UNIT-4

Unit 4: More Design Patterns: Fabrication, Indirection, Singleton, Factory, Facade, Publish-Subscribe

Fabrication

We have explored what are patterns and GRASP (in part I), Information Expert in part II and Creator in part II, Controller in Part IV, "Low Coupling" in part V and "High Cohesion" in part VI and Polymorphism in Part VII. In this part VIII, we would focus on next GRASP pattern named "Pure Fabrication". Generally working on existing system, everybody fumbles on a dilemma about changing the existing design. Imagine scenario where the existing classes have low cohesion and high coupling rather it violates the High cohesion and low coupling. It would be overkill to change the existing classes in entirety or even it could be unviable from the perspective of budget(and time). The principle behind this pattern is to resolve such a dilemma by deciding on ***"Whom to assign responsibilities when assigning to existing domain classes violates High cohesion and Low coupling?"*** These domain classes are generally the Information Expert classes.

Problem: Who should be responsible when an expert violates high cohesion and low coupling?

Solution: Assign the responsibility for handling a system event message to a class which is new fictitious (artificial) and doesn't represent a concept in domain.

Assign the cohesive set of responsibilities to such class in order to support high cohesion, low coupling and reuse.

As this class is fictitious hence it can be called as its outcome of imagination or pure fabrication.

Approach:

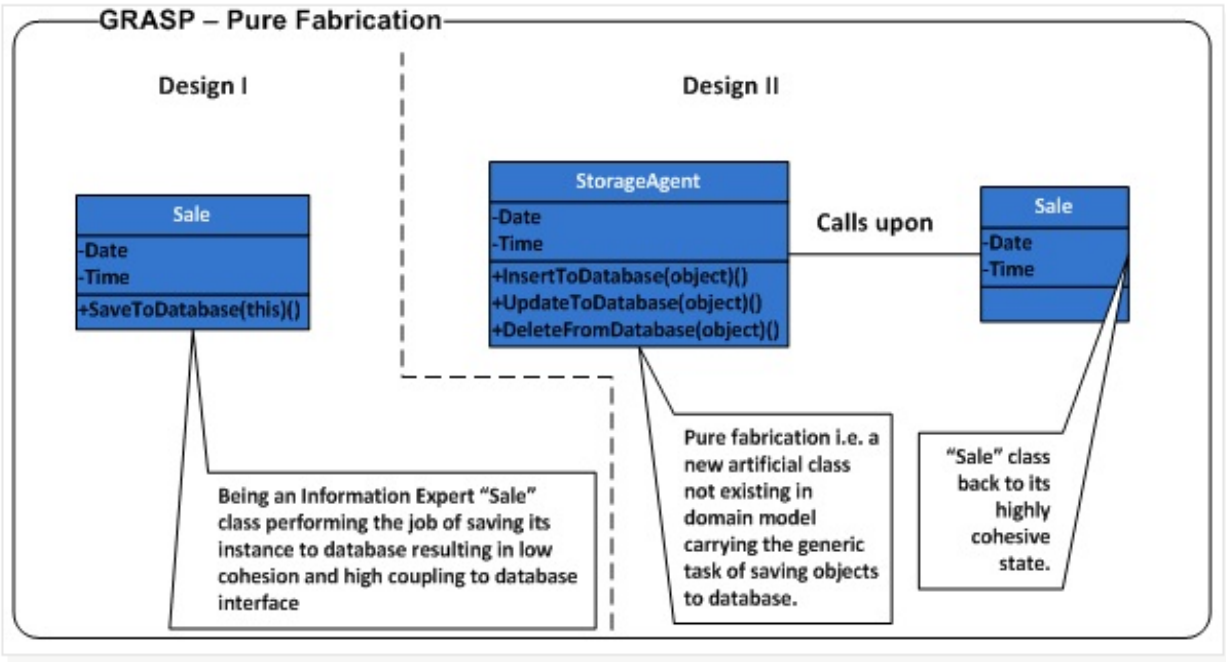
> **Step I:** Closely look at domain/ design model and locate the classes with low cohesion and high coupling. **e.**g. "Sale" class doing all the database operation related to "Sale"
>
> **Step II:** Create a new class to take the responsibility of functionality causing low cohesion and high coupling.

Description

Let's extend an example of POS (Point Of Sale) systems explained in previous part. We are fairly aware of "Sale" class by now. By principle of "Information Expert", the responsibility of saving the instance of "Sale" is with the "Sale" class. (Left part of Fig no.1 i.e. "Design I")

This results in large number of database oriented operations which actually has nothing to do with "Sales". Also there could be additional overhead of transaction related stuff. This leads the "Sale" class towards the low cohesion (remember the definition, "A Class responsible for many

things in related areas") i.e. the database operations are related but it ends up doing many things. Also while performing such database operations; it would need to employ the database interface culminating into low coupling. In fact, such database operations are generic in nature and have potential for reuse.The solution is to create a new class say "StorageAgent"which would interact with database interface and saves the instance of "Sale" class. As "Sale" would be spared from saving its own instance into database thus giving rise to high cohesion and low coupling. In this fashion the "StorageAgent" is also highly cohesive by performing the sole responsibility of saving the instance / object. (Right part of Fig no.1 i.e. "Design II").



Class	Relationship with other classes	Responsibility and method
Sale	Call upon the StorageAgent- delegates the responsibility to other calls	To represent a particular "Sale"
StorageAgent	A separate class having unique responsibility of saving the objects to database. This can be utilized by other classes like "Register", "Payment" etc.	Saving the objects to database through "InsertToDatabase()" "UpdateToDatabase()" "DeleteFromDatabase()"

As demonstrated, the "StorageAgent" is a generic and reusable class. All such classes i.e. pure fabrication (rather purely fabricated) classes are function centric. Other good examples are the adapters, observers and one would find many examples in a service layer.As per Larman, there are 2 approaches of designing which the pure fabrication is the behavioural way. Representation decomposition: Designing the objects the way they represent in the domain

- Behavioural decomposition: Designing the objects the way they do. These are function centric or encapsulate algorithm

Commonly, the pure fabrication is used to place / encapsulate the algorithm or function which doesn't fit well in other classes.

Benefits:

- Supports Low Coupling
- Results in high cohesion
- Promotes reusability

Liabilities /Contradictions:

- Sometimes such design may result into bunch of classes having a single method which is a kind of overkill.

Singleton:

Intent

- Ensure a class has only one instance, and provide a global point of access to it.
- Encapsulated "just-in-time initialization" or "initialization on first use".

Problem

Application needs one, and only one, instance of an object. Additionally, lazy initialization and global access are necessary.

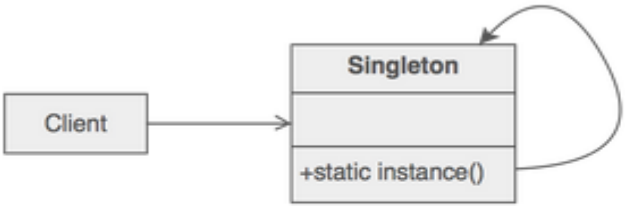
Discussion

Make the class of the single instance object responsible for creation, initialization, access, and enforcement. Declare the instance as a private static data member. Provide a public static ember function that encapsulates all initialization code, and provides access to the instance.The client calls the accessor function (using the class name and scope resolution operator) whenever a reference to the single instance is required. Singleton should be considered only if all three of the following criteria are satisfied:

- Ownership of the single instance cannot be reasonably assigned
- Lazy initialization is desirable
- Global access is not otherwise provided for

If ownership of the single instance, when and how initialization occurs, and global access are not issues, Singleton is not sufficiently interesting. The Singleton pattern can be extended to support access to an application-specific number of instances. The "static member function accessor" approach will not support sub classing of the Singleton class.

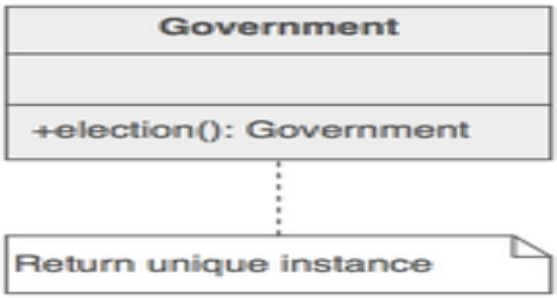
Structure



Make the class of the single instance responsible for access and "initialization on first use". The single instance is a private static attribute. The accessor function is a public static method.

Example

The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance. It is named after the singleton set, which is defined to be a set containing one element. The office of the President of the United States is a Singleton. The United States Constitution specifies the means by which a president is elected, limits the term of office, and defines the order of succession. As a result, there can be at most one active president at any given time. Regardless of the personal identity of the active president, the title, "The President of the United States" is a global point of access that identifies the person in the office.



Check list

1. Define a private `static` attribute in the "single instance" class.
2. Define a public `static`accessor function in the class.
3. Do "lazy initialization" (creation on first use) in the accessor function.
4. Define all constructors to be `protected` or `private`.
5. Clients may only use the accessor function to manipulate the Singleton.

Rules of thumb

- Abstract Factory, Builder, and Prototype can use Singleton in their implementation.
- Facade objects are often Singletons because only one Facade object is required.
- State objects are often Singletons.
- The advantage of Singleton over global variables is that you are absolutely sure of the number of instances when you use Singleton, and, you can change your mind and manage any number of instances.
- The Singleton design pattern is one of the most inappropriately used patterns. Singletons are intended to be used when a class must have exactly one instance, no more, no less. Designers frequently use Singletons in a misguided attempt to replace global variables. A Singleton is, for intents and purposes, a global variable. The Singleton does not do away with the global; it merely renames it.
- When is Singleton unnecessary? Short answer: most of the time. Long answer: 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. The real problem with Singletons is that they give you such a good excuse not to think carefully about the appropriate visibility of an object. Finding the right balance of exposure and protection for an object is critical for maintaining flexibility.
- Our group had a bad habit of using global data, so I did a study group on Singleton. The next thing I know Singletons appeared everywhere and none of the problems related to global data went away. The answer to the global data question is not, "Make it a Singleton." The answer is, "Why in the hell are you using global data?" Changing the name doesn't change the problem. In fact, it may make it worse because it gives you the opportunity to say, "Well I'm not doing that, I'm doing this" – even though this and that are the same thing.

Factory

Intent

- Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- Defining a "virtual" constructor.
- The `new` operator considered harmful.

Problem

A framework needs to standardize the architectural model for a range of applications, but allow for individual applications to define their own domain objects and provide for their instantiation.

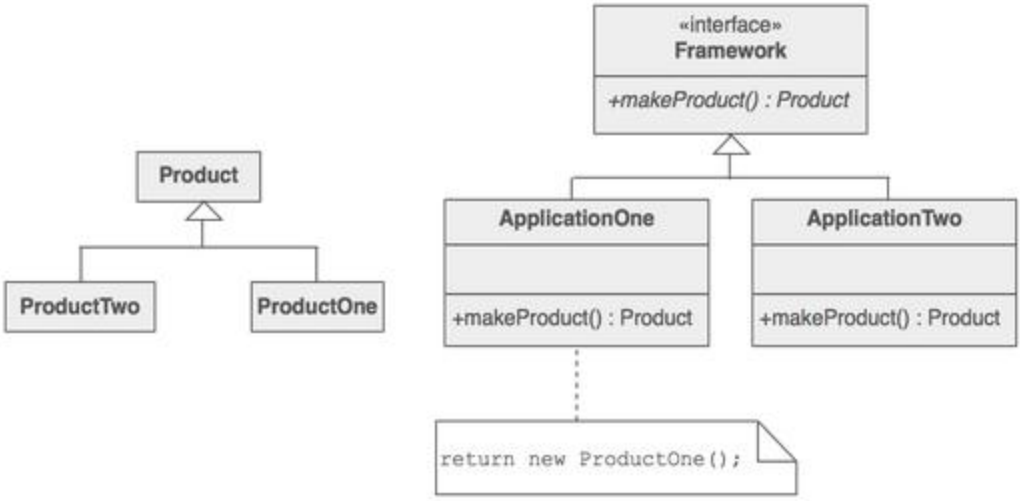
Discussion

Factory Method is to creating objects as Template Method is to implementing an algorithm. A superclass specifies all standard and generic behavior (using pure virtual "placeholders" for creation steps), and then delegates the creation details to subclasses that are supplied by the client. Factory Method makes a design more customizable and only a little more complicated. Other design patterns require new classes, whereas Factory Method only requires a new operation. People often use Factory Method as the standard way to create objects; but it isn't necessary if: the class that's instantiated never changes, or instantiation takes place in an operation that subclasses can easily override (such as an initialization operation).

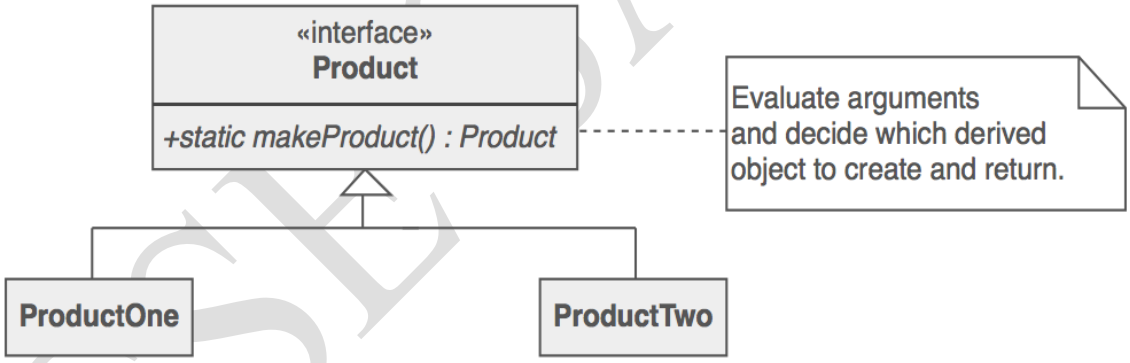
Factory Method is similar to Abstract Factory but without the emphasis on families. Factory Methods are routinely specified by an architectural framework, and then implemented by the user of the framework.

Structure

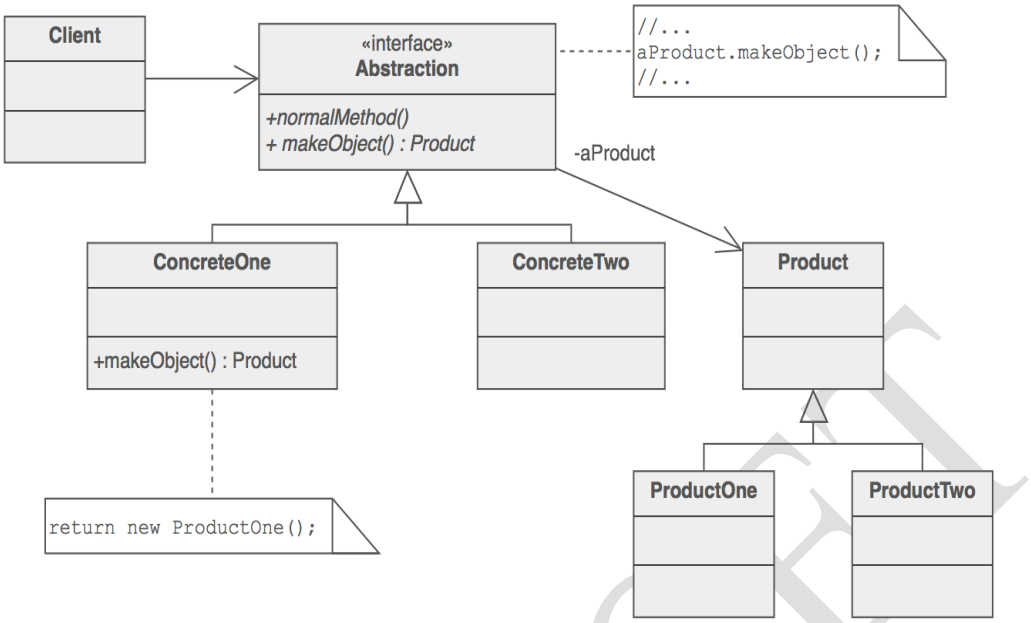
The implementation of Factory Method discussed in the Gang of Four (below) largely overlaps with that of Abstract Factory. For that reason, the presentation in this chapter focuses on the approach that has become popular since.



An increasingly popular definition of factory method is: a static method of a class that returns an object of that class' type. But unlike a constructor, the actual object it returns might be an instance of a subclass. Unlike a constructor, an existing object might be reused, instead of a new object created. Unlike a constructor, factory methods can have different and more descriptive names (e.g. Color.make_RGB_color(float red, float green, float blue) and Color.make_HSB_color(float hue, float saturation, float brightness)

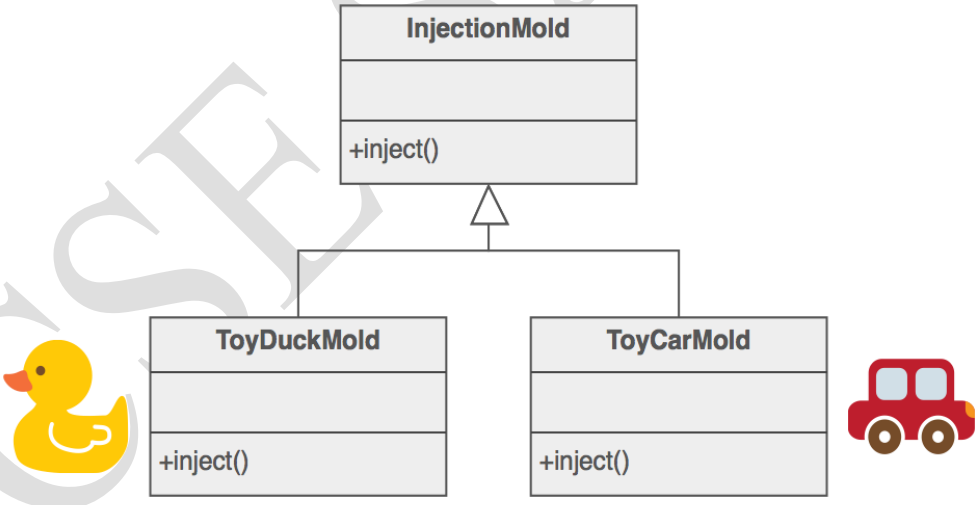


The client is totally decoupled from the implementation details of derived classes. Polymorphic creation is now possible.



Example

The Factory Method defines an interface for creating objects, but lets subclasses decide which classes to instantiate. Injection molding presses demonstrate this pattern. Manufacturers of plastic toys process plastic molding powder, and inject the plastic into molds of the desired shapes. The class of toy (car, action figure, etc.) is determined by the mold.



Check list

1. If you have an inheritance hierarchy that exercises polymorphism, consider adding a polymorphic creation capability by defining a `static` factory method in the base class.
2. Design the arguments to the factory method. What qualities or characteristics are necessary and sufficient to identify the correct derived class to instantiate?

3. Consider designing an internal "object pool" that will allow objects to be reused instead of created from scratch.
4. Consider making all constructors `private` or `protected`.

Rules of thumb

- Abstract Factory classes are often implemented with Factory Methods, but they can be implemented using Prototype.
- Factory Methods are usually called within Template Methods.
- Factory Method: creation through inheritance. Prototype: creation through delegation.
- Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed.
- Prototype doesn't require sub classing, but it does require an Initialize operation. Factory Method requires subclassing, but doesn't require Initialize.
- The advantage of a Factory Method is that it can return the same instance multiple times, or can return a subclass rather than an object of that exact type.
- Some Factory Method advocates recommend that as a matter of language design (or failing that, as a matter of style) absolutely all constructors should be private or protected. It's no one else's business whether a class manufactures a new object or recycles an old one.
- The `new` operator considered harmful. There is a difference between requesting an object and creating one. The `new` operator always creates an object, and fails to encapsulate object creation. A Factory Method enforces that encapsulation, and allows an object to be requested without inextricable coupling to the act of creation.

Façade:

Intent

- Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- Wrap a complicated subsystem with a simpler interface.

Problem

A segment of the client community needs a simplified interface to the overall functionality of a complex subsystem.

Discussion

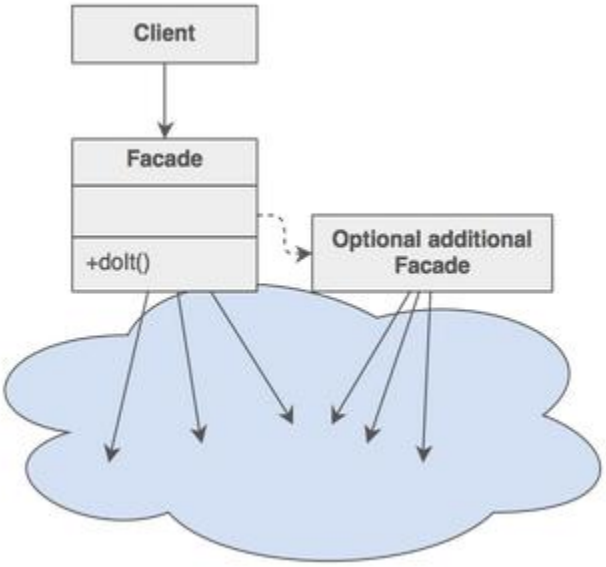
Facade discusses encapsulating a complex subsystem within a single interface object. This reduces the learning curve necessary to successfully leverage the subsystem. It also promotes decoupling the subsystem from its potentially many clients. On the other hand, if the Facade is

the only access point for the subsystem, it will limit the features and flexibility that "power users" may need.

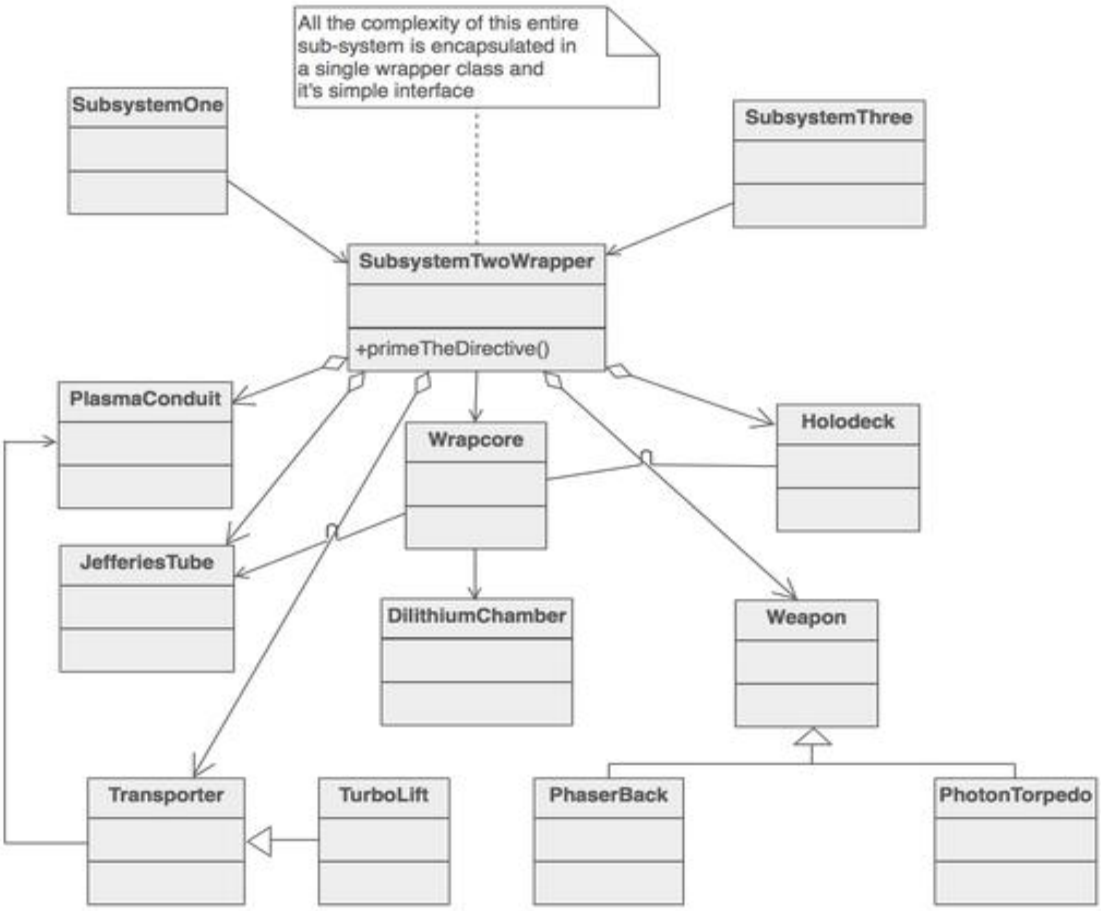
The Facade object should be a fairly simple advocate or facilitator. It should not become an all-knowing oracle or "god" object.

Structure

Facade takes a "riddle wrapped in an enigma shrouded in mystery", and interjects a wrapper that tames the amorphous and inscrutable mass of software.

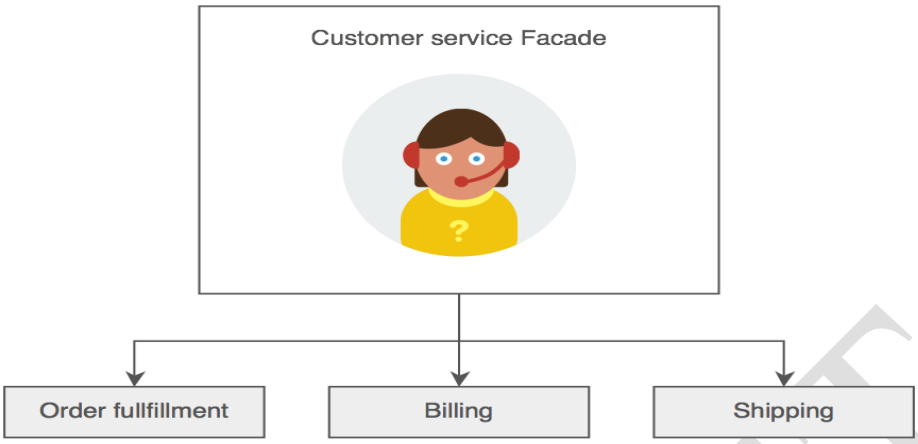


SubsystemOne and SubsystemThree do not interact with the internal components of SubsystemTwo. They use the SubsystemTwoWrapper "facade" (i.e. the higher level abstraction).



Example

The Facade defines a unified, higher level interface to a subsystem that makes it easier to use. Consumers encounter a Facade when ordering from a catalog. The consumer calls one number and speaks with a customer service representative. The customer service representative acts as a Facade, providing an interface to the order fulfillment department, the billing department, and the shipping department.



Check list

1. Identify a simpler, unified interface for the subsystem or component.
2. Design a 'wrapper' class that encapsulates the subsystem.
3. The facade/wrapper captures the complexity and collaborations of the component, and delegates to the appropriate methods.
4. The client uses (is coupled to) the Facade only.
5. Consider whether additional Facades would add value.

Rules of thumb

- 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.
- Whereas Flyweight shows how to make lots of little objects, Facade shows how to make a single object represent an entire subsystem.
- Mediator is similar to Facade in that it abstracts functionality of existing classes. Mediator abstracts/centralizes arbitrary communications between colleague objects. It routinely "adds value", and it is known/referenced by the colleague objects. In contrast, Facade defines a simpler interface to a subsystem, it doesn't add new functionality, and it is not known by the subsystem classes.
- Abstract Factory can be used as an alternative to Facade to hide platform-specific classes.
- 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.

Publish-Subscribe:

The next design pattern is a bit different than the last ones. It is moreArchitectural, in the sense that it pertains to how classes are put together to achieve a certain goal. The motivating scenario is as follows. Suppose we have an object in the system that is in charge of generating news of interest for the rest of the application. For instance, perhaps it is in charge of keeping track of user input, and tells the rest of the application whenever the user does something of interest. Or, it is in charge of maintaining a clock, and tells the rest of the application whenever the clock ticks one time step. Is there a general approach for handling this kind of thing?

If we analyze the situation carefully, you'll notice that we have two sorts of entities around: a publisher that is in charge of publishing or generating items of interest to the rest of the application, and the dualSubscribers that are the parts of the application that are interestedin getting these updates. (The Publish-Subscribe design pattern is sometimes called the Observer design pattern, in which context publishers are called observables, and subscribers are called observers.

Think about the operations that we would like to support on subscribers, rst. Well, the main thing we want a subscriber to be able to do is to be noted when a news item is published. Thus, this calls for a subscriber implementing the following interface, parameterized by aType E of values conveyed during the notification (e.g., the news item itself)

```
public interface Subscriber<E> {
   public void getPublication (E arg);
}
```

What about the other end? What do we want a publisher to do? we need to register (or subscribe) a subscriber, so that that subscriber can be notified when a news item is produced. The other operation, naturally enough, is to publish a piece of data,which should let every subscriber know that the data has been produced. When notifying a subscriber, we will also pass a value (perhaps the news item in question). This leads to the following interface that a publisher should implement, parameterized over a type E of values to pass when notifying a subscriber.