

CMPS310
Software Engineering
Fall 2021

Lecture 8

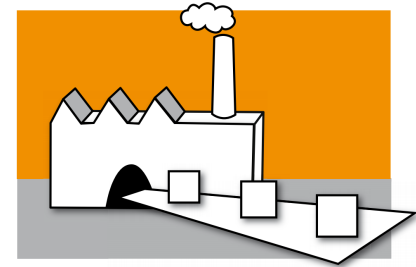
Software Design Pattern

How Design Patterns Achieve Quality

- Pattern is a general reusable solution to commonly known problems
- It is a template for how a problem can be solved
- The solution is proved to be effective to the identified problem
- It can help find correct objects and can create objects without exposing the code
- Pattern can be used to specify object interfaces
- Pattern is used to make object implementation easy with known solutions
- Reuse of pre-existing solution

Factory Design Pattern

Factory Pattern



- Factory: “Create me an object of this type”
- How do we create buttons? Who creates buttons?
 - We use a Factory pattern
 - Handles the creation of complex objects
- In Factory pattern, we
 - create object *without exposing the creation logic to the client*, and
 - refer to newly created object *using a common interface*.
- The intents in employing the Factory pattern are:
 - To insulate the creation of objects from their usage, and
 - To create families of related objects without having to depend on their concrete classes.

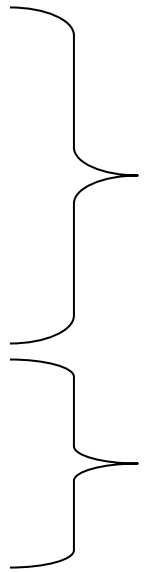
Factory Motivation

- Consider a pizza store that makes different types of pizzas

```
Pizza pizza;
```

```
if (type == CHEESE)
    pizza = new CheesePizza();
else if (type == PEPPERONI)
    pizza = new PepperoniPizza();
else if (type == PESTO)
    pizza = new PestoPizza();
```

```
pizza.prepare();
pizza.bake();
pizza.package();
pizza.deliver();
```



This becomes unwieldy
as we add newer pizza to
our menu

This part stays the same

Idea: pull out the creation code and put it into an object that only deals with creating pizzas - the *PizzaFactory*

Factory Motivation

```
public class PizzaFactory
{
    public Pizza createPizza(int type)
    {
        Pizza pizza = null;
        if (type == CHEESE)
            pizza = new CheesePizza();
        else if (type == PEPPERONI)
            pizza = new PepperoniPizza();
        else if (type == PESTO)
            pizza = new PestoPizza();
        return pizza;
    }
}
```

Replace concrete instantiation
with call to the PizzaFactory to
create a new pizza

Now we don't need to mess with
this code if we add new pizzas

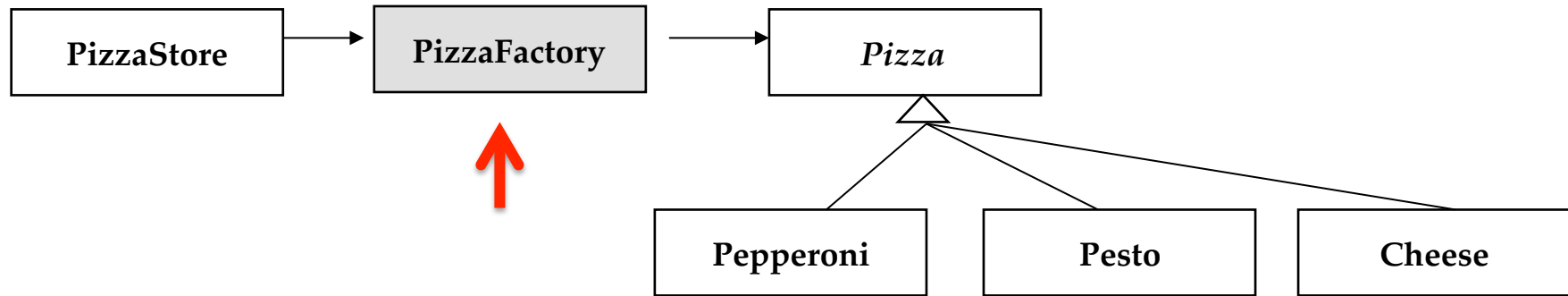
```
Pizza pizza;
PizzaFactory factory;
```

...

→ `pizza = factory.createPizza(type);`

```
pizza.prepare();
pizza.bake();
pizza.package();
pizza.deliver();
```

Pizza Classes



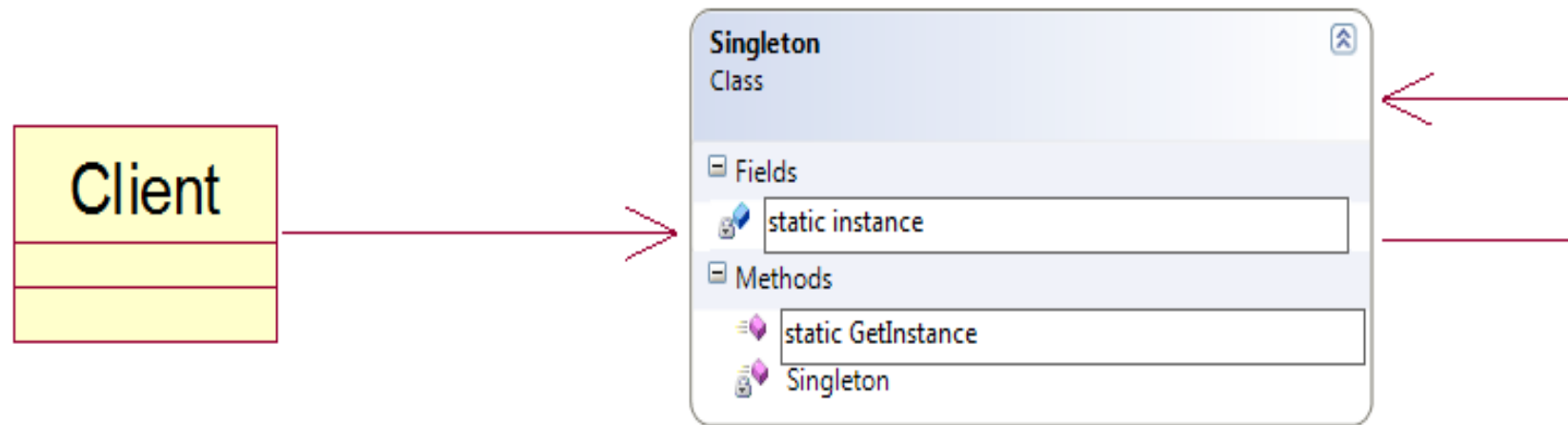
Singleton Pattern

Introduction :Singleton Pattern

- Intent
 - Ensure that a certain class has only one instance
 - Provide global access to it
 - Restricts the instantiation of a class to only one object
- Motivation
 - It's important for some classes to have exactly one instance.
 - This is particularly useful when one object is needed to coordinate actions across a software system.
 - There should be only one file system (or file system manager) and one window manager.

The Singleton pattern ensures that the class (not the programmer) is responsible for the number of instances created.

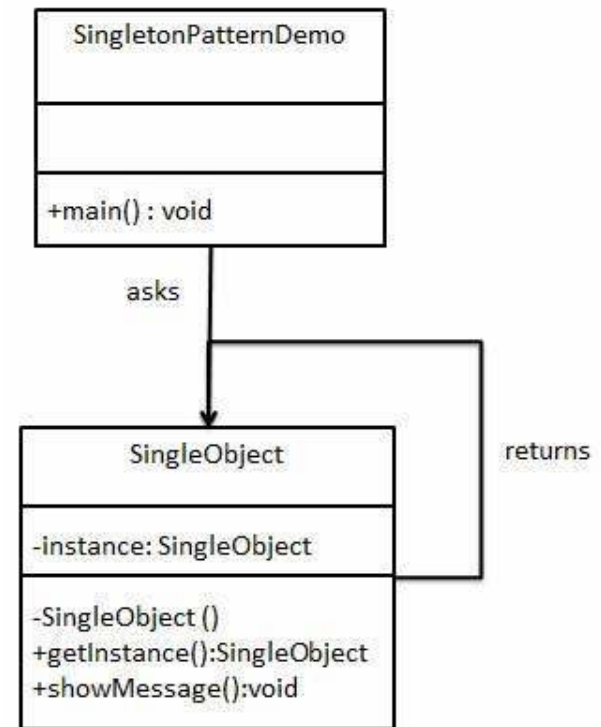
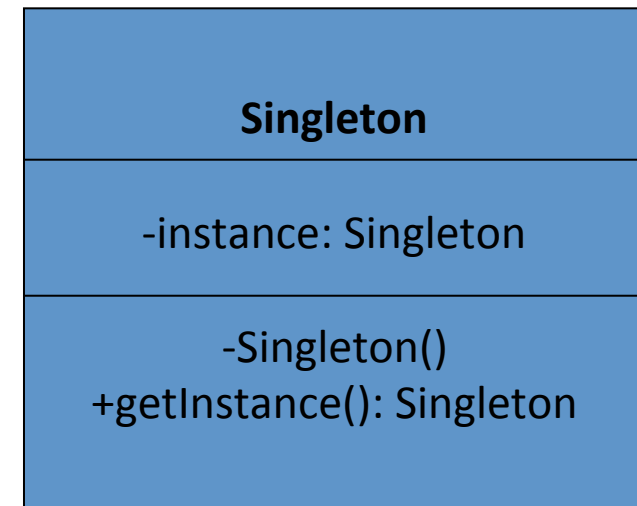
The Singleton Pattern



- The class of the single instance is responsible for access and “initialization on first use”.
- The single instance is a **protected static attribute** in order to guarantee that **a new instance is created if one doesn't already exist, and if one does exist, a reference to that instance is accessible.**
- The constructor is private in order to ensure that the object can only be instantiated via the constructor
- Other objects can use Singleton in their implementations.
- Some objects are singleton such as Facade (we will see Facade pattern later.)

Example

```
public class Singleton {  
    private static Singleton instance = null;  
    private Singleton() {  
    }  
    public static Singleton getInstance() {  
        /* accessor function */  
        if(instance == null) {  
            instance = new Singleton();  
        }  
        return instance;  
    }  
}  
  
public class TestSingleton{  
    public static void main(String[] args){  
        Singleton s = Singleton.getInstance();  
        ...  
    }  
}
```



Important Concepts

1. Why is the constructor private?

Any client that tries to instantiate the **Singleton** class directly will get an error at compile time.

- This ensures that no object other than the **Singleton** class can instantiate the object of this class.

2. How can you access the single instance of the **Singleton** class?

getInstance() is a static method. We can use the class name to reference the method.

- **Singleton.getInstance()**
- This is how we can provide global access to the single instance of the **Singleton** class.

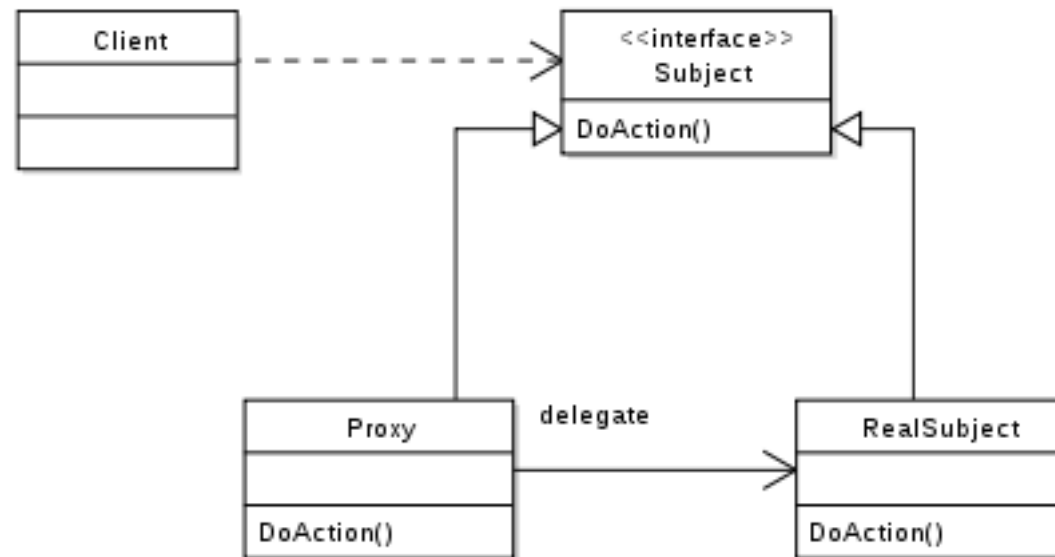
How and When to Use Singleton Pattern

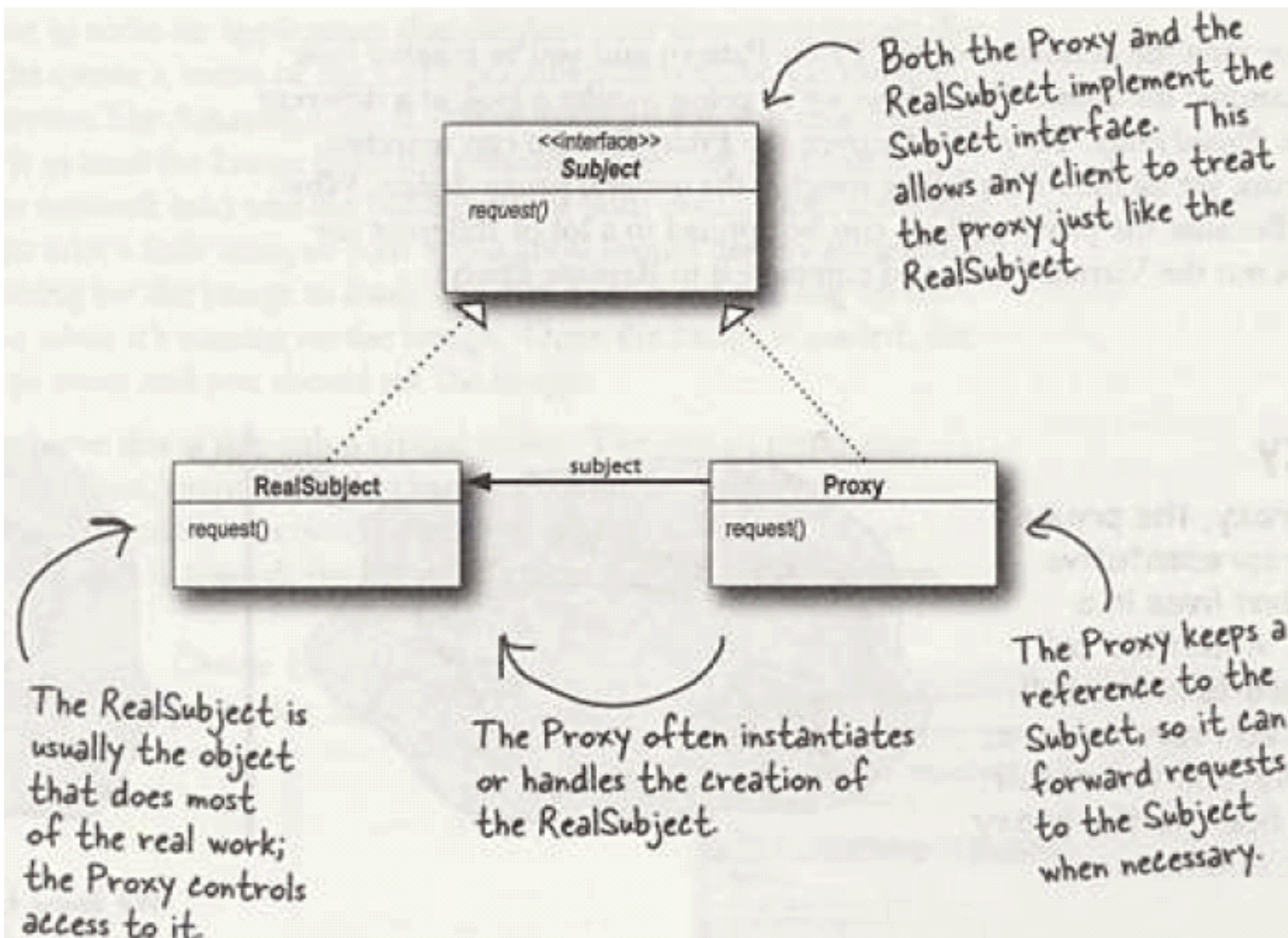
- Make the class of the single instance object responsible for creation, initialization, access, and enforcement.
- Declare the instance as a private static data member.
- Abstract Factory, Builder, and Prototype can use Singleton in their implementation.
- Facade objects are often Singletons because only one Facade object is required.
- Use Singleton pattern when it's simpler to pass an object resource as a reference to the objects that need it, rather than letting objects access the resource globally.
- Singletons are often preferred to global variables because:
 - They do not pollute the global namespace (or, in languages with namespaces, their containing namespace) with unnecessary variables.
 - They permit **lazy allocation** and initialization, whereas global variables in many languages will always consume resources.
- Lazy allocation simply means not allocating a resource until it is actually needed.
- This is common with singleton objects,
- Any time a resource is allocated as late as possible
- By delaying allocation of a resource until actually needed,
 - it can decrease startup time, and even eliminate the allocation entirely if the object is actually never used

Proxy Pattern

Proxy Pattern

- In proxy pattern, a class represents functionality of another class.
- This type of design pattern comes under structural pattern.
- In proxy pattern, we create object having original object to interface its functionality to outer world.
- The proxy could interface to anything: a network connection, a large object in memory, a file, or some other resource that is expensive or impossible to duplicate.



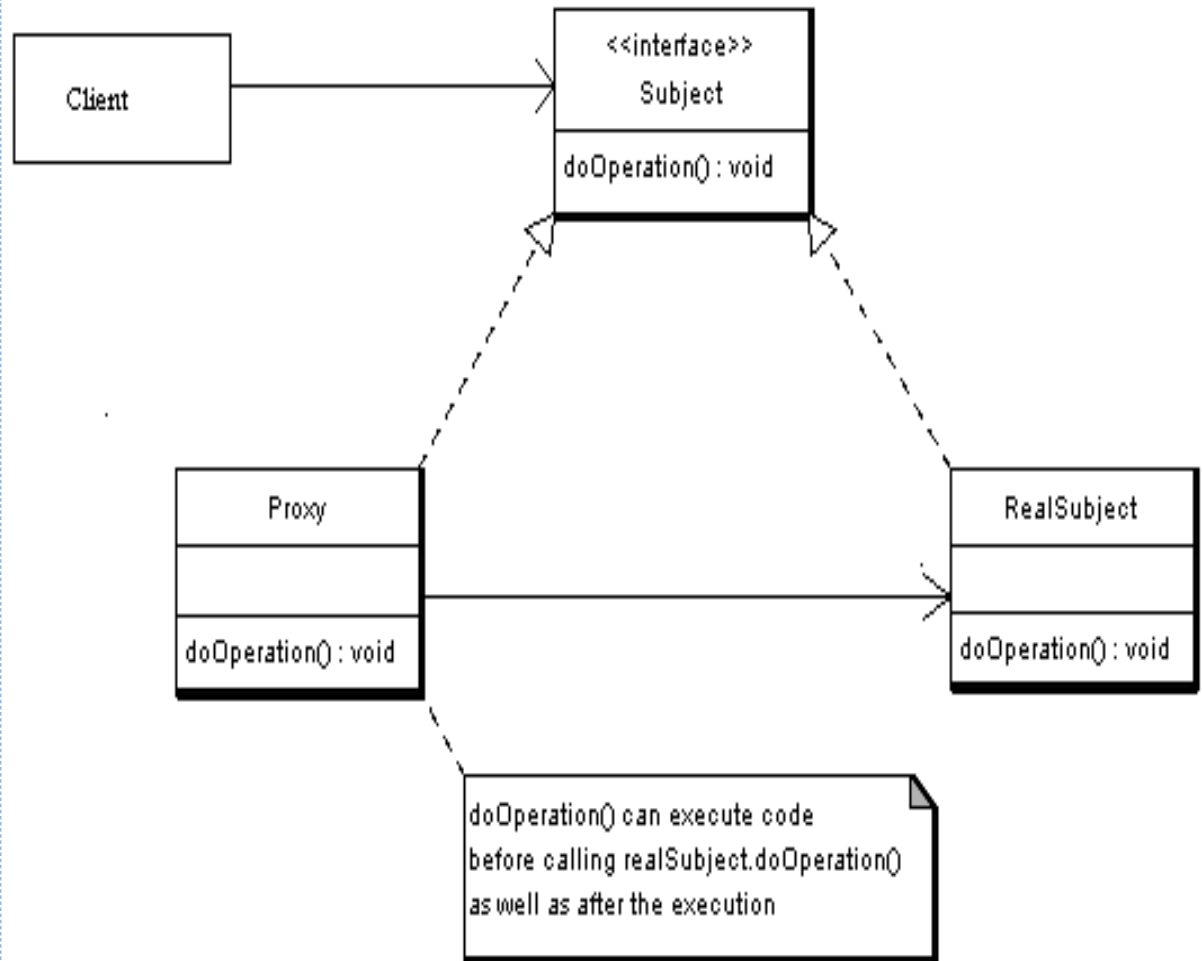


General Scenario

- Client needs to do a task
- Client calls the method of Proxy (send a service message)
- Proxy internally does pre-processing such as message validation
- Proxy sends the real service message to Original
- Proxy does post-processing and return results back to Client

Example: Proxy

- The participants classes in the proxy pattern are:
- Subject - Interface implemented by the RealSubject and representing its services. The interface must be implemented by the proxy as well so that the proxy can be used in any location where the RealSubject can be used.
- Proxy:
 - Maintains a reference that allows the Proxy to access the RealSubject.
 - Implements the same interface implemented by the RealSubject so that the Proxy can be substituted for the RealSubject.
 - Controls access to the RealSubject and may be responsible for its creation and deletion.
 - Other responsibilities depend on the kind of proxy.
- RealSubject - the real object that the proxy represents.



Proxy: Benefits

- A proxy may hide information about the real object to the client.
- A proxy may perform optimization like on demand loading.
- A proxy may do additional house-keeping job like audit tasks.
- Proxy design pattern is also known as surrogate design pattern.

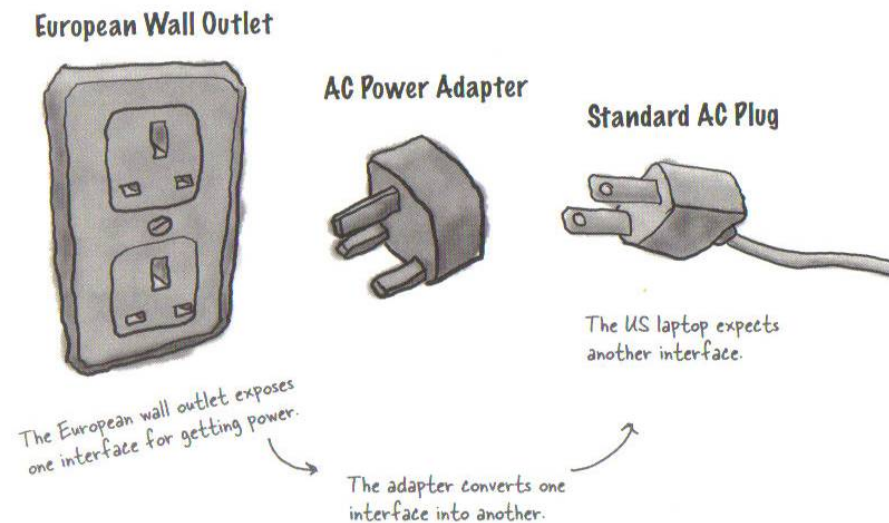
There are some common situations in which the Proxy pattern is applicable.

- A virtual proxy is a placeholder for "expensive to create" objects. The real object is only created when a client first requests/accesses the object.
- A remote proxy provides a local representative for an object that resides in a different address space. This is what the "stub" code in RPC and CORBA provides.
- A protective proxy controls access to a sensitive master object.
- The "surrogate" object checks that the caller has the access permissions required prior to forwarding the request.

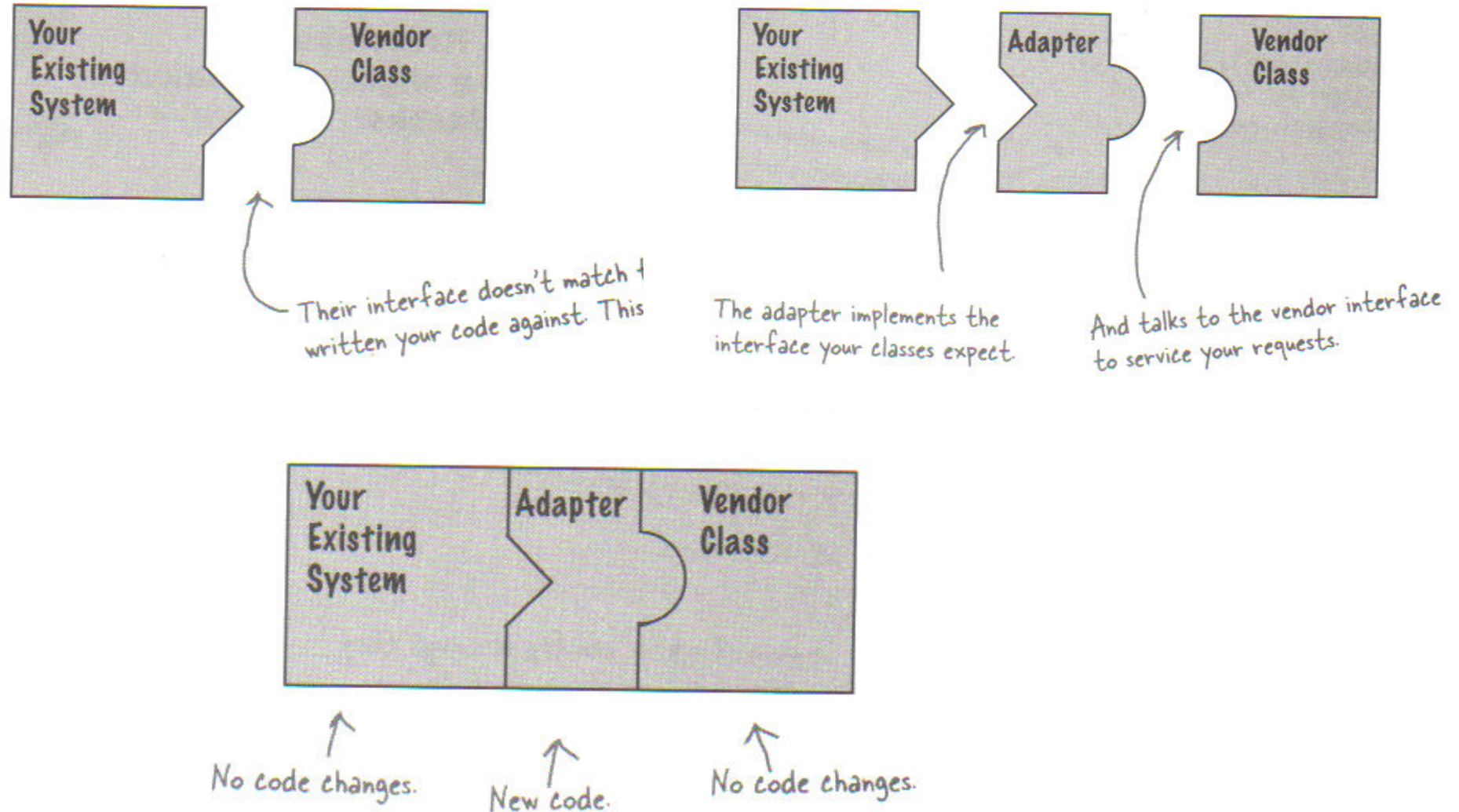
Adapter Pattern

Adapters in real life

- Adapter allows the interface of an existing class to be used from another interface.
- It is often used to make existing classes work with others without modifying their source code.
- **An adapter helps two incompatible interfaces to work together.**
- This is the real world definition for an adapter.
- Interfaces may be incompatible but the inner functionality should suit the need.
- **The Adapter design pattern allows otherwise incompatible classes to work together by converting the interface of one class into an interface expected by the clients.**



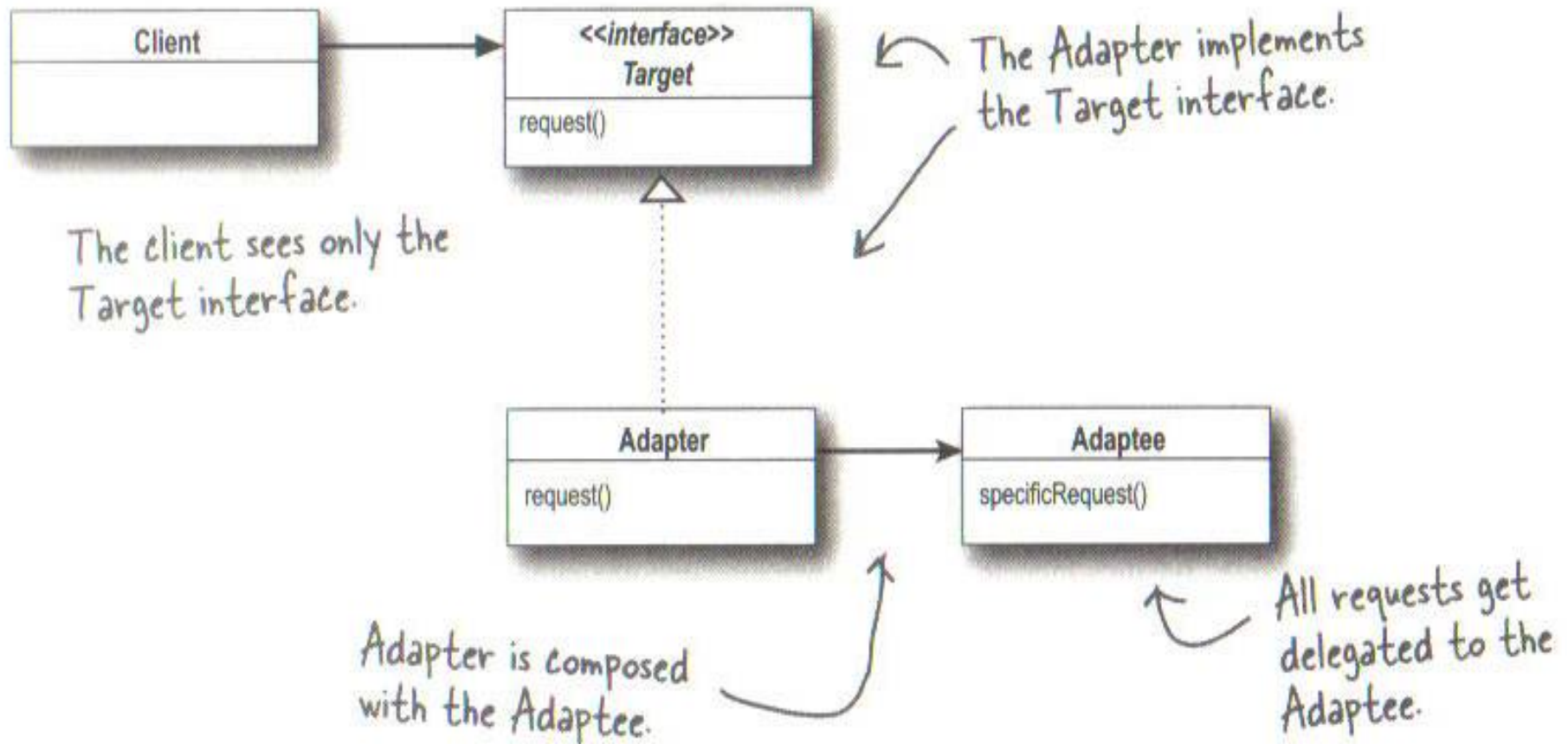
Object-Oriented Adapters



Adapter pattern

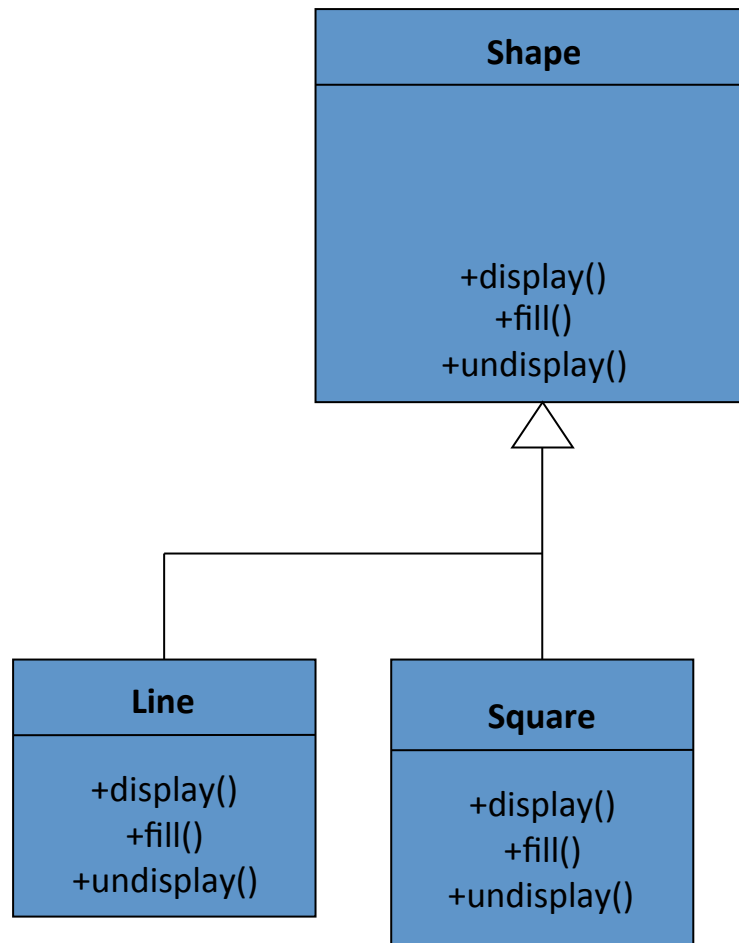
- Intent
 - The Adapter Pattern converts the interface of a class into another interface the clients expect.
 - Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Motivation
 - Sometimes a toolkit or class library can not be used because its interface is incompatible with the interface required by an application
 - We can not change the library interface, since we may not have its source code
 - Even if we did have the source code, we probably should not change the library for each domain-specific application

Adapter Pattern

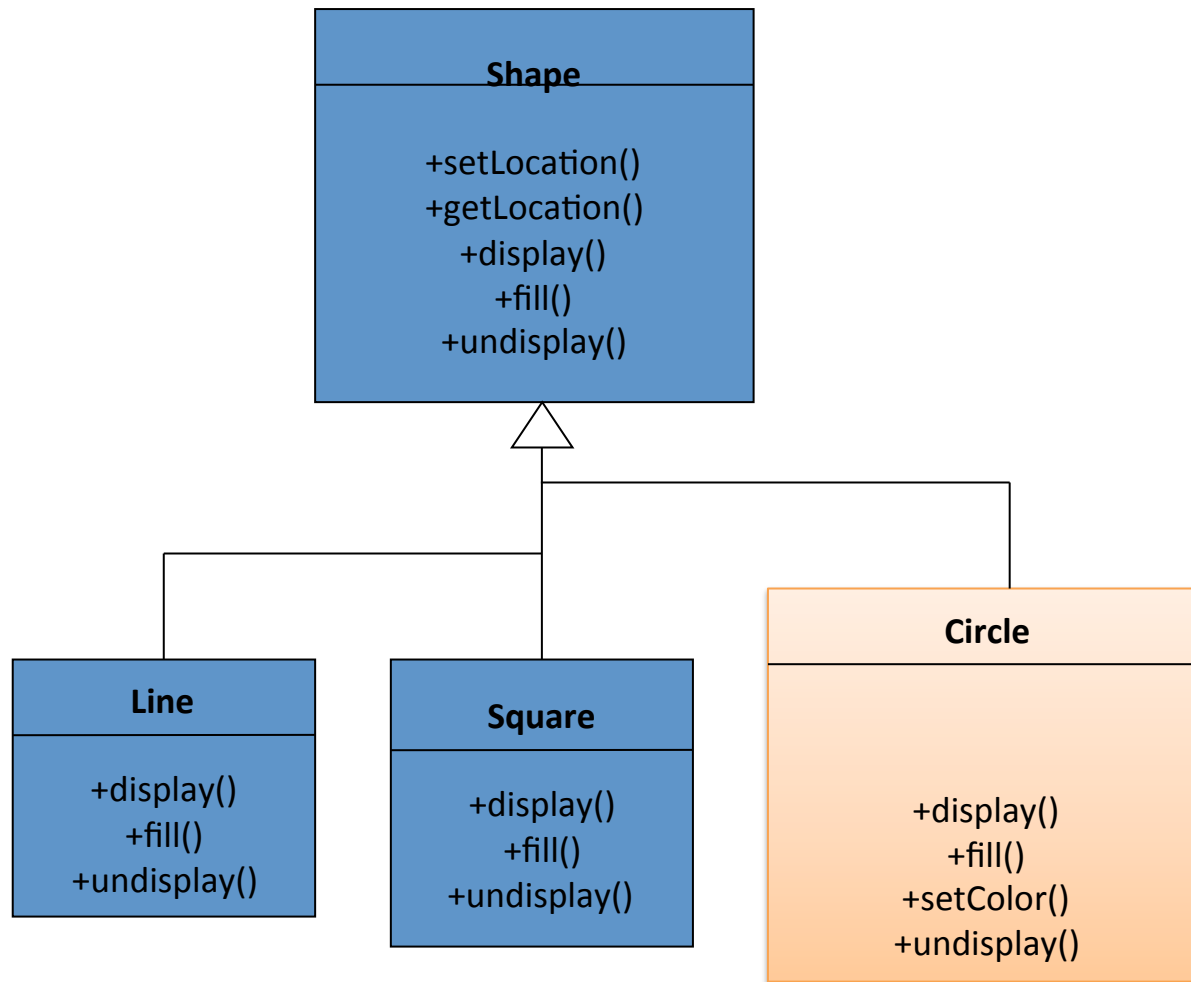


Problem Specification:

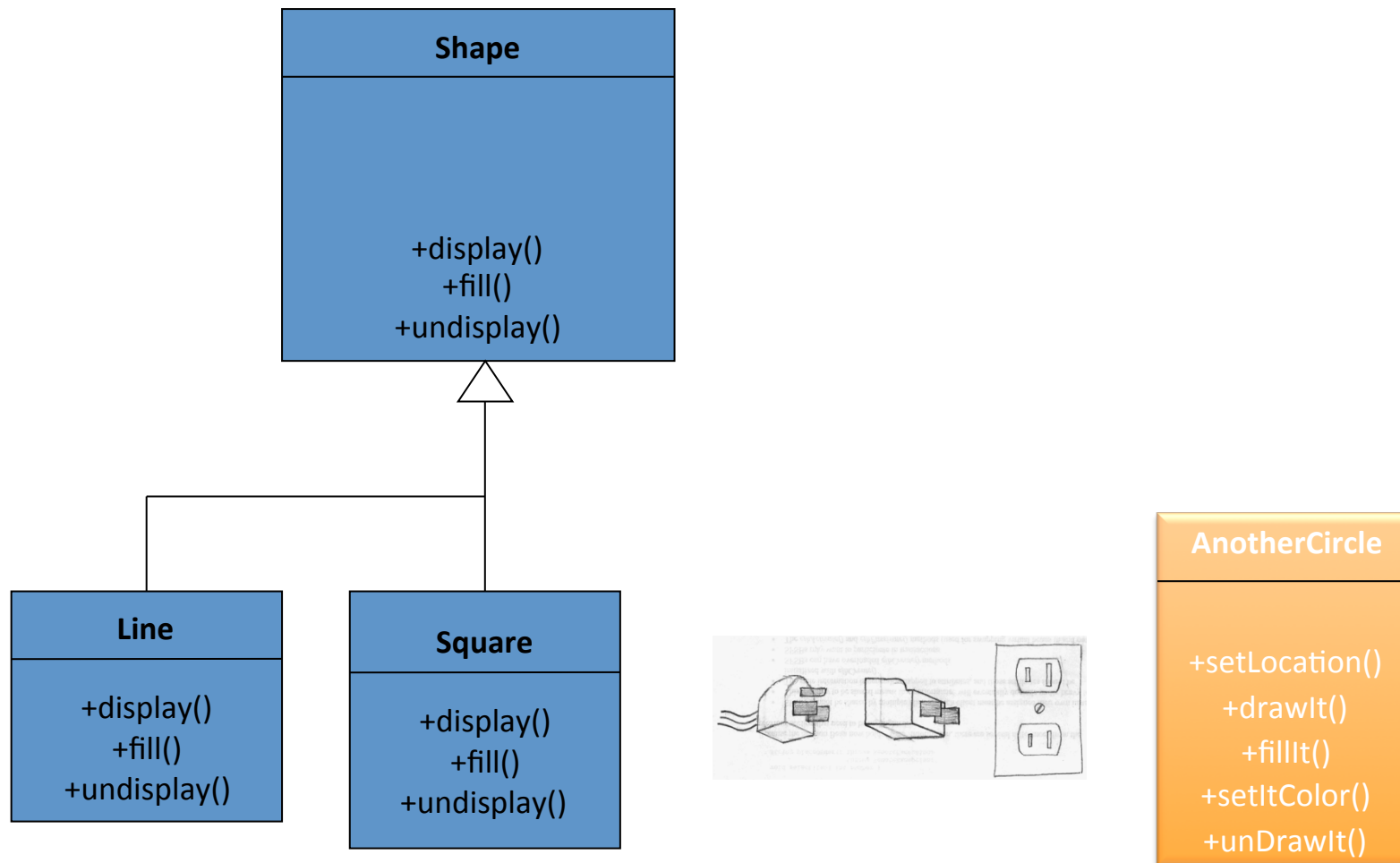
1. You are given the following class library to draw shapes.



2. Now you are asked to add another class to deal with circle. Your plan was:

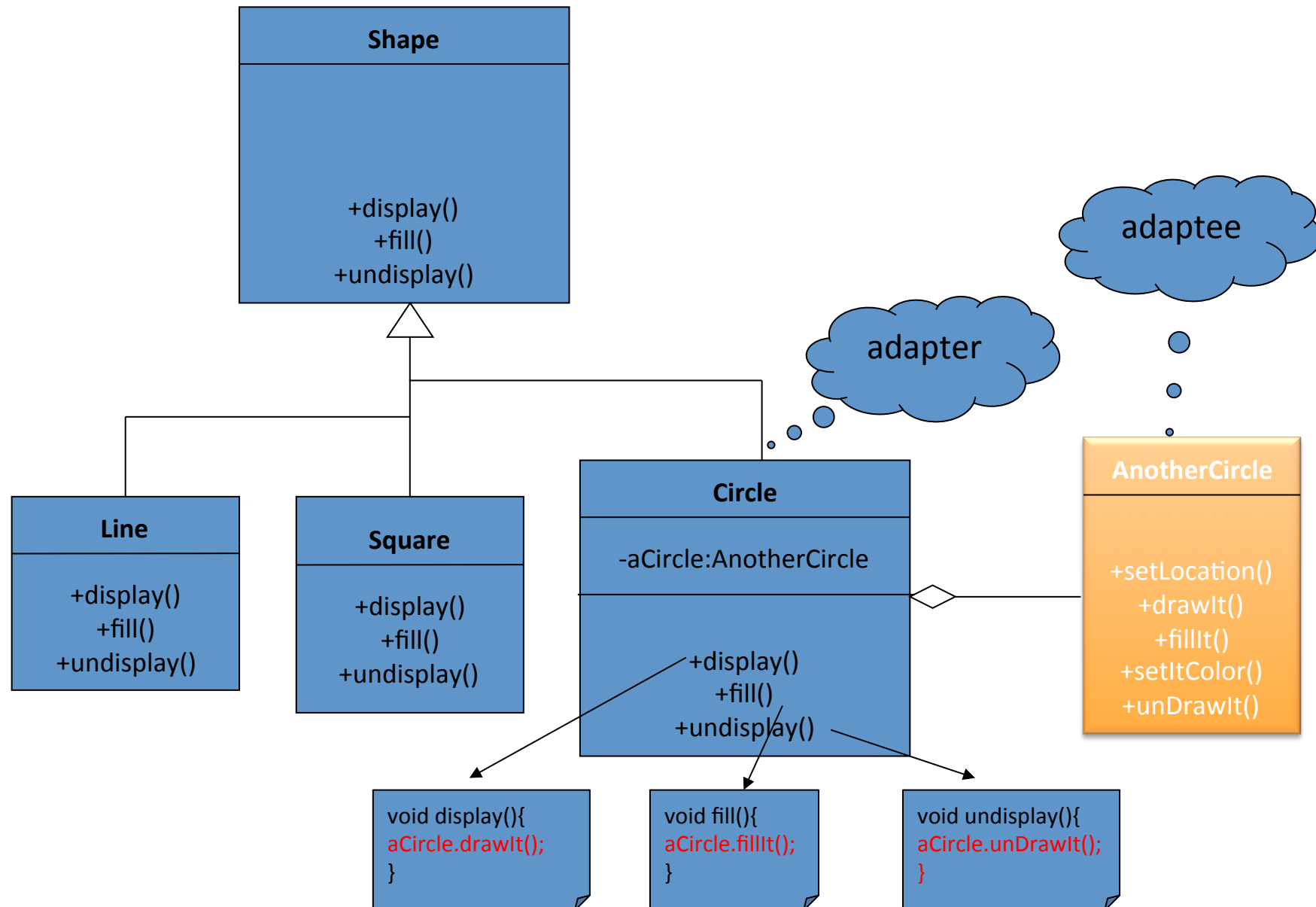


3. Then your client said: “No,no,no”. You have to use this nice class library for drawing circles. Unfortunately, it is from a different vendor, it has a different interface and implemented with different language.



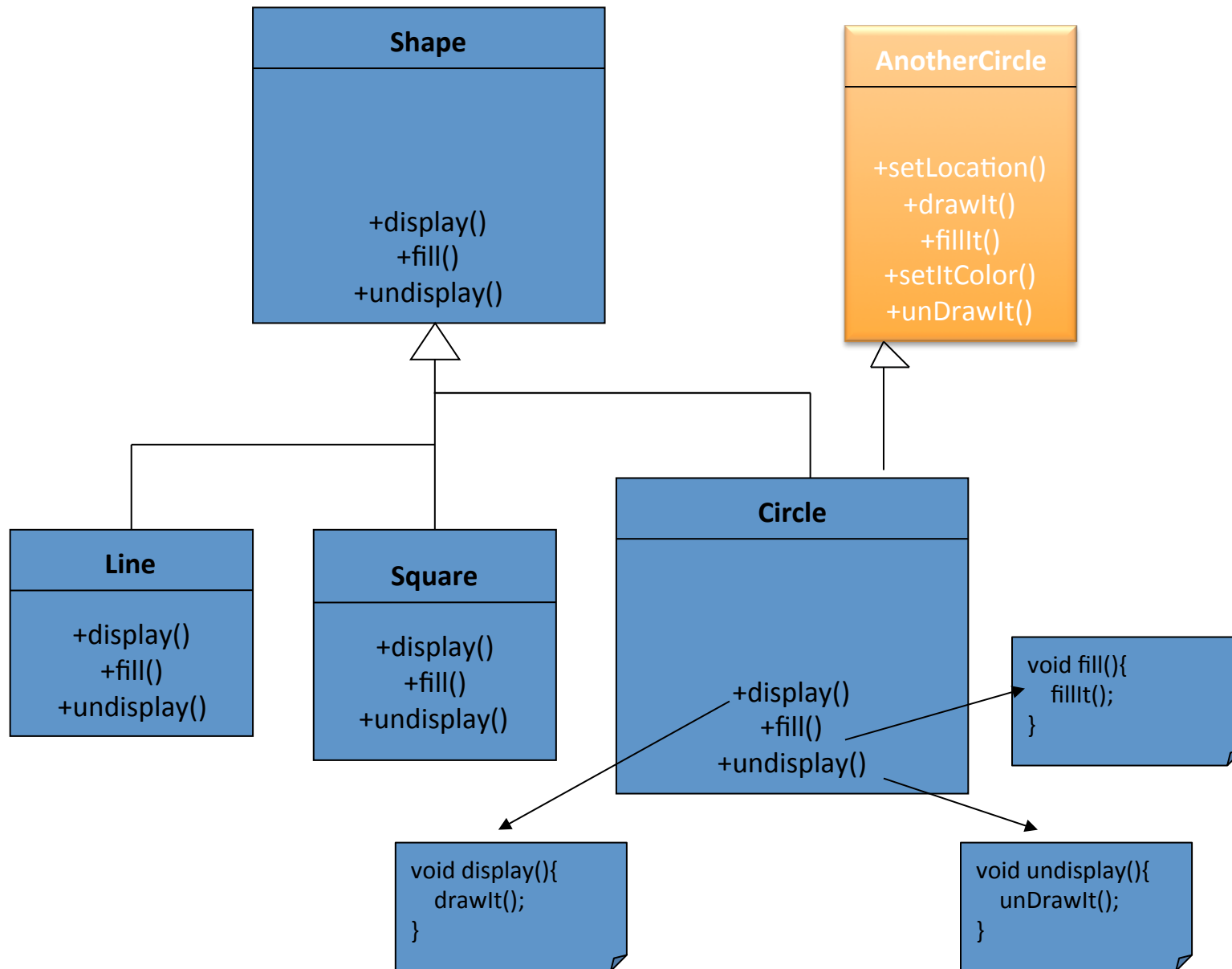
Adapter Design Pattern Solution

Object Adapter



Adapter Design Pattern Solution

Class Adapter

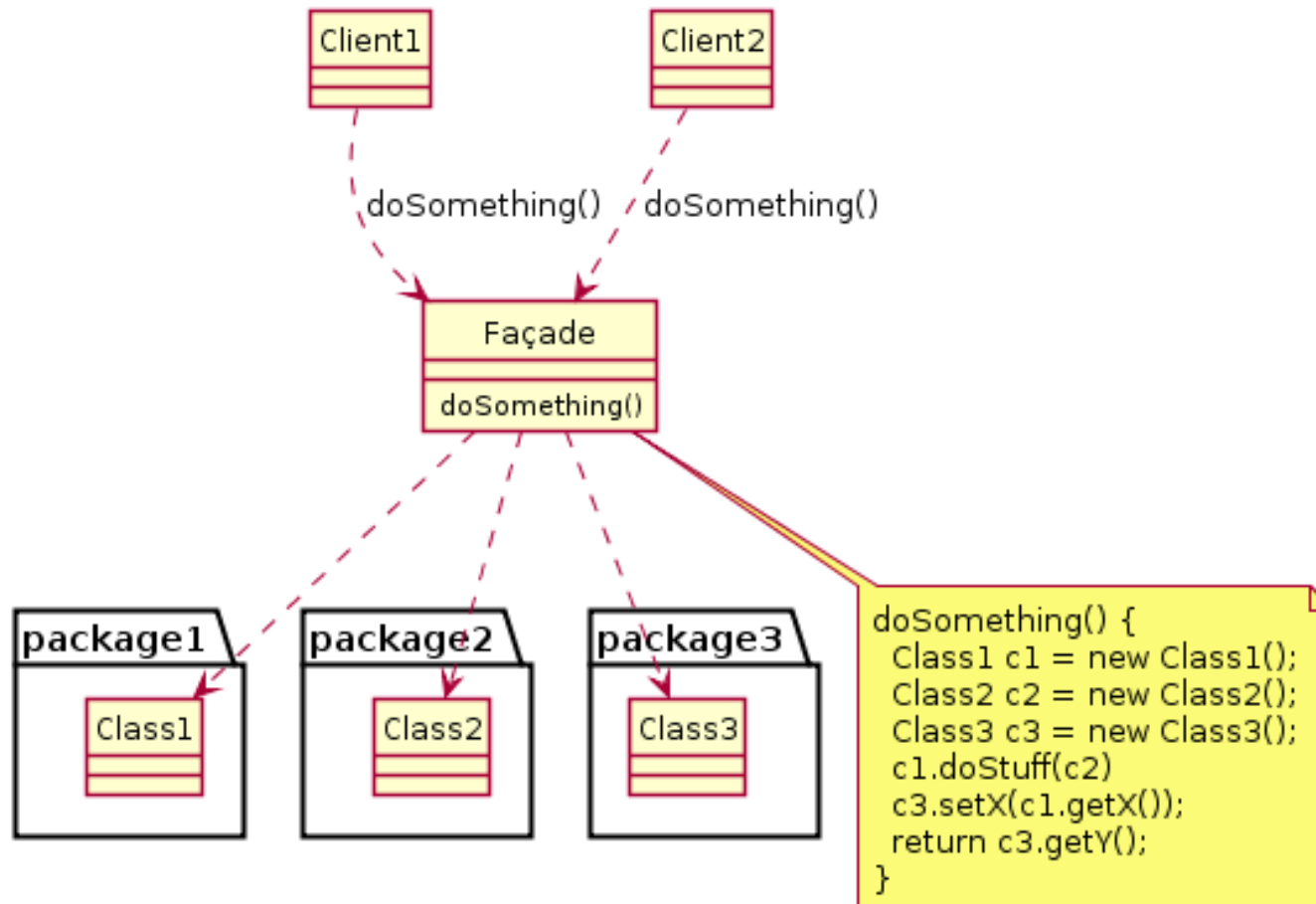


Façade Pattern

When Facade is Used?

- A facade is an object that provides a simplified interface to a larger body of code, such as a class library.
- Wrap a complicated subsystem with a simpler interface.
- The Facade design pattern is often used when
 - a system is very complex or difficult to understand because the system has a large number of interdependent classes or its source code is unavailable.
 - a simple interface is required to access a complex system;
 - the abstractions and implementations of a subsystem are tightly coupled;
 - need an entry point to each level of layered software; or
 - a system is very complex or difficult to understand.
- This pattern hides the complexities of the larger system and provides a simpler interface to the client.
- It typically involves a single wrapper class which contains a set of members required by client.
- These members access the system on behalf of the facade client and hide the implementation details.

Facade Pattern Example



Facade:

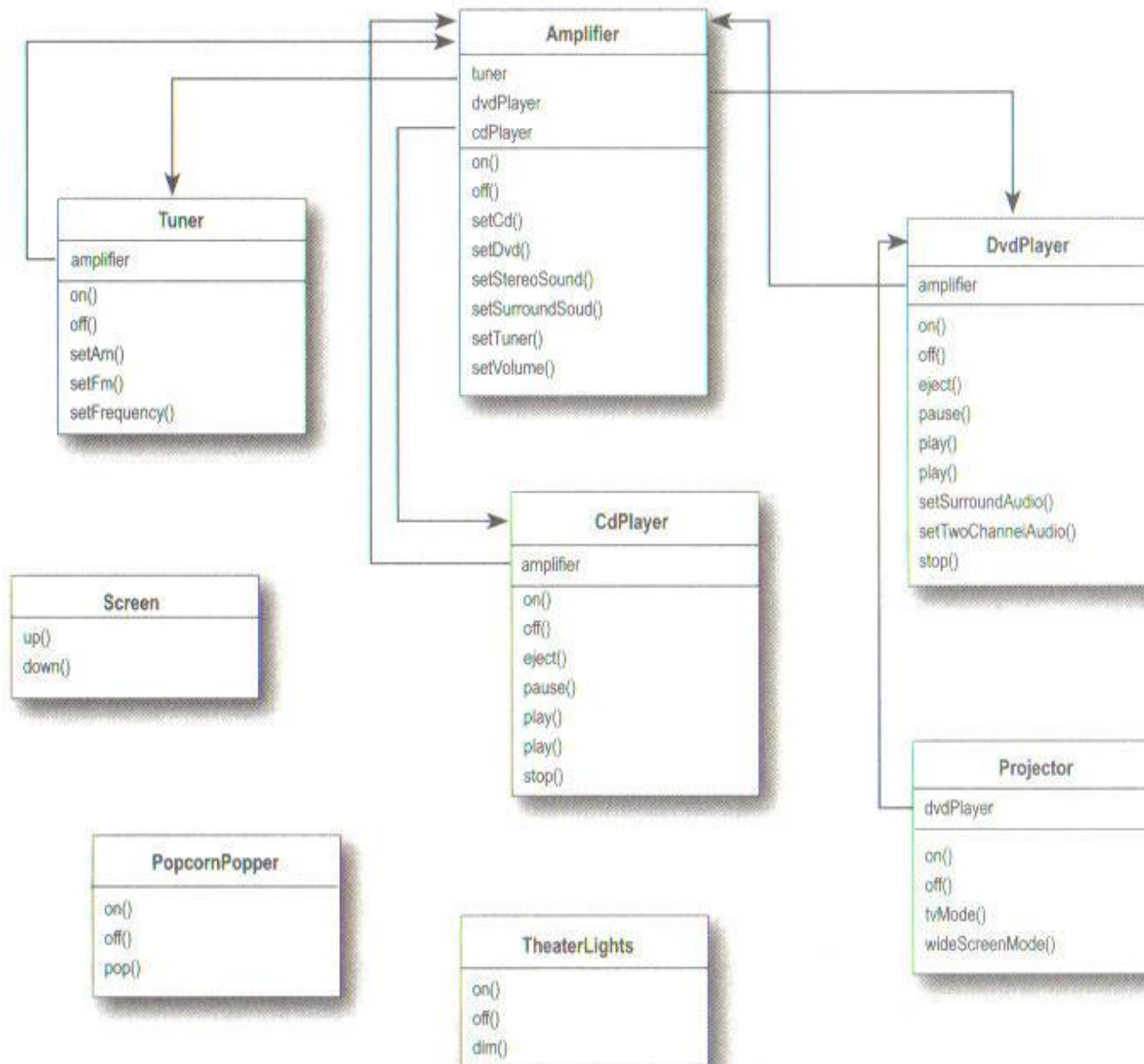
- The facade class abstracts Packages 1, 2, and 3 from the rest of the application.

Clients:

- The objects are using the Facade Pattern to access resources from the Packages.

Watching the 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



That's a lot of classes, a lot of interactions, and a big set of interfaces to learn and use

What needs to be done to watch a movie....

Six different classes involved!

```
popper.on();  
popper.pop();
```

Turn on the popcorn popper and start popping...

```
lights.dim(10);
```

Dim the lights to 10%...

```
screen.down();
```

Put the screen down...

```
projector.on();  
projector.setInput(dvd);  
projector.wideScreenMode();
```

Turn on the projector and put it in wide screen mode for the movie...

```
amp.on();  
amp.setDvd(dvd);  
amp.setSurroundSound();  
amp.setVolume(5);
```

Turn on the amp, set it to DVD, put it in surround sound mode and set the volume to 5...

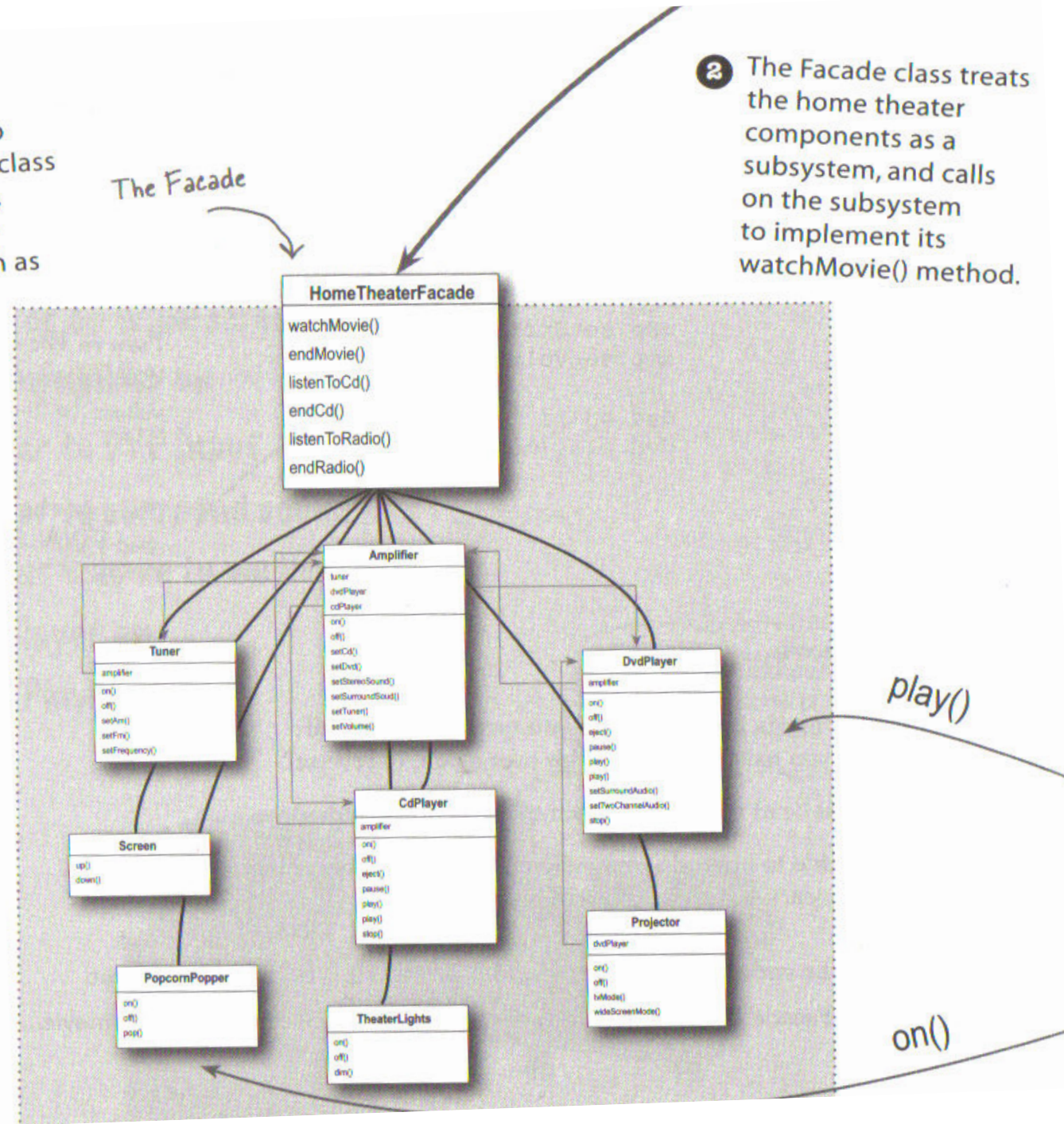
```
dvd.on();  
dvd.play(movie);
```

Turn on the DVD player... and FINALLY, play the movie!

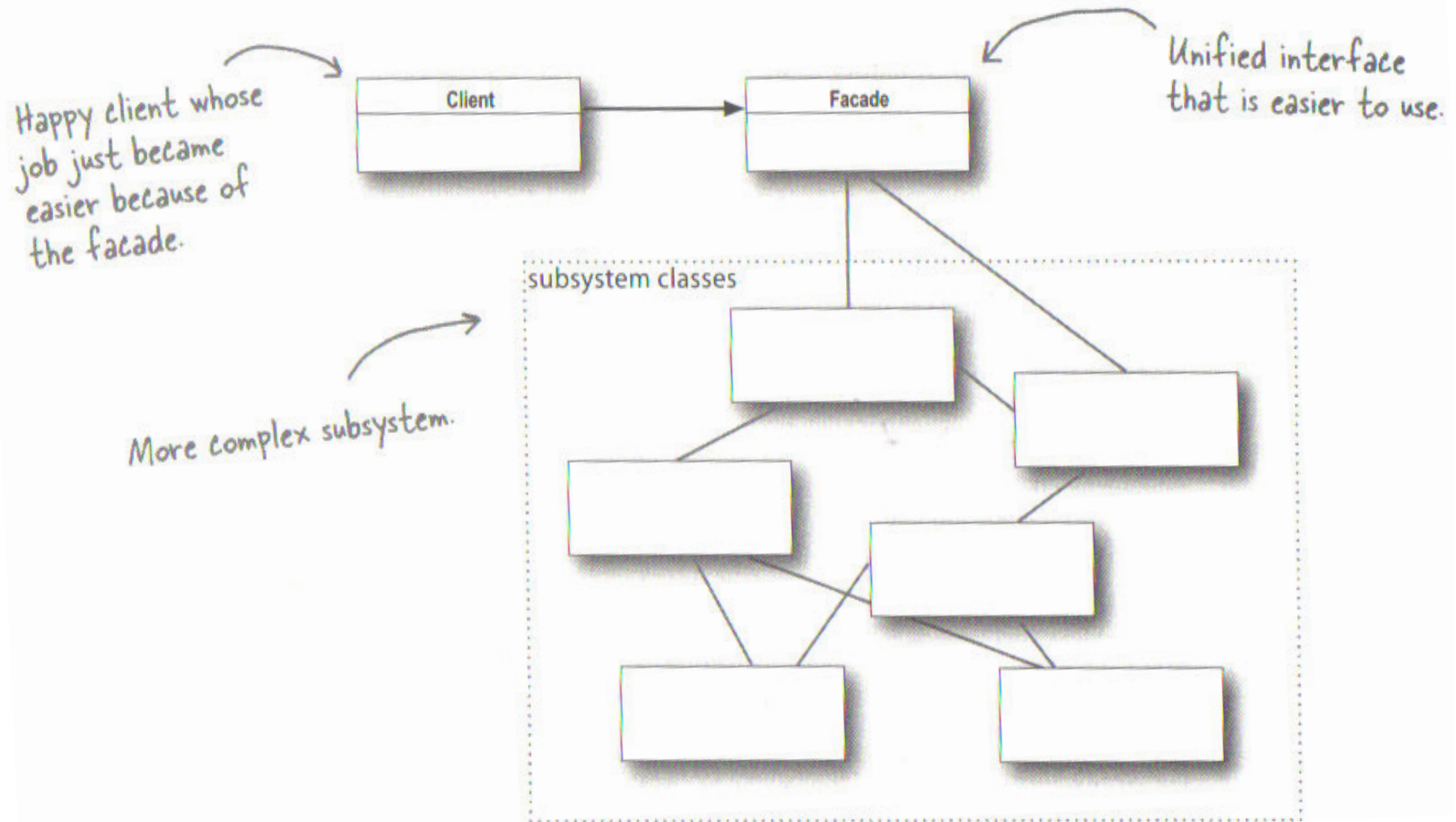
- 1 Okay, time to create a Facade for the home theater system. To do this we create a new class HomeTheaterFacade, which exposes a few simple methods such as watchMovie().

- 2 The Facade class treats the home theater components as a subsystem, and calls on the subsystem to implement its watchMovie() method.

The subsystem the Facade is simplifying.



Façade pattern – Class Diagram



Facade: Compared with Adapter Patterns

- Facade defines a new interface, whereas Adapter uses an old interface.
- Remember that Adapter makes two existing interfaces work together as opposed to defining an entirely new one.
- Facade objects are often Singletons because only one Facade object is required.
- Adapter and Facade are both wrappers; but they are different kinds of wrappers.
 - The intent of Facade is to produce a simpler interface, and the intent of Adapter is to design to an existing interface.
 - While Facade routinely wraps multiple objects and Adapter wraps a single object;
 - Facade could front-end a single complex object and Adapter could wrap several legacy objects.

References

- http://www.tutorialspoint.com/design_pattern/
- Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.
- www.sourcemaking.com/design_patterns
- <http://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns/2707195#2707195>