Object Oriented Analysis & Design
CSC 2210

Dr. Akinul Islam Jony
Associate Professor
Department of Computer Science, FSIT
American International University - Bangladesh (AIUB)

akinul@aiub.edu

Design Patterns

>> What is Design Patterns?
>> Types of Design Patterns
>> What is Good Design?
>> Adapter Pattern
>> Strategy Pattern
>> Observer Pattern
>> Factory Pattern
>> Singleton Pattern
>> Façade Pattern

What is Design Patterns?

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.

What is Design Patterns?

Pattern Definition

<u>Original Definition (Christopher Alexander):</u>

A pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution.

<u>Refined Definition (Dick Gabriel):</u>

A three-part rule, which expresses

- a relation between a certain context, and
- a certain system of forces which occurs repeatedly in that context, and
- a certain software configuration which allows these forces to resolve themselves.

- From: Richard Gabriel, Patterns of Software: Tales From the Software Community.

What is Design Patterns?

<u>What makes Design Patterns Good?</u>

→ They are generalizations of detailed design knowledge from existing systems
→ They provide a shared vocabulary to designers
→ They provide examples of reusable designs
- Polymorphism (Inheritance, sub-classing)
- Delegation (or aggregation)

Types of Design Patterns

There are 3 Types of Design Patterns ("GoF Patterns")

Structural Patterns:

- Reduce coupling between two or more classes
- Introduce an abstract class to enable future extensions
- Encapsulate complex structures

Behavioral Patterns:

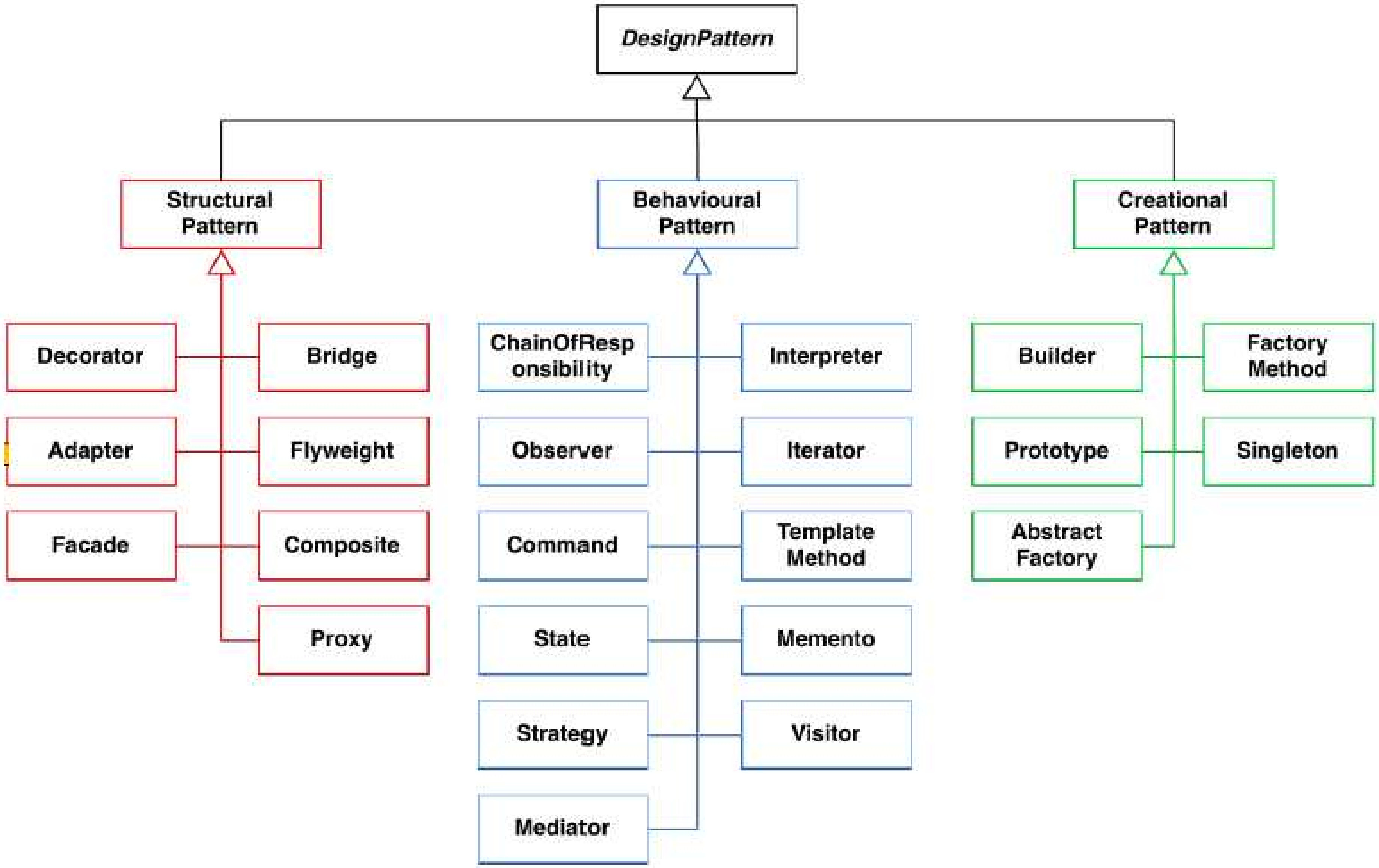
- Allow a choice between algorithms and the assignment of responsibilities to objects ("Who does what?")
- Simplify complex control flows that are difficult to follow at runtime

Creational Patterns:

- Allow a simplified view from complex instantiation processes
- Make the system independent from the way its objects are created, composed and represented.

[**GoF**] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.
Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 2005 (1st ed. 1994)

Types of Design Patterns



Design Patterns Taxonomy (23 Patterns)

What is Good Design?

→ Good design reduces complexity

→ Good design is prepared for change

→ Good design provides low coupling

→ Good design provides high cohesion

What is Good Design?

Coupling:

Measures the dependencies between subsystems

Cohesion:

Measures the dependencies among classes within a subsystem

What makes a good design?

Low coupling:

- The subsystems should be as independent of each other as possible
- A change in one subsystem does not affect any other subsystem

High cohesion:

- A subsystem should contain classes which depend heavily on each other.

Adapter Pattern

→Connects incompatible components

→ Allows the simplified reuse of existing components

→ Converts the interface of an existing component into another interface expected by the calling component

→ Useful in interface engineering projects and in reengineering projects
- Used to provide a new interface for (the ugly interface of) legacy systems

→ Also known as a wrapper.

Putting a Square Peg in a Round Hole!

Adapter Pattern

Main use:

Provide access to legacy systems

Legacy System:

A legacy system is an old system that continues to be used, even though newer technology or more efficient methods are now available

Reason for the continued use of Legacy System:

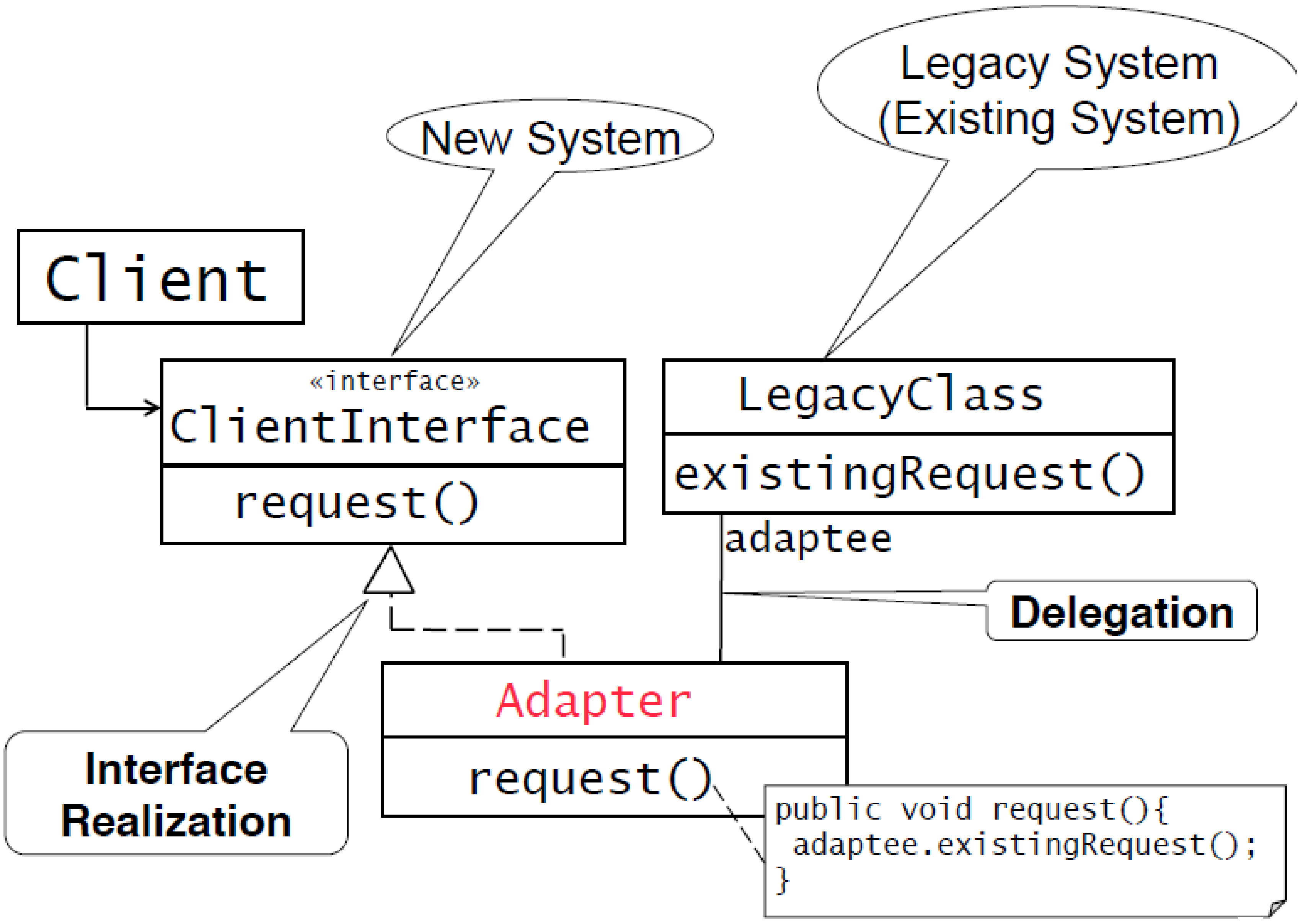
→ **System Cost:**
The system still makes money, the cost of designing a new system with the same functionality is too high

→ **Pragmatism:**
The system is installed and working

→ **Poor Engineering (or Poor Management?):**
The system cannot not be changed because the compiler is no longer available or source code has been lost

→ **Availability:**
The system requires 100%availability, cannot simply be taken out of service to replace it with a new system.

Adapter Pattern

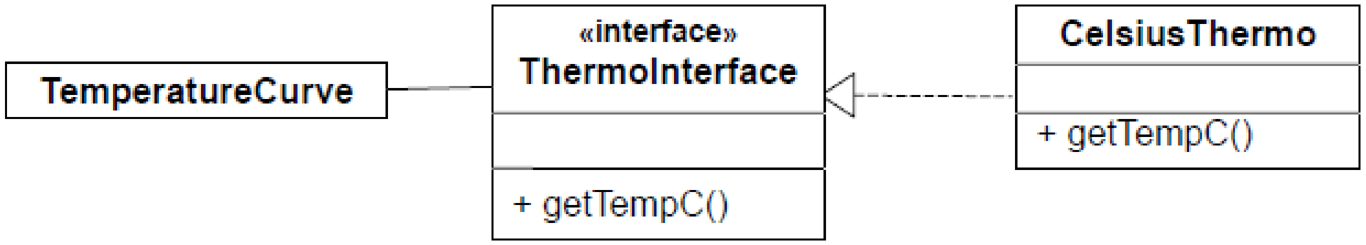


```
public void request(){
 adaptee.existingRequest();
}
```

Adapter Pattern: **Example**

Problem Statement

→ Suppose you need to reliably read the outside temperature for the last ***n*** hours (temperature curve) in Celsius. You have a fancy digital thermometer with software implemented in Java. The program uses a ***ThermoInterface*** which provides the temperature in Celsius. It connects to the outside thermometer which runs software containing a class called ***CelsiusThermo***!

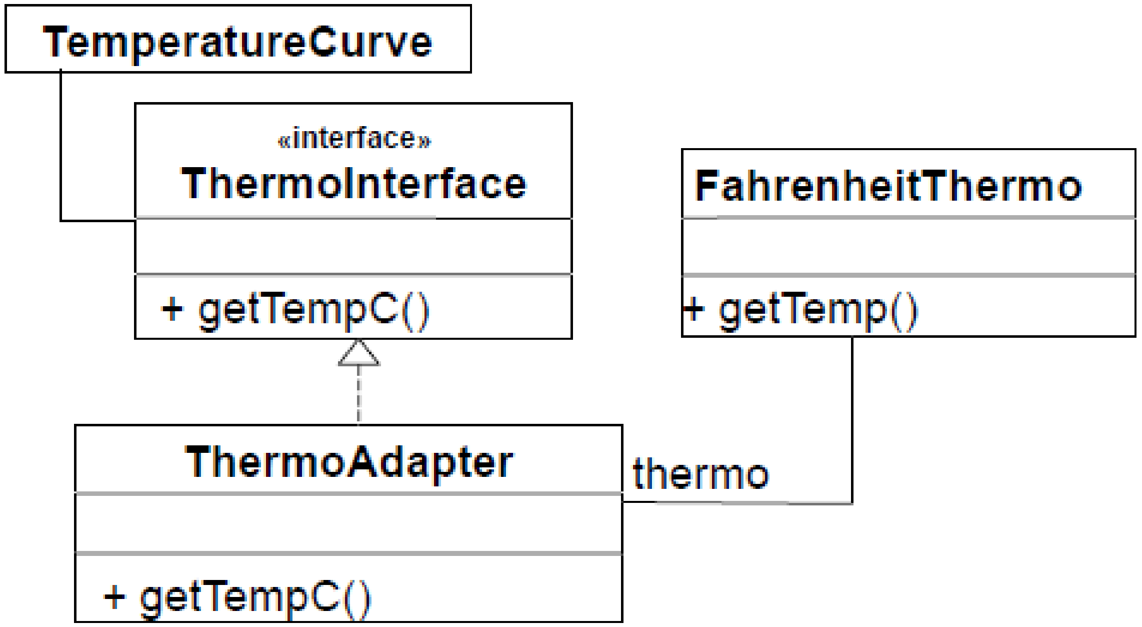


→ Somebody stepped on your outside Celsius thermometer (***CelsiusThermo***) and broke it. There is one more thermometer on the expedition, but this measures the temperature in **Fahrenheit**.

→ Write an **adapter** that solves the problem by reusing the code from the Fahrenheit thermometer (FahrenheitThermo).

*tempCelsius = (tempFahrenheit -32.0) * (5.0/9.0)*

Adapter Pattern: **Example**



```
public class ThermoAdapter implements ThermoInterface {

  private FahrenheitThermo thermo;

  public ThermoAdapter() {
    thermo = new FahrenheitThermo();
  }

  public double getTempC() {
    return (thermo.getTemp()-32.0) * (5.0/9.0);
  }
}
```

Figure: Solution using Adapter Pattern

Adapter Pattern

There are two types of Adapters: Objects Adapters and Class Adapter

Objects Adapters:

→ Based on Delegation

→Object adapters use a compositional technique to adapt one interface to another. The adapter inherits the target interface that the client expects to see, while it holds an instance the adaptee. When the client calls the request() method on its target object (the adapter), the request is translated into the corresponding specific request on the adaptee. Object adapters enable the client and the adaptee to be completely decoupled from each other. Only the adapter knows about both of them.

Class Adapter

→ Class Adapter uses inheritance and can only wrap a class. It cannot wrap an interface since by definition it must derive from some base class.

→ It means that instead of delegating the calls to the Adaptee, it subclasses it. In conclusion it must subclass both the Target and the Adaptee.

→ Class adapters can be implemented in languages supporting multiple inheritance (Java, C# or PHP does not support multiple inheritance).

Adapter Pattern: **Summary**

→When you need to use an existing class and its interface is not the one you need, use an adapter.

→An adapter changes an interface into one a client expects.

→Implementing an adapter may require little work or a great deal of work depending on the size and complexity of the target interface.

→An adapter wraps an object to change its interface.

→ There are two types of Adapters, Objects Adapters and Class Adapter. The only difference between them is that with class adapter we subclass the *Target* and the *Adaptee*, while the object adapter uses composition to pass (delegate) requests to an *adaptee*.

Strategy Pattern

→ Different algorithms exists for a specific task

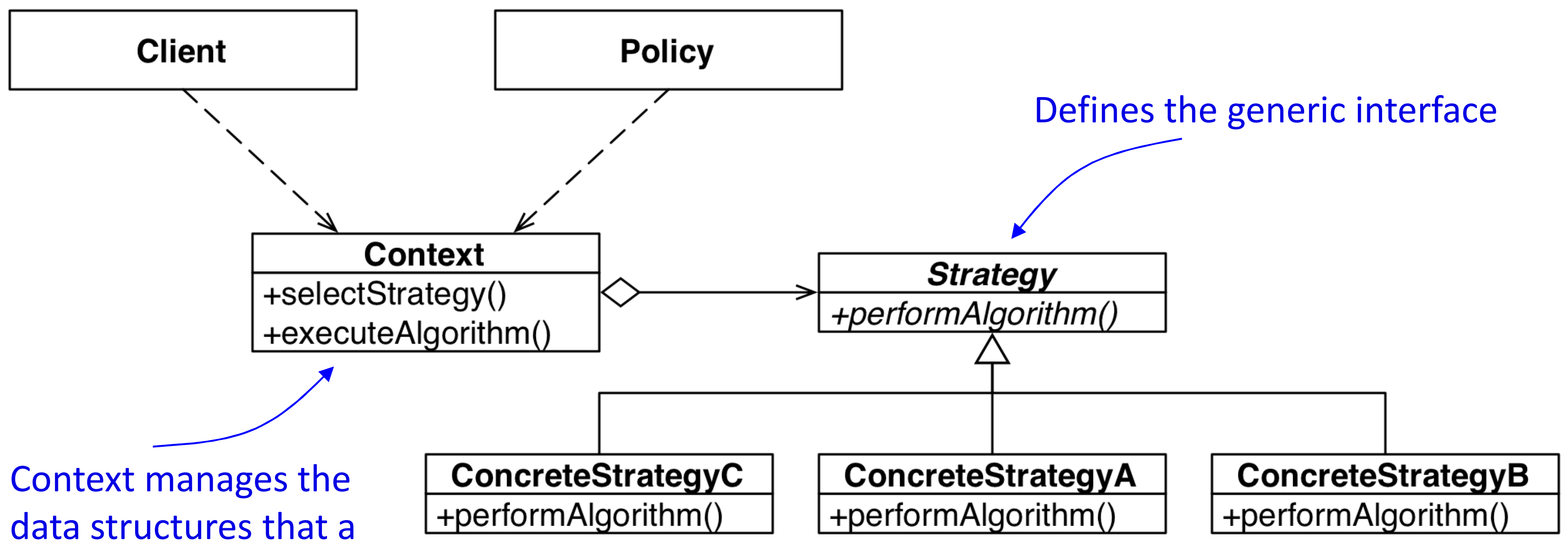
→ Examples of tasks:
- Parsing a set of tokens into an abstract syntax tree (Bottom up, top down)
- Sorting a list of customers (Bubble sort, merge sort, quick sort)

→ The different algorithms will be appropriate at different times
- Use a slow algorithm for rapid prototyping and an optimal algorithm when delivering the final product

→ If we` need a new algorithm, we want to add it easily without disturbing the application where it is using other algorithms.

Strategy Pattern

The **Strategy Design Pattern** defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithms vary independently from the clients that use it.

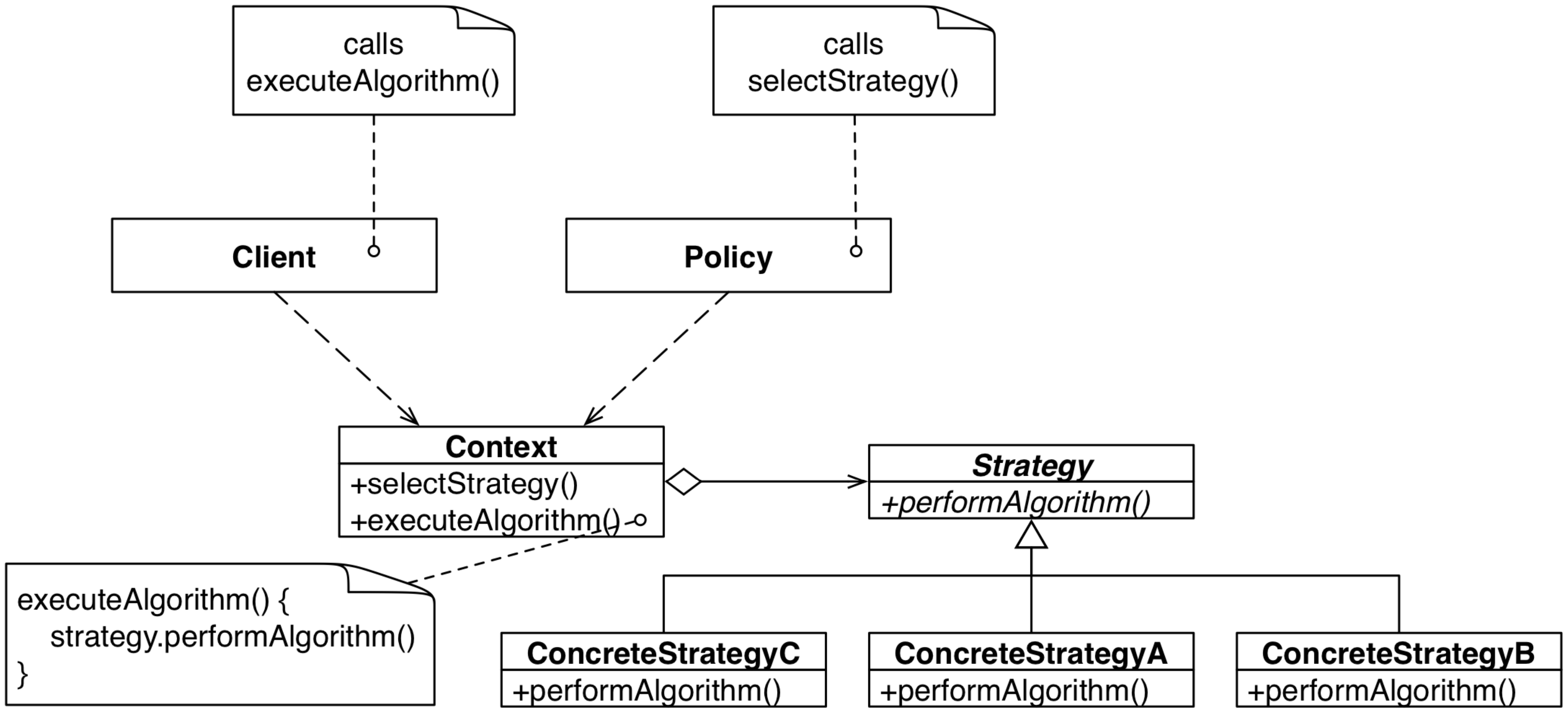


Defines the generic interface

Context manages the data structures that a concrete strategy operates on.

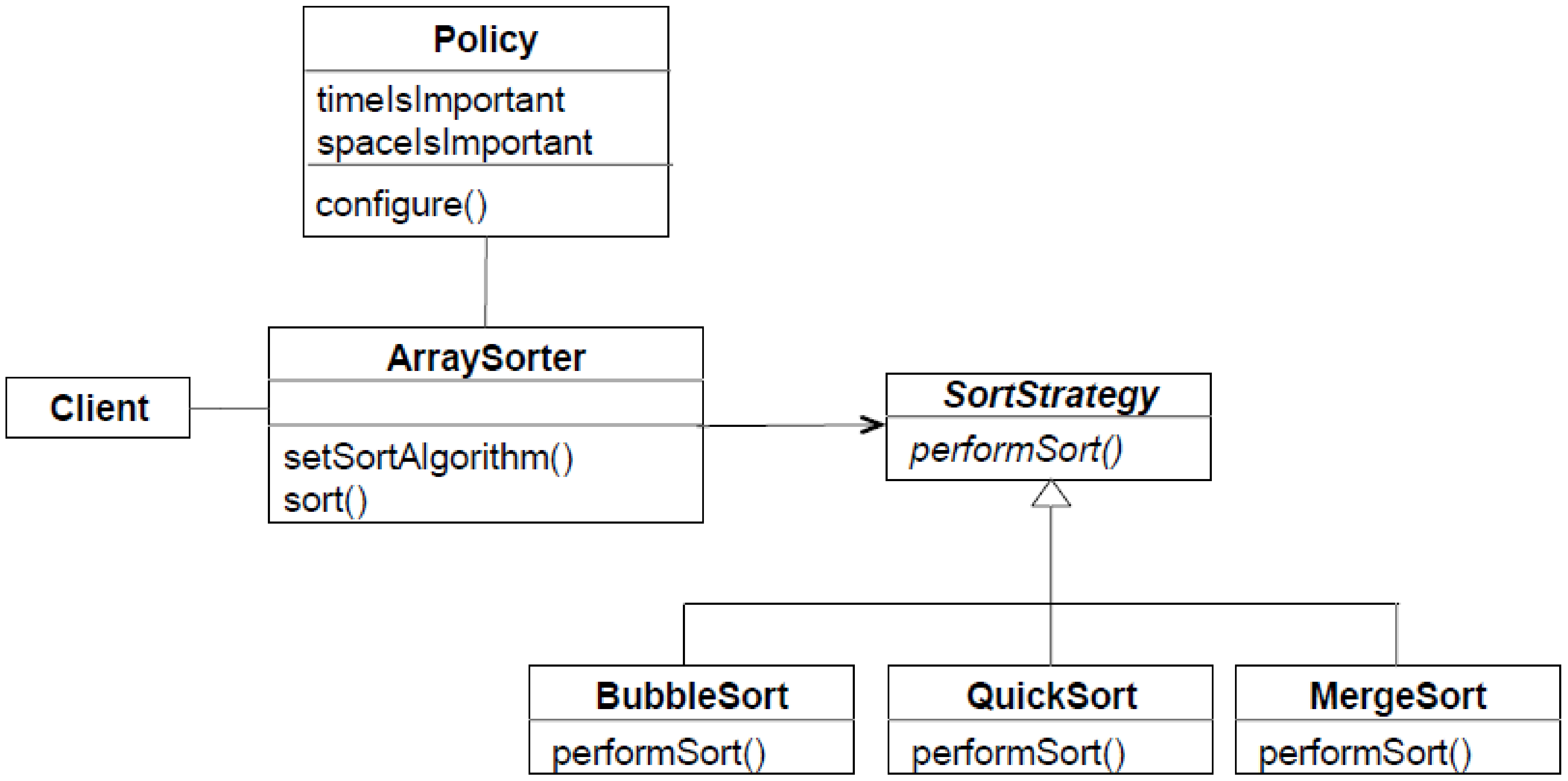
ConcreteStrategy classes provide the implementations of the different strategies. These operate on the data structures in the Context, and can be set dynamically.

Strategy Pattern



The **Policy** class selects the strategy in the **Context** class. This determines which **ConcreteStrategy** is executed, when the Client calls ***executeAlgorithm()***.

Strategy Pattern: **Example**



Observer Pattern

Observer Pattern Motivation

→ **Problem**:

- We have an object that changes its state quite often
  - -- Example: A Portfolio of stocks
- -We want to provide multiple views of the current state of the portfolio
  - -- Example: Histogram view, pie chart view, time line view

→ **Requirements**:

- -The system should maintain consistency across the (redundant) views, whenever the state of the observed object – in our example, the portfolio, changes
- - The system design should be highly extensible
  - -- It should be possible to add new views without having to recompile the observed object or the existing views

Keeping your Objects in the Know!

Observer Pattern

Publishers + Subscribers = Observer Pattern

→ The Publisher (called Subject in the GoF book) represents the entity object

→ Subscribers (called Observer in the GoF book) attach to the Publisher by calling subscribe()

→ Subject can have many observers)

→ Observers (dependent objects) are dependent on the subject to update them when data changes.

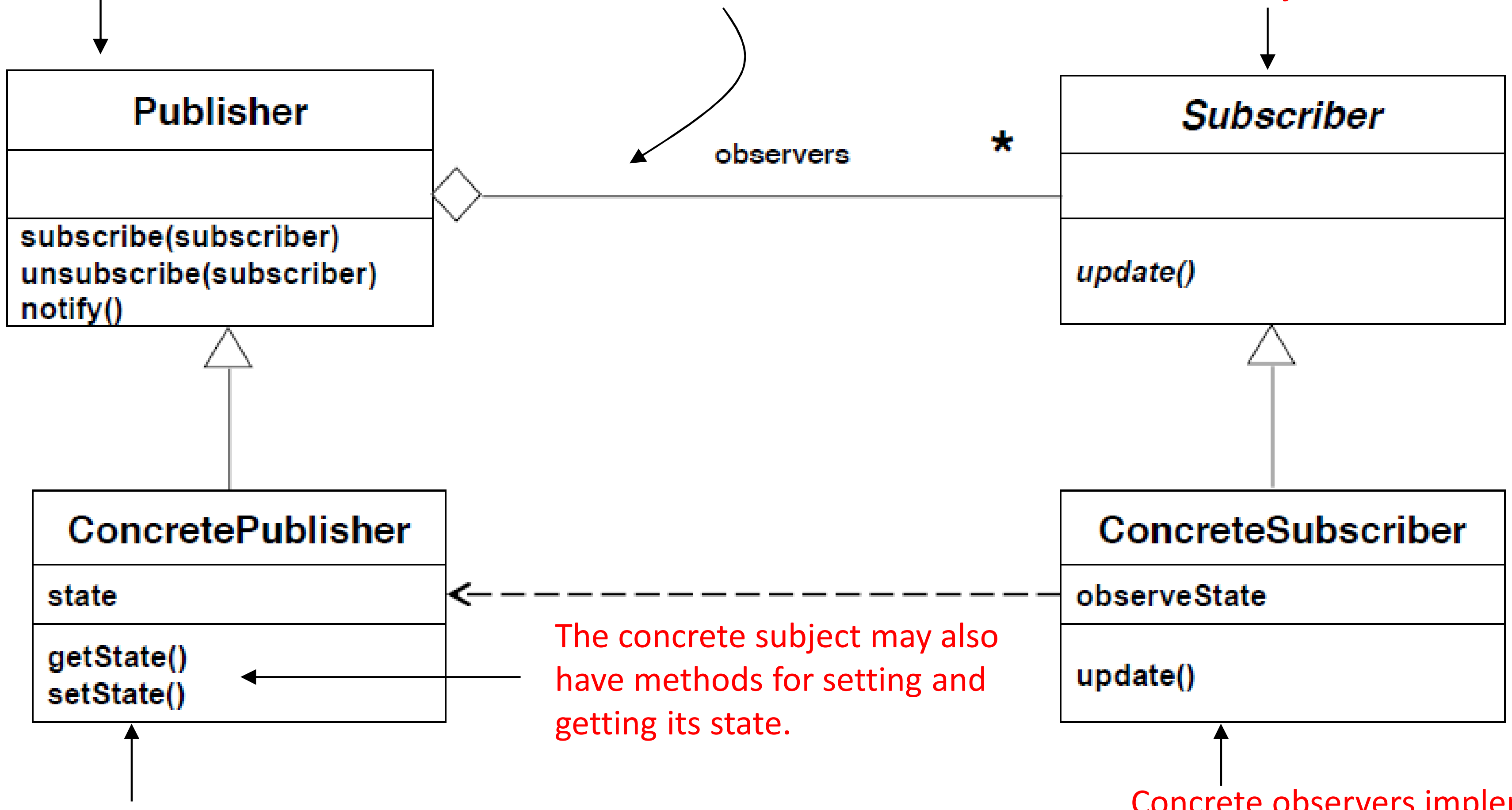
→ Each Subscriber has a different view of the state of the Publisher
- The state is contained in the subclass **ConcretePublisher**
-The state can be obtained and set by subclasses of type **ConcreteSubscriber.**

Observer Pattern

Here's the Subject interface. Objects use this interface to register as observers and also to remove themselves from being observers.

Each subject can have many observers

All observers need to implement the Observer interface. This interface has a method, update ( ), that gets called when the Subject's state changes.



The concrete subject may also have methods for setting and getting its state.

A concrete subject implements the Subject interface. In addition to the register (attach) and remove (detach) methods, the concrete subject implements a notify() method to notify observers whenever state changes.

Concrete observers implements the Observer interface. Each observer registers with a concrete subject to receive updates.

Observer Pattern

>> The **Observer Pattern** defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

>> Connects the state of an observed object (the publisher) with many observing objects (the subscribers)

>> Usage:

- Maintaining consistency across redundant states
- Optimizing a batch of changes to maintain consistency

>> Three variants for maintaining the consistency:

- **Push Notification:** Every time the state of the publisher changes, *all the* subscribers are notified of the change
- **Push-Update Notification:** The publisher also sends the state that has been changed
- **Pull Notification:** A subscriber inquires about the state of the subject

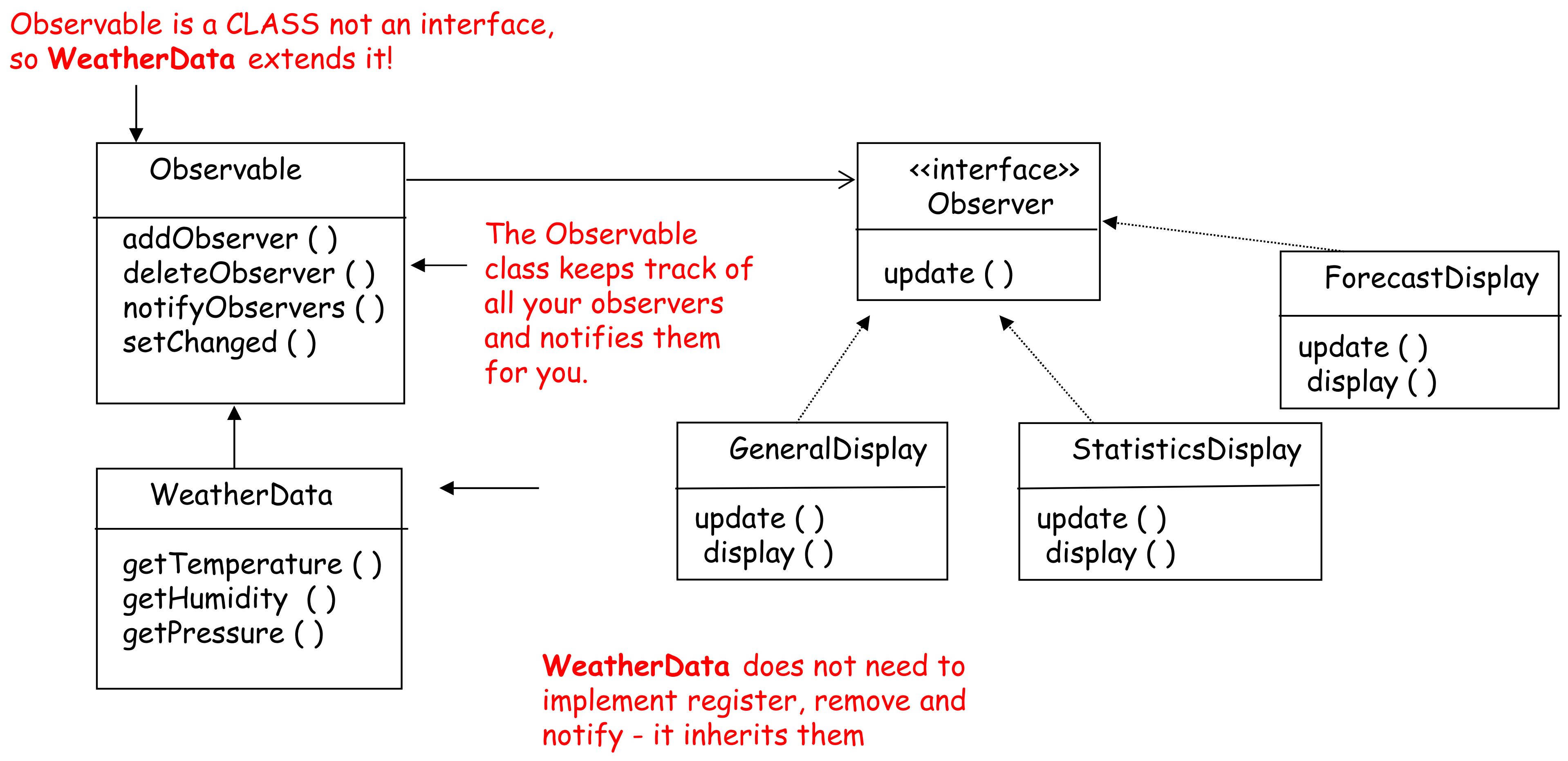
>> Also called Publish and Subscribe.

Observer Pattern: **Example**

Using Java's Built-in Observer Pattern

Observer == Observer/Subscriber

Observable == Subject/Publisher



Factory Pattern

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 the window managers used by different operating systems:
  - -- Example: Chrome on Windows, Chrome on MacOS X

→ Consider a facility management system for an intelligent house that supports different control systems

- Existing Control Systems:
  - -- Johnson Control
  - -- Zumtobel
  - -- EIB (European Installation Bus)
- Can you write a facility management application that encapsulates the manufacturer-specific control system from the rest of the system?

Factory Pattern

>> Good object-oriented design means high coherence and low coupling

- Using the *new operator creates a dependency on a concrete class,* which means high coupling
- But we learned that it is better to program against super classes (e.g. Java interfaces) than against concrete implementations

>> How can we instantiate new objects without depending on concrete implementations?

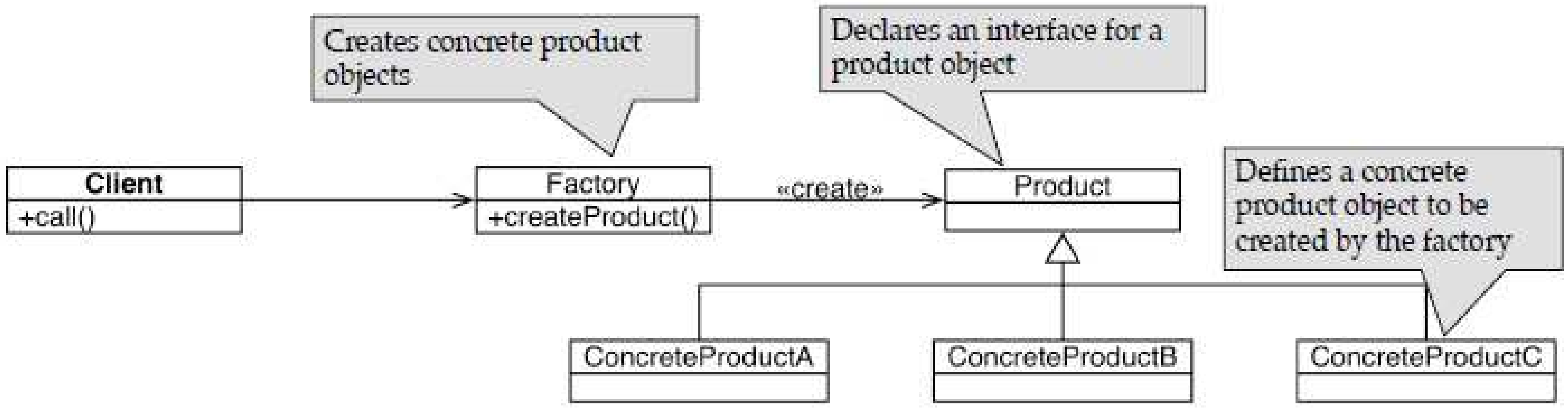
- One possibility is the use of a factory class

>> Factory class: a class responsible for the instantiation of objects.

Factory Pattern

What does a Factory class do?

→ Acts as delegate for the creation of products
→ Allows the Client to use a single interface to the products of a factory
→ Makes it is easy to add additional products (extensibility)
→ Polymorphism allows the client to use each of the concrete products in a uniform way.


Creates concrete product objects
Declares an interface for a product object
Client
+call()
Factory
+createProduct()
«create»
Product
Defines a concrete product object to be created by the factory
ConcreteProductA
ConcreteProductB
ConcreteProductC

Factory Pattern

From the Factory class to Abstract Factory Pattern

→ The factory pattern also puts calls to *new into the factory*, thereby reducing the dependency of the client code on a concrete implementation

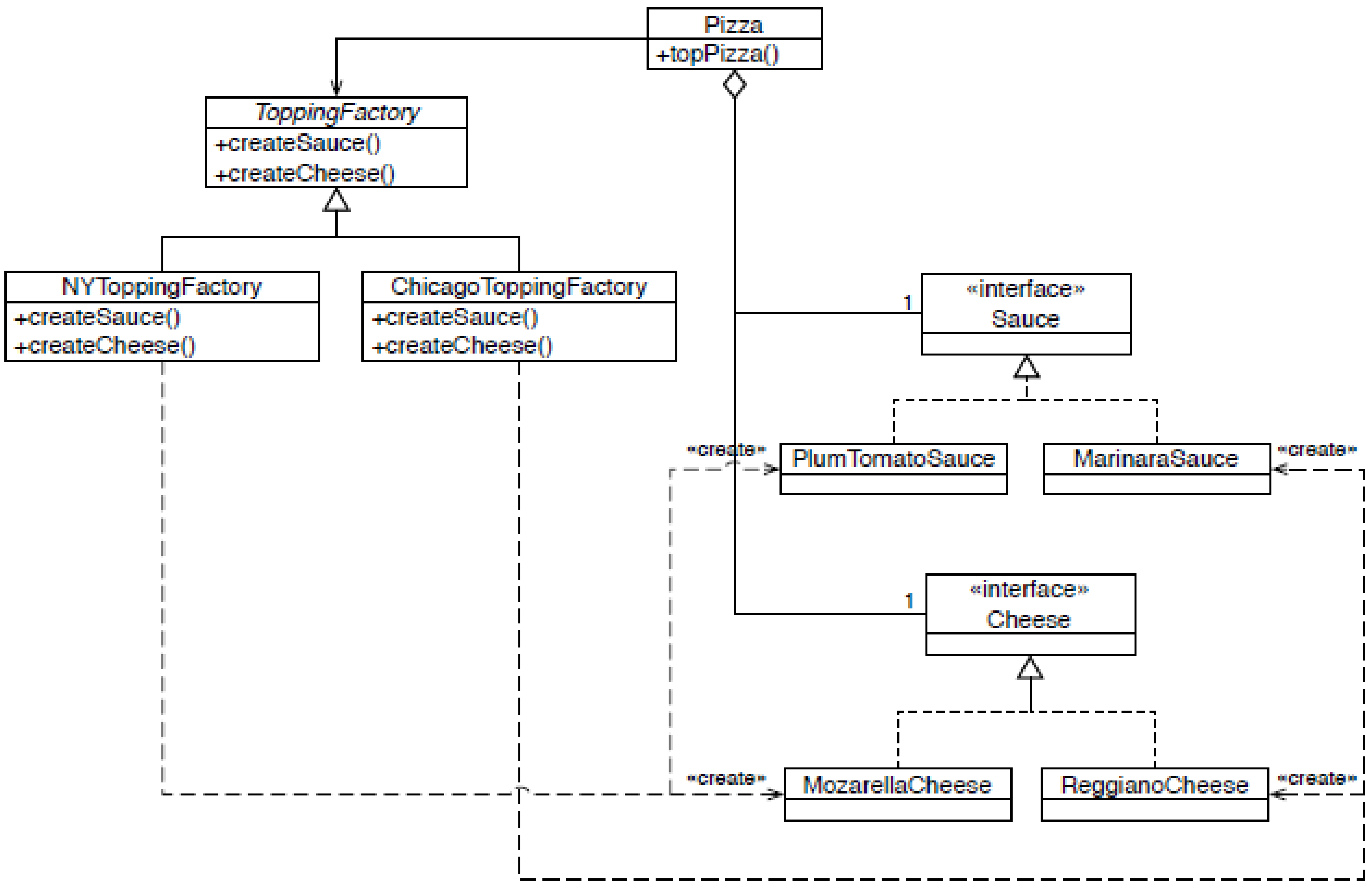
→ But we are bound to a single Factory and still have a lot of “if” statements

→ How can we generalize this?
- For example, New Yorkers like Mozarella, while people fromChicago prefer Reggiano cheese
- New Yorkers prefer Plumsauce, while Chicagoers can only stomach Marina sauce
- Bavarians prefer Weisswurst and sweet mustard

→ How can we write an application for **pizza toppings** which can be customized for people living in different cities
- Can we create factories for Chicago style pizzas and NY style pizzas?
- Can we offer Bavarian style pizzas?

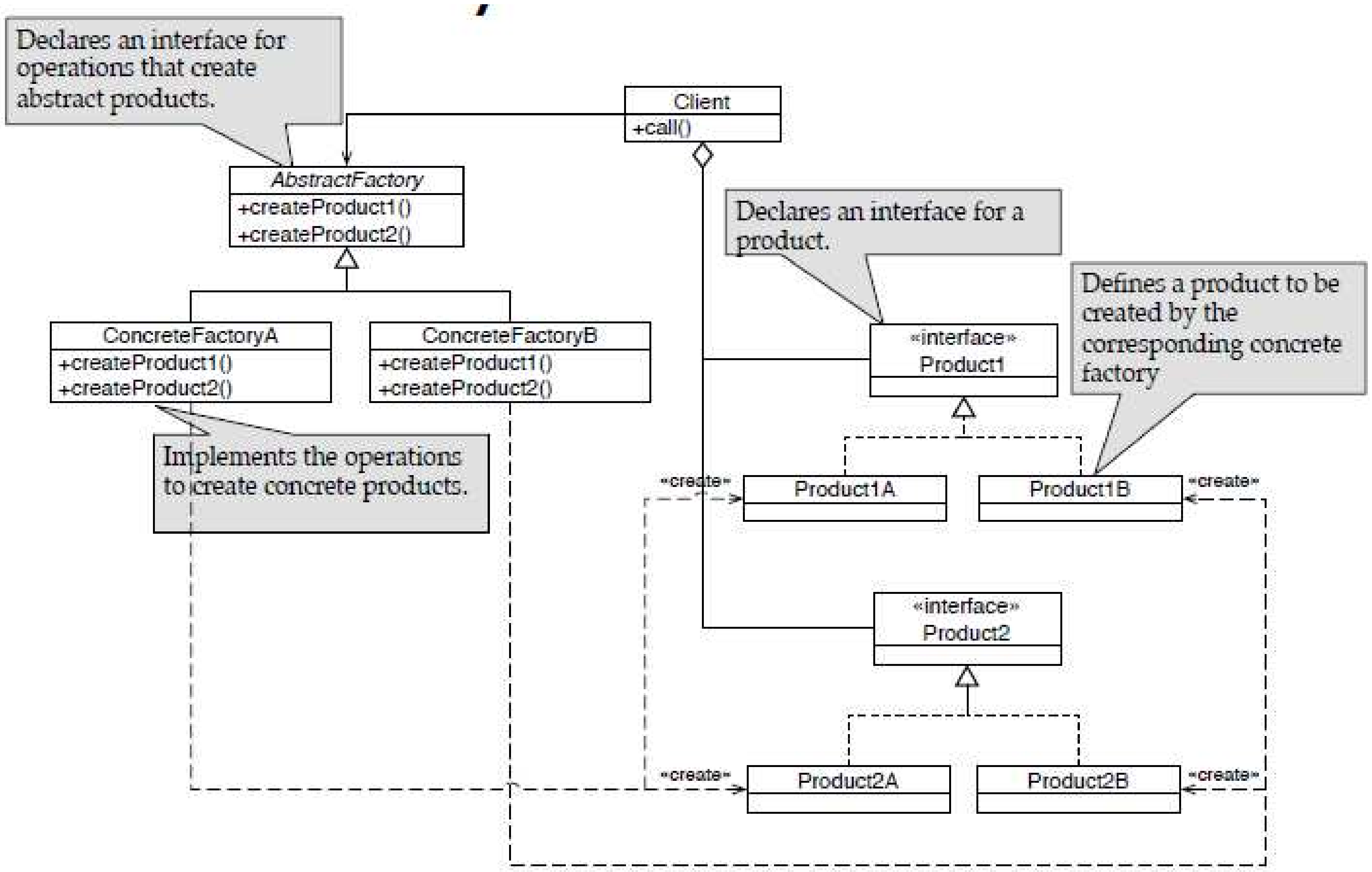
Factory Pattern: **Example**

A Pizza as a Family of Related Objects of Toppings and Sauces



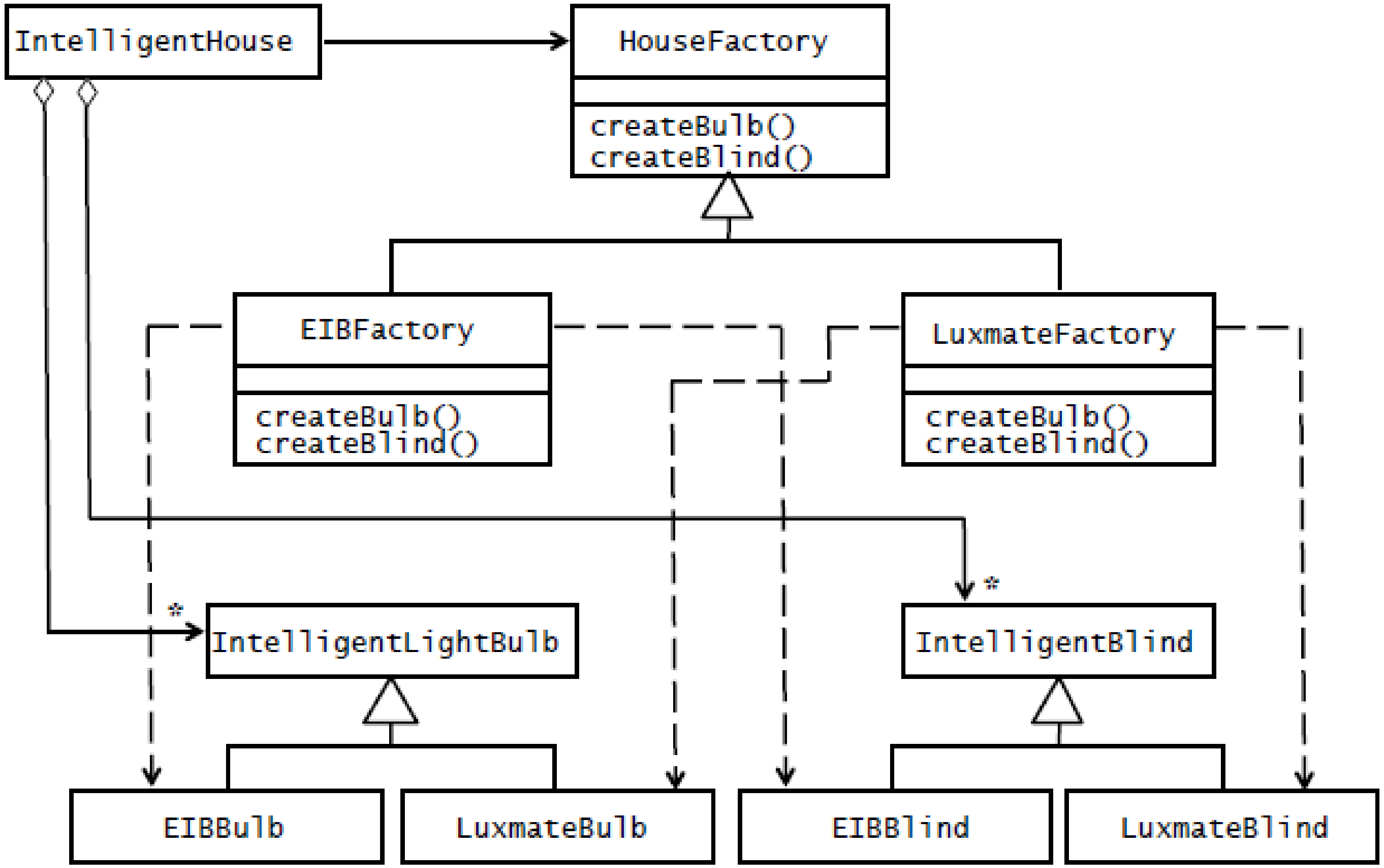
Factory Pattern

The **Abstract Factory Pattern** provides an interface for creating families of related or dependent objects without specifying their concrete classes.



Factory Pattern: **Example**

A Facility Management System for a House can also be modeled with the Abstract Factory Pattern



Singleton Pattern

→ How to instantiate just one object - one and only one!

→ Why we need single object?

- Many objects we need only one of: DB Connection, thread pools, caches, objects that handle preferences, and registry settings etc.
- If more than one instantiated:

  Incorrect program behavior, overuse of resources, inconsistent results

→Alternatives:

-Use a *global variable*

  Downside: assign an object to a global variable then that object might be created when application begins. If application never ends up using it and object is resource intensive --> waste!

- Use a *static variable*

  Downside: how do you prevent creation of more than one class object?

Singleton Pattern

→ Is there any class that could use a private constructor?

→ What's the meaning of the following?

```
public MyClass {
    public static MyClass getInstance ( ) { }
}
```

→ Instantiating a class with a private constructor:

```
public MyClass {
    private MyClass ( ) { }
    public static MyClass getInstance ( ) { }
}
```

Singleton Pattern

```
public class Singleton {
      private static Singleton uniqueInstance;
      // other useful instance variables

      private Singleton  ( ) {  }
      public static Singleton getInstance  ( ) {
            if (uniqueInstance == null) {
                  uniqueInstance = new Singleton ( );
            }
            return uniqueInstance;
      }

      // other useful methods
}
```

Constructor is declared private; only singleton can instantiate this class!

We have a static variable to hold our one instance of the class Singleton.

The getInstance ( ) method gives us a way to instantiate the class and also return an instance of it.

Of course, Singleton is a regular class so it has other useful instances and methods.

Singleton Pattern

uniqueInstance holds our ONE instance; remember it is a static variable

If uniqueInstance is null, then we haven't created the instance yet…

```
if (uniqueInstance == null) {
            uniqueInstance = new Singleton ( );
      }
      return uniqueInstance;
```

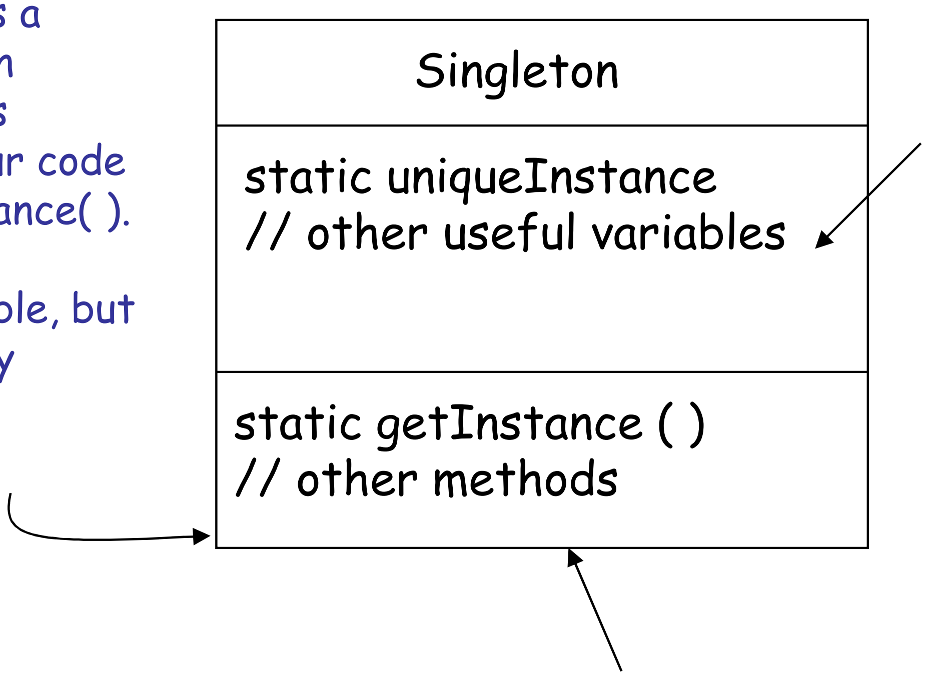
If uniqueInstance wasn't null, then it was previously created. We just fall through to the return statement. In either case, we have an instance and we return it.

…and if it doesn't exist, we instantiate Singleton through its private constructor and assign it to the uniqueInstance. Note that if we never need the uniqueInstance, it never gets created --> **lazy instantiation.**

Singleton Pattern

The **Singleton Pattern** ensures a class has only one instance, and provides a global point of access to it.

The getInstance ( ) method is static, which means it is a class method, so you can conveniently access this method anywhere in your code using Singleton.getInstance( ). That's just as easy as accessing a global variable, but we get benefits like lazy instantiation from the Singleton.



The uniqueInstance class variable holds our one and only one instance of Singleton.

A class implementing a Singleton Pattern is more than a Singleton; it is a general purpose class with its own set of data and methods.

Singleton Pattern: **Summary**

>> The Singleton Pattern ensures you have at most one instance of a class in your application

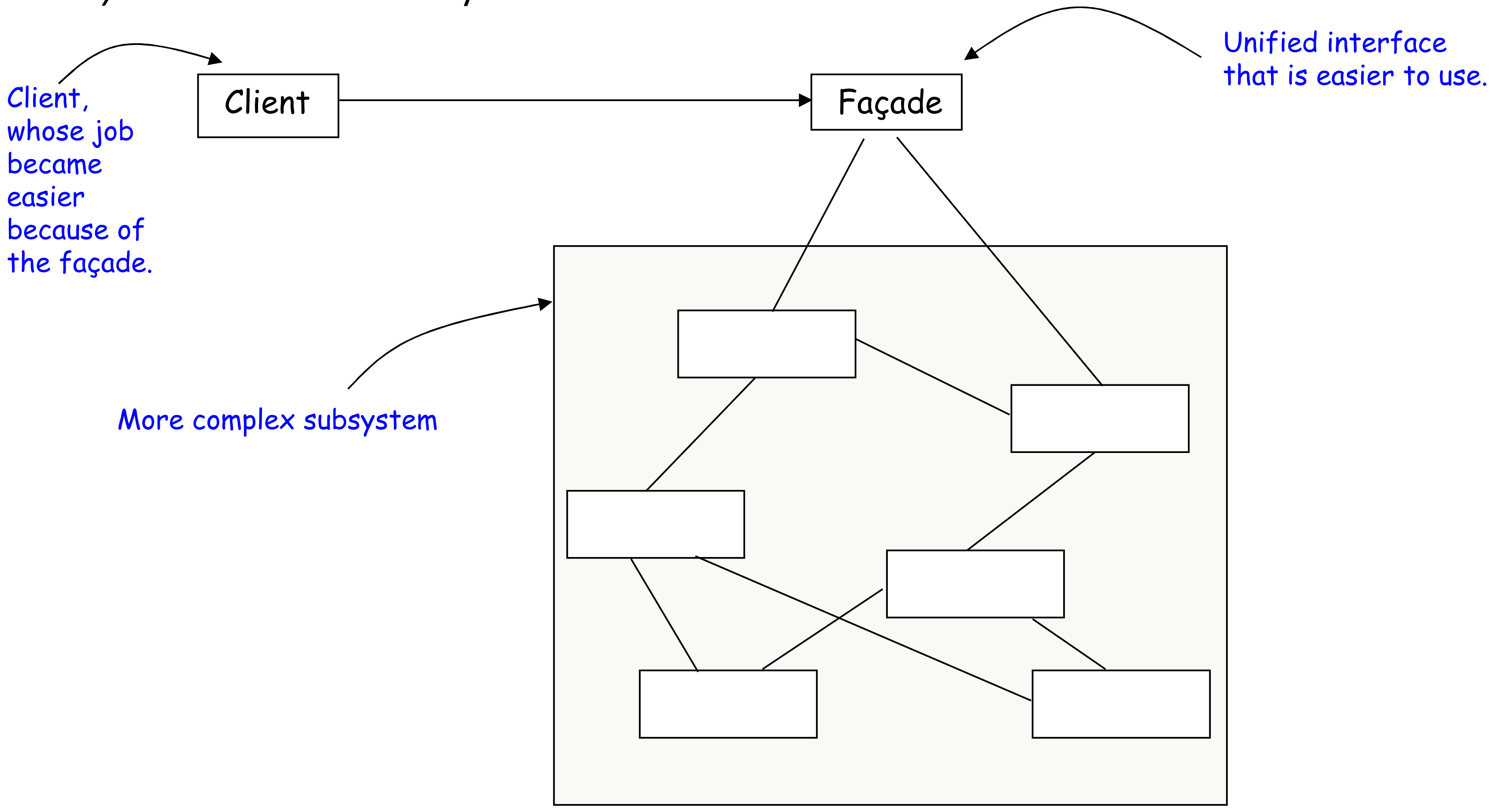
>> The Singleton Pattern also provides a global access point to that instance.

>> Java’s implementation of the Singleton Pattern makes use of a private constructor, a static method combined with a static variable

>> Examine your performance and resource constraints and carefully choose an appropriate Singleton implementation for multi-threaded applications.

Façade Pattern

The **Façade Pattern** provides a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface *(wrap a Complicated interface with a simpler one)* that makes the subsystem easier to use.


Client, whose job became easier because of the façade.
Client
Façade
Unified interface that is easier to use.
More complex subsystem

Façade Pattern

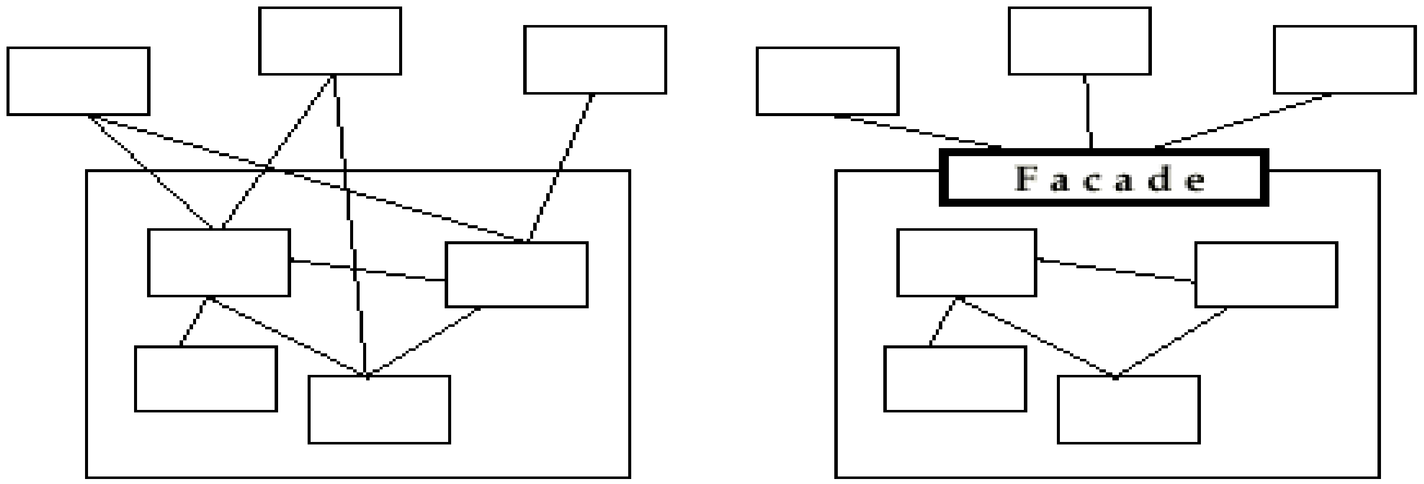
→ Principle of Least Knowledge: talk only to your immediate friends!

→ What does it mean?

- When designing a system, for any object, be careful of the number of classes it interacts with and also how it comes to interact with those classes.

→This principle prevents us from creating designs that have a large number of classes coupled together so that changes in one part of the system cascade to the other parts.

- When you build a lot of dependencies between many classes, you are building a fragile system that will be costly to maintain and complex for others to understand!



Façade Pattern: **Compiler Example**

→ Consider a programming environment that gives applications access to its compiler subsystem.

→ The subsystem contains classes that implement the compiler (such as Scanner, Parser, Program Node, BytecodeStream and ProgramNodeBuilder)

→Some applications may need to access these classes directly, but most applications just want the compiler to compile some code and don't want to have to understand how all the classes work together. The low-level interfaces are powerful, but unnecessarily complex for these applications.

→ In this situation, a Façade can provide a simple interface to the complex subsystem, eg. a class *Compiler*, with the method *compile()*

→ The Façade (Compiler) knows which subsystem classes are responsible for a request and delegates the request to the appropriate subsystem objects

→ The subsystem classes (Scanner, Parser, etc.) implement the subsystem functionality, handle work assigned by the Façade object and have no knowledge of the Façade object (ie, keep no reference to it)

Façade Pattern: **Home Theater System Example**

→ Watching a Movie the Hard Way!

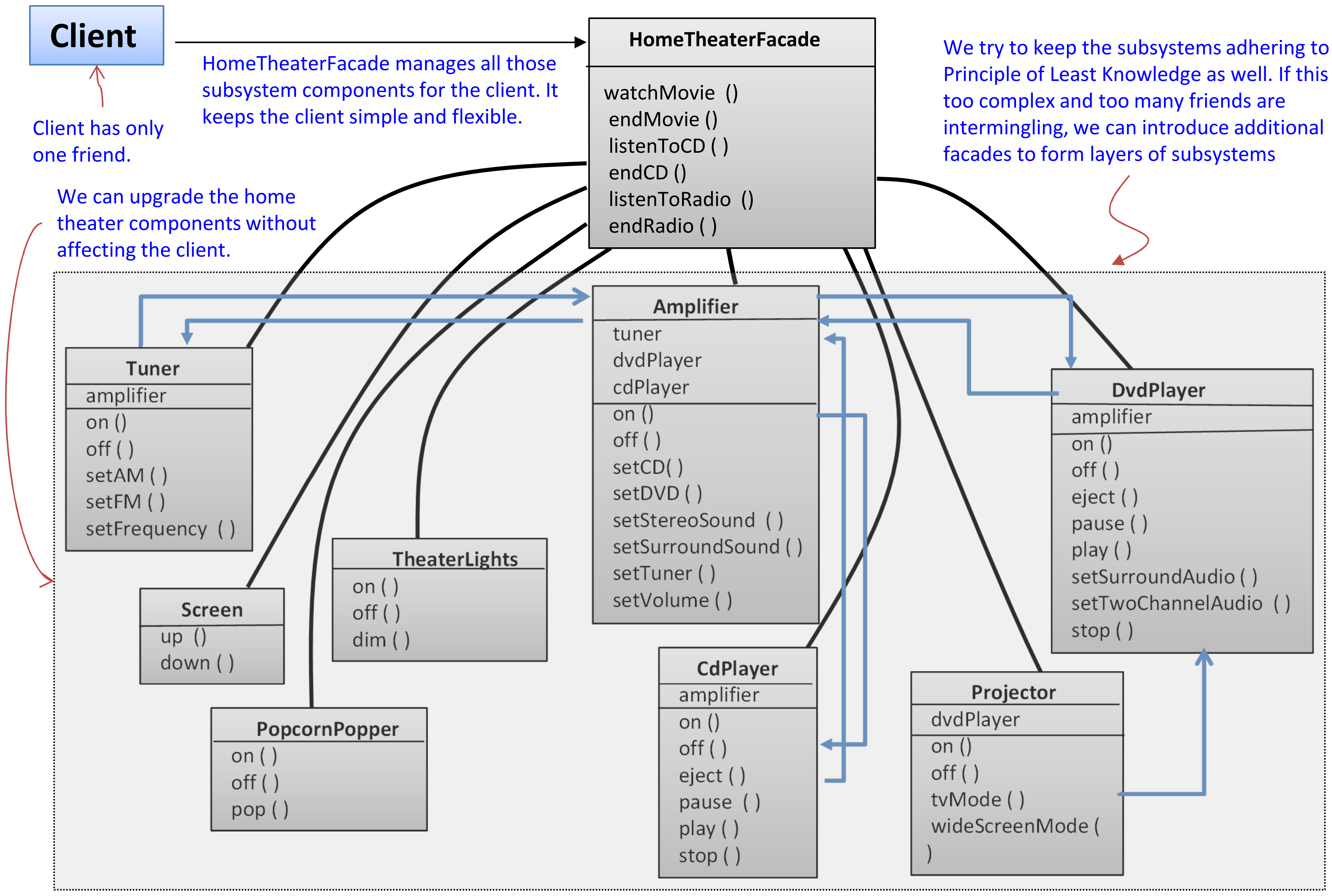
1. Turn on the popcorn popper
2. Start the popper popping
3. Dim the lights
4. Put the screen down
5. Turn the projector on
6. Set the projector input to DVD
7. Put the projector on wide-screen mode
8. Turn the sound amplifier on
9. Set the amplifier to DVD input
10. Set the amplifier to surround sound
11. Set the amplifier volume to medium (5)
12. Turn the DVD player on
13. Start the DVD player playing.
14. Whew!

But there's more!
When the movie is done, how do you turn everything off? Do you reverse all the steps?
Wouldn't it be just as complex to listen to a CD or radio?
If you decide to upgrade your system, you're probably going to have to learn a slightly different procedure!

Façade to the Rescue!!

(1) Create a Façade for the *HomeTheater* which exposes a few simple methods such as *watchMovie ( )*
(2) The Façade treats the home theater components as its subsystem, and calls on the subsystem to implement its *watchMovie ( )* method.
(3) The Client now calls methods on the façade and not on the subsystem.
(4) The Façade still leaves the subsystem accessible to be used directly.

Façade Pattern: **Home Theater System Example**



Client has only one friend.

HomeTheaterFacade manages all those subsystem components for the client. It keeps the client simple and flexible.

We try to keep the subsystems adhering to the Principle of Least Knowledge as well. If this gets too complex and too many friends are intermingling, we can introduce additional facades to form layers of subsystems

We can upgrade the home theater components without affecting the client.

Façade Pattern: **Summary**

→When you need to simplify and unify a large interface or a complex set of interfaces, use a façade.

→ A façade decouples the client from a complex subsystem.

→ Implementing a façade requires that we compose the façade with its subsystem and use delegation to perform the work of the façade.

→ You can implement more than one façade for a subsystem.

→ A façade "wraps" a set of objects to simplify!

References

Professor Bernd Bruegge

Most of the slides have been taken from the lecture on *"Patterns in Software Engineering"* delivered by **Professor Bernd Bruegge**