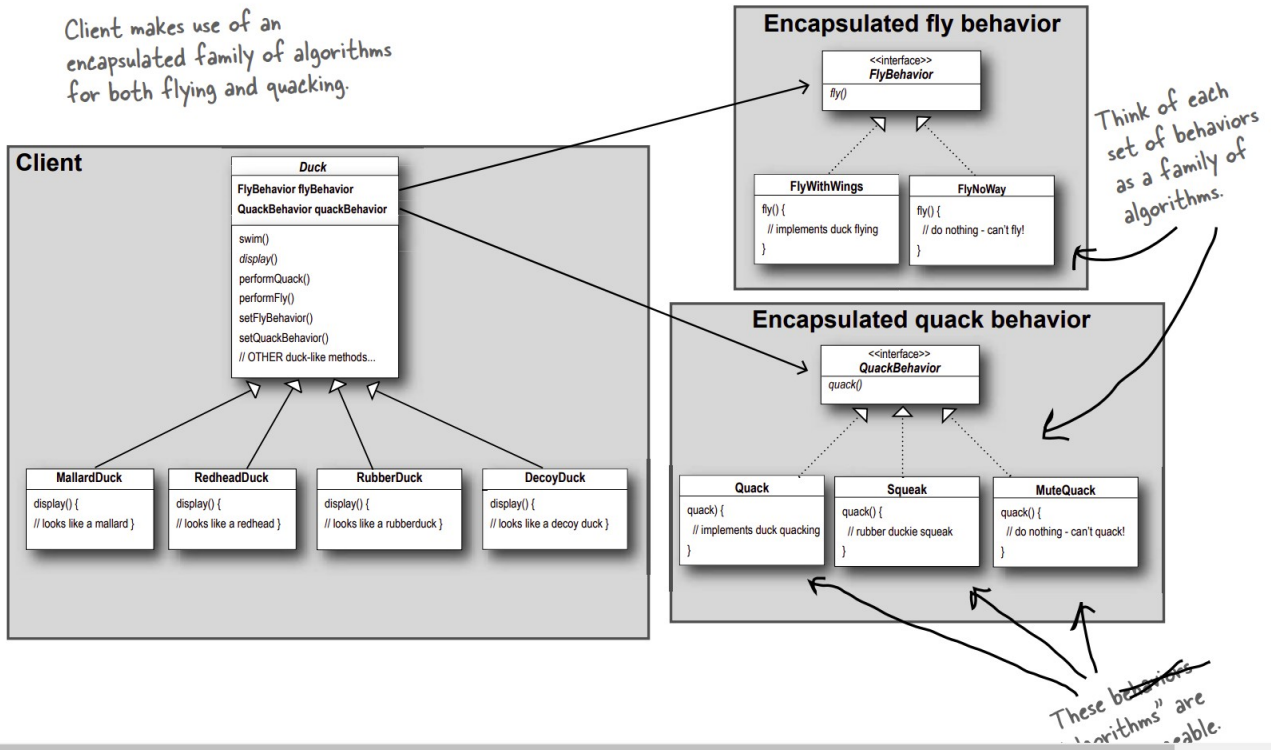
- **Strategy Pattern**
  - defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it
  - **Design Principles.**
    - **separate what vary from what stays the same.**
    - **Favor composition over inheritance.**
    - **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");
```

Handles all the NY style pizzas

Handles all the Chicago style pizzas

- 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,CalefforniaCheesePizza 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
- ……

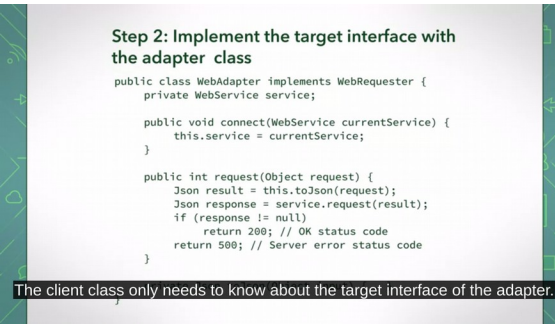
## Adapter Pattern

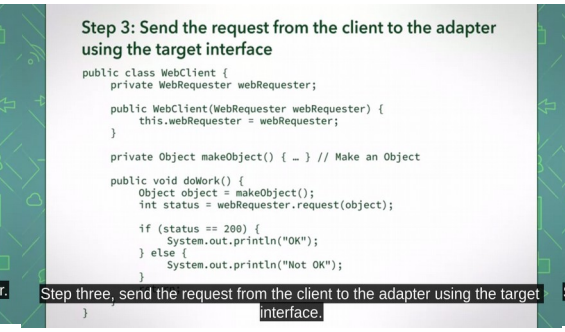
### ➢ Object Adapter

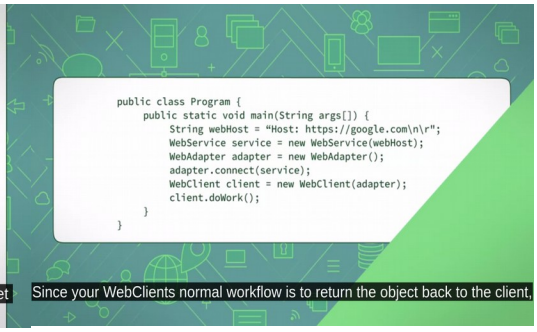
- The Adapter Pattern converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

- 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 is 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 receive 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**



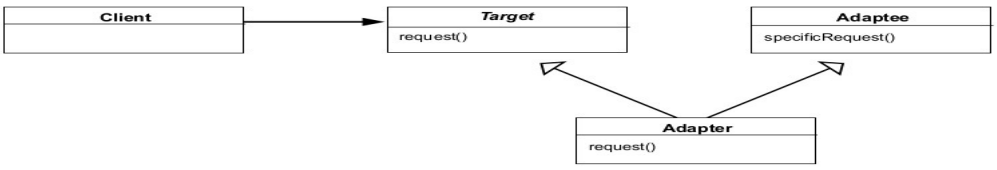



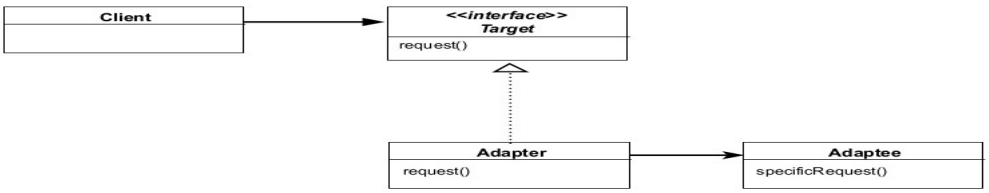

### ➢ Class Adapter (Requires Multiple inheritance)

- 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.

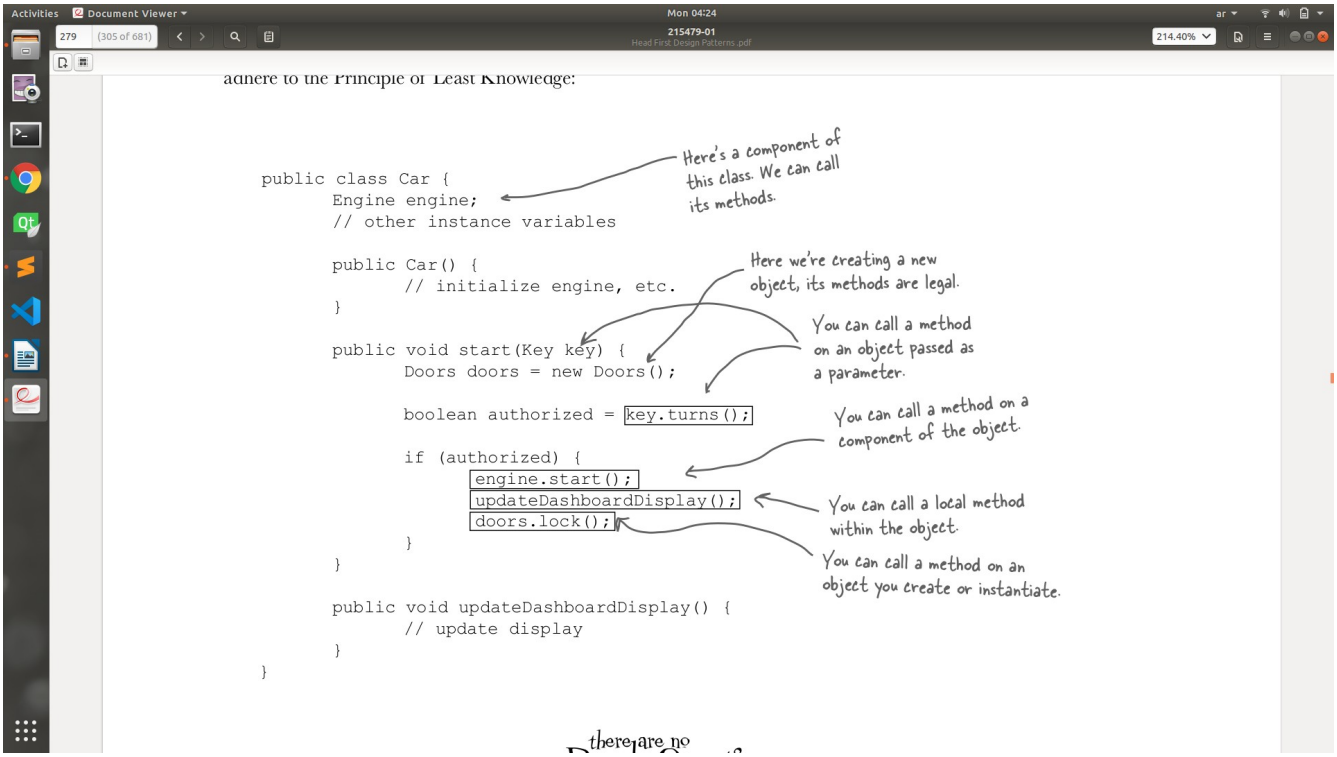
**Principle of Least Knowledge (talk only to your immediate friends)**

- **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

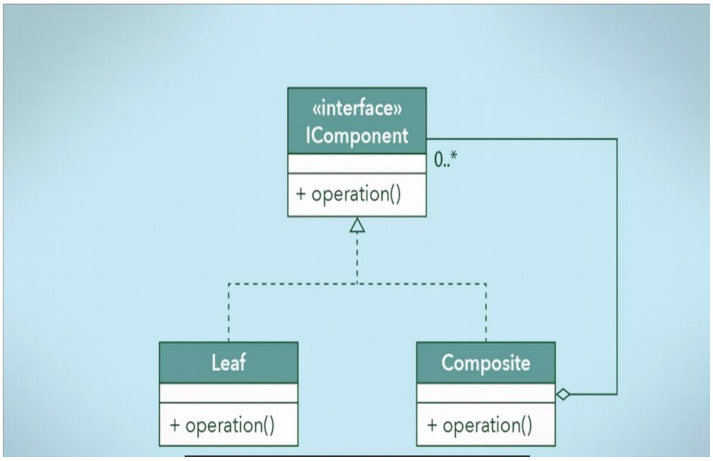
- **The object itself**
- **Objects passed in as a parameter to the method**
- **Any object the method creates or instantiates**
- **Any components of the object**

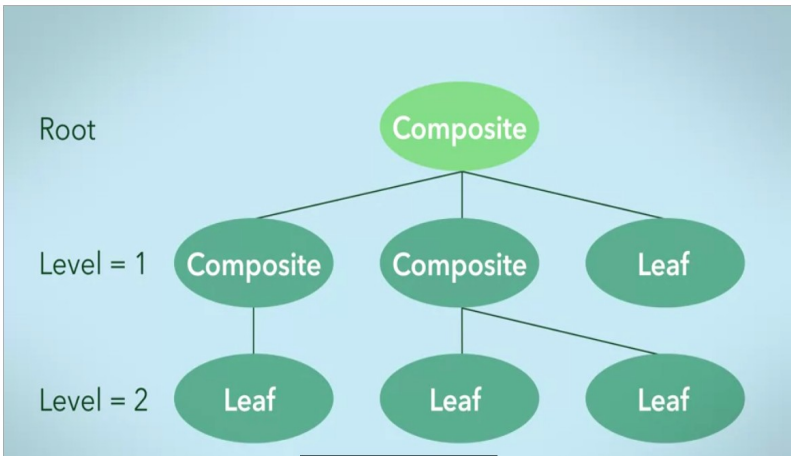
- objects that returned by methods, we can't call methods inside it.



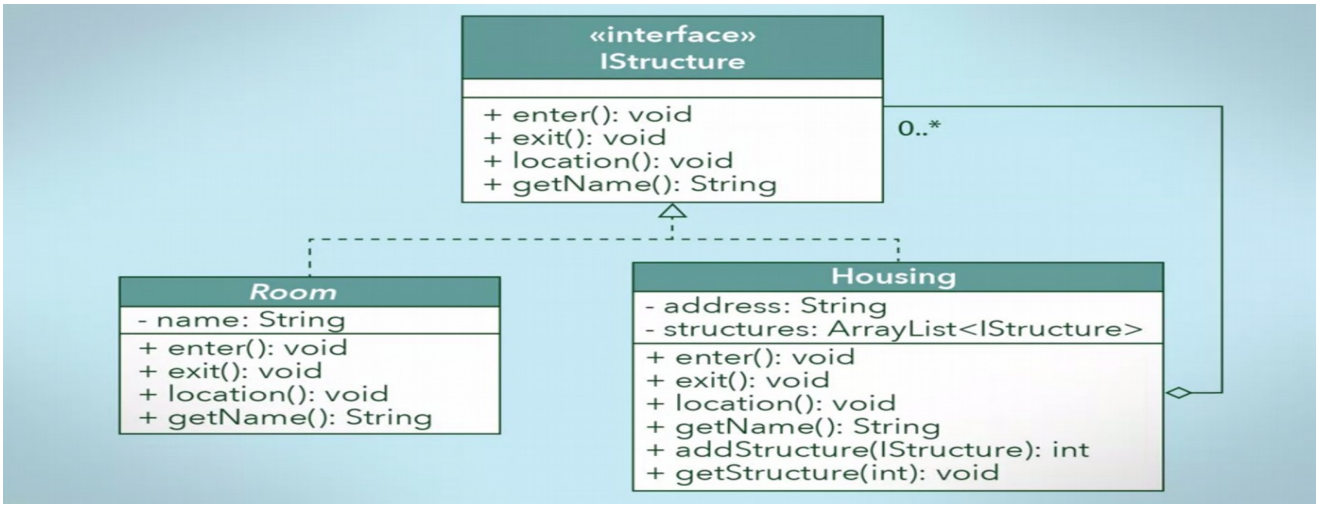
## Composite Design Pattern ( Recursive Composition )

- **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.



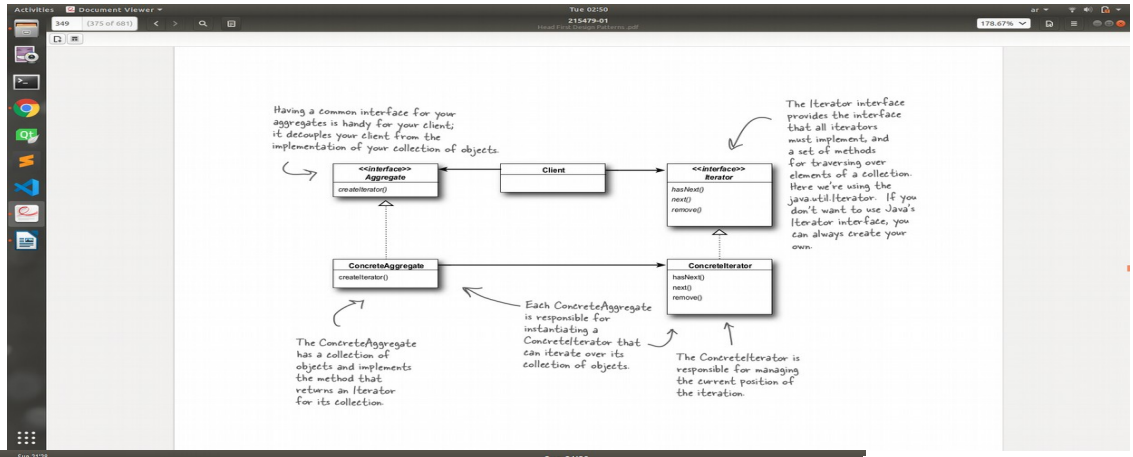
```java
public class Housing implements IStructure {
    private ArrayList<IStructure> structures;
    private String address;

    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() { … }
}
```

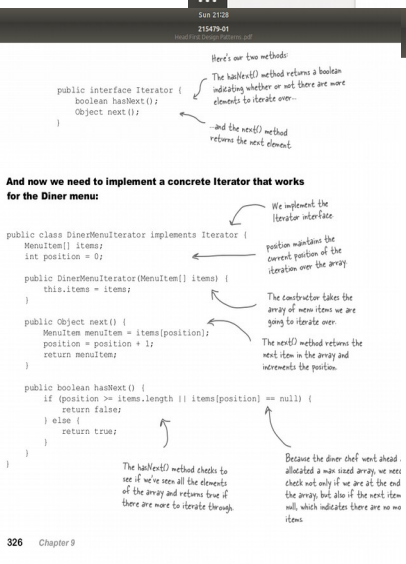
## Step 3: Implement the leaf class

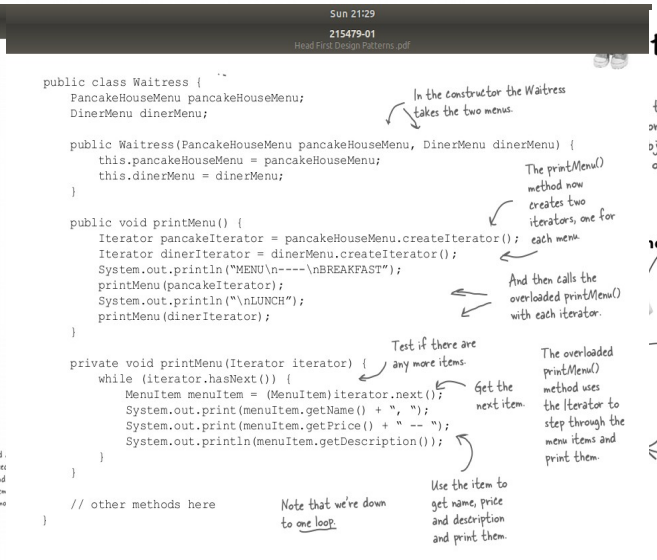
```java
public abstract class Room implements IStructure {
    public String name;

    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;
    }
}
```

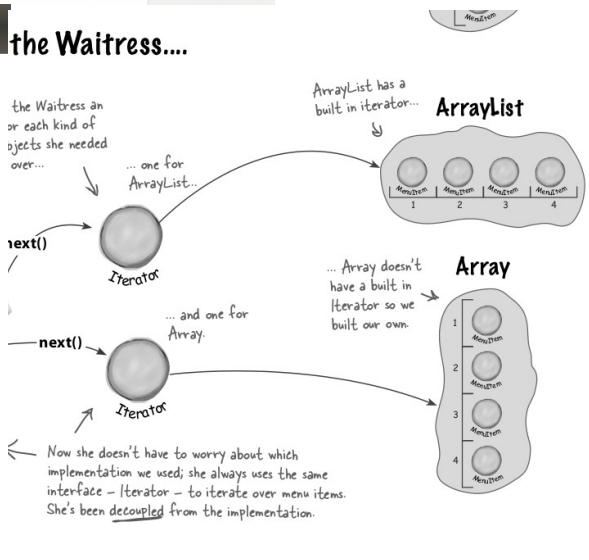
## The Iterator Pattern

- 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



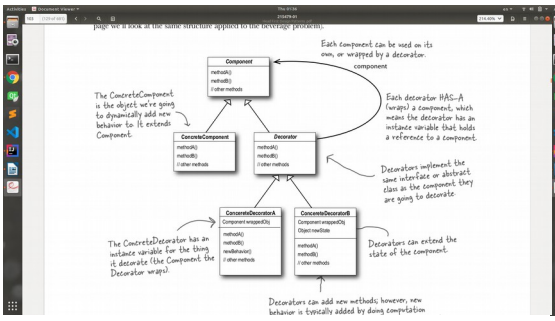



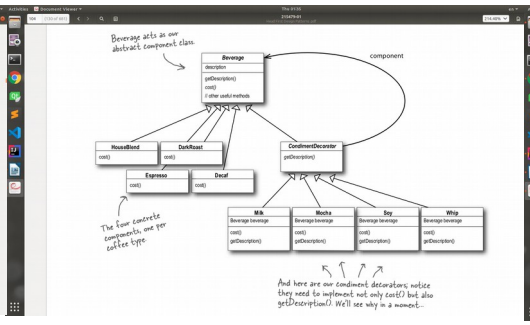



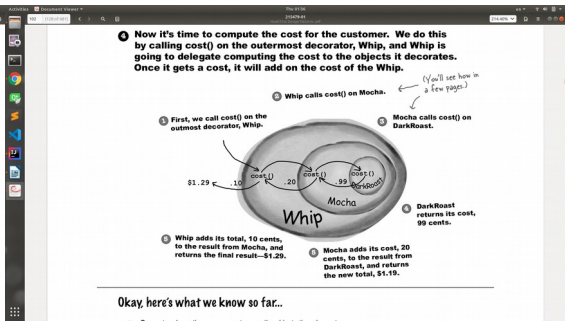



## Decorator Pattern

- 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)
- **<u>Used when</u>**
  - 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)
- **<u>NOTE</u>**: 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

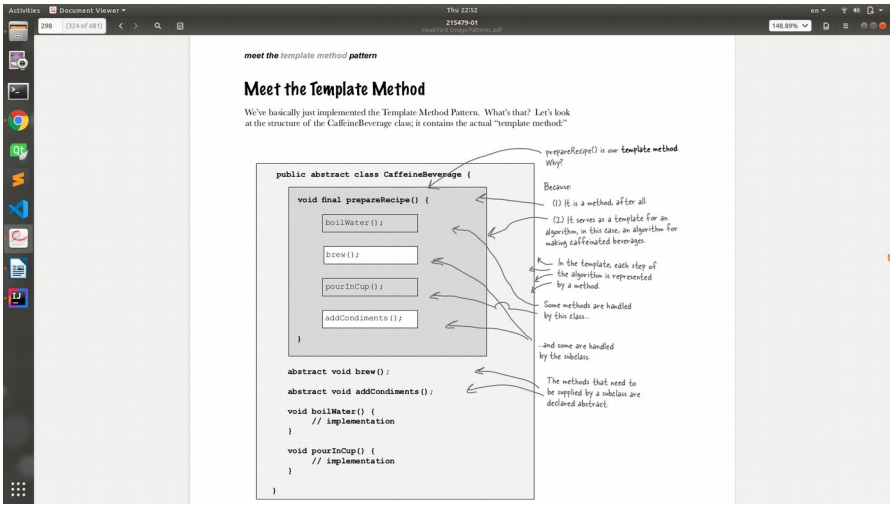
## Template Pattern

- **Encapsulating Algorithms**

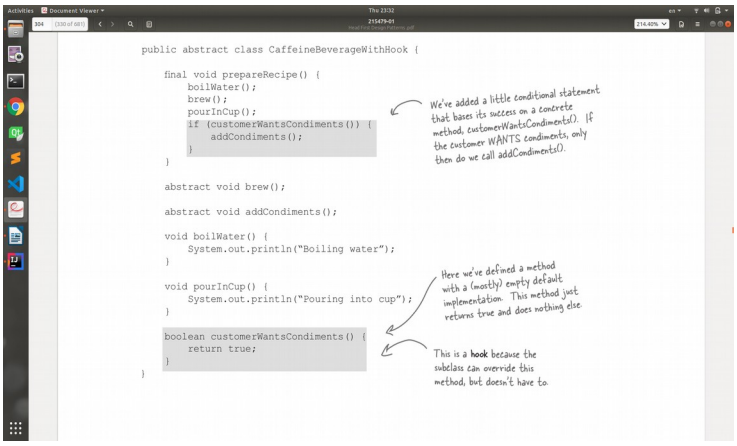
- **The Template Method defines the steps of an algorithm and allows sub-classes to provide the implementation for one or more steps.**

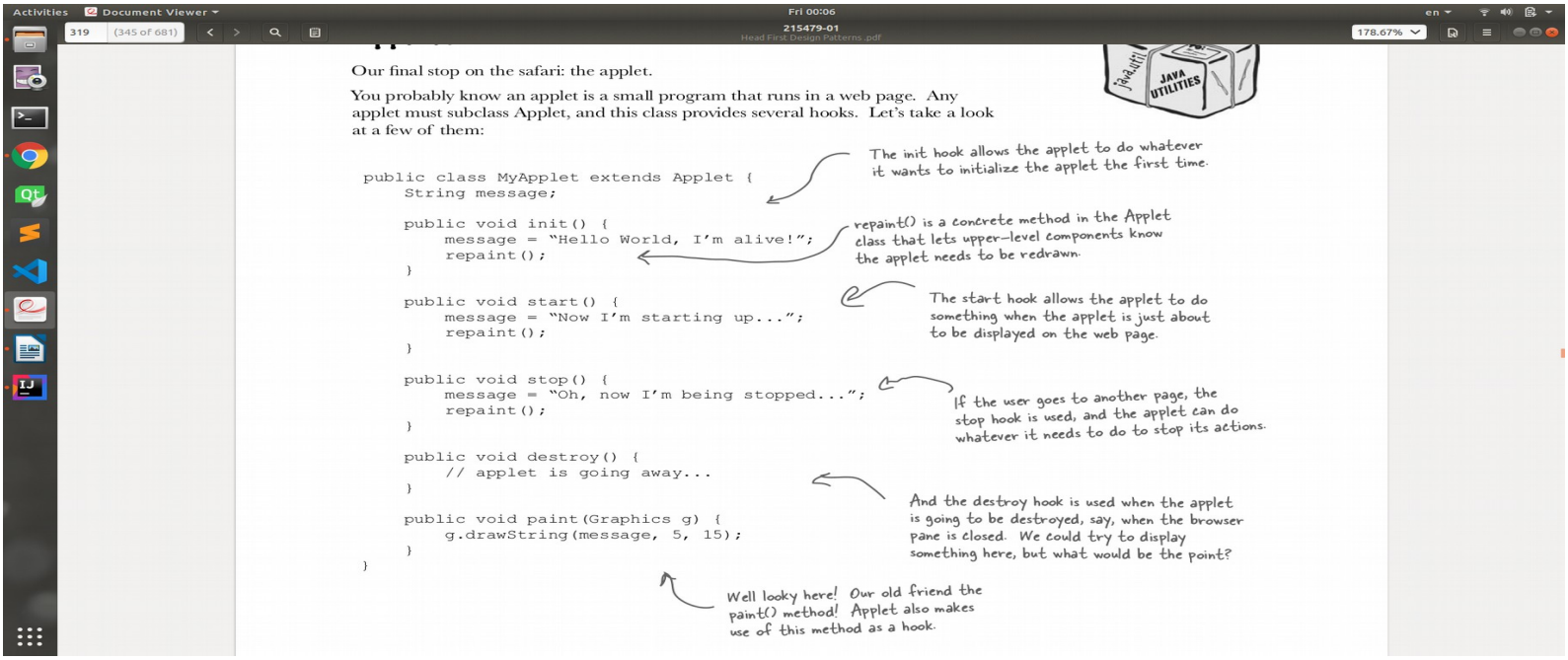
- Class with a method called Template-Method that has the **sequence** of executed **methods**.
- Used with **classes** with **similar sequence** of executed methods, so we can have whole different scenarios of sequence of events or methods, and dynamically choose one of them.
- **Subclasses decide how to implement steps in the algorithm.**

- It is not just a **polymorphism**, we don't execute the template method its self based on concrete classes, we do that in some of the functions it call 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 calls it
  - It is optional to the subclass to implement or not to add functionality to the algorithm.
  - Template method call the hook function with a condition so the control of the algorithm remain with the template, template determines to use the hook or not





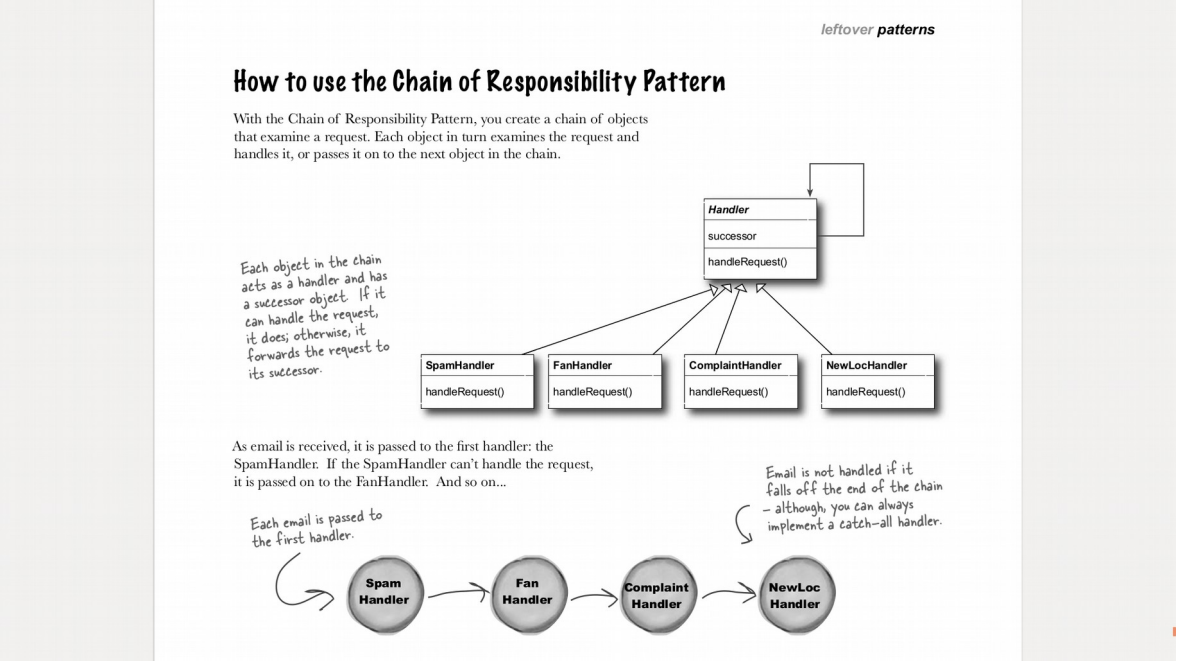

The Strategy and Template Method Patterns both encapsulate algorithms, one by inheritance and one by composition.

## 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



leftover patterns

## How to use the Chain of Responsibility Pattern

With the Chain of Responsibility Pattern, you create a chain of objects that examine a request. Each object in turn examines the request and handles it, or passes it on to the next object in the chain.

Each object in the chain acts as a handler and has a successor object. If it can handle the request, it does; otherwise, it forwards the request to its successor.



As email is received, it is passed to the first handler: the SpamHandler. If the SpamHandler can't handle the request, it is passed on to the FanHandler. And so on...

Each email is passed to the first handler.

Email is not handled if it falls off the end of the chain – although, you can always implement a catch-all handler.

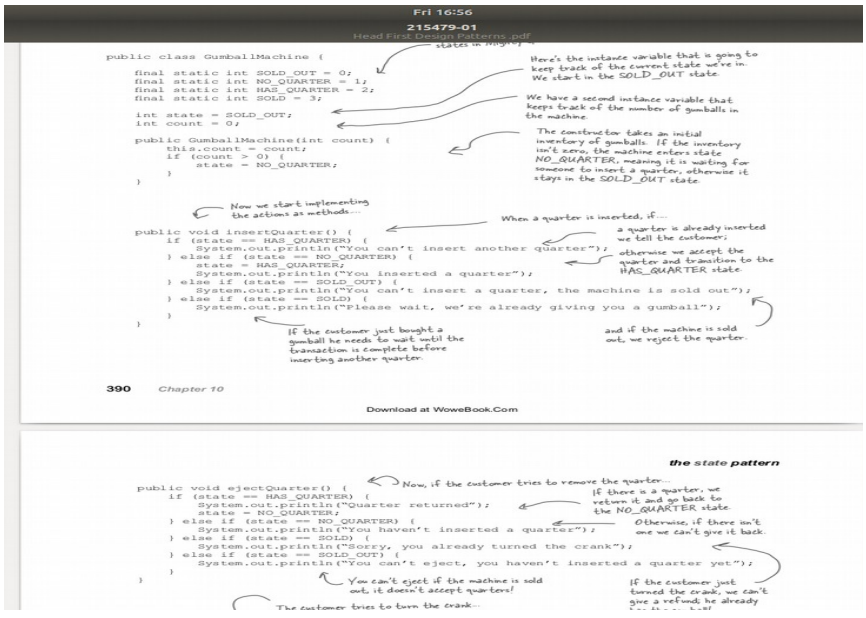


## State Pattern

- 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

```
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 = 3;

    int state = SOLD_OUT;
    int count = 0;

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

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

Here's the instance variable that is going to keep track of the current state we're in. We start in the SOLD_OUT state.

We have a second instance variable that keeps track of the number of gumballs in the machine.

The constructor 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.

Now we start implementing the actions as methods...

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.

If the customer just bought a gumball he needs to wait until the transaction is complete before inserting another quarter.

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");
        state = NO_QUARTER;
    } else if (state == NO_QUARTER) {
        System.out.println("You haven't inserted a quarter");
    } else if (state == SOLD) {
        System.out.println("Sorry, you already turned the crank");
    } else if (state == SOLD_OUT) {
        System.out.println("You can't eject, you haven't inserted a quarter yet");
    }
}
```

Now, if the customer tries to remove the quarter...

If there is a quarter, we return it and go back to the NO_QUARTER state.

Otherwise, if there isn't one we can't give it back.

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

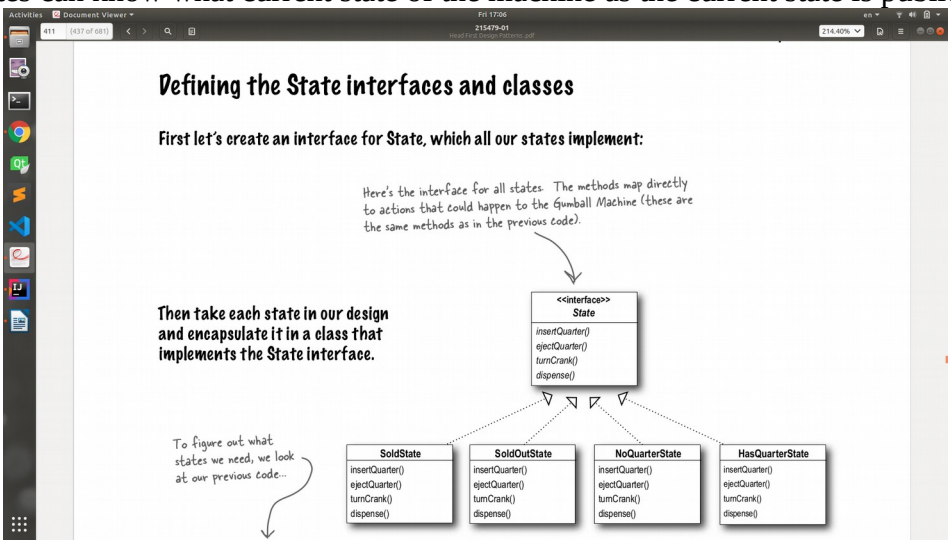
If the customer just turned the crank, we can't give a refund; he already

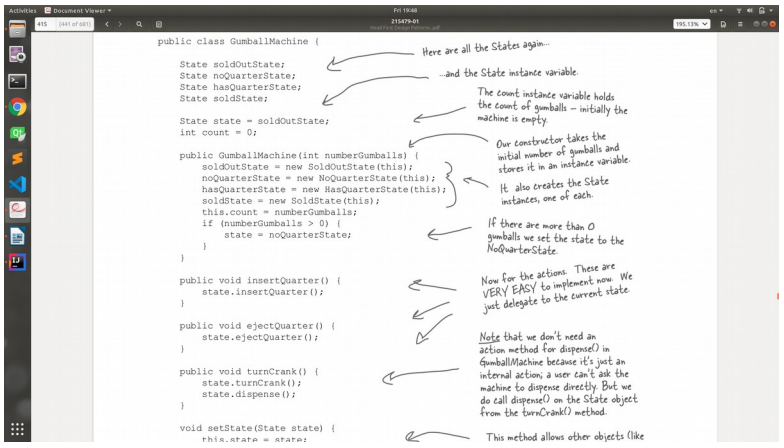
The customer tries to turn the crank...

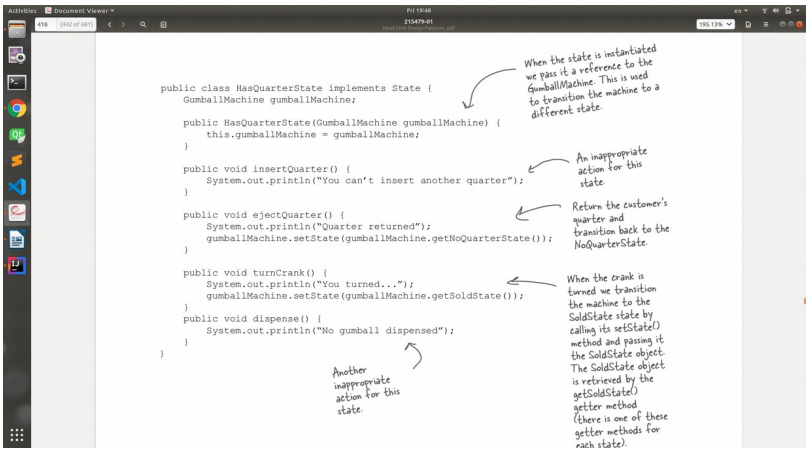
- But when we want to add a state to the system we will need to go to the whole actions(functions) then add the "else if" for that new state
- instead, we will **encapsulate what varies**, we will **implement** the **state** in a **class** that **handles** all its **actions**

- (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 !



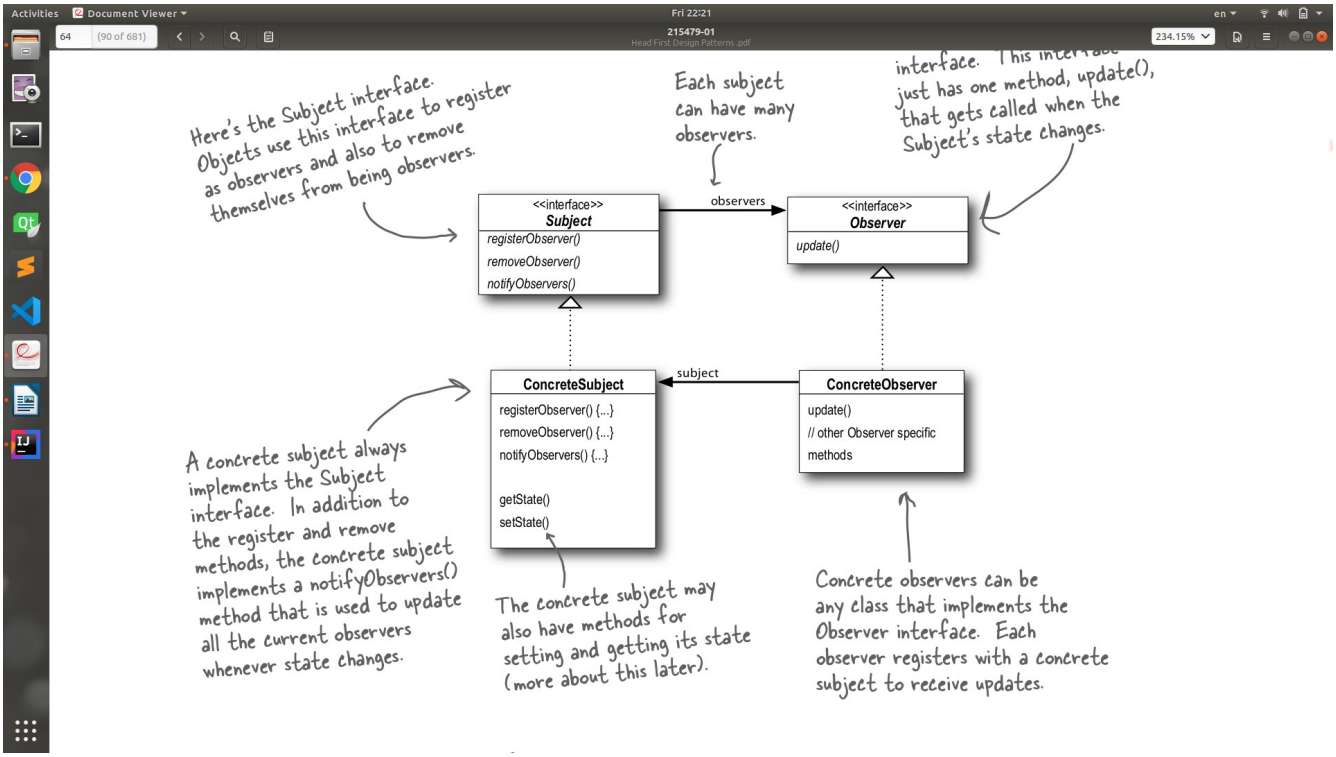




- With this new pattern we can define or add new states dynamically and modify just few code in the other states that their next state is our new defined state

## 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.

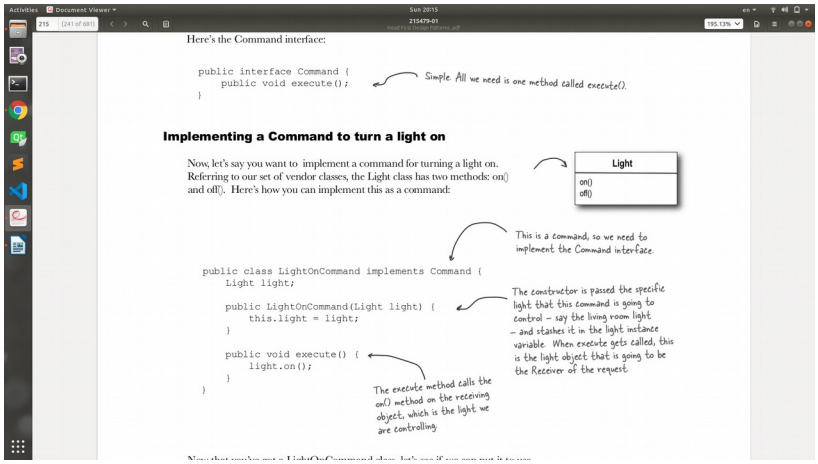
- The power of Loose Coupling
  - When two objects are loosely coupled, they can interact, but have very little knowledge of each other.
  - The Observer Pattern provides an object design where subjects and observers are loosely coupled.

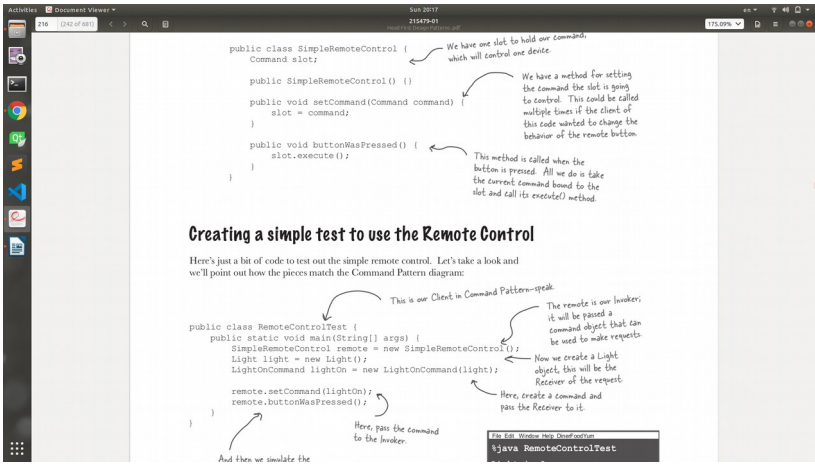



## Command Pattern

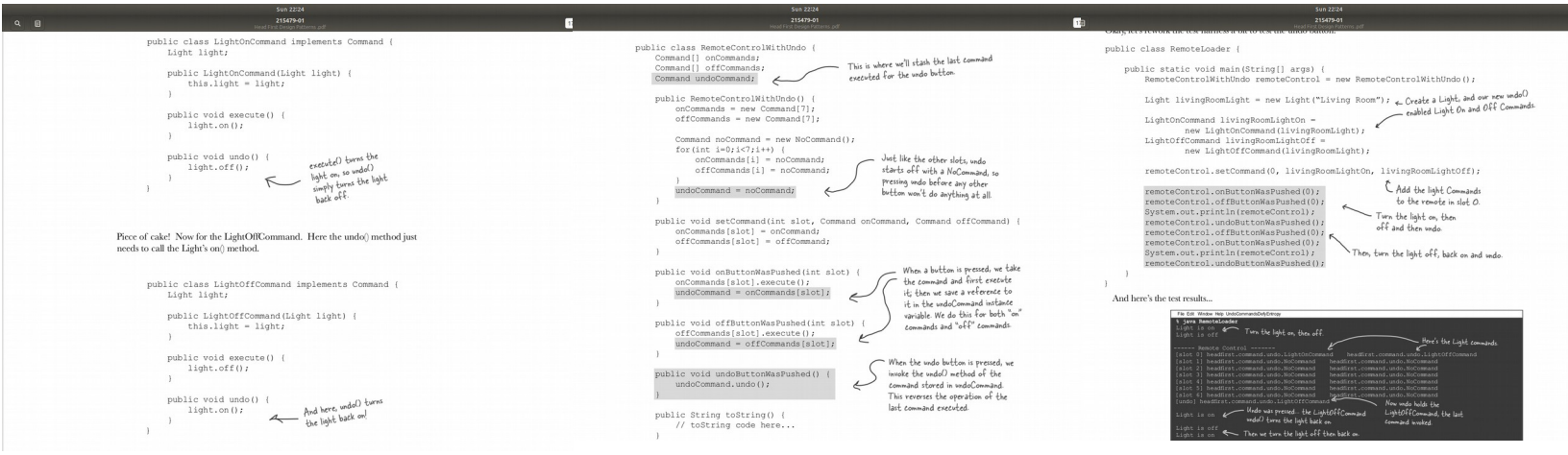
- Decoupling , Undo Operations
- Decoupling Objects : **encapsulate** the **request** (**command**) between 2 objects so the 2 objects don't know the details of each other
  - **ex**: waiter and cook, the waiter dosen't care about how the food is cook,this request or command is encapsulated in "order" 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**



- **Undo Operations**



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

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

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

    public void undo() {
        light.off();
    }
}
```

execute() 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!

```
public class RemoteControlWithUndo {
    Command[] onCommands;
    Command[] offCommands;
    Command undoCommand;

    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;
    }

    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...
    }
}
```

This is where we'll stash the last command executed for the undo button.

Just like the other slots, undo starts off with a NoCommand, so pressing undo before any other button won't do anything at all.

When a button is pressed, we take the command and first execute it; then we save a reference to it in the undoCommand instance variable. We do this for both "on" commands and "off" commands.

When the undo button is pressed, we invoke the undo() method of the command stored in undoCommand. This reverses the operation of the last command executed.

```
public class RemoteLoader {

    public static void main(String[] args) {
        RemoteControlWithUndo remoteControl = new RemoteControlWithUndo();

        Light livingRoomLight = new Light("Living Room");

        LightOnCommand livingRoomLightOn =
                new LightOnCommand(livingRoomLight);
        LightOffCommand livingRoomLightOff =
                new LightOffCommand(livingRoomLight);

        remoteControl.setCommand(0, livingRoomLightOn, livingRoomLightOff);

        remoteControl.onButtonWasPushed(0);
        remoteControl.offButtonWasPushed(0);
        System.out.println(remoteControl);
        remoteControl.undoButtonWasPushed();
        remoteControl.offButtonWasPushed(0);
        remoteControl.onButtonWasPushed(0);
        System.out.println(remoteControl);
        remoteControl.undoButtonWasPushed();
    }
}
```

Create a Light, and our new undo() enabled Light On and Off Commands.

Add the light Commands to the remote in slot 0.

Turn the light on, then off and then undo

Then, turn the light off, back on and undo.

And here's the test results...

```
% java RemoteLoader
Light is on
Light is off

------ Remote Control -------
[slot 0] headfirst.command.undo.LightOnCommand    headfirst.command.undo.LightOffCommand
[slot 1] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 2] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 3] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 4] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 5] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[slot 6] headfirst.command.undo.NoCommand    headfirst.command.undo.NoCommand
[undo] headfirst.command.undo.LightOffCommand

Light is on
Light is off
Light is on
```

Turn the light on, then off

Here's the Light commands

Undo was pressed. the LightOffCommand undo() turns the light back on.

Now undo holds the LightOffCommand, the last command invoked.

Then we turn the light off then back on.

## 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.

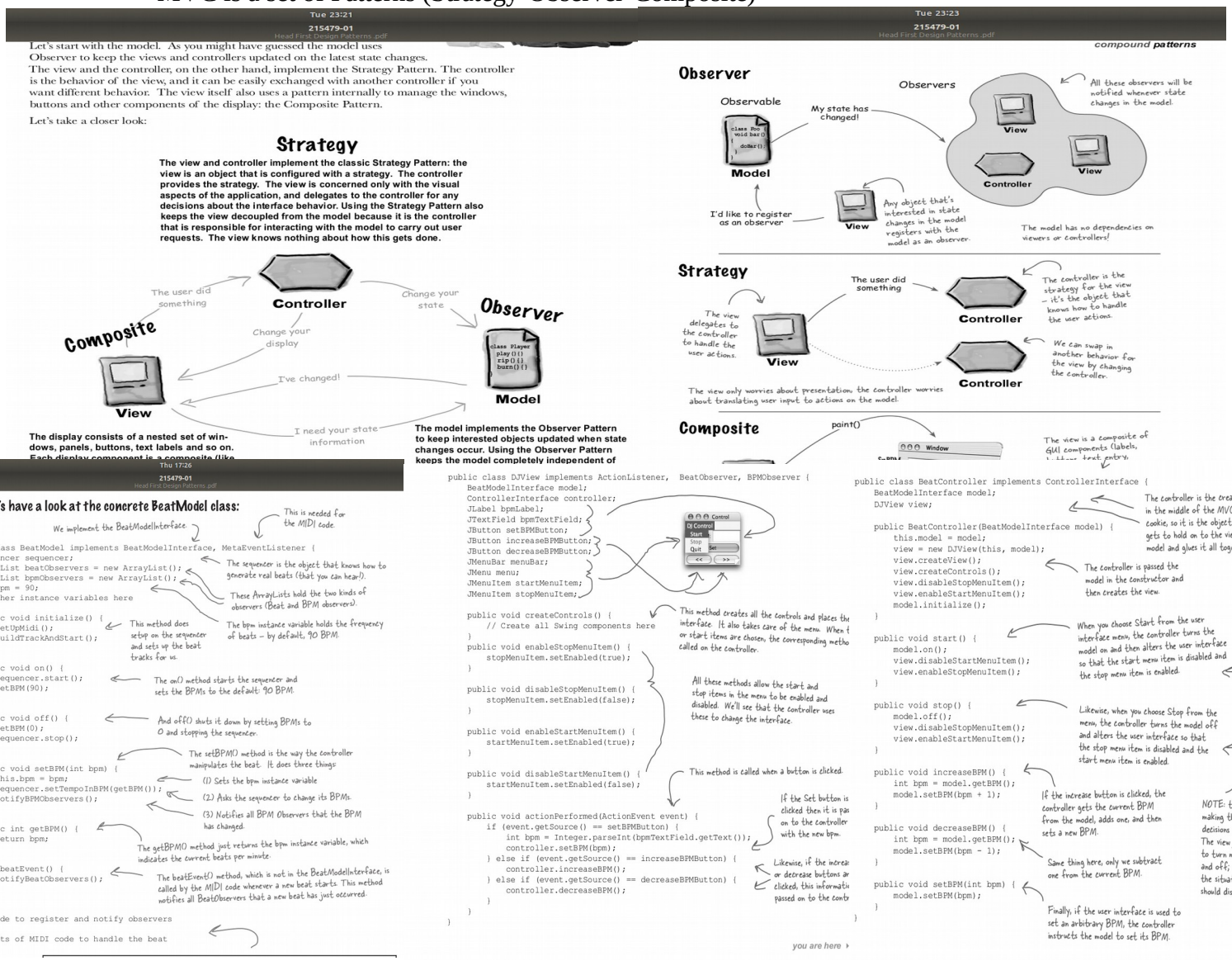
- **Design Principles**
  - **Separation of Concerns**
  - **Entity Objects (Model) , Control Object (Controller) , Boundary Objects (View)**
  - **Decoupling View and Model**
  - **MVC can be used in any UI Software System**

- 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)




- 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.