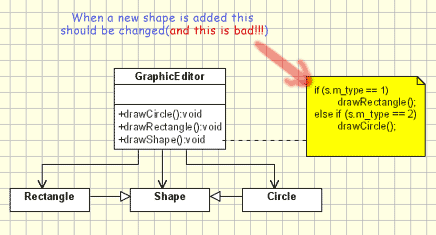
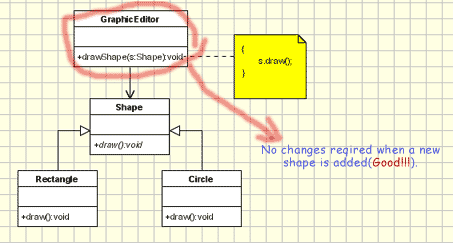
**Design principles and patterns**

1. **Design principles** 
   1. **SOLID**
      1. Single Responsibility (SRP) - A class should have only one reason to change (responsibilities), minimal impact of change. It represents a way of identifying classes during the design phase of an application. This is possible when complete picture of how should an application work is known.
      2. Open-Closed (OCP) - Software entities (classes, modules and functions) should be **open for extension** but **closed for modifications.** Many design patterns that help us to extend code without changing it e.g. Strategy, Template, Decorator, Factory Method, Observer.

OCP violation



OCP in action



* + 1. Liskov's substitution (LSP) - If a program module is using a Base class, then the reference to the Base class can be replaced with a Derived class without affecting the functionality of the program module. **LSP** is just an extension of the **OCP** and it means that we must make sure that new derived classes are extending the base classes without changing their behaviour.
    2. Interface Segregation (ISP) - The **ISP** states that clients should not be forced to implement interfaces they don't use. Instead of one fat interface many small interfaces are preferred based on groups of methods, each one serving one sub module. It may lead to increasing the complexity of code by having lot of interfaces with single methods, so applying should be done based on experience and common sense in identifying the areas where extension of code are more likely to happens in the future.

*If the design is already done fat interfaces can be segregated using the Adapter pattern.*

* + 1. Dependency inversion (DIP) - High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstraction should not depend on details. Details should depend on abstractions.
       1. High Level Classes --> Abstraction Layer --> Low Level Classes.
       2. When **DIP** is applied it means the high level classes are not working directly with low level classes, they are using interfaces as an abstract layer. In this case instantiation of new low level objects inside the high level classes(if necessary) cannot be done using the operator new. Instead, some of the Creational design patterns can be used, such as Factory Method, Abstract Factory, Prototype, IOC, Template
       3. E.g.

//Dependency Inversion Principle - Good example

**interface** Worker {

**public** **void** work();

}

**class** BaseWorker **implements** Worker{

**public** **void** work() {

// ....working

}

}

**class** SuperWorker **implements** Worker{

**public** **void** work() {

//.... working much more

}

}

**class** Manager {

Worker worker;

**public** **void** setWorker(Worker w) {

worker = w;

}

**public** **void** manage() {

worker.work();

}

}

* 1. **KISS**(keep it simple, stupid)
  2. **DRY** (Don't repeat yourself)
  3. **YAGNI** (you aren't gonna need it)

1. **Design patterns**
   1. **Creational**
      1. **Singleton** - Only instance of a class is created which could be accessed using global point of access.
         1. Sometimes it's important to have only one instance for a class. For example, in a system there should be only one window manager (or only a file system or only a print spooler)
         2. Implementation - The implementation involves a static member in the "Singleton" class, a private constructor and a static public method that returns a reference to the static member. e.g. Logger, Configuration, Shared Resource Access, Factories,
         3. How to do - Lazy initialization using double locking mechanism

**class** Singleton

{

**private** **static** Singleton *instance*;

**private** Singleton(){

System.***out***.println("Initializing Instance");

}

**public** **static** Singleton getInstance(){

**if** (*instance* == **null**){

**synchronized**(Singleton.**class**){

**if** (*instance* == **null**){

System.***out***.println("First time invoked!");

*instance* = **new** Singleton();

}

}

}

**return** *instance*;

}

**public** **void** doSomething(){

System.***out***.println("Singleton does something!");

}

}

* + - 1. Important -
         1. Multiple singleton instances

If classes loaded by different class loaders access singleton. If a class(same name, same package) is loaded by 2 different class loaders they represents 2 different classes in memory

* + - * 1. Abstract Factory and Factory Methods implemented as singletons

There are certain situations when the a factory should be unique. Having 2 factories might have undesired effects when objects are created. To ensure that a factory is unique it should be implemented as a singleton.

* + - * 1. Thread safe implementation
        2. Serialization

When Singletons are implementing Serializable interface they have to implement readResolve() method in order to avoid having 2 different objects.

E.g.

**public** **class** Singleton **implements** Serializable {

...

// This method is called immediately after an object of this class is deserialized.

// This method returns the singleton instance.

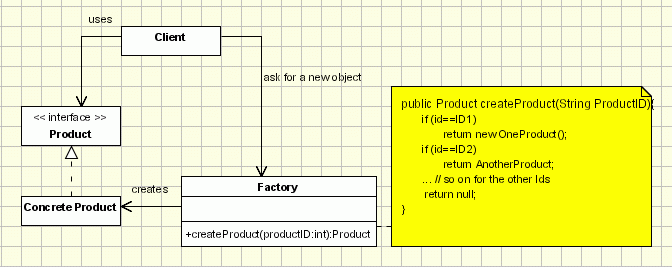
**protected** Object readResolve() {

**return** getInstance();

}

}

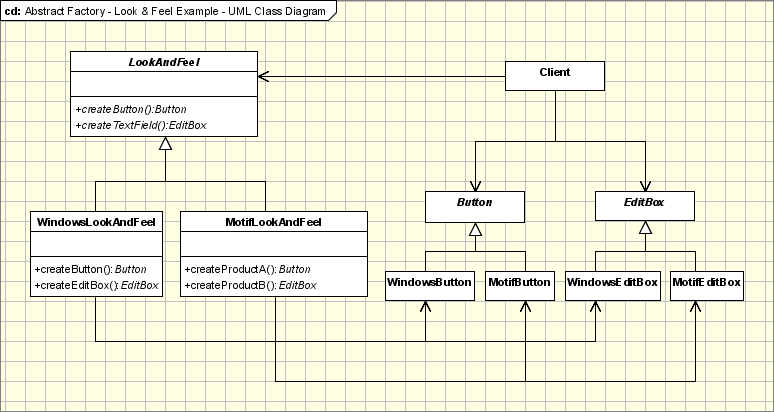
* + 1. **Factory** - The Factory Design Pattern is probably the most used design pattern in modern programming languages. Creates objects without exposing the instantiation logic to the client. Refers to the newly created object through a common interface.
       1. Implementation - The implementation is really simple



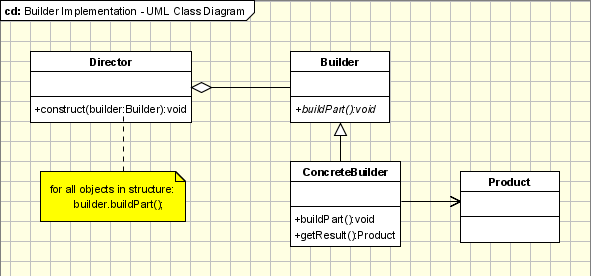
* + - * 1. The client needs a product, but instead of creating it directly using the new operator, it asks the factory object for a new product, providing the information about the type of object it needs.
        2. The factory instantiates a new concrete product and then returns to the client the newly created product(casted to abstract product class).
        3. The client uses the products as abstract products without being aware about their concrete implementation.
      1. There are disadvantages/limitations of Factory pattern
         1. The factory cannot be used as a singleton.
         2. Each factory has to be initialized before using it.
         3. More difficult to implement.
         4. If a new object has to be added a new factory has to be created.
      2. Conclusion
         1. When you design an application just think if you really need it a factory to create objects.
         2. If you decided to go for a factory, I would recommend using one of class registration implementations(with or without reflection) and to avoid the Factory Method (Factory design pattern with abstractions). Please note the procedural switch-case (noob) implementation is the simplest, violates the OCP principle is used only to explain the theory.
    1. **Factory Method** - It defines an interface for creating an object, but leaves the choice of its type to the subclasses, creation being deferred at run-time.
       1. Defines an interface for creating objects, but let subclasses to decide which class to instantiate
       2. Refers to the newly created object through a common interface
       3. Real life example -
          1. A simple real life example of the Factory Method is the hotel. When staying in a hotel you first have to check in. The person working at the front desk will give you a key to your room after you've paid for the room you want and this way he can be looked at as a "room" factory. While staying at the hotel, you might need to make a phone call, so you call the front desk and the person there will connect you with the number you need, becoming a "phone-call" factory, because he controls the access to calls, too.
       4. Implementation -



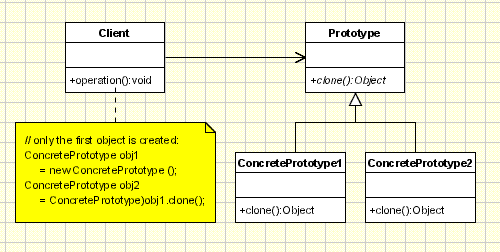
* + - * 1. Product:- defines the interface for objects the factory method creates.
        2. ConcreteProduct:- implements the Product interface.
        3. Creator:- (also refered as Factory because it creates the Product objects) declares the method FactoryMethod, which returns a Product object. May call the generating method for creating Product objects
        4. ConcreteCreator:- overrides the generating method for creating ConcreteProduct objects
      1. Advantages -
         1. The main reason for which the factory pattern is used is that it introduces a separation between the application and a family of classes (it introduces weak coupling instead of tight coupling hiding concrete classes from the application). It provides a simple way of extending the family of products with minor changes in application code.
         2. It provides customization hooks. When the objects are created directly inside the class it's hard to replace them by objects which extend their functionality. If a factory is used instead to create a family of objects the customized objects can easily replace the original objects, configuring the factory to create them.
      2. Drawbacks-
         1. The factory has to be used for a family of objects. If the classes doesn't extend common base class or interface they cannot be used in a factory design template.
      3. Java APIs
         1. [java.util.Calendar#getInstance()](http://docs.oracle.com/javase/8/docs/api/java/util/Calendar.html#getInstance--)
         2. [java.util.ResourceBundle#getBundle()](http://docs.oracle.com/javase/8/docs/api/java/util/ResourceBundle.html#getBundle-java.lang.String-)
         3. [java.text.NumberFormat#getInstance()](http://docs.oracle.com/javase/8/docs/api/java/text/NumberFormat.html#getInstance--)
         4. [java.nio.charset.Charset#forName()](http://docs.oracle.com/javase/8/docs/api/java/nio/charset/Charset.html#forName-java.lang.String-)
         5. [java.net.URLStreamHandlerFactory#createURLStreamHandler(String)](http://docs.oracle.com/javase/8/docs/api/java/net/URLStreamHandlerFactory.html) (Returns singleton object per protocol)
         6. [java.util.EnumSet#of()](https://docs.oracle.com/javase/8/docs/api/java/util/EnumSet.html#of(E))
         7. [javax.xml.bind.JAXBContext#createMarshaller()](https://docs.oracle.com/javase/8/docs/api/javax/xml/bind/JAXBContext.html#createMarshaller--) and other similar methods
    1. **Abstract Factory** - Abstract Factory is a super-factory which creates other factories (Factory of factories). Abstract Factory offers the interface for creating a family of related objects, without explicitly specifying their classes. It is recognizable by creational methods returning the factory itself which in turn can be used to create another abstract/interface type.
       1. Implementation



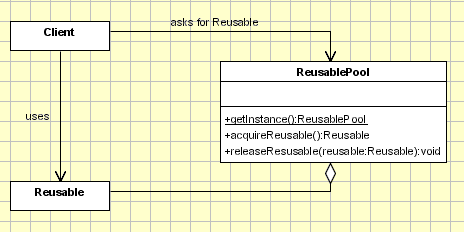
* + - * 1. AbstractFactory - declares a interface for operations that create abstract products.
        2. ConcreteFactory - implements operations to create concrete products.
        3. AbstractProduct - declares an interface for a type of product objects.
        4. Product - defines a product to be created by the corresponding ConcreteFactory; it implements the AbstractProduct interface.
        5. Client - uses the interfaces declared by the AbstractFactory and AbstractProduct classes.
      1. Examples -
         1. the system needs to be independent from the way the products it works with are created.
         2. the system is or should be configured to work with multiple families of products.
         3. a family of products is designed to work only all together.
         4. the creation of a library of products is needed, for which is relevant only the interface, not the implementation
      2. Advantages -
         1. On one hand it isolates the creation of objects from the client that needs them, giving the client only the possibility of accessing them through an interface, which makes the manipulation easier.
         2. The exchanging of product families is easier, as the class of a concrete factory appears in the code only where it is instantiated. Also if the products of a family are meant to work together, the Abstract Factory makes it easy to use the objects from only one family at a time.
      3. Drawbacks -
         1. Adding new products to the existing factories is difficult, because the Abstract Factory interface uses a fixed set of products that can be created. That is why adding a new product would mean extending the factory interface, which involves changes in the AbstractFactory class and all its subclasses.
      4. Implementation in Java APIs
         1. [javax.xml.parsers.DocumentBuilderFactory#newInstance()](http://docs.oracle.com/javase/8/docs/api/javax/xml/parsers/DocumentBuilderFactory.html#newInstance--)
         2. [javax.xml.transform.TransformerFactory#newInstance()](http://docs.oracle.com/javase/8/docs/api/javax/xml/transform/TransformerFactory.html#newInstance--)
         3. [javax.xml.xpath.XPathFactory#newInstance()](http://docs.oracle.com/javase/8/docs/api/javax/xml/xpath/XPathFactory.html#newInstance--)
    1. **Builder** - This pattern allows client object to construct a complex object by specifying only its type and content, being shielded from the details related to the object's representation
       1. Implementation - The Builder design pattern uses the Factory Builder pattern to decide which concrete class to initiate in order to build the desired type of object.



* + - * 1. The **Builder** class specifies an abstract interface for creating parts of a Product object.
        2. The **ConcreteBuilder** constructs and puts together parts of the product by implementing the Builder interface. It defines and keeps track of the representation it creates and provides an interface for saving the product.
        3. The **Director** class constructs the complex object using the Builder interface.
        4. The **Product** represents the complex object that is being built.
      1. How is it different than Abstract Factory?
         1. It is very similar to Abstract Factory pattern but there is distinction, In the case of the Abstract Factory, the client uses the factory's methods to create its own objects. In the Builder's case, the Builder class is instructed on how to create the object and then it is asked for it, but the way that the class is put together is up to the Builder class
         2. Also abstract factory creates objects derived from a common type while products created by the concrete builders have a structure significantly different.
      2. Java APIs
         1. [java.lang.StringBuilder#append()](http://docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html#append-boolean-) (unsynchronized)
         2. [java.lang.StringBuffer#append()](http://docs.oracle.com/javase/8/docs/api/java/lang/StringBuffer.html#append-boolean-) (synchronized)
         3. [java.nio.ByteBuffer#put()](http://docs.oracle.com/javase/8/docs/api/java/nio/ByteBuffer.html#put-byte-) (also on [CharBuffer](http://docs.oracle.com/javase/8/docs/api/java/nio/CharBuffer.html#put-char-), [ShortBuffer](http://docs.oracle.com/javase/8/docs/api/java/nio/ShortBuffer.html#put-short-), [IntBuffer](http://docs.oracle.com/javase/8/docs/api/java/nio/IntBuffer.html#put-int-), [LongBuffer](http://docs.oracle.com/javase/8/docs/api/java/nio/LongBuffer.html#put-long-), [FloatBuffer](http://docs.oracle.com/javase/8/docs/api/java/nio/FloatBuffer.html#put-float-) and [DoubleBuffer](http://docs.oracle.com/javase/8/docs/api/java/nio/DoubleBuffer.html#put-double-))
         4. [javax.swing.GroupLayout.Group#addComponent()](http://docs.oracle.com/javase/8/docs/api/javax/swing/GroupLayout.Group.html#addComponent-java.awt.Component-)
         5. All implementations of [java.lang.Appendable](http://docs.oracle.com/javase/8/docs/api/java/lang/Appendable.html)
    1. **Prototype** - It allows an object to create customized objects without knowing their class or any details of how to create them. It sounds lot like Factory Method, the difference being the fact that for the Factory the palette of prototypical objects never contains more than one object.
       1. Create new objects by copying this prototype
       2. Implementation



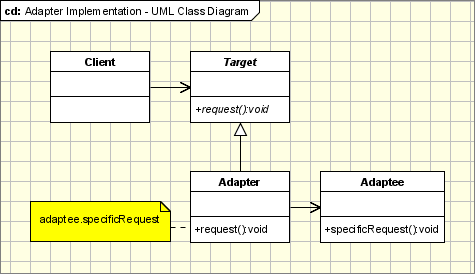
* + - * 1. **Client** - creates a new object by asking a prototype to clone itself.
        2. **Prototype** - declares an interface for cloning itself.
        3. **ConcretePrototype** - implements the operation for cloning itself.
      1. Java APIS -
         1. [java.lang.Object#clone()](http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#clone--) (the class has to implement [java.lang.Cloneable](http://docs.oracle.com/javase/8/docs/api/java/lang/Cloneable.html))
         2. [Singleton](http://en.wikipedia.org/wiki/Singleton_pattern) (recognizable by creational methods returning the same instance (usually of itself) every time)
         3. [java.lang.Runtime#getRuntime()](http://docs.oracle.com/javase/8/docs/api/java/lang/Runtime.html#getRuntime--)
         4. [java.awt.Desktop#getDesktop()](http://docs.oracle.com/javase/8/docs/api/java/awt/Desktop.html#getDesktop--)
         5. [java.lang.System#getSecurityManager()](http://docs.oracle.com/javase/8/docs/api/java/lang/System.html#getSecurityManager--)
    1. **Object pool** - Object Pool pattern offer a mechanism to reuse objects that are expensive to create
       1. Implementation



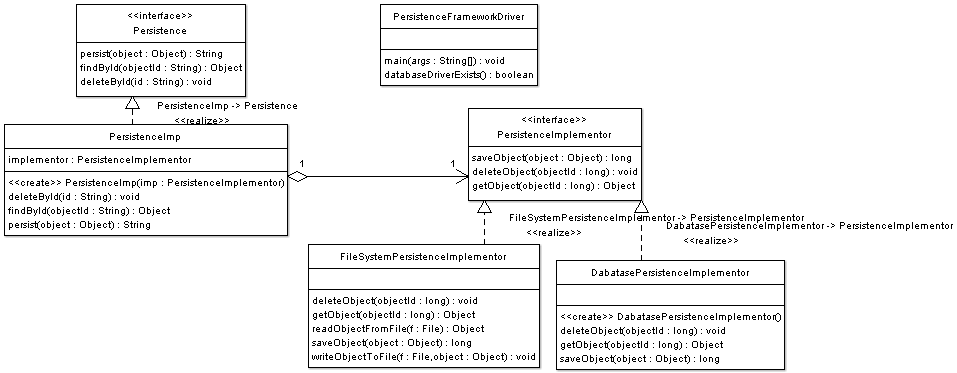
* 1. **Structural** - Structural patterns are concerned with how classes and objects are composed to form larger structures.
* [**Adapter**](https://sourcemaking.com/design_patterns/adapter) - Match interfaces of different classes
* [**Bridge**](https://sourcemaking.com/design_patterns/bridge) - Separates an object's interface from its implementation
* [**Composite**](https://sourcemaking.com/design_patterns/composite) - A tree structure of simple and composite objects
* [**Decorator**](https://sourcemaking.com/design_patterns/decorator) - Add responsibilities to objects dynamically
* [**Facade**](https://sourcemaking.com/design_patterns/facade) - A single class that represents an entire subsystem
* [**Flyweight**](https://sourcemaking.com/design_patterns/flyweight) - A fine-grained instance used for efficient sharing
* [**Proxy**](https://sourcemaking.com/design_patterns/proxy) - An object representing another object

## Rules of thumb

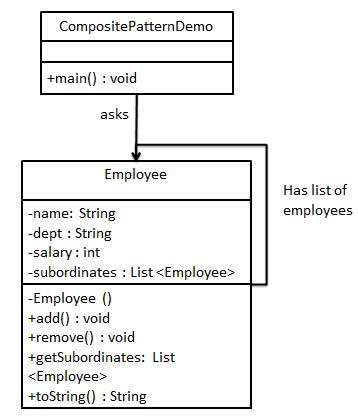
1. [**Adapter**](https://sourcemaking.com/design_patterns/adapter) makes things work after they're designed; [**Bridge**](https://sourcemaking.com/design_patterns/bridge) makes them work before they are.
2. [**Bridge**](https://sourcemaking.com/design_patterns/bridge) is designed up-front to let the abstraction and the implementation vary independently. [**Adapter**](https://sourcemaking.com/design_patterns/adapter) is retrofitted to make unrelated classes work together.
3. [**Adapter**](https://sourcemaking.com/design_patterns/adapter) provides a different interface to its subject. [**Proxy**](https://sourcemaking.com/design_patterns/proxy) provides the same interface. [**Decorator**](https://sourcemaking.com/design_patterns/decorator) provides an enhanced interface.
4. [**Adapter**](https://sourcemaking.com/design_patterns/adapter) changes an object's interface, [**Decorator**](https://sourcemaking.com/design_patterns/decorator) enhances an object's responsibilities. [**Decorator**](https://sourcemaking.com/design_patterns/decorator) is thus more transparent to the client. As a consequence, [**Decorator**](https://sourcemaking.com/design_patterns/decorator) supports recursive composition, which isn't possible with pure [**Adapters**](https://sourcemaking.com/design_patterns/adapter).
5. [**Composite**](https://sourcemaking.com/design_patterns/composite) and [**Decorator**](https://sourcemaking.com/design_patterns/decorator) have similar structure diagrams, reflecting the fact that both rely on recursive composition to organize an open-ended number of objects.
6. [**Composite**](https://sourcemaking.com/design_patterns/composite) can be traversed with [**Iterator**](https://sourcemaking.com/design_patterns/iterator). [**Visitor**](https://sourcemaking.com/design_patterns/visitor) can apply an operation over a [**Composite**](https://sourcemaking.com/design_patterns/composite). [**Composite**](https://sourcemaking.com/design_patterns/composite) could use [**Chain of responsibility**](https://sourcemaking.com/design_patterns/chain_of_responsibility) to let components access global properties through their parent. It could also use [**Decorator**](https://sourcemaking.com/design_patterns/decorator) to override these properties on parts of the composition. It could use [**Observer**](https://sourcemaking.com/design_patterns/observer) to tie one object structure to another and [**State**](https://sourcemaking.com/design_patterns/state) to let a component change its behaviour as its state changes.
7. [**Composite**](https://sourcemaking.com/design_patterns/composite) can let you compose a [**Mediator**](https://sourcemaking.com/design_patterns/mediator) out of smaller pieces through recursive composition.
8. [**Decorator**](https://sourcemaking.com/design_patterns/decorator) lets you change the skin of an object. [**Strategy**](https://sourcemaking.com/design_patterns/strategy) lets you change the guts.
9. [**Decorator**](https://sourcemaking.com/design_patterns/decorator) is designed to let you add responsibilities to objects without subclassing. [**Composite**](https://sourcemaking.com/design_patterns/composite)'s focus is not on embellishment but on representation. These intents are distinct but complementary. Consequently, [**Composite**](https://sourcemaking.com/design_patterns/composite) and [**Decorator**](https://sourcemaking.com/design_patterns/decorator) are often used in concert.
10. [**Decorator**](https://sourcemaking.com/design_patterns/decorator) and [**Proxy**](https://sourcemaking.com/design_patterns/proxy) have different purposes but similar structures. Both describe how to provide a level of indirection to another object, and the implementations keep a reference to the object to which they forward requests.
11. [**Facade**](https://sourcemaking.com/design_patterns/facade) defines a new interface, whereas [**Adapter**](https://sourcemaking.com/design_patterns/adapter) reuses an old interface. Remember that [**Adapter**](https://sourcemaking.com/design_patterns/adapter) makes two existing interfaces work together as opposed to defining an entirely new one.
12. [**Facade**](https://sourcemaking.com/design_patterns/facade) objects are often [**Singleton**](https://sourcemaking.com/design_patterns/singleton) because only one [**Facade**](https://sourcemaking.com/design_patterns/facade) object is required.
13. [**Mediator**](https://sourcemaking.com/design_patterns/mediator) is similar to [**Facade**](https://sourcemaking.com/design_patterns/facade) in that it abstracts functionality of existing classes. [**Mediator**](https://sourcemaking.com/design_patterns/mediator) abstracts/centralizes arbitrary communication between colleague objects, it routinely "adds value", and it is known/referenced by the colleague objects. In contrast, [**Facade**](https://sourcemaking.com/design_patterns/facade) defines a simpler interface to a subsystem, it doesn't add new functionality, and it is not known by the subsystem classes.
14. [**Abstract Factory**](https://sourcemaking.com/design_patterns/abstract_factory) can be used as an alternative to [**Facade**](https://sourcemaking.com/design_patterns/facade) to hide platform-specific classes.
15. Whereas [**Flyweight**](https://sourcemaking.com/design_patterns/flyweight) shows how to make lots of little objects, [**Facade**](https://sourcemaking.com/design_patterns/facade) shows how to make a single object represent an entire subsystem.
16. [**Flyweight**](https://sourcemaking.com/design_patterns/flyweight) is often combined with [**Composite**](https://sourcemaking.com/design_patterns/composite) to implement shared leaf nodes.
17. [**Flyweight**](https://sourcemaking.com/design_patterns/flyweight) explains when and how [**State**](https://sourcemaking.com/design_patterns/state) objects can be shared.
    * 1. **Adapter** - The adapter pattern is adapting between classes and objects. Like any adapter in the real world it is used to be an interface, a bridge between two objects.
         1. Adapter lets classes work together, that could not otherwise because of incompatible interfaces
         2. Implementation



* + - * 1. **Target** - defines the domain-specific interface that Client uses.
        2. **Adapter** - adapts the interface Adaptee to the Target interface.
        3. **Adaptee** - defines an existing interface that needs adapting.
        4. **Client** - collaborates with objects conforming to the Target interface.
      1. Objects Adapters - Based on Delegation
         1. Objects Adapters are the classical example of the adapter pattern. It uses composition, the Adaptee delegates the calls to Adaptee (opposed to class adapters which extends the Adaptee).
      2. How Much the Adapter Should Do?
         1. it should do how much it has to in order to adapt. It's very simple, if the Target and Adaptee are similar then the adapter has just to delegate the requests from the Target to the Adaptee. If Target and Adaptee are not similar, then the adapter might have to convert the data structures between those and to implement the operations required by the Target but not implemented by the Adaptee.
      3. Two-Ways Adapters
         1. The Two-Ways Adapters are adapters that implements both interfaces of Target and Adaptee.
      4. When to use -
         1. Adapter makes things work after they're designed while Bridge makes them work before upfront (while designing)
         2. Adapter is retrofitted to make unrelated classes work together
         3. Adapter reuses an old interface
      5. Adapter Pattern and Strategy Pattern
         1. Adapter Pattern and Strategy Pattern - there are many cases when the adapter can play the role of the Strategy Pattern. If we have several modules implementing the same functionality and we wrote adapters for them, the adapters are implementing the same interface. We can simply replace the adapters objects at run time because they implements the same interface.
      6. Use in Java API
         1. [java.util.Arrays#asList()](http://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#asList-T...-)
         2. [java.util.Collections#list()](https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#list-java.util.Enumeration-)
         3. [java.util.Collections#enumeration()](https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#enumeration-java.util.Collection-)
         4. [java.io.InputStreamReader(InputStream)](http://docs.oracle.com/javase/8/docs/api/java/io/InputStreamReader.html#InputStreamReader-java.io.InputStream-) (returns a Reader)
         5. [java.io.OutputStreamWriter(OutputStream)](http://docs.oracle.com/javase/8/docs/api/java/io/OutputStreamWriter.html#OutputStreamWriter-java.io.OutputStream-) (returns a Writer)
         6. [javax.xml.bind.annotation.adapters.XmlAdapter#marshal()](http://docs.oracle.com/javase/8/docs/api/javax/xml/bind/annotation/adapters/XmlAdapter.html#marshal-BoundType-) and [#unmarshal()](http://docs.oracle.com/javase/8/docs/api/javax/xml/bind/annotation/adapters/XmlAdapter.html#unmarshal-ValueType-)
    1. **Bridge** - The intent of this pattern is to decouple abstraction from implementation so that the two can vary independently
       1. The bridge pattern applies when there is a need to avoid permanent binding between an abstraction and an implementation and when the abstraction and implementation need to vary independently. Using the bridge pattern would leave the client code unchanged with no need to recompile the code.
       2. Implementation - Object Persistence API



* + - 1. Related Patterns
         1. **Abstract Factory Pattern** - An Abstract Factory pattern can be used create and configure a particular Bridge, for example a factory can choose the suitable concrete implementor at runtime.
      2. Use in Java API
         1. None comes to mind yet. A fictive example would be new LinkedHashMap(LinkedHashSet<K>, List<V>) which returns an unmodifiable linked map which doesn't clone the items, but *uses* them. The  [java.util.Collections#newSetFromMap()](http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#newSetFromMap-java.util.Map-) and [singletonXXX()](http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#singleton-T-) methods however comes close.
    1. **Composite** - Composite pattern is used where we need to treat a group of objects in similar way as a single object. Composite pattern composes objects in term of a tree structure to represent part as well as whole hierarchy. This pattern creates a class that contains group of its own objects. This class provides ways to modify its group of same objects.
       1. When to use
          1. Recursive composition
          2. "Directories contain entries, each of which could be a directory."
          3. 1-to-many "has a" up the "is a" hierarchy
       2. Implementation



* + - * 1. Employee which acts as composite pattern actor class.
        2. CompositePatternDemo demo class will use Employee class to add department level hierarchy
        3. Another example

// A Java program to demonstrate working of

// Composite Design Pattern with example

// of a company with different

//  employee details

import java.util.ArrayList;

import java.util.List;

// A common interface for all employee

interface Employee{

    public void showEmployeeDetails();

}

class Developer implements Employee{

    private String name;

    private long empId;

    private String position;

    public Developer(long empId, String name, String position){

        // Assign the Employee id,

        // name and the position

        this.empId = empId;

        this.name = name;

        this.position = position;

    }

    @Override

    public void showEmployeeDetails()

    {

        System.out.println(empId+" " +name+ " " + position );

    }

}

class Manager implements Employee{

    private String name;

    private long empId;

    private String position;

    public Manager(long empId, String name, String position){

        this.empId = empId;

        this.name = name;

        this.position = position;

    }

    @Override

    public void showEmployeeDetails(){

        System.out.println(empId+" " +name+ " " + position );

    }

}

// Class used to get Employee List

// and do the opertions like

// add or remove Employee

class CompanyDirectory implements Employee{

    private List<Employee> employeeList = new ArrayList<Employee>();

    @Override

    public void showEmployeeDetails(){

        for(Employee emp:employeeList){

            emp.showEmployeeDetails();

        }

    }

    public void addEmployee(Employee emp){

        employeeList.add(emp);

    }

    public void removeEmployee(Employee emp){

        employeeList.remove(emp);

    }

}

// Driver class

public class Company{

    public static void main (String[] args){

        Developer dev1 = new Developer(100, "Lokesh Sharma", "Pro Developer");

        Developer dev2 = new Developer(101, "Vinay Sharma", "Developer");

        CompanyDirectory engDirectory = new CompanyDirectory();

        engDirectory.addEmployee(dev1);

        engDirectory.addEmployee(dev2);

        Manager man1 = new Manager(200, "Kushagra Garg", "SEO Manager");

        Manager man2 = new Manager(201, "Vikram Sharma ", "Kushagra's Manager");

        CompanyDirectory accDirectory = new CompanyDirectory();

        accDirectory.addEmployee(man1);

        accDirectory.addEmployee(man2);

        CompanyDirectory directory = new CompanyDirectory();

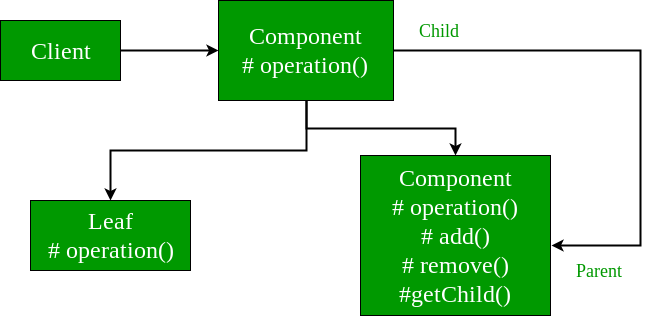
        directory.addEmployee(engDirectory);

        directory.addEmployee(accDirectory);

        directory.showEmployeeDetails();

    }

}

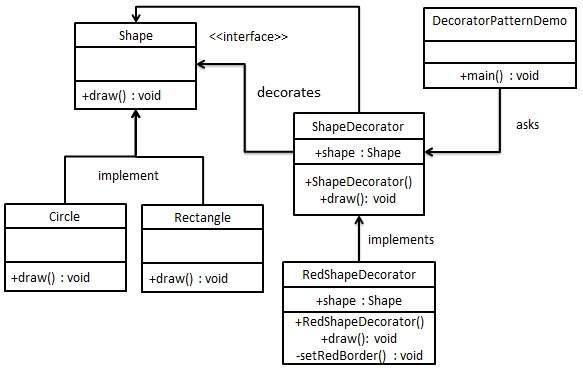


* + - * 1. Java APIs

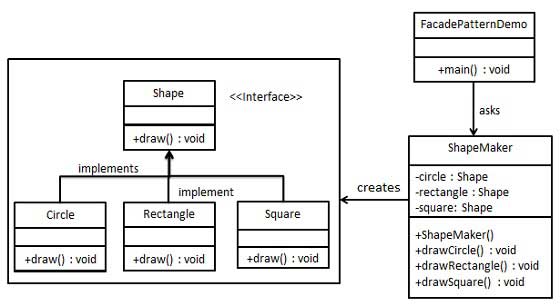
[java.awt.Container#add(Component)](http://docs.oracle.com/javase/8/docs/api/java/awt/Container.html#add-java.awt.Component-) (practically all over Swing thus)

[javax.faces.component.UIComponent#getChildren()](http://docs.oracle.com/javaee/7/api/javax/faces/component/UIComponent.html#getChildren--) (practically all over JSF UI thus)

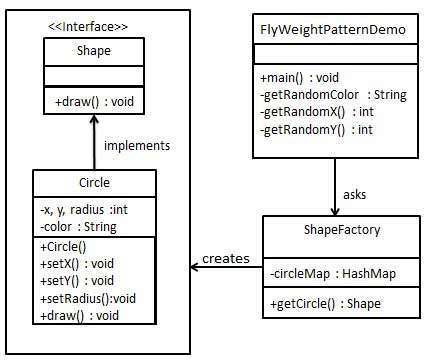
* + 1. **Decorator** - Decorator pattern allows a user to add new functionality to an existing object without altering its structure.
       1. This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.
       2. Decorator do implement same interface as that of original class
       3. Implementation -



* + - 1. Decorator enhances an object's responsibilities
      2. Decorator can be viewed as a degenerate Composite with only one component. However, a Decorator adds additional responsibilities - it isn't intended for object aggregation.
      3. Decorator is designed to let you add responsibilities to objects without subclassing.
      4. Java APIs
         1. All subclasses of [java.io.InputStream](http://docs.oracle.com/javase/8/docs/api/java/io/InputStream.html), [OutputStream](http://docs.oracle.com/javase/8/docs/api/java/io/OutputStream.html), [Reader](http://docs.oracle.com/javase/8/docs/api/java/io/Reader.html) and [Writer](http://docs.oracle.com/javase/8/docs/api/java/io/Writer.html) have a constructor taking an instance of same type.
         2. [java.util.Collections](http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html), the [checkedXXX()](http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#checkedCollection-java.util.Collection-java.lang.Class-), [synchronizedXXX()](http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#synchronizedCollection-java.util.Collection-) and [unmodifiableXXX()](http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#unmodifiableCollection-java.util.Collection-)methods.
         3. [javax.servlet.http.HttpServletRequestWrapper](http://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpServletRequestWrapper.html) and [HttpServletResponseWrapper](http://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpServletResponseWrapper.html)
    1. **Facade** - 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.
       1. Wrap a complicated subsystem with a simpler interface
       2. Implementation -



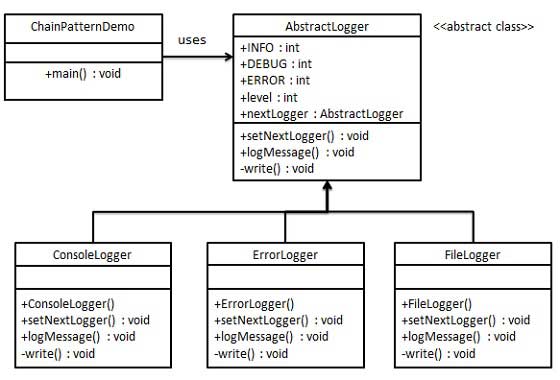
* + - 1. Facade defines a new interface, whereas Adapter uses an old interface. Adapter makes two existing interfaces work together as opposed to defining an entirely new one.
      2. Flyweight shows how to make lots of little objects, where as Facade shows how to make a single object represent an entire subsystem.
      3. 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.
      4. Abstract Factory can be used as an alternative to Facade to hide platform-specific classes.
      5. Facade objects are often Singletons because only one Facade object is required.
      6. 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.
    1. **Flyweight** - Flyweight pattern is primarily used to reduce the number of objects created and to decrease memory footprint and increase performance.
       1. Flyweight pattern tries to reuse already existing similar kind objects by storing them and creates new object when no matching object is found.
       2. Flyweights are usually created using a factory and the singleton is applied to that factory so that for each type or category of flyweights a single instance is returned.
       3. Implementation -



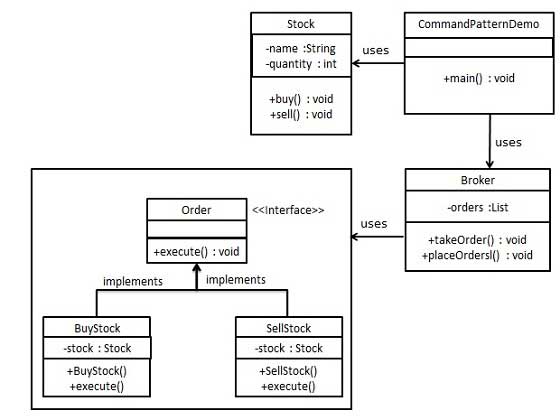
* + - 1. Java API
         1. [java.lang.Integer#valueOf(int)](http://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html#valueOf-int-) (also on [Boolean](http://docs.oracle.com/javase/8/docs/api/java/lang/Boolean.html#valueOf-boolean-), [Byte](http://docs.oracle.com/javase/8/docs/api/java/lang/Byte.html#valueOf-byte-), [Character](http://docs.oracle.com/javase/8/docs/api/java/lang/Character.html#valueOf-char-), [Short](http://docs.oracle.com/javase/8/docs/api/java/lang/Short.html#valueOf-short-), [Long](http://docs.oracle.com/javase/8/docs/api/java/lang/Long.html#valueOf-long-) and [BigDecimal](https://docs.oracle.com/javase/8/docs/api/java/math/BigDecimal.html" \l "valueOf-long-int-))
    1. **Proxy -** 
       1. Provide a surrogate or placeholder for another object to control access to it.
       2. Use an extra level of indirection to support distributed, controlled, or intelligent access.
       3. Add a wrapper and delegation to protect the real component from undue complexity.
       4. Proxy Pattern focuses on controlling access to an object
       5. Relationship between a *Proxy* and the real subject is typically set at compile time, *Proxy* instantiates it in some way
       6. Java APIs
          1. [java.lang.reflect.Proxy](http://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Proxy.html)
          2. [java.rmi.\*](http://docs.oracle.com/javase/8/docs/api/java/rmi/package-summary.html)
          3. [javax.inject.Inject](http://docs.oracle.com/javaee/7/api/javax/inject/Inject.html) ([explanation here](https://stackoverflow.com/questions/29651008/field-getobj-returns-all-nulls-on-injected-cdi-managed-beans-while-manually-i/29672591#29672591))
          4. [javax.persistence.PersistenceContext](http://docs.oracle.com/javaee/7/api/javax/persistence/PersistenceContext.html)
  1. **Behavioural** - Behavioural patterns are concerned with algorithms and the assignment of responsibilities between objects. Behavioural patterns describe not just the patterns of objects or classes but also the patterns of communication between them.
* [**Chain of responsibility**](https://sourcemaking.com/design_patterns/chain_of_responsibility) - A way of passing a request between a chain of objects
* [**Command**](https://sourcemaking.com/design_patterns/command) - Encapsulate a command request as an object
* [**Interpreter**](https://sourcemaking.com/design_patterns/interpreter) - A way to include language elements in a program
* [**Iterator**](https://sourcemaking.com/design_patterns/iterator) - Sequentially access the elements of a collection
* [**Mediator**](https://sourcemaking.com/design_patterns/mediator) - Defines simplified communication between classes
* [**Memento**](https://sourcemaking.com/design_patterns/memento) - Capture and restore an object's internal state
* [**Null Object**](https://sourcemaking.com/design_patterns/null_object) - Designed to act as a default value of an object
* [**Observer**](https://sourcemaking.com/design_patterns/observer) - A way of notifying change to a number of classes
* [**State**](https://sourcemaking.com/design_patterns/state) - Alter an object's behavior when its state changes
* [**Strategy**](https://sourcemaking.com/design_patterns/strategy) - Encapsulates an algorithm inside a class
* [**Template method**](https://sourcemaking.com/design_patterns/template_method) - Defer the exact steps of an algorithm to a subclass
* [**Visitor**](https://sourcemaking.com/design_patterns/visitor) - Defines a new operation to a class without change

## Rules of thumb

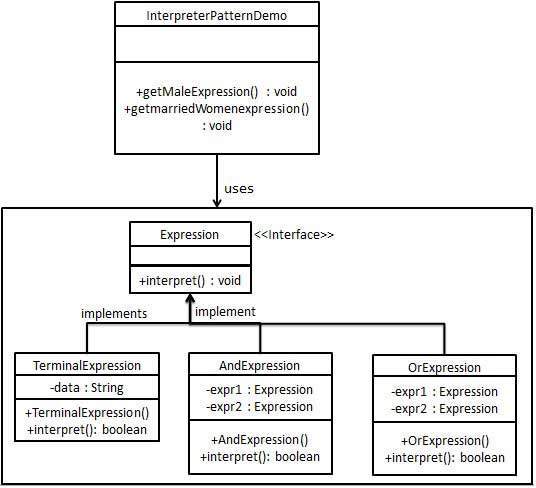
1. **Behavioural** patterns are concerned with the assignment of responsibilities between objects, or, encapsulating behaviour in an object and delegating requests to it.
2. [**Chain of responsibility**](https://sourcemaking.com/design_patterns/chain_of_responsibility), [**Command**](https://sourcemaking.com/design_patterns/command), [**Mediator**](https://sourcemaking.com/design_patterns/mediator), and [**Observer**](https://sourcemaking.com/design_patterns/observer), address how you can decouple senders and receivers, but with different trade-offs. [**Chain of responsibility**](https://sourcemaking.com/design_patterns/chain_of_responsibility) passes a sender request along a chain of potential receivers. [**Command**](https://sourcemaking.com/design_patterns/command) normally specifies a sender-receiver connection with a subclass. [**Mediator**](https://sourcemaking.com/design_patterns/mediator) has senders and receivers reference each other indirectly. [**Observer**](https://sourcemaking.com/design_patterns/observer) defines a very decoupled interface that allows for multiple receivers to be configured at run-time.
3. [**Chain of responsibility**](https://sourcemaking.com/design_patterns/chain_of_responsibility) can use [**Command**](https://sourcemaking.com/design_patterns/command) to represent requests as objects.
4. [**Chain of responsibility**](https://sourcemaking.com/design_patterns/chain_of_responsibility) is often applied in conjunction with [**Composite**](https://sourcemaking.com/design_patterns/composite). There, a component's parent can act as its successor.
5. [**Command**](https://sourcemaking.com/design_patterns/command) and [**Memento**](https://sourcemaking.com/design_patterns/memento) act as magic tokens to be passed around and invoked at a later time. In [**Command**](https://sourcemaking.com/design_patterns/command), the token represents a request; in [**Memento**](https://sourcemaking.com/design_patterns/memento), it represents the internal state of an object at a particular time. Polymorphism is important to [**Command**](https://sourcemaking.com/design_patterns/command), but not to [**Memento**](https://sourcemaking.com/design_patterns/memento) because its interface is so narrow that a memento can only be passed as a value.
6. [**Command**](https://sourcemaking.com/design_patterns/command) can use [**Memento**](https://sourcemaking.com/design_patterns/memento) to maintain the state required for an undo operation.
7. Macro[**Command**](https://sourcemaking.com/design_patterns/command)s can be implemented with [**Composite**](https://sourcemaking.com/design_patterns/composite).
8. A [**Command**](https://sourcemaking.com/design_patterns/command) that must be copied before being placed on a history list acts as a [**Prototype**](https://sourcemaking.com/design_patterns/prototype).
9. [**Interpreter**](https://sourcemaking.com/design_patterns/interpreter) can use [**State**](https://sourcemaking.com/design_patterns/state) to define parsing contexts.
10. The abstract syntax tree of [**Interpreter**](https://sourcemaking.com/design_patterns/interpreter) is a [**Composite**](https://sourcemaking.com/design_patterns/composite) (therefore **[Iterator](https://sourcemaking.com/design_patterns/iterator" \o "Used to access the elements of an aggregate object sequentially without exposing its underlying representation.)** and [**Visitor**](https://sourcemaking.com/design_patterns/visitor) are also applicable).
11. Terminal symbols within [**Interpreter**](https://sourcemaking.com/design_patterns/interpreter)'s abstract syntax tree can be shared with [**Flyweight**](https://sourcemaking.com/design_patterns/flyweight).
12. [**Iterator**](https://sourcemaking.com/design_patterns/iterator) can traverse a [**Composite**](https://sourcemaking.com/design_patterns/composite). [**Visitor**](https://sourcemaking.com/design_patterns/visitor) can apply an operation over a [**Composite**](https://sourcemaking.com/design_patterns/composite).
13. Polymorphic **[Iterator](https://sourcemaking.com/design_patterns/iterator" \o "Used to access the elements of an aggregate object sequentially without exposing its underlying representation.)**s rely on [**Factory Method**](https://sourcemaking.com/design_patterns/factory_method)s to instantiate the appropriate **[Iterator](https://sourcemaking.com/design_patterns/iterator" \o "Used to access the elements of an aggregate object sequentially without exposing its underlying representation.)**subclass.
14. [**Mediator**](https://sourcemaking.com/design_patterns/mediator) and [**Observer**](https://sourcemaking.com/design_patterns/observer) are competing patterns. The difference between them is that [**Observer**](https://sourcemaking.com/design_patterns/observer) distributes communication by introducing "observer" and "subject" objects, whereas a [**Mediator**](https://sourcemaking.com/design_patterns/mediator) object encapsulates the communication between other objects. We've found it easier to make reusable [**Observer**](https://sourcemaking.com/design_patterns/observer)s and Subjects than to make reusable [**Mediator**](https://sourcemaking.com/design_patterns/mediator)s.
15. On the other hand, [**Mediator**](https://sourcemaking.com/design_patterns/mediator) can leverage [**Observer**](https://sourcemaking.com/design_patterns/observer) for dynamically registering colleagues and communicating with them.
16. [**Mediator**](https://sourcemaking.com/design_patterns/mediator) is similar to [**Facade**](https://sourcemaking.com/design_patterns/facade) in that it abstracts functionality of existing classes. **[Mediator](https://sourcemaking.com/design_patterns/mediator" \o "Provides a unified interface to a set of interfaces in a subsystem.)**abstracts/centralizes arbitrary communication between colleague objects, it routinely "adds value", and it is known/referenced by the colleague objects (i.e. it defines a multidirectional protocol). In contrast, [**Facade**](https://sourcemaking.com/design_patterns/facade) defines a simpler interface to a subsystem, it doesn't add new functionality, and it is not known by the subsystem classes (i.e. it defines a unidirectional protocol where it makes requests of the subsystem classes but not vice versa).
17. [**Memento**](https://sourcemaking.com/design_patterns/memento) is often used in conjunction with **[Iterator](https://sourcemaking.com/design_patterns/iterator" \o "Used to access the elements of an aggregate object sequentially without exposing its underlying representation.)**. An **[Iterator](https://sourcemaking.com/design_patterns/iterator" \o "Used to access the elements of an aggregate object sequentially without exposing its underlying representation.)** can use a [**Memento**](https://sourcemaking.com/design_patterns/memento) to capture the state of an iteration. The **[Iterator](https://sourcemaking.com/design_patterns/iterator" \o "Used to access the elements of an aggregate object sequentially without exposing its underlying representation.)** stores the [**Memento**](https://sourcemaking.com/design_patterns/memento) internally.
18. [**State**](https://sourcemaking.com/design_patterns/state) is like [**Strategy**](https://sourcemaking.com/design_patterns/strategy) except in its intent.
19. [**Flyweight**](https://sourcemaking.com/design_patterns/flyweight) explains when and how [**State**](https://sourcemaking.com/design_patterns/state) objects can be shared.
20. [**State**](https://sourcemaking.com/design_patterns/state) objects are often [**Singleton**](https://sourcemaking.com/design_patterns/singleton)s.
21. [**Strategy**](https://sourcemaking.com/design_patterns/strategy) lets you change the guts of an object. [**Decorator**](https://sourcemaking.com/design_patterns/decorator) lets you change the skin.
22. [**Strategy**](https://sourcemaking.com/design_patterns/strategy) is to algorithm. as [**Builder**](https://sourcemaking.com/design_patterns/builder) is to creation.
23. [**Strategy**](https://sourcemaking.com/design_patterns/strategy) has 2 different implementations, the first is similar to [**State**](https://sourcemaking.com/design_patterns/state). The difference is in binding times ([**Strategy**](https://sourcemaking.com/design_patterns/strategy) is a bind-once pattern, whereas [**State**](https://sourcemaking.com/design_patterns/state) is more dynamic).
24. [**Strategy**](https://sourcemaking.com/design_patterns/strategy) objects often make good [**Flyweight**](https://sourcemaking.com/design_patterns/flyweight)s.
25. [**Strategy**](https://sourcemaking.com/design_patterns/strategy) is like [**Template method**](https://sourcemaking.com/design_patterns/template_method) except in its granularity.
26. [**Template method**](https://sourcemaking.com/design_patterns/template_method) uses inheritance to vary part of an algorithm. [**Strategy**](https://sourcemaking.com/design_patterns/strategy) uses delegation to vary the entire algorithm.
27. The [**Visitor**](https://sourcemaking.com/design_patterns/visitor) pattern is like a more powerful [**Command**](https://sourcemaking.com/design_patterns/command) pattern because the visitor may initiate whatever is appropriate for the kind of object it encounters.
    * 1. **Chain of Responsibility** - The chain of responsibility pattern creates a chain of receiver objects for a request. Pass the request along the chain until an object handles it.
         1. The base class maintains a "next" pointer.
         2. Each derived class implements its contribution for handling the request.
         3. If the request needs to be "passed on", then the derived class "calls back" to the base class, which delegates to the "next" pointer.
         4. Implementation -



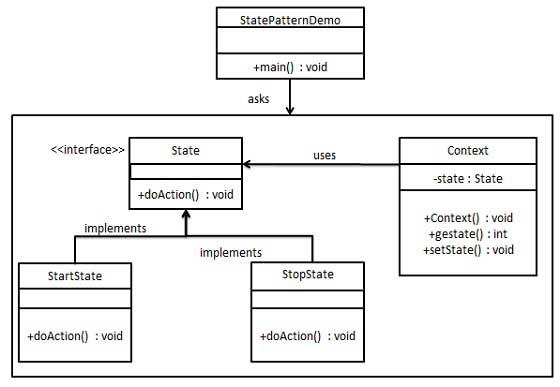
* + - 1. Java APIs
         1. [java.util.logging.Logger#log()](http://docs.oracle.com/javase/8/docs/api/java/util/logging/Logger.html#log-java.util.logging.Level-java.lang.String-)
         2. [javax.servlet.Filter#doFilter()](http://docs.oracle.com/javaee/7/api/javax/servlet/Filter.html#doFilter-javax.servlet.ServletRequest-javax.servlet.ServletResponse-javax.servlet.FilterChain-)
    1. **Command** - A request is wrapped under an object as command and passed to invoker object. Invoker object looks for the appropriate object which can handle this command and passes the command to the corresponding object which executes the command.
       1. Implementation -



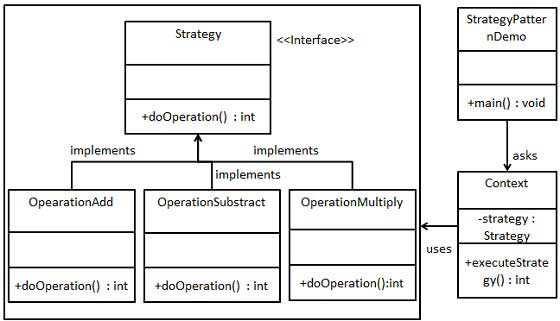
* + - * 1. Create a command interface - Order.
        2. Create a request class - Stock
        3. Create concrete classes implementing the Order interface
        4. Create command invoker class - Broker
        5. CommandPatternDemo - Use the Broker class to take and execute commands
      1. Java APIs
         1. All implementations of [java.lang.Runnable](http://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html)
         2. All implementations of [javax.swing.Action](http://docs.oracle.com/javase/8/docs/api/javax/swing/Action.html)
    1. **Interpreter** - Interpreter pattern provides a way to evaluate language grammar or expression. This pattern involves implementing an expression interface which tells to interpret a particular context. This pattern is used in SQL parsing, symbol processing engine
       1. Implementation -



* + - 1. Java APIs
         1. [java.util.Pattern](http://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html)
         2. [java.text.Normalizer](http://docs.oracle.com/javase/8/docs/api/java/text/Normalizer.html)
         3. All subclasses of [java.text.Format](http://docs.oracle.com/javase/8/docs/api/java/text/Format.html)
         4. All subclasses of [javax.el.ELResolver](http://docs.oracle.com/javaee/7/api/javax/el/ELResolver.html)
    1. Iterator
    2. Mediator
    3. Observer
    4. Visitor
    5. State - In Sate pattern a class behaviour changes based on its state.
       1. In State pattern, we create objects which represent various states and a context object whose behaviour varies as its state object changes.
       2. Implementation -



* + 1. **Strategy** -
       1. Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.
       2. Capture the abstraction in an interface, bury implementation details in derived classes.
       3. In Strategy pattern, a class behaviour or its algorithm can be changed at run time
       4. Implementation -



* + - 1. Java APIs
         1. [java.util.Comparator#compare()](http://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html#compare-T-T-), executed by among others Collections#sort().
         2. [javax.servlet.http.HttpServlet](http://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpServlet.html), the service() and all doXXX() methods take HttpServletRequest and HttpServletResponse and the implementer has to process them (and not to get hold of them as instance variables!).
         3. [javax.servlet.Filter#doFilter()](http://docs.oracle.com/javaee/7/api/javax/servlet/Filter.html#doFilter-javax.servlet.ServletRequest-javax.servlet.ServletResponse-javax.servlet.FilterChain-)
    1. **Template method** -
       1. Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
       2. Create an abstract class with a template method being final
       3. Implementation -



* + - 1. Java APIs
         1. All non-abstract methods of [java.io.InputStream](http://docs.oracle.com/javase/8/docs/api/java/io/InputStream.html), [java.io.OutputStream](http://docs.oracle.com/javase/8/docs/api/java/io/OutputStream.html), [java.io.Reader](http://docs.oracle.com/javase/8/docs/api/java/io/Reader.html)and [java.io.Writer](http://docs.oracle.com/javase/8/docs/api/java/io/Writer.html).
         2. All non-abstract methods of [java.util.AbstractList](http://docs.oracle.com/javase/8/docs/api/java/util/AbstractList.html), [java.util.AbstractSet](http://docs.oracle.com/javase/8/docs/api/java/util/AbstractSet.html) and [java.util.AbstractMap](http://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html).
         3. [javax.servlet.http.HttpServlet](http://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpServlet.html), all the doXXX() methods by default sends a HTTP 405 "Method Not Allowed" error to the response. You're free to implement none or any of them.
    1. Visitor
    2. Null object

1. In practice

[Behavioural patterns](http://en.wikipedia.org/wiki/Behavioral_pattern)

[Iterator](http://en.wikipedia.org/wiki/Iterator_pattern) (recognizable by behavioural methods sequentially returning instances of a *different* type from a queue)

* All implementations of [java.util.Iterator](http://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html) (thus among others also [java.util.Scanner](http://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html)!).
* All implementations of [java.util.Enumeration](http://docs.oracle.com/javase/8/docs/api/java/util/Enumeration.html)

[Mediator](http://en.wikipedia.org/wiki/Mediator_pattern) (recognizable by behavioural methods taking an instance of different abstract/interface type (usually using the command pattern) which delegates/uses the given instance)

* [java.util.Timer](http://docs.oracle.com/javase/8/docs/api/java/util/Timer.html) (all scheduleXXX() methods)
* [java.util.concurrent.Executor#execute()](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executor.html#execute-java.lang.Runnable-)
* [java.util.concurrent.ExecutorService](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html) (the invokeXXX() and submit() methods)
* [java.util.concurrent.ScheduledExecutorService](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ScheduledExecutorService.html) (all scheduleXXX() methods)
* [java.lang.reflect.Method#invoke()](http://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Method.html#invoke-java.lang.Object-java.lang.Object...-)

[Memento](http://en.wikipedia.org/wiki/Memento_pattern) (recognizable by behavioural methods which internally changes the state of the *whole* instance)

* [java.util.Date](http://docs.oracle.com/javase/8/docs/api/java/util/Date.html) (the setter methods do that, Date is internally represented by a long value)
* All implementations of [java.io.Serializable](http://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html)
* All implementations of [javax.faces.component.StateHolder](http://docs.oracle.com/javaee/7/api/javax/faces/component/StateHolder.html)

[Observer (or Publish/Subscribe)](http://en.wikipedia.org/wiki/Observer_pattern) (recognizable by behavioural methods which invokes a method on an instance of *another* abstract/interface type, depending on own state)

* [java.util.Observer](http://docs.oracle.com/javase/8/docs/api/java/util/Observer.html)/[java.util.Observable](http://docs.oracle.com/javase/8/docs/api/java/util/Observable.html) (rarely used in real world though)
* All implementations of [java.util.EventListener](http://docs.oracle.com/javase/8/docs/api/java/util/EventListener.html) (practically all over Swing thus)
* [javax.servlet.http.HttpSessionBindingListener](http://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpSessionBindingListener.html)
* [javax.servlet.http.HttpSessionAttributeListener](http://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpSessionAttributeListener.html)
* [javax.faces.event.PhaseListener](http://docs.oracle.com/javaee/7/api/javax/faces/event/PhaseListener.html)

[Visitor](http://en.wikipedia.org/wiki/Visitor_pattern) (recognizable by two *different* abstract/interface types which has methods defined which takes each the *other* abstract/interface type; the one actually calls the method of the other and the other executes the desired strategy on it)

* [javax.lang.model.element.AnnotationValue](http://docs.oracle.com/javase/8/docs/api/javax/lang/model/element/AnnotationValue.html) and [AnnotationValueVisitor](http://docs.oracle.com/javase/8/docs/api/javax/lang/model/element/AnnotationValueVisitor.html)
* [javax.lang.model.element.Element](http://docs.oracle.com/javase/8/docs/api/javax/lang/model/element/Element.html) and [ElementVisitor](http://docs.oracle.com/javase/8/docs/api/javax/lang/model/element/ElementVisitor.html)
* [javax.lang.model.type.TypeMirror](http://docs.oracle.com/javase/8/docs/api/javax/lang/model/type/TypeMirror.html) and [TypeVisitor](http://docs.oracle.com/javase/8/docs/api/javax/lang/model/type/TypeVisitor.html)
* [java.nio.file.FileVisitor](http://docs.oracle.com/javase/8/docs/api/java/nio/file/FileVisitor.html) and [SimpleFileVisitor](http://docs.oracle.com/javase/8/docs/api/java/nio/file/SimpleFileVisitor.html)
* [javax.faces.component.visit.VisitContext](http://docs.oracle.com/javaee/7/api/javax/faces/component/visit/VisitContext.html) and [VisitCallback](http://docs.oracle.com/javaee/7/api/javax/faces/component/visit/VisitCallback.html)