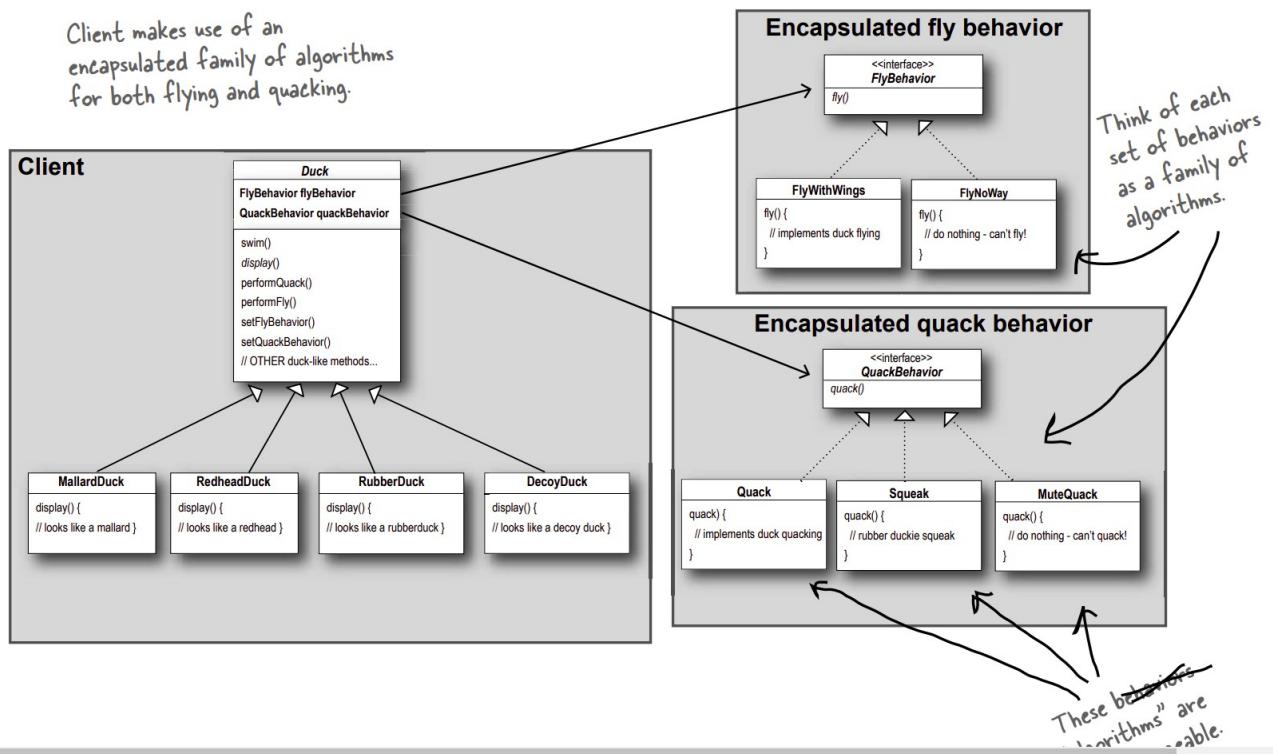


• Strategy Pattern

- defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it
- **Design Principles.**
 - **separate what vary from what stays the same.**
 - **Favor composition over inheritance.**
 - **Program to an interface, not an implementation.**
 - **Code more flexible and changeable**
 - **Classes should be open for extension, but closed for modification.**
 - You can change behaviors by adding new classes through polymorphism.
 - Without changing the existing code as it has been tested (don't touch)



• The Singleton Pattern

- ensures a class has **only one instance**, and provides a global point of access to it
- if you use threading make sure that you use **synchronized or Lazy Initialization**

```

class Singleton
{
    // static variable single_instance of type Singleton
    private static Singleton single_instance = null;

    // variable of type String
    public String s;

    // private constructor restricted to this class itself
    private Singleton()
    {
        s = "Hello I am a string part of Singleton class";
    }

    // static method to create instance of Singleton class
    public static Singleton getInstance()
    {
        if (single_instance == null)
            single_instance = new Singleton();

        return single_instance;
    }
}

```

Factory Pattern

- Separate Creation of Concrete Objects from the application.
- Used with Programming to Interface
 - Programming to Interface used when we have many inherited class with polymorphism it makes the program flexible and adaptable with change .
- Used if your code is probably changeable and you probably will add a class.
 - If it is impossible to change, so this pattern is not suitable.
 - a class you write is likely to change, you have some good techniques like Factory Method to encapsulate that change.
- If we have a many if condition to decide the type of the created object and we repeat them each time we create an object
- we make another class for creation(factory class), and if there is a many ways for creation for that many classes we make an inherited classes from that factory

```
public class PizzaStore {  
    SimplePizzaFactory factory;  
  
    public PizzaStore(SimplePizzaFactory factory) {  
        this.factory = factory;  
    }  
  
    public Pizza orderPizza(String type) {  
        Pizza pizza;  
  
        pizza = factory.createPizza(type);  
  
        pizza.prepare();  
        pizza.bake();  
        pizza.cut();  
        pizza.box();  
        return pizza;  
    }  
  
    public class DependentPizzaStore {  
        public Pizza createPizza(String style, String type) {  
            Pizza pizza = null;  
            if (style.equals("NY")) {  
                if (type.equals("cheese")) {  
                    pizza = new NYStyleCheesePizza();  
                } else if (type.equals("veggie")) {  
                    pizza = new NYStyleVeggiePizza();  
                } else if (type.equals("clam")) {  
                    pizza = new NYStyleClamPizza();  
                } else if (type.equals("pepperoni")) {  
                    pizza = new NYStylePepperoniPizza();  
                }  
            } else if (style.equals("Chicago")) {  
                if (type.equals("cheese")) {  
                    pizza = new ChicagoStyleCheesePizza();  
                } else if (type.equals("veggie")) {  
                    pizza = new ChicagoStyleVeggiePizza();  
                } else if (type.equals("clam")) {  
                    pizza = new ChicagoStyleClamPizza();  
                } else if (type.equals("pepperoni")) {  
                    pizza = new ChicagoStylePepperoniPizza();  
                }  
            } else {  
                System.out.println("Error: invalid type of pizza");  
            }  
        }  
    }  
}
```

- if there is a many ways for creation for that many classes we make factory for each category.
- **Abstract Factory**
 - When we have many categories, each category has many classes (like New-York category has cheese and peproni.. etc classes) and other categories also like calefornia .. etc has the same classes.
 - We can make a classes of NY factory , calefornia factory ... etc and pass them to the constructor of the pizza, and the prepare function will change according to that factory
 - there is no need to make all that classes to all that categories, you can just have cheesePizza , peproniPizza not NycheesePiza,CaleforniaCheesePizza for each city for each pizza
 - **Abstract Factory** relies on object composition: object creation is implemented in methods exposed in the factory interface.
-
- another way is to combine the place we create the class with the factory class and make a create function in that class(like store class)
- and when we want to have many creations we inherit from that main class
-

Adapter Pattern

➤ Object Adapter

- The Adapter Pattern converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- 2 classes have incompatible interfaces (the input of one is not the output of the other).
- adapter class(wrapper) converts the output of the client class to the input of the adaptee class.
- Why** we can't just modify our systems ?
 - Adaptee is an external system or a third-party system that we don't have access to.
 - We can change our system to keep up with the adaptee but this may lead to problems. If someone uses the client system that may cause a problem.
- Ex: client class sends request to web-server, but the client has an object request while the server receives JSON Request. We need an adapter class that converts the output of the client class to the right input of the external system(server) without changing any of the systems.
- We use Target interface (Adapter implements it) to allow multiple adapting through the same interface.
 - Client composite Target (which is reference to adapter (polymorphism))
 - Adapter implements Target & **Adapter Composite Adaptee**

Step 2: Implement the target interface with the adapter class

```
public class WebAdapter implements WebRequester {
    private WebService service;
    public void connect(WebService currentService) {
        this.service = currentService;
    }
    public int request(Object request) {
        Json result = this.toJson(request);
        Json response = service.request(result);
        if (response != null)
            return 200; // OK status code
        return 500; // Server error status code
    }
}
```

The client class only needs to know about the target interface of the adapter.

Step 3: Send the request from the client to the adapter using the target interface

```
public class WebClient {
    private WebRequester webRequester;
    public WebClient(WebRequester webRequester) {
        this.webRequester = webRequester;
    }
    private Object makeObject() { ... } // Make an Object
    public void doWork() {
        Object object = makeObject();
        int status = webRequester.request(object);
        if (status == 200) {
            System.out.println("OK");
        } else {
            System.out.println("Not OK");
        }
    }
}
```

Step three, send the request from the client to the adapter using the target interface.

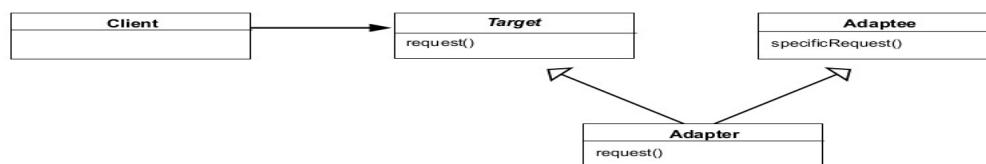
```
public class Program {
    public static void main(String args[]) {
        String webHost = "Host: https://google.com\n";
        WebService service = new WebService(webHost);
        WebAdapter adapter = new WebAdapter();
        adapter.connect(service);
        WebClient client = new WebClient(adapter);
        client.doWork();
    }
}
```

Since your WebClients normal workflow is to return the object back to the client.

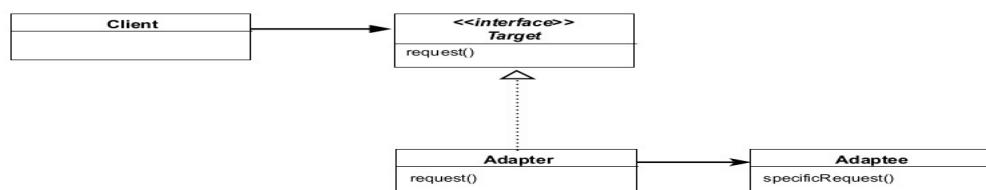
➤ Class Adapter (Requires Multiple inheritance)

- Adapter class inherits Adaptee and Target Class
- Client composite Adapter (with Target reference (polymorphism))

Class Adapter



Object Adapter



Facade Pattern (Makes an interface simpler)

- The facade design pattern is a means to **hide the complexity of a subsystem** by **encapsulating** it behind a unifying **wrapper** called a **facade class**;
- removes the need for client classes to manage a subsystem on their own, resulting in less coupling between the subsystem and the client classes
- provides client classes with a **simplified interface** for the **subsystem** acts simply as a point of entry to a subsystem and **does not add more functionality** to the subsystem.
- The facade design pattern is a way to provide client classes with an **easier** means of **interacting** with the parts of your system.

Principle of Least Knowledge (talk only to your immediate friends)

- **reduces the dependencies between objects**
- be careful of the number of classes it interacts with and also how it comes to interact with those classes. This principle prevents us from creating designs that have a large number of classes coupled together so that changes in one part of the system cascade to other parts. When you build a lot of dependencies between many classes, you are building a fragile system that will be costly to maintain and complex for others to understand.

JUST CALL METHOD IN

- **The object itself**
- **Objects passed in as a parameter to the method**
- **Any object the method creates or instantiates**
- **Any components of the object**

- objects that returned by methods, we can't call methods inside it.

adhere to the principle of Least Knowledge:

```
public class Car {  
    Engine engine;  
    // other instance variables  
  
    public Car() {  
        // initialize engine, etc.  
    }  
  
    public void start(Key key) {  
        Doors doors = new Doors();  
  
        boolean authorized = key.turns();  
  
        if (authorized) {  
            engine.start();  
            updateDashboardDisplay();  
            doors.lock();  
        }  
    }  
  
    public void updateDashboardDisplay() {  
        // update display  
    }  
}
```

Here's a component of this class. We can call its methods.

Here we're creating a new object, its methods are legal.

You can call a method on an object passed as a parameter.

You can call a method on a component of the object.

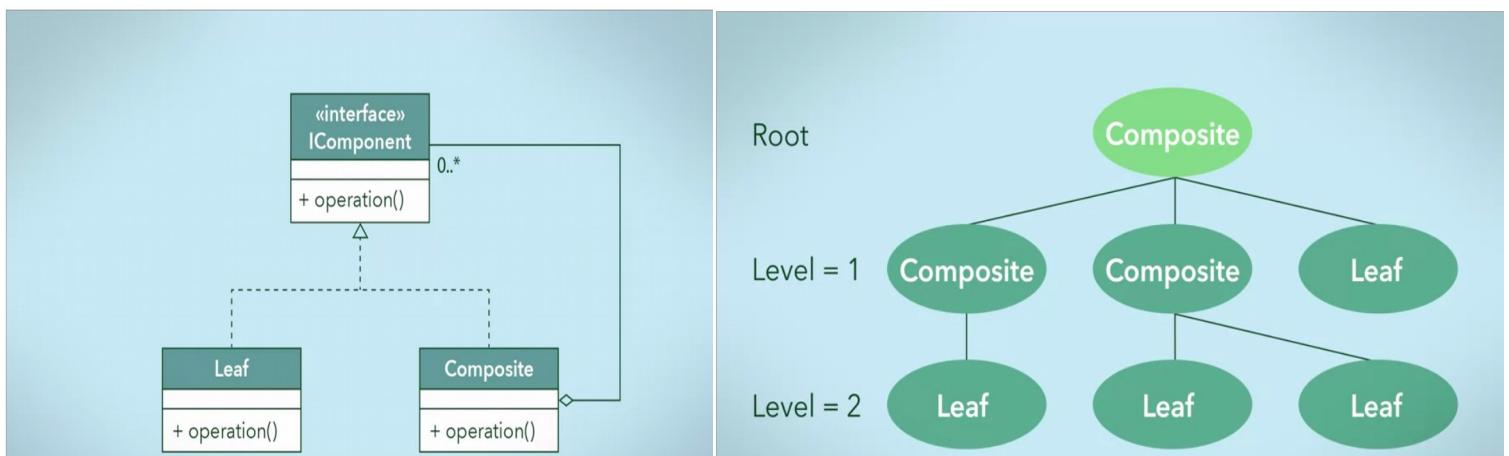
You can call a local method within the object.

You can call a method on an object you create or instantiate.

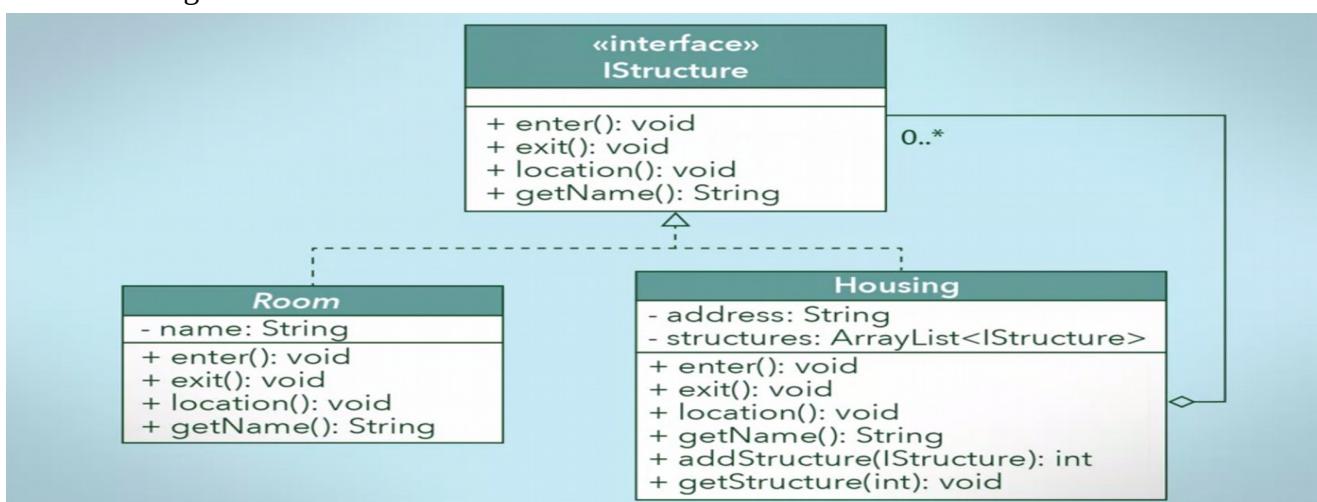
therefore no ..

Composite Design Pattern (Recursive Composition)

- The Composite Pattern allows us to build structures of objects in the form of trees that contain both compositions of objects and individual objects as nodes.
-
- To compose nested structures of objects, and to deal with the classes for these objects uniformly
- when we have a **composition** of classes to a classes of the same type(or **similar type**)
 - like **building** , **floor** and **room** each of them is Structure that we can enter, exit, storing similar structures (building has floors and each floor has rooms)
- Based on **Polymorphism**
 - we must have **abstract** class for all classes that been composed.
- It is a **tree**, a **root** or **leaf**, the root can contain another composite objects(building contains floors – floor contain rooms) && Leaf can not contain the composite objects (room is the end of tree)
- We treat **every component** of the tree with the **same (+operation())**, this makes things generic and flexible, we **program to interface**.



- applying the decomposition and generalization object-oriented design principles together to break a whole into parts, but having the whole and parts both conform to a common type. The composite design pattern lets you build complex structures by constructing them using composite objects and leaf objects which belong to a unified component type. This makes it easier to understand and manipulate the structure, and will lead to more readable, reusable, and meaningful code.



```

public class Housing implements IStructure {
    private ArrayList<IStructure> structures;
    private String address;

    public Housing (String address) {
        this.structures = new ArrayList<IStructure>();
        this.address = address;
    }

    public String getName() {
        return this.address;
    }

    public int addStructure(IStructure component) {
        this.structures.add(component);
        return this.structures.size() - 1;
    }

    public IStructure getStructure(int componentNumber) {
        return this.structures.get(componentNumber);
    }

    public void location() {
        System.out.println("You are currently in " + this.getName() +
            ". It has ");
        for (IStructure struct : this.structures)
            System.out.println(struct.getName());
    }

    /* Print out when you enter and exit the building */
    public void enter() { ... }
    public void exit() { ... }
}

```

Step 3: Implement the leaf class

```

public abstract class Room implements IStructure {
    public String name;

    public void enter() {
        System.out.println("You have entered the " + this.name);
    }

    public void exit() {
        System.out.println("You have left the " + this.name);
    }

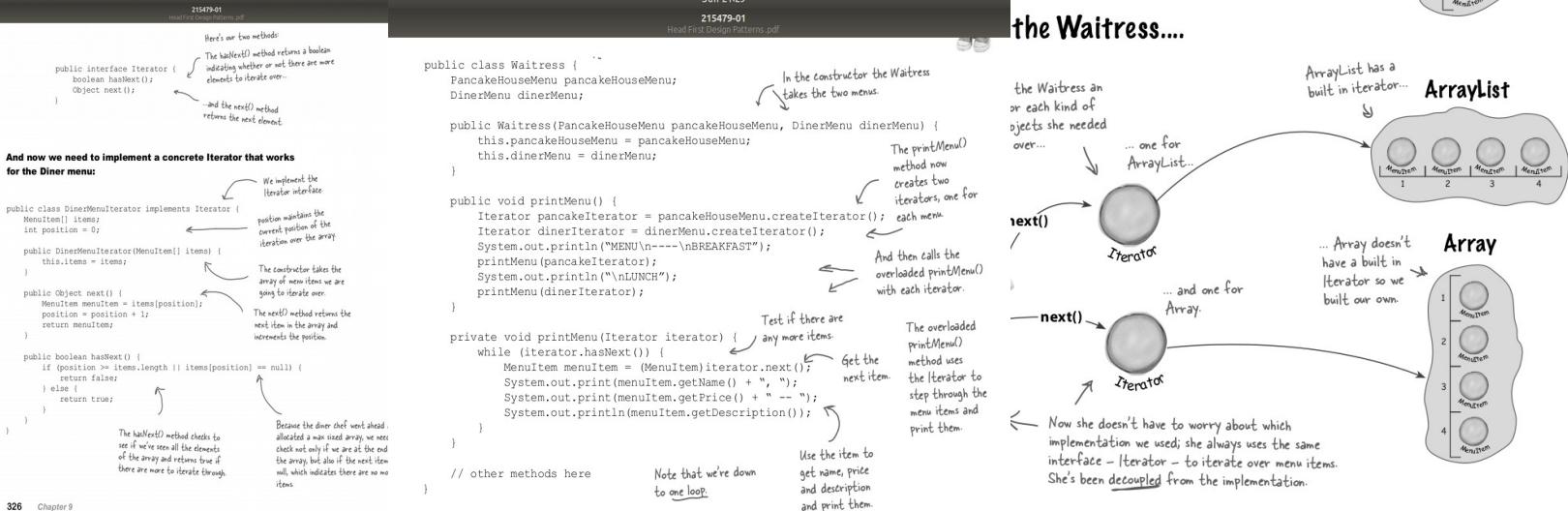
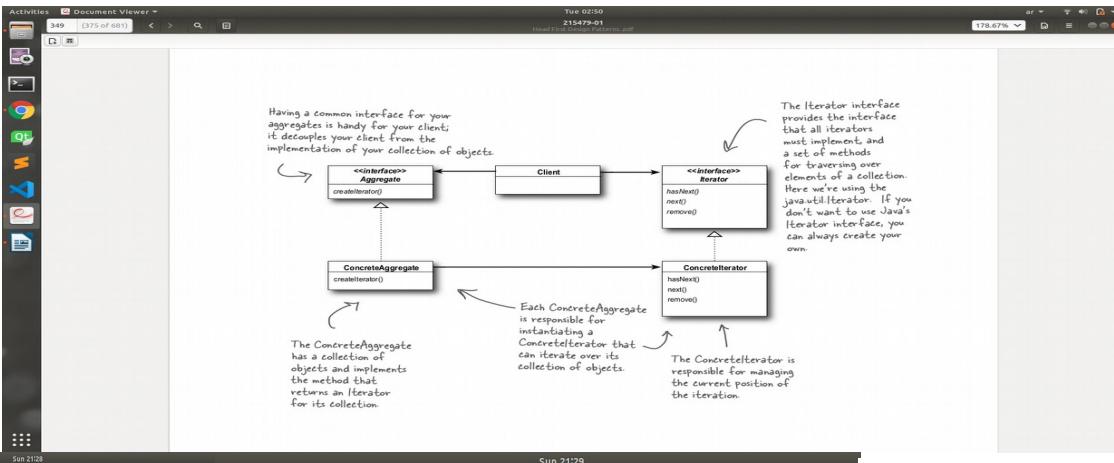
    public void location() {
        System.out.println("You are currently in the " + this.name);
    }

    public String getName() {
        return this.name;
    }
}

```

The Iterator Pattern

- Iteration Interface to encapsulate the way the data stored ex (STL iterators for map, list..etc).
- Iterator Interface & many concrete classes that implements it for different kind of data or different kind of encapsulated classes.
- The encapsulated class aggregate the concrete iterator.
- Iterator has functions that control the iteration based on the kine of the concrete iterator.
- Iterator has a reference to the encapsulated class

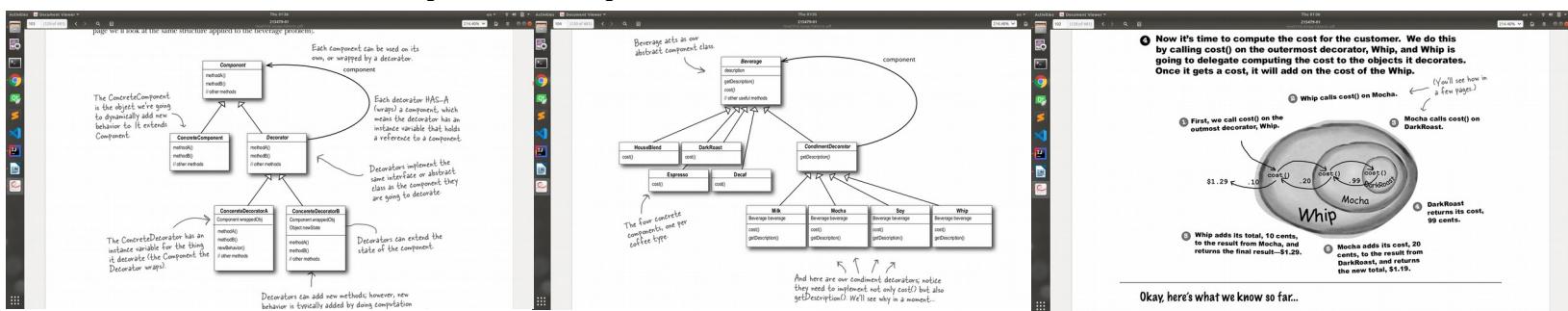


Decorator Pattern

- add **many behaviors (functionality)** dynamically. Stack-Build Oriented
- Polymorphism – inheritance – composition
- its **usage is similar** to the usage of the **strategy** pattern
- separate what varies from what remain the same.
- Favor composition over inheritance.
- Open-Closed Principle (opened for extension, closed for modification)
- build the objects in the stack of each other
 - Decorator Composite a Component which can be(concrete class(drink)or another decorator)
- **Used when**
 - we want to add unknown number of behaviors.
 - Apply functions on the decorators that similar to the original component
 - ex:Drinks with additions (milk,mocha...etc)
- **NOTE:** this pattern in some cases can be replaced by just composition.
 - Adding an ArrayList inside the component to have all additions then loop over them with the order you determine.

https://github.com/AmrElsersy/Design-Patterns/tree/master/src/StrategyPattern_Payment

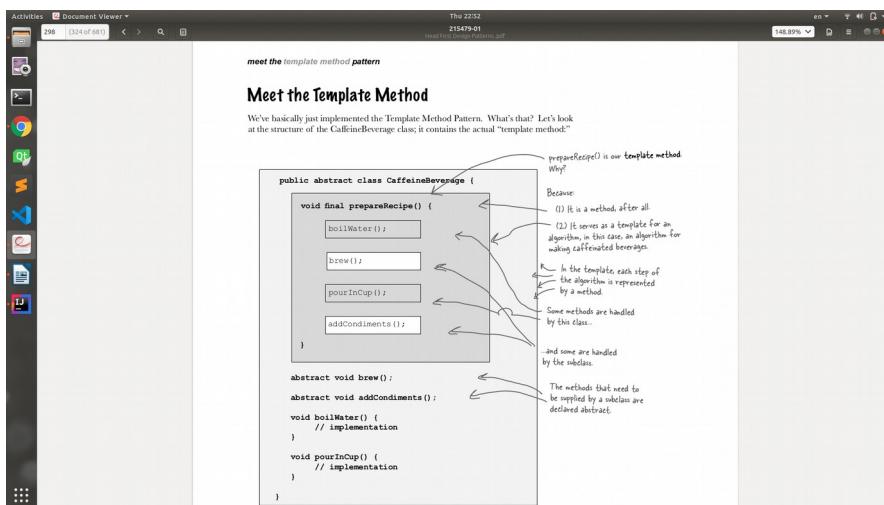
- The order is important in this pattern.



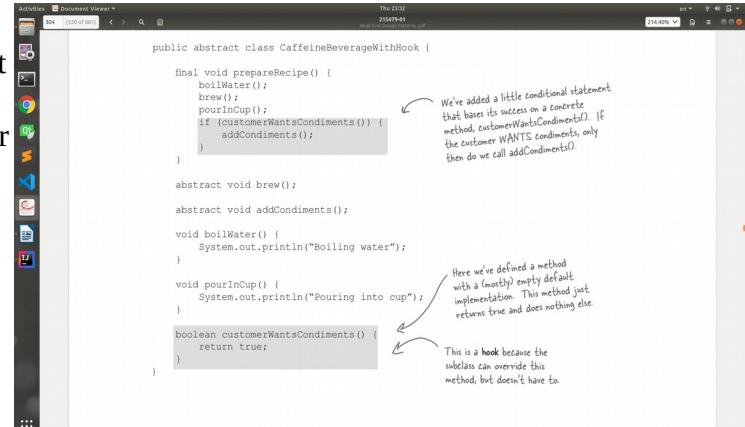
Behavioral Patterns

Template Pattern

- **Encapsulating Algorithms**
- **The Template Method defines the steps of an algorithm and allows sub-classes to provide the implementation for one or more steps.**
- Class with a method called Template-Method that has the **sequence** of executed **methods**.
- Used with **classes** with **similar sequence** of executed methods, so we can have whole different scenarios of sequence of events or methods, and dynamically choose one of them.
- **Subclasses decide how to implement steps in the algorithm.**
- It is not just a **polymorphism**, we don't execute the template method itself based on concrete classes, we do that in some of the functions it calls yes but not to the sequence (the algorithm)
-
- The template method can be helpful if you have two classes with similar functionality, When you notice two classes with a very similar order of operations, you can choose to use a template method, The template method pattern is a practical application of generalization, and inheritance, When writing software, you might notice two separate classes that share similarities like each having a method with a very similar algorithm, Rather than making changes to these algorithms in two places, you can consolidate the algorithms to one place



- **Hook Functions**
 - empty functions ... template methods call it
 - It is optional to the subclass to implement or not to add functionality to the algorithm.
 - Template method calls the hook function with a condition so the control of the algorithm remains with the template, template determines to use the hook or not



Our final stop on the safari: the applet.

You probably know an applet is a small program that runs in a web page. Any applet must subclass Applet, and this class provides several hooks. Let's take a look at a few of them:

```

public class MyApplet extends Applet {
    String message;

    public void init() {
        message = "Hello World, I'm alive!";
        repaint();
    }

    public void start() {
        message = "Now I'm starting up...";
        repaint();
    }

    public void stop() {
        message = "Oh, now I'm being stopped...";
        repaint();
    }

    public void destroy() {
        // applet is going away...
    }

    public void paint(Graphics g) {
        g.drawString(message, 5, 15);
    }
}

```



The init hook allows the applet to do whatever it wants to initialize the applet the first time.

repaint() is a concrete method in the Applet class that lets upper-level components know the applet needs to be redrawn.

The start hook allows the applet to do something when the applet is just about to be displayed on the web page.

If the user goes to another page, the stop hook is used, and the applet can do whatever it needs to do to stop its actions.

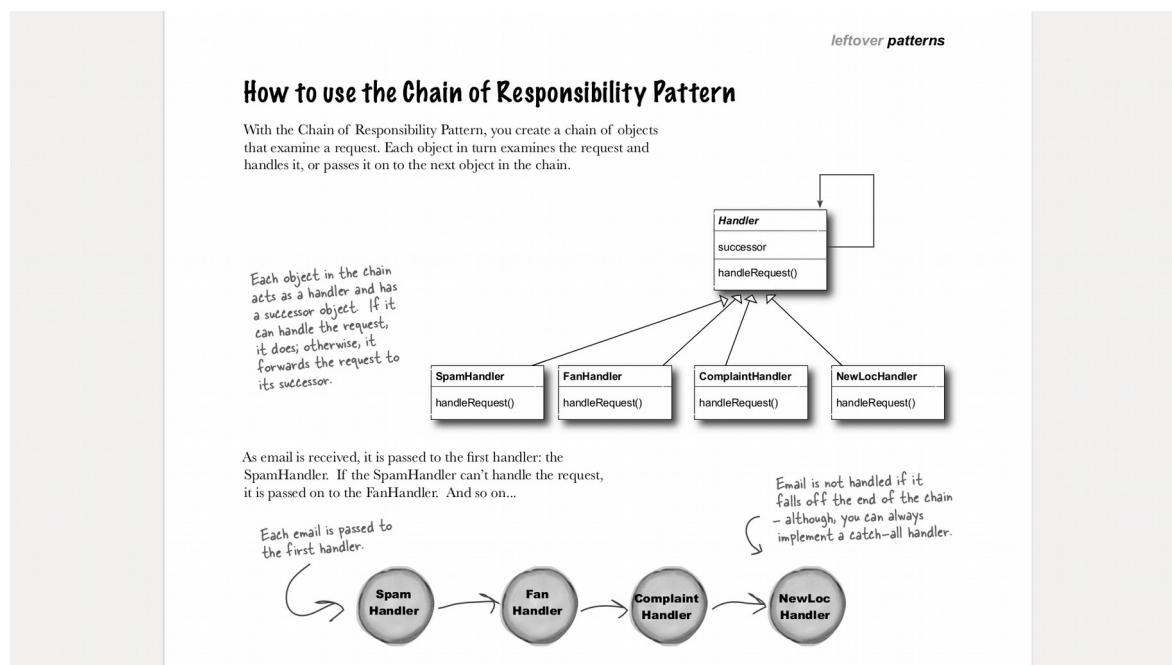
And the destroy hook is used when the applet is going to be destroyed, say, when the browser pane is closed. We could try to display something here, but what would be the point?

Well looky here! Our old friend the paint() method! Applet also makes use of this method as a hook.

The Strategy and Template Method Patterns both encapsulate algorithms, one by inheritance and one by composition.

Chain of Responsibility Pattern

- used when you want to have many ways to handle a request of object in sequence.
- if email is sent and you want to handle it in different ways like
 - span email , hacker email , job email ... etc
 - you pass the email request through many handlers in a sequence way.
- Exception handling
- abstract handler class and many handlers inherits from it to implement their handling.
- Each handler sub-classes has a handler reference to the next handler to pass the request.
- If the next handler is null so it is the last element of the chain then request fails.
- https://www.tutorialspoint.com/design_pattern/chain_of_responsibility_pattern.htm



State Pattern

- implement the class in a **finite state machine**.
- The state pattern is useful when you need to change the behavior of an object based upon changes to its internal state
- a well-defined set of transitions between state
- before we have a state pattern to represent a finite state machine, we will write code like this.

Fri 16:56
215479-01 Head First Design Patterns.pdf
states in ring.pdf

```

public class GumballMachine {
    final static int SOLD_OUT = 0;
    final static int NO_QUARTER = 1;
    final static int HAS_QUARTER = 2;
    final static int SOLD_OUT = 0;

    int state = SOLD_OUT;
    int count = 0;

    public GumballMachine(int count) {
        this.count = count;
        if (count > 0)
            state = HAS_QUARTER;
    }

    Now we start implementing the actions as methods.

    public void insertQuarter() {
        if (state == HAS_QUARTER) {
            System.out.println("you can't insert another quarter");
        } else if (state == NO_QUARTER) {
            System.out.println("you inserted a quarter");
            state = HAS_QUARTER;
        } else if (state == SOLD_OUT) {
            System.out.println("you can't insert a quarter, the machine is sold out");
        } else {
            System.out.println("Please wait, we're already giving you a gumball");
        }
    }

    If the customer just bought a gumball he needs to turn the crank before inserting another quarter.

    When a quarter is inserted, if a quarter is already inserted we tell the customer; otherwise we accept the quarter and transition to the HAS_QUARTER state.

    The machine now takes an initial inventory of gumballs. If the inventory isn't zero, the machine enters state NO_QUARTER, meaning it is waiting for someone to insert a quarter; otherwise it stays in the SOLD_OUT state.

    and if the machine is sold out, we reject the quarter.
  
```

390 Chapter 10
Download at WoweBook.Com

the state pattern

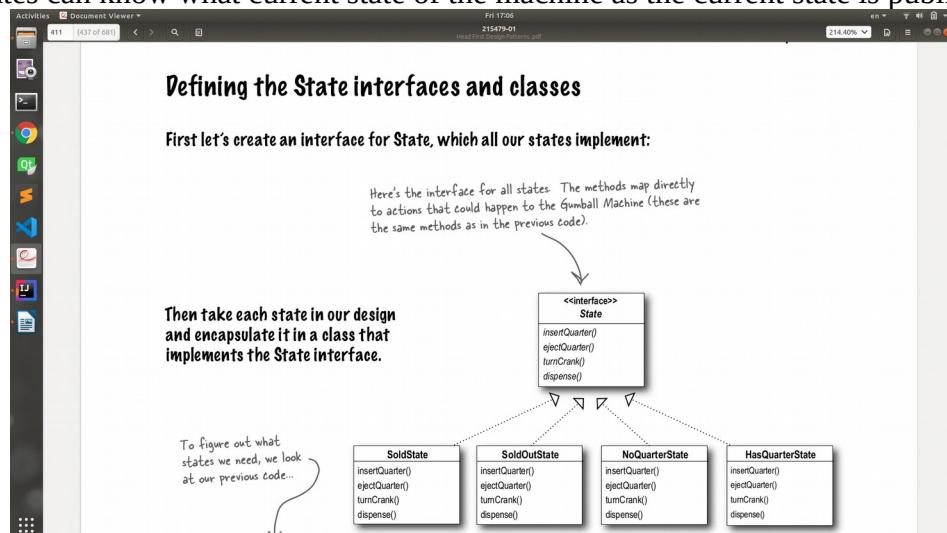
```

public void ejectQuarter() {
    if (state == HAS_QUARTER) {
        System.out.println("Quarter returned");
        if there is a quarter we return it and go back to the NO_QUARTER state.
    } else if (state == NO_QUARTER) {
        System.out.println("You haven't inserted a quarter");
        Otherwise, if there isn't one we can't give it back.
    } else if (state == SOLD_OUT) {
        System.out.println("Sorry, you already turned the crank");
    } else {
        System.out.println("You can't eject, you haven't inserted a quarter yet");
    }
}

The customer tries to turn the crank.

You can't eject if the machine is sold out, it doesn't accept quarters!
  
```

- But when we want to add a state to the system we will need to go to the whole actions(functions) then add the “else if” for that new state
- instead, we will **encapsulate what varies**, we will **implement the state** in a **class** that **handles** all its **actions**
- (**note** we can't encapsulate actions, as we will still need to add the state to the whole classes of actions so we solve nothing!)
- we make state interface with functions that represent actions of the finite state machine.
- this states can know what current state of the machine as the current state is public static !



GumballMachine.java

```

public class GumballMachine {
    State soldOutState;
    State noQuarterState;
    State hasQuarterState;
    State soldState;

    State state = soldOutState;
    int count = 0;

    public GumballMachine(int numberGumballs) {
        soldOutState = new SoldOutState(this);
        noQuarterState = new NoQuarterState(this);
        hasQuarterState = new HasQuarterState(this);
        soldState = new SoldState(this);
        this.count = numberGumballs;
        if (numberGumballs > 0)
            state = noQuarterState;
    }

    public void insertQuarter() {
        state.insertQuarter();
    }

    public void ejectQuarter() {
        state.ejectQuarter();
    }

    public void turnCrank() {
        state.turnCrank();
        state.dispense();
    }

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

```

HasQuarterState.java

```

public class HasQuarterState implements State {
    GumballMachine gumballMachine;

    public HasQuarterState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    public void insertQuarter() {
        System.out.println("You can't insert another quarter");
    }

    public void ejectQuarter() {
        System.out.println("Quarter returned");
        gumballMachine.setState(gumballMachine.getNoQuarterState());
    }

    public void turnCrank() {
        System.out.println("You turned...");
        gumballMachine.setState(gumballMachine.getSoldState());
    }

    public void dispense() {
        System.out.println("No gumball dispensed");
    }
}

```

Annotations on the left side of the first screenshot:

- Here are all the States again...
- ...and the State instance variable.
- The count instance variable holds the count of gumballs - initially the machine is empty.
- Our constructor takes the initial number of gumballs and stores it in an instance variable. It also creates the State instances, one of each.
- If there are more than 0 gumballs we set the state to the NoQuarterState.
- Now for the actions. There are VERY EASY to implement now. We just delegate to the current state.
- Note that we don't need an action method for dispense() in GumballMachine because it's just an internal action, a user can't ask the machine to dispense directly. But we do call dispense() on the State object from the turnCrank() method.
- This method allows other objects (like

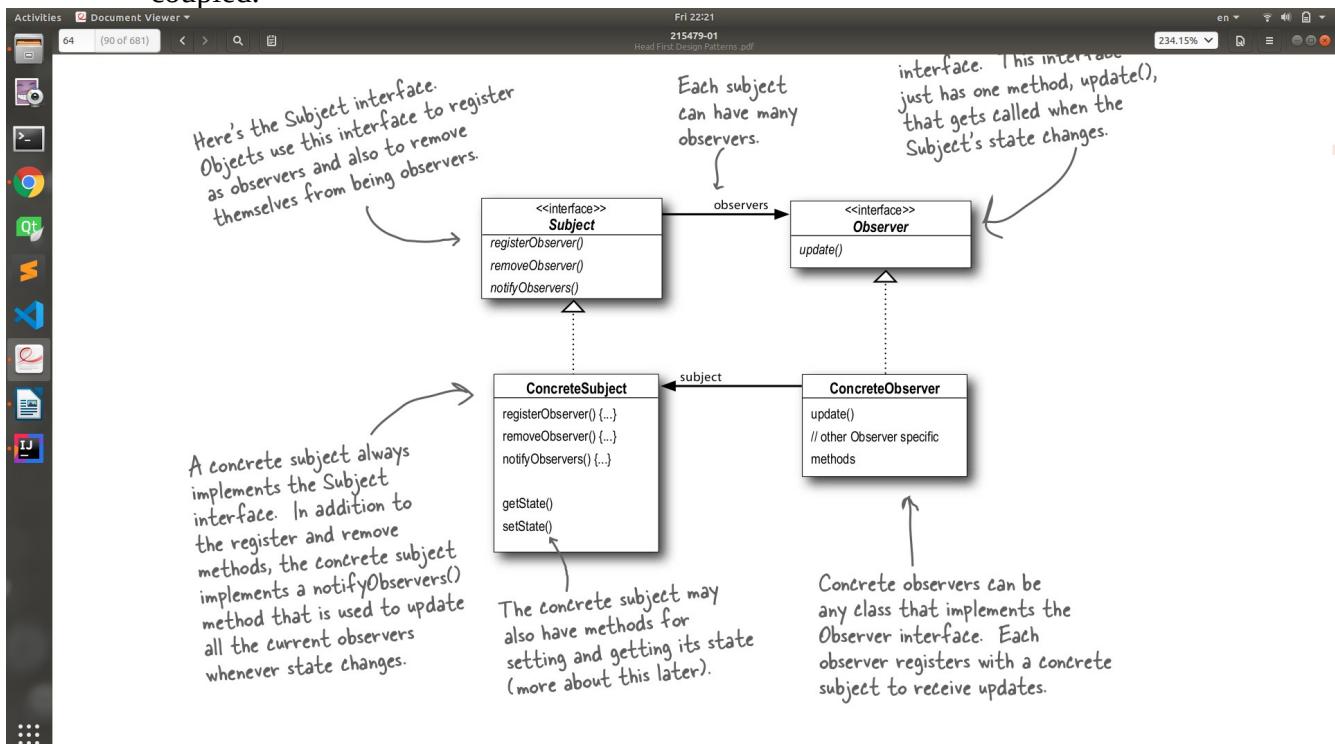
Annotations on the right side of the second screenshot:

- When the state is instantiated we pass it a reference to the GumballMachine. This is used to transition the machine to a different state.
- An inappropriate action for this state.
- Return the customer's quarter and transition back to the NoQuarterState.
- When the crank is turned we transition the machine to the SoldState state by calling its setState() method and passing it the SoldState object. The SoldState object is referenced by the getState() getter method (there is one of these getter methods for each state).
- Another inappropriate action for this state.

- With this new pattern we can define or add new states dynamically and modify just few code in the other states that their next state is our new defined state

Observer Pattern

- defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.
- As a behavioral pattern, it makes it easy to distribute and handle notifications of changes across systems, in a manageable and controlled way.
- The power of Loose Coupling
 - When two objects are loosely coupled, they can interact, but have very little knowledge of each other.
 - The Observer Pattern provides an object design where subjects and observers are loosely coupled.



Command Pattern

- Decoupling , Undo Operations
 - Decoupling Objects : **encapsulate the request (command)** between 2 objects so the 2 objects don't know the details of each other
 - **ex:** waiter and cook, the waiter doesn't care about how the food is cooked, this request or command is encapsulated in "order" object.
 - command object encapsulates a request by **binding** together a **set of actions** on a specific **receiver** no other objects really know what actions get performed on what receiver; they just know that if they call the **execute()** method, their request will be serviced.

1. Command Interface with execute() & undo()
 2. Commands-Concrete-Classes implement the command interface & has a composition of the **encapsulated object**
 3. invoker object has the command object and uses execute & undo() without knowing who is executed, completely decoupled from the **encapsulated object**

Here's the Command interface:

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

Simple! All we need is one method called execute().

Implementing a Command to turn a light on

Now, let's say you want to implement a command for turning a light on. Referring to our set of vendor classes, the Light class has two methods: `on()` and `off()`. Here's how you can implement this as a command:

```
public class LightOnCommand implements Command {
    Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.on();
    }
}
```

This is a command, so we need to implement the Command interface.

The constructor is passed the specific light that this command is going to control - say the living room light - and stash it in the `light` instance variable. When `execute` gets called, this is the `Light` object that is going to be the Receiver of the request.

The `execute` method calls the `on()` method on the receiving `Light` object, which is the `Light` we are controlling.

```
public class SimpleRemoteControl {
    Command slot;
    public SimpleRemoteControl() {}
    public void setCommand(Command command) {
        slot = command;
    }
    public void buttonWasPressed() {
        slot.execute();
    }
}
```

We have one slot to hold our commands which will control one device.

We have a method for setting the command the slot is going to control. This could be called multiple times if the client of this code wanted to change the behavior of the remote button.

This method is called when the button is pressed. All we do is take the current command bound to the slot and call its execute() method.

Creating a simple test to use the Remote Control

Here's just a bit of code to test out the simple remote control. Let's take a look and we'll point out how the pieces match the Command Pattern diagram:

- **Undo Operations**

Sun 22/24
21:54:01
https://www.udemy.com/learn-the-command-design-pattern/

```
public class LightOnCommand implements Command {
    Light light;
    public LightOnCommand(Light light) {
        this.light = light;
    }
    public void execute() {
        light.on();
    }
    public void undo() {
        light.off(); // executed) turns the
                    // light on, so undo)
                    // simply turns the light
                    // back off.
    }
}
```

Piece of cake! Now for the LightOffCommand. Here the undo() method just needs to call the Light's on() method.

```
public class LightOffCommand implements Command {
    Light light;
    public LightOffCommand(Light light) {
        this.light = light;
    }
    public void execute() {
        light.off();
    }
    public void undo() {
        light.on(); // And here undo() turns
                    // the light back on!
    }
}
```

Sun 22/24
21:54:01
https://www.udemy.com/learn-the-command-design-pattern/

```
public class RemoteControlWithUndo {
    Command[] onCommands;
    Command[] offCommands;
    Command undoCommand;
    This is where we'll stash the last command
    executed for the undo button.
}
public RemoteControlWithUndo() {
    onCommands = new Command[7];
    offCommands = new Command[7];
    Command noCommand = new NoCommand();
    for(int i=0;i<7;i++) {
        onCommands[i] = noCommand;
        offCommands[i] = noCommand;
    }
    undoCommand = noCommand;
}
Just like the other slots, undo
starts off with a NoCommand, so
pressing undo before any other
button won't do anything at all.
public void setCommand(int slot, Command onCommand, Command offCommand) {
    onCommands[slot] = onCommand;
    offCommands[slot] = offCommand;
}
public void onButtonWasPushed(int slot) {
    onCommands[slot].execute();
    undoCommand = onCommands[slot];
}
public void offButtonWasPushed(int slot) {
    offCommands[slot].execute();
    undoCommand = offCommands[slot];
}
public void undoButtonWasPushed() {
    undoCommand.undo();
}
public String toString() {
    // toString code here...
}
}
```

Sun 22/24
21:54:01
https://www.udemy.com/learn-the-command-design-pattern/

```
public class RemoteLoader {
    public static void main(String[] args) {
        RemoteControlWithUndo remoteControl = new RemoteControlWithUndo();
        Light livingRoomLight = new Light("Living Room");
        LivingOnCommand livingRoomLightOn =
            new LivingOnCommand(livingRoomLight);
        LivingOffCommand livingRoomLightOff =
            new LivingOffCommand(livingRoomLight);
        remoteControl.setCommand(0, livingRoomLightOn, livingRoomLightOff);
        remoteControl.onButtonWasPushed(0);
        remoteControl.offButtonWasPushed(0);
        System.out.println(remoteControl);
        remoteControl.undoButtonWasPushed();
        remoteControl.setCommand(0, livingRoomLightOn, livingRoomLightOff);
        System.out.println(remoteControl);
        remoteControl.undoButtonWasPushed();
    }
}
And here's the test results...


```
java -jar my_commands.jar
light is on
light is off
living room control
slot 0: headfirst_command.undo.LightOffCommand
slot 1: headfirst_command.undo.LightOffCommand
slot 2: headfirst_command.undo.LightOffCommand
slot 3: headfirst_command.undo.LightOffCommand
slot 4: headfirst_command.undo.LightOffCommand
slot 5: headfirst_command.undo.LightOffCommand
slot 6: headfirst_command.undo.LightOffCommand
undo: was pressed. The Living Room's
undo) turns the light off on
Light is on
Then we turn the light off then back on
Now undo holds the
last command. The last
command is on
Here's the Light commands

```


```

MVC Model-View-Controller

- MVC is the separation of concerns of the system components
 - the View that is Responsible for just taking inputs and showing outputs
 - Controller is Responsible for Understanding the View's interactions and update model.
 - Model is the Data and Application Logic of the System.
- Design Principles
 - Separation of Concerns
 - Entity Objects (Model) , Control Object (Controller) , Boundary Objects (View)
 - Decoupling View and Model
 - MVC can be used in any UI Software System
- The model makes use of the Observer Pattern so that it can keep observers updated yet stay decoupled from them
- The controller is the strategy for the view. The view can use different implementations of the controller to get different behavior.
- The view uses the Composite Pattern to implement the user interface, which usually consists of nested components like panels, frames and buttons.
- MP3 Player Program :

- ① You're the user — you interact with the view.
The view is your window to the model. When you do something to the view (like click the Play button) then the view tells the controller what you did. It's the controller's job to handle that.
- ② The controller asks the model to change its state.
The controller takes your actions and interprets them. If you click on a button, it's the controller's job to figure out what that means and how the model should be manipulated based on that action.
- ③ The controller may also ask the view to change.
When the controller receives an action from the view, it may need to tell the view to change as a result. For example, the controller could enable or disable certain buttons or menu items in the interface.
- ④ The model notifies the view when its state has changed.
When something changes in the model, based either on some action you took (like clicking a button) or some other internal change (like the next song in the playlist has started), the model notifies the view that its state has changed.
- ⑤ The view asks the model for state.
The view gets the state it displays directly from the model. For instance, when the model notifies the view that a new song has started playing, the view requests the song name from the model and displays it. The view might also ask the model for state as the result of the controller requesting some change in the view.

- MVC is a set of Patterns (Strategy-Observer-Composite)

Tue 23:21
215479-01
HeadFirstDesignPatterns.pdf

compound patterns

Let's start with the model. As you might have guessed the model uses Observer to keep the views and controllers updated on the latest state changes. The view and the controller, on the other hand, implement the Strategy Pattern. The controller is the behavior of the view, and it can be easily exchanged with another controller if you want different behavior. The view itself also uses a pattern internally to manage the windows, buttons and other components of the display: the Composite Pattern.

Let's take a closer look:

Strategy

The view and controller implement the classic Strategy Pattern: the view is an object that is configured with a strategy. The controller provides the strategy. The view is concerned only with the visual aspects of the application, and delegates to the controller for any decisions about the interface behavior. Using the Strategy Pattern also keeps the view decoupled from the model because it is the controller that is responsible for interacting with the model to carry out user requests. The view knows nothing about how this gets done.

The user did something → Controller → Change your state → View → I've changed!

Observer

Model → My state has changed! → Observers (View, Controller, View) → All these observers will be notified whenever state changes in the model.

I'd like to register as an observer → View → Any object that's interested in state changes in the model registers with the model as an observer.

Composite

View → The display consists of a nested set of windows, panels, buttons, text labels and so on. Each display component is a composite (like a window) or a leaf (like a button). When the controller tells the view to update, it only has to tell the top view component, and Composite takes care of the rest.

Controller → Change your display → View → I've changed!

Observer → I need your state information → Model → The model implements the Observer Pattern to keep interested objects updated when state changes occur. Using the Observer Pattern keeps the model completely independent of the views and controllers. It allows us to use different views with the same model, or even use multiple views at once.

Strategy

View → The user did something → Controller → The controller is the strategy for the view - it's the object that knows how to handle the user actions.

View → The view delegates to the controller to handle the user actions → Controller → We can swap in another behavior for the view by changing the controller.

Composite

View → point() → Set BMP: → Window → The view is a composite of GUI components (labels, buttons, text entry field etc). The top level component contains other components which contain other components and so on until you get to the leaf nodes.

Now let's have a look at the concrete BeatModel class:

```
public class BeatModel implements BeatModelInterface, MetaEventListener {
    Sequence sequencer;
    ArrayList beatObservers = new ArrayList();
    ArrayList bpmObservers = new ArrayList();
    int bpm = 90;
    // other instance variables here

    public void initialize() {
        setUpMidi();
        buildTracksAndStart();
    }

    public void on() {
        sequencer.start();
        setBPM(90);
    }

    public void off() {
        sequencer.stop();
    }

    public void setBPM(int bpm) {
        this.bpm = bpm;
        sequencer.setTempoInBPM(getBPM());
        notifyBPMObservers();
    }

    public int getBPM() {
        return bpm;
    }

    void beatEvent() {
        notifyBeatObservers();
    }

    // Code to register and notify observers
    // Lots of MIDI code to handle the beat
}
```

This is needed for the MIDI code.

We implement the BeatModelInterface.

The sequencer is the object that knows how to generate real beats (that you can hear!).

These ArrayLists hold the two kinds of observers (Beat and BPM observers).

This method does setup on the sequencer and sets up the beat tracks for us.

The on() method starts the sequencer and sets the BPMs to the default 90 BPM.

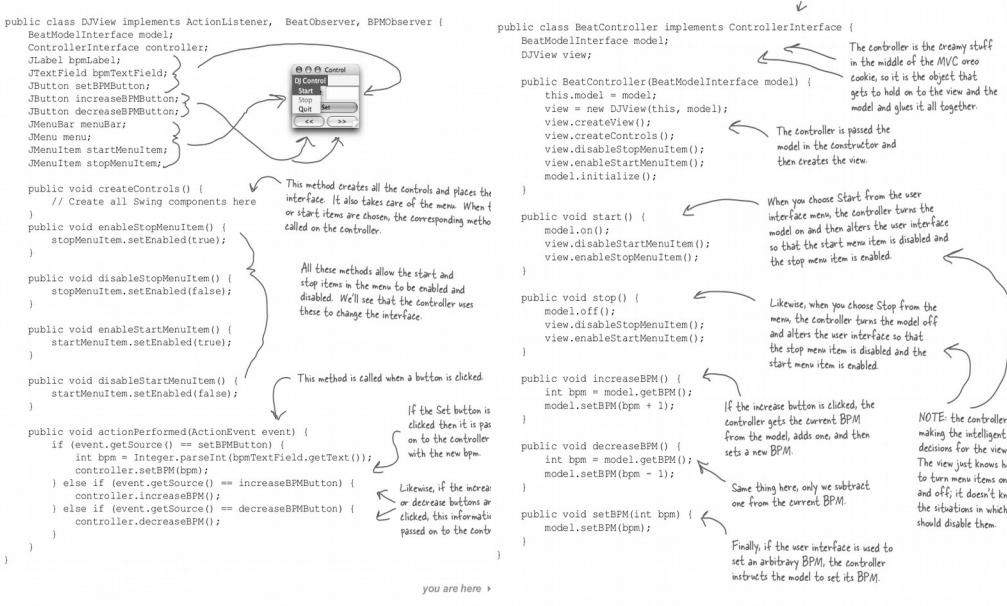
And off() shuts it down by setting BPMs to 0 and stopping the sequencer.

The setBPM() method is the way the controller manipulates the beat. It does three things:

- (1) Sets the bpm instance variable
- (2) Asks the sequencer to change its BPMs
- (3) Notifies all BPM Observers that the BPM has changed

The getBPM() method just returns the bpm instance variable, which indicates the current beats per minute.

The beatEvent() method, which is not in the BeatModelInterface, is called by the MIDI code whenever a new beat starts. This method notifies all BeatObservers that a new beat has just occurred.



- MVC is widely used in Web , see page 549 and search for JSP
- Important question

Q: Does the controller ever implement any application logic?

A: No, the controller implements behavior for the view. It is the smarts that translates the actions from the view to actions on the model. The model takes those actions and implements the application logic to decide what to do in response to those actions. The controller might have to do a little work to determine what method calls to make on the model, but that's not considered the "application logic." The application logic is the code that manages and manipulates your data and it lives in your model.

Code Smells

- Comments (too low or too high)**
- Duplicated Code (Don't Repeat Yourself Principle)**
 - too much efforts to update in multiple locations
 - encapsulate the code in one location and use it
- Long Method – Large Class**
 - Low Cohesion
 - Split this to many classes or methods – Single Responsibility Principle
- Data Class**
 - Class with just variables + getters and setters
 - each class must have functionality , effective methods
- Long Parameter List**
- divergent change (due to large class)**
 - when you have to change a class in many different ways for many different reasons
 - Low cohesion , Single Responsibility
- Shotgun Surgery**
 - when you change one area (adding features , fixing bug) you have to change many places, tack a shotgun and edit every function in the code !
 - The opposite of the polymorphism benefits.

- **Feature Envy**
 - when a method in class are more interested in other class data rather than the class it is in.
 - it is better to put them together , or to combine them into one class.
- **inappropriate intimacy**
 - Classes have 2 way communication in a circular way (a method in one class calls a method in another , and vice versa)
 - encapsulate the circular functionality into another class that associate them.
 - Or make the communicate one way
 - circular communication is not necessarily a bad thing but if we could reduce it will be nice
- **Law of Demeter**
 - It is not allowed to use deep nested call methods within classes (use only one dot)
 - Decreasing Dependency between classes
 - driver class calls methods of car class , but it shouldn't call methods of the engine of the class , it breaks abstraction.
 - a method, M, of an object should only call other methods if they are:
 - encapsulated within the same object, encapsulated within an object that is in the parameters of M,
 - encapsulated within an object that is instantiated inside of M, or encapsulated within an object that is reference in an instance variable of the class for M
 - <https://www.facebook.com/sameh.serag.deabes/posts/2038661573061674>
- **Switch Statements Smell**
 - Solve with Polymorphism
- **Speculative Generality**
 - when you create a class and don't need it at this time
 - follow agile , implement just what you need now
- **Refused Request**
 - when class inherits something that doesn't need
 - Interface Segregation
 - <https://www.facebook.com/sameh.serag.deabes/posts/2008818352712663>