DESIGN PATTERNS

Design Patterns are very popular among software developers. A design pattern is a well described solution to a common software problem.

Benefits of using design patterns are:

1. Design Patterns are already defined and provides **industry standard approach** to solve a recurring problem, so it saves time if we sensibly use the design pattern. There are many java design patterns that we can use in our java based projects.
2. Using design patterns promotes **reusability** that leads to more **robust** and highly maintainable code. It helps in reducing total cost of ownership (TCO) of the software product.
3. Since design patterns are already defined, it makes our code easy to understand and debug. It leads to faster development and new members of team understand it easily.

Java Design Patterns are divided into three categories – **creational**, **structural**, and **behavioral** design patterns. This post serves as an index for all the java design patterns articles I have written so far.

I. Creational Design Patterns

1. Singleton Pattern
2. Factory Pattern
3. Abstract Factory Pattern
4. Builder Pattern
5. Prototype Pattern

II. Structural Design Patterns

1. Adapter Pattern
2. Composite Pattern
3. Proxy Pattern
4. Flyweight Pattern
5. Facade Pattern
6. Bridge Pattern
7. Decorator Pattern

III. Behavioral Design Patterns

1. Template Method Pattern
2. Mediator Pattern
3. Chain of Responsibility Pattern
4. Observer Pattern
5. Strategy Pattern
6. Command Pattern
7. State Pattern
8. Visitor Pattern
9. Interpreter Pattern
10. Iterator Pattern
11. Memento Pattern

II. Behavioral Design Patterns:

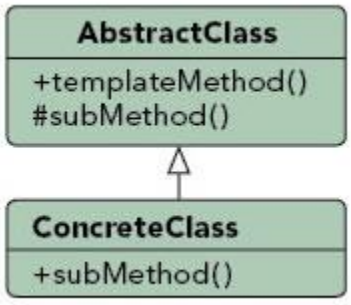
- Behavioral patterns provide solution for the better interaction between objects and how to provide lose coupling and flexibility to extend easily.
- These patterns are responsible for the efficient and safe distribution of behaviors among the program's objects.

1. Template Method Pattern:

- **Purpose**: Identifies the framework of an algorithm, allowing implementing classes to define the actual behavior.
- Template Method design pattern is used to create a method stub and deferring some of the steps of implementation to the subclasses.
- Template method defines the steps to execute an algorithm and it can provide default implementation that might be common for all or some of the subclasses.
- It makes it easier to implement complex algorithms by encapsulating logic in a single method.
- Given the pattern's definition, **the algorithm's structure will be defined in a base class that defines the template** *build()* **method**.
- In Template pattern, an abstract class exposes defined way(s)/template(s) to execute its methods. Its subclasses can override the method implementation as per need but the invocation is to be in the same way as defined by an abstract class.

Real Life Example:

A parent class, InstantMessage, will likely have all the methods required to handle sending a message. However, the actual serialization of the data to send may vary depending on the implementation. A video message and a plain text message will require different algorithms in order to serialize the data correctly. Subclasses of InstantMessage can provide their own implementation of the serialization method, allowing the parent class to work with them without understanding their implementation details.



Template Method Design Pattern in JDK:

- All non-abstract methods of java.io.Writer.
- All non-abstract methods of java.util.AbstractList, java.util.AbstractSet and java.util.AbstractMap.
- This pattern is widely used in the Java core libraries, for example by **java.util.AbstractList, or java.util.AbstractSet.**
  For instance, *Abstract List* provides a skeletal implementation of the *List* interface.

An example of a template method can be the *addAll()* method, although it's not explicitly defined as *final:*

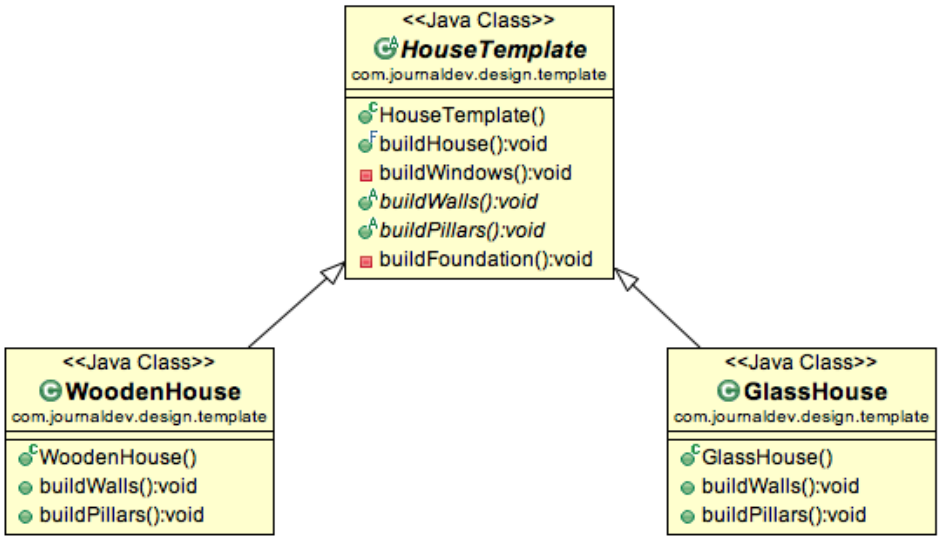
```
public boolean addAll(int index, Collection<? extends E> c) {
    rangeCheckForAdd(index);
    boolean modified = false;
    for (E e : c) {
        add(index++, e);
        modified = true;
    }
    return modified;
}
```

Users only need to implement the *add()* method:

```
public void add(int index, E element) {
    //Code goes here
}
```

Example:

Suppose we want to provide an algorithm to build a house. The steps need to be performed to build a house are – building foundation, building pillars, building walls and windows. The important point is that we can't change the order of execution because we can't build windows before building the foundation. So in this case we can create a template method that will use different methods to build the house.
Now building the foundation for a house is same for all type of houses, whether it's a wooden house or a glass house. So we can provide base implementation for this, if subclasses want to override this method, they can but mostly it's common for all the types of houses.
To make sure that subclasses don't override the template method, we should make it final.



Template Method Abstract Class

Since we want some of the methods to be implemented by subclasses, we have to make our base class as abstract class.

```
public abstract class HouseTemplate {

	//template method, final so subclasses can't override
	public final void buildHouse(){
		buildFoundation();
		buildPillars();
		buildWalls();
		buildWindows();
		System.out.println("House is built.");
	}

	//default implementation
	private void buildWindows() {
		System.out.println("Building Glass Windows.");
	}

	//methods to be implemented by subclasses
	public abstract void buildWalls();
	public abstract void buildPillars();

	private void buildFoundation() {
		System.out.println("Building foundation with cement, iron rods
and sand.");
	}
}
```

buildHouse() is the template method and defines the order of execution for performing several steps.

Template Method Concrete Classes:
We can have different type of houses, such as Wooden House and Glass House.

```
public class WoodenHouse extends HouseTemplate {

	@Override
	public void buildWalls() {
		System.out.println("Building Wooden Walls.");
	}

	@Override
	public void buildPillars() {
		System.out.println("Building Pillars with Wooden coating.");
	}
}


public class GlassHouse extends HouseTemplate {

	@Override
	public void buildWalls() {
		System.out.println("Building Glass Walls.");
	}

	@Override
	public void buildPillars() {
		System.out.println("Building Pillars with glass coating.");
	}
}
```

We can override other methods also.

Template Method Design Pattern Client
Template method pattern example with a test program.

```
public class HousingClient {

	public static void main(String[] args) {

		HouseTemplate houseType = new WoodenHouse();

		//using template method
		houseType.buildHouse();
		System.out.println("************");

		houseType = new GlassHouse();

		houseType.buildHouse();
	}
}
```

Note: Client is invoking the template method of base class and depending of implementation of different steps, it's using some of the methods from base class and some of them from subclass.

Important Points:

- Template method should consists of certain steps whose order is fixed and for some of the methods, implementation differs from base class to subclass. Template method should be final.
- Most of the times, subclasses calls methods from super class but in template pattern, superclass template method calls methods from subclasses, this is known as Hollywood Principle – "don't call us, we'll call you."
- Methods in base class with default implementation are referred as **Hooks** and they are intended to be overridden by subclasses, if you want some of the methods to be not overridden, you can make them final, for example in our case we can make buildFoundation() method final because if we don't want subclasses to override it.
- **The template method pattern promotes code reuse and decoupling, but at the expense of using inheritance.**

When to use Template Method Pattern:

- A single abstract implementation of an algorithm is needed.
- Common behavior among subclasses should be localized to a common class.
- Parent classes should be able to uniformly invoke behavior in their subclasses.
- Most or all subclasses need to implement the behavior.

2. Mediator Pattern:

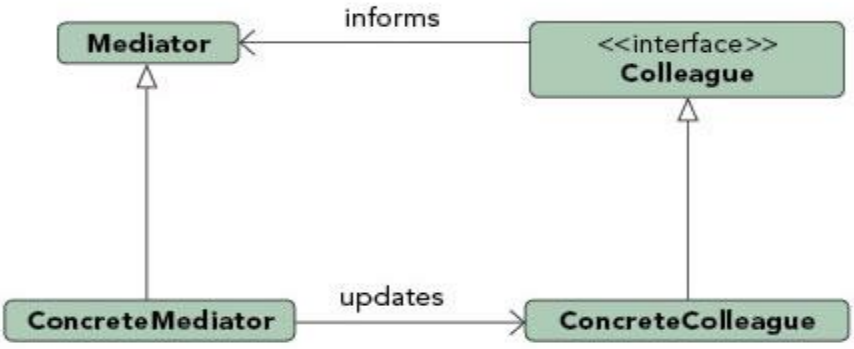
- **Purpose**: Allows loose coupling by encapsulating the way disparate sets of objects interact and communicate with each other. Allows for the actions of each object set to vary independently of one another.
- Mediator design pattern is used to **provide a centralized communication medium** between different objects in a system.
- Mediator design pattern is very helpful in an enterprise application where multiple objects are interacting with each other.
- If the objects interact with each other directly, the system components are tightly-coupled with each other that makes maintainability cost higher and not flexible to extend easily.
- Mediator pattern focuses on provide a mediator between objects for communication and help in implementing lose-coupling between objects.
- Mediator pattern is used to reduce communication complexity between multiple objects or classes.
- Mediator design pattern is very helpful in an enterprise application where multiple objects are interacting with each other. If the objects interact with each other directly, the system components are tightly-coupled with each other that makes higher maintainability cost and not hard to extend.

Mediator Pattern Example in JDK:

- java.util.Timer class scheduleXXX() methods
- Java Concurrency Executor execute() method.
- java.lang.reflect.Method invoke() method.

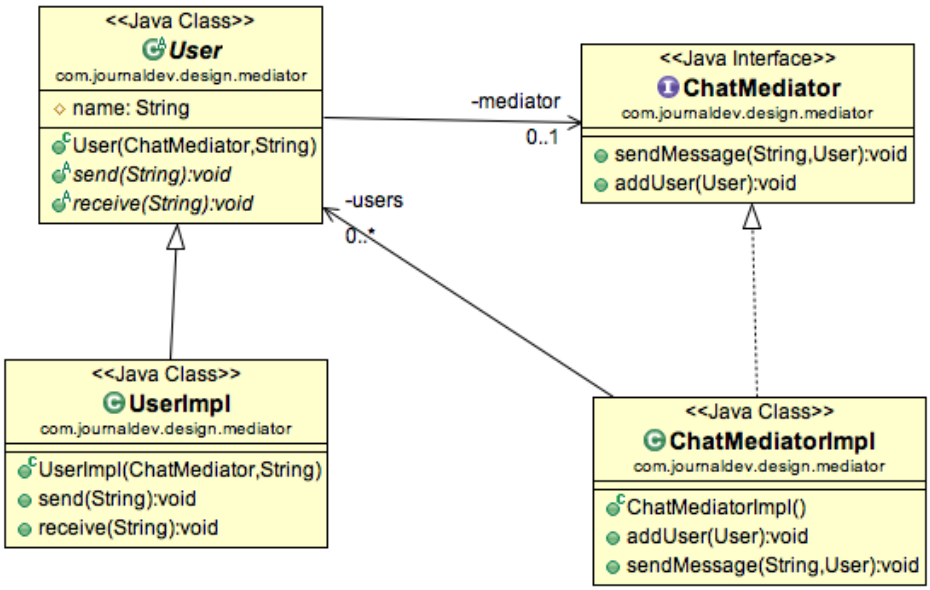
Real Life Example:
Air traffic controller is a great example of mediator pattern where the airport control room works as a mediator for communication between different flights. Mediator works as a router between objects and it can have its own logic to provide way of communication.

The system objects that communicate each other are called **Colleagues**. Usually we have an interface or abstract class that provides the contract for communication and then we have concrete implementation of mediators.



Another example:
Mailing list software keeps track of who is signed up to the mailing list and provides a single point of access through which any one person can communicate with the entire list. Without a mediator implementation a person wanting to send a message to the group would have to constantly keep track of who was signed up and who was not. By implementing the mediator pattern the system is able to receive messages from any point then determine which recipients to forward the message on to, without the sender of the message having to be concerned with the actual recipient list.

Example:
Create a chat application where users can do group chat. Every user will be identified by its name and they can send and receive messages. The message sent by any user should be received by all the other users in the group.



Mediator Pattern Colleague Interface
Users can send and receive messages, so we can have User interface or abstract class.

```
public abstract class User {

	protected ChatMediator mediator;
	protected String name;

	public User(ChatMediator med, String name){
		this.mediator = med;
		this.name = name;
	}

	public abstract void send(String msg);
	public abstract void receive(String msg);
}
```

Note: User has a reference to the mediator object, it's required for the communication between different users.

Mediator Pattern Interface:

Create Mediator interface that will define the contract for concrete mediators.

```
public interface ChatMediator {

	public void sendMessage(String msg, User user);
	void addUser(User user);
}
```

Concrete Mediator:

Create concrete mediator class, it will have a list of users in the group and provide logic for the communication between the users.

```
public class ChatMediatorImpl implements ChatMediator {

	private List<User> users;

	public ChatMediatorImpl() {
		this.users=new ArrayList<>();
	}

	@Override
	public void addUser(User user) {
		this.users.add(user);
	}
	@Override
	public void sendMessage(String msg, User user) {
		this.users.forEach(u -> {
			//message should not be received by the user sending it
			if(u != user)
				u.receive(msg);
		});
	}
}
```

Mediator Design Pattern Concrete Colleague

Create concrete User classes to be used by client system.

```
public class UserImpl extends User {

	public UserImpl(ChatMediator med, String name) {
		super(med, name);
	}

	@Override
	public void send(String msg){
		System.out.println(this.name + ": Sending Message=" + msg);
		mediator.sendMessage(msg, this);
	}

	@Override
	public void receive(String msg) {
		System.out.println(this.name + ": Received Message:" + msg);
	}
}
```

Note: **send**() method is using mediator to send the message to the users and it has no idea how it will be handled by the mediator.

Mediator Pattern Example Client Program Code:

Test the chat application with a simple program where we will create mediator and add users to the group and one of the user will send a message.

```
public class ChatClient {
	public static void main(String[] args) {

		ChatMediator mediator = new ChatMediatorImpl();

		User user1 = new UserImpl(mediator, "Pankaj");
		User user2 = new UserImpl(mediator, "Lisa");
		User user3 = new UserImpl(mediator, "Saurabh");
		User user4 = new UserImpl(mediator, "David");

		mediator.addUser(user1);
		mediator.addUser(user2);
		mediator.addUser(user3);
		mediator.addUser(user4);

		user1.send("Hi All");
	}
}
```

Note: Client program is very simple and it has no idea how the message is getting handled and if mediator is getting user or not.

Important Points

- Mediator pattern is useful when the communication logic between objects is complex, we can have a central point of communication that takes care of communication logic.
- Java Message Service (JMS) uses Mediator pattern along with Observer pattern to allow applications to subscribe and publish data to other applications.
- We should not use mediator pattern just to achieve lose-coupling because if the number of mediators will grow, then it will become hard to maintain them.

When to use Mediator Pattern:

- Communication between sets of objects is well defined and complex.
- Too many relationships exist and common point of control or communication is needed.

3. Chain of Responsibility Pattern:

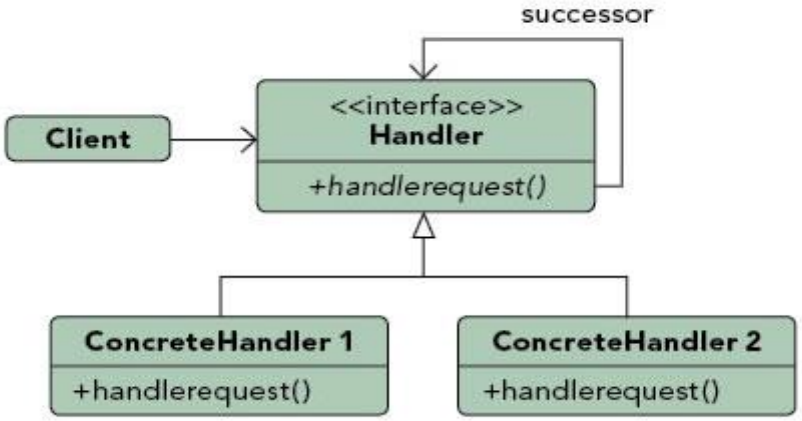
- **Purpose:** Gives more than one object an opportunity to handle a request by linking receiving objects together.
- Chain of responsibility pattern is used to achieve lose coupling in software design where a request from client is passed to a chain of objects to process them. Then the object in the chain will decide themselves who will be processing the request and whether the request is required to be sent to the next object in the chain or not.
- As the name suggests, the chain of responsibility pattern creates a chain of receiver objects for a request. This pattern decouples sender and receiver of a request based on type of request.
- In this pattern, normally each receiver contains reference to another receiver. If one object cannot handle the request then it passes the same to the next receiver and so on.

Chain of Responsibility Pattern Example in JDK:

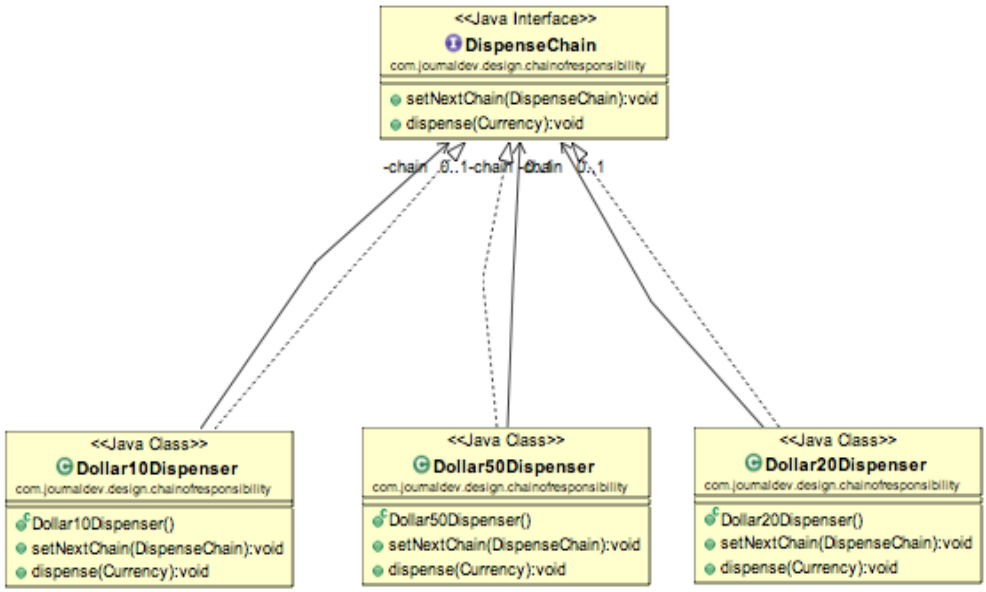
- java.util.logging.Logger#log()
- javax.servlet.Filter#doFilter()
- We know that we can have multiple catch blocks in a try-catch block code. Here every catch block is kind of a processor to process that particular exception.
  So when any exception occurs in the try block, its send to the first catch block to process. If the catch block is not able to process it, it forwards the request to next object in chain i.e. next catch block. If even the last catch block is not able to process it, the exception is thrown outside of the chain to the calling program.
- **Another such classic example is Servlet Filters** in Java that allow multiple filters to process an HTTP request. Though in that case, **each filter invokes the chain instead of the next filter.**

Real Life Example:

One of the great example of Chain of Responsibility pattern is **ATM Dispense machine**. The user enters the amount to be dispensed and the machine dispense amount in terms of defined currency bills such as 50$, 20$, 10$ etc.
If the user enters an amount that is not multiples of 10, it throws error. We will use Chain of Responsibility pattern to implement this solution.



Example:
We can implement ATM dispenser solution that will create a chain of dispense systems to dispense bills of 50$, 20$ and 10$.



Create a class Currency that will store the amount to dispense and used by the chain implementations.

```
public class Currency {

	private int amount;

	public Currency(int amt) {
		this.amount = amt;
	}

	public int getAmount() {
		return this.amount;
	}
}
```

The base interface should have a method to define the next processor in the chain and the method that will process the request. ATM Dispense interface like below;

```
public interface DispenseChain {

	void setNextChain(DispenseChain nextChain);
	void dispense(Currency cur);
}
```

Chain of Responsibilities Pattern – Chain Implementations

We need to create different processor classes that will implement the **DispenseChain** interface and provide implementation of dispense methods. Since we are developing our system to work with three types of currency bills – 50$, 20$ and 10$, we will create three concrete implementations.

```
public class Dollar50Dispenser implements DispenseChain {

	private DispenseChain chain;

	@Override
	public void setNextChain(DispenseChain nextChain) {
		this.chain = nextChain;

	}

	@Override
	public void dispense(Currency cur) {
		if(cur.getAmount() >= 50) {
			int num = cur.getAmount()/50;
			int remainder = cur.getAmount() % 50;
			System.out.println("Dispensing "+ num + " 50$ note");

			if(remainder !=0)
				this.chain.dispense(new Currency(remainder));

		} else {
			this.chain.dispense(cur);
		}
	}
}
public class Dollar20Dispenser implements DispenseChain {

	private DispenseChain chain;

	@Override
	public void setNextChain(DispenseChain nextChain) {
		this.chain = nextChain;
	}

	@Override
	public void dispense(Currency cur) {
		if(cur.getAmount() >= 20){
			int num = cur.getAmount()/20;
			int remainder = cur.getAmount() % 20;
			System.out.println("Dispensing " + num + " 20$ note");

			if(remainder != 0)
				this.chain.dispense(new Currency(remainder));

		} else {
			this.chain.dispense(cur);
		}
	}
}
```

```
public class Dollar10Dispenser implements DispenseChain {

	private DispenseChain chain;

	@Override
	public void setNextChain(DispenseChain nextChain) {
		this.chain = nextChain;
	}

	@Override
	public void dispense(Currency cur) {
		if(cur.getAmount() >= 10){
			int num = cur.getAmount()/10;
			int remainder = cur.getAmount() % 10;
			System.out.println("Dispensing " + num + " 10$ note");

			if(remainder != 0)
				this.chain.dispense(new Currency(remainder));

		} else {
			this.chain.dispense(cur);
		}
	}
}
```

Note: The implementation of dispense method. You will notice that every implementation is trying to process the request and based on the amount, it might process some or full part of it.

If one of the chain not able to process it fully, it sends the request to the next processor in chain to process the remaining request. If the processor is not able to process anything, it just forwards the same request to the next chain.

Chain of Responsibilities Design Pattern – Creating the Chain

This is a very important step and chain should be created carefully, otherwise a processor might not be getting any request at all. For example, in our implementation if we keep the first processor chain as Dollar10Dispenser and then Dollar20Dispenser, then the request will never be forwarded to the second processor and the chain will become useless.

Here is our ATM Dispenser implementation to process the user requested amount.

```
public class ATMDispenseChain {

	private DispenseChain dispense50;

	public ATMDispenseChain() {
		// initialize the chain
		this.dispense50 = new Dollar50Dispenser();
		DispenseChain dispense20 = new Dollar20Dispenser();
		DispenseChain dispence10 = new Dollar10Dispenser();

		// set the chain of responsibility
		dispense50.setNextChain(dispense20);
		dispense20.setNextChain(dispence10);
	}
```

```
	public static void main(String[] args) {
		ATMDispenseChain atmDispenser = new ATMDispenseChain();

		while (true) {
			int amount = 0;
			System.out.println("Enter amount to dispense: ");

			Scanner input = new Scanner(System.in);
			amount = input.nextInt();
			if (amount % 10 != 0) {
				System.out.println("Amount should be in multiple of
				10s.");
				return;
			}
			// process the request
			atmDispenser.dispense50.dispense(new Currency(amount));

			input.close();
			System.exit(0);
		}
	}
}
```

Important Points:

- Client doesn't know which part of the chain will be processing the request and it will send the request to the first object in the chain. For example, in our program ATMDispenseChain is unaware of who is processing the request to dispense the entered amount.
- Each object in the chain will have its own implementation to process the request, either full or partial or to send it to the next object in the chain.
- Every object in the chain should have reference to the next object in chain to forward the request to, it's achieved by **java composition**.
- Creating the chain carefully is very important otherwise there might be a case that the request will never be forwarded to a particular processor or there are no objects in the chain who are able to handle the request. In our implementation, we have added the check for the user entered amount to make sure it gets processed fully by all the processors but we might not check it and throw exception if the request reaches the last object and there are no further objects in the chain to forward the request to. This is a design decision.
- Chain of Responsibility design pattern is good to achieve lose coupling but it comes with the trade-off of having a lot of implementation classes and maintenance problems if most of the code is common in all the implementations.

Disadvantages of Chain of Responsibility Pattern:

- Mostly, it can get broken easily:
  - If a processor fails to call the next processor, the command gets dropped.
  - If a processor calls the wrong processor, it can lead to a cycle.
- It can create deep stack traces, which can affect performance.
- It can lead to duplicate code across processors, increasing maintenance.

When to use Chain of Responsibility Pattern:

- Multiple objects may handle a request and the handler doesn't have to be a specific object.
- A set of objects should be able to handle a request with the handler determined at runtime.
- A request not being handled is an acceptable potential outcome.

4. Observer Pattern:

- **Purpose**: Lets one or more objects be notified of state changes in other objects within the system.
- Observer design pattern is useful when you are interested in the state of an object and want to get notified whenever there is any change.
- In observer pattern, the object that watch on the state of another object are called **Observer** and the object that is being watched is called **Subject**.
- Observer design pattern intent is**; define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.**
- **Subject** contains a list of observers to notify of any change in its state, so it should provide methods using which observers can register and unregister themselves. Subject also contain a method to notify all the observers of any change and either it can send the update while notifying the observer or it can provide another method to get the update.
- **Observer** should have a method to set the object to watch and another method that will be used by Subject to notify them of any updates.
- It specifies communication between objects: *observable* and *observers*.
- **An** *observable*/subject **is an object which notifies** *observers* **about the changes in its state.**

Observer Pattern Implementation in JDK:

- Java provides inbuilt platform for implementing Observer pattern through *java.util.Observable* class and *java.util.Observer* interface.
  However it's not widely used because the implementation is really simple and most of the times we don't want to end up extending a class just for implementing Observer pattern as java doesn't provide multiple inheritance in classes.

- Java Message Service (JMS) uses **Observer design pattern** along with **Mediator pattern** to allow applications to subscribe and publish data to other applications.

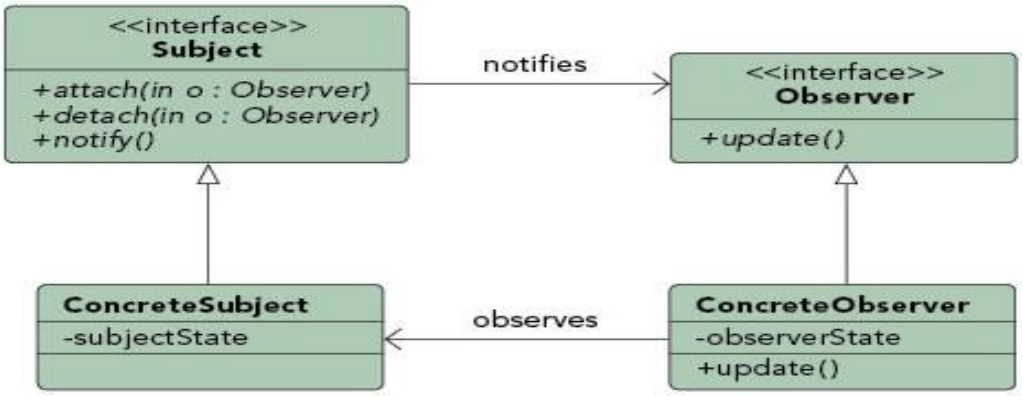
- Model-View-Controller (MVC) frameworks also use Observer pattern where Model is the Subject and Views are observers that can register to get notified of any change to the model.

Observer design pattern is also called as publish-subscribe pattern. Some of its implementations are;

- java.util.EventListener in Swing
- javax.servlet.http.HttpSessionBindingListener
- javax.servlet.http.HttpSessionAttributeListener

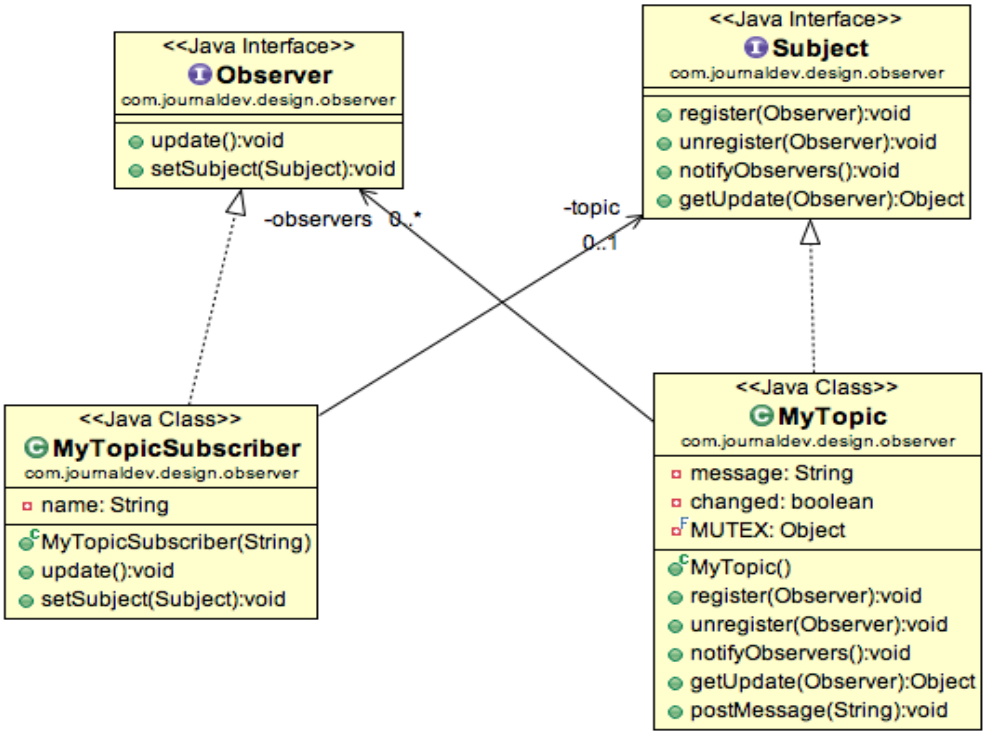
Real Life Example:

This pattern can be found in almost every GUI environment. When buttons, text, and other fields are placed in applications the application typically registers as a listener for those controls. When a user triggers an event, such as clicking a button, the control iterates through its registered observers and sends a notification to each.



Example:

Implementing a simple topic and observers can register to this topic. Whenever any new message will be posted to the topic, all the registers observers will be notified and they can consume the message.



Based on the requirements of Subject, **the base Subject interface defines the contract methods to be implemented by any concrete subject.**

```
public interface Subject {

	//methods to register and unregister observers
	public void register(Observer obj);
	public void unregister(Observer obj);

	//method to notify observers of change
	public void notifyObservers();

	//method to get updates from subject
	public Object getUpdate(Observer obj);
}
```

Create contract for Observer, where a method to attach the Subject to the observer and another method to be used by Subject to notify of any change.

```
public interface Observer {

	//method to update the observer, used by subject
	public void update();
	//attach with subject to observe
	public void setSubject(Subject sub);
}
```

Contract is ready, the concrete implementation.

```
public class MyTopic implements Subject {

	private List<Observer> observers;
	private String message;
	private boolean changed;
	private final Object MUTEX = new Object();

	public MyTopic() {
		this.observers = new ArrayList<>();
	}

	@Override
	public void register(Observer obj) {
		if(obj == null)
			throw new NullPointerException("Null Observer");

		synchronized (MUTEX) {
			if(!observers.contains(obj)) observers.add(obj);
		}
	}
```

```
	@Override
	public void unregister(Observer obj) {
		synchronized (MUTEX) {
			observers.remove(obj);
		}
	}

	@Override
	public void notifyObservers() {
		List<Observer> observersLocal = null;

		//synchronization is used to make sure any observer registered
after message is received is not notified
		synchronized (MUTEX) {
			if (!changed)
				return;
			observersLocal = new ArrayList<>(this.observers);
			this.changed = false;
		}
		observersLocal.forEach(obj -> obj.update());
	}

	@Override
	public Object getUpdate(Observer obj) {
		return this.message;
	}

	//method to post message to the topic
	public void postMessage(String msg){
		System.out.println("Message Posted to Topic: " + msg);
		this.message = msg;
		this.changed = true;
		notifyObservers();
	}
}
```

The method implementation to register and unregister an observer is very simple, the extra method is *postMessage()* that will be used by client application to post String message to the topic. Notice the boolean variable to keep track of the change in the state of topic and used in notifying observers. This variable is required so that if there is no update and somebody calls *notifyObservers()* method, it doesn't send false notifications to the observers.

Note: use of **synchronization** in *notifyObservers()* method to make sure the notification is sent only to the observers registered before the message is published to the topic.

Implementation of Observers that will watch over the subject:

```
public class MyTopicSubscriber implements Observer {

	private String name;
	private Subject topic;

	public MyTopicSubscriber(String nme) {
		this.name = nme;
	}

	@Override
	public void update() {
		String msg = (String) topic.getUpdate(this);
		if(msg == null) {
			System.out.println(name + ":: No new message");
		} else
			System.out.println(name + ":: Consuming message::" + msg);
	}

	@Override
	public void setSubject(Subject sub) {
		this.topic = sub;
	}
}
```

Note: implementation of *update()* method where it's calling Subject *getUpdate()* method to get the message to consume. We could have avoided this call by passing message as argument to *update()*method.

Test program to consume our topic implementation:

```
public class ObserverPattern {

	public static void main(String[] args) {
		//create subject
		MyTopic topic = new MyTopic();

		//create observers
		Observer obj1 = new MyTopicSubscriber("Obj1");
		Observer obj2 = new MyTopicSubscriber("Obj2");
		Observer obj3 = new MyTopicSubscriber("Obj3");

		//register observers to the subject
		topic.register(obj1);
		topic.register(obj2);
		topic.register(obj3);

		//attach observer to subject
		obj1.setSubject(topic);
		obj2.setSubject(topic);
		obj3.setSubject(topic);

		//check if any update is available
		obj1.update();
```

```
            //now send message to subject
            topic.postMessage("New Message");
        }
}
```

When to use Observer Pattern:

- State changes in one or more objects should trigger behavior in other objects
- Broadcasting capabilities are required.
- An understanding exists that objects will be blind to the expense of notification.

5. Strategy Pattern:

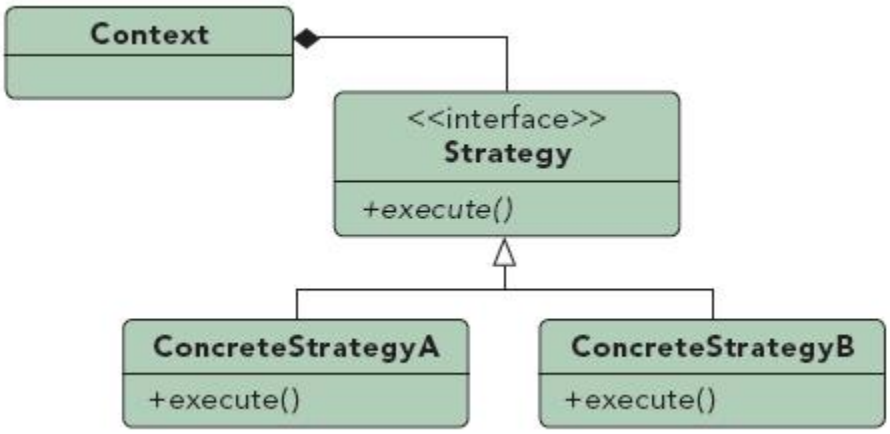
- **Purpose:** Defines a set of encapsulated algorithms that can be swapped to carry out a specific behavior.
- Strategy pattern is used when we have multiple algorithm for a specific task and client decides the actual implementation to be used at runtime.
- Strategy pattern is also known as Policy Pattern. We define multiple algorithms and let client application pass the algorithm to be used as a parameter.
- Typically, we would start with an interface which is used to apply an algorithm, and then implement it multiple times for each possible algorithm. Typically, we would start with an interface which is used to apply an algorithm, and then implement it multiple times for each possible algorithm.
- In Strategy pattern, a class behavior or its algorithm can be changed at run time.
- 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.

Implementation of Strategy Pattern in JDK:

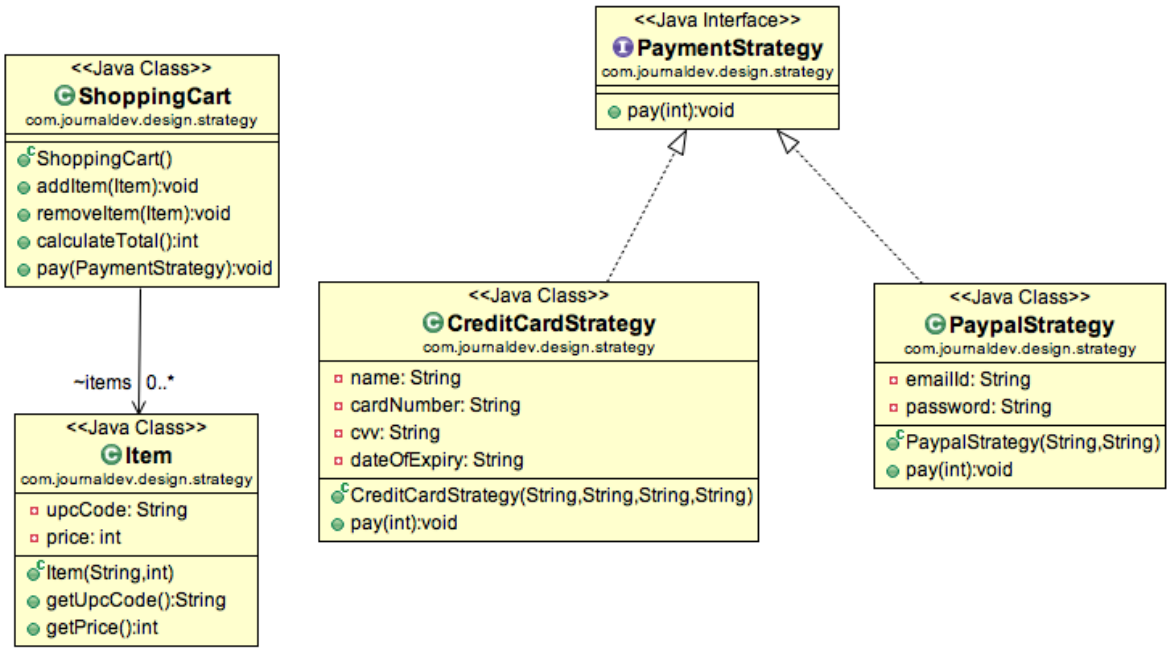
One of the best example of strategy pattern is **Collections.sort()** method that takes Comparator parameter. Based on the different implementations of Comparator interfaces, the Objects are getting sorted in different ways.

Real Life Example:

When importing data into a new system different validation algorithms may be run based on the data set. By configuring the import to utilize strategies the conditional logic to determine what validation set to run can be removed and the import can be decoupled from the actual validation code. This will allow us to dynamically call one or more strategies during the import.



Example:
Implementing a simple Shopping Cart where we have two payment strategies – using Credit Card or using PayPal.



Create an interface for our strategy pattern example, in our case to pay the amount passed as argument.

```
public interface PaymentStrategy {

	public void pay(int amount);
}
```

Create concrete implementation of algorithms for payment using credit/debit card or through Paypal.

```
public class CreditCardStrategy implements PaymentStrategy {

	private String name;
	private String cardNumber;
	private String cvv;
	private String dateOfExpiry;

	public CreditCardStrategy(String nme, String ccNum, String cvv, String
expiryDate) {
		this.name = nme;
		this.cardNumber = ccNum;
		this.cvv = cvv;
		this.dateOfExpiry = expiryDate;
	}
```

```
	@Override
	public void pay(int amount) {
		System.out.println(amount +" paid with credit/debit card");
	}
}


public class PayPalStrategy implements PaymentStrategy {

	private String emailId;
	private String password;

	public PayPalStrategy(String email, String pwd) {
		this.emailId = email;
		this.password = pwd;
	}

	@Override
	public void pay(int amount) {
		System.out.println(amount + " paid using Paypal.");
	}
}
```

Implement Shopping Cart and payment method will require input as Payment strategy.

```
public class Item {

	private String upcCode;
	private int price;

	public Item(String upc, int cost) {
		this.upcCode = upc;
		this.price = cost;
	}

	public String getUpcCode() {
		return upcCode;
	}

	public int getPrice() {
		return price;
	}
}

public class ShoppingCart {

	//List of items
	List<Item> items;

	public ShoppingCart() {
		this.items = new ArrayList<>();
	}

	public void addItem(Item item) {
		this.items.add(item);
	}
```

```
	public void removeItem(Item item) {
		this.items.remove(item);
	}

	public int calculateTotal() {
		int sum = 0;

		for(Item item : items) {
			sum += item.getPrice();
		}
		return sum;
	}

	public void pay(PaymentStrategy paymentMethod) {
		int amount = calculateTotal();
		paymentMethod.pay(amount);
	}
}
```

Note: Payment method of shopping cart requires payment algorithm as argument and doesn't store it anywhere as instance variable.

```
public class StrategyPattern {

	public static void main(String[] args) {
		ShoppingCart cart = new ShoppingCart();

		Item item1 = new Item("1234", 10);
		Item item2 = new Item("5678", 40);

		cart.addItem(item1);
		cart.addItem(item2);

		//pay by paypal
		cart.pay(new PayPalStrategy("myemail@example.com", "mypwd"));

		//pay by credit card
		cart.pay(new CreditCardStrategy("Pankaj Kumar",
"1234567890123456", "786", "12/15"));
	}
}
```

Output:

```
50 paid using Paypal.
50 paid with credit/debit card
```

Important Points:

- We could have used composition to create instance variable for strategies but we should avoid that as we want the specific strategy to be applied for a particular task. Same is followed in **Collections.sort() and Arrays.sort()** method that take comparator as argument.

- Strategy Pattern is very similar to **State Pattern**. One of the difference is that Context contains state as instance variable and there can be multiple tasks whose implementation can be dependent on the state whereas in strategy pattern strategy is passed as argument to the method and context object doesn't have any variable to store it.

- Strategy pattern is useful when we have multiple algorithms for specific task and we want our application to be flexible to choose any of the algorithm at runtime for specific task.

When to use Strategy Pattern:

- The only difference between many related classes is their behavior.
- Multiple versions or variations of an algorithm are required.
- Algorithms access or utilize data that calling code shouldn't be exposed to.
- The behavior of a class should be defined at runtime.
- Conditional statements are complex and hard to maintain.

Note: **Also refer, https://www.baeldung.com/java-strategy-pattern**

6. **Command Pattern:**

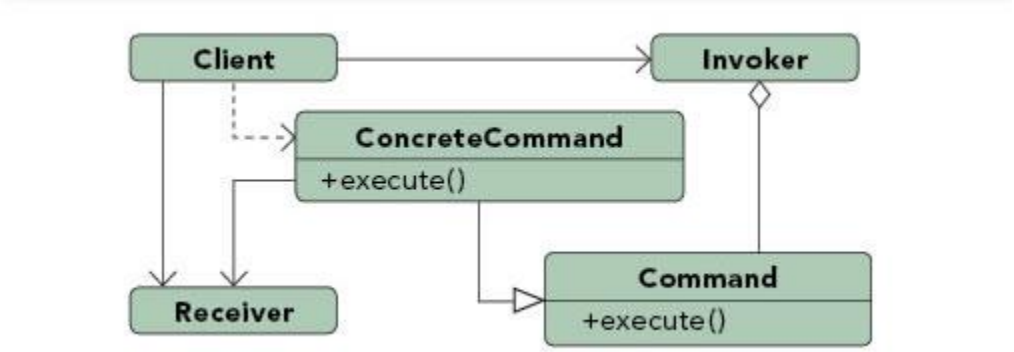
- **Purpose:** Encapsulates a request allowing it to be treated as an object. This pattern allows the request to be handled in traditionally object based relationships such as queuing and callbacks.
- In command pattern, the request is send to the **invoker** and invoker pass it to the encapsulated **command** object.
- Command object passes the request to the appropriate method of **Receiver** to perform the specific action.
- The client program create the receiver object and then attach it to the Command. Then it creates the invoker object and attach the command object to perform an action.
- Now when client program executes the action, it's processed based on the command and receiver object.
- Command pattern is a data driven design.
- 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.

Command Design Pattern JDK Example:

- **Runnable interface** (java.lang.Runnable) and **Swing Action** (javax.swing.Action) uses command pattern.

Real Life Example:

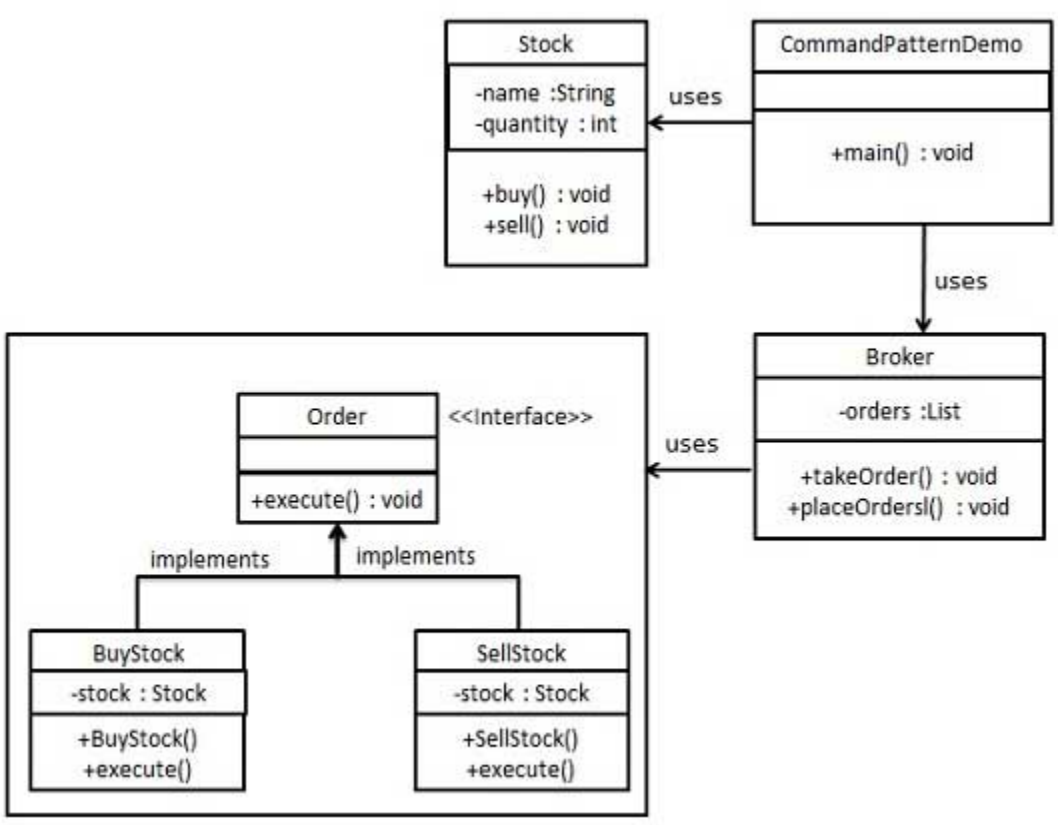
Job queues are widely used to facilitate the asynchronous processing of algorithms. By utilizing the command pattern the functionality to be executed can be given to a job queue for processing without any need for the queue to have knowledge of the actual implementation it is invoking. The command object that is en-queued implements its particular algorithm within the confines of the interface the queue is expecting.



Example:

Created an interface *Order* which is acting as a command. We have created a *Stock* class which acts as a request. We have concrete command classes **BuyStock** and **SellStock** implementing **Order** interface which will do actual command processing. A class *Broker* is created which acts as an invoker object. It can take and place orders.

Broker object uses command pattern to identify which object will execute which command based on the type of command. **CommandPatternDemo**, our demo class, will use *Broker* class to demonstrate command pattern.



Create a command interface.

```
public interface Order {
      void execute();
}
```

Create a request class.

```
public class Stock {

      private String name = "ABC";
      private int quantity = 10;
```

```
	public void buy(){
		System.out.println("Stock [ Name: "+name+", Quantity: " +
		quantity +" ] bought");
	}

	public void sell(){
		System.out.println("Stock [ Name: "+name+",     Quantity: " +
		quantity +" ] sold");
	}
}
```

Create concrete classes implementing the *Order* interface.

```
public class BuyStock implements Order {

	private Stock abcStock;

	public BuyStock(Stock abcStock) {
		this.abcStock = abcStock;
	}

	public void execute() {
		abcStock.buy();
	}
}

public class SellStock implements Order {

	private Stock abcStock;

	public SellStock(Stock abcStock) {
		this.abcStock = abcStock;
	}

	public void execute() {
		abcStock.sell();
	}
}
```

Create command invoker class.

```
public class Broker {

	private List<Order> orderList = new ArrayList<>();

	public void takeOrder(Order order) {
		orderList.add(order);
	}


	public void placeOrders() {
		orderList.forEach(Order::execute);
		orderList.clear();
	}
}
```

Use the Broker class to take and execute commands.

```
public class CommandPattern {

	public static void main(String[] args) {
		Stock abcStock = new Stock();

		BuyStock buyStockOrder = new BuyStock(abcStock);
		SellStock sellStockOrder = new SellStock(abcStock);

		Broker broker = new Broker();
		broker.takeOrder(buyStockOrder);
		broker.takeOrder(sellStockOrder);

		broker.placeOrders();
	}
}
```

Output:

```
Stock [ Name: ABC, Quantity: 10 ] bought
Stock [ Name: ABC, Quantity: 10 ] sold
```

When to use Command Pattern:

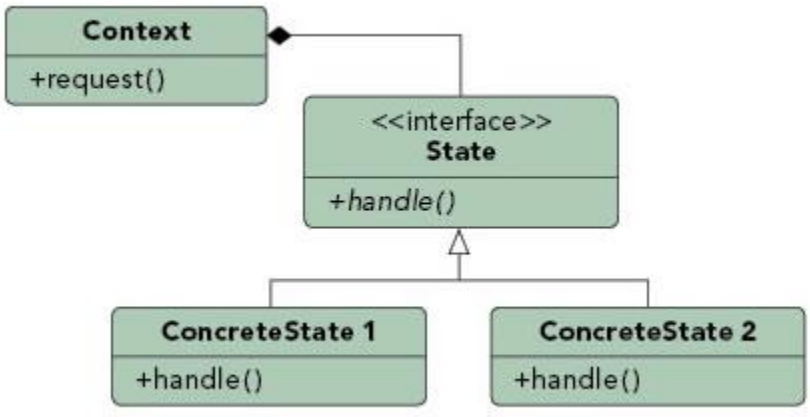
- You need callback functionality.
- Requests need to be handled at variant times or in variant orders.
- A history of requests is needed.
- The invoker should be decoupled from the object handling the invocation.

7. **State Pattern:**

- **Purpose:** Ties object circumstances to its behavior, allowing the object to behave in different ways based upon its internal state.
- In State pattern a class behavior changes based on its state. This type of design pattern comes under behavior pattern.
- In State pattern, we create objects which represent various states and a context object whose behavior varies as its state object changes.
- The main idea of State pattern is to **allow the object for changing its behavior without changing its class.** Also, by implementing it, the code should remain cleaner without many if/else statements.
- If we have to change the behavior of an object based on its state, we can have a state variable in the Object. Then use if-else condition block to perform different actions based on the state.
- State design pattern is used to provide a systematic and loosely coupled way to achieve this through **Context and State** implementations.
- State Pattern Context is the class that has a **State** reference to one of the concrete implementations of the State. **Context** forwards the request to the state object for processing.

Real-life Example:

An email object can have various states, all of which will change how the object handles different functions. If the state is "not sent" then the call to send() is going to send the message while a call to recallMessage() will either throw an error or do nothing. However, if the state is "sent" then the call to send() would either throw an error or do nothing while the call to recallMessage() would attempt to send a recall notification to recipients. To avoid conditional statements in most or all methods there would be multiple state objects that handle the implementation with respect to their particular state. The calls within the Email object would then be delegated down to the appropriate state object for handling.



Example:

Suppose we want to implement a TV Remote with a simple button to perform action. If the State is ON, it will turn on the TV and if state is OFF, it will turn off the TV.

We can implement it using if-else condition like below;

```
public class TVRemoteBasic {

	private String state = "";

	public void setState(String state) {
		this.state = state;
	}

	public void doAction() {
		if(state.equalsIgnoreCase("ON")) {
			System.out.println("TV is turned ON.");
		} else if(state.equalsIgnoreCase("OFF")) {
			System.out.println("TV is turned OFF");
		}
	}

	public static void main(String[] args) {
		TVRemoteBasic remote = new TVRemoteBasic();

		remote.setState("ON");
		remote.doAction();

		remote.setState("OFF");
		remote.doAction();
	}
}
```

Output:

```
TV is turned ON.
TV is turned OFF.
```

Note: Client code should know the specific values to use for setting the state of remote. Furthermore, if number of states increase then the tight coupling between implementation and the client code will be very hard to maintain and extend.

Implementing State Pattern for above example:

Create State interface that will define the method that should be implemented by different concrete states and context class.

```
public interface State {

	public void doAction();
}
```

State Design Pattern Concrete State Implementations:

We can have two states – one for turning TV on and another to turn it off. So we will create two concrete state implementations for these behaviors.

```
public class TVStartState implements State {

	@Override
	public void doAction() {
		System.out.println("TV is turned ON.");
	}
}


public class TVStopState implements State {

	@Override
	public void doAction() {
		System.out.println("TV is turned OFF.");
	}
}
```

State Design Pattern Context Implementation:

Implement our Context object that will change its behavior based on its internal state.

```
public class TVContext implements State {

	private State tvState;

	public State getState() {
		return tvState;
	}

	public void setState(State state) {
		this.tvState = state;
	}

	@Override
	public void doAction() {
		this.tvState.doAction();
	}
}
```

Note: Context also implements State and keep a reference of its current state and forwards the request to the state implementation.

Context class has an associated State which is going to change during program execution.

Our context is going to delegate the behavior to the state implementation. In other words, all incoming requests will be handled by the concrete implementation of the state.

State pattern implementation:

```
public class TVRemote {

	public static void main(String[] args) {

		TVContext context = new TVContext();
		State tvStartState = new TVStartState();
		State tvStopState = new TVStopState();

		context.setState(tvStartState);
		context.doAction();

		context.setState(tvStopState);
		context.doAction();
	}
}
```

Output:

```
TV is turned ON.
TV is turned OFF.
```

Note: Output of above program is same as the basic implementation of TV Remote without using state pattern.

As we've been changing the state of our context, the behavior was changing but the class remains the same. As well as the API we make use of.

Also, the transition between the states has occurred, our class changed its state and consequentially its behavior.

Benefits of using State Pattern:

- The benefits of using State pattern to implement polymorphic behavior is clearly visible. The chances of error are less and it's very easy to add more states for additional behavior. Thus making our code more robust, easily maintainable and flexible.
- Also State pattern helped in avoiding if-else or switch-case conditional logic in this scenario.
- Context delegates the behavior to the state implementation. All incoming requests are handled by the concrete implementation of the state.
- Logic is separated and adding new states is simple – it comes down to adding another State implementation if needed.

Drawbacks of State Pattern:

- State pattern drawback is the payoff when implementing transition between the states. That makes the state hardcoded, which is a bad practice in general.
- But, depending on our needs and requirements, that might or might not be an issue.

State Pattern is very similar to Strategy Pattern.
State vs. Strategy Pattern:

- First, the strategy pattern **defines a family of interchangeable algorithms**. Generally, they achieve the same goal, but with a different implementation, for example, sorting or rendering algorithms. **In state pattern, the behavior might change completely**, based on actual state.

- Next, **in strategy, the client has to be aware of the possible strategies to use and change them explicitly.** Whereas in **state pattern**, each state is linked to another and create the flow as in Finite State Machine.

When to use State Pattern:

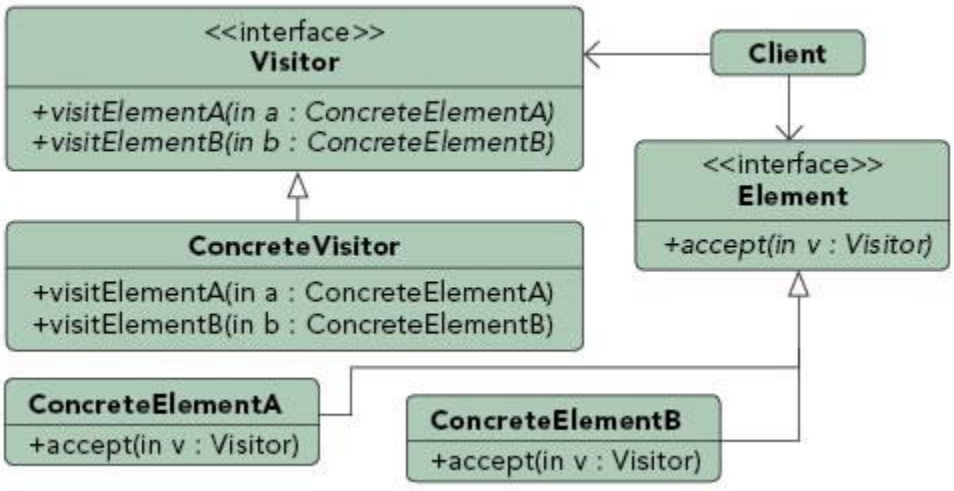
- The behavior of an object should be influenced by its state.
- Complex conditions tie object behavior to its state.
- Transitions between states need to be explicit.
- The state design pattern is great when we want to **avoid primitive if/else statements**. Instead, we **extract the logic to separate classes** and let our **context object delegate the behavior** to the methods implemented in the state class.
- Besides, we can leverage the transitions between the states, where one state can alter the state of the context.

8. Visitor Pattern:

- **Purpose:** Allows for one or more operations to be applied to a set of objects at runtime, decoupling the operations from the object structure.
- Visitor pattern is used when we have to perform an operation on a group of similar kind of Objects.
- With the help of visitor pattern, we can move the operational logic from the objects to another class.
- Visitor pattern defines a new operation without introducing the modifications to an existing object structure.
- In Visitor pattern, we use a **visitor** class (that adds a function) which changes the executing algorithm of an element class. By this way, execution algorithm of element can vary as and when visitor varies.
- As per the pattern, element object has to accept the visitor object so that visitor object handles the operation on the element object.
- In other words, we'll extract the algorithm which will be applied to the object structure from the classes.
- Consequently, we'll make good use of the Open/Closed principle as we won't modify the code, but we'll still be able to extend the functionality by providing a new Visitor implementation.

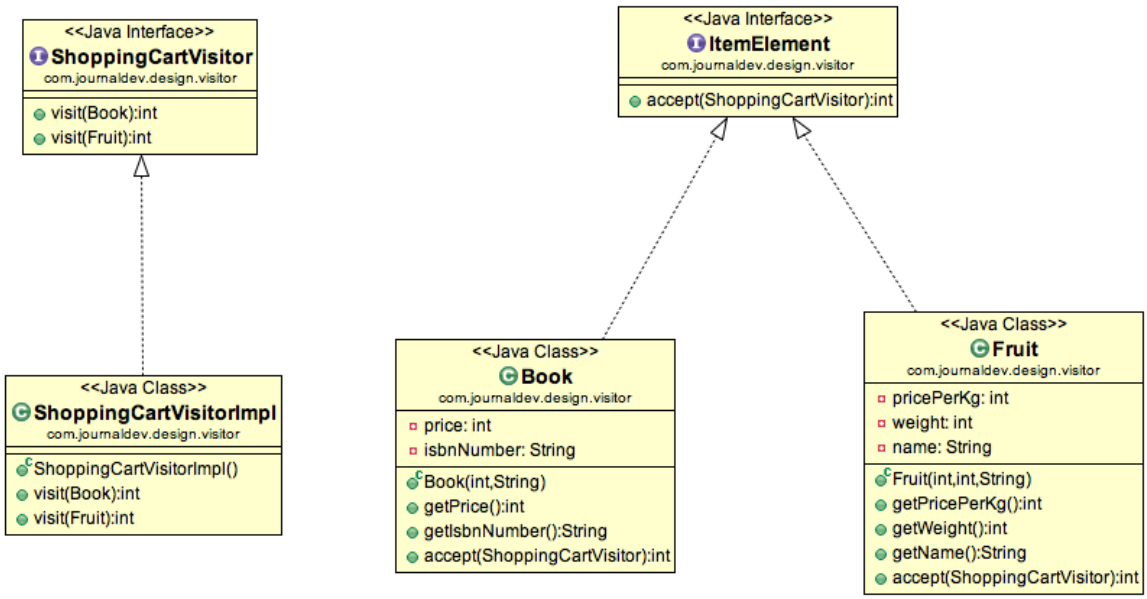
Real-Life Example:

Calculating taxes in different regions on sets of invoices would require many different variations of calculation logic. Implementing a visitor allows the logic to be decoupled from the invoices and line items. This allows the hierarchy of items to be visited by calculation code that can then apply the proper rates for the region. Changing regions is as simple as substituting a different visitor.



Example:

Create a Shopping cart where we can add different type of items (Elements). When we click on checkout button, it calculates the total amount to be paid. Now we can have the calculation logic in item classes or we can move out this logic to another class using visitor pattern. Let's implement this in our example of visitor pattern.



To implement visitor pattern, first of all we will create different type of items (Elements) to be used in shopping cart.

```
public interface ItemElement {

	public int accept(ShoppingCartVisitor visitor);
}
```

Note: accept() method takes Visitor argument. We can have some other methods also specific for items but for simplicity I am not going into that much detail and focusing on visitor pattern only.

Create some concrete classes for different types of items (Book, Fruit).

```
public class Book implements ItemElement {

	private int price;
	private String isbnNumber;

	public Book(int cost, String isbn) {
		this.price = cost;
		this.isbnNumber = isbn;
	}

	public int getPrice() {
```

```
            return price;
      }

      public String getIsbnNumber() {
            return isbnNumber;
      }

      @Override
      public int accept(ShoppingCartVisitor visitor) {
            return visitor.visit(this);
      }
}


public class Fruit implements ItemElement {

      private int pricePerKg;
      private int weight;
      private String name;

      public Fruit(int pricePerKg, int weight, String name) {
            this.pricePerKg = pricePerKg;
            this.weight = weight;
            this.name = name;
      }

      public int getPricePerKg() {
            return pricePerKg;
      }

      public int getWeight() {
            return weight;
      }

      public String getName() {
            return name;
      }

      @Override
      public int accept(ShoppingCartVisitor visitor) {
            return visitor.visit(this);
      }
}
```

Note: The implementation of accept() method in concrete classes, its calling visit() method of Visitor and passing itself as argument.

Create visitor interface:

```
public interface ShoppingCartVisitor {

	int visit(Book book);
	int visit(Fruit fruit);
}
```

Note: visit() method for different type of items in Visitor interface that will be implemented by concrete visitor class.

Implement visitor interface and every item will have its own logic to calculate the cost.

```
public class ShoppingCartVisitorImpl implements ShoppingCartVisitor {

	@Override
	public int visit(Book book) {
		int cost = 0;

		//apply 5$ discount if book price is greater than 50
		if(book.getPrice() > 50)
			cost = book.getPrice() - 5;
		else
			cost = book.getPrice();

		System.out.println("Book ISBN :: " + book.getIsbnNumber() + ",
		cost = " + cost);

		return cost;
	}

	@Override
	public int visit(Fruit fruit) {

		int cost = fruit.getPricePerKg() * fruit.getWeight();
		System.out.println("Fruit :: " + fruit.getName() + ", cost = " +
		cost);

		return cost;
	}
}
```

Output:

```
Book ISBN :: 1256, cost = 90
Book ISBN :: 1588, cost = 95
Fruit :: Banana, cost = 20
Fruit :: Apple, cost = 40
Total Cost : 245
```

Benefits of Visitor Pattern:

- The benefit of this pattern is that if the logic of operation changes, then we need to make change only in the visitor implementation rather than doing it in all the item classes.
- Another benefit is that adding a new item to the system is easy, it will require change only in visitor interface and implementation and existing item classes will not be affected.

Limitations of Visitor Pattern:

- The drawback of visitor pattern is that we should know the return type of visit() methods at the time of designing otherwise we will have to change the interface and all of its implementations.
- Another drawback is that if there are too many implementations of visitor interface, it makes it hard to extend.
- When using this pattern, the business logic related to one particular object gets spread over all visitor implementations.

When to use Visitor Pattern:

- An object structure must have many unrelated operations performed upon it.
- The object structure can't change but operations performed on it can.
- Operations must be performed on the concrete classes of an object structure.
- Exposing internal state or operations of the object structure is acceptable.
- Operations should be able to operate on multiple object structures that implement the same interface sets.
- The Visitor pattern is great to separate the algorithm from the classes on which it operates. Besides that, it makes adding new operation more easily, just by providing a new implementation of the Visitor.
- Furthermore, we don’t depend on components interfaces, and if they are different, that’s fine, since we have a separate algorithm for processing per concrete element.
- Moreover, the Visitor can eventually aggregate data based on the element it traverses.

9. **Interpreter Pattern:**

- **Purpose:** Defines a representation for a grammar as well as a mechanism to understand and act upon the grammar.
- It is used to define a grammatical representation for a language and provides an interpreter to deal with this grammar.
- 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** etc.
- In short, the pattern defines the grammar of a particular language in an object-oriented way which can be evaluated by the interpreter itself.
- Having that in mind, technically we could build our custom regular expression, a custom DSL interpreter or we could parse any of the human languages, build abstract syntax trees and then run the interpretation.
- These are only some of the potential use cases, but if we think for a while, we could find even more usages of it, for example in our IDEs, since they're continually interpreting the code we're writing and thus supplying us with priceless hints.
- The interpreter pattern generally should be used when the grammar is relatively simple.

Interpreter Pattern usage in JDK:

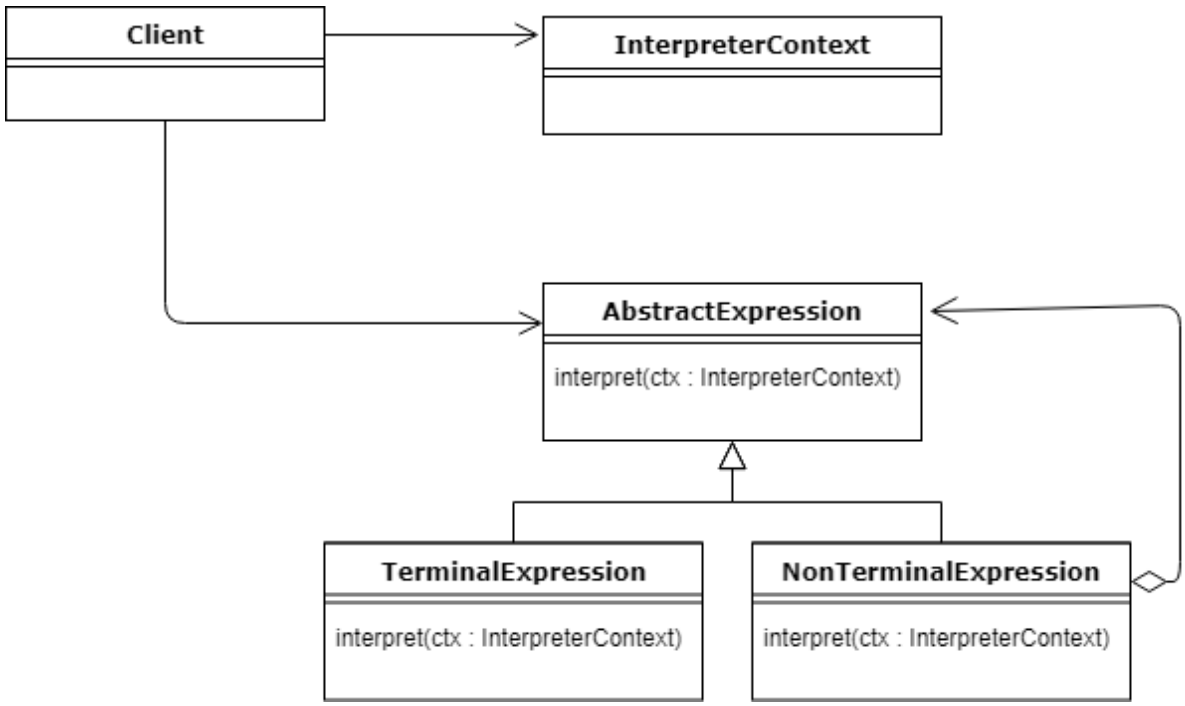
- *java.util.Pattern*
- *java.text.Format* or *java.text.Normalizer*.

Real-Life Example:

Text based adventures, wildly popular in the 1980's, provide a good example of this. Many had simple commands, such as "step down" that allowed traversal of the game. These commands could be nested such that it altered their meaning. For example, "go in" would result in a different outcome than "go up". By creating a hierarchy of commands based upon the command and the qualifier (non-terminal and terminal expressions) the application could easily map many command variations to a relating tree of actions.

The best example of interpreter design pattern is java compiler that interprets the java source code into byte code that is understandable by JVM.

Google Translator is also an example of interpreter pattern where the input can be in any language and we can get the output interpreted in another language.



Above diagram shows two main entities: the ***Context*** and the ***Expression***.

Now, any language needs to be expressed in some way, and the words (expressions) are going to have some meaning based on the given context.

AbstractExpression defines one abstract method which takes the context as a parameter. Thanks to that, **each expression will affect the context**, change its state and either continue the interpretation or return the result itself.

Therefore, the context is going to be the holder of the global state of processing, and it's going to be reused during the whole interpretation process.

So **what's the difference between the *TerminalExpression* and *NonTerminalExpression*?**

A ***NonTerminalExpression*** may have one or more other ***AbstractExpressions*** associated in it, therefore it can be recursively interpreted. In the end, **the process of interpretation has to finish with** *a TerminalExpression* **that will return the result.**

Note: *NonTerminalExpression* is a composite**.**

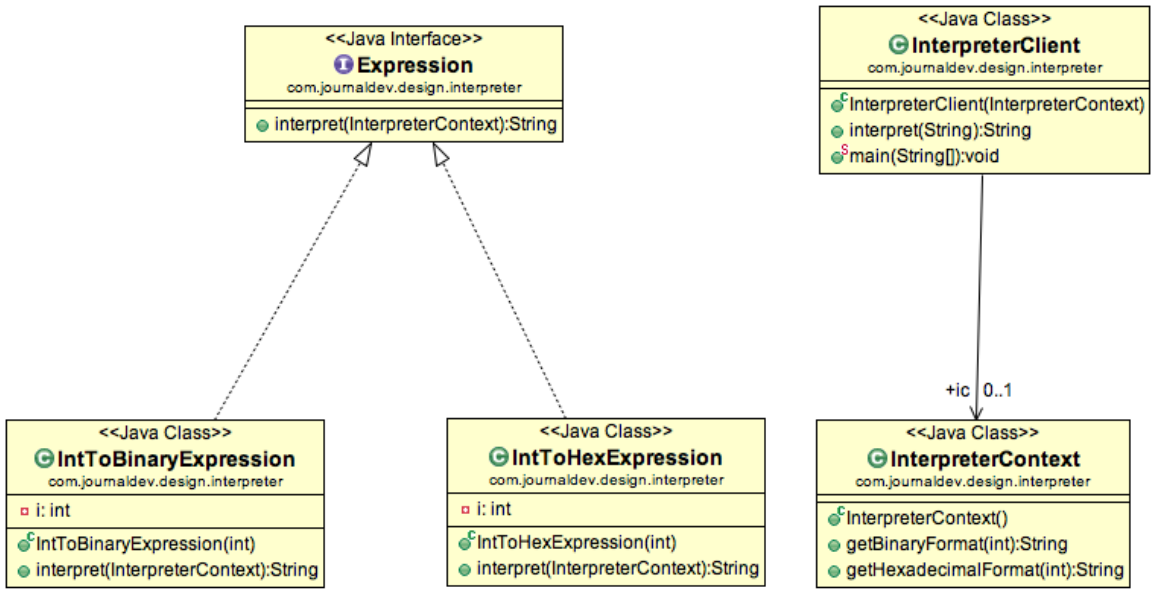
Finally, the role of the client is to create or use an already created **abstract syntax tree**, which is nothing more than a **sentence defined in the created language.**

Example:

To implement interpreter pattern, we need to create Interpreter context engine that will do the interpretation work. Then we need to create different Expression implementations that will consume the functionalities provided by the interpreter context. Finally we need to create the client that will take the input from user and decide which Expression to use and then generate output for the user.

Let's understand this with an example where the user input will be of two forms –

1. "<Number> in Binary" or "<Number> in Hexadecimal." Our interpreter client should return it in format "<Number> in Binary= <Number_Binary_String>"
2. "<Number> in Hexadecimal= <Number_Binary_String>" respectively.



Interpreter context class that will do the actual interpretation.

```
public class InterpreterContext {

	public String getBinaryFormat(int i) {
		return Integer.toBinaryString(i);
	}

	public String getHexadecimalFormat(int i) {
		return Integer.toHexString(i);
	}
}
```

Create different types of Expressions that will consume the interpreter context class.

```
public interface Expression {
	String interpret(InterpreterContext ic);

}
```

Expression implementations, one to convert int to binary and other to convert int to hexadecimal format.

```
public class IntToBinaryExpression implements Expression {

	private int i;

	public IntToBinaryExpression(int number) {
		this.i = number;
	}

	@Override
	public String interpret(InterpreterContext ic) {
		return ic.getBinaryFormat(i);
	}
}
```

```
public class IntToHexExpression implements Expression {

	private int i;

	public IntToHexExpression(int number) {
		this.i = number;
	}

	@Override
	public String interpret(InterpreterContext ic) {
		return ic.getHexadecimalFormat(i);
	}

}
```

Client application that will have the logic to parse the user input and pass it to correct expression and then use the output to generate the user response.

```
public class InterpreterClient {


	public InterpreterContext ic;

	public InterpreterClient(InterpreterContext ic) {
		this.ic = ic;
	}

	public String interpret(String str) {
		Expression exp;

		//Create rules for Expression
		if(str.contains("Hexadecimal")) {
			exp = new
			IntToHexExpression(Integer.parseInt(str.substring(0,
			str.indexOf(" "))));

		}
```

```
            else if(str.contains("Binary")) {
                  exp = new
                  IntToBinaryExpression(Integer.parseInt(str.substring(0,str.
                  indexOf(" "))));

            } else {
                  return str;
            }
            return exp.interpret(ic);
      }

      public static void main(String args[]){
            String str1 = "28 in Binary";
            String str2 = "28 in Hexadecimal";

            InterpreterClient ec = new InterpreterClient(new
            InterpreterContext());
            System.out.println(str1 + " = " + ec.interpret(str1));
            System.out.println(str2 + " = " + ec.interpret(str2));

      }

}
```

Output:

```
28 in Binary = 11100
28 in Hexadecimal = 1c
```

Important Points:

- Interpreter pattern can be used when we can create a syntax tree for the grammar we have.
- Interpreter design pattern requires a lot of error checking and a lot of expressions and code to evaluate them. It gets complicated when the grammar becomes more complicated and hence hard to maintain and provide efficiency.

Drawbacks of Interpreter Pattern:

- When the grammar is getting more complex, it becomes harder to maintain.
- It can be seen in the presented example. It'd be reasonably easy to add another expression, yet it won't be too easy to maintain if we'd decide to keep extending it with all other expressions.

When to use Interpreter Pattern:

- The interpreter design pattern is great **for relatively simple grammar interpretation**, which doesn't need to evolve and extend much.
- There is grammar to interpret that can be represented as large syntax trees.
- The grammar is simple.
- Efficiency is not important.
- Decoupling grammar from underlying expressions is desired.

10. Iterator Pattern:

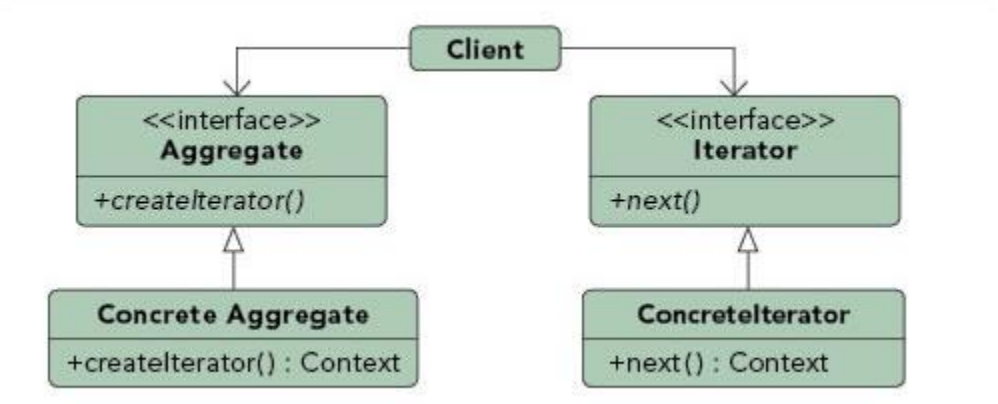
- **Purpose**: Allows for access to the elements of an aggregate object without allowing access to its underlying representation.
- Used to provide a standard way to traverse through a group of Objects.
- Iterator pattern is widely **used in Java Collection Framework** where Iterator interface provides methods for traversing through a collection.
- Provides a way to access the elements of an aggregate object without exposing its underlying representation.
- Iterator pattern is not only about traversing through a collection, we can provide different kind of iterators based on our requirements.
- Iterator design pattern hides the actual implementation of traversal through the collection and client programs just use iterator methods.
- This pattern is used to get a way to access the elements of a collection object in sequential manner without any need to know its underlying representation.

Iterator Design Pattern in JDK

- Collection framework Iterator is the best example of iterator pattern implementation.
- java.util.Scanner class also Implements Iterator interface.

Real-Life Example:

The Java implementation of the iterator pattern allows users to traverse various types of data sets without worrying about the underlying implementation of the collection. Since clients simply interact with the iterator interface, collections are left to define the appropriate iterator for themselves. Some will allow full access to the underlying data set while others may restrict certain functionalities, such as removing items.



Example:

Suppose we have a list of Radio channels and the client program want to traverse through them one by one or based on the type of channel. For example some client programs are only interested in English channels and want to process only them, they don't want to process other types of channels.

So we can provide a collection of channels to the client and let them write the logic to traverse through the channels and decide whether to process them. But this solution has lots of issues such as client has to come up with the logic for traversal. We can't make sure that client logic is correct. Furthermore if the number of client grows then it will become very hard to maintain.

Here we can use Iterator pattern and provide iteration based on type of channel. We should make sure that client program can access the list of channels only through the iterator.

Define the contract for collection and iterator interfaces.
java enum that defines all the different types of channels.

```
public enum ChannelTypeEnum {

	ENGLISH, HINDI, FRENCH, ALL;
}
```

Simple POJO class that has attributes frequency and channel type.

```
public class Channel {

	private double frequency;
	private ChannelTypeEnum TYPE;

	public Channel(double frequency, ChannelTypeEnum type) {
		this.frequency = frequency;
		this.TYPE = type;
	}

	public double getFrequency() {
		return frequency;
	}

	public ChannelTypeEnum getTYPE() {
		return TYPE;
	}

	@Override
	public String toString() {
		return "Channel [frequency=" + frequency + ", TYPE=" + TYPE +
		"]";
	}
}
```

Interface that defines the contract for our collection class implementation.

```
public interface ChannelCollection {

	public void addChannel(Channel c);

	public void removeChannel(Channel c);

	public ChannelIterator iterator(ChannelTypeEnum type);
}
```

Note: There are methods to add and remove a channel but there is no method that returns the list of channels. ChannelCollection has a method that returns the iterator for traversal.

ChannelIterator interface defines following methods;

```
public interface ChannelIterator {

	boolean hasNext();
	Channel next();
}
```

Implementation of collection class and iterator.

```
public class ChannelCollectionImpl implements ChannelCollection {

	private List<Channel> channelsList;

	public ChannelCollectionImpl() {
		channelsList = new ArrayList<>();
	}

	public void addChannel(Channel c) {
		this.channelsList.add(c);
	}

	public void removeChannel(Channel c) {
		this.channelsList.remove(c);
	}

	@Override
	public ChannelIterator iterator(ChannelTypeEnum type) {
		return new ChannelIteratorImpl(type, this.channelsList);
	}

	private class ChannelIteratorImpl implements ChannelIterator {

		private ChannelTypeEnum type;
		private List<Channel> channels;
		private int position;

		public ChannelIteratorImpl(ChannelTypeEnum type,
				List<Channel> channelsList) {
			this.type = type;
			this.channels = channelsList;
		}
```

```
		@Override
		public boolean hasNext() {
			while (position < channels.size()) {
				Channel c = channels.get(position);
				if (c.getTYPE().equals(type) ||
				type.equals(ChannelTypeEnum.ALL)) {
					return true;
				} else {
					position++;
				}
			}
			return false;
		}

		@Override
		public Channel next() {
			Channel c = channels.get(position);
			position++;
			return c;
		}

	}
}
```

Note: The **inner class** implementation of iterator interface so that the implementation can't be used by any other collection. Same approach is followed by collection classes also and all of them have inner class implementation of Iterator interface.

Program to use our collection and iterator to traverse through the collection of channels.

```
public class IteratorPattern {

	public static void main(String[] args) {

		ChannelCollection channels = populateChannels();

		ChannelIterator baseIterator =
		channels.iterator(ChannelTypeEnum.ALL);

		while (baseIterator.hasNext()) {
			Channel c = baseIterator.next();
			System.out.println(c.toString());
		}

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

		// Channel Type Iterator
		ChannelIterator englishIterator =
		channels.iterator(ChannelTypeEnum.ENGLISH);

		while (englishIterator.hasNext()) {
			Channel c = englishIterator.next();
			System.out.println(c.toString());
		}
	}
```

```
	private static ChannelCollection populateChannels() {
		ChannelCollection channels = new ChannelCollectionImpl();
		channels.addChannel(new Channel(98.5, ChannelTypeEnum.ENGLISH));
		channels.addChannel(new Channel(99.5, ChannelTypeEnum.HINDI));
		channels.addChannel(new Channel(100.5, ChannelTypeEnum.FRENCH));
		channels.addChannel(new Channel(101.5, ChannelTypeEnum.ENGLISH));
		channels.addChannel(new Channel(102.5, ChannelTypeEnum.HINDI));
		channels.addChannel(new Channel(103.5, ChannelTypeEnum.FRENCH));
		channels.addChannel(new Channel(104.5, ChannelTypeEnum.ENGLISH));
		channels.addChannel(new Channel(105.5, ChannelTypeEnum.HINDI));
		channels.addChannel(new Channel(106.5, ChannelTypeEnum.FRENCH));
		return channels;
	}
}
```

Important Points:

- Iterator pattern is useful when you want to provide a standard way to iterate over a collection and hide the implementation logic from client program.
- The logic for iteration is embedded in the collection itself and it helps client program to iterate over them easily.

When to use Iterator Pattern:

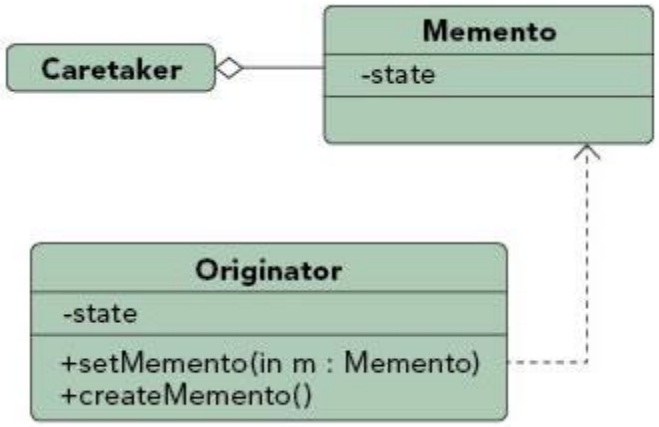
- Access to elements is needed without access to the entire representation.
- Multiple or concurrent traversals of the elements are needed.
- A uniform interface for traversal is needed.
- Subtle differences exist between the implementation details of various iterators.

11. **Memento Pattern:**

- **Purpose:** Allows for capturing and externalizing an object's internal state so that it can be restored later, all without violating encapsulation.
- Memento design pattern is used when we want to save the state of an object so that we can restore later on.
- Memento pattern is used to implement this in such a way that the saved state data of the object is not accessible outside of the object, this protects the integrity of saved state data.
- Memento pattern is implemented with two objects – **Originator** and **Caretaker**.
- **Originator** is the object whose state needs to be saved and restored and it uses an inner class to save the state of Object. The inner class is called Memento and its private, so that it can't be accessed from other objects.
- **Caretaker** is the helper class that is responsible for storing and restoring the Originator's state through Memento object**. Since Memento is private to Originator, Caretaker can't access it and it's stored as an Object within the caretaker**.

Real-Life Examples:

One of the best real life example is the **text editors** where we can save its data anytime and use undo to restore it to previous saved state. **Undo functionality** can nicely be implemented using the memento pattern. By serializing and deserializing the state of an object before the change occurs we can preserve a snapshot of it that can later be restored should the user choose to undo the operation.



Example:
Implement the text editor feature and provide a utility where we can write and save contents to a File anytime and we can restore it to last saved state.

Memento Pattern Originator Class

```
public class FileWriterUtil {

	private String fileName;
	private StringBuilder content;

	public FileWriterUtil(String fileName) {
		this.fileName = fileName;
		this.content = new StringBuilder();
	}

	@Override
	public String toString() {
		return this.content.toString();
	}

	public void write(String str) {
		content.append(str);
	}

	public Memento save() {
		return new Memento(this.fileName, this.content);
	}

	public void undoToLastSave(Object obj) {
		Memento memento = (Memento) obj;
		this.fileName = memento.fileName;
		this.content = memento.content;
	}

	private class Memento {

		private String fileName;
		private StringBuilder content;

		public Memento(String fileName, StringBuilder content) {
			this.fileName = fileName;
			//notice the deep copy so that Memento and FileWriterUtil
content variables don't refer to same object
			this.content = new StringBuilder(content);
		}
	}
}
```

Note: The Memento inner class and implementation of save and undo methods. Now we can continue to implement Caretaker class.

Memento Pattern Caretaker Class:

```
public class FileWriterCaretaker {

	private Object obj;

	public void save(FileWriterUtil fileWriter) {
		this.obj = fileWriter.save();
	}

	public void undo(FileWriterUtil fileWriter) {
		fileWriter.undoToLastSave(obj);
	}
}
```

Note: Caretaker object contains the saved state in the form of Object, so it can't alter its data and also it has no knowledge of its structure.

Memento Pattern Class:

```
public class FileWriterClient {

	public static void main(String[] args) {

		FileWriterCaretaker caretaker = new FileWriterCaretaker();

		FileWriterUtil fileWriter = new FileWriterUtil("data.txt");
		fileWriter.write("First Set of Data.\n");
		System.out.println(fileWriter);

		//Save the file
		caretaker.save(fileWriter);

		//Write another set of data
		fileWriter.write("Second Set of Data.\n");

		//Check file Contents
		System.out.println(fileWriter);

		//Undo to last save
		caretaker.undo(fileWriter);

		System.out.println(fileWriter);
	}
}
```

Output:

First Set of Data.

First Set of Data.

Second Set of Data.

First Set of Data.

Important Points:

- Memento pattern is simple and easy to implement, one of the thing needs to take care is that Memento class should be accessible only to the Originator object. Also in client application, we should use caretaker object for saving and restoring the originator state.

- Also if Originator object has properties that are not immutable, we should use deep copy or cloning to avoid data integrity issue.

- Serialization can be used to achieve memento pattern implementation that is more generic rather than Memento pattern where every object needs to have its own Memento class implementation.

Drawbacks of Memento Pattern:

- If Originator object is very huge then Memento object size will also be huge and use a lot of memory.

When to use Memento Pattern:

- The internal state of an object must be saved and restored at a later time.
- Internal state cannot be exposed by interfaces without exposing implementation.
- Encapsulation boundaries must be preserved.