COMP2511 Notes

Alperen Onur

November 2024

1 Factory Pattern

Instead of constructing different objects in one class, call a factory method which returns the object of type which you want. To make sure that the products made by a factory method all perform some relative shared functionality, we make sure each "sub-factory" implement a shared interface.

```
public interface CarBuilder {

    void setType(Type type);
    void setSeats(int seats);
    void setEngine(Engine engine);
    void setTransmission(Transmission transmission);
    void setTripComputer(TripComputer tripComputer);
    void setGPSNavigator(GPSNavigator gpsNavigator);

}
```

Figure 1: Builder Interface

Now, whenever we want to make a particular object, we call the factory method which has a switch statement that handles which sub-factory to call.

```
public static Button getButton(String platform) {
        Button btn = null;

        if (platform.equalsIgnoreCase("Html")) {
                btn = new ButtonHtml();
        } else if (platform.equalsIgnoreCase("Windows 10")) {
                btn = new ButtonWin10();
        } else if (platform.equalsIgnoreCase("MacOs")) {  //
                btn = new ButtonMacOs();
        } else if (platform.equalsIgnoreCase("Linux")) {
                btn = new ButtonLinux();
        } else {
                new Exception("Unknwon platform type!");
        }

        return btn;
}
```

Figure 2: Factory method in a factory class

So any client that would like their object to create objects from our factory, they should create an instance of our factory (factory = new Factory) then call the factory method (factory.createSomething("ID")) to create that class.

```
public class ButtonHtml implements Button {
        private String label = "";
        private String colour = "";


        @Override
        public String getLabel() {
                return this.label;
        }

        @Override
        public void setLabel(String label) {
                this.label = label;
        }

        @Override
        public void setBgColour(String colour) {
                this.colour = colour;
        }
```

Figure 3: Sub-Factory implementation

2 Abstract factory

There will be scenarios where a client want's to make many different objects of different types. The key here is that objects of different types may have the same general functionality. For example, both MAC and Linux operating systems may need a button object. Now, MAC and Linux are of different types, but they both want to create a button object. MAC buttons and Linux buttons must be different and as such, an abstract factory (ie- A factory of factories) makes sense here. The client creates instance of factories for all the types they made need. Now when they want a specific object of a certain type, all they need to do is call the corresponding factory and it method to make some object. The implementation of this is similar to the factory method. For each factory we might need (MAX, Linux, Windows, etc), we define a set of methods that each factory should have. These should define the objects that each factory may need to make.

```
public interface GUIFactory {

        public Button    getButton();
        public CheckBox  getCheckBox();

```

Figure 4: Abstract factory interface

Now for a factory of a particular type, we implement this interface.

```
public class GUIFactoryLinux implements GUIFactory {

        @Override
        public Button getButton() {
                return new ButtonLinux();
        }

        @Override
        public CheckBox getCheckBox() {
                return new CheckBoxLinux();
        }

}
```

Figure 5: A factory that makes objects of a particular type

This factory should create specific objects that are exclusive and match the type that the factory is creating. We do this by having these objects implement an interface that describes the general functionality of the object.

```
public interface Button {

        public void setLabel(String labelText);
        public void click();

}
```

Figure 6: Interface for a general button object

Now for our specific factory, all it needs to do is create a object that implements this interface such that it fullfills the requirements of the type we're trying to make.

```
public class ButtonLinux implements Button {

        private String labelText = "";
        @Override
        public void setLabel(String labelText) {
                this.labelText = labelText;
                System.out.println("ButtonLinux: Setting labe

        }

        @Override
        public void click() {

                System.out.println("ButtonLinux: clicking but

        }
```

Figure 7: An object of particular type that our specific factory will create

3 Builder Pattern

Builder patterns are used when a client needs to make some different permutation of traits of a single class multiple times. Say a client wants to make a car with a manual engine, sports mode, 2 seats & silver color as well as a car with a automatic engine, air conditioning, 4 seats & blue color. This can be encapsulated in one car class, but it will be a long constructor with clients that want to make this car object needing to add null as several parameters to make the object they want. In these cases a builder pattern is perfect for a easing the process of class construction for a client. We begin by making a builder interface which covers all different configurations for constructing an object.

```
public interface CarBuilder {

    void setType(Type type);
    void setSeats(int seats);
    void setEngine(Engine engine);
    void setTransmission(Transmission transmission);
    void setTripComputer(TripComputer tripComputer);
    void setGPSNavigator(GPSNavigator gpsNavigator);

}
```

Figure 8: Builder Interface

Builder classes implement this interface such that whenever a specific object needs to be built.

```
public class AbcCarBuilder implements CarBuilder {
        private Type type;
        private int seats;
        private Engine engine;
        private Transmission transmission;
        private TripComputer tripComputer;
        private GPSNavigator gpsNavigator;

        @Override
        public void setType(Type type) {
                this.type = type;
        }

        @Override
        public void setSeats(int seats) {
                this.seats = seats;
        }

        @Override
        public void setEngine(Engine engine) {
                this.engine = engine;
        }
```

Figure 9: Car Builder

All the client needs to do is create an instance of this builder class and use setters to specify what type of object.

```
AbcCarBuilder builder1 = new AbcCarBuilder();

builder1.setType(Type.SPORTS_CAR);
builder1.setSeats(2);
builder1.setEngine(new Engine(3.0, 0));
builder1.setTransmission(Transmission.SEMI_AUTOMATIC);
builder1.setTripComputer(new TripComputer());
builder1.setGPSNavigator(new GPSNavigator());
```

Figure 10: Builder Pattern example 1

The builder pattern can also have a preset defined building steps that a client may need called a manual.

```
public class Manual {
    private final Type type;
    private final int seats;
    private final Engine engine;
    private final Transmission transmission;
    private final TripComputer tripComputer;
    private final GPSNavigator gpsNavigator;

    public Manual(Type type, int seats, Engine engine, Transmi
                  TripComputer tripComputer, GPSNavigator gpsN
        this.type = type;
        this.seats = seats;
        this.engine = engine;
        this.transmission = transmission;
        this.tripComputer = tripComputer;
        this.gpsNavigator = gpsNavigator;
    }
```

Figure 11: Manual

A manual is not much different to a regular builder but to make the object, the manual must be passed into a director; a class that executes the building steps.

```
public class CarDirector {

    public void constructSportsCar(CarBuilder builder) {
        builder.setType(Type.SPORTS_CAR);
        builder.setSeats(2);
        builder.setEngine(new Engine(3.0, 0));
        builder.setTransmission(Transmission.SEMI_AUTOMATIC);
        builder.setTripComputer(new TripComputer());
        builder.setGPSNavigator(new GPSNavigator());
    }

    public void constructCityCar(CarBuilder builder) {
        builder.setType(Type.CITY_CAR);
        builder.setSeats(2);
        builder.setEngine(new Engine(1.2, 0));
        builder.setTransmission(Transmission.AUTOMATIC);
        builder.setTripComputer(new TripComputer());
        builder.setGPSNavigator(new GPSNavigator());
    }
```

Figure 12: Builder Director

These manuals have these own builders and those are the builders that should be sent to the director.

```
public class AbcCarManualBuilder implements CarBuilder{
    private Type type;
    private int seats;
    private Engine engine;
    private Transmission transmission;
    private TripComputer tripComputer;
    private GPSNavigator gpsNavigator;

    @Override
    public void setType(Type type) {
        this.type = type;
    }

    @Override
    public void setSeats(int seats) {
        this.seats = seats;
    }

    @Override
    public void setEngine(Engine engine) {
        this.engine = engine;
    }
```

Figure 13: Builder Manual

Now, all the client should do is pass an empty manual into the director which defines the method (the series of steps rather) to build the specific object.

```
AbcCarManualBuilder manualBuilder = new AbcCarManualBuilder()

// Director may know several building recipes.
director.constructCityCar(manualBuilder);
Manual carManual = manualBuilder.getResult();
System.out.println( "\n" + carManual.print() );
```

Figure 14: Builder Pattern example 2

4 Singleton Pattern

While clients use your app/program, they may be accessing the same object at the same time (if different clients are accessing the same class multiple times). This can lead to issues when any change is undergone by these classes since come clients may be working with the updated version and others may not be. A singleton pattern aims to fix this issue by ensuring that the frequently accessed class has a single created instance. If it doesn't exits, we create it for the first time and otherwise we return that object.

```
public class MySingleton {

        private static MySingleton single_instance = null ;
        private static String  db = null;

        private LocalTime localTime;

        private MySingleton() {
                localTime = LocalTime.now();
                /**
                 * code to create db connection
                 * String db = ???
                 *
                 */
        }

        public static synchronized MySingleton getInstance() {
                if (single_instance == null) {

                        single_instance = new MySingleton();
                }
                return single_instance;
        }

        public LocalTime getTime() {
                return localTime;
        }
```

Figure 15: Singleton Class

A key note here is that the singleton class has a private constructor. This means that this class can only be created by it's own method which verifies a single instance of the class. Another note is the syncronised keyword in the method signature for getting the class, this ensures thread locking between multiple threads/processes accessing the class.

5 Adapter Pattern

Imagine a scenario where a client refuses to update to a new version of your app and you've recently implemented a change such that your backend will only accept data of a particular format. The client with the old version of the app still needs to use it however so we can support backwards compatibility with an adapter pattern. The adapter pattern works by interacting with existing client interfaces that define shared methods between classes and objects that implement those interfaces (the objects that we'll need to convert data between).

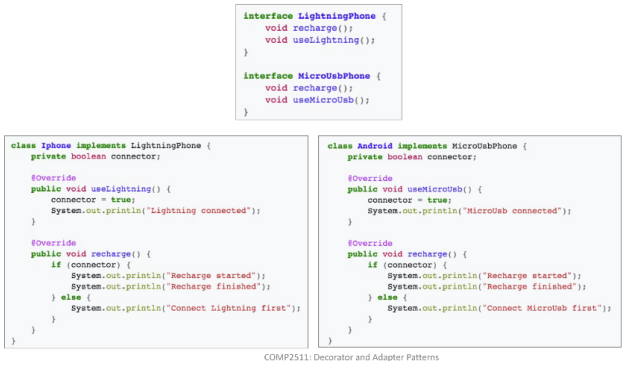

Figure 16: Client interface with classes that may need to interact

For the several client classes to be able to interact with each other, we'll need an adapter that implements the same functionality which those client classes implement.

```
class LightningToMicroUsbAdapter implements MicroUsbPhone {
    private final LightningPhone lightningPhone;

    public LightningToMicroUsbAdapter(LightningPhone lightningPhone) {
        this.lightningPhone = lightningPhone;
    }

    @Override
    public void useMicroUsb() {
        System.out.println("MicroUsb connected");
        lightningPhone.useLightning();
    }

    @Override
    public void recharge() {
        lightningPhone.recharge();
    }
}
```

Figure 17: Adapter which allows devices to use a microUSB

Now a client can create a class of some type and use the functionality of a different class via the adapter. In this example, the adapter is implemented in one direction but it's possible to implement it both ways.

6 Composite Pattern

The composite is a good way to solve patterns of iteratively going through objects for some query. It uses a recursive approach instead to query the next object in the chain and repeat until some base case is reached. In the composite patterns context, a composite node contains a list of children in which we may need to query.

```java
public class Composite implements Component {

	private String name;
	private double cost;

	ArrayList<Component>  children = new ArrayList<Component>();

	public Composite(String name, double cost) {
		super();
		this.name = name;
		this.cost = cost;
	}

	@Override
	public double calculateCost() {
		double answer = this.getCost();
		for(Component c : children) {
			answer += c.calculateCost();
		}

		return answer;
	}
```

Figure 18: Composite Node

The simple case is that these children are lead nodes which simply return the queried value.

```java
public class Leaf implements Component {

	private String name;
	private double cost;


	public Leaf(String name, double cost) {
		super();
		this.name = name;
		this.cost = cost;
	}

	@Override
	public String nameString() {
		return this.name;
	}

	@Override
	public double calculateCost() {
		return this.cost;
	}
```

Figure 19: Leaf Node

However, these children can be either composite nodes or leaf nodes but a key thing here is that is that the specific composite node doesn't care since both the composite and leaf nodes implement the same interface.

```java
public interface Component {

        public String nameString();
        public double calculateCost();

}
```

Figure 20: Composite Interface

7 Decorator Pattern

The decorator pattern is a brilliant pattern to consider when a client needs to edit their object during it's runtime. Say the client wants to add a feature to a class during its runtime. Inheritance can not solve this since inheritance is static, the object either has it or not as it's all initlised when the program begins. On top of this issue, subclasses can only extend a single class and as such, only so many features can be implemented by continuously extending classes. Using aggregation instead, we can wrap objects around others to implement this functionality instead. This is the key idea behind the Decorator pattern. We begin by making a base class for our client, there can be several different base cases with which a client can decorate.

```java
public class HouseBlend extends Beverage {
        public HouseBlend() {
                description = "House Blend Coffee";
        }

        public double cost() {
                return .89;
        }
}
```

Figure 21: Base Decorator

To customise this base, the decorators accept a base decorator as a parameter in its constructor such that it wraps around the base. The client will only interact with the decorator that wraps all other decorators. These decorators also query information recursively.

```
public class Mocha extends CondimentDecorator {
        Beverage beverage;

        public Mocha(Beverage beverage) {
                this.beverage = beverage;
        }

        public String getDescription() {
                return beverage.getDescription() + ", Mocha";
        }

        public double cost() {
                double beverage_cost = beverage.cost();

                System.out.println("Mocha: beverage.cost() is: " + beverage_cost );
                System.out.println("  - adding One Mocha cost of 0.20c ");
                System.out.println("  - new cost is: " + (0.20 + beverage_cost ) );

                return 0.20 + beverage_cost ;
        }
}
```

Figure 22: Wrapping Decorator

The only reason we can do this is that both base and wrapped decorators extend the same interface, they are treated as the same to each other and the client!

```
public abstract class CondimentDecorator extends Beverage {
        public abstract String getDescription();

}
```

Figure 23: Base Interface

```
public abstract class Beverage {
        String description = "Unknown Beverage";

        public String getDescription() {
                return description;
        }

        public abstract double cost();
}
```

Figure 24: Parent of all decorators

The client creates a a base class and now can freely and easily customise it however they like! The client will only every interact with the outer most wrapped decorator and the pattern does most of the work to query information.

```
public class StarbuzzCoffee {

        public static void main(String args[]) {

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

                Beverage beverage = new Espresso();
                System.out.println(beverage.getDescription()
                                + " $" + beverage.cost());
                System.out.println("------------------------- ");
                Beverage beverage2 = new DarkRoast();
                beverage2 = new Mocha(beverage2);
                beverage2 = new Mocha(beverage2);
                beverage2 = new Whip(beverage2);
                System.out.println(beverage2.getDescription()
                                + " $" + beverage2.cost());

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

                Beverage beverage3 = new HouseBlend();
                beverage3 = new Soy(beverage3);
                beverage3 = new Mocha(beverage3);
                beverage3 = new Whip(beverage3);
                System.out.println(beverage3.getDescription()
                                + " $" + beverage3.cost());
                System.out.println("------------------------- ");
```

Figure 25: Decorator Pattern used by client

8 Facade Pattern

The Facade pattern is a useful design pattern that allows a client to easily use a functionality of a complex library. Instead of a client going through all the necessary steps to convert a file to an mp4 for example, the client creates an instance of a facade which interacts with the relevant libraries in order to achieve the same outcome.

```
// We create a facade class to hide the framework's complexity
// behind a simple interface. It's a trade-off between
// functionality and simplicity.
class VideoConverter is
    method convert(filename, format):File is
        file = new VideoFile(filename)
        sourceCodec = (new CodecFactory).extract(file)
        if (format == "mp4")
            destinationCodec = new MPEG4CompressionCodec()
        else
            destinationCodec = new OggCompressionCodec()
        buffer = BitrateReader.read(filename, sourceCodec)
        result = BitrateReader.convert(buffer, destinationCodec)
        result = (new AudioMixer()).fix(result)
        return new File(result)

// Application classes don't depend on a billion classes
// provided by the complex framework. Also, if you decide to
// switch frameworks, you only need to rewrite the facade class.
class Application is
    method main() is
        convertor = new VideoConverter()
        mp4 = convertor.convert("funny-cats-video.ogg", "mp4")
        mp4.save()
```

Figure 26: Facade pattern example

A key concept of the Facade pattern is that it doesn't encapsulate the underlying interface and tools. The client can still interact with them directly if they'd like a specific interaction. Otherwise, you'd need to make a Facade class for each possible function the client might use and that is not maintainable.

9 Iterator pattern

An iterable is a object that can be iterated over (by an iterator). All iterators are iterable but not all iterables can be iterators. The iterator pattern gives us a way to define different iterators for some collection of objects that we want to iterate over. Not all collections of objects will be as simple as a list and as such we need to define ways to go through these objects.

```
public class PancakeHouseMenu implements Menu {
        ArrayList<MenuItem> menuItems;

        public PancakeHouseMenu() {
                menuItems = new ArrayList<MenuItem>();

                addItem("K&B's Pancake Breakfast",
                        "Pancakes with scrambled eggs, and toast",
                        true,
                        2.99);

                addItem("Regular Pancake Breakfast",
                        "Pancakes with fried eggs, sausage",
                        false,
                        2.99);

                addItem("Blueberry Pancakes",
                        "Pancakes made with fresh blueberries",
                        true,
                        3.49);

                addItem("Waffles",
                        "Waffles, with your choice of blueberries or strawberries",
                        true,
                        3.59);
        }

        public void addItem(String name, String description,
                            boolean vegetarian, double price)
        {
                MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
                menuItems.add(menuItem);
        }

        public ArrayList<MenuItem> getMenuItems() {
                return menuItems;
        }

        public Iterator<MenuItem> createIterator() {
                return new PancakeHouseMenuIterator(menuItems);
        }

        public String toString() {
                return "Objectville Pancake House Menu";
        }
```

Figure 27: Custom Object

A defined iterator will give us a way to traverse the collection in different ways depending on what the client needs. An iterator for a tree may choose to DFS and another may choose to BFS for example. Each iterator tracks it's own progress and as such we can even traverse the same object with different iterators at the same time.

```
public class PancakeHouseMenuIterator implements Iterator<MenuItem> {
        ArrayList<MenuItem> items;
        int position = 0;

        public PancakeHouseMenuIterator(ArrayList<MenuItem> items) {
                this.items = items;
        }

        public MenuItem next() {
                MenuItem object = items.get(position);
                position = position + 1;
                return object;
        }

        public boolean hasNext() {
                if (position >= items.size()) {
                        return false;
                } else {
                        return true;
                }
        }
}
```

Figure 28: Custom iterator

A key point here is that these iterators implement a shared interface that defines what each iterator needs to do.

```
public interface Iterator<T> {
        boolean hasNext();
        T next();
}
```

Figure 29: Iterator interface

A client now can easily work with the collection and iterators via their interfaces. There is no coupling of the code here that locks a client to a specific class since the iterators implement the same iterator class. They can choose to iterator through any of the collections they'd like to as long as those iterators exits and the collections contain the same elements.

```
private void printMenu(Iterator<MenuItem> iterator) {
        while (iterator.hasNext()) {
                MenuItem menuItem =  iterator.next();
                System.out.print(menuItem.getName() + ", ");
                System.out.print(menuItem.getPrice() + " -- ");
                System.out.println(menuItem.getDescription());
        }
}
```

Figure 30: Client using iterator

10 Observer Pattern

The Observer Pattern is a way we can design proper Event Driven Programming; when an event happens then do something. An issue in event driven programming is, which classes should an event communicate with? Not all classes might care about the specific event that's taken place. You could also have each class check the object where events happen and see if they are interested in that particular one, but this highly couples the event class with all others making the code unmaintainable. We can solve this using a Publisher; a class required for the implementation of an observer pattern.

The publisher acts as a notifier for an event to all other classes that have subscribed to it. It does this by keeping an array of all these subscribers that are interested in the event.

```
public class Thermometer implements SubjectThermometer {

        ArrayList<ObserverThermometer> listObservers = new ArrayList<ObserverThermometer>();
        double temperatureC = 0.0;

        @Override
        public void attach(ObserverThermometer o) {
                if(! listObservers.contains(o)) { listObservers.add(o); }
        }

        @Override
        public void detach(ObserverThermometer o) {
                listObservers.remove(o);
        }

        @Override
        public void notifyObservers() {
                for( ObserverThermometer obs : listObservers) {
                        obs.update(this);
                }
        }
```

Figure 31: Observer Publisher

All classes that might be interested in this event must implement a subscriber interface that has methods relevant to subscribing or unsubscribing from an event.

```
public class DisplayUSA implements ObserverHygrometer , ObserverThermometer {
        Subject subject;
        double temperatureC = 0.0;
        double humidity = 0.0;

        @Override
        public void update(SubjectHygrometer obj) {

                this.humidity = obj.getHumidity();
                displayHumidity();

        }

        @Override
        public void update(SubjectThermometer obj) {
                        this.temperatureC = obj.getTemperatureC();
                        displayTemperature();
        }


        public void display() {
                System.out.printf("From DisplayUSA: Temperature is %.2f F, "
                                + "Humidity is %.2f\n", convertToF(), humidity);
        }

        public void displayTemperature() {
                System.out.printf("From DisplayUSA: Temperature is %.2f F\n",
                                 convertToF() );
        }

        public void displayHumidity() {
                System.out.printf("From DisplayUSA: Humidity is %.2f\n", humidity);
        }

        public double convertToF() {
                return (temperatureC *(9.0/5.0) + 32);
        }
}
```

Figure 32: Observer Class

Whenever a client is interacting this these classes, they might want a class to be interested in some event and so all they need to do is subscribe them to that event.

```
public interface SubjectThermometer {

        public void attach(ObserverThermometer o);
        public void detach(ObserverThermometer o);
        public void notifyObservers();

        public double getTemperatureC();


}
```

Figure 33: Observer Interface

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

        Thermometer thermo = new Thermometer();
        DisplayUSA usaDisplay = new DisplayUSA();
        thermo.attach(usaDisplay);

        DisplayAustralia ausDisplay = new DisplayAustralia();
        thermo.attach(ausDisplay);

        System.out.println("\n----------------- thermo.setTemperatureC(30) ------------ ");
        thermo.setTemperatureC(30);
        System.out.println("\n----------------- thermo.setTemperatureC(12) ------------ ");
        thermo.setTemperatureC(12);
```

Figure 34: Client using observer pattern

11 State Pattern

A state pattern is a well designed implementation of a Finite-State Machine. These machines described a set amount of states and the transitions between states. To implement something like this, you cold have the class that is changing do all the work of switching states after some method is executed. This delegates all the work for transitioning a state to a class that is doing much more work then it really needs to. The state pattern proposes a solution by storing instances of state classes in the object so that when a transition is needed, a simple method call can do the trick.

```
public class GumballMachine {

        State soldOutState;
        State noQuarterState;
        State hasQuarterState;
        State soldState;
        State winnerState;

        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);
                winnerState = new WinnerState(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;
        }
```

Figure 35: Finite amount of states in class

The states need to be aware of each other for this to work and as such, there may only be a finite amount of states. In other words, it's not exactly easy to make more states but the idea is that you should not need to. Each of the individual states now expressed how to transition to another. This reduces the coupling of the main class to the states and delegates individual responsibility to classes.

```
public class HasQuarterState implements State {
        Random randomWinner = new Random(System.currentTimeMillis());
        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...");
                int winner = randomWinner.nextInt(10);
                if ((winner == 0) && (gumballMachine.getCount() > 1)) {
                        gumballMachine.setState(gumballMachine.getWinnerState());
                } else {
                        gumballMachine.setState(gumballMachine.getSoldState());
                }
        }

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

Figure 36: A particular state in the finite state machine

Each of the states need to implement a particular state interface that declares some state specific methods. These can be empty in any state if it doesn't make sense for that state.

```
public interface State {

        public void insertQuarter();
        public void ejectQuarter();
        public void turnCrank();
        public void dispense();
}
```

Figure 37: State Interface

12 Strategy Pattern

Often times you may want a context class to perform many different algorithms to solve different problems. Depending on the situation, the class may act differently to solve these problems. The issue arises when new features need to be added to the class. Well now you need to write up more and more methods that contain algorithms to solve these problems (as well as editing old ones). This makes maintainability extremely difficult. A solution is the make certain algorithms into different strategies that the context can switch between.

```
public class Car {
        protected EngineType engine ;
        protected BrakeStrategy brake;

        /**
         * other fields and methods here
         */

        public Car(EngineType engine, BrakeStrategy brake) {
                this.brake = brake;
                this.engine = engine;
        }

        /**
         * other fields and methods here
         */

        public void brakeApply() {
                brake.apply();
        }

        public void engineRun() {
                engine.run();
        }
}
```

Figure 38: A class with strategies

The context simply accepts a particular strategy and relies on the strategy class to do the work when certain methods are called. This can only work if all the strategies implement the same strategy interface which declares functionality common to all strategies.

```
public interface BrakeStrategy {

                public void apply();

}
```

Figure 39: Strategy Interface

The context class now does not need to care about the specifics about how it needs to act when a certain method is called, that code is all positioned in it's strategy! The great thing is that we can continue to add new strategies to provide additional functionality whilst keeping the context loosely coupled and concise.

13 Template Method

The Template method defines a skeleton of an algorithm, allowing subclasses to use the implemented steps, skip steps of the algorithm and override specific steps without changing the original algorithms structure.

```
public abstract class MyReportTemplate {


        public void genReport() {

                InputStream f1 = openFile();

                SortedMap<String, ArrayList<String>> data = parseFile( f1 );

                generateReport ( data );

                if( isRequestedSummary()) {
                        generateSummary(data);
                }


        }

        public void generateSummary(SortedMap<String, ArrayList<String>> data) {
                System.out.println("generating Summary (default from MyReportTemplat ...");

        }

        public boolean isRequestedSummary() {

                return false;
        }

        public void generateReport(SortedMap<String, ArrayList<String>> data) {
                System.out.println("generating report (default from MyReportTemplat ...");

        }

        protected abstract SortedMap<String, ArrayList<String>> parseFile(InputStream f1);

        public InputStream openFile() {

                String filename = getFilename();
                InputStream f1 = null;

                try {
                        f1 = new FileInputStream(filename);
                } catch (FileNotFoundException e) {
                        e.printStackTrace();
                }

                return f1;
        }

        public abstract String getFilename();


}
```

Figure 40: Template Method Class

Implementations of this algorithm can now choose to implement the algorithm however they like. A key here is that the original template is an abstract class and so some of the methods must be implemented by the subclasses.

```java
public class CSVReport extends MyReportTemplate{
        private String fname = "";
        private boolean reqSummary = false;

        public CSVReport() {
                super();
                fname = "src/example/data.csv";
                reqSummary = false;
        }
        public CSVReport(String filename, boolean requestSummary) {
                this.fname = filename;
                this.reqSummary = requestSummary;
        }

        @Override
        protected SortedMap<String, ArrayList<String>> parseFile(InputStream f1) {
                // CSV parsing code here
                System.out.println("parsing CSV data file: " + getFilename());
                TreeMap<String, ArrayList<String>> data = new TreeMap<String, ArrayList<String>>();
                // populate data object in this method  ..

                return data;
        }

        @Override
        public String getFilename() {
                // ask user for a file name..
                return fname;
        }

        @Override
        public boolean isRequestedSummary() {

                return this.reqSummary;
        }
```

Figure 41: Algorithm that overrides a template method

```java
public class XMLReport extends MyReportTemplate {

        private String fname = "";
        public XMLReport(String filename) {
                fname = filename;
        }

        @Override
        protected SortedMap<String, ArrayList<String>> parseFile(InputStream f1) {
                System.out.println("parsing XML  data file: " + getFilename());
                TreeMap<String, ArrayList<String>> data = new TreeMap<String, ArrayList<String>>();
                // populate data object in this method  ..
                return data;
        }

        @Override
        public String getFilename() {
                return fname;
        }


}
```

Figure 42: Algorithm that skips steps of template method

14 Visitor Pattern

The Visitor Pattern allows you to separate a algorithms from a context class into a class called a visitor. It achieves this with a Visitor class which accepts various different objects, including the context, and performing the needed algorithms on the context.

```
public class XMLExportVisitor implements Visitor {

    public String export(Shape... args) {
        StringBuilder sb = new StringBuilder();
        sb.append("<?xml version=\"1.0\" encoding=\"utf-8\"?>" + "\n")

        for (Shape shape : args) {
            sb.append(shape.accept(this)).append("\n");
            //System.out.println(sb.toString());
            //sb.setLength(0);
        }
        return sb.toString();
    }

    public String visitDot(Dot d) {
        return "<dot>" + "\n" +
                "    <id>" + d.getId() + "</id>" + "\n" +
                "    <x>" + d.getX() + "</x>" + "\n" +
                "    <y>" + d.getY() + "</y>" + "\n" +
                "</dot>";
    }

    public String visitCircle(Circle c) {
        return "<circle>" + "\n" +
                "    <id>" + c.getId() + "</id>" + "\n" +
                "    <x>" + c.getX() + "</x>" + "\n" +
                "    <y>" + c.getY() + "</y>" + "\n" +
                "    <radius>" + c.getRadius() + "</radius>" + "\n" +
                "</circle>";
    }

    public String visitRectangle(Rectangle r) {
        return "<rectangle>" + "\n" +
                "    <id>" + r.getId() + "</id>" + "\n" +
                "    <x>" + r.getX() + "</x>" + "\n" +
                "    <y>" + r.getY() + "</y>" + "\n" +
                "    <width>" + r.getWidth() + "</width>" + "\n" +
                "    <height>" + r.getHeight() + "</height>" + "\n" +
                "</rectangle>";
    }

    public String visitCompoundGraphic(CompoundShape cg) {
        return "<compound_graphic>" + "\n" +
                "   <id>" + cg.getId() + "</id>" + "\n" +
                _visitCompoundGraphic(cg) +
                "</compound_graphic>";
    }
```

Figure 43: Visitor Class

A key point here is that instead of taking in a certain class and checking its type for the algorithm that needs to be performed, we use the concept of Double Dispatching which helps execute the proper method on the object without a bloated switch/if-else statement.

```
public class Rectangle implements Shape {
    private int id;
    private int x;
    private int y;
    private int width;
    private int height;

    public Rectangle(int id, int x, int y, int width, int hei
        this.id = id;
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }

    @Override
    public String accept(Visitor visitor) {
        return visitor.visitRectangle(this);
    }
```

Figure 44: Class takes in a visitor and executes relevant method

We can have multiple types of visitors which the classes can accepts and which apply different types of algorithms. This is very easy to do since all visitors share an interface that states the visiting methods that can take any context class as an parameter.

```
public interface Visitor {
    String visitDot(Dot dot);

    String visitCircle(Circle circle);

    String visitRectangle(Rectangle rectangle);

    String visitCompoundGraphic(CompoundShape cg);
}
```

Figure 45: Visitor Interface

15 Command Pattern

The command pattern allows a Receiver of some information to be decoupled with the class that is performing the action to produce that information; the Invoker. Whenever an action is taken by the invoker, it delegates the work to some class where the logic is executed via several implemented Commands. The key idea here is that when implementing a command pattern, you can make any class be the receiver of information.

```
// The application class sets up object relations. It acts as a
// sender: when something needs to be done, it creates a command
// object and executes it.
class Application is
    field clipboard: string
    field editors: array of Editors
    field activeEditor: Editor
    field history: CommandHistory

    // The code which assigns commands to UI objects may look
    // like this.
    method createUI() is
        // ...
        copy = function() { executeCommand(
            new CopyCommand(this, activeEditor)) }
        copyButton.setCommand(copy)
        shortcuts.onKeyPress("Ctrl+C", copy)

        cut = function() { executeCommand(
            new CutCommand(this, activeEditor)) }
        cutButton.setCommand(cut)
        shortcuts.onKeyPress("Ctrl+X", cut)

        paste = function() { executeCommand(
            new PasteCommand(this, activeEditor)) }
        pasteButton.setCommand(paste)
        shortcuts.onKeyPress("Ctrl+V", paste)

        undo = function() { executeCommand(
            new UndoCommand(this, activeEditor)) }
        undoButton.setCommand(undo)
        shortcuts.onKeyPress("Ctrl+Z", undo)

    // Execute a command and check whether it has to be added to
    // the history.
    method executeCommand(command) is
        if (command.execute())
            history.push(command)
```

Figure 46: Invoker

The Invoker will initiate requests for information by then triggering the command rather than interacting with the receiver directly. The command is not made by the Invoker but is rather passed in by the client via the Invokers constructor. The Invoker must be able to store a reference to these command object.

Commands that are called all implement the same interface and pass in the work to the Receiver object which will execute the relevant logic. A client can create more and more commands to suit their needs but they must pass all request parameters including the Receiver into the commands constructor.

```
// The concrete commands go here.
class CopyCommand extends Command is
    // The copy command isn't saved to the history since it
    // doesn't change the editor's state.
    method execute() is
        app.clipboard = editor.getSelection()
        return false

class CutCommand extends Command is
    // The cut command does change the editor's state, therefore
    // it must be saved to the history. And it'll be saved as
    // long as the method returns true.
    method execute() is
        saveBackup()
        app.clipboard = editor.getSelection()
        editor.deleteSelection()
        return true

class PasteCommand extends Command is
    method execute() is
        saveBackup()
        editor.replaceSelection(app.clipboard)
        return true

// The undo operation is also a command.
class UndoCommand extends Command is
    method execute() is
        app.undo()
        return false
```

Figure 47: Commands Implementation

The Receiver class now contains the logic to handle the command that it has received. The command would have done some work to specify how the information is passed to the Receiver, but most of the logic will be done by the Receiver. This can only be achieved because the parameters required to handle a request can be declared as fields in the command (and hence provided to the Receiver).

```
// The editor class has actual text editing operations. It plays
// the role of a receiver: all commands end up delegating
// execution to the editor's methods.
class Editor is
    field text: string

    method getSelection() is
        // Return selected text.

    method deleteSelection() is
        // Delete selected text.

    method replaceSelection(text) is
        // Insert the clipboard's contents at the current
        // position.
```

Figure 48: Receiver