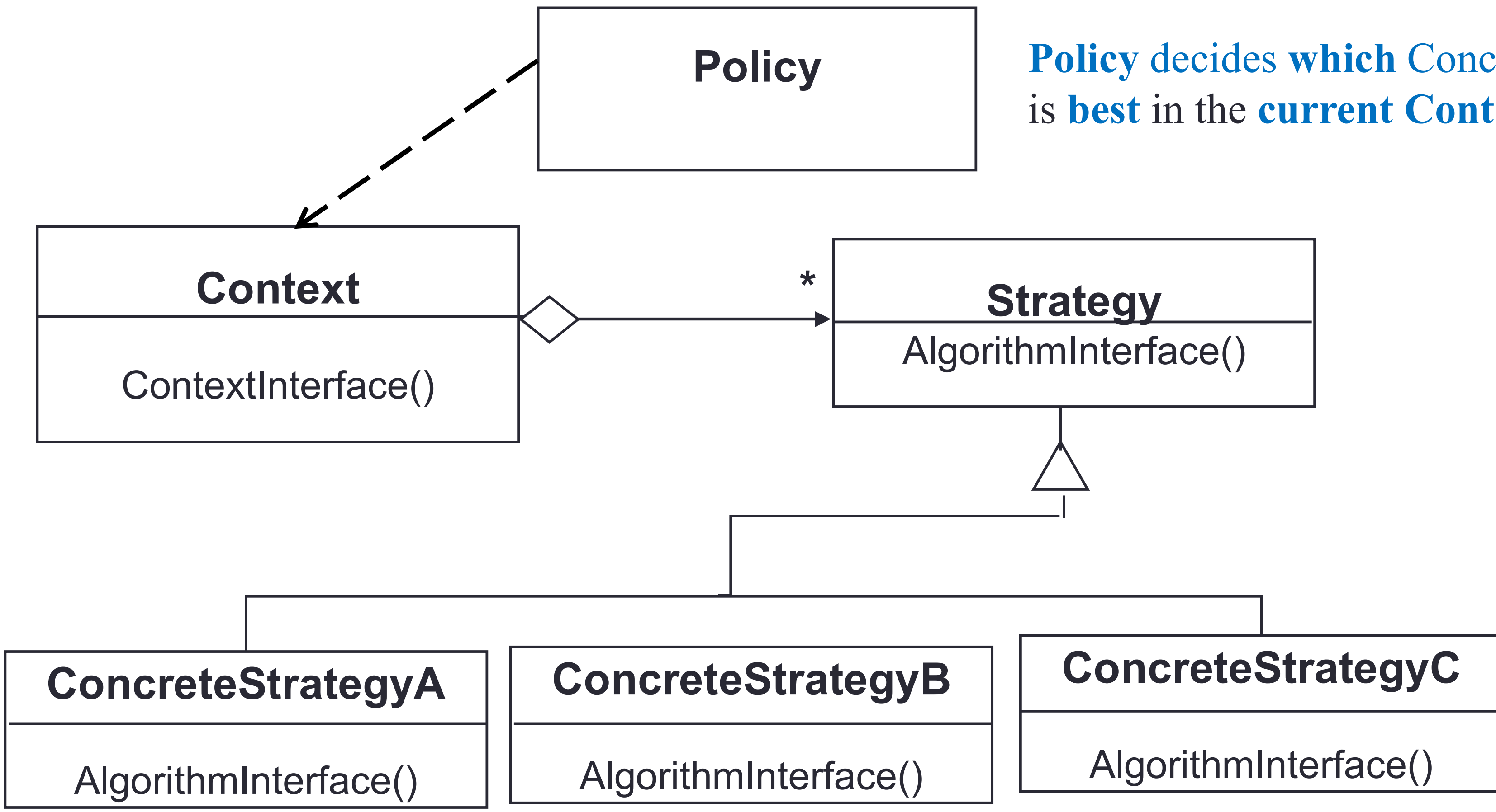


Many design patterns use a ***combination*** *of* ***inheritance*** *and* ***delegation***

Strategy Pattern

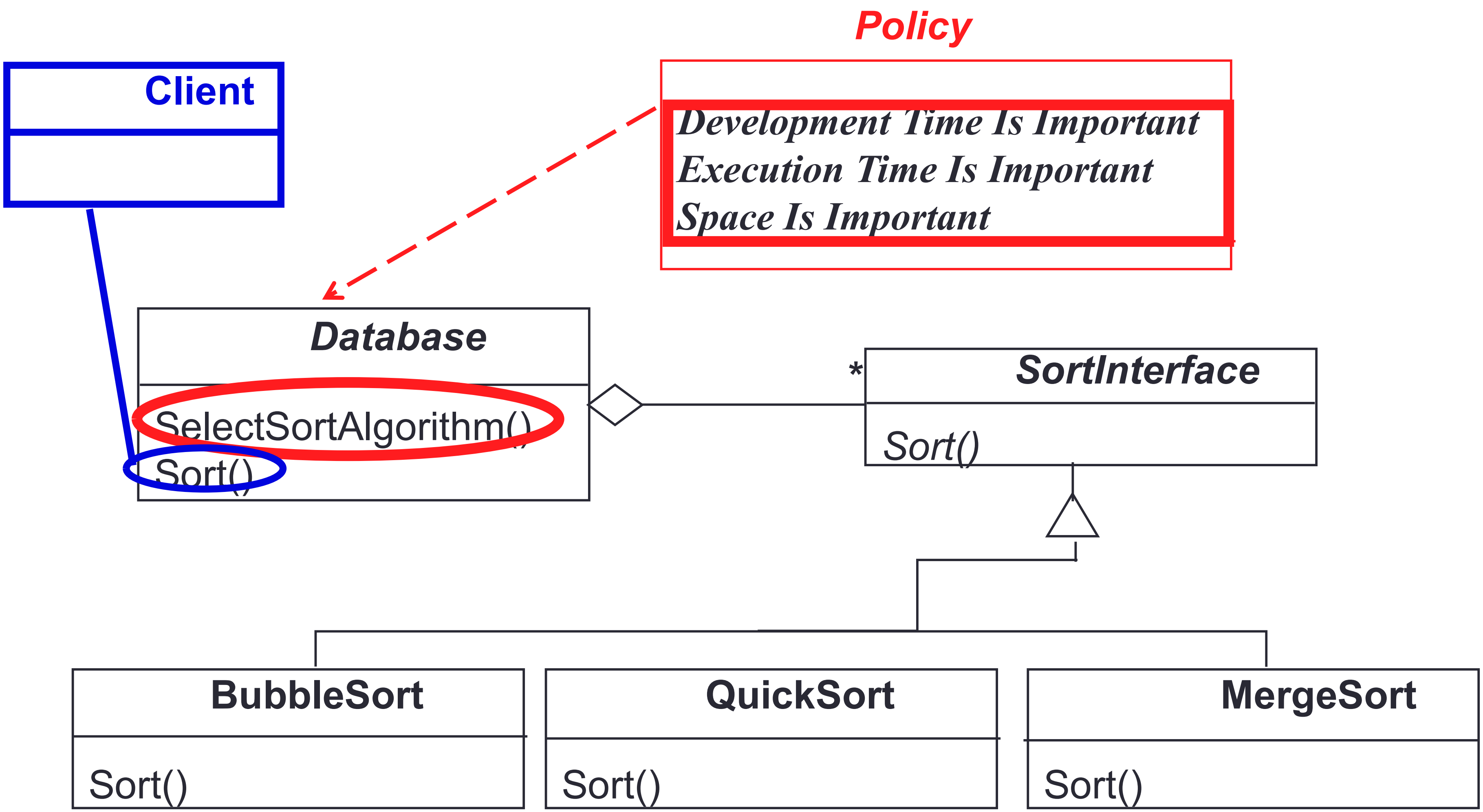
- Different algorithms exists for a specific task
  - ✓ Different algorithms will be appropriate at different times
  - ✓ We can **switch** between the algorithms at run time
    - Ex) **Different collision strategies** for objects in **video games**
    - Ex) **Parsing** a set of **tokens** into an abstract syntax tree (Bottom up, top down)
    - Ex) **Sorting** a **list** of **customers** (Bubble sort, mergesort, quicksort)
  - ✓ If we need a **new algorithm**, we can **add** it **without disturbing** the application or the other algorithms.

Strategy Pattern

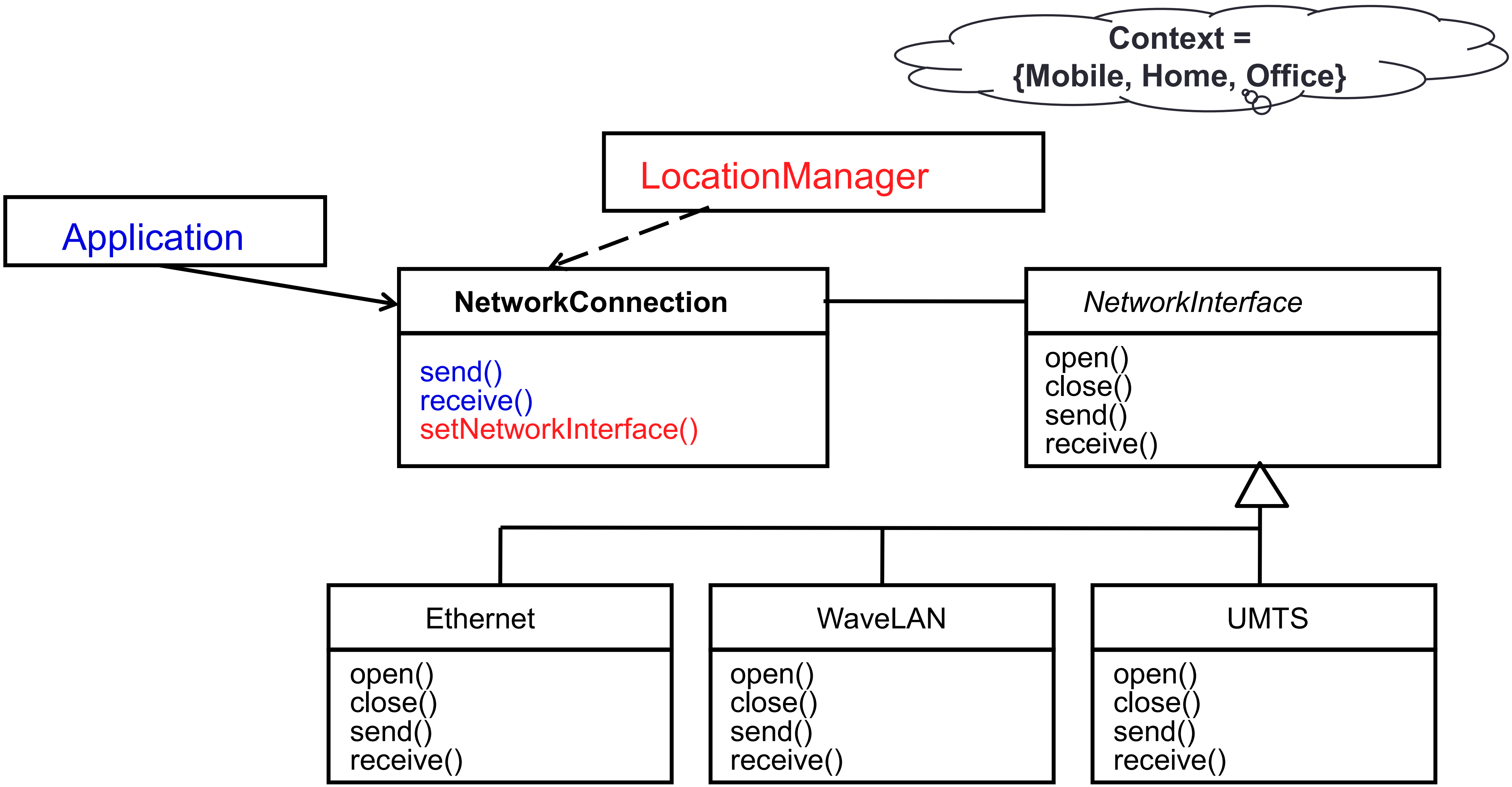


Policy decides **which** Concrete **Strategy** is **best** in the **current Context**.

Using a Strategy Pattern to Decide between Algorithms at Runtime

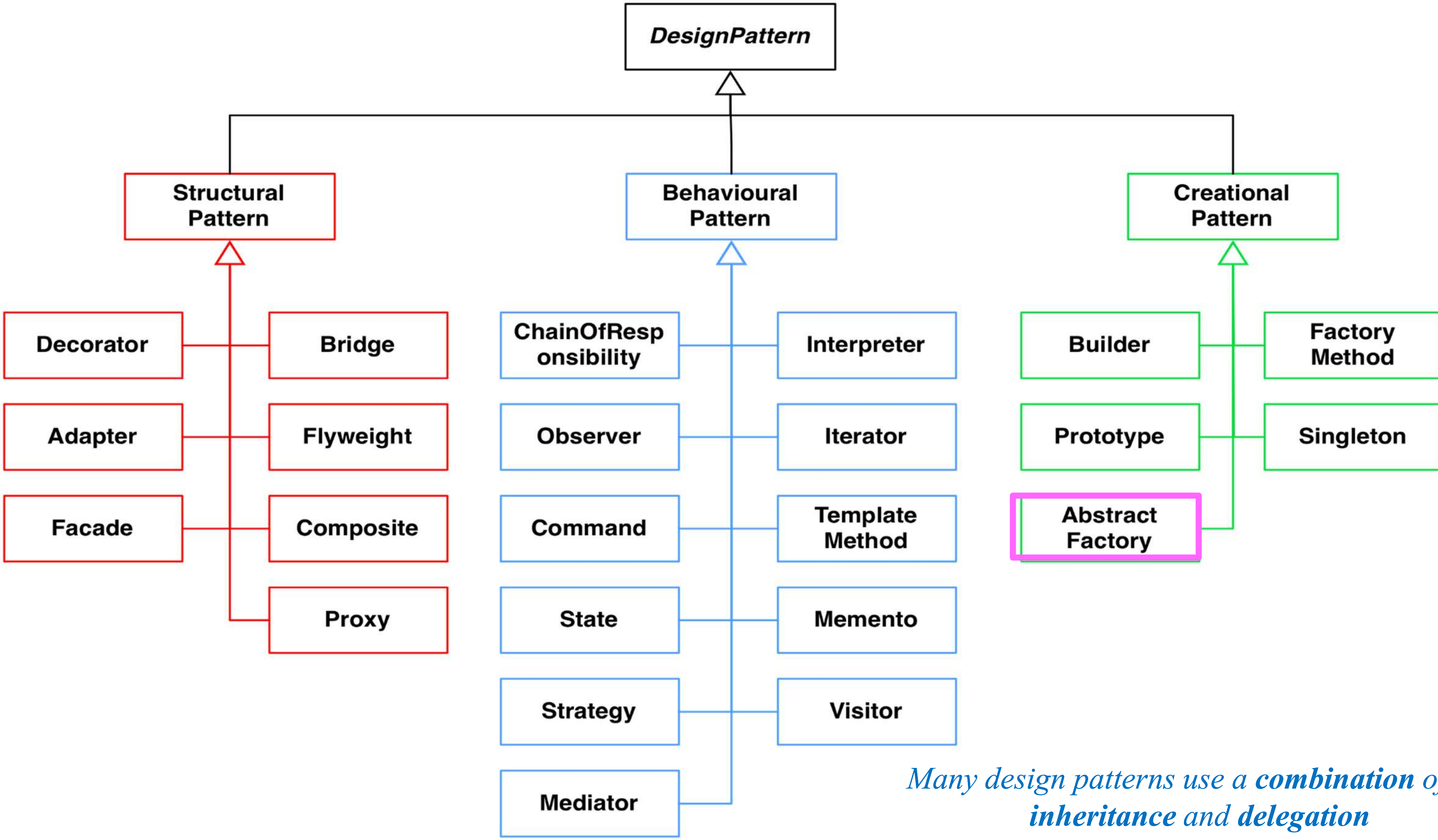


Supporting Multiple implementations of a Network Interface



ABSTRACT FACTORY PATTERN

Taxonomy of Design Patterns (23 Patterns)



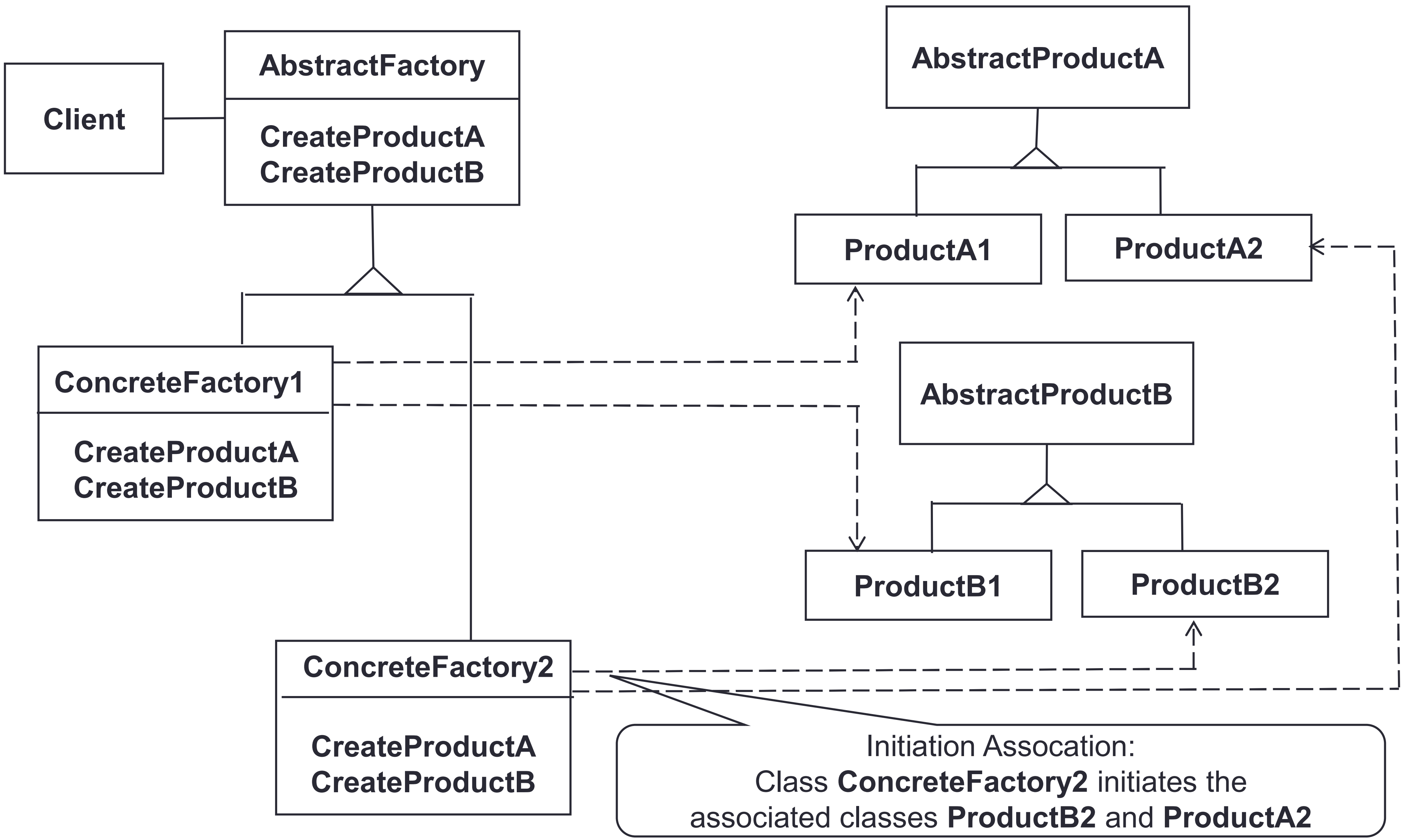
Many design patterns use a ***combination*** *of* ***inheritance*** *and* ***delegation***

Abstract Factory Pattern Motivation

- Consider a user interface toolkit that supports **multiple** looks and feel **standards** for **different OSes**
  - ✓ How can you write a single user interface and make it portable across the different look and feel standards for these window managers?

- Consider a facility management system for an intelligent house that supports **different control systems**
  - ✓ How can you write a single control system that is independent from the manufacturer?

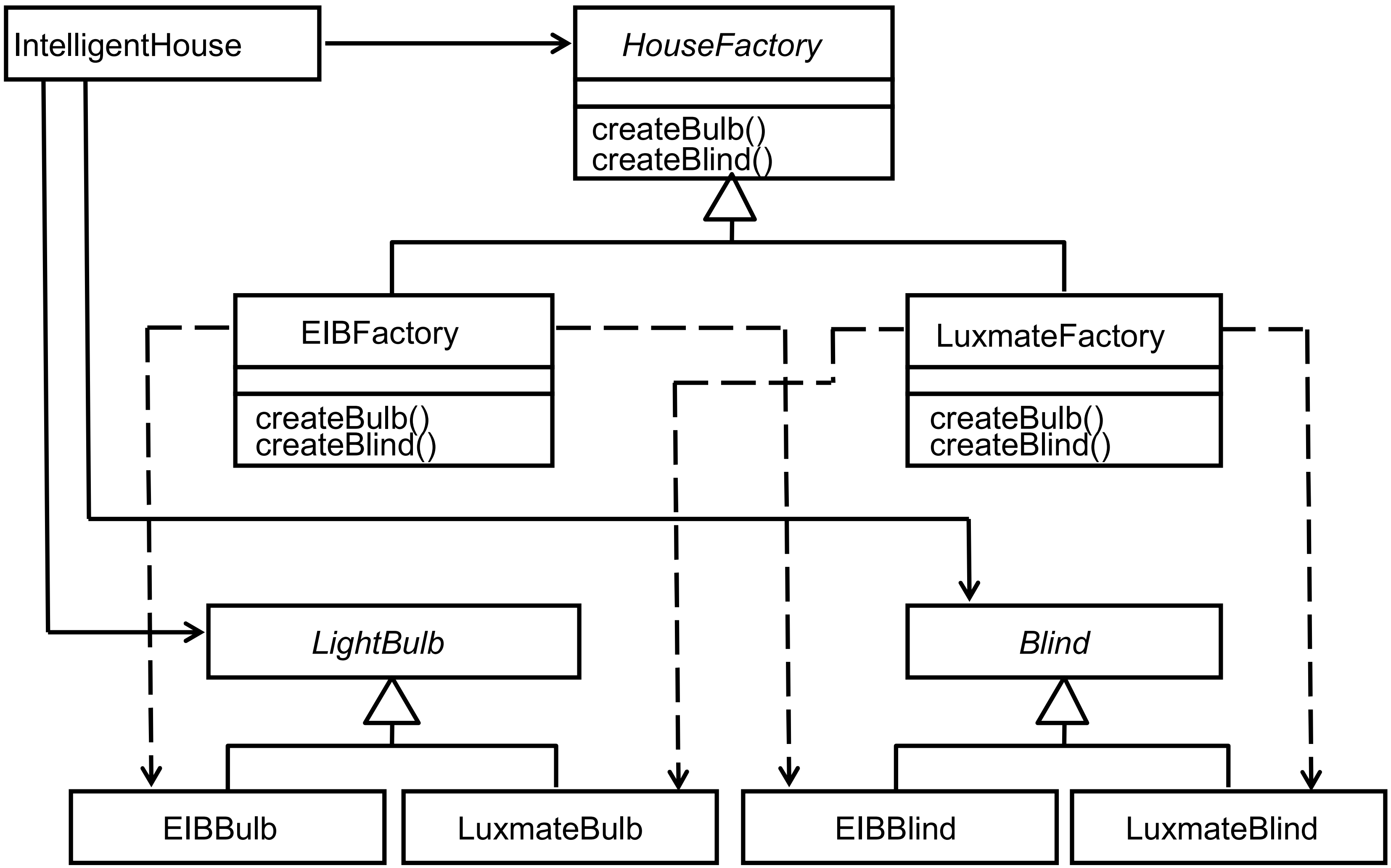
Abstract Factory



Applicability for Abstract Factory Pattern

- Independence from **creation**, **composition**, or **Representation**
  - ✓ The system should be independent of how its products are created, composed or represented
- **Manufacturer Independence**
  - ✓ A system should be configured as one family of products
    - where one has a choice from many different families.
- **Constraints** on related products
  - ✓ A family of related products is designed to be used together and you need to enforce this constraint

A Facility Management System for a House



Nonfunctional Requirements and Patterns

- NF: "*manufacturer independent*", "*device independent*", "*must support a family of products*"
  → **Abstract Factory** Pattern
- NF: "*must interface with an existing object*"
  → **Adapter** Pattern
- NF: "*must interface to several systems, some of them to be developed in the future*", "*an early prototype must be demonstrated*"
  → **Bridge** Pattern
- NF: "*must interface to existing set of objects*"
  → **Façade** Pattern

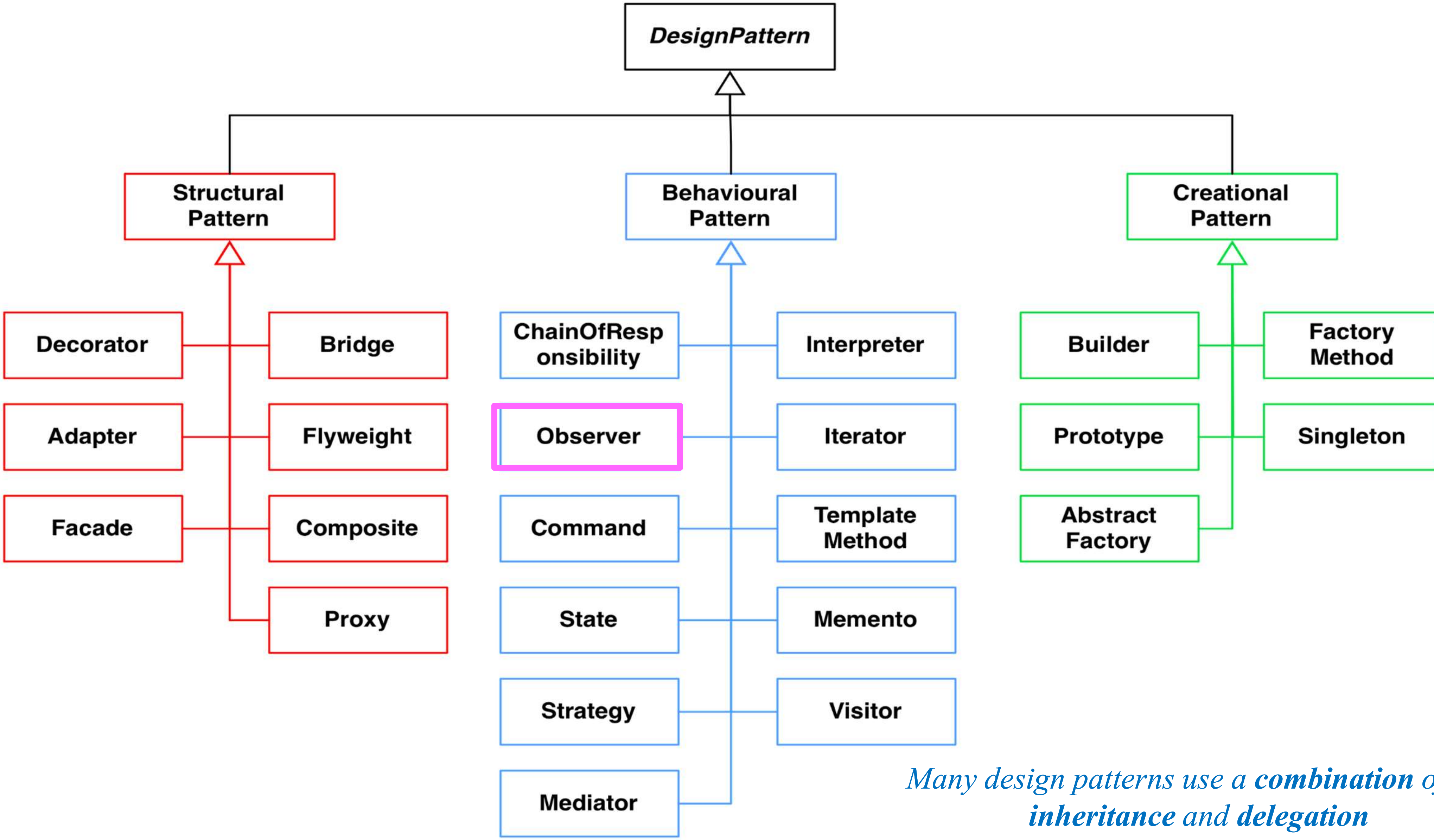
Nonfunctional Requirements and Patterns

- *NF:* "*complex structure*", "*must have variable depth and width*" "
  - ➔ Composite Pattern
- *NF:* "*must provide a policy independent **from** the mechanism*"
  - ➔ **Strategy** Pattern
- *NF:* "*must be location transparent*"
  - ➔ Proxy Pattern
- *NF:* "*must be extensible*", "*must be scalable*"
  - ➔ **Observer** Pattern (MVC Architectural Pattern)

MVC PATTERN

Observer Pattern

Taxonomy of Design Patterns (23 Patterns)

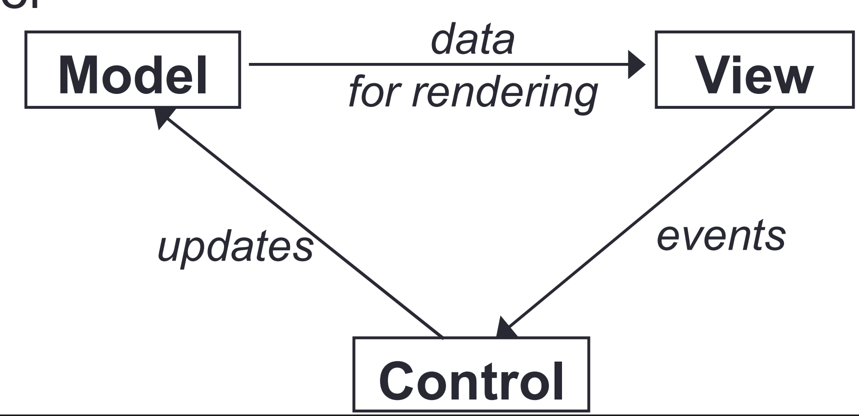


Many design patterns use a ***combination*** *of* ***inheritance*** *and* ***delegation***

Model-View-Controller Pattern

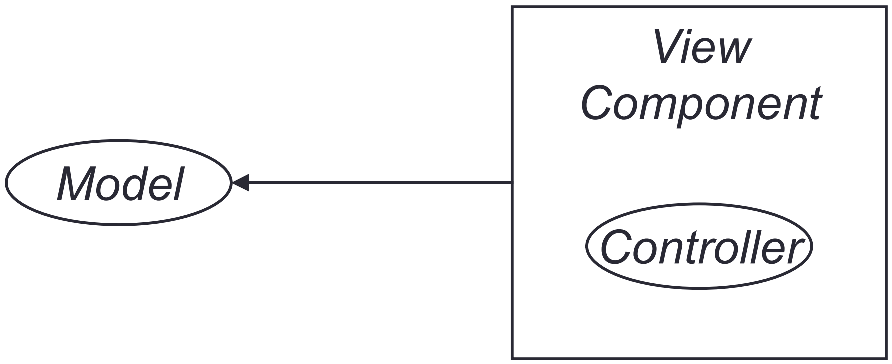
Model

- **Classes** in your system that are related to the internal representation of **data** and **state** of the **system**
  - often part of the model is connected to **file**(s) or **database**(s)
  - Ex) **Card** game - **Card**, **Deck**, **Player**
  - EX) **Bank** system - **Account**, **User**, **UserList**
- What it does
  - implements all the **functionality**
- Does not do
  - does not care about **which functionality** is used **when**, **how results** are **shown** to the user



MVC Pattern

- **Controller**
  - Classes that connect **model** and **view**
    - defines how user interface reacts to user input (events)
    - receives messages from **view** (where events come from)
    - sends messages to **model** (tells what data to display)
  - What it does
    - **Takes** user inputs, tells **model** what to **do** and **view** what to **display**
  - Does not do
    - does **not care how model implements** functionality, screen layout to **display results**



MVC Pattern

- **View**
  - ✓ Classes in your system that **display** the state of the model to the user
    - generally, this is your **GUI** (could also be a **text UI**)
    - should not contain crucial application data
    - **Different views** can represent the same data in different ways
      - Ex) Bar chart vs. pie chart
  - ✓ What it does
    - **display** results to user
  - ✓ Does not do
    - does not care **how** the **results** were **produced**, **when** to **respond** to user action

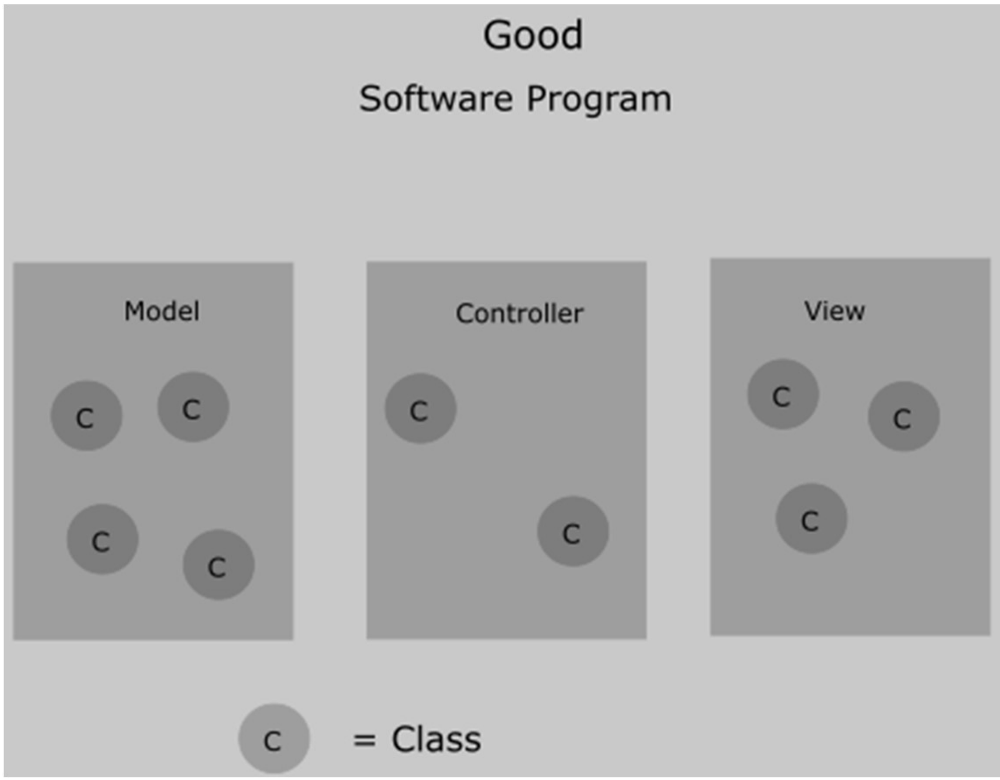
Advantages of MVC

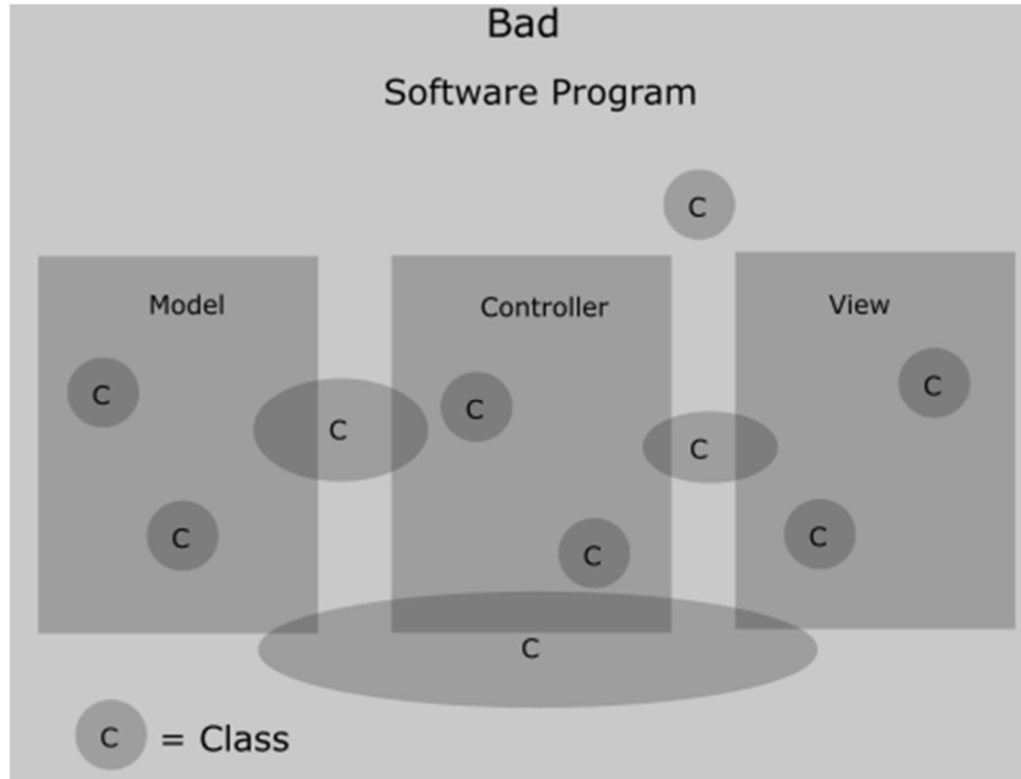
- Separating **Model** (Data Representation) from **View** (Data Presentation)
  - ✓ easy to add **multiple data presentations** for the same data,
  - ✓ facilitates adding **new types** of data presentation as technology develops.
  - ✓ Model and View components can vary independently enhancing maintainability, extensibility, and testability.

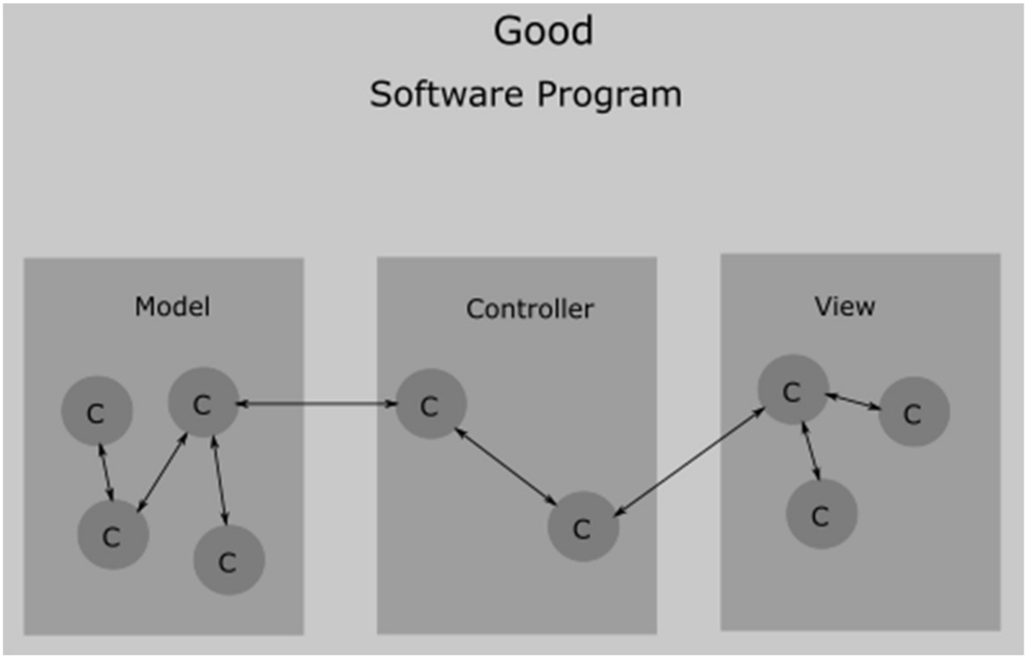
- Separating **Controller** (Application Behavior) from **View** (Application Presentation)
  - ✓ permits run-time selection of appropriate
    - – Views based on workflow, user preferences, or Model state.

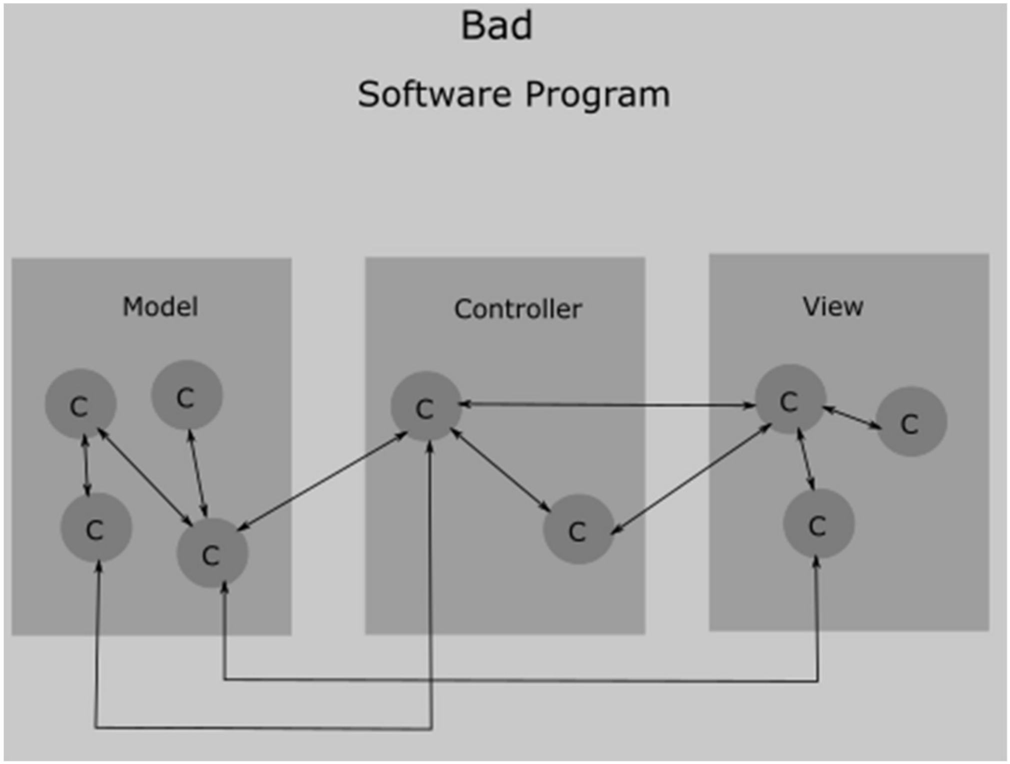
Advantages of MVC

- Separating **Controller** (Application Behavior) from **Model** (Application Representation)
  - ✓ allows **configurable mapping** of **user actions** on the Controller to application **functions** on the Model.


Good
Software Program
Model
Controller
View
C
= Class


Bad
Software Program
Model
Controller
View
C
= Class


Good
Software Program
Model
Controller
View
C
C
C
C
C
C
C
C
C


Bad
Software Program
Model
Controller
View
C
C
C
C
C
C
C
C
C
C

MVC: Pet Store

- Has set of modules, each tightly coupled internally, and loosely coupled between modules.
  - ✓ User Account
  - ✓ Product Catalog
  - ✓ Order Processing
  - ✓ Messaging
  - ✓ Inventory
  - ✓ Control
- Each module has an interface that defines the module's functional requirements and provides a place where third-party products may be integrated.

MVC: Pet Store

- **Model**
  - ✓ represents the structure of the data in the application, as well as application-specific operations on data
    - CartModel, InventoryModel, CustomerModel, and others
- **Views**
  - ✓ Java server pages (JSPs)
  - ✓ composed with templates and displayed in an HTML browser.
- **Controller**
  - ✓ Server-side java program (Servlet)
  - ✓ maps user input from the browser to request events, and forwards those events to the Shopping Client Controller
  - ✓ dispatches browser requests to other controller objects
    - Ex) ShoppingClientController.java, AdminClientController.java, and their related support classes.

View: JSP Example

View: ShoppingCart.jsp

- Java Server Pages (JSP)