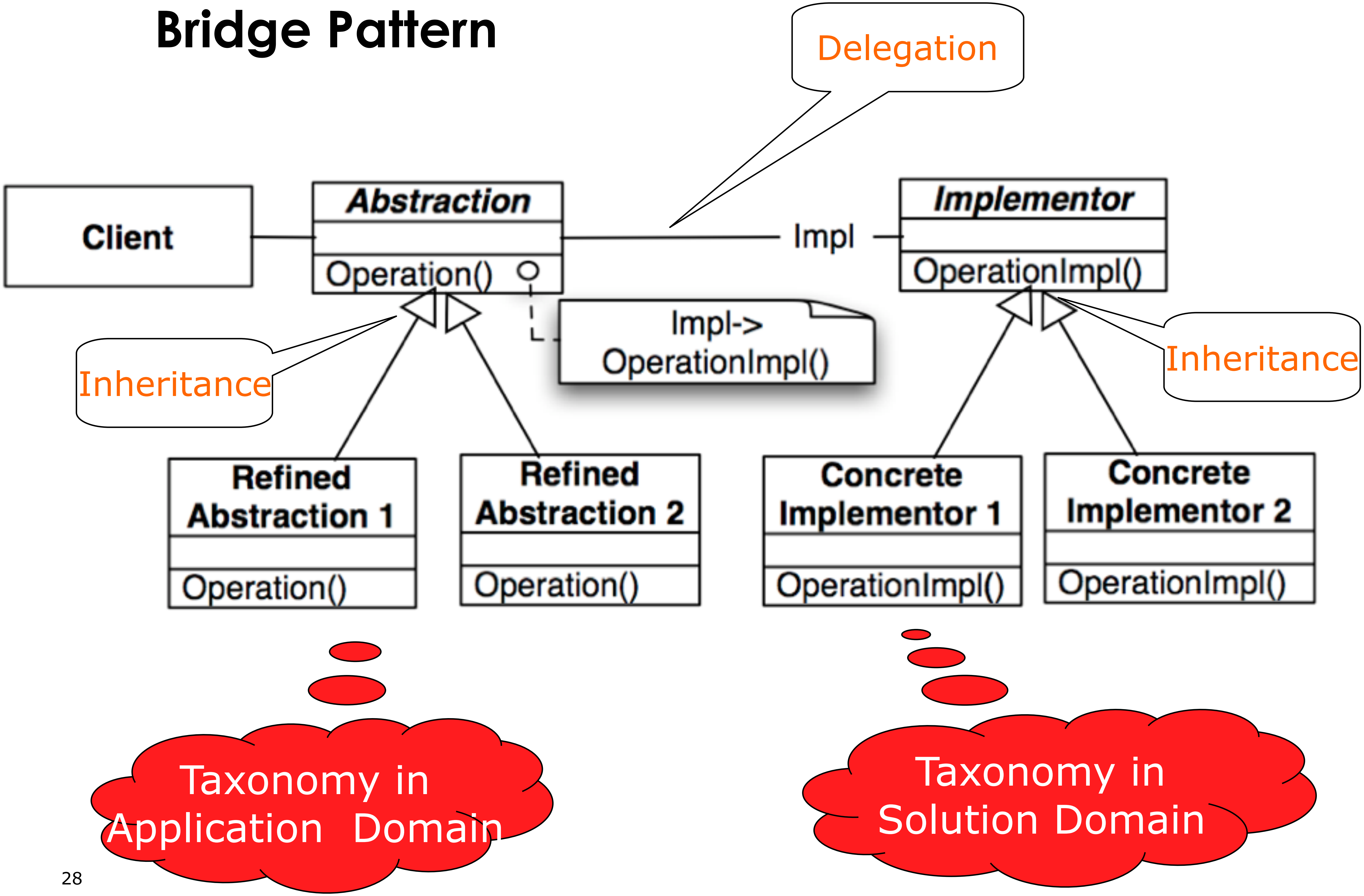
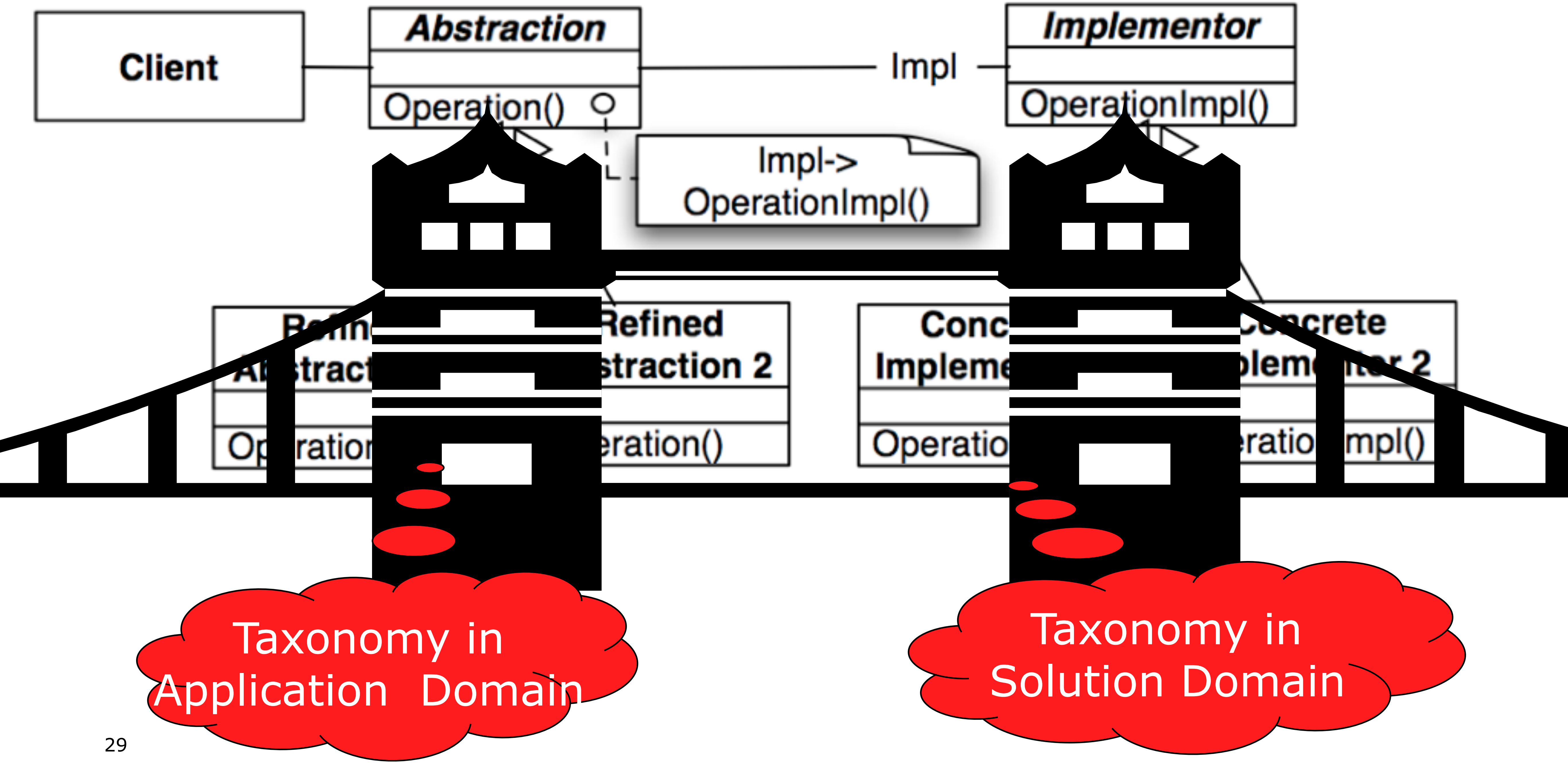
Bridge Pattern



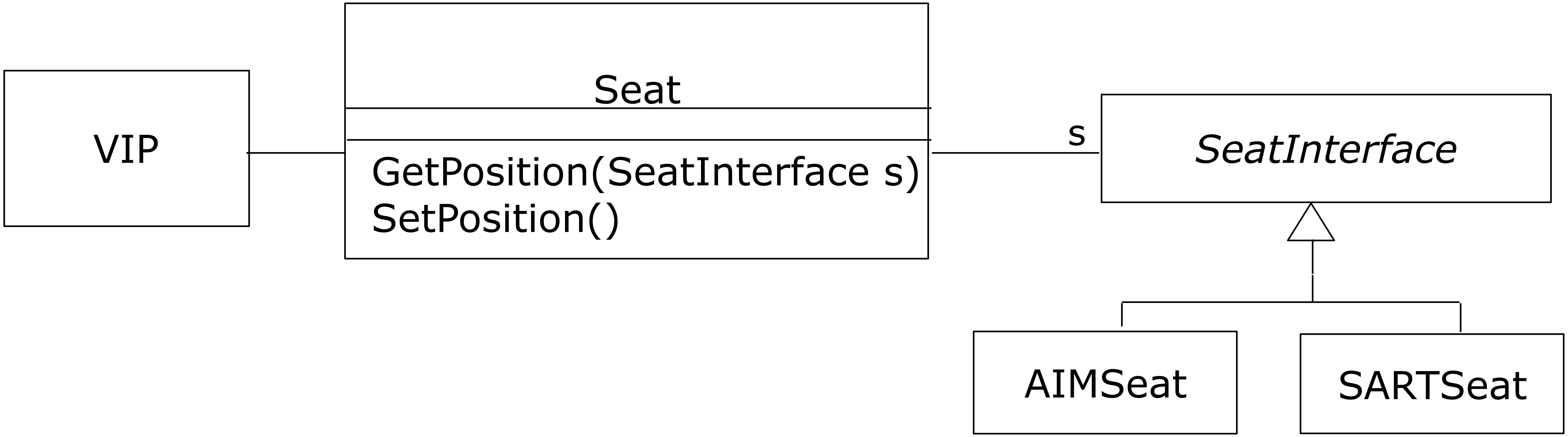
Why the Name Bridge Pattern?

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



Using a Bridge

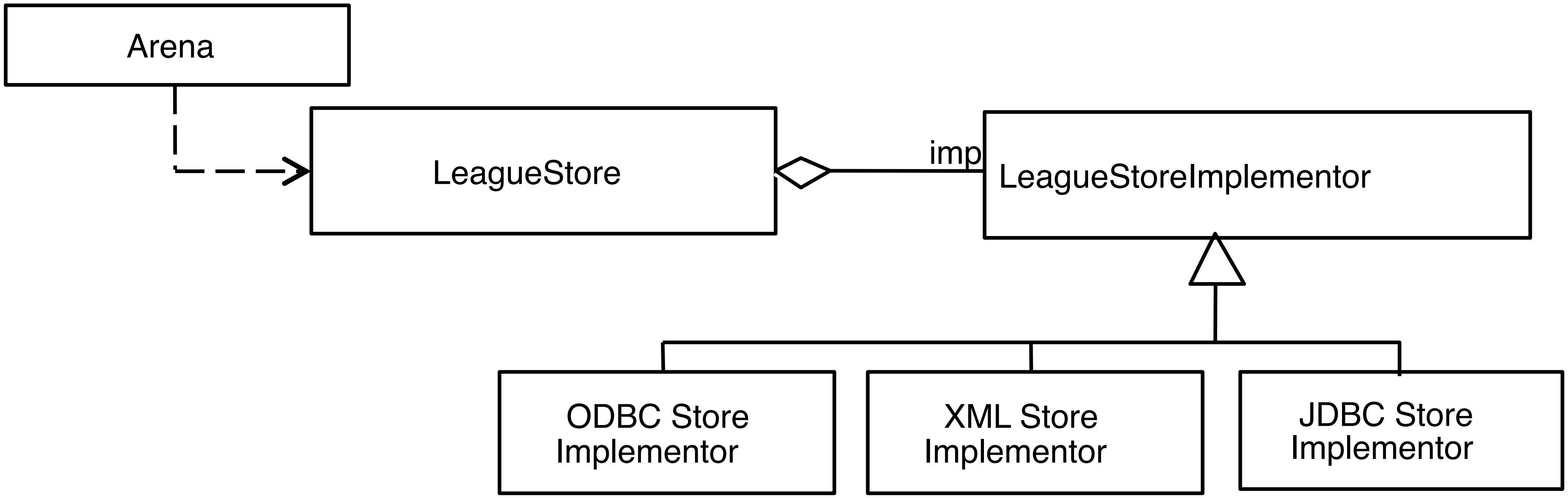
- The bridge pattern can be used to provide multiple implementations under the same interface
  - Example: Interface to a component that is incomplete, not yet known or unavailable during testing
    - GetPosition() is needed by VIP, but the class Seat is only available by two simulations (AIMSeat and SARTSeat). To switch between these, the bridge pattern can be used:



Seat Implementation

```
public interface SeatInterface {
  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
 }
  ...
}
public class Seat{
  public int GetPosition(SeatInterface s) {
                  s.getPosition()
}
  ...
}
```

Another use of the Bridge Pattern: Supporting multiple Database Vendors



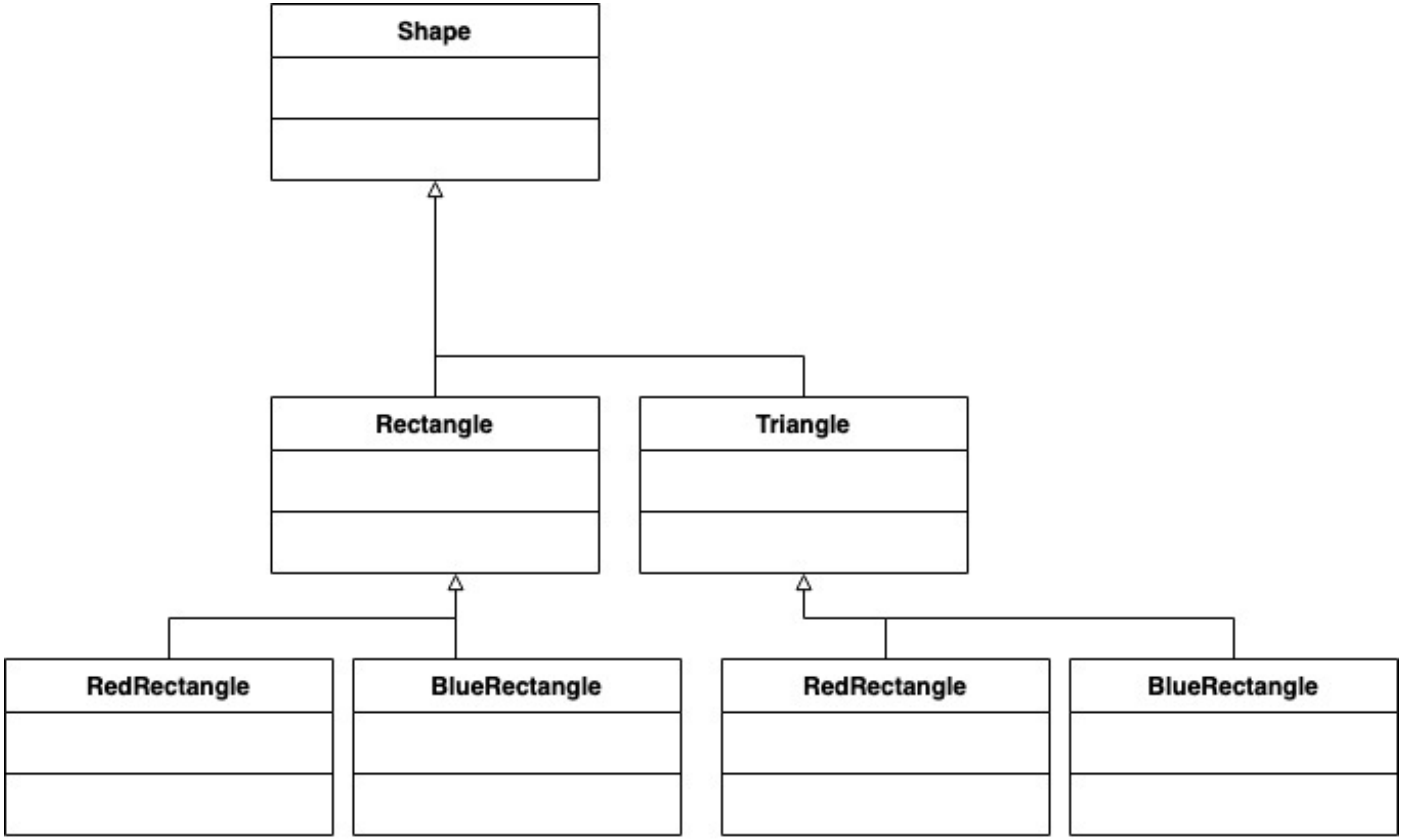
The Bridge Pattern allows to postpone Design Decisions to the startup time of a system

- 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).

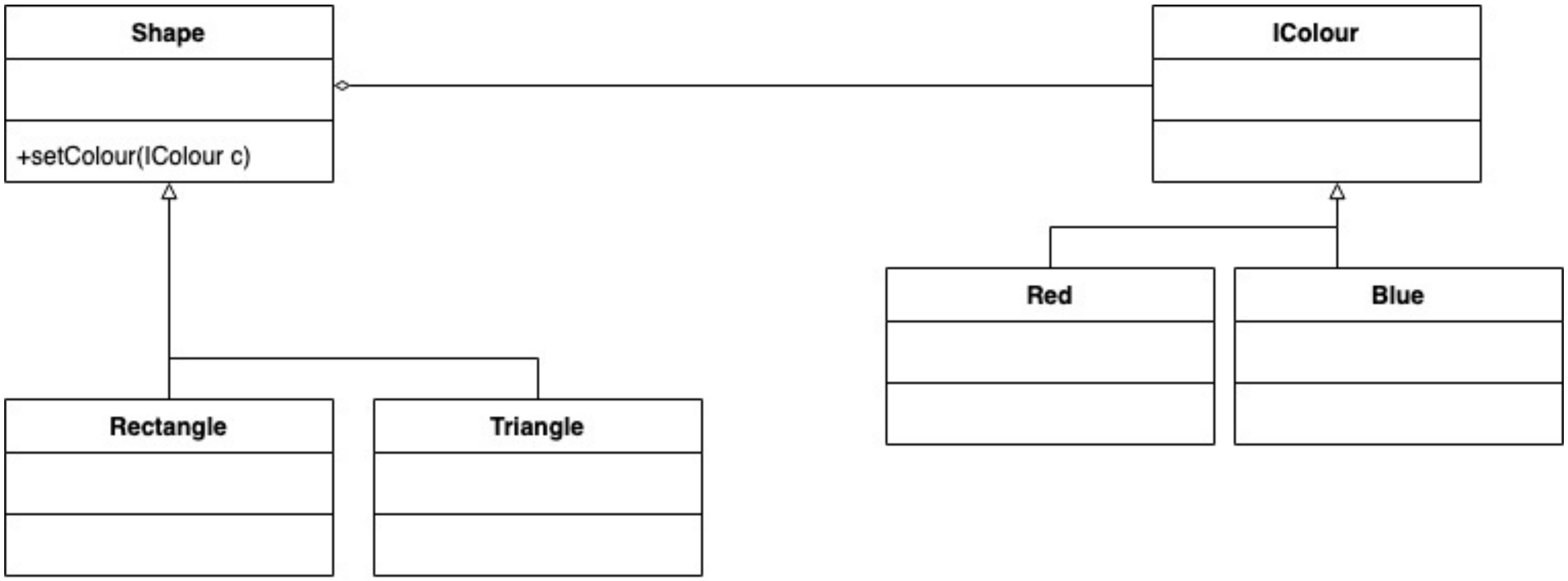
Adapter vs Bridge

- 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
    - Green field engineering of an "extensible system"
    - 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".

How to solve this problem?

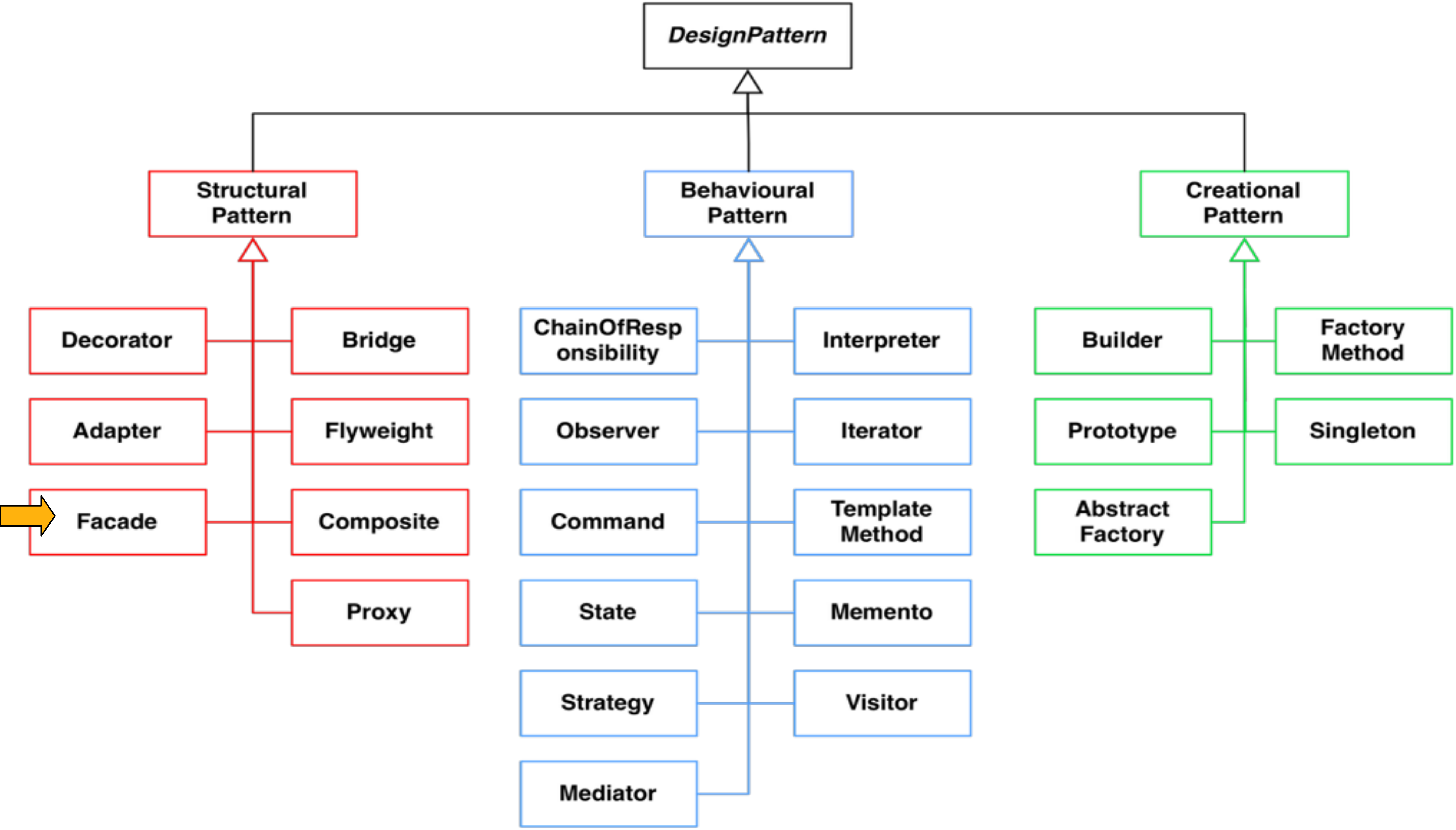


How to solve this problem?



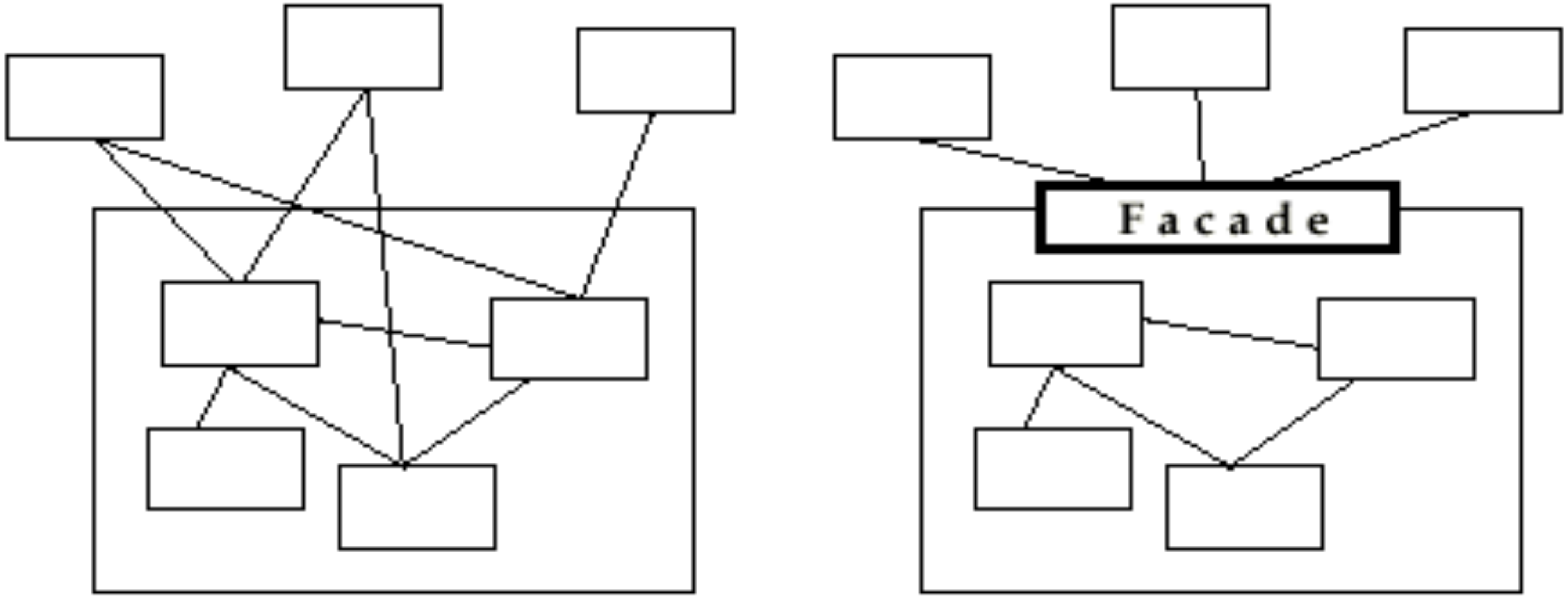
Using the Bridge pattern

Taxonomy of Design Patterns (23 Patterns)



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 (i.e. it abstracts out the gory details).



Subsystem Design with Façade, Adapter, Bridge

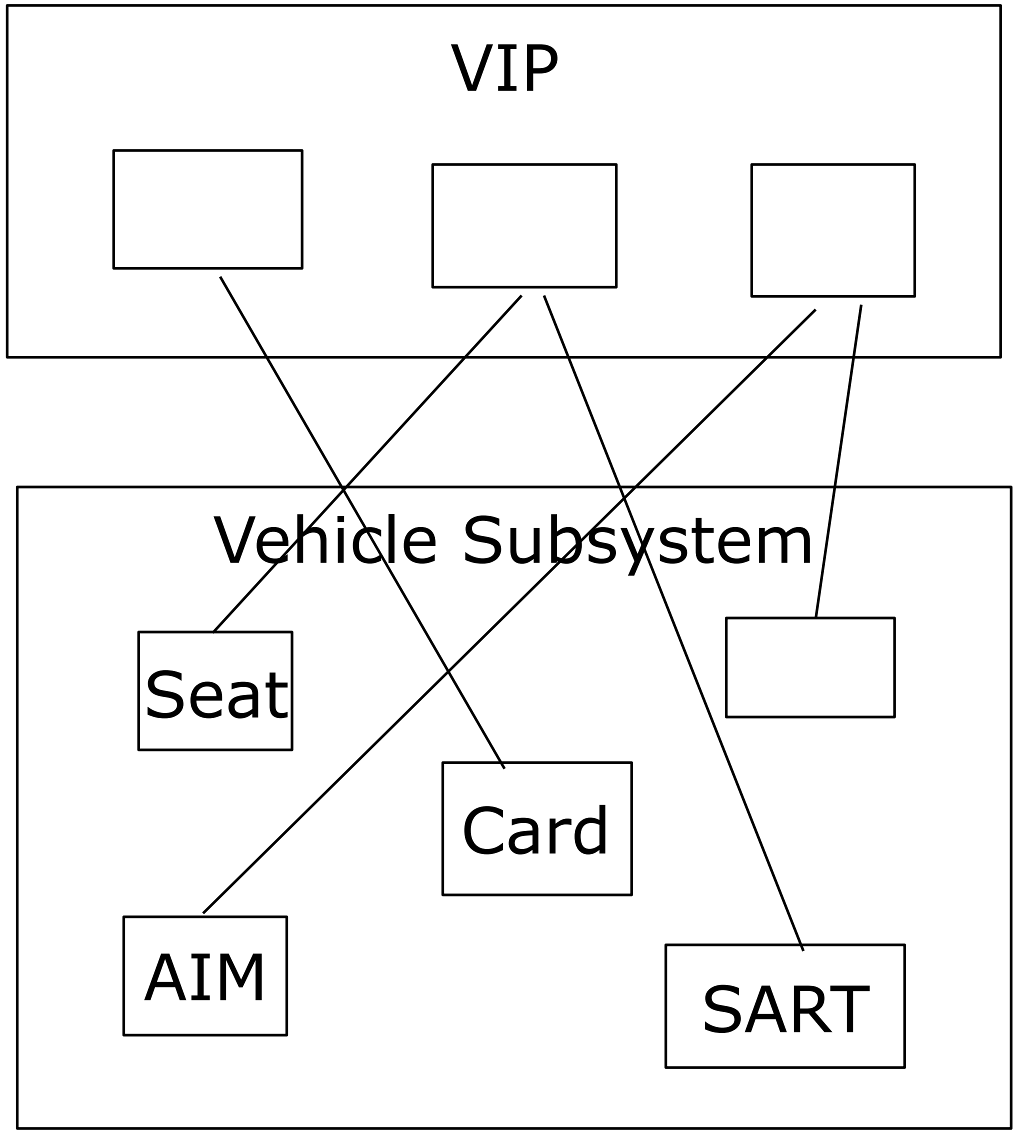
- The ideal structure of a subsystem consists of
  - an interface object
  - a set of entity objects modeling real entities or existing systems
    - Some of these entity objects are interfaces to existing systems
  - one or more control objects
- We can use design patterns to realize this subsystem structure
- 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!

Good Design with Façade, Adapter and Bridge

- A façade should be offered by all subsystems in a software system which provide a set of services
  - The façade delegates requests to the appropriate components within the subsystem. The façade usually does not have to be changed, when the components are changed
- The adapter pattern should be used to interface to existing components and legacy systems
  - Example: A smart card software system should use an adapter for a smart card reader from a specific manufacturer
- The bridge pattern should be used to interface to a set of objects with a large probability of change
  - When the full set of objects is not completely known at analysis or design time (-> Mock Object Pattern)
  - When there is a chance that a subsystem or component must be replaced later after the system has been deployed and client programs use it in the field.

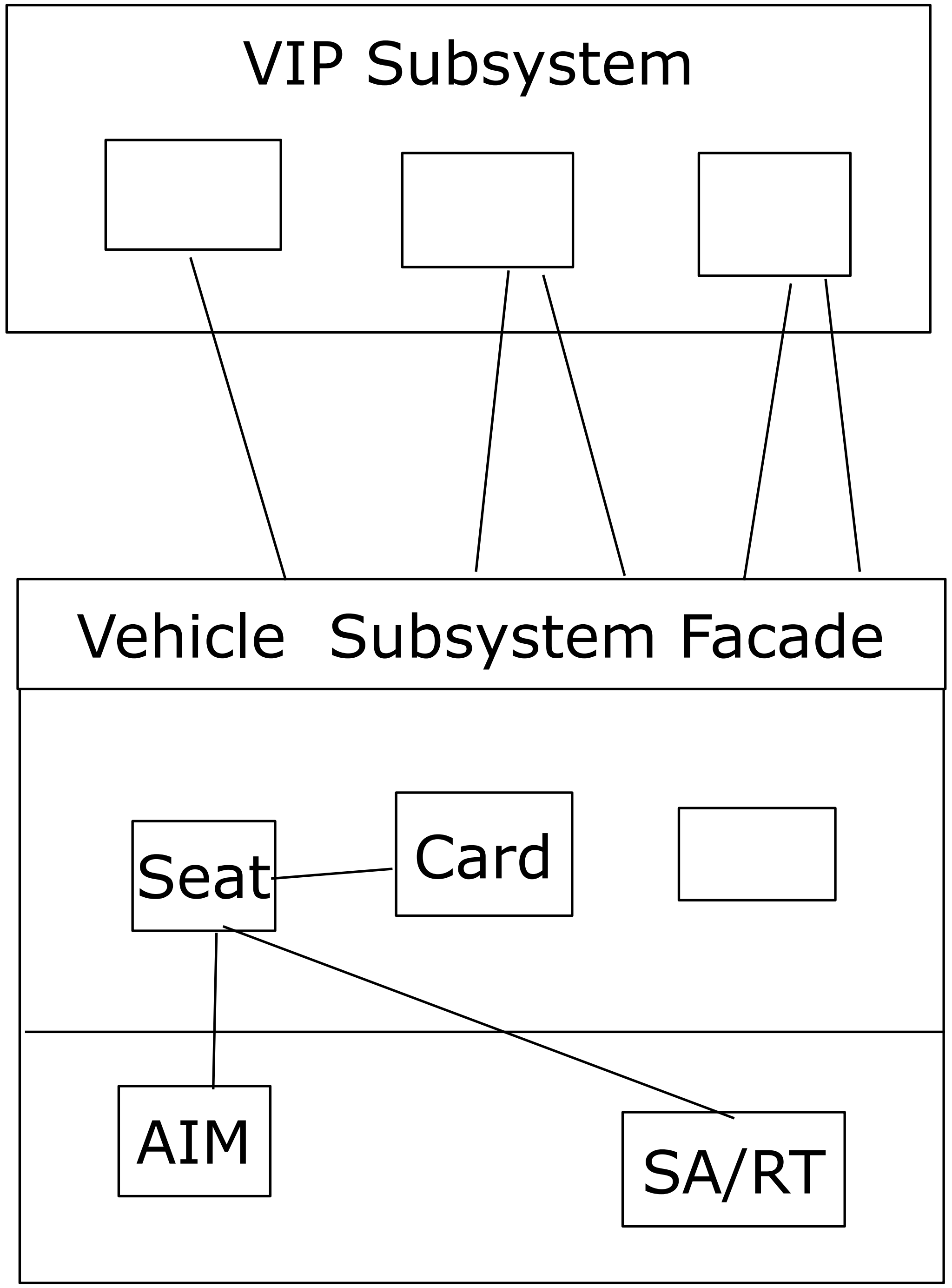
Design Example

- Subsystem 1 VIP can call on any component or class operation look in Subsystem 2 (Vehicle Subsystem).



Realizing an Opaque Architecture with a Facade

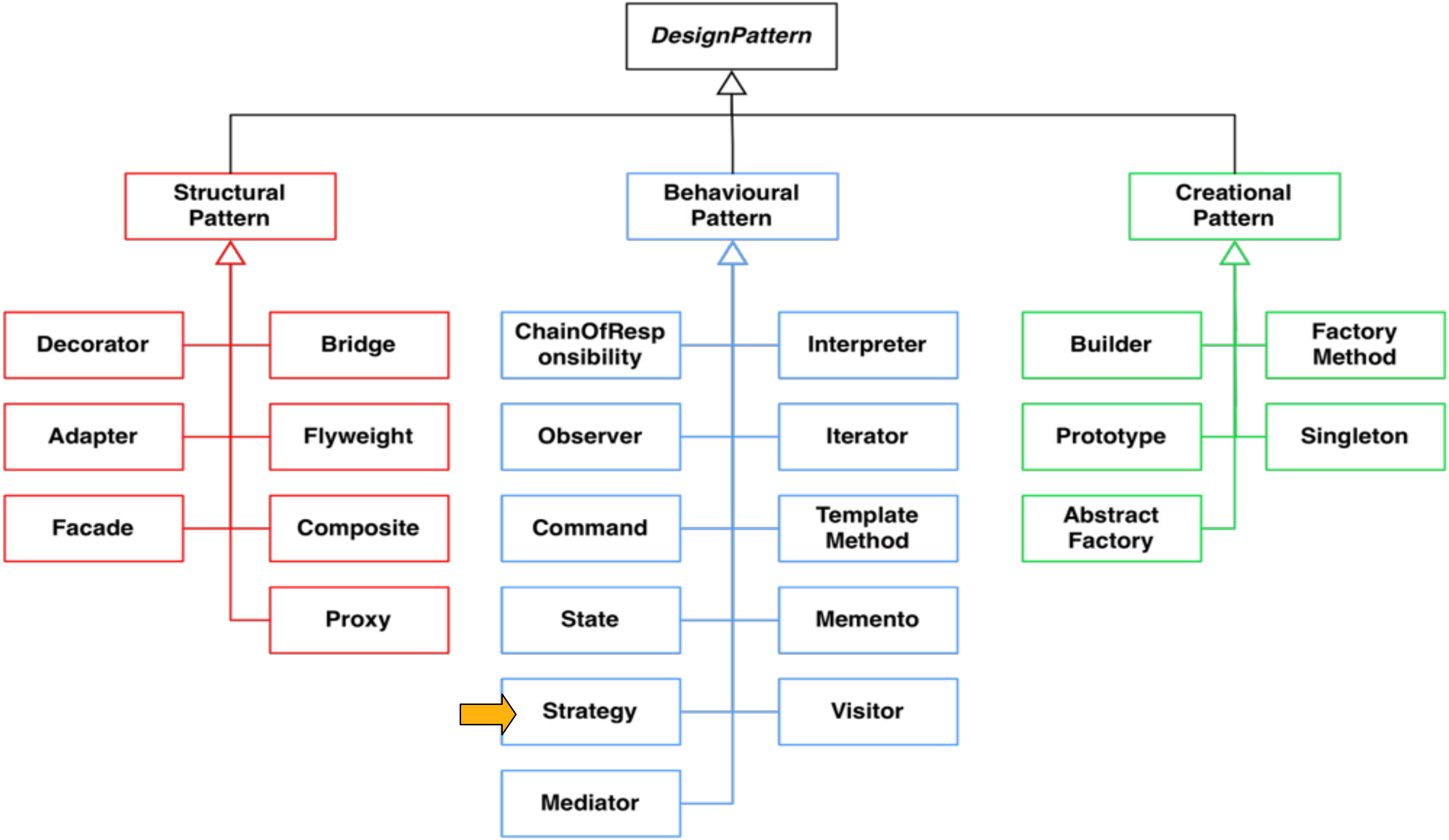
- The Vehicle Subsystem decides exactly how it is accessed
- No need to worry about misuse by callers



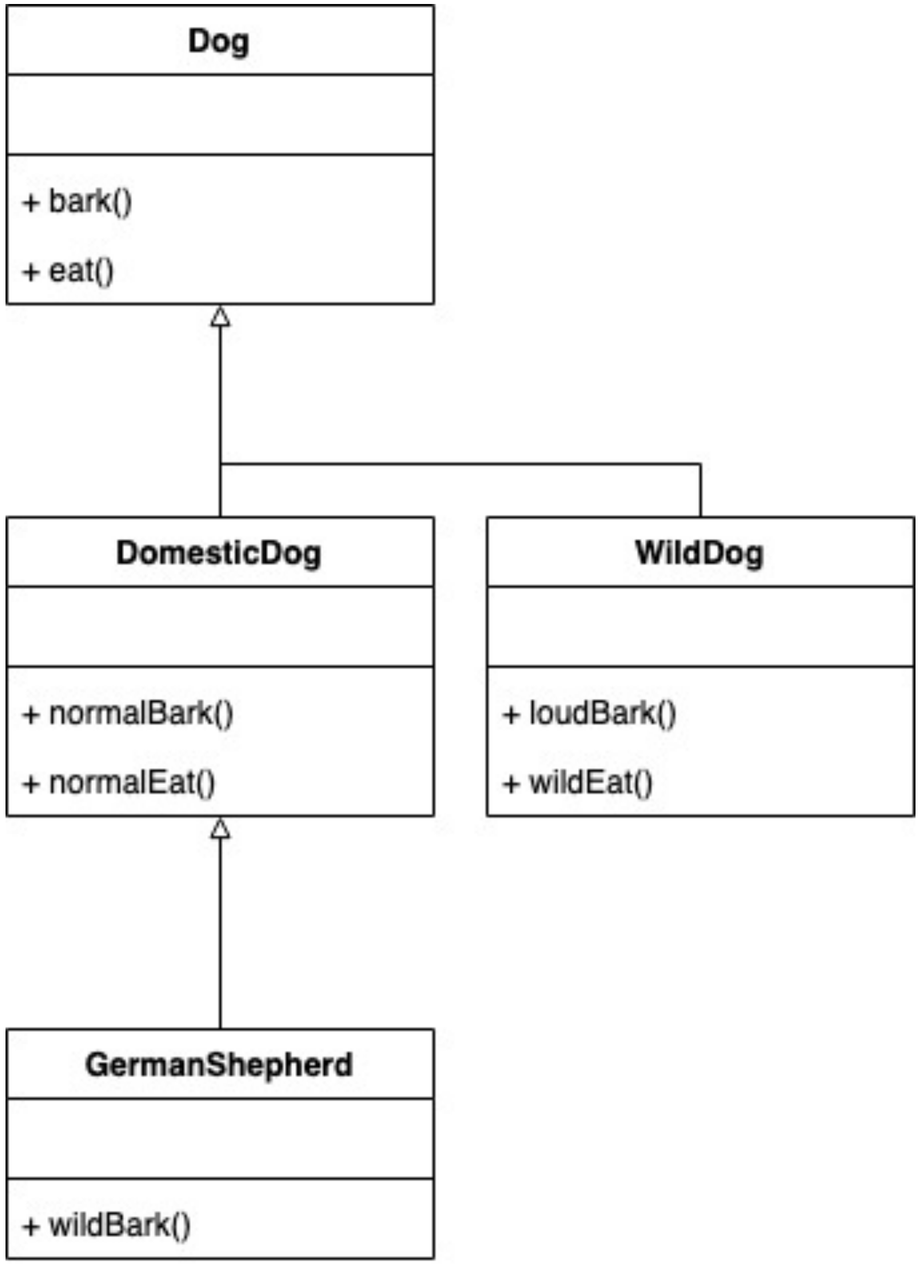
Facade example

```
public class Rectangle{ public void draw();}

public class Square{ public void draw();}

public class Facade{
    Rectangle r = new Rectangle();
    Square s = new Square();
    drawRectangle() { ….}

    drawSquare() { ….}

    }
```

Taxonomy of Design Pattern

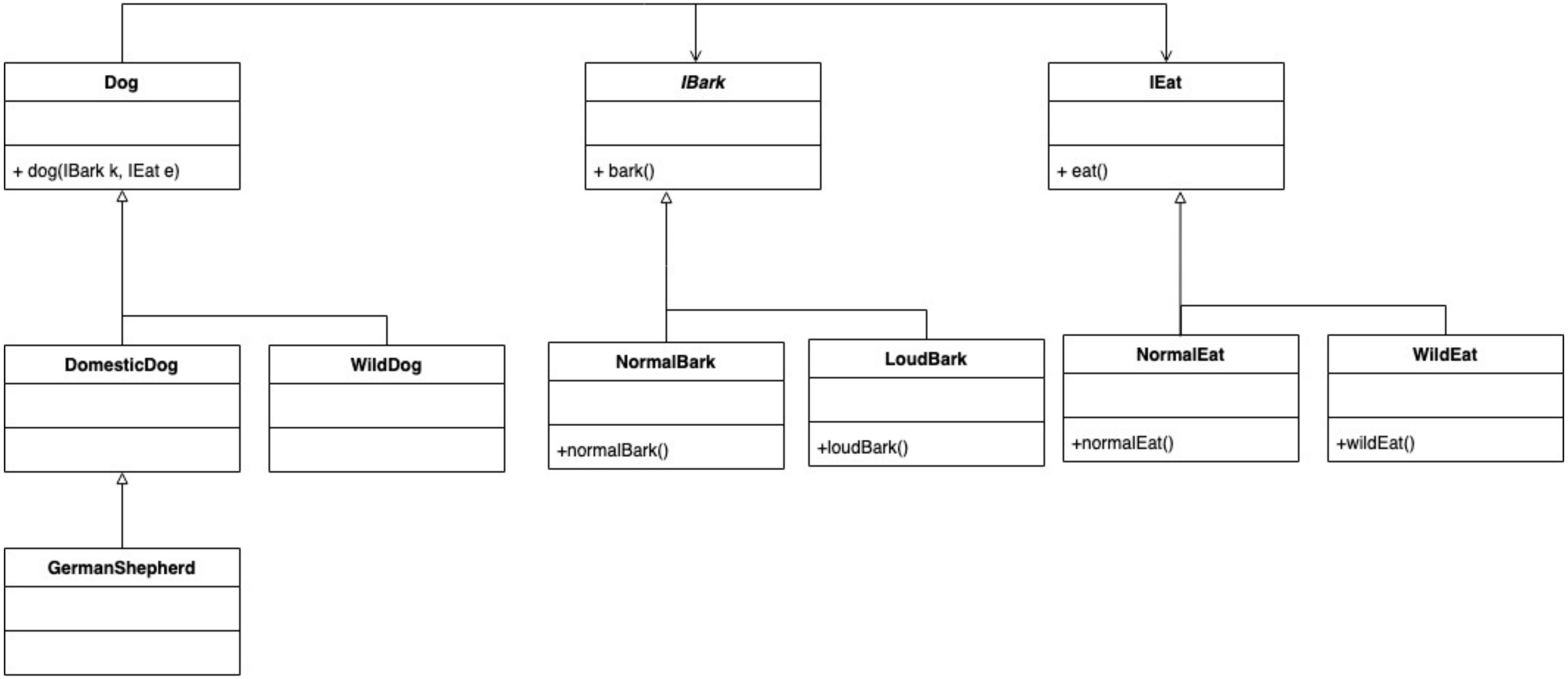


Strategy Pattern



How to inherit wildEat in GermanShepherd?
Java does not allow multiple inheritance!!

Strategy Pattern

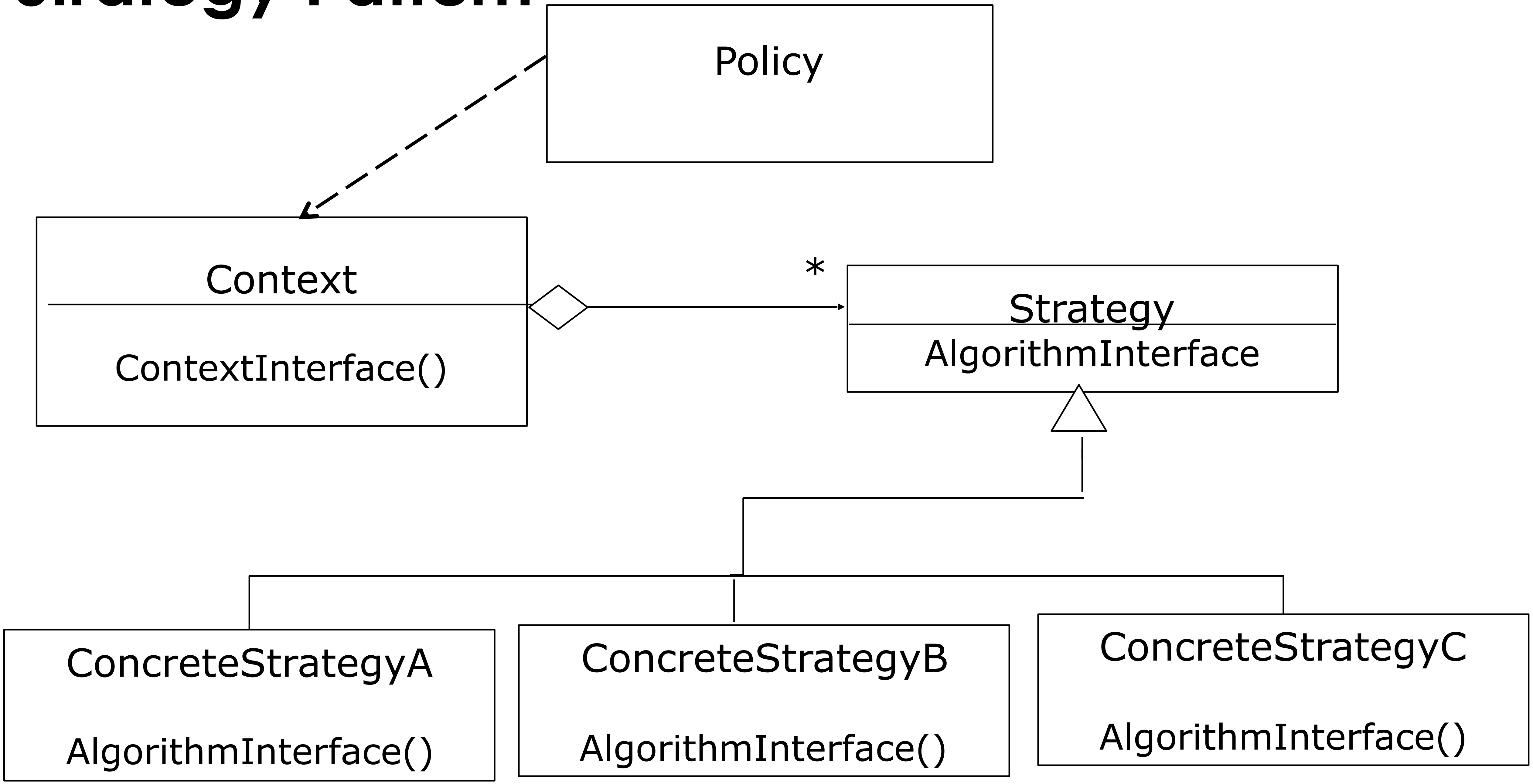


Strategy pattern solves this problem by allowing to model the Eat and Bark behaviours to be modelled as classes

Strategy Pattern

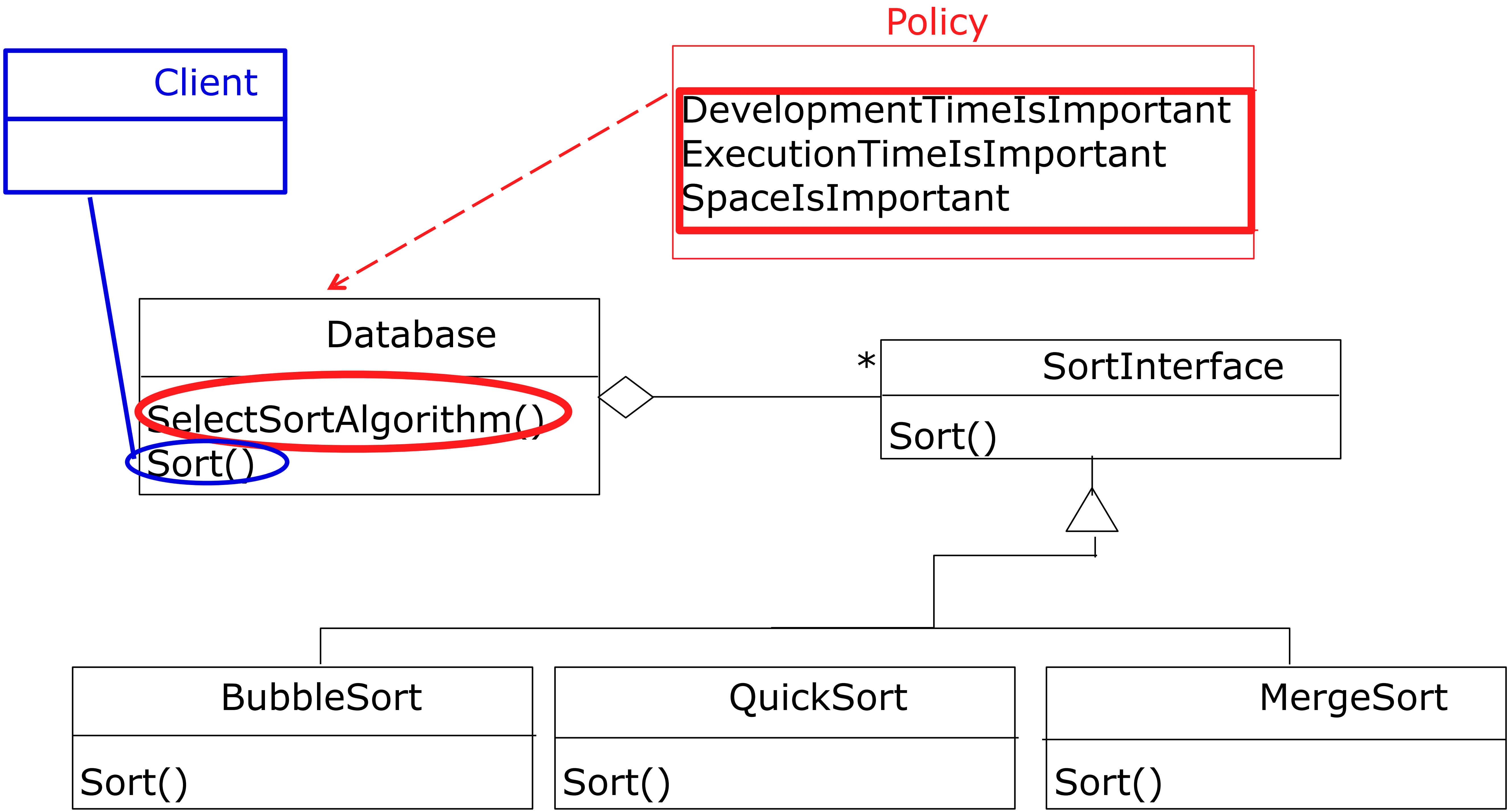
- Different algorithms exists for a specific task
  - We can switch between the algorithms at run time
- Examples of tasks:
  - Different collision strategies for objects in video games
  - Parsing a set of tokens into an abstract syntax tree (Bottom up, top down)
  - Sorting a list of customers (Bubble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

Strategy Pattern

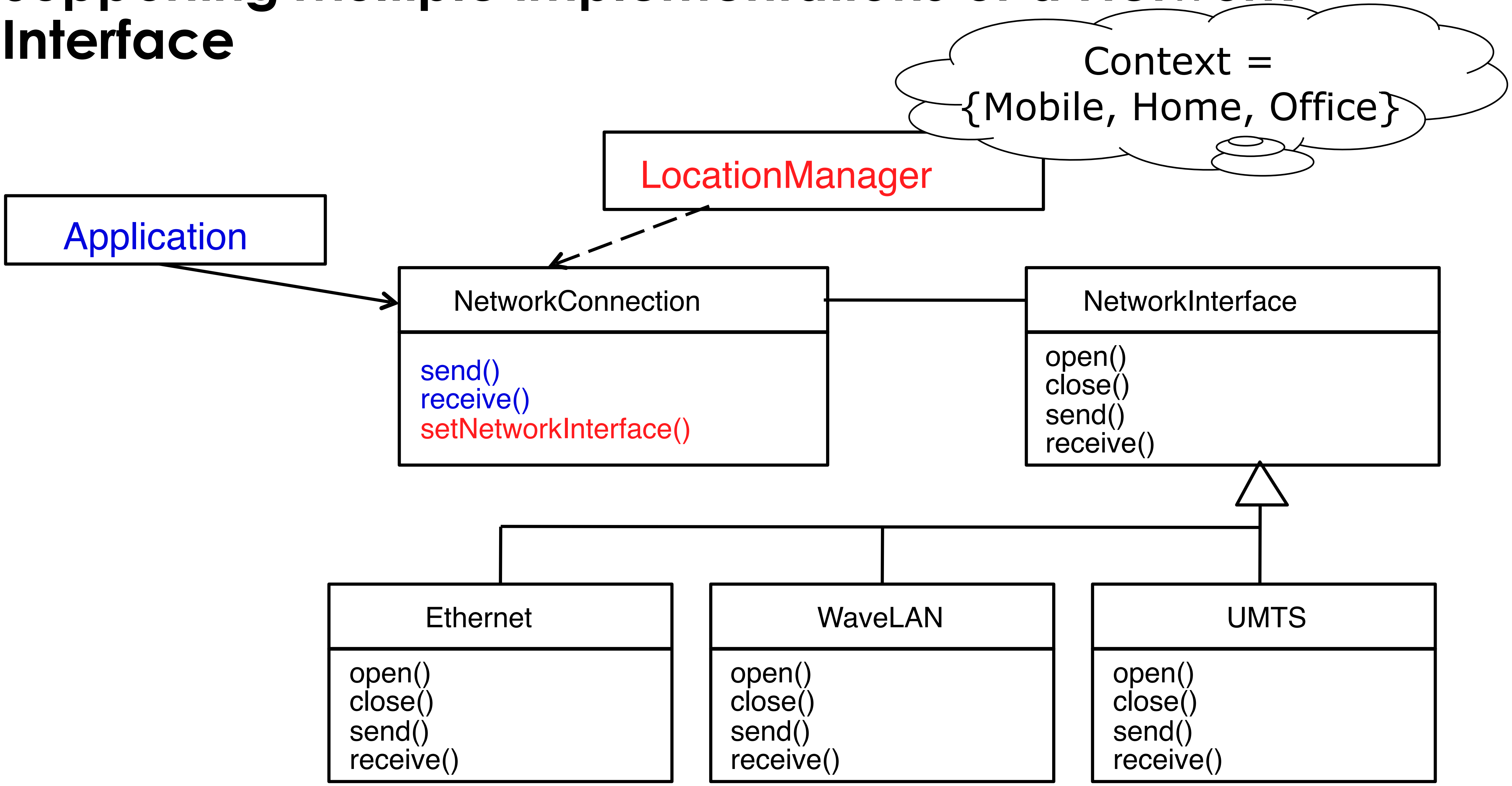


Policy decides which ConcreteStrategy is best in the current Context.

Using a Strategy Pattern to Decide between Algorithms at Runtime



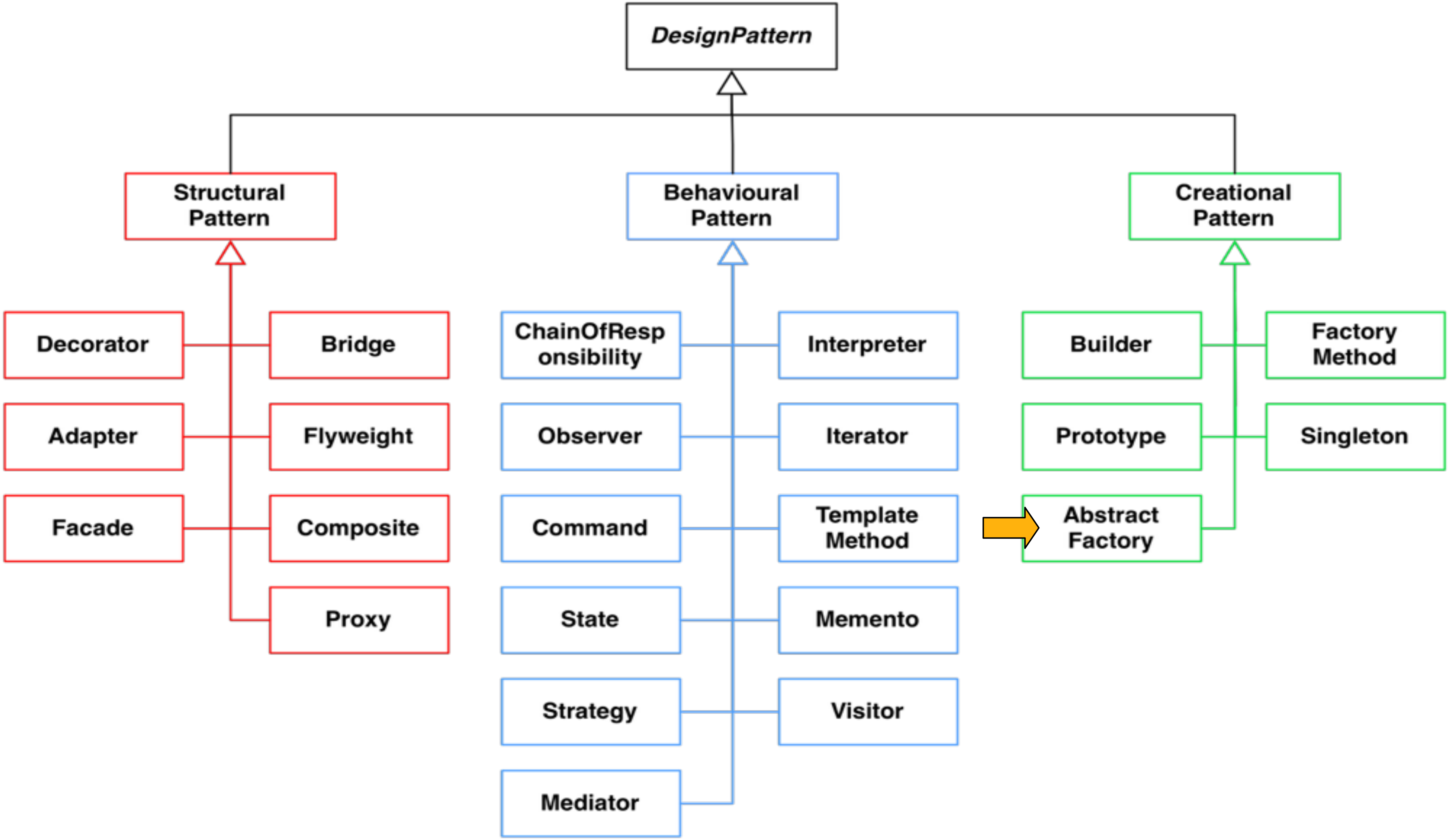
Supporting Multiple implementations of a Network Interface



Strategy example

```
public interface Strategy { public int doOperation(int num1, int num2); }

public class OperationAdd implements Strategy{
  public int doOperation(int num1, int num2) { ..}
}
public class OperationSubstract implements Strategy{
  public int doOperation(int num1, int num2) { ..}
}

public class Context {
   private Strategy strategy;

   public Context(){
   // do stuff to choose strategy
      this.strategy = strategy;
   }

   public int executeStrategy(int num1, int num2){
      return strategy.doOperation(num1, num2);
   }
}
```

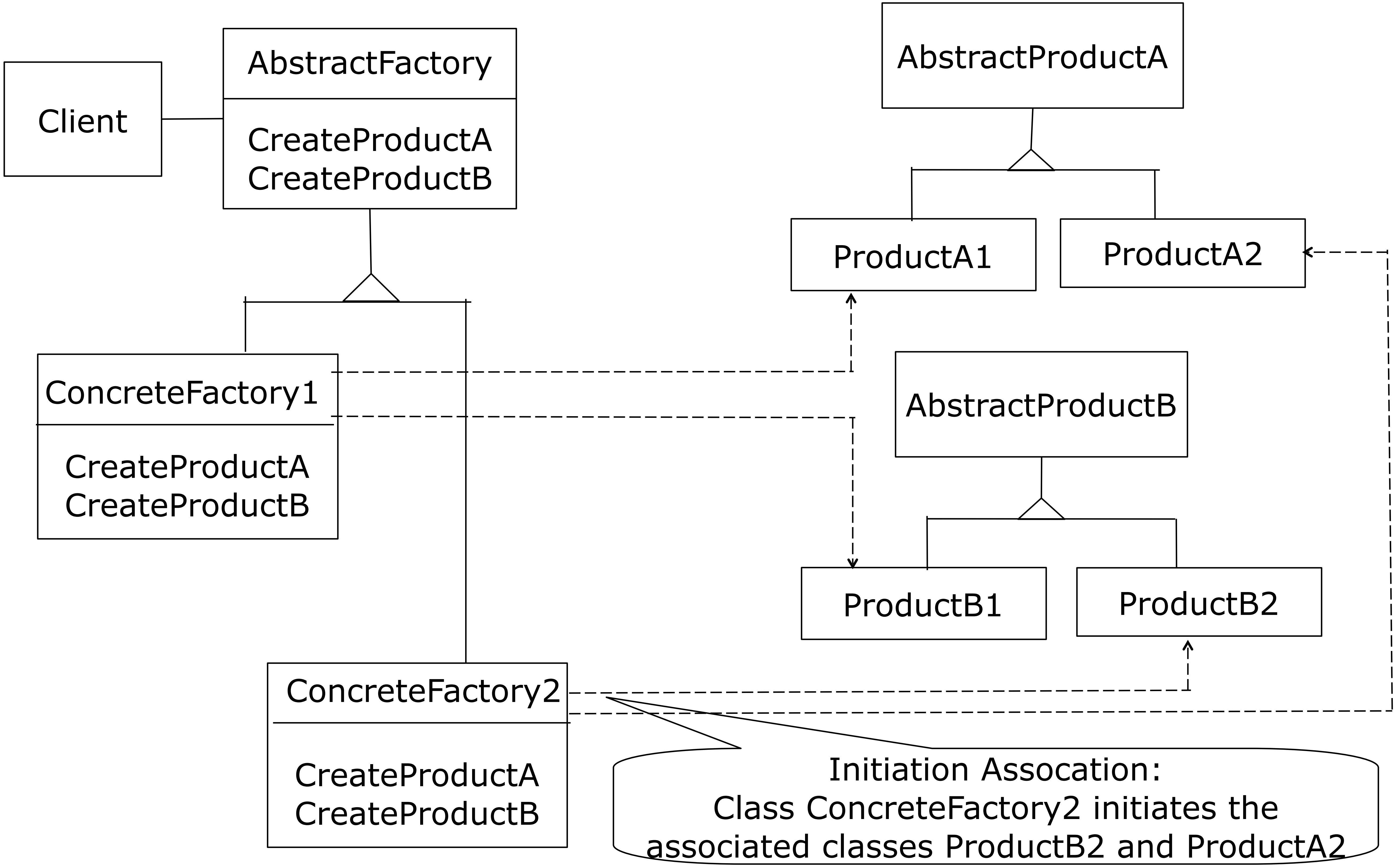
Taxonomy of Design Patterns



Abstract Factory Pattern Motivation

- Consider a user interface toolkit that supports multiple looks and feel standards for different operating systems:
  - 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



Client

AbstractFactory

CreateProductA
CreateProductB

ConcreteFactory1

CreateProductA
CreateProductB

ConcreteFactory2

CreateProductA
CreateProductB

AbstractProductA

ProductA1

ProductA2

AbstractProductB

ProductB1

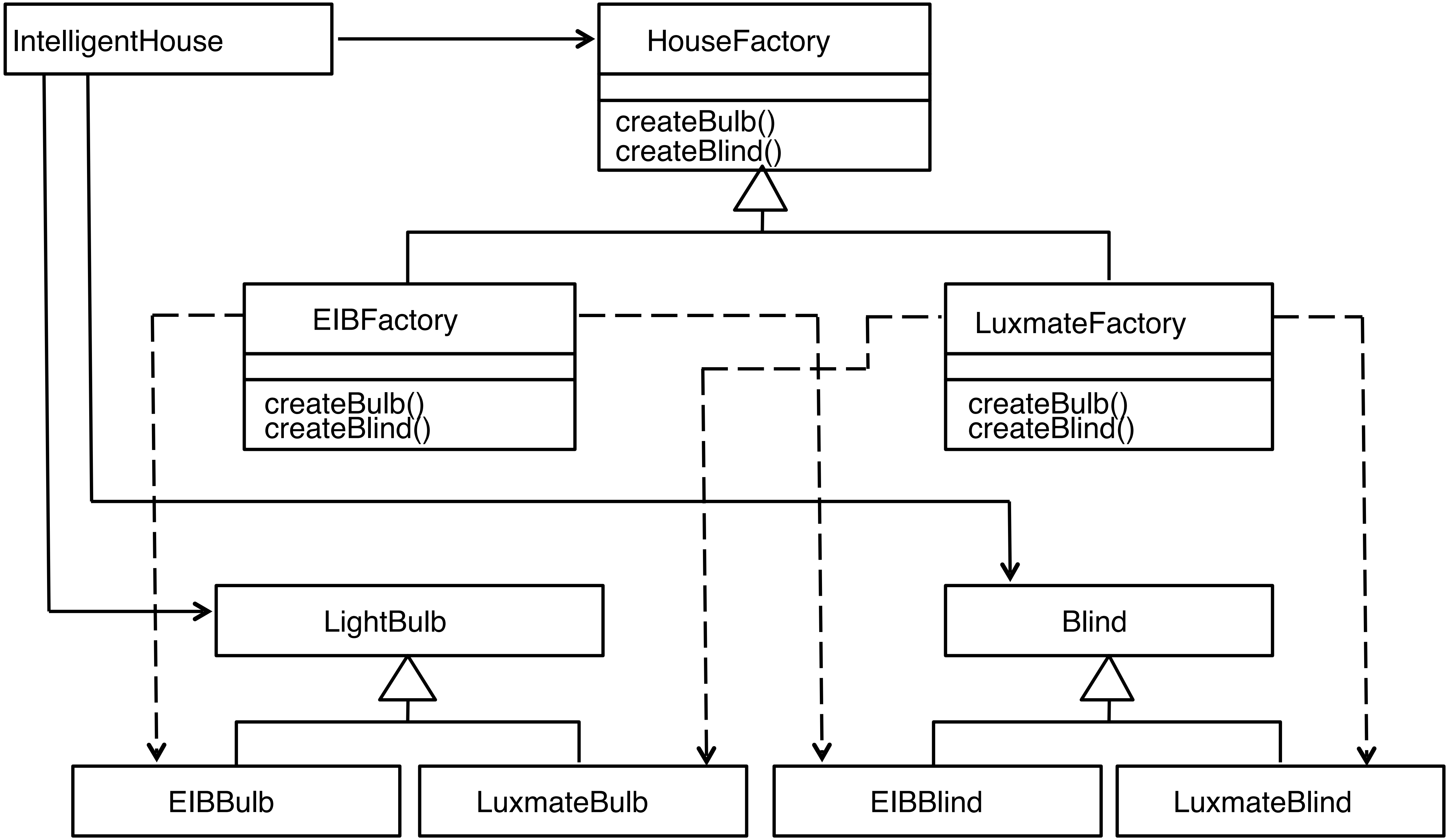
ProductB2

Initiation Assocation:
Class ConcreteFactory2 initiates the
associated classes ProductB2 and ProductA2

Applicability for Abstract Factory Pattern

- Independence from Initialization or Representation
- Manufacturer Independence
- Constraints on related products
- Cope with upcoming change

Example: A Facility Management System for a House



Clues in Nonfunctional Requirements for the Use of Design Patterns

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

Clues in Nonfunctional Requirements for use of Design Patterns (2)

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