# Design Patterns

Design_Patterns_Glossary.pdf

1. “spaghetti code” -- for example, source code that has no structure or tangled program flow.
2. UML diagram is important. Learn it. In UML:
   italic means abstract
   + means public
   - means private
   + means protected
   0..* means any number 0 or more than 0

## Creational Patterns

### Singleton Pattern

```
public class Example {

    private static Example unique = null;

    private Example() {}

    public static Example getInstance() {
        if(unique == null) {
            unique = new Example();
        }
        return unique;
    }
}
```

1. Object creation is lazy
2. Need to keep in mind about using in multi threading program
3. Example- preferences object of an app, db clients

---

### Factory object/ factory MEthod Pattern

#### Factory Object pattern

```
public class KnifeFactory {
    public Knife createKnife(String type) {
        Knife knife = null;

        if(type.equals("steak")) {
            knife = new SteakKnife(); // these are sublclasses of Knife
        }
        else if () {
            ...
        }
```

```
        return knife;
    }
}



public class KnifeStore {
    private KnifeFactory factory;

    private KnifeStore(KnifeFactory factory) {
        this.factory = factory;
    }

    public Knife orderKnife(String type) {

        Knife knife = factory.createKnife(type); // generalization

        knife.sharpen();
        // .. other method

        return knife;

    }
}
```

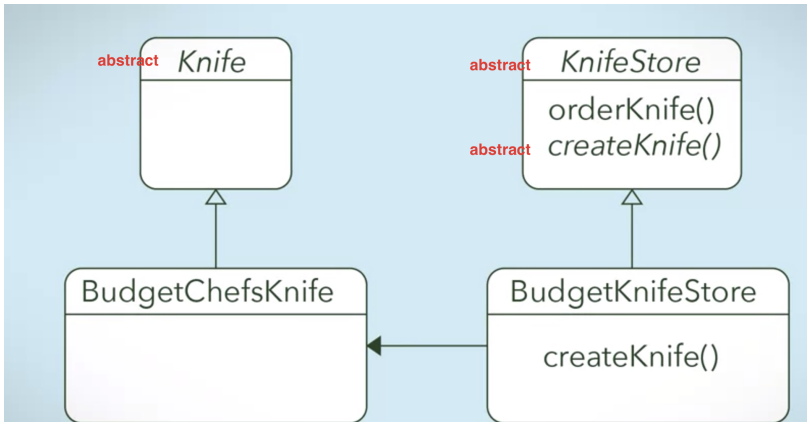
1. Concrete Instantiation - using new operator
2. Now other clients can use the same factory.
3. Coding to an interface and not implementation. KnifeStore only knows about Knife and doesn't care about its subclasses - SteakKnife, KitchenKnife etc.

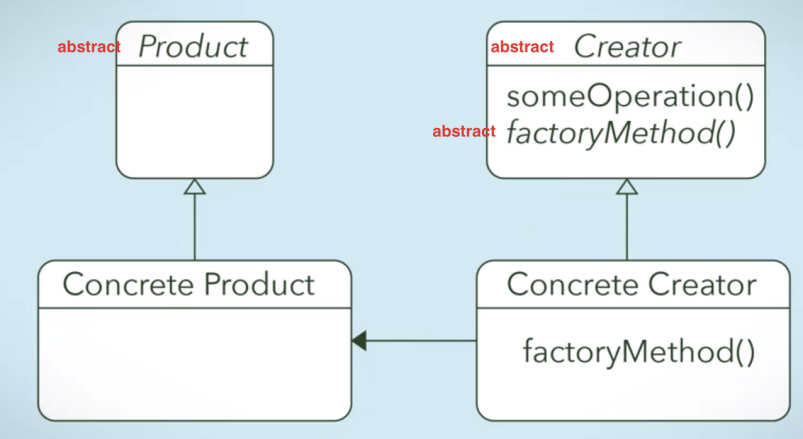
**Factory Method Pattern.**

- instead of using a separate factory object to create knives, factory method uses a separate method to create knives
- The factory method design intent is to define interface for creating objects but let the subclasses decide which class to instansiate.

```
public abstract class KnifeStore {


    public Knife orderKnife(String type) {

        Knife knife = createKnife(type); // each subclass will define this

        knife.sharpen();
        // .. other method

        return knife;

    }
```

```
    abstract Knife createKnife(String type);
}


public class BudgetKnifeStore extends KnifeStore {

    Knife createKnife(String type) {
        if(type.equals("steak")) {
            return new BudgetSteakKnife();
        }
        else if (type.equals("chefs")) {
            return new BudgetChefsKnife();
        }
        else
            return null;
    }

    // don't forget orderKnife method is also inherited
}

public class QualityKnifeStore extends KnifeStore {

    Knife createKnife(String type) {
        if(type.equals("steak")) {
            return new QualitySteakKnife();
        }
        else if (type.equals("chefs")) {
            return new QualityChefsKnife();
        }
        else
            return null;
    }

    // don't forget orderKnife method is also inherited
}
```






italic variables are abstract in pic.

---

## Structural Patterns

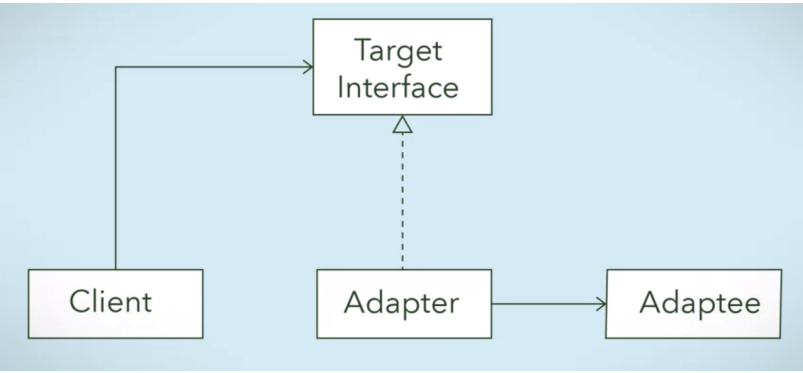
### FACADE PATTERN

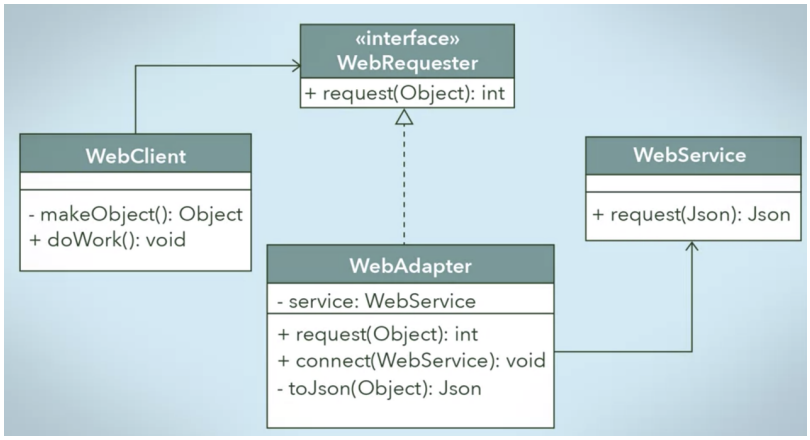
- Adding an extra layer of abstraction. Provide client classes with a simplified interface for the subsystem. Acts only as entry point to subsystem. Doesn't add anything.

---

#### ADAPTER PATTERN

- The output of one system may not conform to the expected input of another system.
- Adapter is basically a wrapper over the Adaptee.





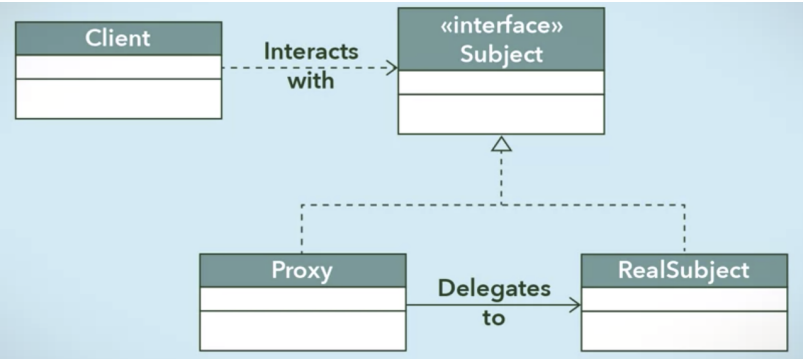

```
public interface WebRequester {

    public int request(Object);
}


public class WebAdapter implements WebRequester {
    private WebService service;

    public void connect(WebService service) {
        this.service = service;
    }

    public int request(Object object) {

        Json json = toJson(object);

        return service.request(json);

    }

    private Json toJson(Object object) {
        ...
    }
}


public class WebClient {
    private WebRequester webRequester;

    public WebClient(WebRequester webRequester // adapter interface) {
        this.webRequester = webRequester;
    }

    public void doWork() {
        Object object = makeObject();

        int x = webRequester.request(object);
    }
}
```

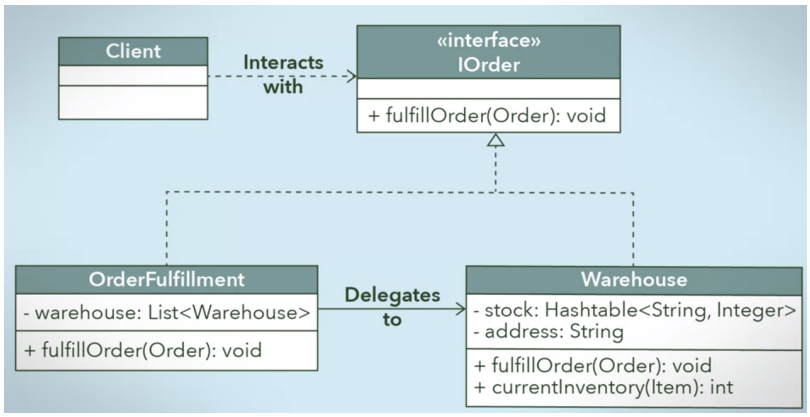
```
public class Main {
     public static void main(String[] args) {


          WebService service = new WebService();
          WebAdapter adapter = new WebAdapter();

          adapter.connect(service);


          new WebClient(adapter).doWork(); // webclient doesn't know anything about web


     }
}
```

---

## PROXY PATTERN

- Proxy class wraps the real class i.e. hides a reference to an instance of the real class.
- Real class may contain sensitive info or may be resource intensive to instantiate.
- Client classes will interact with proxy.
- Uses-
    - Not instantiating real class in all cases.
    - To act as a protection proxy in order to control access to real subject class. Ex- in authorising different users; So that they can only access functions permitted by their role.
    - When real class is remote, proxy is local. Ex- Google doc
    - Some sort of verification of requests to decide if requests need to be forwarded and to whom






```
// Routing orders to warehouse only if the warehouse has that order.


public IOrder {
    public void fulfillOrder(Order);
}


public class Warehouse implements IOrder {

    private HashMap<String, Integer> stock;

    public void fulfillOrder(Order order) {

        for(Item item: order.items)
            // subtract one from stock

        // ship and deliver
    }


    public int currentInventory(Item item) {
        if(stock.contains(item.name))
            return stock.get(item);
        return 0;
    }
}


// proxy
public class OrderFulfillment implements IOrder {
    private List<Warehouse> warehouses;

    public void fulfillOrder(Order order) {

        // an order will be sent to Warehouse only if it is stock
        for(Item item: order.items) {
            for(Warehouse warehouse: warehouses) {
                if(warehouse.currentInventory(item) > 0)
                    // give order to this warehouse for this item
```

```
            }
        }
    }

    // other functions like priortizing certain orders can be in this class
}
```

---

#### DECORATOR PATTERN

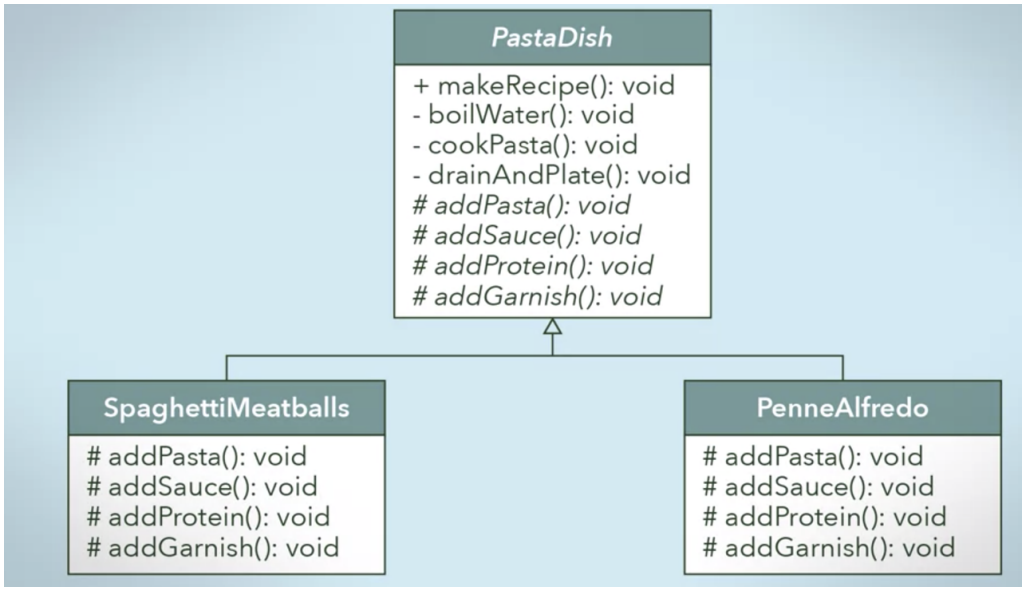
fdf

---

## Behavioural Design Patterns

#### TEMPLATE METHOD PATTERN

- Used when you can generalize between two classes into a new superclass.
- When you have two separate classes with very similar functionality in order of operations.
- It's just a application of generalisation and inheritance
- final method in java cannot be overriden by subclass




```
public abstract class PastaDish {
    public final void makeReciepe() { // this can't be overriden
        boilWater();
```

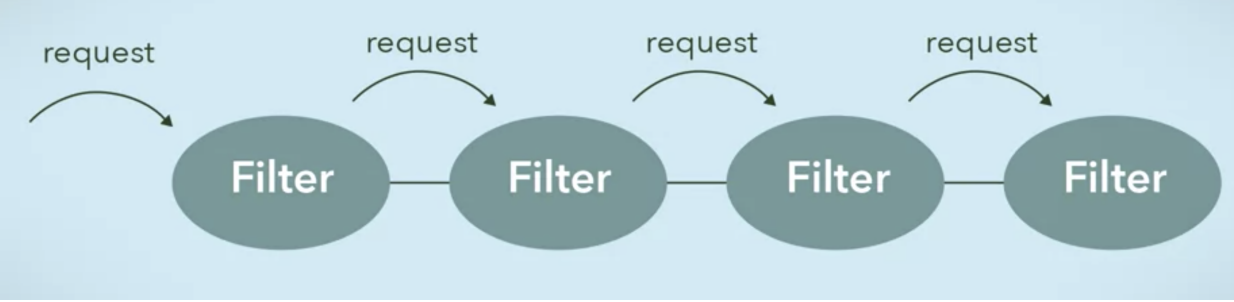
```
        addPasta();
        addProtein();
        cookPasta();
    }

    protected abstract void addPasta();
    protected abstract void addProtein();  // each subclass will override this
}
```

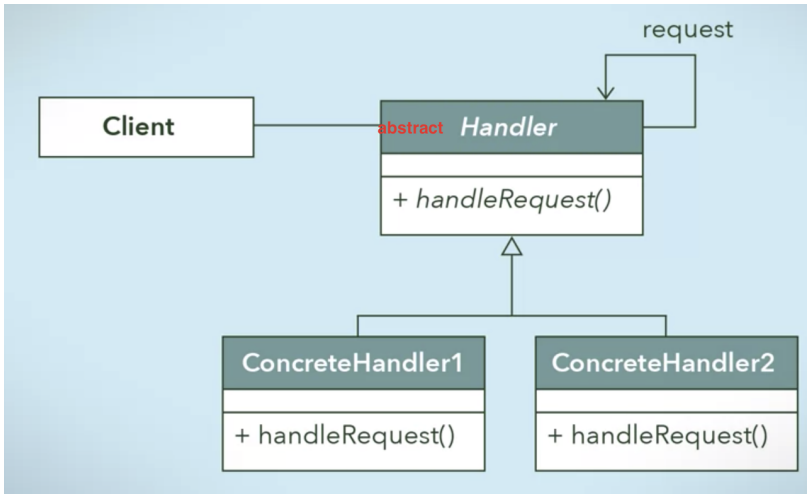
---

**CHAIN OF RESPONSIBILTY**

- Having diff handlers in a series. Request first comes to first handler, if it is unable to process then it is passed to next handler, and so on. Ex- Like you trying different screwdrivers for a screw
- Ex- Exception handling in Java, exceptions over different layers
- Ex- Different filters on an email
- Problem- If one handler fails, request doesn't propagate further in chain. We need to make sure requests are handled in similar fashion in all handlers.
  Check if it can be handled.
  If yes, then do somethings
  else, pass to next handler

  Template pattern can be used to achieve this.
- It avoids coupling with sender and receiver. Receiver doesn't need to know about all handlers.

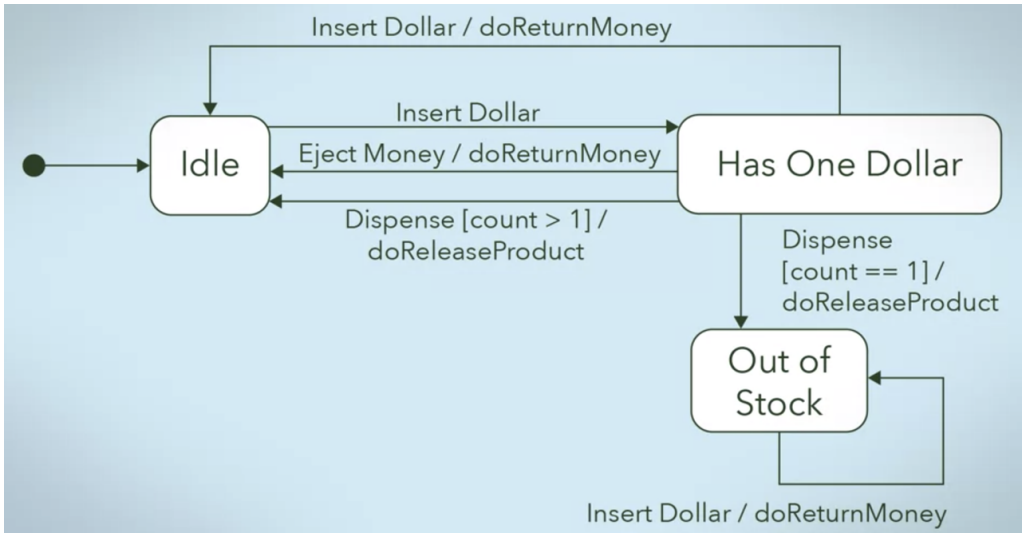





---

**STATE PATTERN**

- Used when you need to change the behaviour of an object based upon the state that it is in at run time. Ex- a vending machine represents a state machine
- Can be used to avoid long if else statements; Using ifs in utility methods is fine.
- Branching over a "type code" is a code smell
- https://stackoverflow.com/questions/1554180/why-is-the-if-statement-considered-evil

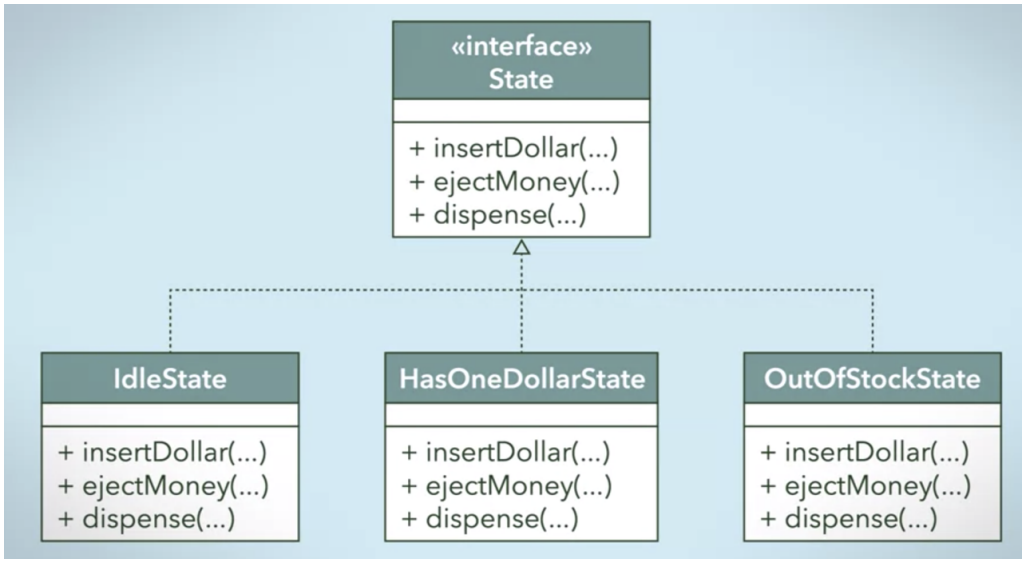



```
States - Idle, hasOneDollar, outOfStock
Events/Triggers - Insert dollar, eject money, dispense
Actions - doReleaseProduct, doReturnMoney
```

Bad code:

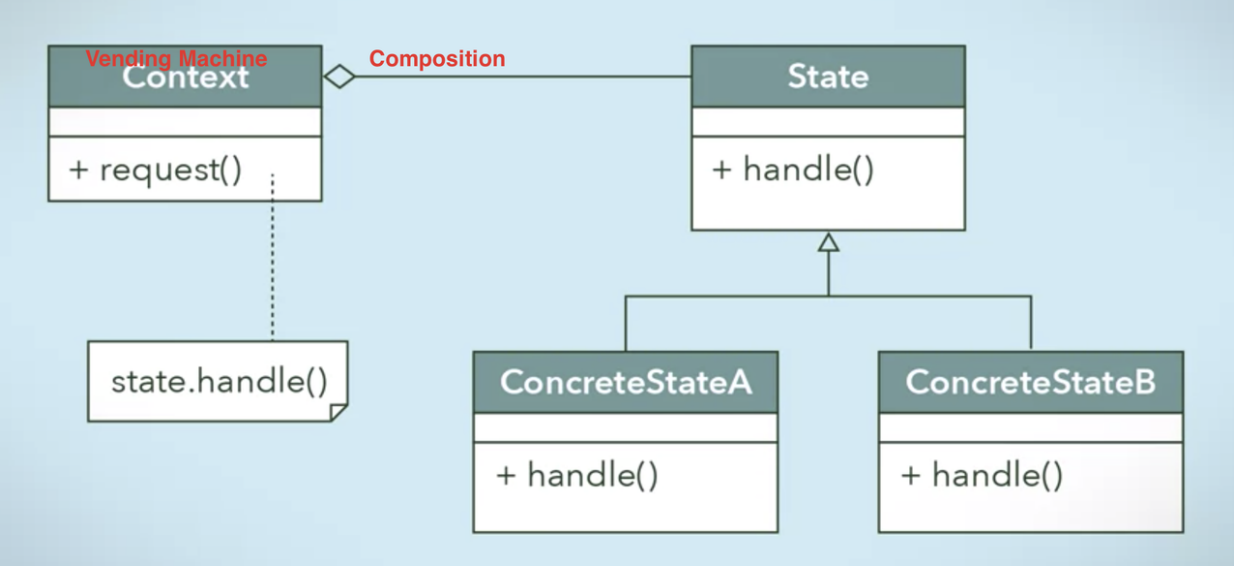
```
final class State { // singleton object for states; use enum for these

    private State() {}

    public final static State Idle = new State();
    public final static State HasOneDollar = new State();
    public final static State OutOfStock = new State();
}



public class VendingMachine {
    private State currentState;
    private int count;

    public VendingMachine(int count) {
        if(count > 0) {
            currentState = State.Idle;
            this.count = count;
        }
        else {
            currentState = State.OutOfStock;
            this.count = 0;
        }
    }


    public void insertDollar() {
        if(currentState == State.Idle) {
            currentState = State.HasOneDollar;
        }
        else if(currentState == State.HasOneDollar) {
            doReturnMoney();
            currentState = State.Idle;
        }
        else if(currentState == State.OutOfStock) {
            doReturnMoney();
            // state stays same
        }
    }
}
```

Good Code using State pattern:




```
public interface State {
    public void insertDollar(VendingMachine VendingMachine);
    public void ejectMoney(VendingMachine VendingMachine);
    public void dispense(VendingMachine VendingMachine);

}

public class IdleState implements State {
    public void insertDollar(VendingMachine vendingMachine) {
        // dollar  inserted

        vendingMachine.setState(vendingMachine.getHasOneDollarState()); // this is so
    }

    public void ejectMoney(VendingMachine vendingMachine) {
        // no money to return
    }

    public void dispense(VendingMachine vendingMachine) {
        // payment required to dispense
    }
}

// Similarly other state classes are implemented

public class HasOneDollar implements State {
    ...

    public void dispense(VendingMachine vendingMachine) {
        // releasing product

        vendingMachine.doReleaseProduct();
        if(vendingMachine.getCount() > 1) {
            vendingMachine.setState(vendingMachine.getIdleState());
        }
```

```
        else {
            vendingMachine.setState(vendingMachine.getOutOfStockState());
        }
    }
}


public class VendingMachine {
    private State idleState = new IdleState();
    private State hasOneDollarState = new HasOneDollarState();
    private State outOfStockState = new outOfStockState();

    private State currentState;
    private int count;

    public VendingMachine(int count) {


        if(count > 0) {
            currentState = idleState;
            this.count = count;
        }
        else {
            currentState = outOfStockState;
            this.count = 0;
        }
    }

    public void insertDollar() {
        currentState.insertDollar(this);
    }

    public void ejectMoney() {
        currentState.ejectMoney(this);
    }
    public void dispense() {
        currentState.dispense(this);
    }

}
```




---

**COMMAND PATTERN**

- Ex- A boss writes tasks on a memo and asks his secretary to delegate these tasks to his directs.
- Encapsulates the request as an object of its own.
- Invoker is used to invoke the command object. Command manager can also be used to keep track of the commands. Ex- secretary is the invoker
- Uses-
    - Store and schedule different requests.
    - Command objects can be stored, manipulated, or put in a queue to schedule.
    - Ex- Command pattern can be used to ring alarm in Calendar software. When an event is created in Calendar, a command object could be created to ring alarm. This command can be put on a queue.
    - Undo/ redo commands. Ex- undo/redo in a doc

Achieving undo/redo:
Maintain two stacks:

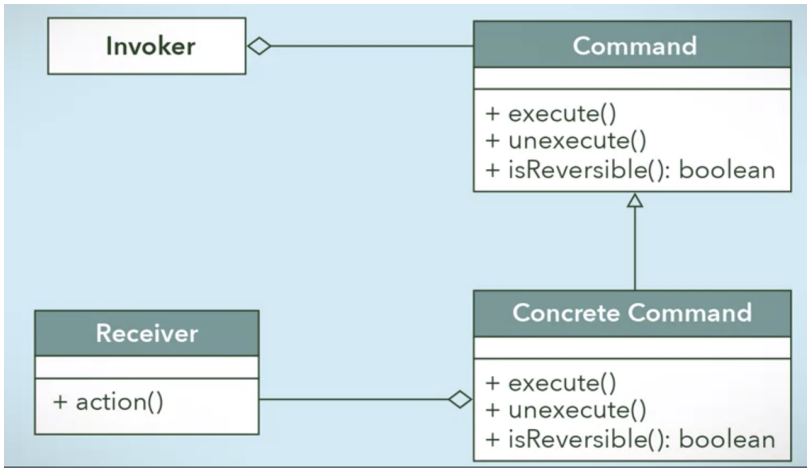
(rightmost is top)
History stack: Bottom | 1, 2, 3
Redo stack: Bottom |

everytime a command is done, it is pushed to history stack.
For undo: pop from history list; push to redo stack
For Redo: Pop from redo stack; do the command; and push to history stack
Redo stack is emptied after everytime a command is executed because you can't redo a previous edit(command) after a new edit has been made.

There could be commands that can't be undone. Ex- Doesn't make sense to undo save

Command vs Adapter: https://stackoverflow.com/questions/28392556/difference-between-command-and-adapter-patterns

- It also decouples the requester from receiver object.
- Ex- Pull out logic from UI. Usually code to handle request is put into event handlers of UIs.
  The UI code shouldn't have any other extra logic. It should only get to and fro info from user. Command pattern can be used to create commands on a button click and all the logic of button click will go into this layer.



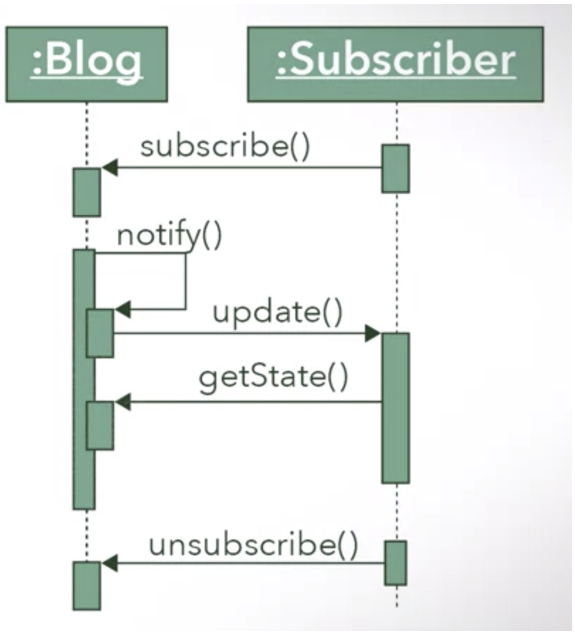

```
public class PasteCommand extends Command {
    private Document doc; //receiver
    private int position;
    private String text;

    // a command needs to keep a lot of things(variables) inorder
    // for the commands to be reversible

    public PasteCommand(Document doc, int position, String text) {
        this.doc = doc;
        ...
    }

    public void execute() {
        doc.insert(position, text);
    }

    public void unexceute() {
        doc.remove(position, text.length);
    }

    public boolean isReversible() {
        return true;
    }
```

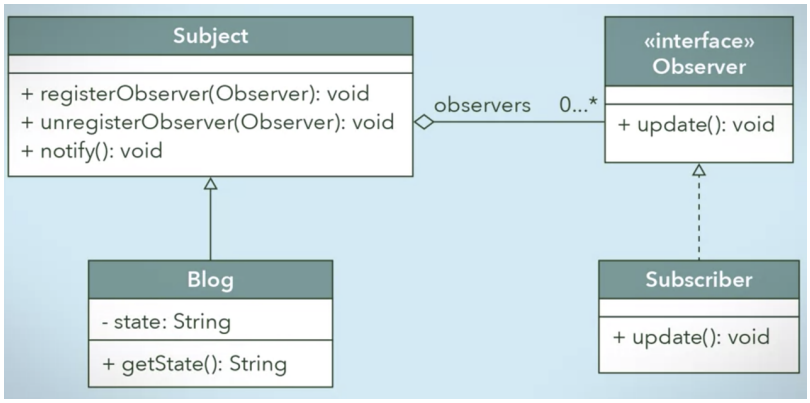
```
}


// Invoker code-

CommandManager commandManager = CommandManager.getInstance();

Command command = new PasteCommand(doc, 1, "cdc");

commandManager.invokeCommand(command); // it executes the command
```

---

**OBSERVER PATTERN**

- Ex- Checking for new blog posts on a page. For pull based method you need to write scripts to check every hour or so. But that wouldn't be real time.
  Blog should push notification to you if you are subscribed.
- Blog is the Subject in our case, and it will keep a list of observers.
- There is one subject and many observers. Ex- Auction and Bidders; News broadcast and viewers
- When many objects rely on a state of one.






```
public class Subject {
    private ArrayList<Observer> observers = new ArrayList();

    public void registerObserver(Observer observer) {
        observers.add(observer);
    }

    public void unregisterObserver(Observer observer) {
        observers.remove(observer);
    }

    public void update() {
        for(Observer observer: observers) {
            observer.update();
        }
    }
}


public class Blog extends Subject {
    private String state;

    public String getState() {
        return this.state;
    }

    ...
}


public interface Observer {
    public void update();
}

public Subscriber implements Observer {
    public void update() {
        // get the blog

        ...
```

```
        }
    }
```

- Java has observable class. Read about it.
- Develop your own observer pattern using generics.

---

---

## MVC Pattern

---

Questions:

1. That Alexa wala
2. Joel W wiki
3. Reducing logs wala