Exploring and Using Creational, Structural and Behavioural Design Patterns

Luke Bruni | 12 December 2018

Contents

Overview

This report explores the nature of creational, structural and behavioral design patterns. To understand design patterns, we first need to understand patterns in general. Patterns are an equivalent to recipes in cooking or blueprints to a building; plans or instructions used for creating code. Design Patterns are like this but can be taken and customized to solve a particular problem that would be seen in the code[1].

Different Design Patterns

THE BRIDGE PATTERN

Structural

Structural patterns are a type of design patterns that puts emphasis on providing solutions and the efficient standards on class composition and object structures[2]. It eases the design of the program by using a method of identifying the relationship between entities. This type relies on the use of inheritance to help with the classes to work together to create a single working program. They also ensure that the overall system doesn't have to change massively just because one part of the system has[3].

Specific Design Patterns Under This Type

- Adapter

The Adapter pattern is one pattern that falls under the structural design. It allows for two unrelated interfaces to help work together by using an adapter to connect the two together and adapting them to work together. An example to understand this is how by scanning in a document allows for said document to be adapted to allow storage on a PC[4][2].

- Bridge

This design pattern is used to separate the abstract class from that of their implementations and uses a bridge between them. It extracts the implementations into a hierarchy of its own, allowing for individual development between the two and not interfering when modifying both elements.

- Composite

A composite pattern is mainly used for representing the group of objects in a hierarchy, describing them as a single type of object and composing them into a tree structure, where each branch has a function for it to perform[5].

Importance of the Bridge Pattern

The bridge pattern is a particularly important structural design pattern as it allows for there to be an independence between that of the idea and the implementation, affectively allowing for change without having a negative effect to the rest of the class. This pattern can allow for

an easier development environment to that of the programmer; cleaner code and reduction in size can help dramatically.

Review

The idea of this design pattern is to allow for a way to alter the abstracted class without messing with the implementation, allowing for change to either the abstraction or the implementation, but doesn't alter its counterpart.

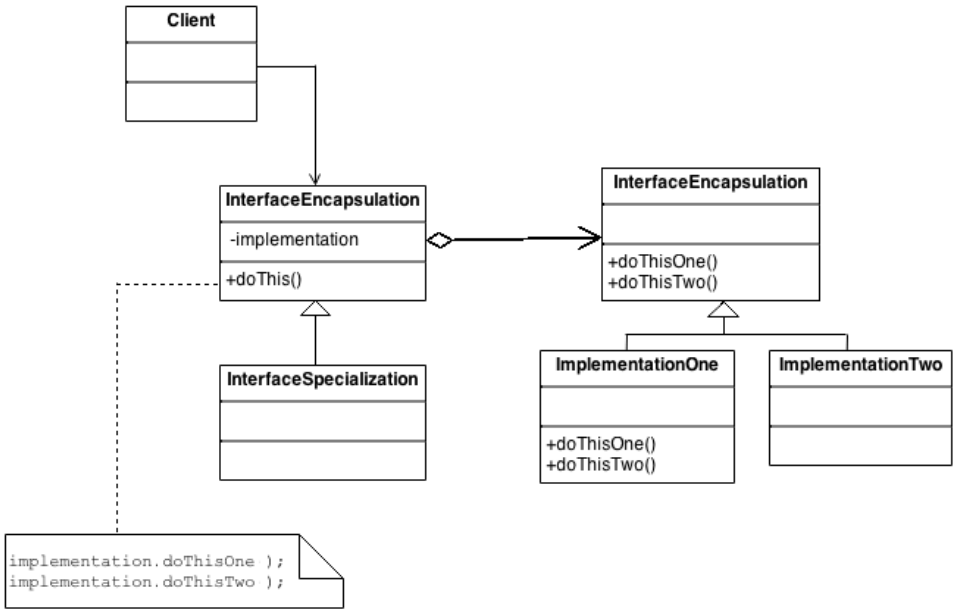
From what was gathered, it seems to be a programmer friendly pattern that allows for ease of use and cleaner code when put into action, as well as the lack of consequence for changing the source code too much. It's fairly good with independence, given the fact it is what the whole design is based around, where it can extend the independency of both the idea and implementation.

When to Use it

- To hide the implementation info from the clients.
- Changing the implementation at run-time.
- Sharing the implementation around multiple objects and reducing the number of subclasses.
- Mapping out structural class hierarchies.

Examples

UML



The UML diagram shows the bridge pattern in action, where the Interface Encapsulation class is divided into two variants; the left side, abstraction, the right-side implementation. The left side states the function that the abstraction will perform, and the right side takes

that abstraction and implements it into other classes that will use it. An example to help understand would be the left side is a switch; It's operation would be to turn on, close a circuit and the right side would be what that switch could be attached to, be it a Television, Dishwasher or light bulb[8].

Code

```
class Program
{
    static void Main(string[] args)
    {
        IAppliance tv = new TV("Bedroom TV");  //implementation object
        IAppliance vacuum = new VaccumCleaner
        ("My Vacuum Cleaner");  //implementation object

        Switch s1 = GetSwitch(tv);  //convert to abstraction
        Switch s2 = GetSwitch(vaccum);  //convert to abstraction

        //*** client code works only with the abstraction objects,
   //not the implementation objects ***
        s1.TurnOn();
        s2.TurnOn();
    }

    //convert implementation object to abstraction object
    static Switch GetSwitch(IAppliance a)
    {
        return new RemoteControl(a);
    }
}

//the abstraction
public abstract class Switch
{
    protected IAppliance appliance;
    public abstract void TurnOn();
}

//the implementation
public interface IAppliance
{
    void Run();
}

//concrete abstraction
public class RemoteControl : Switch
{
    public RemoteControl(IAppliance i)
    {
        this.appliance = i;
    }

    public override void TurnOn()
    {
        appliance.Run();
    }
}
```

```
//concrete implementation
public class TV : IAppliance
{
    private string name;

    public TV(string name)
    {
        this.name = name;
    }

    void IAppliance.Run()
    {
        Console.WriteLine(this.name + " is running");
    }
}

//concrete implementation
public class VaccumCleaner : IAppliance
{
    private string name;

    public VaccumCleaner(string name)
    {
        this.name = name;
    }

    void IAppliance.Run()
    {
        Console.WriteLine(this.name + " is running");
    }
}
```

The example above showcases the design in action, with the code detailing a switch class being used for two appliances; the television and the vacuum cleaner. The abstraction notes the appliances and functions that the class holds, and the implementation notes the switches control which appliance and receiving what appliance is currently turned on[7].

THE BUILDER PATTERN

Creational

Creational Design Patterns mainly emphasize with object creation, mainly by reducing the complexities and instability issues when creating objects, by doing so in a controlled manner[7]. This pattern combats the design problems and complexity when adding objects basically by controlling how the objects are added[8]. This type of design pattern divides the object creation, composition and representation into their own classes, away from the user program.

Specific Design Patterns Under This Type

- Singleton

The Singleton Pattern is the simplest pattern variant of the Creational Patterns. What is dies is utilizes a single class that creates an object and provides a way to access that object without any need for instantiation[9].

- Factory

The Factory Pattern mainly relies on the creation of objects utilizing main classes, but having the subclasses rely on the instantiation of a specific class[10]. The factory design is unique as it allows for more customization in a class, at the expense of making it a bit more complicated, and not reliant on class families but rather a new operation[11].

- Builder

The builder pattern is designed on creating complex objects step by step[11], mainly by taking all the subclasses representing parts of the main item and using them to create the final item. That final item would be made up of the implemented subclasses that would be used in the main class[12].

Importance of the Builder Pattern

The builder pattern is a mainly useful as a way to build up more complex objects by utilizing other subclasses, which in turn allows for easier control when constructing the objects needed. The separation between the objects that make up the main class and the representation are clearer cut, by seeing the function of the objects that make up both the subclasses and the main object allows for a better understanding of how the item works. This design pattern supports change, letting the user change how the functions work for each class, adding a good amount of flexibility[13].

Review

Using the builder pattern in turn is designed to allow, as said, allows for simple objects or classes to make up complex objects or classes and from a practical standpoint, it works as the user can create a main object without having to code in any individual part of the item all in one object, which would be risky enough as at that point, any change would cause the class to break, making it hard to clean up, and would cause complication. What the pattern

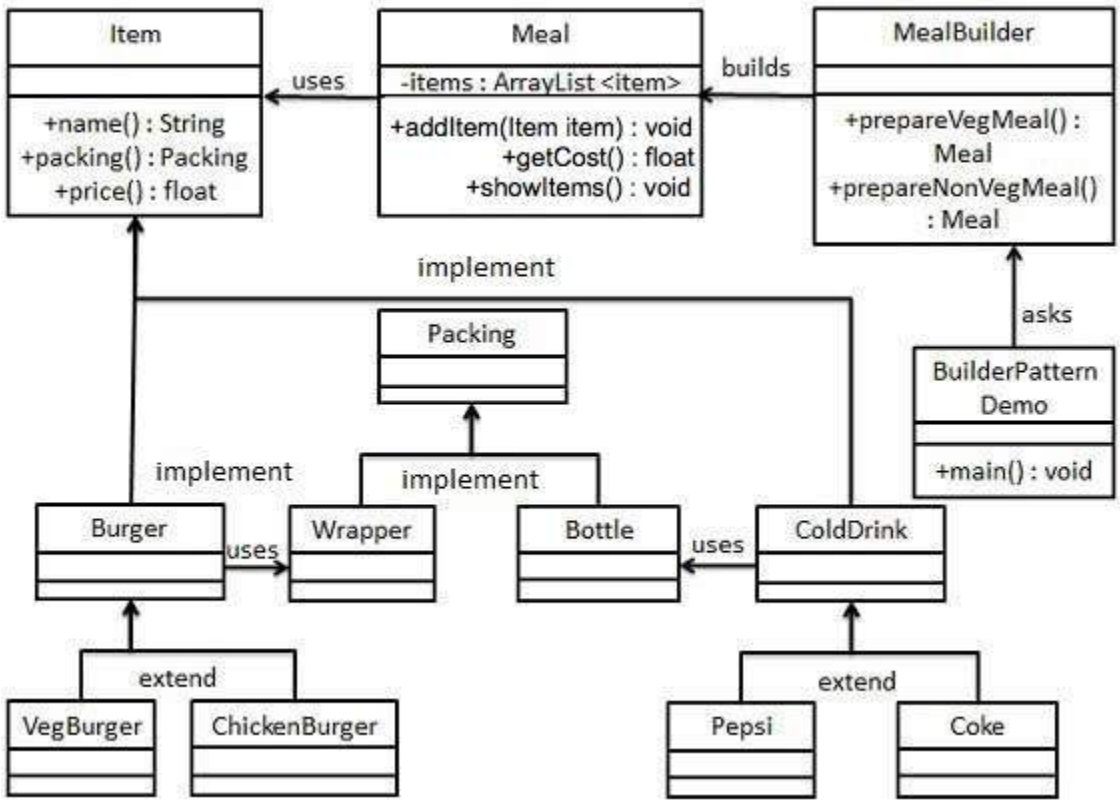
seeks to do is make the ability to create more in-depth objects easier, to which the pattern has done indeed; allowing for much easier and flexible objects, with the ability to change their function without causing any unwarranted errors.

When to Use it

- When you need to build more complex objects easily.
- When there needs to be control over each step in the creation process.
- If the objects require a step by step process rather then what would be represented[15].

Examples

UML



The UML diagram showcases how the builder design works. The individual objects are part of making up what the main object will be. The example here is for a fast food ordering system. Here, the meal from what the customer has ordered gets made up from what said meal includes. This is also true for the other objects that are made up; the 'Burger' class is made up of two burgers, the 'VegBurger' and the 'ChickenBurger', and the same thing happens to the 'ColdDrink' class. These classes go into the 'item' class as one whole main class.

Code

```
1.  interface HousePlan
2.  {
```

```
3.      public void setBasement(String basement);
4.
5.      public void setStructure(String structure);
6.
7.      public void setRoof(String roof);
8.
9.      public void setInterior(String interior);
10. }
11.
12. class House implements HousePlan
13. {
14.
15.     private String basement;
16.     private String structure;
17.     private String roof;
18.     private String interior;
19.
20.     public void setBasement(String basement)
21.     {
22.         this.basement = basement;
23.     }
24.
25.     public void setStructure(String structure)
26.     {
27.         this.structure = structure;
28.     }
29.
30.     public void setRoof(String roof)
31.     {
32.         this.roof = roof;
33.     }
34.
35.     public void setInterior(String interior)
36.     {
37.         this.interior = interior;
38.     }
39.
40. }
41.
42.
43. interface HouseBuilder
44. {
45.
46.     public void buildBasement();
47.
48.     public void buildStructure();
49.
50.     public void bulidRoof();
51.
52.     public void buildInterior();
53.
54.     public House getHouse();
55. }
56.
57. class IglooHouseBuilder implements HouseBuilder
58. {
59.     private House house;
60.
61.     public IglooHouseBuilder()
62.     {
```

```
63.         this.house = new House();
64.     }
65.
66.     public void buildBasement()
67.     {
68.         house.setBasement("Ice Bars");
69.     }
70.
71.     public void buildStructure()
72.     {
73.         house.setStructure("Ice Blocks");
74.     }
75.
76.     public void buildInterior()
77.     {
78.         house.setInterior("Ice Carvings");
79.     }
80.
81.     public void bulidRoof()
82.     {
83.         house.setRoof("Ice Dome");
84.     }
85.
86.     public House getHouse()
87.     {
88.         return this.house;
89.     }
90. }
91.
92. class TipiHouseBuilder implements HouseBuilder
93. {
94.     private House house;
95.
96.     public TipiHouseBuilder()
97.     {
98.         this.house = new House();
99.     }
100.
101.            public void buildBasement()
102.            {
103.                house.setBasement("Wooden Poles");
104.            }
105.
106.            public void buildStructure()
107.            {
108.                house.setStructure("Wood and Ice");
109.            }
110.
111.            public void buildInterior()
112.            {
113.                house.setInterior("Fire Wood");
114.            }
115.
116.            public void bulidRoof()
117.            {
118.                house.setRoof("Wood, caribou and seal skins");
119.            }
120.
121.            public House getHouse()
122.            {
```

```
123.                return this.house;
124.            }
125.
126.        }
127.
128.        class CivilEngineer
129.        {
130.
131.            private HouseBuilder houseBuilder;
132.
133.            public CivilEngineer(HouseBuilder houseBuilder)
134.            {
135.                this.houseBuilder = houseBuilder;
136.            }
137.
138.            public House getHouse()
139.            {
140.                return this.houseBuilder.getHouse();
141.            }
142.
143.            public void constructHouse()
144.            {
145.                this.houseBuilder.buildBasement();
146.                this.houseBuilder.buildStructure();
147.                this.houseBuilder.bulidRoof();
148.                this.houseBuilder.buildInterior();
149.            }
150.        }
151.
152.        class Builder
153.        {
154.            public static void main(String[] args)
155.            {
156.                HouseBuilder iglooBuilder = new IglooHouseBuilder();
157.                CivilEngineer engineer = new CivilEngineer(iglooBuilder);
158.
159.                engineer.constructHouse();
160.
161.                House house = engineer.getHouse();
162.
163.                System.out.println("Builder constructed: "+ house);
164.            }
165.        }
```

The code here showcases the creation of a house, with many different elements that go into the overall house being built. The ‘HouseBuilder’ class is made up of the actual construction that would go into every instance of the house and even setting about the engineers that will be said parts of the house that were previously defined, such as the basement, roof and interior[14].

STATE DESIGN PATTERN

Behavioural

Behavioral design patterns emphasize on how objects interact and their responsibility. These interaction patterns are then realized and can then increase the flexibility of said interaction in-between. Within these patterns, it should be noted that the interactions should be in a way that the objects can communicate easily with each other and should be loosely coupled with each other. This in turn allows the pattern to eliminate any potential hard coding and any object dependencies where unwanted. An example to help understand would be that a city be responsible with the streets and roads, combating the problems that may occur such as potholes in the roads[16][17].

Specific Design Patterns Under This Type

- Command

The command design pattern is more reliant that relies on giving out commands and executing those commands. This pattern utilizes and invoker that passes the command to any object that is capable of running that command. Once the object has been found, that command is then executed[18]. An example would be how a mailman, handling the collection of letters would determine when the letters would be delivered[19].

- Mediator

The idea of a mediator is reducing the chaotic dependencies that would arise between objects, in turn restricting the direct communications and only allowing them to communicate via a mediator object[21]. The mediator pattern is designed to allow any form of communication, despite how difficult the situation would be[20].

- State

The state design pattern is a behavioural pattern that is commonly put into use when an object changes its behaviour from its internal state[22]. Objects here are created representing various states and a context object that changes its behaviour based on the states on the objects[23].

Importance of the Builder Pattern

The state design pattern relies on the behaviour of the object with no need to rely on code statements, such as if statements, making the code cleaner and even to the point that the context class doesn't need to be aware of the implementation of the state logic. It improves the cohesion of the code as the behaviours from these states are aggregated into the concrete classes, place within a unique location within the code[22]. Overall, this pattern allows for loose coupled and systematic method of changing the states of the required object.

Review

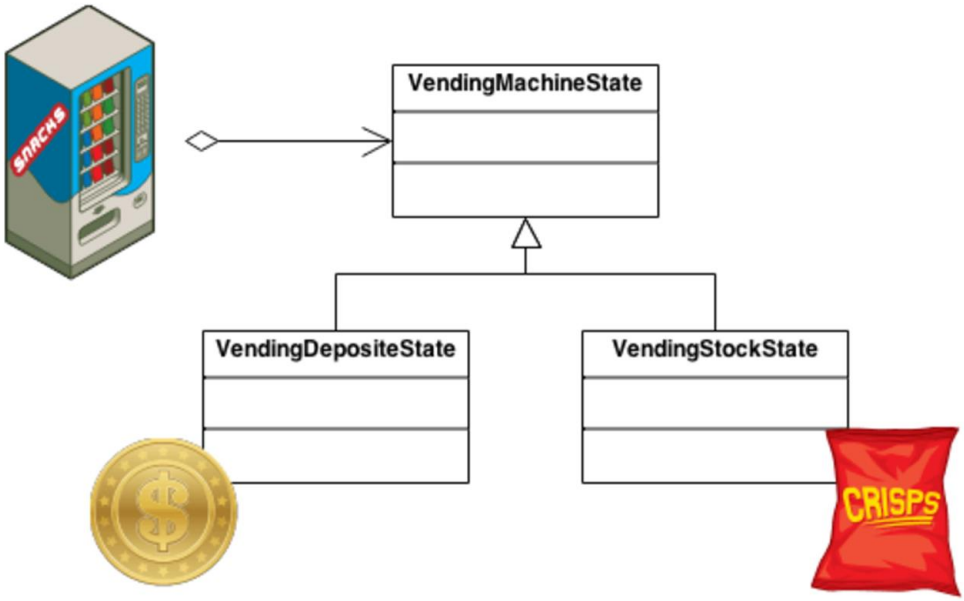
From the given research, the state design pattern is used primarily for changing the behaviour of objects, given as an organised substitute to changing the states via if/else statements and an organised method of changing the states of objects in general. It does make for an organised method of changing behaviour based on the state, allowing the user to extract the class' logic in order for the context class define the behaviour, in turn making it a fairly good substitute for the if/else statements. The only thing gathered is that the state will become hardcoded and considering it being bad practice, it would make this state pattern more questionable to use in certain projects.

When to Use it

- Within simple applications where coding practice is more relaxed but where the developer can have more of an advanced approach[25].
- When an object in the code has a complex amount of states, with different rules on the state transitions as well as what happens when the state changes.
- For use for objects within real-world concepts that have complex work flows[26].

Examples

UML



The example above shows how a vending machine and the stock that it has as well has the coins deposited. Though simplistic, the code has various states showing if the machine has any coins in it or if specific snacks are still in stock, going through to the vending machine which has its own states. If there no more crisps in the machine, that state would be

equivalent to 0 and overall return to the machine as being out of stock, changing what may be available.

Code

```java
// Java program to demonstrate working of
// State Design Pattern

interface MobileAlertState
{
    public void alert(AlertStateContext ctx);
}

class AlertStateContext
{
    private MobileAlertState currentState;

    public AlertStateContext()
    {
        currentState = new Vibration();
    }

    public void setState(MobileAlertState state)
    {
        currentState = state;
    }

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

class Vibration implements MobileAlertState
{
    @Override
    public void alert(AlertStateContext ctx)
    {
        System.out.println("vibration...");
    }

}

class Silent implements MobileAlertState
{
    @Override
    public void alert(AlertStateContext ctx)
    {
        System.out.println("silent...");
    }

}

class StatePattern
{
    public static void main(String[] args)
    {
        AlertStateContext stateContext = new AlertStateContext();
```

```
54.         stateContext.alert();
55.         stateContext.alert();
56.         stateContext.setState(new Silent());
57.         stateContext.alert();
58.         stateContext.alert();
59.         stateContext.alert();
60.     }
61. }
```

The code here showcases a mobile phone displaying different states based on the alert given in the context. It shows two different states, silent and vibration, and would call the context in case that state changes. The alert and silent states are called to show the states changing via the context class[24].

REFERENCES

1. Refactoring Guru [online]. (2014). Available from: <https://refactoring.guru/design-patterns>. [Accessed 30 November 2018].
2. David Landup (2018). Structural Design Patterns in Java. [online]. Stack Abuse. Available from: <https://stackabuse.com/structural-design-patterns-in-java/>. [Accessed 1 December 2018].
3. O'Reilly [online]. (n.d). Available from: <https://www.oreilly.com/library/view/learning-javascript-design/9781449334840/ch07s02.html>. [Accessed 1 December 2018].
4. Pankaj (2013). Java Design Patterns – Example Tutorial. [online]. JournalDev. Available from: <https://www.journaldev.com/1827/java-design-patterns-example-tutorial#structural-patterns>. [Accessed 1 December 2018].
5. GeeksForGeeks [online]. (2018). Available from: <https://www.geeksforgeeks.org/composite-design-pattern/>. [Accessed 1 December 2018].
6. baeldung (2018). Introduction to Creational Design Patterns. [online]. Baeldung. Available from: <https://www.baeldung.com/creational-design-patterns>. [Accessed 2 December 2018].
7. DevLake (2011). Bridge Design Pattern. [online]. Code Project. Available from: <https://www.codeproject.com/articles/183655/bridge-design-pattern>. [Accessed 10 December 2018].
8. Source Making [online]. (2018). Available from: <https://sourcemaking.com/design_patterns/creational_patterns>. [Accessed 2 December 2018].
9. Tutorials Point [online]. (2018). Available from: <https://www.tutorialspoint.com/design_pattern/singleton_pattern.htm>. [Accessed 2 December 2018].
10. dofactory [online]. (2014). Available from: <https://www.dofactory.com/net/factory-method-design-pattern>. [Accessed 11 December 2018].
11. Source Making [online]. (2007). Available from: <https://sourcemaking.com/design_patterns/factory_method>. [Accessed 11 December 2018].

12. TutorialsPoint [online]. (2013). Available from: <https://www.tutorialspoint.com/design_pattern/builder_pattern.htm>. [Accessed 12 December 2018].
13. JavaTPoint [online]. (2013). Available from: <https://www.javatpoint.com/builder-design-pattern>. [Accessed 12 December 2018].
14. GeeksForGeeks [online]. (2017). Available from: <https://www.geeksforgeeks.org/builder-design-pattern/>. [Accessed 12 December 2018].
15. Joydip Kanjilal (2015). How to implement the Builder design pattern. [online]. InfoWorld. Available from: <https://www.infoworld.com/article/3005198/application-development/how-to-implement-the-builder-design-pattern.html>. [Accessed 12 December 2018].
16. Emmanouil Gkatziouras (2018). Behavioural Design Patterns: Chain of Responsibility. [online]. Java Code Geeks. Available from: <https://www.javacodegeeks.com/2018/10/behavioural-patterns-chain-responsibility.html>. [Accessed 12 December 2018].
17. JavaTPoint [online]. (2013). Available from: <https://www.javatpoint.com/behavioral-design-patterns>. [Accessed 12 December 2018].
18. TutorialsPoint [online]. (2013). Available from: <https://www.tutorialspoint.com/design_pattern/command_pattern.htm>. [Accessed 12 December 2018].
19. Andrew Powell-Morse (2017). Behavioral Design Patterns: Command. [online]. Airbrake. Available from: <https://airbrake.io/blog/design-patterns/behavioral-command-design-pattern>. [Accessed 12 December 2018].
20. Andrew Powell-Morse (2017). Behavioral Design Patterns: Mediator. [online]. Airbrake. Available from: <https://airbrake.io/blog/design-patterns/mediator-design-pattern>. [Accessed 12 December 2018].
21. Refactoring Guru [online]. (n.d). Available from: <https://refactoring.guru/design-patterns/mediator>. [Accessed 12 December 2018].
22. GeeksForGeeks [online]. (2017). Available from: <https://www.geeksforgeeks.org/state-design-pattern/>. [Accessed 12 December 2018].
23. TutorialsPoint [online]. (2013). Available from: <https://www.tutorialspoint.com/design_pattern/state_pattern.htm>. [Accessed 12 December 2018].
24. Source Making [online]. (2007). Available from: <https://sourcemaking.com/design_patterns/state>. [Accessed 12 December 2018].
25. Denis Szczukocki (2018). State Design Pattern in Java. [online]. Baeldung. Available from: <https://www.baeldung.com/java-state-design-pattern>. [Accessed 12 December 2018].
26. DevIQ [online]. (n.d). Available from: <https://deviq.com/state-design-pattern/>. [Accessed 12 December 2018].