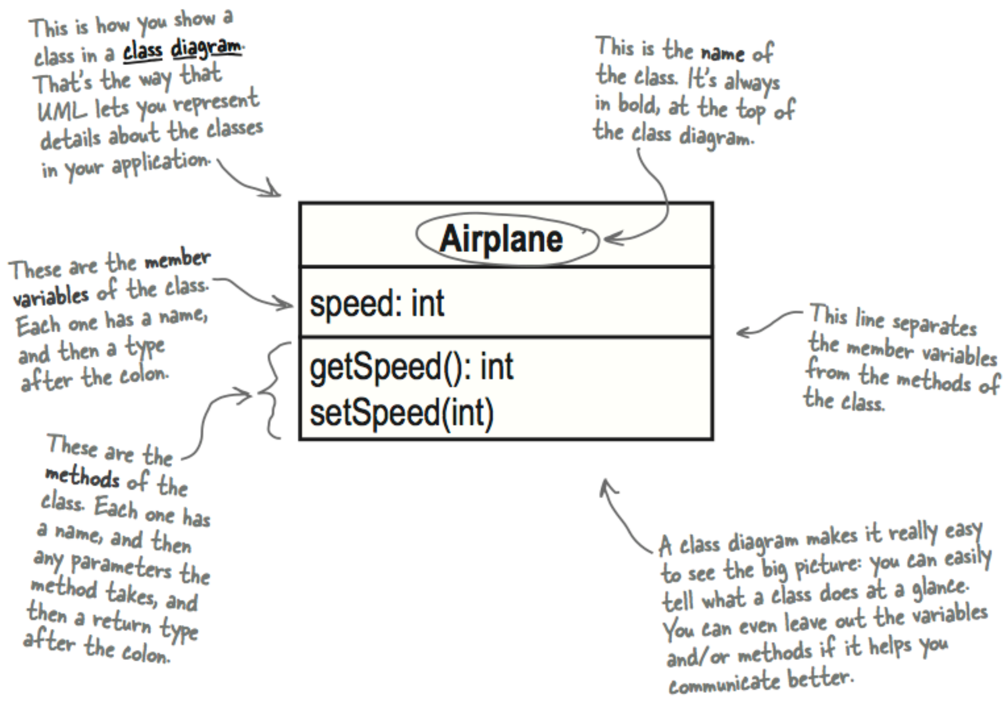
1. UML and Class diagrams


This is how you show a class in a class diagram. That's the way that UML lets you represent details about the classes in your application.
This is the name of the class. It's always in bold, at the top of the class diagram.
Airplane
speed: int
getSpeed(): int
setSpeed(int)
These are the member variables of the class. Each one has a name, and then a type after the colon.
This line separates the member variables from the methods of the class.
These are the methods of the class. Each one has a name, and then any parameters the method takes, and then a return type after the colon.
A class diagram makes it really easy to see the big picture: you can easily tell what a class does at a glance. You can even leave out the variables and/or methods if it helps you communicate better.

2. Important OOAD Concepts

Inheritance:
One class inherits behavior from another class, and can then reuse or change that behavior if needed. It lets you build classes based on other classes, and avoid duplicating and repeating code.
A subclass can override its superclass's behavior to change how a method works.
Some concepts about inheritance: **superclass, subclass, overriding.**

Polymorphism:
Polymorphism has two major applications in an OOP language. The first is that an object may provide different implementations of one of its methods depending on the type of the input parameters. The second is that code written for a given type of data may be used on data with a derived type, i.e. methods understand the class hierarchy of a type.
Polymorphism is closely related to inheritance. When one class inherits from another, then polymorphism allows a subclass to stand in for superclass.
Example: Airplane plane = new Jet();
Benefit: if you need new functionality, you could write a new subclass of Airplane, But since your code uses the superclass, your new subclass will work without any changes to the rest of our code.
Polymorphism is a built-in feature in python (thanks to strong and dynamic type system)

Encapsulation:
Encapsulation is when you **separate** or **hide** one part of your code from the rest of your code in such a way that it It can protect information in your code from being used incorrectly.

3. Great Software

Three steps to achieve great software:

1. Make sure your software does what the customer wants it to do
2. Apply basic OO principles to add flexibility
3. Strive for a maintainable, reusable design

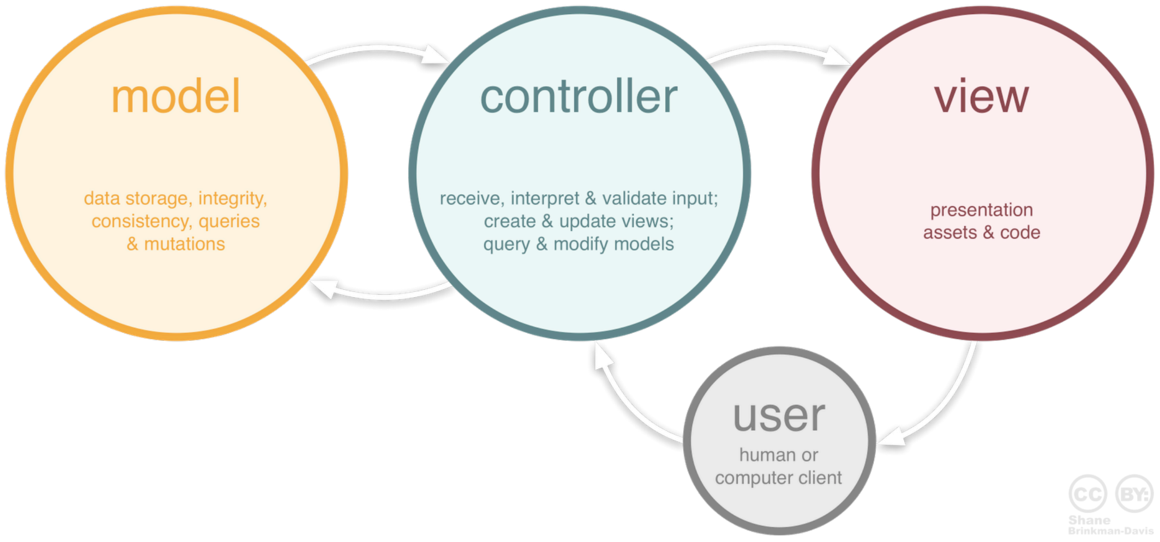
Design Patterns

Explanation and java implementations can be found here:
http://www.tutorialspoint.com/design_pattern/index.htm
Python implementations can be found in github/Misc/DesignPattern

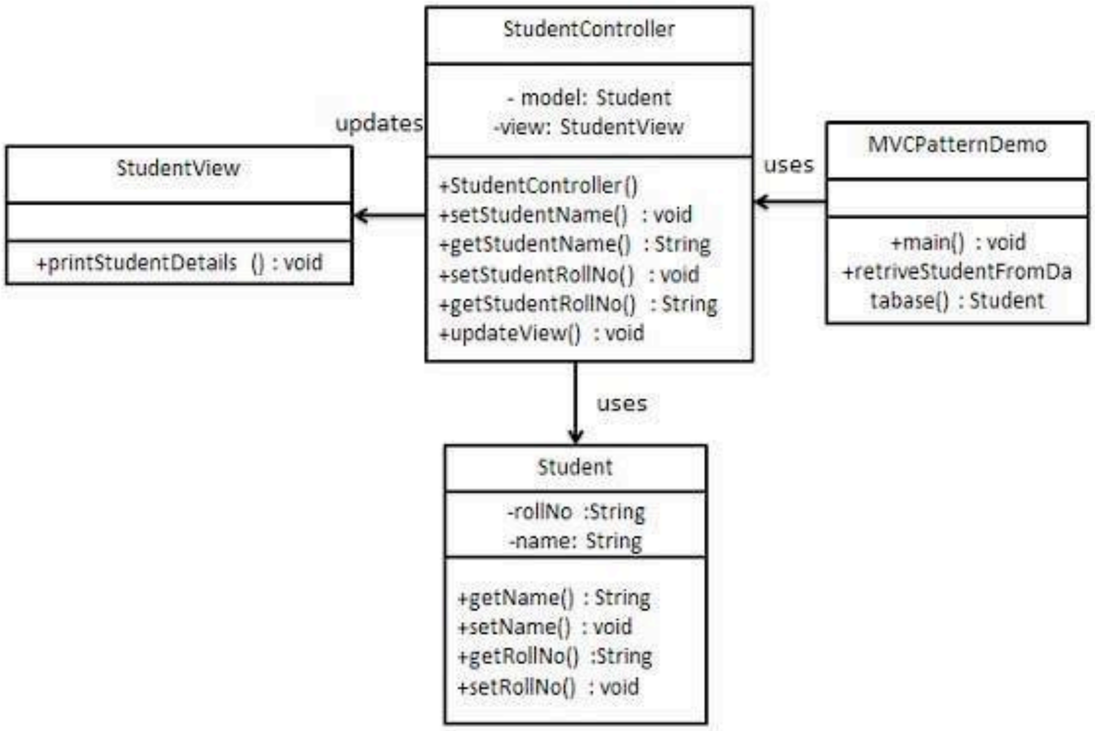
#MVC design pattern

MVC Pattern stands for Model-View-Controller Pattern. This pattern is used to separate application's concerns.

- **Model** - Model represents an object or carrying data. It can also have logic to update controller if its data changes.
- **View** - View represents the visualization of the data that model contains.
- **Controller** - Controller acts on both model and view. It controls the data flow into model object and updates the view whenever data changes. It keeps view and model separate.


model
data storage, integrity, consistency, queries & mutations
controller
receive, interpret & validate input; create & update views; query & modify models
view
presentation assets & code
user
human or computer client
Shane Brinkman-Davis

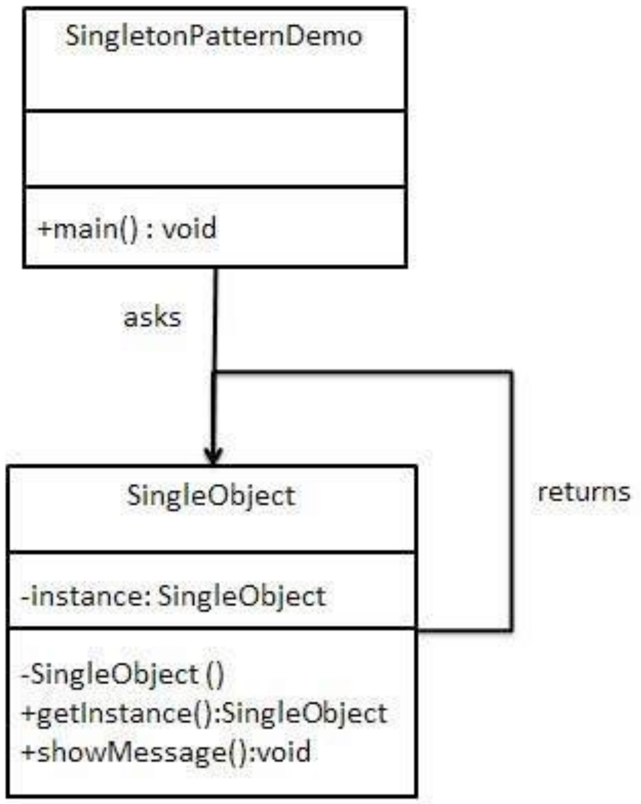
For example, in java


StudentController
- model: Student
-view: StudentView
+StudentController()
+setStudentName() : void
+getStudentName() : String
+setStudentRollNo() : void
+getStudentRollNo() : String
+updateView() : void
updates
StudentView
+printStudentDetails () : void
uses
MVCPatternDemo
+main() : void
+retriveStudentFromDatabase() : Student
uses
Student
-rollNo :String
-name: String
+getName() : String
+setName() : void
+getRollNo() :String
+setRollNo() : void

#Singleton design pattern

The singleton pattern is a design pattern that restricts the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system. The concept is sometimes generalized to systems that operate more efficiently when only one object exists, or that restrict the instantiation to a certain number of objects.

This pattern involves a single class, which is responsible to create an object while making sure that only single object gets created. This class provides a way to access its only object that can be accessed directly without need to instantiate the object of the class.



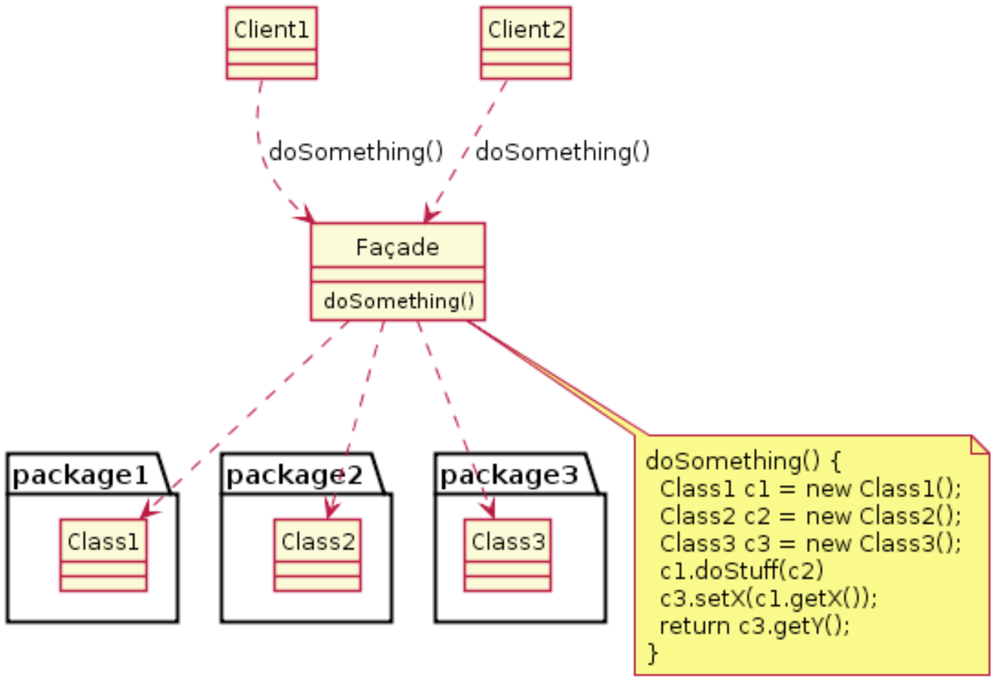
#Facade design pattern

A facade is an object that provides a simplified interface to a larger body of code, such as a class library.

Façade pattern falls under the hood of Structural Design Patterns. Façade is nothing but an interface that hides the inside details and complexities of a system and provides a simplified ―front end‖ to the client. With façade pattern, client can work with the interface easily and get the job done without being worried of the complex operations being done by the system.

An important point to understand about the Facade pattern is that it provides a simplified interface to a part of the system, thereby providing ease-of-use for a sub-set of the functionality rather than complete functionality. Beauty of this is that the underlying classes are still available to the client if the client wants additional features/greater control and customization that are not provided in the current context of Facade pattern implementation.

Because of the above reason, Facade pattern is not about ―encapsulating‖ the sub-system, rather about providing a simplified interface for a chosen functionality.



```
doSomething() {
  Class1 c1 = new Class1();
  Class2 c2 = new Class2();
  Class3 c3 = new Class3();
  c1.doStuff(c2)
  c3.setX(c1.getX());
  return c3.getY();
}
```

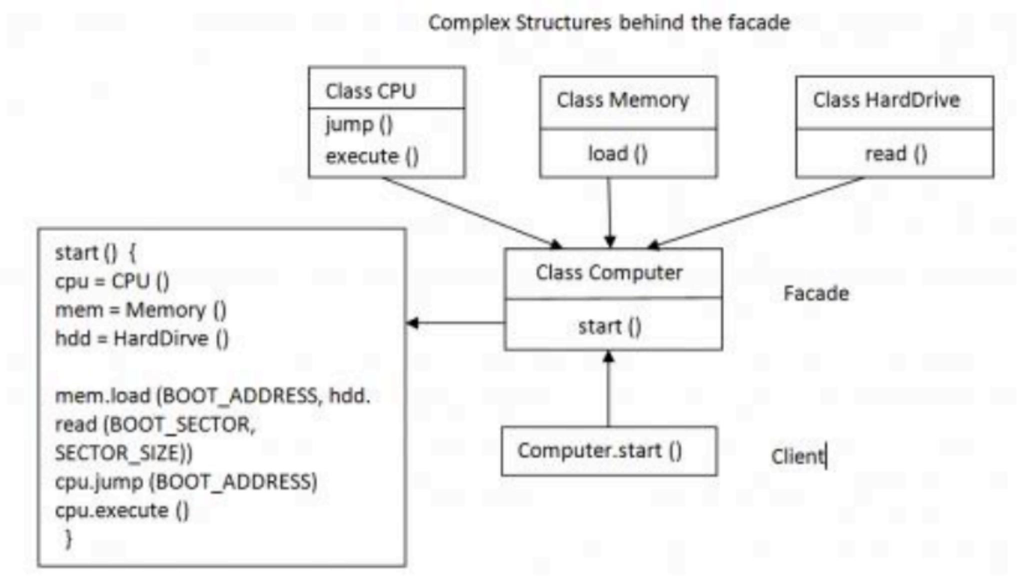
Facade

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.

For example,


Complex Structures behind the facade
Class CPU
jump ()
execute ()
Class Memory
load ()
Class HardDrive
read ()
Class Computer
start ()
Facade
Computer.start ()
Client
start () {
cpu = CPU ()
mem = Memory ()
hdd = HardDirve ()
mem.load (BOOT_ADDRESS, hdd.
read (BOOT_SECTOR,
SECTOR_SIZE))
cpu.jump (BOOT_ADDRESS)
cpu.execute ()
}

#Observer design pattern

The observer pattern (a subset of the publish/subscribe pattern) is a software design pattern in which an object, called the subject, maintains a list of its dependants, called observers, and notifies them automatically of any state changes, usually by calling one of their methods. It is mainly used to implement distributed event handling systems .

Typically in the Observer Pattern, we would have:

1. Publisher class that would contain methods for:

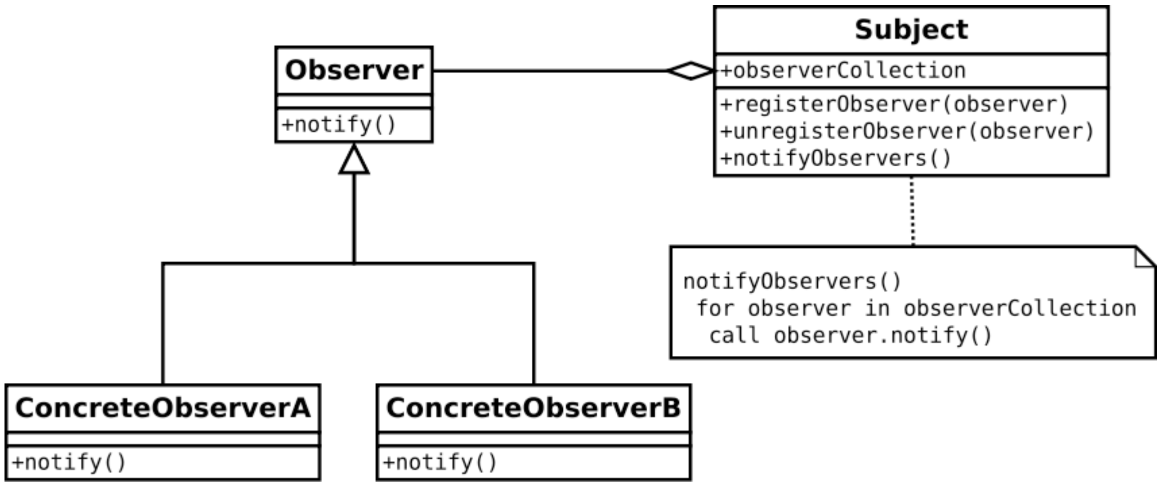
- Registering other objects which would like to receive notifications
- Notifying any changes that occur in the main object to the registered objects (via registered object's method)
- Unregistering objects that do not want to receive any further notifications

2. Subscriber Class that would contain:

- A method that is used by the Publisher Class, to notify the objects registered with it, of any change that occurs.

3. An event that triggers a state change that leads the Publisher to call its notification method.

To summarize, Subscriber objects can register and unregister with the Publisher object. So whenever an event, that drives the Publisher's notification method, occurs, the Publisher notifies the Subscriber objects. The notifications would only be passed to the objects that are registered with the Subject at the time of occurrence of the event.



```
notifyObservers()
 for observer in observerCollection
  call observer.notify()
```

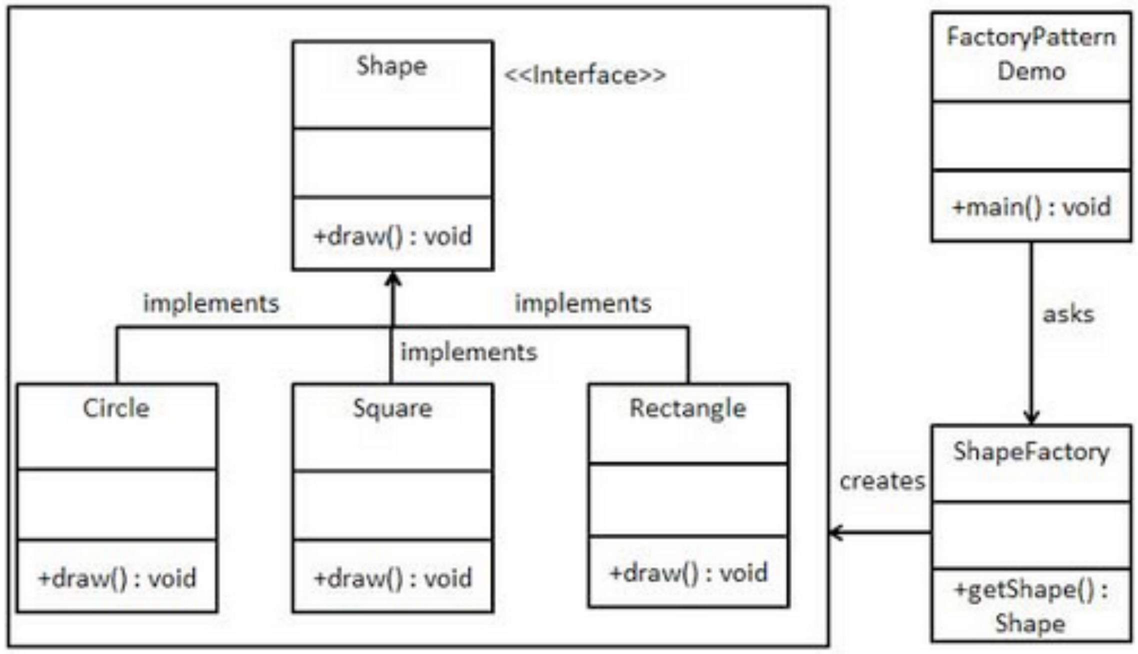
#Factory design pattern

The factory pattern is a creational design pattern used in software development to encapsulate the processes involved in the creation of objects.

Factory pattern is used in cases when based on a type got as an input at run- time; the corresponding object has to be created. In such situations, implementing code based on Factory pattern can result in scalable and maintainable code i.e. to add a new type, one need not modify existing classes; it involves just addition of new subclasses that correspond to this new type.

In short, use Factory pattern when:

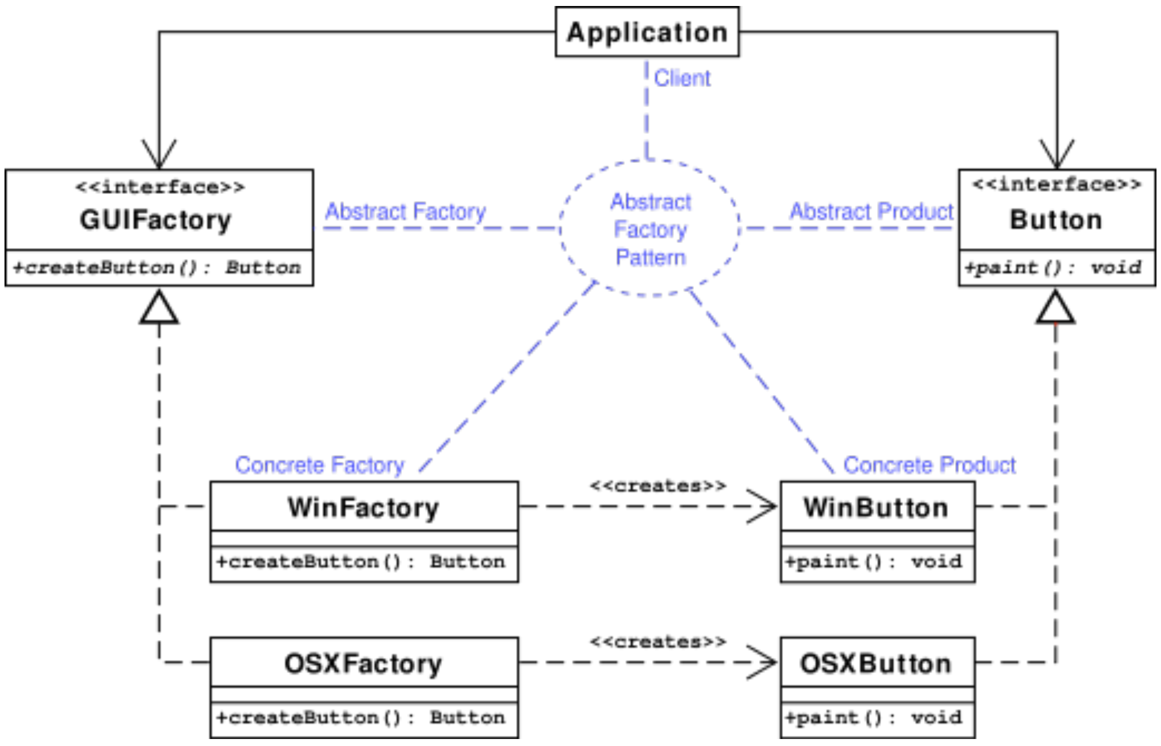
- A class does not know what kind of object it must create on a user's request
- You want to build an extensible association between this creator class and classes corresponding to objects that it is supposed to create.



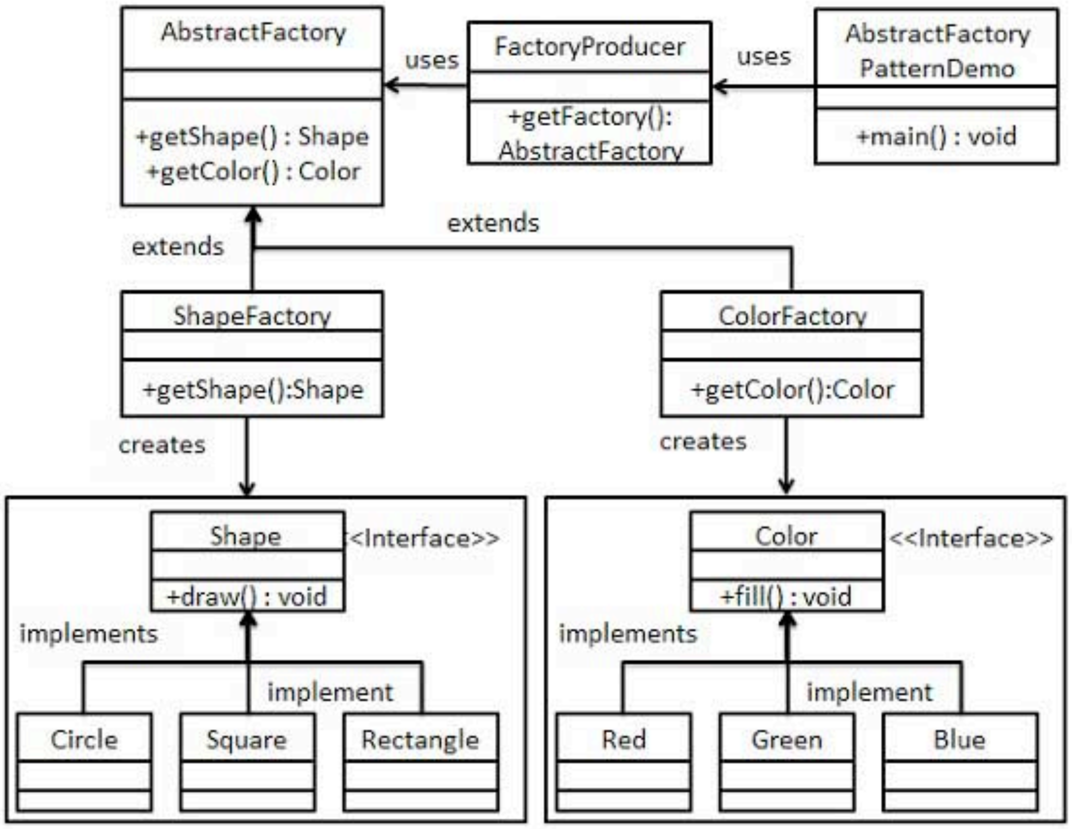
#Abstract Factory design pattern

Abstract Factory patterns work around a super-factory, which creates other factories. This factory is also called as factory of factories. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

In Abstract Factory pattern an interface is responsible for creating a factory of related objects without explicitly specifying their classes. Each generated factory can give the objects as per the Factory pattern.



A Java example,

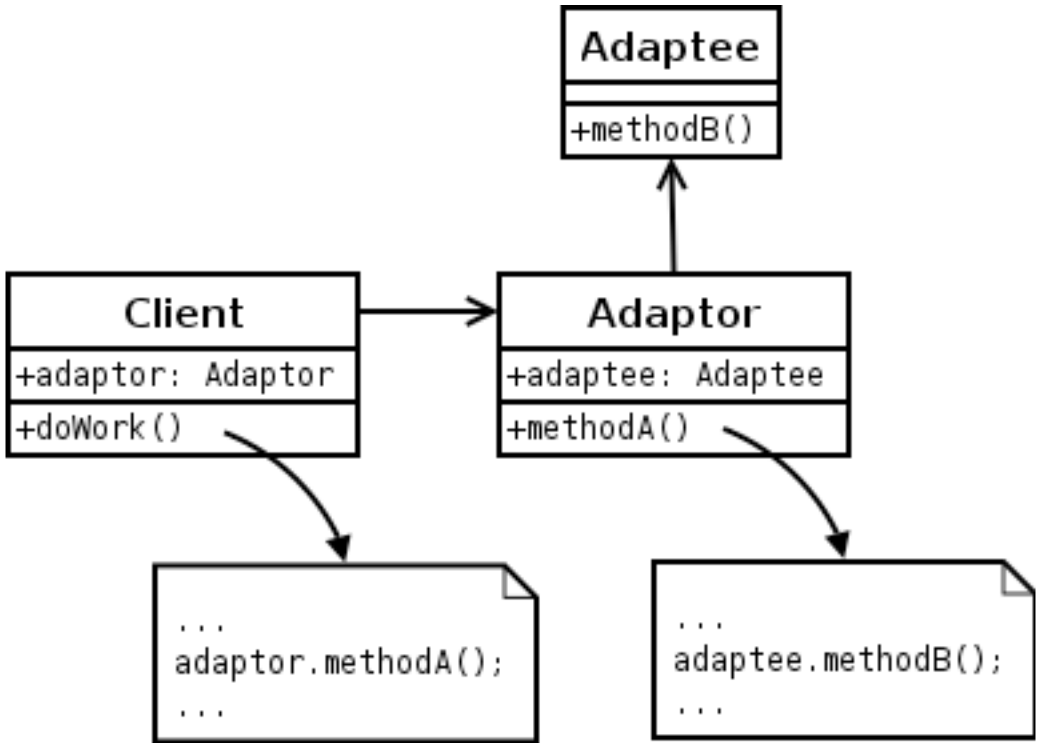

AbstractFactory
+getShape() : Shape
+getColor() : Color
uses
FactoryProducer
+getFactory(): AbstractFactory
uses
AbstractFactory PatternDemo
+main() : void
extends
extends
ShapeFactory
+getShape():Shape
ColorFactory
+getColor():Color
creates
creates
Shape
<<Interface>>
+draw() : void
Color
<<Interface>>
+fill() : void
implements
implement
implements
implement
Circle
Square
Rectangle
Red
Green
Blue

#Adapter design pattern

The adapter pattern is a software design pattern that 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.

Adapter pattern works as a bridge between two incompatible interfaces. This type of design pattern comes under structural pattern as this pattern combines the capability of two independent interfaces.

This pattern involves a single class, which is responsible to join functionalities of independent or incompatible interfaces. A real life example could be a case of card reader, which acts as an adapter between memory card and a laptop. You plugin the memory card into card reader and card reader into the laptop so that memory card can be read via laptop.

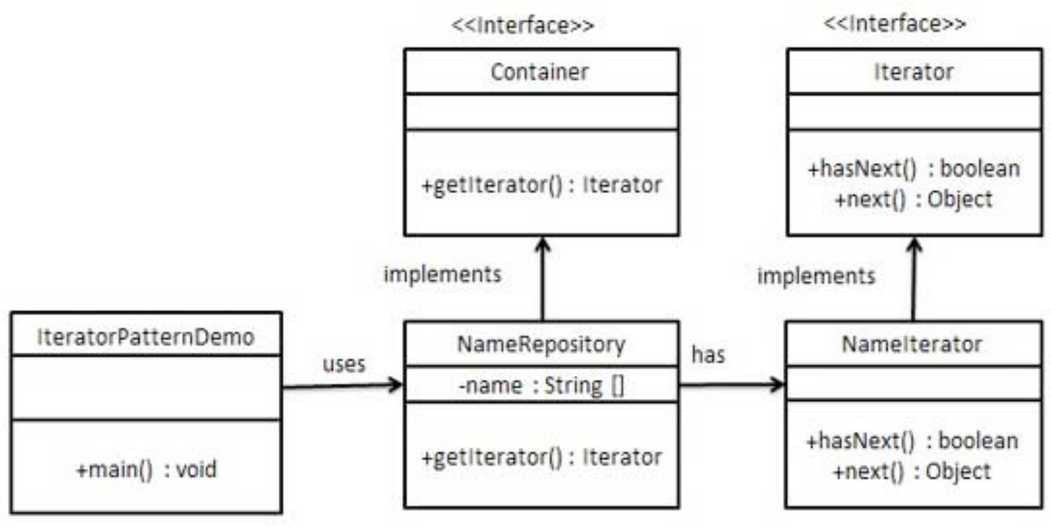


#Iterator design pattern

The essence of the Iterator Factory method Pattern is to "Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation."

Iterator pattern falls under behavioral pattern category.

Java example:

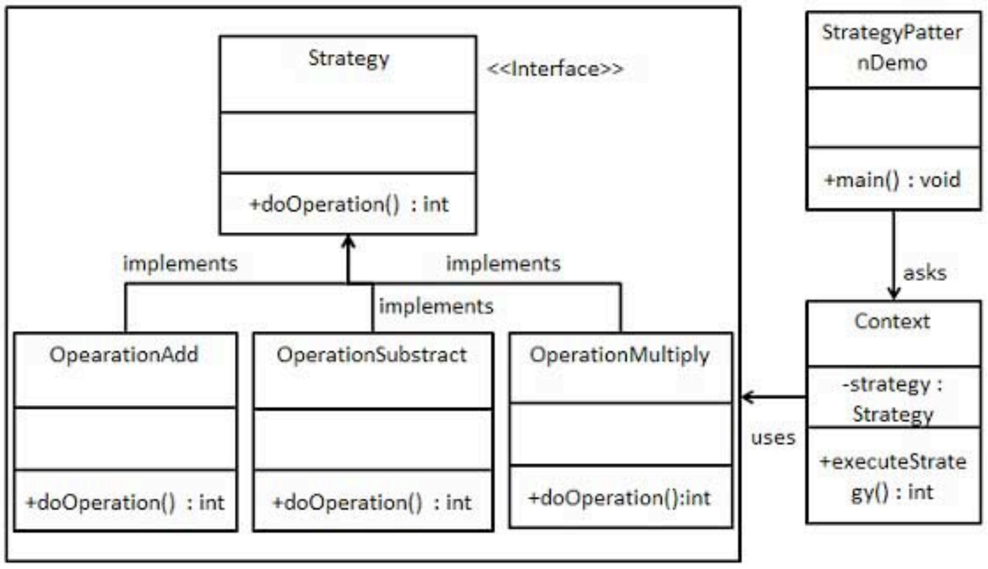


#Strategy design pattern

In Strategy pattern, a class behavior or its algorithm can be changed at run time. This type of design pattern comes under behavior pattern.

In Strategy pattern, we create objects, which represent various strategies, and a context object whose behavior varies as per its strategy object. The strategy object changes the executing algorithm of the context object.

For instance, a class that performs validation on incoming data may use a strategy pattern to select a validation algorithm based on the type of data, the source of the data, user choice, or other discriminating factors. These factors are not known for each case until run-time, and may require radically different validation to be performed. The validation strategies, encapsulated separately from the validating object, may be used by other validating objects in different areas of the system (or even different systems) without code duplication.


Strategy
<<Interface>>
+doOperation() : int
implements
implements
implements
OpearationAdd
+doOperation() : int
OperationSubstract
+doOperation() : int
OperationMultiply
+doOperation():int
StrategyPatternDemo
+main() : void
asks
Context
-strategy : Strategy
+executeStrategy() : int
uses