Chapter 1: Introduction

1 Software Design

Design

In software engineering, design refers to the phase that comes before the actual coding. It involves steps like creating algorithms, breaking down a complex system into manageable parts (modules), and brainstorming the general ideas and logic for each part, including how they will communicate with each other. A well-crafted design is crucial, as it not only makes the implementation and testing phases smoother but also ensures that maintenance is more manageable. Proper maintenance is key to making the software last for a long time, so it's important to focus on sustainable design practices.

2 Maintenance

Maintenance

Maintenance is the final phase in the lifecycle of any software. It involves fixing bugs, adding, removing, or modifying features. A good design plays a crucial role in making maintenance more manageable and efficient.

Note

A good designer knows how to make the right choices and weigh the pros and cons. In real-life applications, designers face very complex systems, and it's not always possible to have all the advantages.

3 Modularity (Decomposition)

Modularity

Modularity divides a complex system into smaller parts called modules:

1. **Architectural Design**: Defines the modules and how they will communicate with each other.
2. **Unit Design**: Defines each module's elements independently of the others.

4 Evaluation Criteria

Criteria for a Good Decomposition

Many decompositions (modularities) might be proposed for the same system, and the criteria for a good decomposition are:

- High Cohesion
- Low Coupling

4.1 Cohesion

4.1.1 Definition

Definition

Cohesion refers to the degree of structuring and affinity between the elements of a module. It essentially describes how tightly related the goals and tasks of these elements are. There are several types of cohesion, which we will list from weakest to strongest. A strong, high cohesion is desirable.

4.1.2 Why High Cohesion

High Cohesion

The stronger the cohesion, the more tightly related the elements of the module are, making the module adhere to the Single Responsibility Principle. This results in easier maintenance because when updating a specific feature, we can easily pinpoint which module is related to it. Since each module has a single responsibility, changes are more localized and manageable.

4.1.3 Types Of Cohesion

Random Cohesion (Coincidental Cohesion)

Random

The elements of a module are grouped together randomly with no meaningful relationship. This is the weakest form of cohesion and should always be avoided.

Logical Cohesion

Logical

Groups elements of a module that perform logically similar tasks.

Temporal Cohesion

Temporal

Groups elements of a module that execute within the same time phase.

Procedural Cohesion

Procedural

Groups elements of a module that execute in a well-defined order (collaboration of procedures).

Communicational Cohesion

Communicational

Groups elements of a module that operate on the same set of data.

Sequential Cohesion

Sequential

Groups elements of a module so that the output of one element serves as the input for another.

Functional Cohesion

Functional

All elements contribute to a single, well-defined task or are indispensable to its execution.

Note

- Random cohesion must be avoided. If it appears in a test or exam, it results in an automatic 0. We aim for functional cohesion, as the systems studied in exercises are simple.
- In real-life applications, achieving pure functional cohesion is not always possible, and hybrid cohesion may be necessary.

4.2 Coupling

4.2.1 Definition

Definition

Coupling refers to the degree of dependency (Interconnection) between modules. We aim for low, weak coupling to promote modularity and maintainability,it has many types we will list them from strongest to weakest.

Note

No coupling is impossible, as any system requires minimum communication between modules.

4.2.2 Why Low Coupling

Low Coupling

We want low coupling so that modules are loosely coupled to:

- **Failure Prevention:** If several modules depend on a central module, and it fails, the whole system crashes.
- **Updating Modules:** Having low coupling means that when updating a module, it will impact fewer modules, making the process easier.

4.3 Types Of Coupling

4.3.1 Content Coupling

Content

One module directly accesses or modifies the internal content of another module. This is the strongest type of coupling and should be avoided.

4.3.2 Common Coupling

Common

Modules that share global data.

4.3.3 External Coupling

External

Modules that communicate with each other through external entities.

4.3.4 Control Coupling

Control

One module controls the behavior of another by passing control parameters.

4.3.5 Stamp Coupling

Stamp

Modules that share complete complexe data structures (non-primitive : collection, array , struct , object) , but not all the elements are used by each module..

4.3.6 Data Coupling

Data

Modules share only essential simple data (primitive data) through parameters or arguments.

4.4 Relation Between Cohesion & Coupling

Relation

- **High cohesion leads to low coupling:** A highly cohesive module depends less on other modules and more on its internal elements, resulting in low coupling.
- **Low coupling leads to high cohesion:** Low coupling means that modules are not highly dependent on each other, so the elements within each module need to be sufficient to perform a task, leading to high cohesion.

$$\boxed{\text{High Cohesion} \iff \text{Low Coupling}}$$

5 Scalability

Definition

As the number of users, data size, and server requests grow, our software must continue running smoothly and adapt to exponential growth without slowing down or crashing. To achieve this, we need to ensure that our software is scalable.

5.1 Principles of Scalability

Principles

- **Scalable Database**: Ensures efficient data storage and retrieval as demand increases.
- **Distributed Architecture**: Spreads the workload across multiple servers to prevent bottlenecks.
- **Asynchronous Processes**: Improves efficiency by allowing tasks to run in the background without blocking other operations, enabling the system to handle more requests simultaneously.
- **Microservices**: Breaks applications into smaller, independent services deployed in different servers for better scalability.
- **Load Balancing**: Distributes traffic evenly to prevent server overload.
- **Auto-scaling**: Adjusts resources dynamically based on demand.
- **Caching**: Stores frequently accessed data to reduce load and improve speed.

5.2 Types of Scalability

5.2.1 Horizontal (Flat) Scalability

Horizontal

Scaling by adding more servers to distribute the load. For example, deploying additional servers in different regions to improve performance.

5.2.2 Vertical Scalability

Vertical

Scaling by upgrading the same server with better hardware. For example, increasing RAM to enhance caching performance.

6 User Experience (UX)

UX

The interface of an application serves as the bridge between humans and machines. It has evolved into what we now know as **user experience (UX)**, incorporating elements such as sensors, voice control, and touchscreens.

In the design process, it is crucial to ensure that the software experience is intuitive, efficient, and pleasant for the user. A smooth experience not only enhances user satisfaction but also improves software adoption.

For example, if two similar software applications are available, users will naturally prefer the one with a more intuitive and user-friendly interface, even if the other offers more advanced features or better overall functionality.

6.1 Key Principles of Good UX

Principles

- **Simplicity & Clarity**: A simple and clear interface helps users focus on the core functionality of the software without being distracted by unnecessary details or an overly cluttered design.
- **Consistency**: The interface elements should remain uniform and cohrent throughout the entire software to create a seamless experience.
- **Understanding the End User**: It is essential to understand user behavior, expectations, and needs to design a more intuitive experience.
- **Mobile-First Approach**: Designing for mobile devices first ensures a smooth experience across all platforms, as it is easier to scale up for desktops than to adapt a complex desktop design for mobile.
- **Ease of Learning**: Users prefer software that is intuitive and easy to use without requiring extensive training or special instruction.
- **Customization**: Allowing users to configure settings and personalize their experience improves satisfaction and engagement.
- **Accessibility**: Software should be inclusive, ensuring usability for individuals with disabilities. For example, adding image descriptions allows screen readers to assist visually impaired users.
- **Responsiveness & Feedback**: Users should feel that the software responds immediately to their actions. Providing visual or auditory feedback, such as loading indicators or confirmation messages, keeps users informed and reduces confusion.
- **User Testing**: Involving real users in testing and gathering feedback is crucial for improving the interface and ensuring a user-friendly experience.

Chapter 2: Design Pattern

1 Design Pattern

Definition

A design pattern describes **proven** and **abstract** solutions to **recurring** problems in software design.

Why Proven

Proven solutions have been tested on real projects, ensuring reliability and effectiveness.

Why Abstract

Abstract solutions can be adapted and customized to meet specific needs.

Why Recurring

We want model for recurring problems to address patterns in design problems that repeat across different contexts.

2 Categories

Categories

The first influential book on design patterns, **Design Patterns: Elements of Reusable Object-Oriented Software** by the "Gang of Four" (Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides), was published in its second edition in 1995. It presents a collection of 23 design patterns, classified into three categories.

2.1 Creational Design Patterns (5 from the Gang of Four Collection)

Creational

Describes how to fix problems of class instantiations.

2.2 Structural Design Patterns (7 from the Gang of Four Collection)

Structural Design Pattern

Describes how to structure classes with a minimum of dependencies.

2.3 Behavioral Design Patterns (11 from the Gang of Four Collection)

Behavioral Design Pattern

Describe how to ensure a specific behavior of the application.

Note

The design patterns from the "Gang of Four" are intended for object-oriented design.

3 Pros & Cons

Pros & Cons

Pros

- Solves recurring problems with proven and reliable solutions.
- Improves quality and speed.
- Design patterns provide a common language for communication between designers.
- Design patterns are independent of implementation languages and sufficiently generic (abstract) to be applied in various situations.

Cons

- Design patterns need to be mastered and thoroughly studied.
- Design patterns always require adaptation when applied.
- Design patterns can increase the complexity of simple solutions.

Formalism Of A Design Pattern

The formalism of a design pattern (description of a design pattern) consists of:

- Name of the design pattern.
- Addressed problem.
- Proposed solution.
- Other sections (pros & cons, usage examples and context before and after usage, usage recommendations, etc.).

Principles Favored by Design Patterns

Design patterns of the "Gang of Four" promote good design (low coupling, high cohesion, etc.) and two principles:

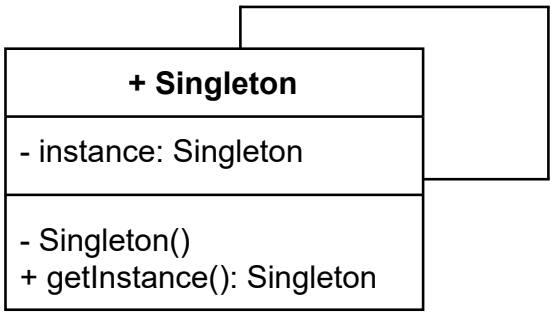
- **Principle**$_1$: Favor composition over inheritance (favor $\neq$ replace)
- **Principle**$_2$: Clients interact with the abstractions rather than the implementations

4 Some Design Pattern

4.1 Singleton Creational Design Pattern

Singleton

The Singleton is a creational design pattern that limits the number of instantiations to a single object and makes it globally accessible.



Explication

We first initialize a static private attribut called instance to null , when ever we want to get instance of singleton we call the public static method getInstance that checks if instance is null if yes it will call the private constructor and in either case it returns the instance

Java Code:

```java
public class Singleton {
  private static Singleton instance = null;

  private Singleton() {
    System.out.println("Instance Of Singleton");
  }

  public static Singleton getInstance() {

    if(instance == null) {
      instance = new Singleton();
    }

    return instance;
  }

}
```

```java
public class Main {

  public static void main(String[] args) {

    Singleton.getInstance();
    Singleton.getInstance();
  }

}
```

Output:

```
Instance Of Singleton
```

Thread Safe

This implementation isn't thread-safe. If two or more threads run `getInstance()`, they may call the constructor at the same time, creating multiple objects and violating the Singleton pattern.

Java Code:

```
public class Main {

  public static void main(String[] args) {

     Runnable task = () -> {
               System.out.println("Running on thread: " + Thread.currentThread().getName());
               Singleton s =  Singleton.getInstance();
          };

          Thread thread1 = new Thread(task);
          Thread thread2 = new Thread(task);
          Thread thread3 = new Thread(task);

          thread1.start();
          thread2.start();
          thread3.start();
  }

}
```

Output:

```
Running on thread: Thread-1
Running on thread: Thread-0
Running on thread: Thread-2
Instance Of Singleton
Instance Of Singleton
```

How To Make It Thread-Safe

As we can see, in a multi-threaded scenario, the current implementation fails to maintain a single instance. To fix that, we will use `volatile` and `synchronized` keywords:

- `synchronized`: Ensures that the function, class, instance, or block of code is executed by only one thread at a time (mutual exclusion).
- `volatile`: Ensures the variable is stored in the main memory (RAM) rather than the cache memory. It is used to prevent two issues:
  - **Memory cache inconsistency**: Sometimes, a thread may change a variable in its local memory cache, but OS protocols may not propagate the update to the other thread's cache in time, causing other threads to read an outdated value.
  - **Partially initialized object**: A thread might start creating the instance, but another thread checks if the instance is not null, which is true, and returns an incomplete object because the first thread hasn't finished its creation, potentially crashing the application.

Java Code:

```java
public class Singleton {

  private static volatile Singleton instance = null;

  private Singleton() {
    System.out.println("Instance Of Singleton");
  }

  public static Singleton getInstance() {

    synchronized (Singleton.class) {

      if(instance == null) {
        instance = new Singleton();
      }
  }

    return instance;
  }

}
```

Output:

```
Running on thread: Thread-2
Running on thread: Thread-1
Instance Of Singleton
Running on thread: Thread-0
```

Optimization

The implementation is correct and thread-safe, but we can optimize it:

- Add a check if `instance` is null before entering the `synchronized` block to avoid unnecessary overhead and only enter the mutex when `instance` is null.
- Accessing main memory is slow, so we can use a local variable inside the `getInstance()` function to reduce redundant memory reads.

Check If Null Before Synchronized Block:

```
public class Singleton {
  private static volatile Singleton instance = null;


  private Singleton() {
    System.out.println("Instance Of Singleton");
  }

  public static Singleton getInstance() {

    if(instance == null) {

    synchronized (Singleton.class) {

          if(instance == null) {
            instance = new Singleton();
          }
  }

    }

    return instance;
  }

}
```

Local Variable:

```
public class Singleton {
  private static volatile Singleton instance = null;


  private Singleton() {
    System.out.println("Instance Of Singleton");
  }

  public static Singleton getInstance() {
    Singleton result = instance;

    if(result == null) {

    synchronized (Singleton.class) {
      result  = instance;
      if(result == null) {
        instance = result = new Singleton();
      }
  }

    }

    return result;
  }

}
```

Breaking With Reflection

Even though the current implementation is thread-safe it is weak against reflection as we can fetch constrcutor of the `Singelton` and set them as `public` at run time.

Reflection

```
import java.lang.reflect.Constructor;

public class ReflectionSingletonTest {

    public static void main(String[] args) {
        Singleton instanceOne = Singleton.getInstance();
        Singleton instanceTwo = null;
        try {
            Constructor[] constructors = Singleton.class.getDeclaredConstructors(); //fetch
                constructors of Singleton Class
            for (Constructor constructor : constructors) {     // loop through array of constructor
                constructor.setAccessible(true);               // make constructor public
                instanceTwo = (Singleton) constructor.newInstance(); // create instance
                break;
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println(instanceOne.hashCode());  //print reference of 1st instance
        System.out.println(instanceTwo.hashCode());  //print reference of 2nd instance
    }

}
```

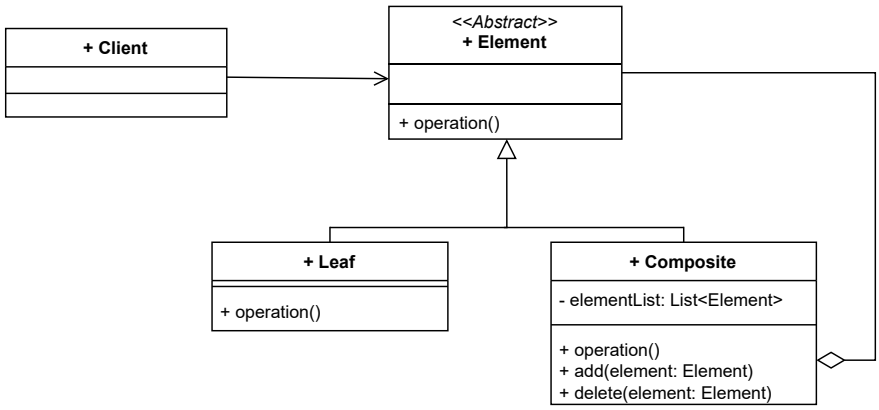
Output

```
Instance Of Singleton
Instance Of Singleton
804564176
1421795058
```

4.2 Composite Structural Design Pattern

Composite

The Composite pattern is a structural design pattern that imposes a hierarchical tree structure, consisting of simple elements and composite elements, and Allows manipulating different objects in a uniform way



Explication

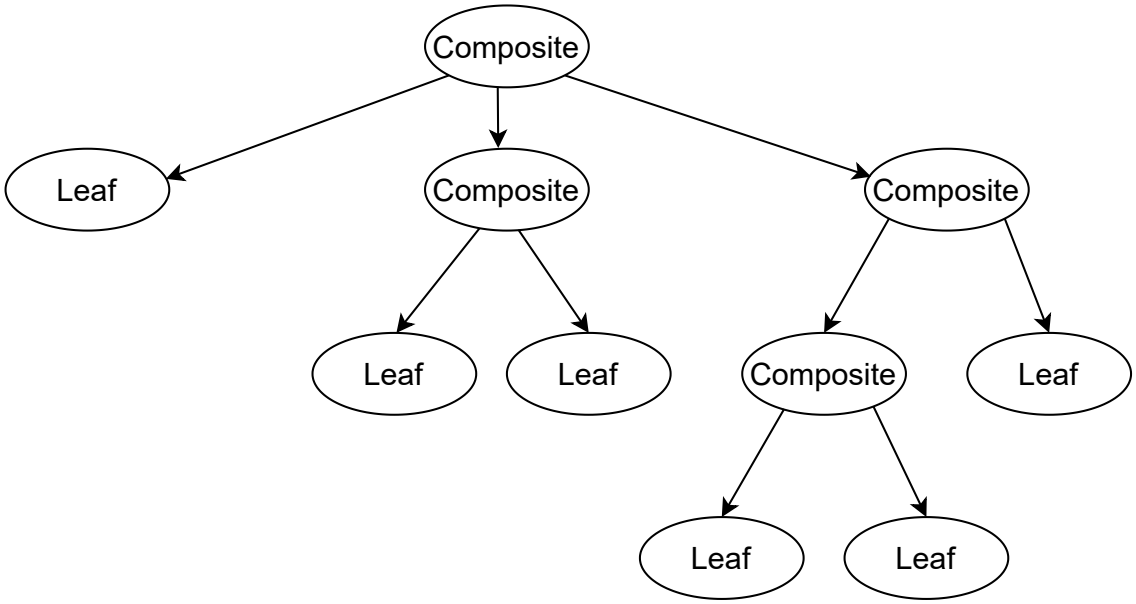
The Element class serves as an abstract parent class with an unimplemented method, operation(). The Leaf class inherits from Element and overrides the operation() method to provide its specific implementation. Similarly, the Composite class also inherits from Element and implements operation(). However, it has an attribute that is a collection of Element objects. This design allows the collection to hold both simple elements (Leaf) and complex elements (Composite), organizing them into a tree structure.

This abstraction provides a unified interface and defines a common behavior (`operation()`) for all elements, allowing the client class to manipulate them uniformly.

Note

Element can be an abstract class or an interface both implementations are valid.

Graphical Representation Of The Composite



Java Code:

```java
public abstract class Element {

  public Element(){

  }

  public abstract void operation();
}
```

```java
public class Leaf extends Element {

  @Override
  public void operation() {
    System.out.println("Leaf Operation");
  }

}
```

```java
import java.util.LinkedList;
import java.util.List;

public class Composite extends Element {
private List<Element> elementList = new LinkedList<>();

 @Override
 public void operation() {
  System.out.println("Composite Operation");
 }

 public void add(Element element) {
   this.elementList.add(element);
 }

 public void delete(Element element) {
   this.elementList.remove(element);
 }

 public List<Element> getElementList(){
   return this.elementList;
 }

 public void printTree(int deepness,List<Element> elementList) {
   for(Element element : elementList) {
     if(element instanceof Leaf) {
      System.out.println(" ".repeat(deepness)+"Leaf");
     }

     else if(element instanceof Composite) {
        ++deepness;
        System.out.println(" ".repeat(deepness)+"Composite");
        Composite com = (Composite)element;
        printTree(deepness+1,com.elementList);
     }
   }
 }
}
```

```
public class Main {

  public static void main(String[] args) {
    Composite com = new Composite();

    Leaf l1 = new Leaf();
    Leaf l2 = new Leaf();
    Leaf l3 = new Leaf();
    Leaf l4 = new Leaf();
    Leaf l5 = new Leaf();

    com.add(l1);
    com.add(l2);

    Composite comInner1 = new Composite();
    comInner1.add(l3);

    Composite comInner2 = new Composite();
    comInner2.add(l4);
    comInner2.add(l5);

    comInner1.add(comInner2);

    com.add(comInner1);

    for (Element element : com.getElementList()) {
      element.operation();
    }

    System.out.println("\n");

    com.printTree(0,com.getElementList());

  }

}
```

Output:

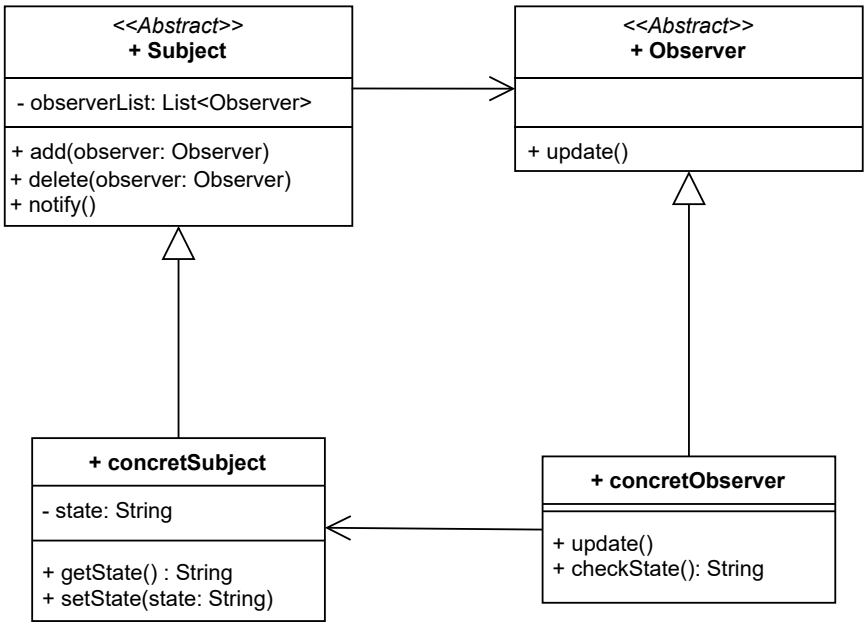
```
Leaf Operation
Leaf Operation
Composite Operation


Leaf
Leaf

 Composite
  Leaf
   Composite
    Leaf
    Leaf
```

4.3 Observer Behavioral Design Pattern

Observer

The Observer pattern consists of two parties: the **Subject** and the **Observers**:

- **Subject**: Represents data that is prone to change state at runtime. It notifies the **Observers** of any changes.
- **Observers**: Are external class interested in the **Subject** and trigger actions based on its state.



Explication

The **Observer** is an abstract class that defines an unimplemented method, `update()`, which is executed each time the state changes.

The **Subject** is also an abstract class, which maintains a collection of observers and declares unimplemented methods:

- `add(observer)` : Add new observer to the list.
- `delete(observer)` : Remove an existing from the list.
- `notify()`: Loops through all the observer of the list and calls their `update()` method .

A **ConcreteObserver** inherits from **Observer** , holds reference of **concreteSubject** implements the `update()` method, along with a new method `checkState()` that calls the `getState()` invoked within `update()`.

A **ConcreteSubject** inherits from **Subject**, overrides all unimplemented methods, and includes a state attribute with its getter and setter. Whenever the setter is called, the `notify()` method is invoked within its body to ensure that all observers are updated.

Note

Subject and observer can be interfaces too.

Java Code:

```java
public abstract class Observer {


   private String name;

   public Observer(String name) {
     this.name = name;
   }

   public abstract void update();

   public void setName(String name) {
     this.name = name;
   }

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

}
```

```java
import java.util.LinkedList;
import java.util.List;

public abstract class Subject {

  private List<Observer> observerList = new LinkedList<>();

  public abstract void add(Observer observer);
  public abstract void delete(Observer observer);
  public abstract void notifyObservers();

  public List<Observer> getObserverList() {
    return this.observerList;
  }

}
```

```java
public class concretObserver extends Observer{

  private concretSubject sub;

  public concretObserver(String name,concretSubject sub) {
    super(name);
    this.sub = sub;
  }

  @Override
  public void update() {
    String state= checkState();
    System.out.println("Observer "+this.getName()+" New State : "+state);

    }

    public String checkState() {
      return sub.getState();
    }

}
```

```java
public class concretSubject extends Subject {

  private String state;

  @Override
  public void add(Observer observer) {
    this.getObserverList().add(observer);

  }

  @Override
  public void delete(Observer observer) {
    this.getObserverList().remove(observer);
  }

  @Override
  public void notifyObservers() {
    for(Observer obs : getObserverList()) {
      obs.update();
    }
  }

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

  public void setState(String state) {
        this.state = state;
        System.out.println("Setting state to "+state);
        notifyObservers();
    }

}
```

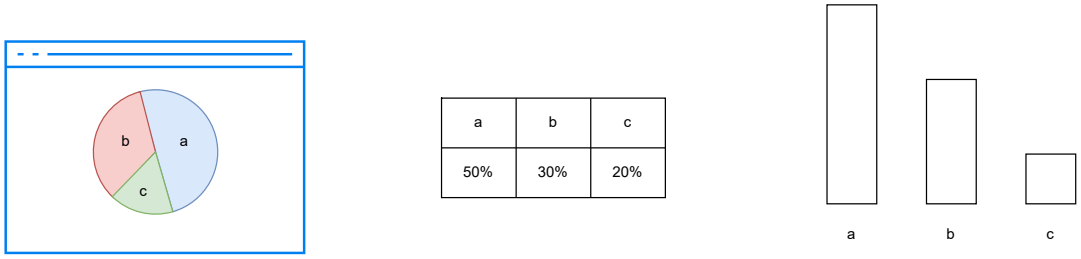
```java
public class Main {
    public static void main(String[] args) {

        concretSubject subject = new concretSubject();

        Observer observer1 = new concretObserver("Alice",subject);
        Observer observer2 = new concretObserver("Bob",subject);

        subject.add(observer1);
        subject.add(observer2);

        subject.setState("State 1");

        subject.delete(observer1);

        subject.setState("State 2");

        subject.setState("State");
    }
}
```

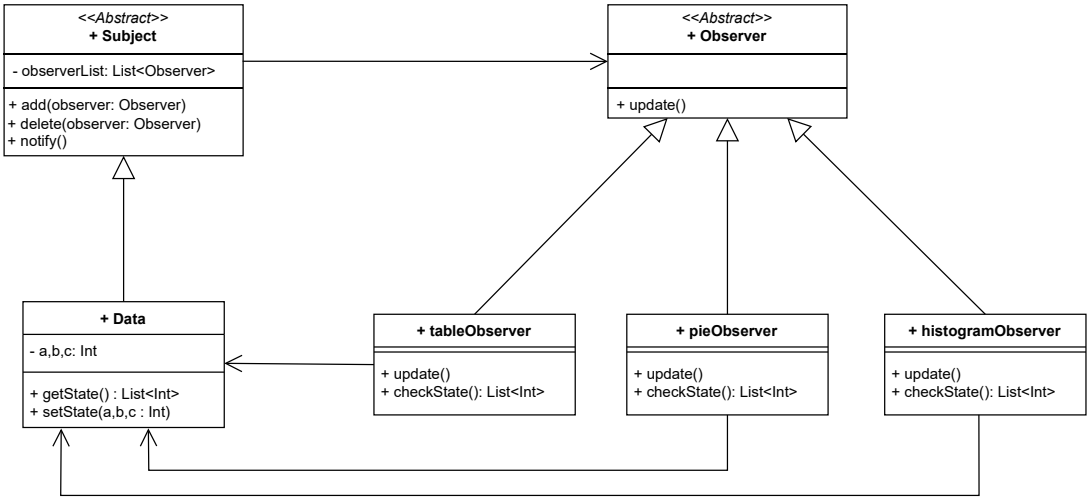
Output:

```
Setting state to State 1
Observer Alice New State : State 1
Observer Bob New State : State 1
Setting state to State 2
Observer Bob New State : State 2
Setting state to State
Observer Bob New State : State
```

Example :

An application presents the data points (a, b, c) in three formats: table, a histogram, and a pie chart as shown below:





Explication

- The data elements a, b and c are subject to change and thus constitute the **subject**.
- The views `table`, `pie chart` and `histogram` are external classes that are interested to the subject's state and must update whenever it changes, making them **observers**.

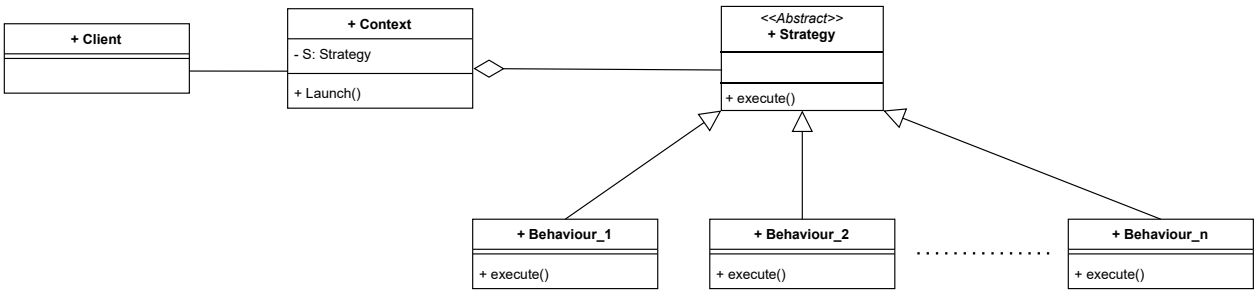
4.4 Strategy Behavioral Design Pattern

Strategy

The Strategy pattern consists of two parties: the **Context** and the **Strategies**:

- **Context**: Represents the class that users will interact with to choose the needed behavior at runtime
- **Strategies**: Are all the interchangeable behavior classes a user might choose

To say Strategy is implemented correctly, the user must be able to select any behavior possible. The Strategy pattern allows us to separate the object from its behaviors and to make them interchangeable and selectable at runtime.



Explication

The **Strategy** is an abstract class that defines an unimplemented method, `execute()`, which is executed when the user chooses a behaviour at runtime. The **Behaviour**$_i$ are concrete classes that extend the **Strategy** class and implements the `execute()` method. They represent the different interchangeable behaviours a user might choose. A **Context** is a class that the user will interact with to choose the needed behaviour. It holds a reference to a **Strategy** object and has a method `Launch()` that will call the strategy's `execute()` method according to the user's choice. The context can switch between different strategies at runtime by changing which concrete behaviour implementation it references.

Note

Strategy can be an interface.

Java Code:

```java
public abstract class Strategy {

  public abstract void execute();
}
```

```java
public class Behaviour_1 extends Strategy{

  @Override
  public void execute() {
    System.out.println("Behaviour_1 execute()");

  }

}
```

```java
public class Behaviour_2 extends Strategy{

  @Override
  public void execute() {
    System.out.println("Behaviour_2 execute()");

  }

}
```

```java
public class Behaviour_n extends Strategy{

  @Override
  public void execute() {
    System.out.println("Behaviour_n execute()");

  }

}
```

```java
public class Context {
 private Strategy s;

 public void Launch(int Case) {
   switch(Case) {
    case 1:
      s = new Behaviour_1();
        break;
    case 2:
      s = new Behaviour_2();
      break;
    case 3:
      s = new Behaviour_n();
      break;

   }
   s.execute();
 }
}
```

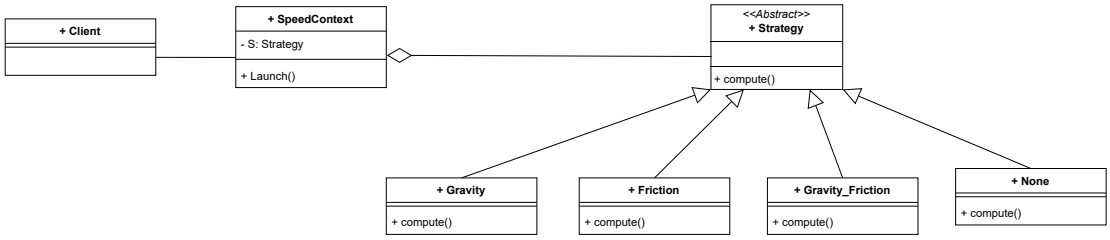
```java
import java.util.Scanner;

import javax.naming.Context;

public class Main {

  public static void main(String[] args) {
    Context con = new Context();
    Scanner sc = new Scanner(System.in);

    int choice = -1;

    while(choice!=0) {
    System.out.println("Press 1 For Strategy 1\nPress 2 For Strategy 2\nPress 3 For Strategy
        n\nPress 0 to Quit");
    choice = sc.nextInt();
    if(choice!=0) {
       con.Launch(choice);
     }
    }

  }

}
```

Output:

```
Press 1 For Strategy 1
Press 2 For Strategy 2
Press 3 For Strategy n
Press 0 to Quit
1
Behaviour_1 execute()
Press 1 For Strategy 1
Press 2 For Strategy 2
Press 3 For Strategy n
Press 0 to Quit
2
Behaviour_2 execute()
Press 1 For Strategy 1
Press 2 For Strategy 2
Press 3 For Strategy n
Press 0 to Quit
3
Behaviour_n execute()
Press 1 For Strategy 1
Press 2 For Strategy 2
Press 3 For Strategy n
Press 0 to Quit
0
```

Example :

An application that computes the vertical speed of an element and taking account of gravity and friction.



Explication

- When computing vertical speed there are 4 cases : with gravity only , with firction only , both and none , all of these represents different strategies of our application , therefor each case will be a class extending **Strategy** class.
- The **speedContext** is the class user will interact with to choose the needed strategy at run time .