Ministry of Education, Culture and Research of Moldova

Technical University of Moldova

Software Engineering and Automatics Department

REPORT

Course work
at TMPS
„Organiser App”

Performed by:
st. gr. FAF-161 V.Bantuș
V.Metei
G.Zaharia

Verified by:
conf. univ. A.Poștaru
asist. univ. M.Gavriliță

Chişinău, 2019

Contents

1. Introduction and Theory

1.1 Project Description

The Organiser App is a mobile application. Its main goal is to help people manage, prioritize, and complete the most important things they need to achieve every day.

All the people equipped with smart-phones or tablets will be able to register in our application. Also there will be an option of signing in with other social media accounts, such as facebook, twitter, google and others.

Our application will provide 2 roles for user accounts. The one reserved for managing purposes will be the admin. And the other one will be the client.

Clients will be able to schedule their events. When creating an event there will be the possibility to add a title and a description in case that some additional information is associated with that event. Also, let's not forget about deadlines. Keep track of deadlines by adding reminders and due dates. Whether for work or personal tasks, our application helps you be more productive.

The system will display a calendar view where all the scheduled events will appear. Starting the morning with a quick snapshot at the application's calendar will help our users plan their day easier.

In case that the user didn't show any activity for a long period of time his/her account will be locked at first and if no requirements of unlocking will appear it will be deleted from out system's database.

I hope such an application will save people's time, energy and budget.

1.1 User Stories

As it was mentioned earlier, our application is going to have 2 types of users. First let's have look from the admin's perspective. It is pretty obvious here. Being an admin means to have the right to manage all the existing accounts and the information associated to it. There is no way of registering a new admin. In order to become one, an already existing admin should add the corresponding role to the account. The admin is the person who knows the policy of the application and makes users to respect it.

When speaking about the second type of user is very important to know that he/she is the reason why the application was built. A lot of possibilities are given to account with such a role. The main feature consists in creating the events and scheduling them in a way that will make our clients become more productive and happy. After the events were created they can be visualized in 2 ways: calendar view or list view.

2 Design Patterns

2.1 Design Patterns

In software engineering, a design pattern is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.[3]

2.2 Uses of Design Pattern

Design patterns can speed up the development process by providing tested, proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation. Reusing design patterns helps to prevent subtle issues that can cause major problems and improves code readability for coders and architects familiar with the patterns.

Often, people only understand how to apply certain software design techniques to certain problems. These techniques are difficult to apply to a broader range of problems. Design patterns provide general solutions, documented in a format that doesn't require specifics tied to a particular problem.[3]

In addition, patterns allow developers to communicate using well-known, well understood names for software interactions. Common design patterns can be improved over time, making them more robust than ad-hoc designs.

2.3 Types of the Design Patterns

There are mainly three types of design patterns illustrated in the Figure 1:

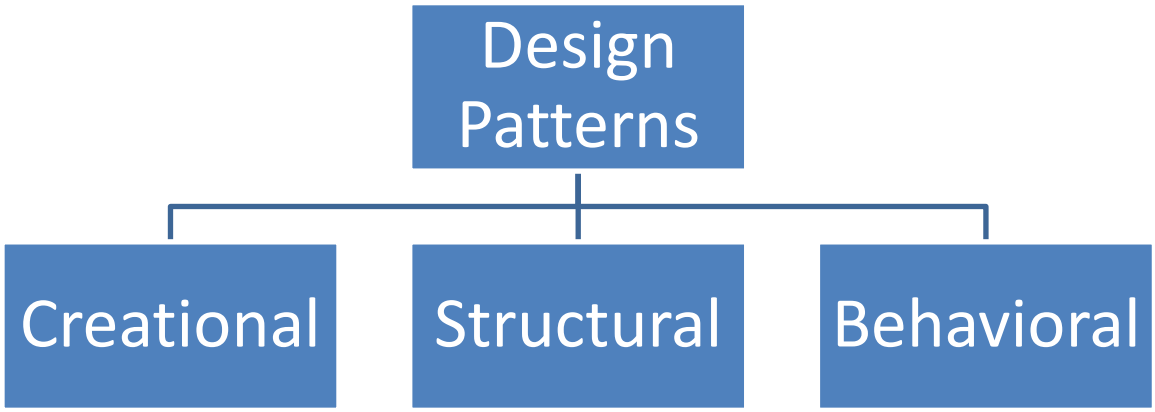


Figure 1: Types of the Design Pattern

2.4 Creational Design Patterns

These design patterns are all about class instantiation or object creation. These patterns can be further categorized into Class-creational patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.[1]

Creational design patterns are Factory Method, Abstract Factory, Builder, Singleton, Object Pool and Prototype.

Factory Method

Factory Method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.[2] The structure of this pattern is shown in Figure 2 using the UML class diagram.

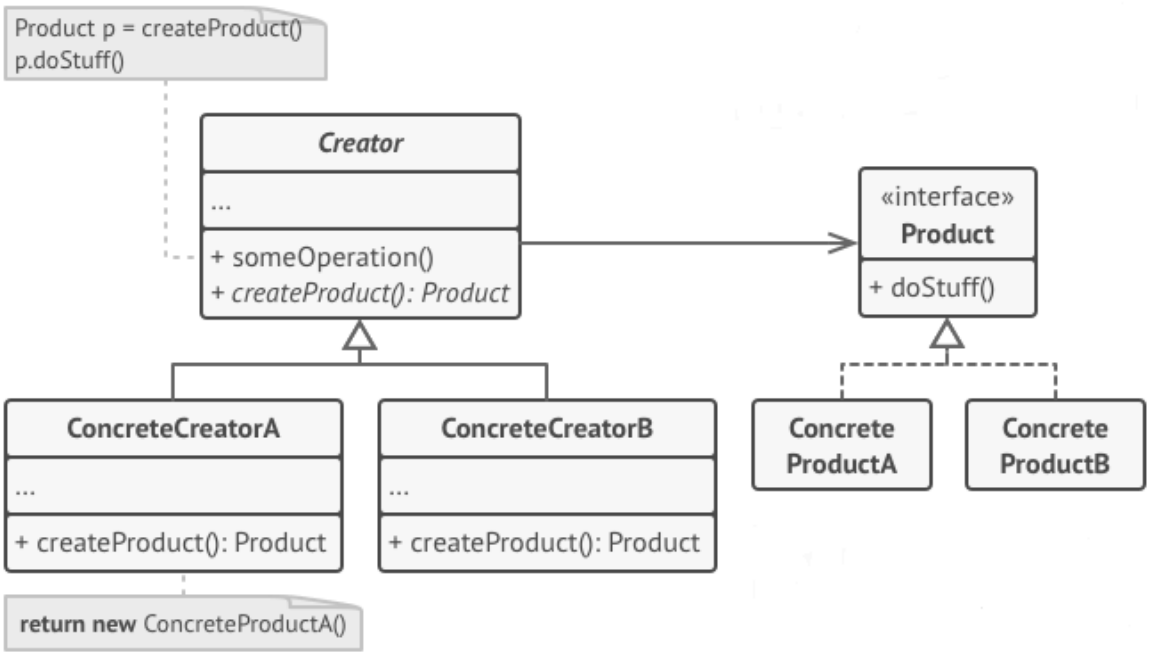


Figure 2: Factory Method Structure

Advantages:

- Separates product construction code from the code that actually uses the product. Therefore it's easier to extend the product construction code independently from the rest of the code;
- Good to use when you want to save system resources by reusing existing objects instead of rebuilding them each time.

Abstract Factory

Abstract Factory is a creational design pattern that lets you produce families of related objects without specifying their concrete classes.[2] The structure of this pattern is shown in Figure 3 using the UML class diagram.

Advantages:

- Provides you with an interface for creating objects from each class of the product family. As long as your code creates objects via this interface, you don't have to worry about creating the wrong variant of a product which doesn't match the products already created by your app.

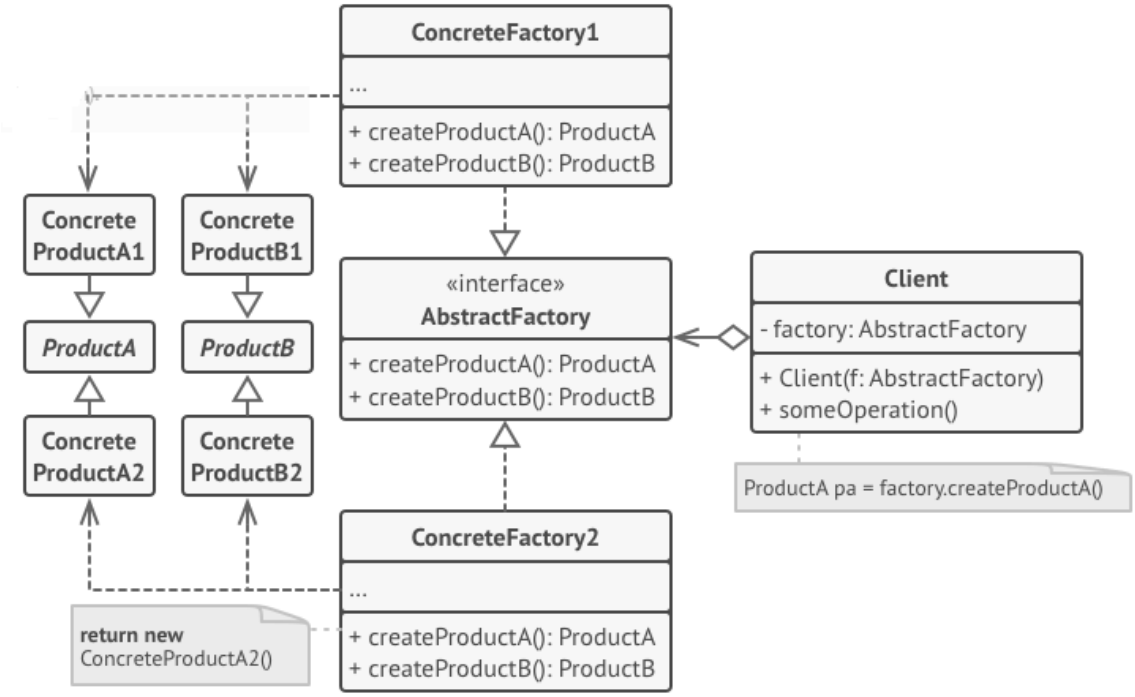


Figure 3: Abstract Factory Structure

Builder

Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.[2] The structure of this pattern is shown in Figure 4 using the UML class diagram.

Advatages:

- Get rid of a “telescopic constructor”;
- Makes the code be able to create different representations of some product;
- Lets you construct products step-by-step. You could defer execution of some steps without breaking the final product. You can even call steps recursively, which comes in handy when you need to build an object tree.

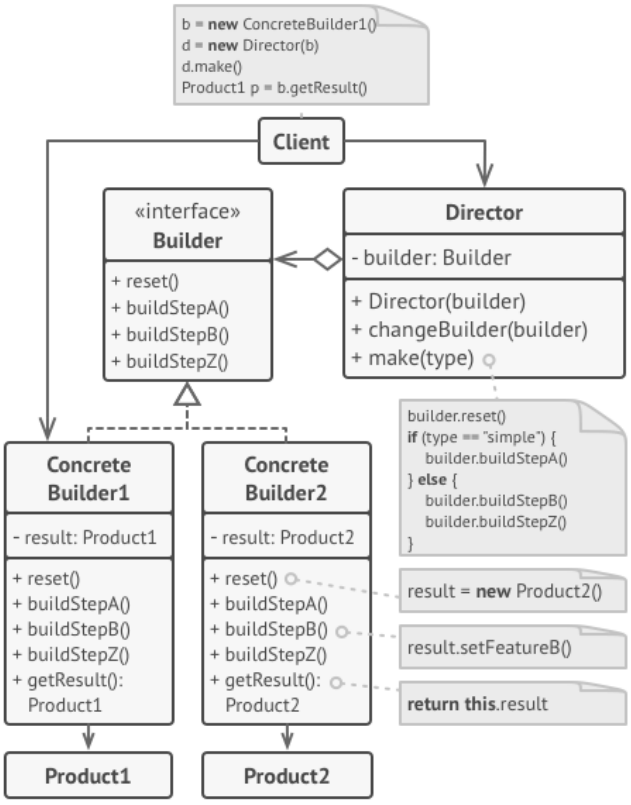


Figure 4: Builder Structure

Prototype

Prototype is a creational design pattern that lets you copy existing objects without making your code dependent on their classes.[2] The structure of this pattern is shown in Figure 5 using the UML class diagram.

Advantages:

- Makes the code not dependent on the concrete classes of objects that you need to copy;
- Reduces the number of subclasses that only differ in the way they initialize their respective objects.

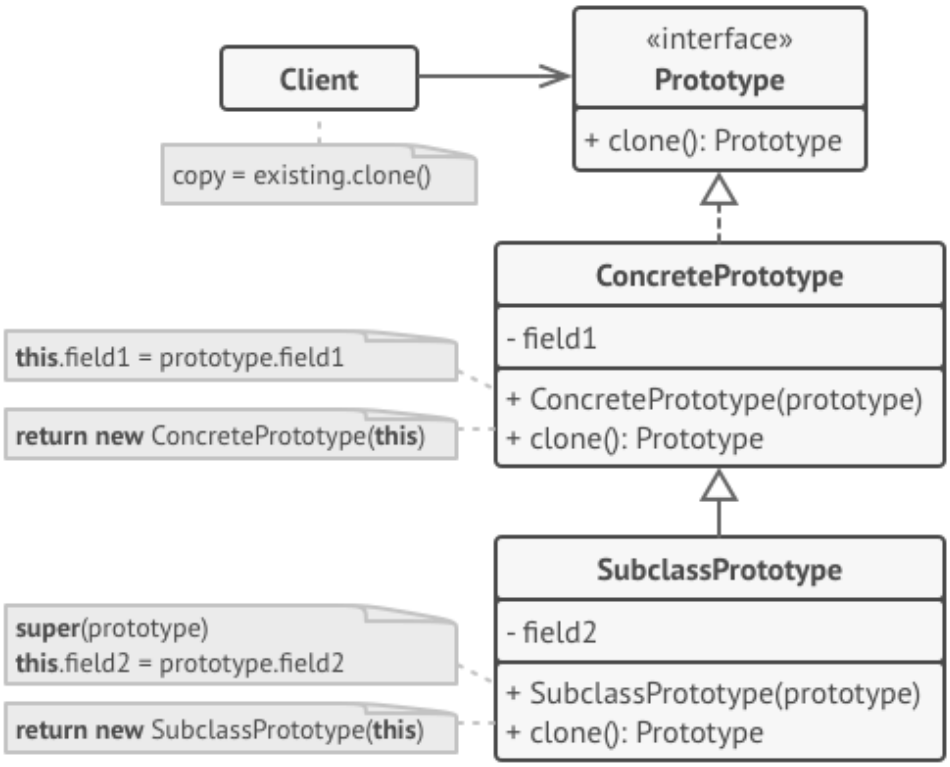


Figure 5: Prototype Structure

Singleton

Singleton is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.[2] The structure of this pattern is shown in Figure 6 using the UML class diagram.

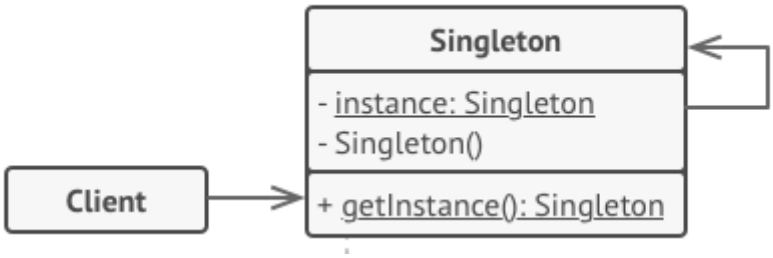


Figure 6: Singleton Structure

Advantages:

- Makes a class in your program have just a single instance available to all clients; for example, a single database object shared by different parts of the program.
- Disables all other means of creating objects of a class except for the special creation method. This method either creates a new object or returns an existing one if it has already been created. So the control over global variables can be restricted.

2.5 Structural Design Patterns

These design patterns are about organizing different classes and objects to form larger structures and provide new functionality, while keeping this structures flexible and efficient.[1]

Structural design patterns are Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Private Class Data and Proxy.

Adapter

Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate. This means that the classes work together because another way for being incompatible are not allowed. Another name for this pattern is: Wrapper.[9] The structure of this pattern is shown in Figure 7 using the UML class diagram.

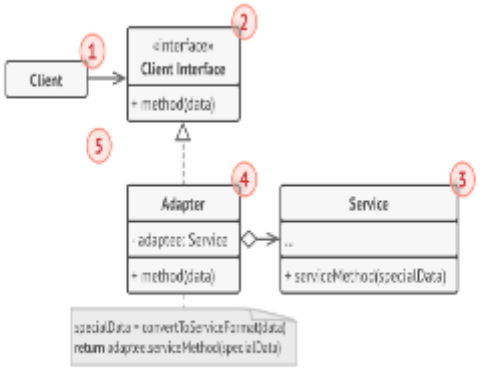


Figure 7: Adapter Structure

Advantages:

- Single responsibility principle: separate the interface or data from the primary business logic;
- Open/closed principle: you can introduce new types of adapters.

Bridge

The bridge pattern let's you split a large class or a set of closely related classes into two separate hierarchies-abstraction and implementation-which can be developed independently of each other.[9]

The bridge pattern should be used in case of modifying or switching the properties in run-time. Also, through this pattern you can separate the abstractization and the implementation independently, decoupled.[9] The structure of this pattern is shown in Figure 8 using the UML class diagram.

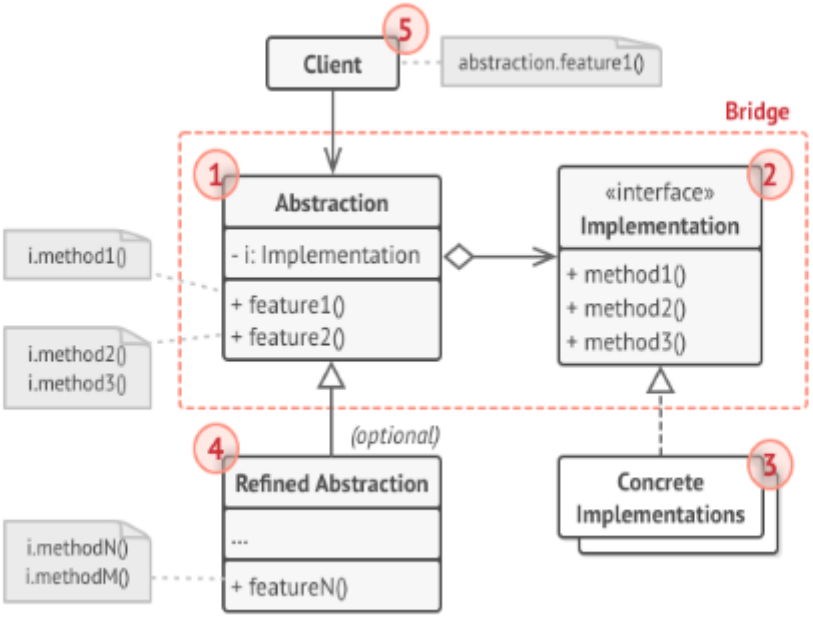


Figure 8: Bridge Structure

Advantages:

- You can create platform-independent classes and apps.
- The client code works with high-level of abstractization.
- Open/closed principle.
- Single resposability principle.

Composite

The composite pattern compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. This pattern is based on composition relation between objects.[9] The structure of this pattern is shown in Figure 9 using the UML class diagram.

Advantages:

- You can work with complex tree structures more conveniently, use polymorphism and recursion to your advantage.
- Open/closed principle.

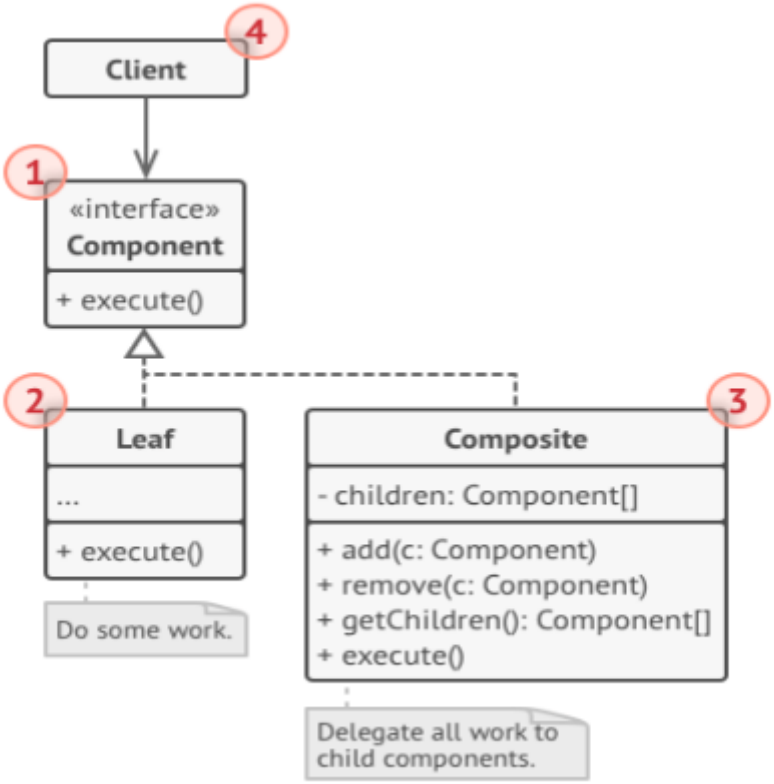


Figure 9: Composite Structure

Decorator

Decorator is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors. Attach additional functionalities to an object dynamically, providing a flexible alternative to subclassing for extinding functionality.[9] The structure of this pattern is shown in Figure 10 using the UML class diagram.

Advantages:

- You can extend an object behavior without making a new subclass.
- You can add or remove responsabilities from an object at runtime.
- You can combine several behaviors by wrapping an object into multiple decorators.

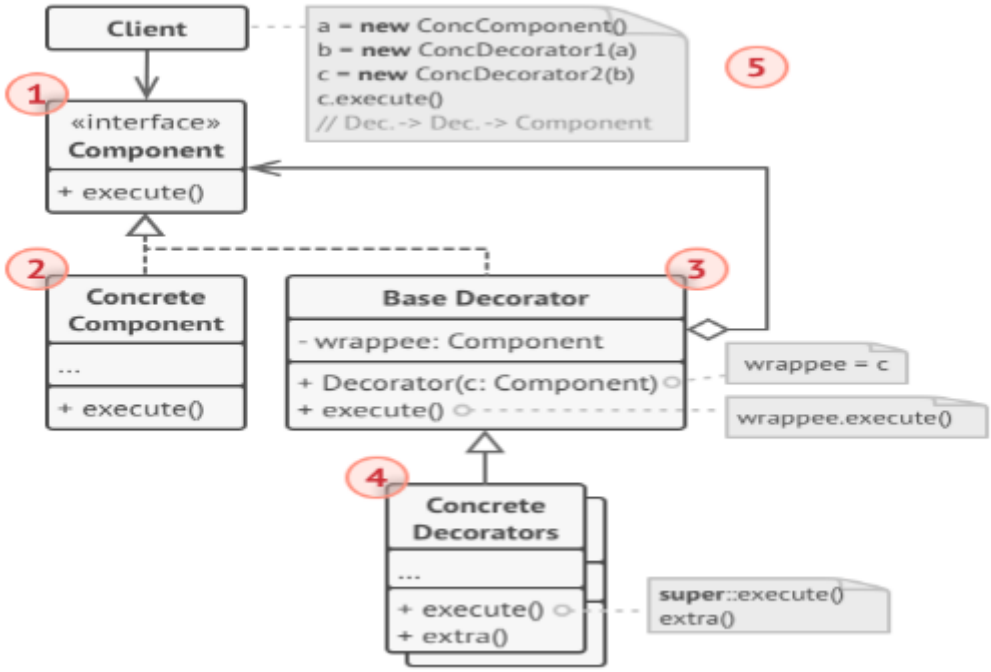


Figure 10: Decorator Structure

Facade

Facade design pattern provides a simplified interface to a library, a framework, or any other complex set of classes. This helps us structuring a system into subsystems help us to reduce the complexity. So, the common design goal is to minimize the communication and dependencies between subsystems.[9] The structure of this pattern is shown in Figure 11 using the UML class diagram.

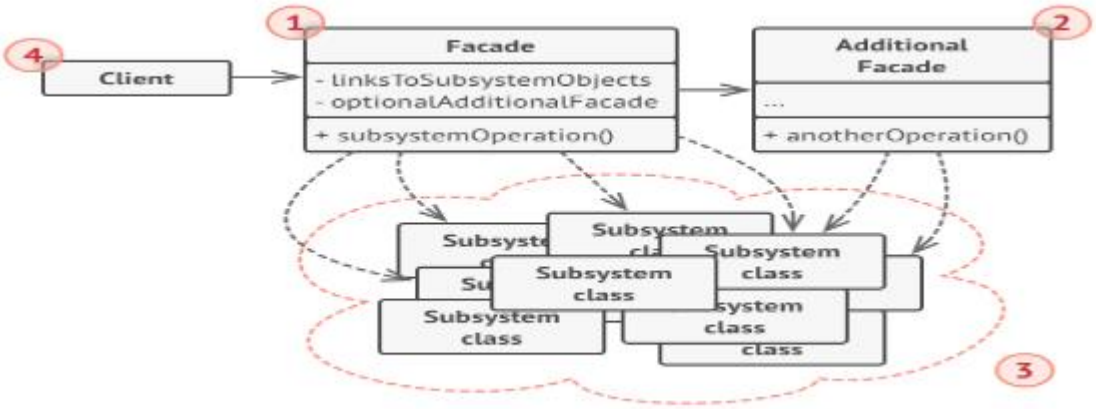


Figure 11: Facade Structure

Advantages:

- You can isolate your code from the complexity of a subsystem.

Flyweight

Flyweight is a structural design pattern that lets you fit more into the available amount of RAM by sharing common parts of state between multiple objects instead keeping all of the data in each object. So, we need to use this pattern when the resources cost is high, because the amount of objects is high or then the application does not depend on the objects identity. The flyweight object has 2 types of data: intrinsic and extrinsic.[9] The structure of this pattern is shown in Figure 12 using the UML class diagram.

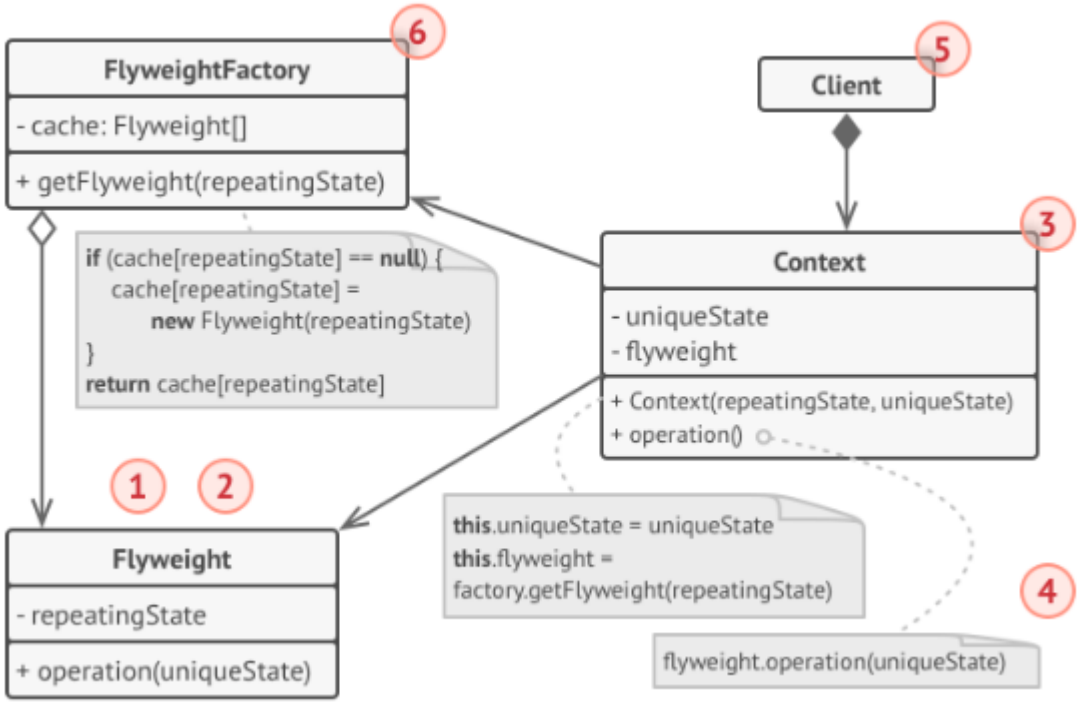


Figure 12: Flyweight Structure

Advantages:

- You can save lots of RAM, assuming your program has tons of similar objects.

Proxy

Proxy design pattern that lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.[9] The structure of this pattern is shown in Figure 13 using the UML class diagram.

Advantages:

- You can control the service object without clients knowing about it.
- You can manage the lifecycle of the service object when clients don't care about it.

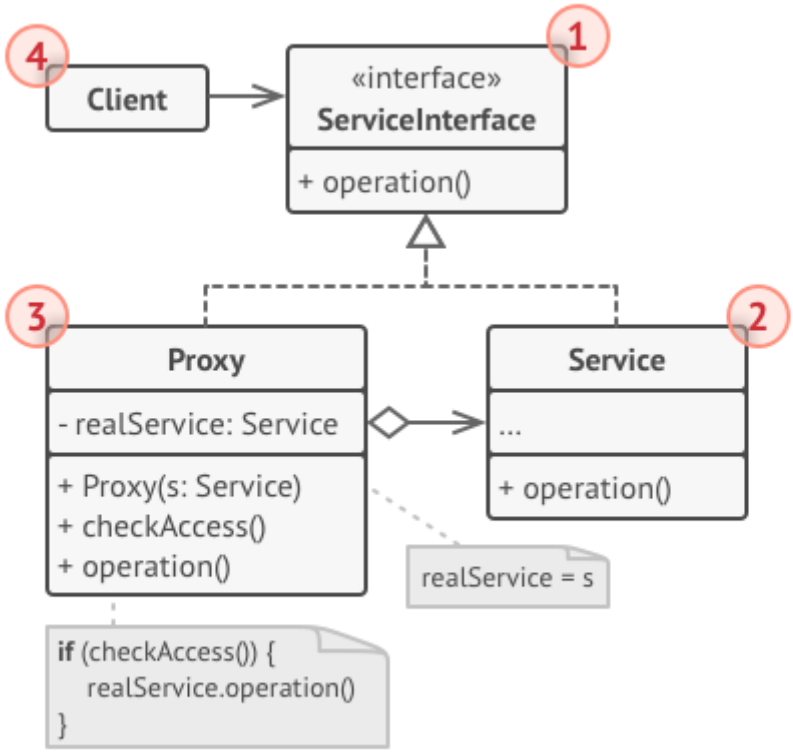


Figure 13: Proxy Structure

2.6 Behavioral Design Patterns

Behavioral patterns are about identifying common communication patterns between objects and realize these patterns.[1]

Behavioral patterns are Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Null Object, Observer, State, Strategy, Template method, Visitor.

Chain of responsibility

Chain of Responsibility is a behavioral design pattern that lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.[4] The structure of this pattern is shown in Figure 14 using the UML class diagram.

Advantages:

- Use the Chain of Responsibility pattern when your program is expected to process different kinds of requests in various ways, but the exact types of requests and their sequences are unknown beforehand.
- Executes several handlers in a particular order.
- The set of handlers and their order are supposed to change at runtime.

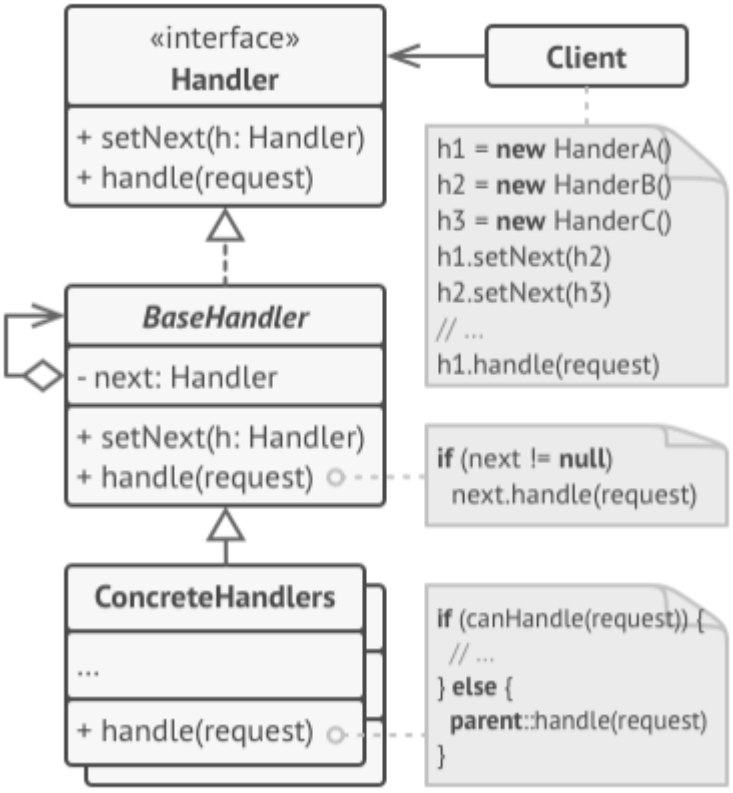


Figure 14: Chain of responsibility Structure

Command

Command is a behavioral design pattern that turns a request into a stand-alone object that contains all information about the request. This transformation lets you parameterize methods with different requests, delay or queue a request's execution, and support undoable operations. In the Figure you can find the structure of this pattern.[4] The structure of this pattern is shown in Figure 15 using the UML class diagram.

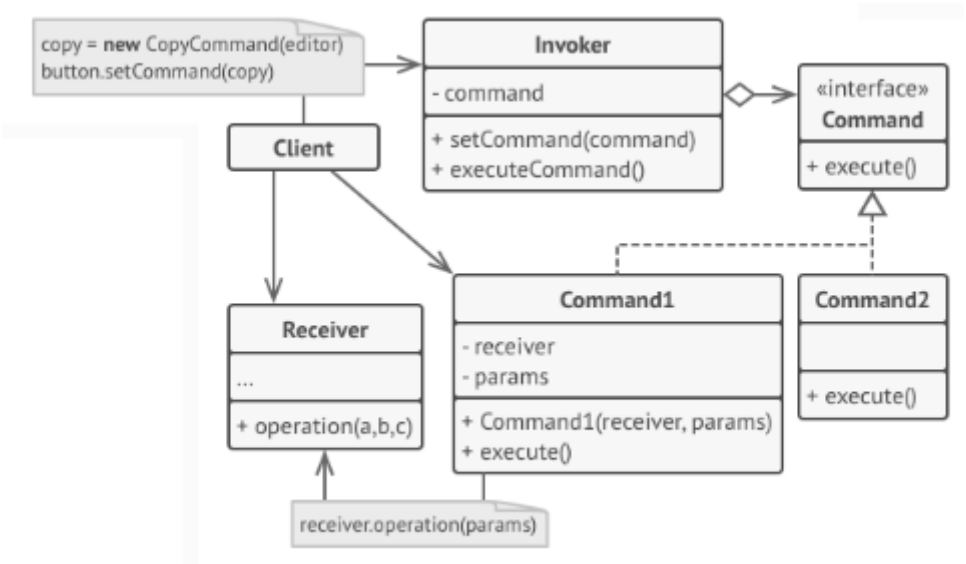


Figure 15: Command Structure

Advantages:

- Allows you parametrize objects with operations.
- Allows you queue operations, schedule their execution, or execute them remotely.
- Permits to implement reversible operations.

Iterator

Iterator is a behavioral design pattern that lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.). In the Figure you can find the structure of this pattern.[4] The structure of this pattern is shown in Figure 16 using the UML class diagram.

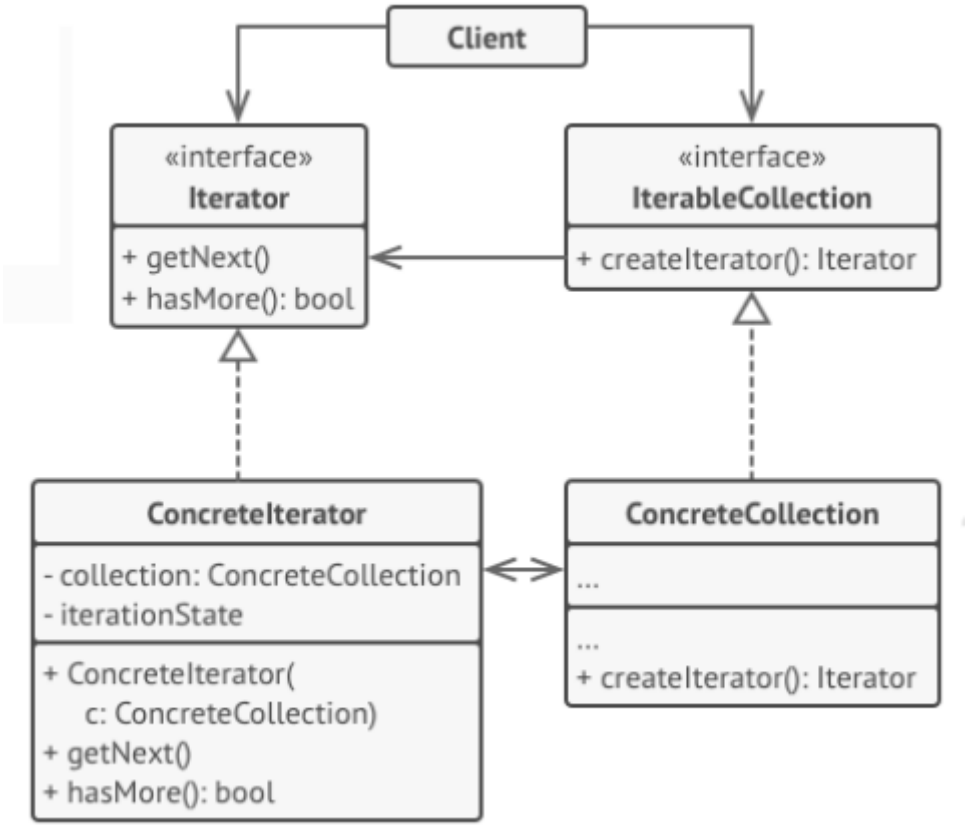


Figure 16: Iterator Structure

Advantages:

- Reduces duplication of the traversal code across your app.
- Makes the iterating process of a complex data structure easier
- Makes the code be able to traverse different data structures or when types of these structures are unknown beforehand.

Mediator

Mediator is a behavioral design pattern that lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.[4] The structure of this pattern is shown in Figure 17 using the UML class diagram.

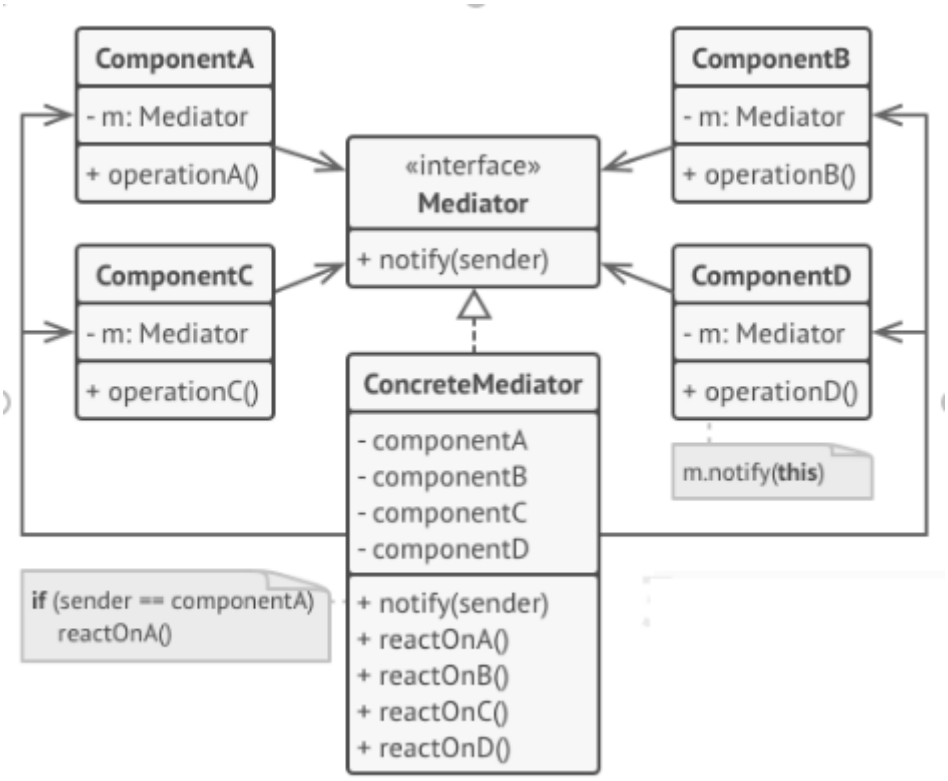


Figure 17: Mediator Structure

Advantages:

- The pattern lets you extract all the relationships between classes into a separate class, isolating any changes to a specific component from the rest of the components.

Memento

Memento is a behavioral design pattern that lets you save and restore the previous state of an object without revealing the details of its implementation.[4] The structure of this pattern is shown in Figure 18 using the UML class diagram.

Advantages:

- Produces snapshots of the object's state to be able to restore a previous state of the object.

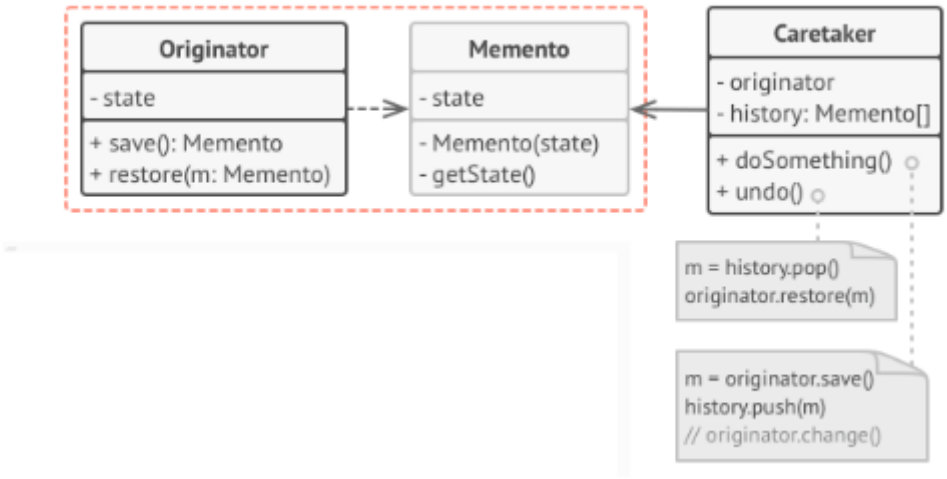


Figure 18: Memento Structure

Observer

Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.[4] The structure of this pattern is shown in Figure 19 using the UML class diagram.

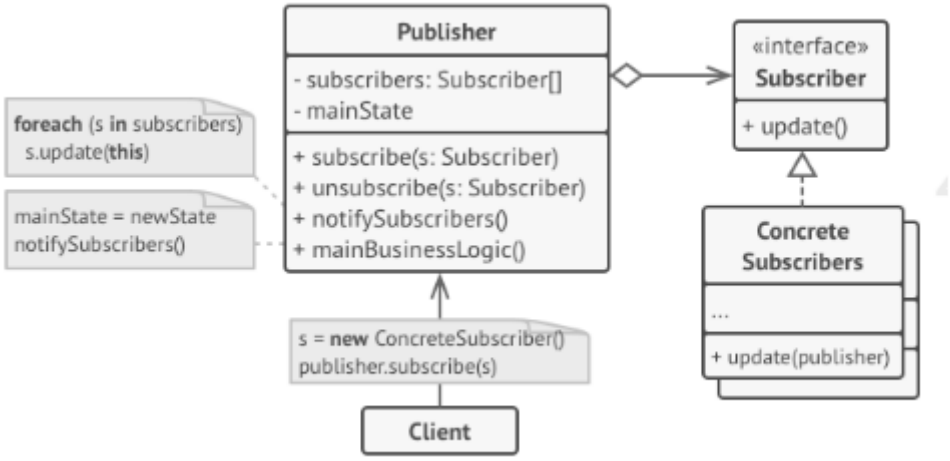


Figure 19: Observer Structure

Advantages:

- Makes some objects in your app observe others, but only for a limited time or in specific cases.
- Use the Observer pattern when changes to the state of one object may require changing other objects, and the actual set of objects is unknown beforehand or changes dynamically.

State

State is a behavioral design pattern that lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.[4] The structure of this pattern is shown in Figure 20 using the UML class diagram.

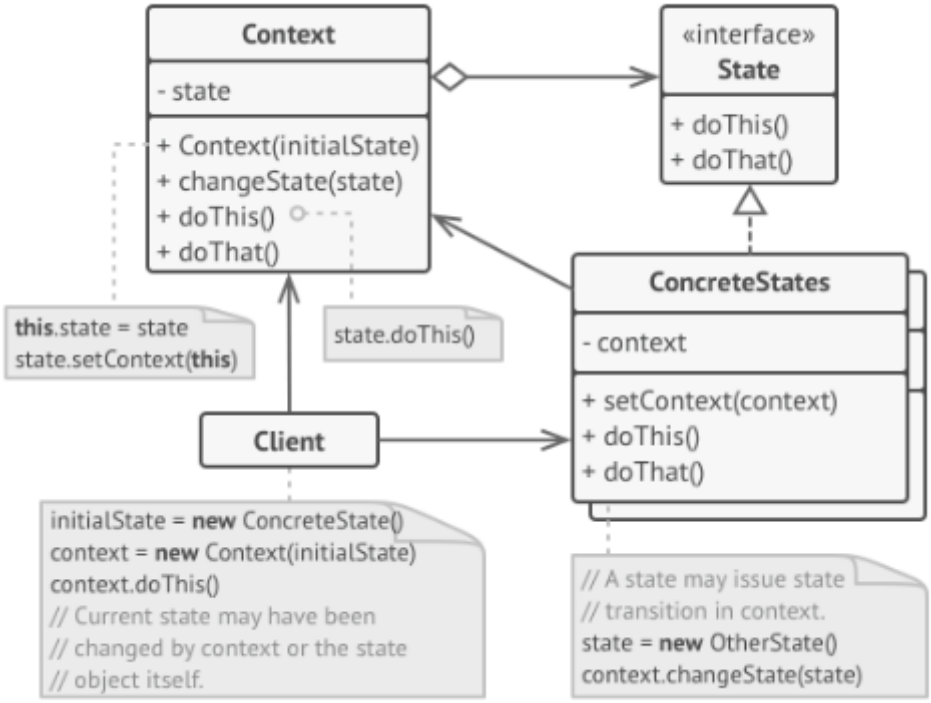


Figure 20: State Pattern

Advantages:

- Use the State pattern when you have an object that behaves differently depending on its current state, the number of states is enormous, and the state-specific code changes frequently.
- Use State when you have a lot of duplicate code across similar states and transitions of a condition-based state machine.
- Use the pattern when you have a class polluted with massive conditionals that alter how the class behaves according to the current values of the class's fields.

Strategy

Strategy is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.[4] The structure of this pattern is shown in Figure 21 using the UML class diagram.

Advantages:

- Use the Strategy pattern when you want to use different variants of an algorithm within an object and be able to switch from one algorithm to another during runtime.
- Use the Strategy when you have a lot of similar classes that only differ in the way they execute some behavior.

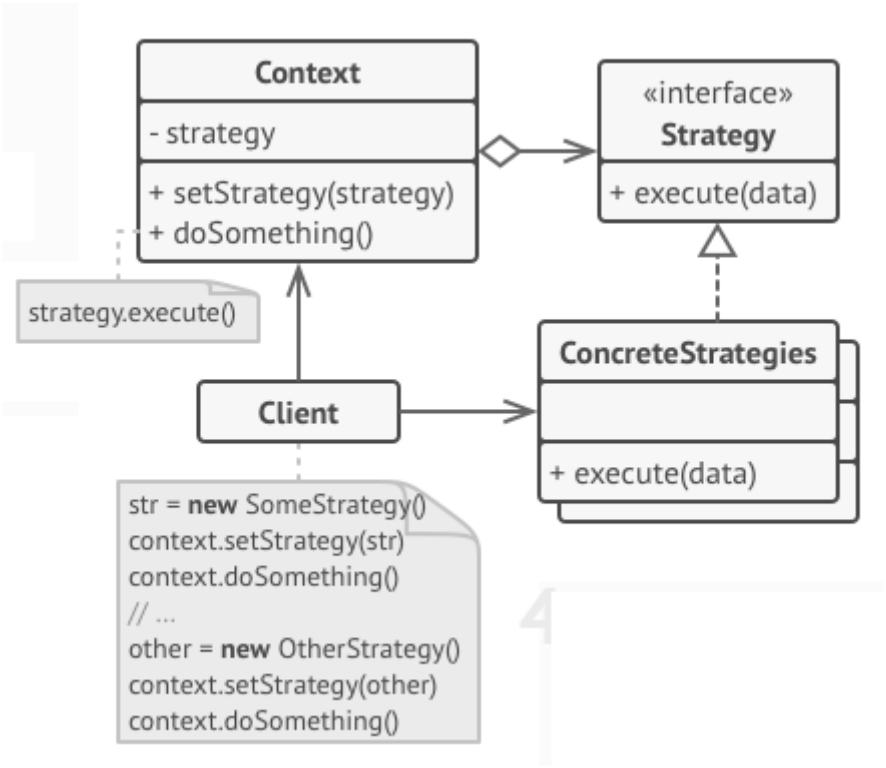


Figure 21: Strategy Structure

Template Method

Template Method is a behavioral design pattern that defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.[4] The structure of this pattern is shown in Figure 22 using the UML class diagram.

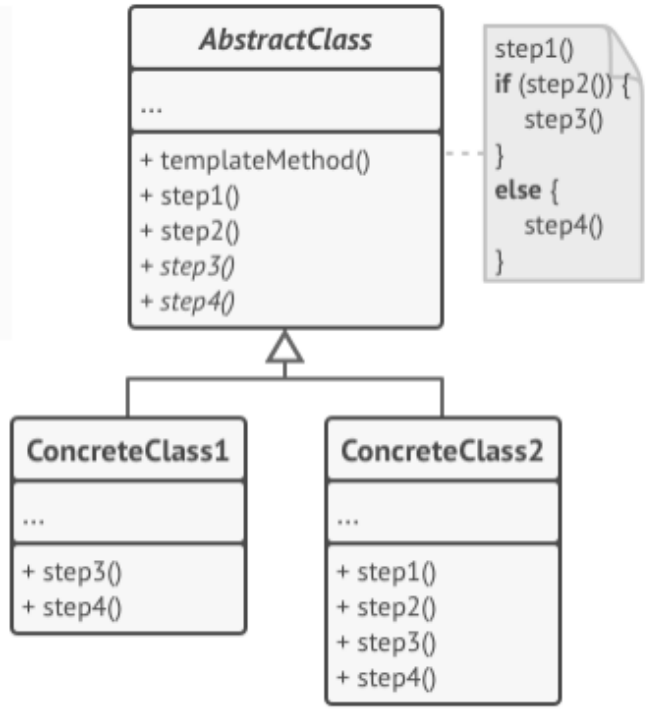


Figure 22: Template Method Structure

Advantages:

- Use the Template Method pattern when you want to let clients extend only particular steps of an algorithm, but not the whole algorithm or its structure.
- Use the Template Method pattern when you want to let clients extend only particular steps of an algorithm, but not the whole algorithm or its structure.

Visitor

Visitor is a behavioral design pattern that lets you separate algorithms from the objects on which they operate.[4] The structure of this pattern is shown in Figure 23 using the UML class diagram.

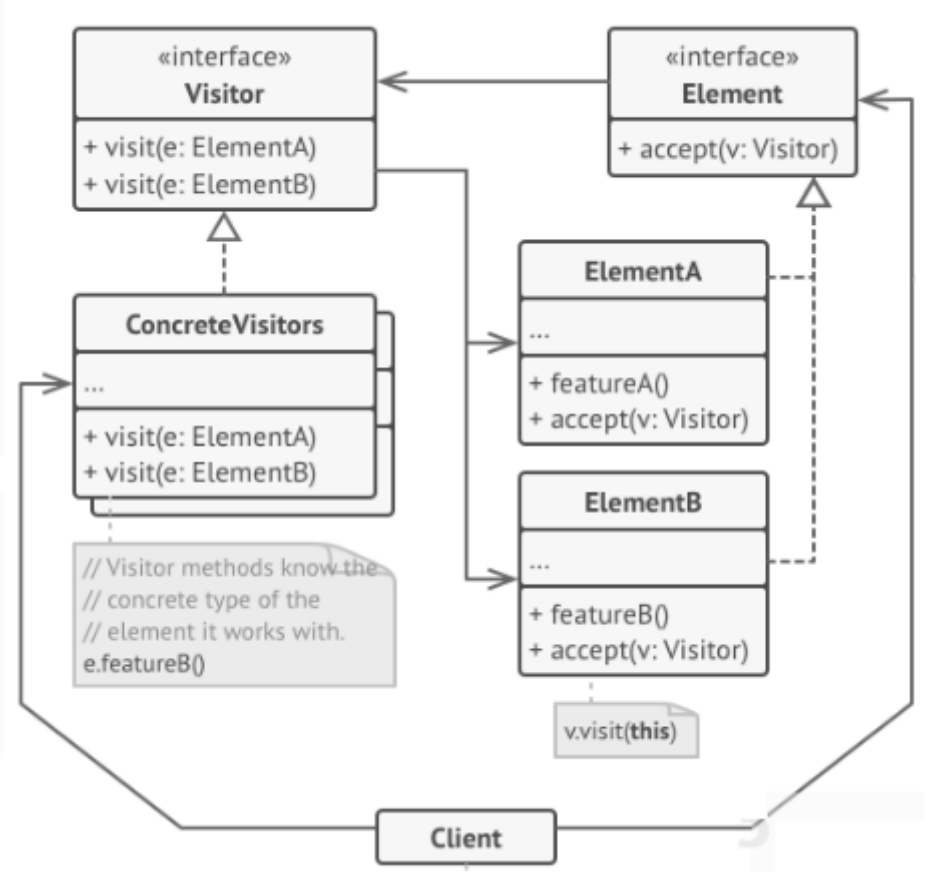


Figure 23: Visitor Structure

Advantages:

- Use the Visitor when you need to perform an operation on all elements of a complex object structure (for example, an object tree).
- Use the Visitor to clean up the business logic of auxiliary behaviors.
- Use the pattern when a behavior makes sense only in some classes of a class hierarchy, but not in others.

3. Domain Analysis

Domain's Description:

- An application that has a simple purpose: help its users. The assistance will cover 3 aspects: time saving, energy saving and productivity improving. We thought about the exact domain and we came up with the idea that our application will fit better in Social Organization.

Theme's Importance:

- As it was said the main goal of our application is to help people manage their time correctly and make sure everything is going as they have planned it. Having a society which know exactly what, when and how they have to act, will reduce the chaos.

Similar Projects and Comparison:

- There are similar projects on the market. Some of them are develop by IT giants such as Google or Apple, other by smaller companies. We have analyzed some of them which we consider to be the best and came up with our own solution which will encapsulate all the good points developed by other teams and at the same time will correct the defects we have observed there. Also, we did not forget about our own signature.

Goals and Objectives:

- Improve productivity;
- Save time;
- Save energy.

4. System Analysis

In order to describe behaviour of the system we have developed we used 2 types of UML diagrams: use case diagram and activity diagram.

4.1 Use Case Diagram

The Use Case Diagram was created in order to describe the behaviour of a “client” actor, its illustration can be found in Figure 24.

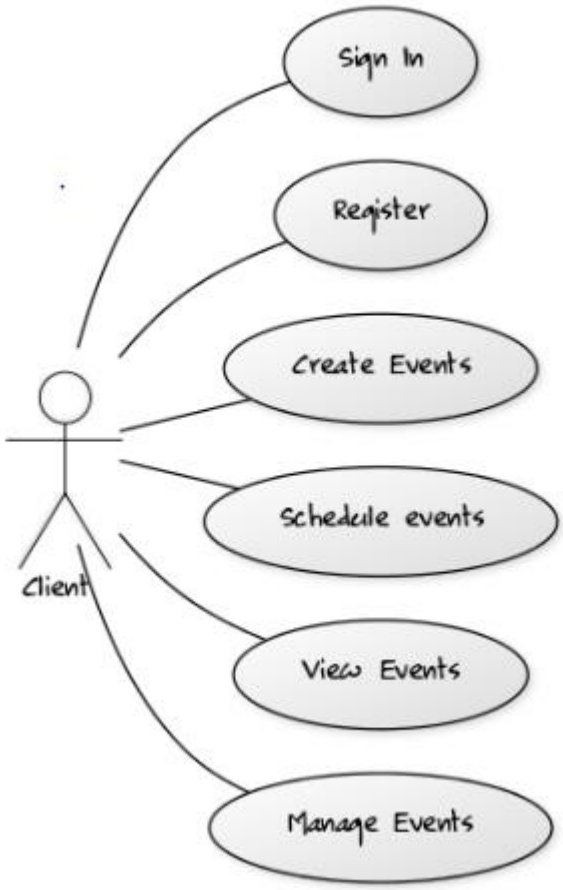


Figure 24[5]: Use Case Diagram of the System which describes what a client can do

4.2 Activity Diagram

In Figure 25, you may find the Activity Diagram where are shown the states and the transitions used in order to Create an Event in our application. As it can be observed, an event can be created only by the users which are logged in. Otherwise, it is not possible. This thing is represented in the Diagram using the Diamond symbol which represents a decision and has 2 alternate paths "yes" and "no". A fork node is used to split a single incoming flow into 3 concurrent flows. In order to join those 3 concurrent flows back into a single outgoing flow a join node is used. After entering necessary information the user can save the changes he/she made.

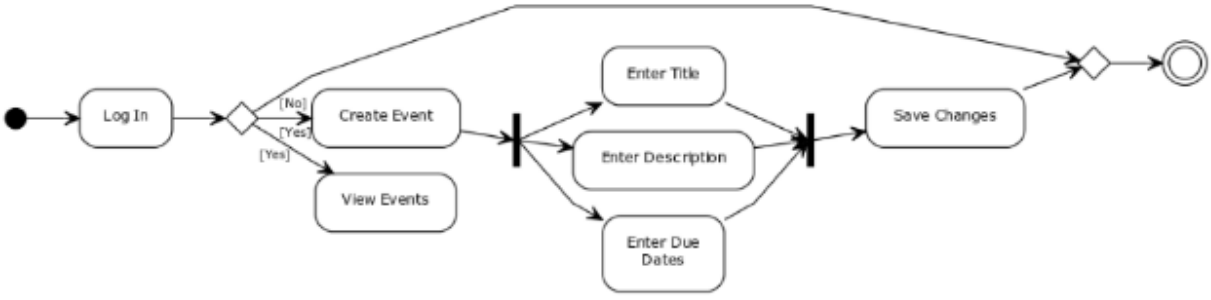


Figure 25[5]: Activity Diagram of the System which describes “Create an Event” process

5. System Design

Design of a system can be easily represented using UML diagrams. Depending on the needs different diagrams can be created. For a better understanding about how our application was built and how it is working we have used next types of diagrams: Statechart diagram, Sequence diagram and Component Diagram.

5.1 Statechart Diagram

A statechart diagram which describes the process of viewing the events together with its available options: calendar view and list view is presented in Figure 26. In order to describe this process during its life cycle simple and composite states were used. The composite state is ViewEvents.

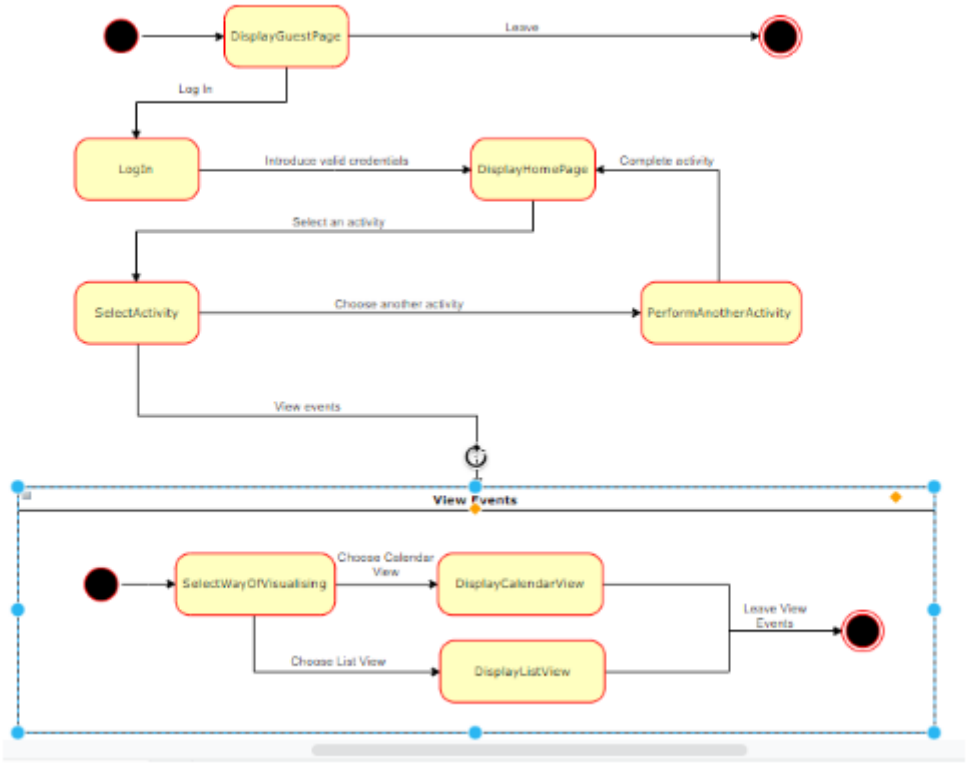
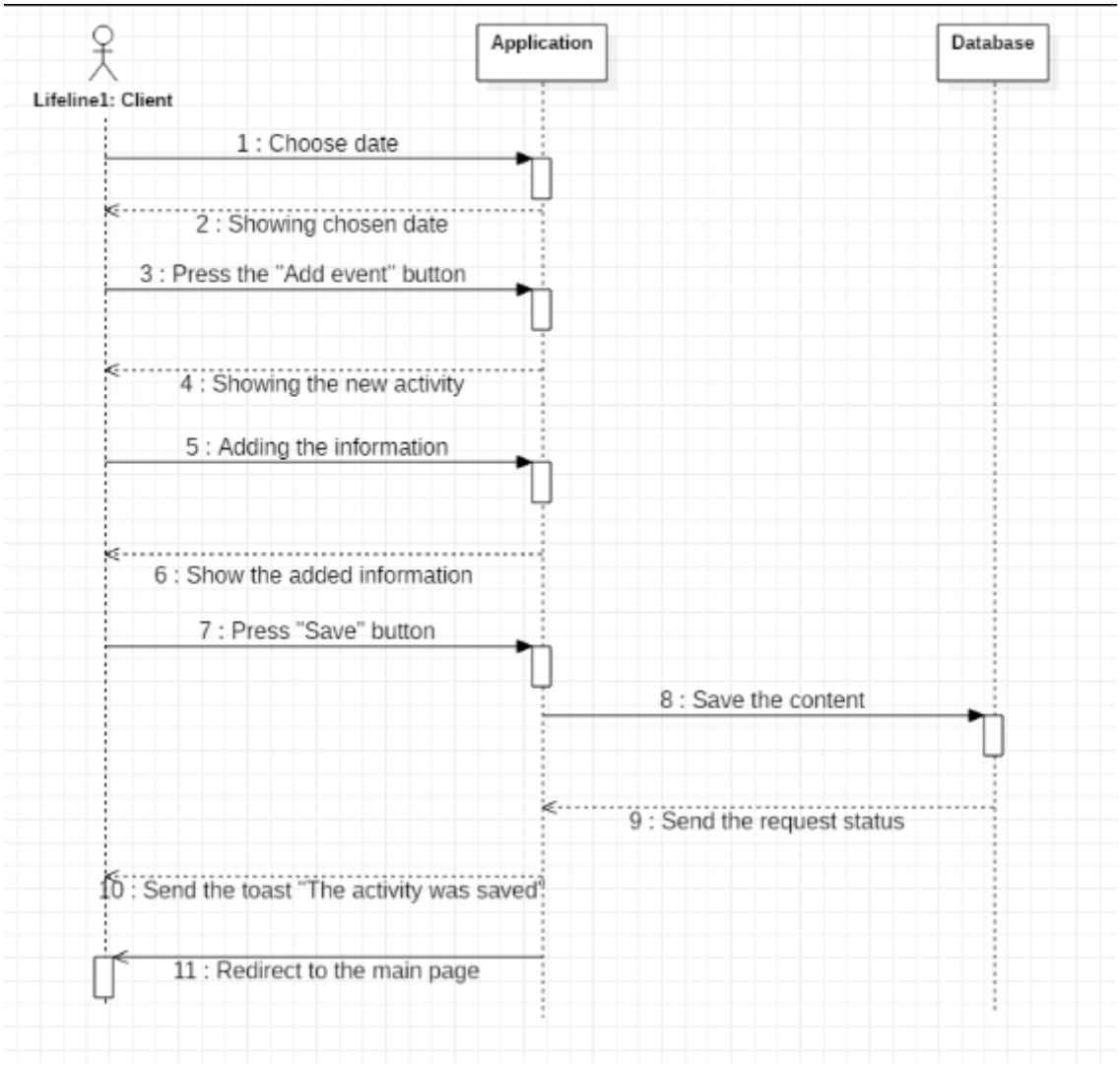


Figure 26[8]: Statechart Diagram of the System which describes the "View Events" process

5.2 Sequence Diagram

The sequence diagram is used to describe some type of interactions among the different elements in the model. This interaction is a part of dynamic behaviour of the system. Sequence diagram emphasizes on time sequence of messages.[7]

Component diagrams are different in terms of nature and behaviour. Component diagrams are used to model the physical aspects of a system. Physical aspects are the elements such as executables, libraries, files, documents, etc. which reside in a node.[7] In Figure 27 is shown our application's sequence and component diagrams.



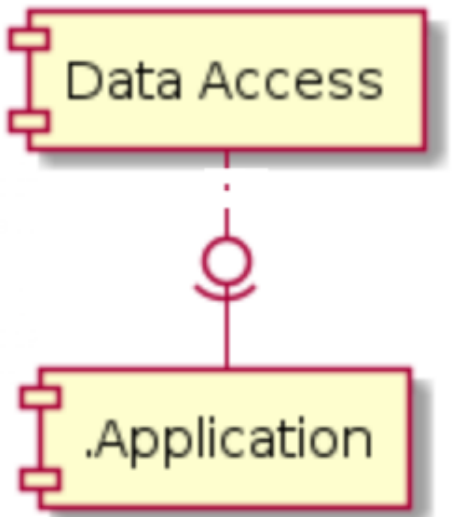


Figure 27: Sequence and Component Diagrams of the System

6. Implementation of the System

6.1 Used Technologies

The reason why we have chosen to develop a mobile application is that businesses today are turning to mobile apps to expand their strategies to tap a higher customer base. With over 86.8% share, Android OS dominates the mobile app development market. It is expected to surge even further in the coming years. Taking in consideration the above mentioned facts I think it is obvious why our choice felt other the Android OS.[6]

Android mobile applications have influenced most of the industries as part of the digital revolution today. Even though iOS is a popular platform, listed in Figure 28 there are some of the reasons why android development is by far the best and a leading platform for businesses.[6]

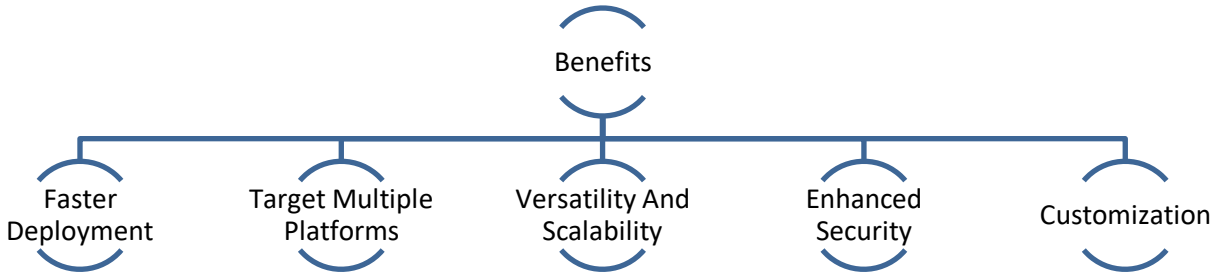


Figure 28: Benefits of Android Development

6.2 Used Design Patterns

Here we will describe the design patterns we have implemented in our application. Also we will try to specify why certain pattern were used and how they have helped us.

Singleton

Using Singleton we ensure that there is going to be just one instance of *CustomEventXMLParser* class which help us in performing CRUD operations with client events. In Figure 29 can be find the class diagram which describes this pattern's structure in our application.

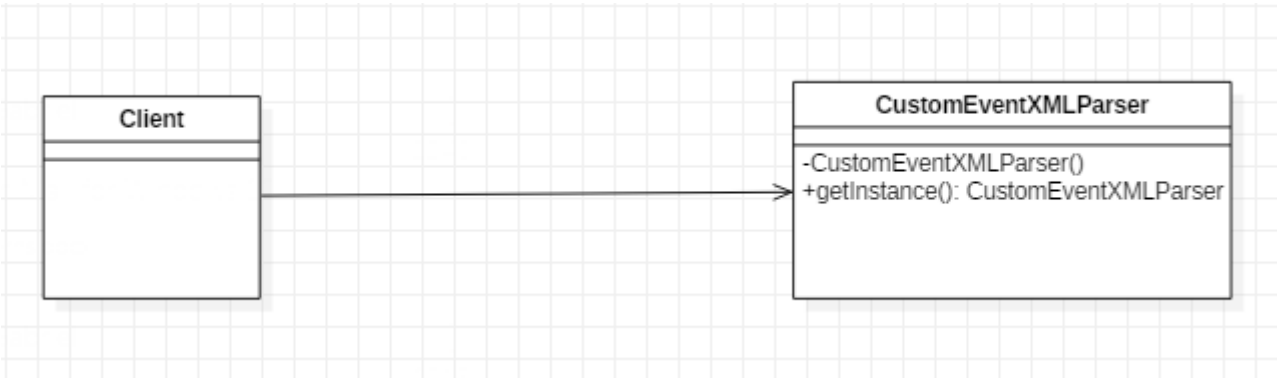


Figure 29: Singleton in our application

Builder

We build objects that have many parameters. Build custom event objects which has many arguments. In Figure 30 can be find the class diagram which describes this pattern's structure in our application.

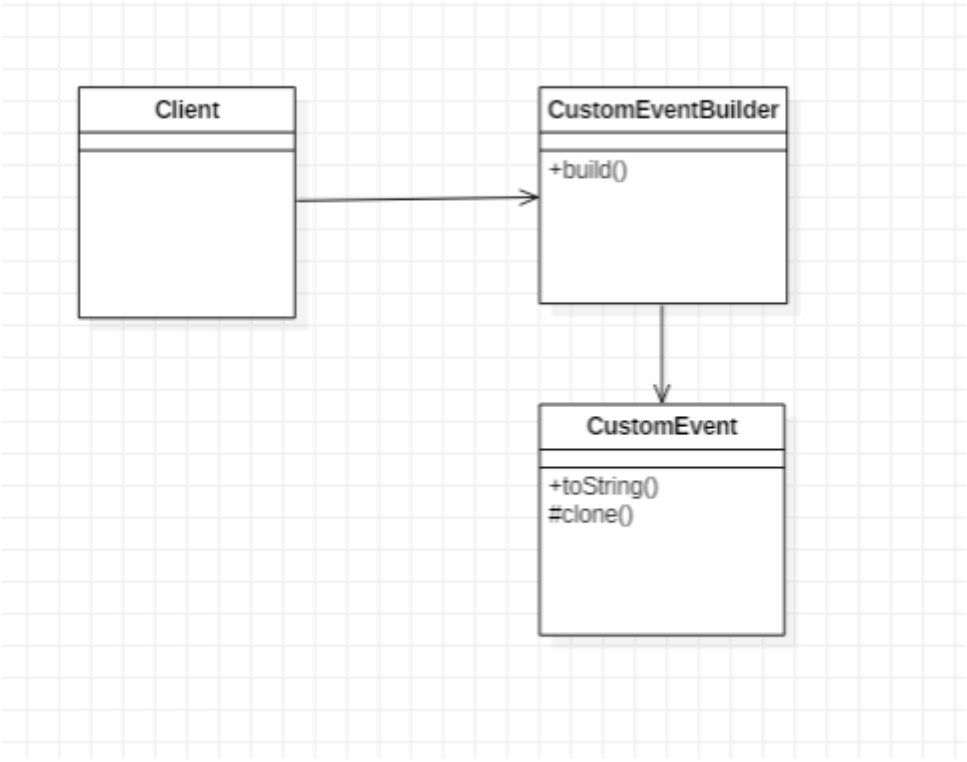


Figure 30: Builder in our application

Adapter

We use Adapter for allowing communication between activities and *CustomEventXMLParser* through *CustomEventParserXMLAdapter*. In Figure 31 can be find the class diagram which describes this pattern's structure in our application.

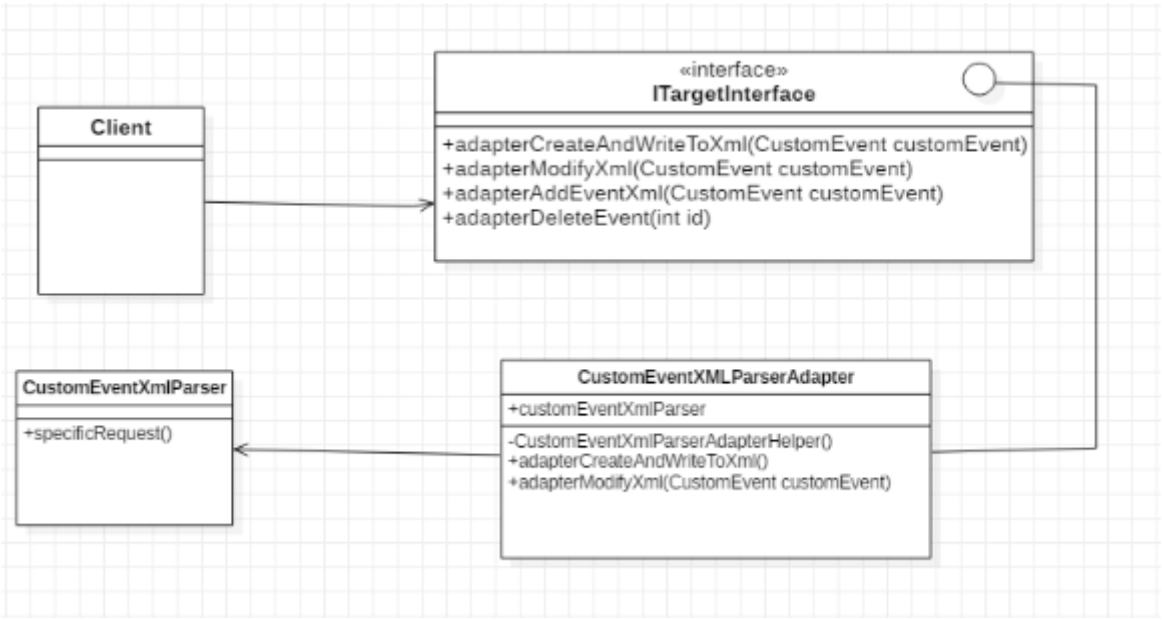


Figure 31: Adapter in our application

Flyweight

The Flyweight helps us to reuse the *CustomEvent* object without creating this one more time. In Figure 32 can be find the class diagram which describes this pattern's structure in our application.

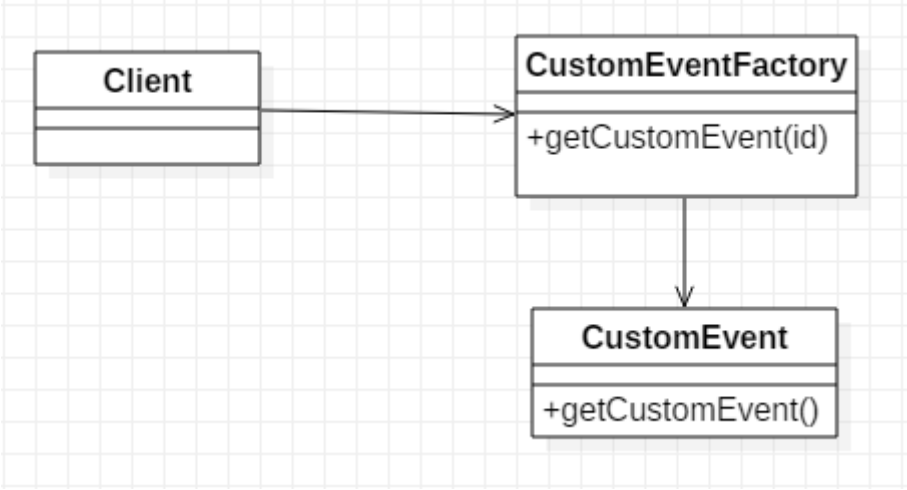


Figure 32: Flyweight in our application

Prototype

We have used this for copying the *CustomEvent* object to a new object and then modify it according to our needs. In Figure 33 can be find the class diagram which describes this pattern's structure in our application.

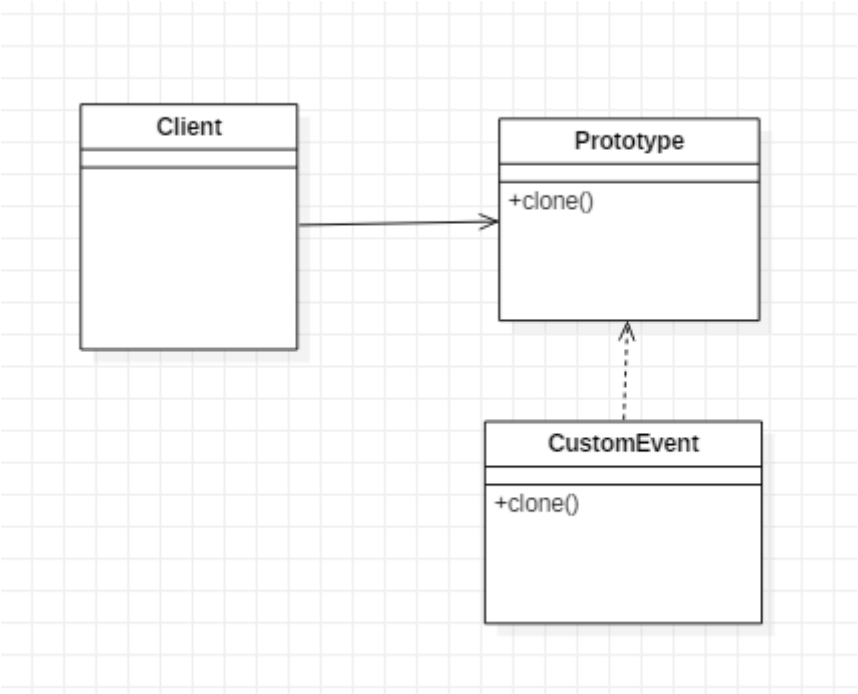


Figure 33: Prototype in our application

Null Object

We have used Null Object pattern to prevent handling of *NullPointerException* by forming the default behavior of Object. In Figure 34 can be find the class diagram which describes this pattern's structure in our application.

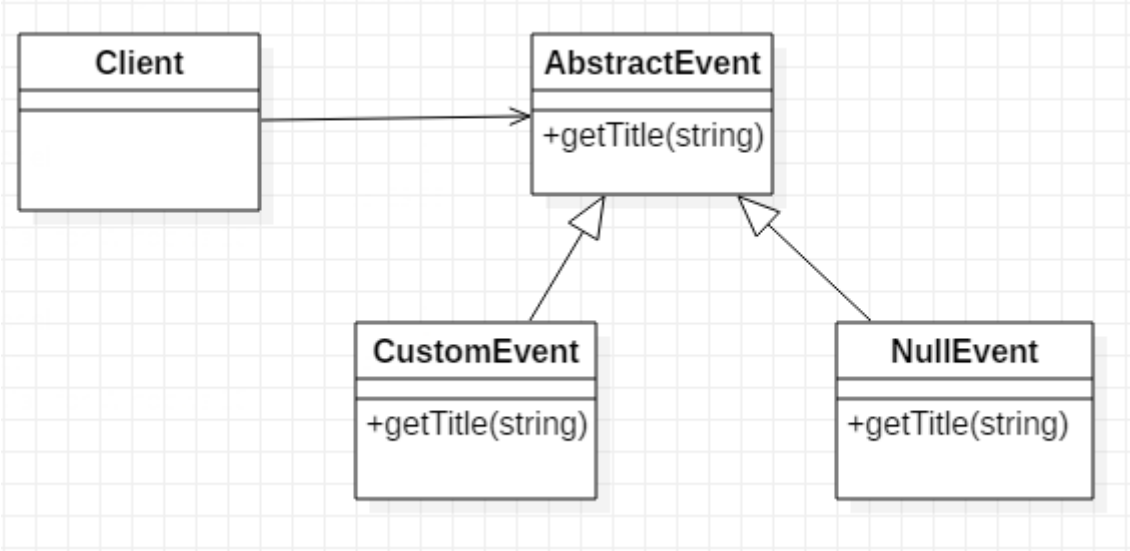


Figure 34: Null Object in our application

Strategy

We have used Strategy for having multiple algorithms of saving our events. In Figure 35 can be find the class diagram which describes this pattern's structure in our application.

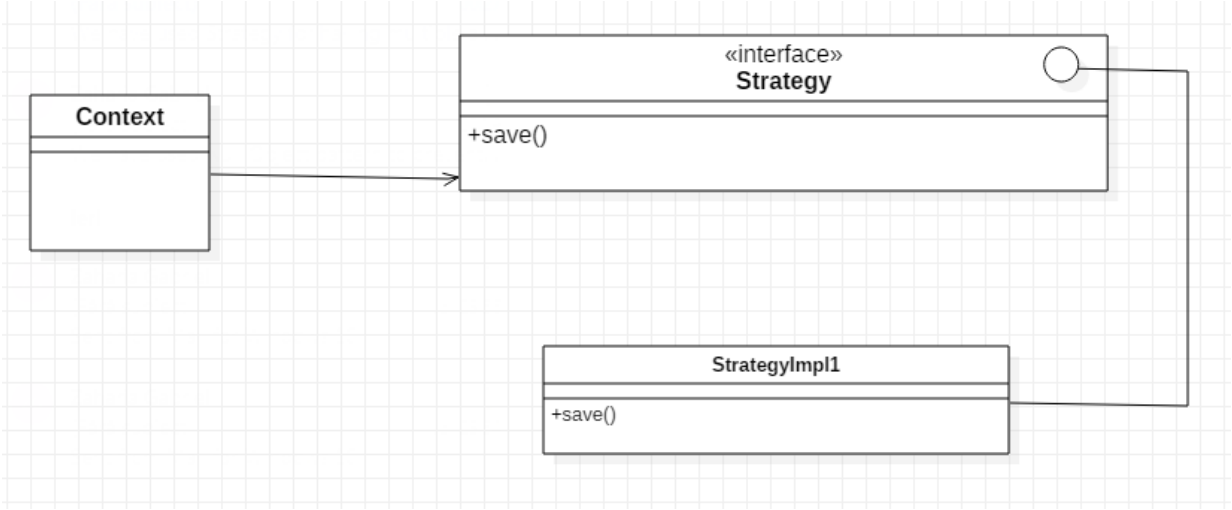


Figure 35: Strategy in our application

Iterator

We use the Iterator pattern for traversing through our collection of events. In Figure 36 can be find the class diagram which describes this pattern's structure in our application.

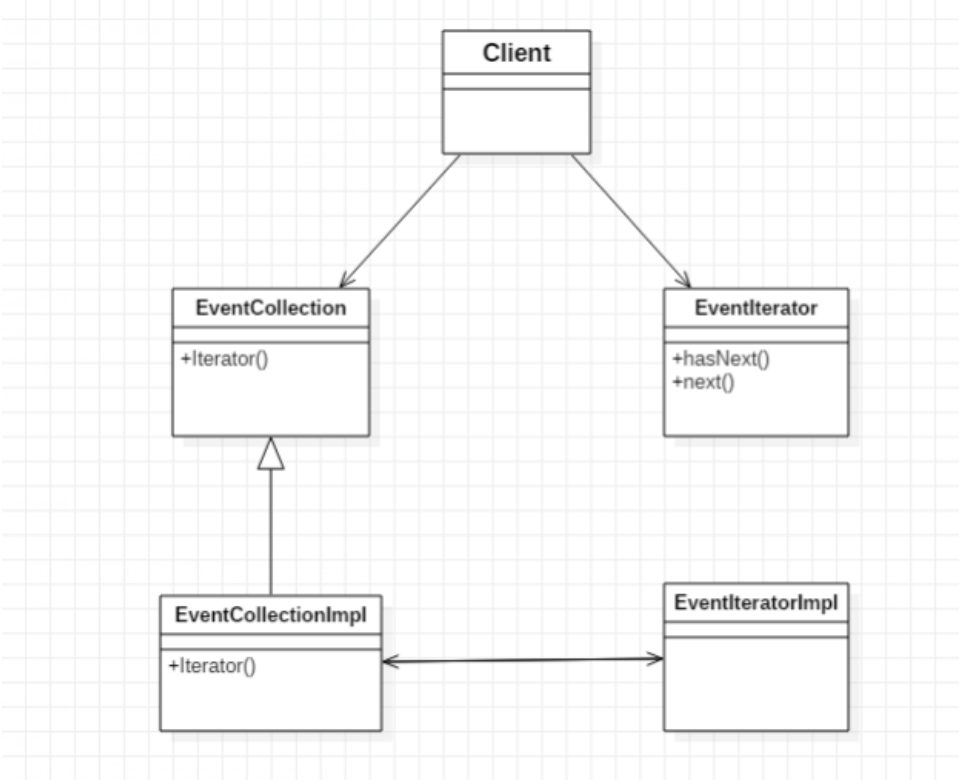


Figure 36: Iterator in our application

7. Conclusion

In this course work we have learned how to develop an application using some of the good practices which are known under “design patterns” name. By means of them our code was built in a clean and readable way. Also let’s not forget about their impact over the performance. Applying the corresponding patterns helped us avoid some of the commonly occurring problems.

We were given the opportunity to choose the application’s subject and scope. Based on our own needs and experience we have decided to build an application which will help us manage the time in a correct way. The main purpose of the application is to save people’s time, energy and make them more productive.

One problem which was not solved in this version was synchronizing the phone’s calendar with the application’s calendar. This is going to be our new challenge for the further development. Also one thing we thought will be nice to have is to make as a desktop version too. So there are things to work on.

From our point of view there are no critical weak points taking in consideration the period of time given for its development. Although there is a thing which might be not so good, here I am referring to the UI. It is user friendly but the style can be improved.

References

[1] https://www.geeksforgeeks.org/design-patterns-set-1-introduction/

[2] https://refactoring.guru/design-patterns

[3] https://sourcemaking.com/design_patterns

[4] https://www.tutorialspoint.com/design_pattern/

[5] https://yuml.me/diagram/scruffy/class/draw

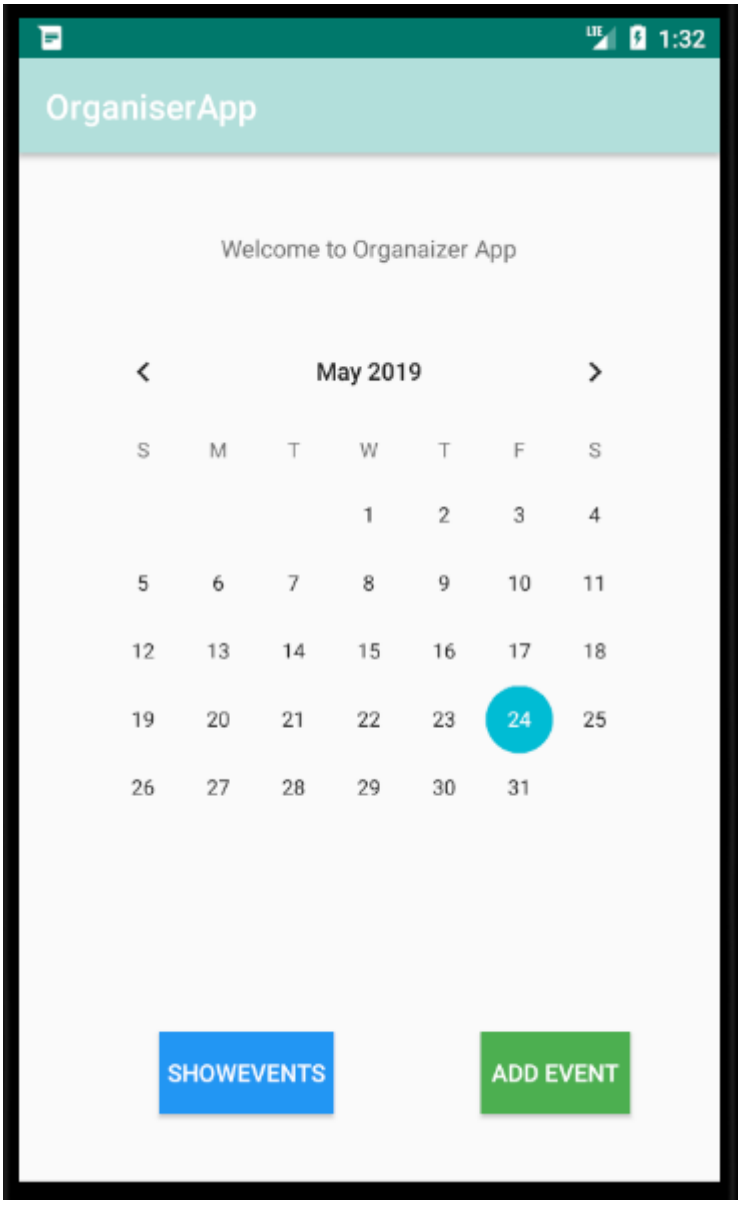
[6] https://www.rishabhsoft.com/blog/5-advantages-of-android-app-development-for-your-business

[7] https://www.tutorialspoint.com/uml/index.htm

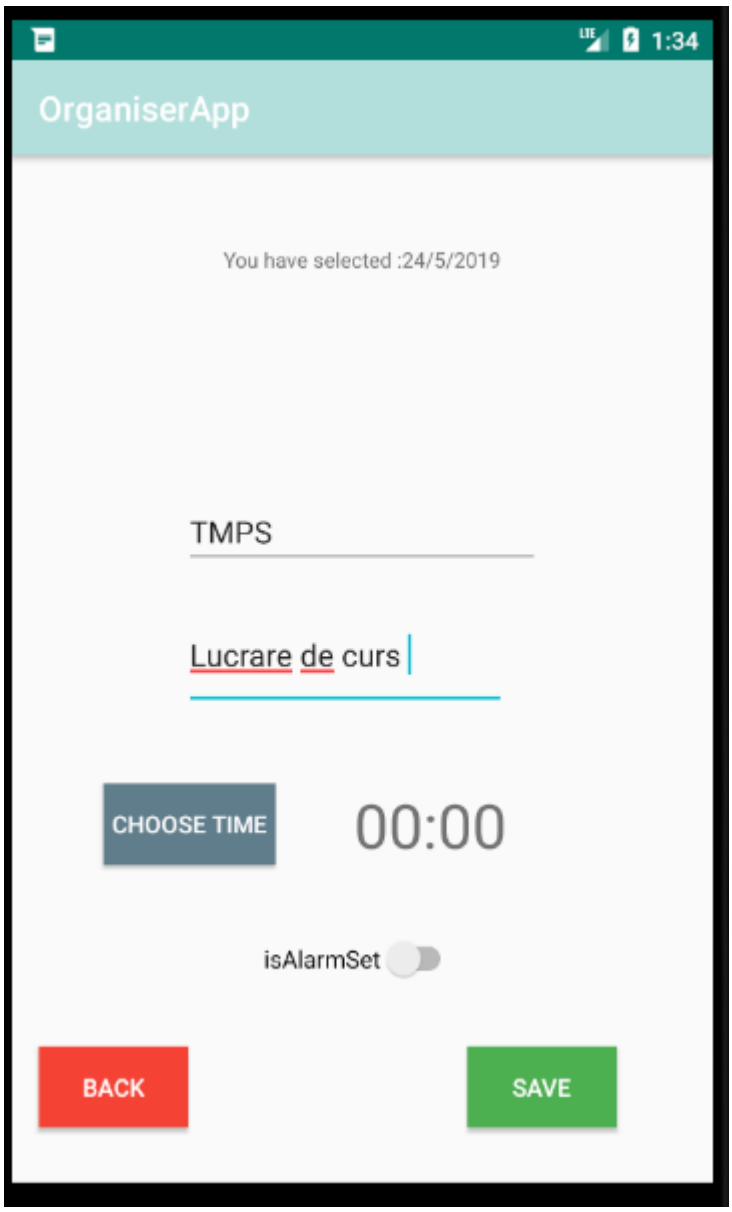
[8] https://www.draw.io/

[9] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissodes – Design Patterns

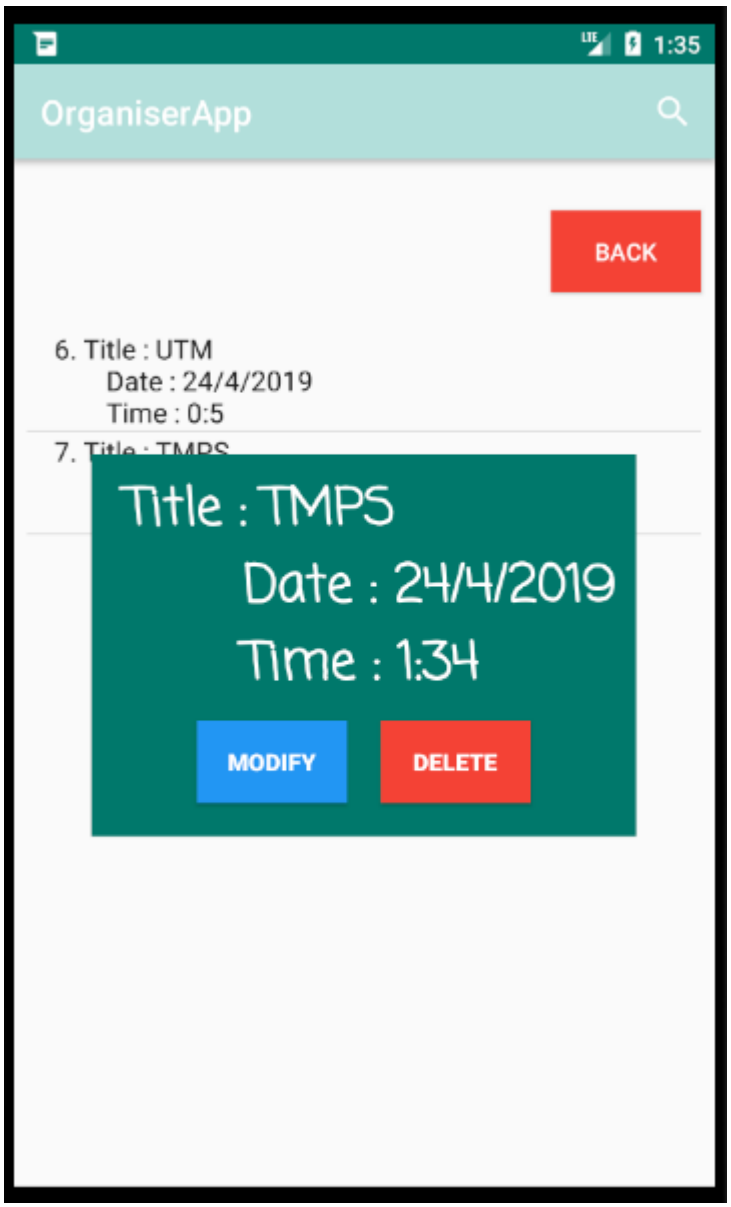
Annex



Annex 1: Events Calendar View



Annex 2: Create Event View



Annex 3: Events List View, Manage Event View