CS4227 Group Project

April 2022

# Reservify

## Lua Lipa

|  |  |
|---|---|
| Ranya El-Hwigi | 18227449 |
| John O'Keeffe | 18249531 |
| Nutsa Chichilidze | 18131856 |
| Erona Aliu | 18228402 |

Reservify GitHub Repository

# Table of contents

# 1. Requirements

## 1.1 Scenario

We have been commissioned by a client to develop a reservation system, Reservify, that can be used to support multiple industries across different sectors. Reserving a book at a library, a hotel room at a hotel or a gym session at a gym are all fundamentally handled the same way by a system, so why develop frameworks that are specific to one sector and limit the number of users of the framework when it can be made to support reservations in general. Using Reservify business can then set parameters that are specific to them such as what information is required, etc.

Minimum requirements of Reservify:

- Support basic services of reservations such as creating a reservation and deleting a reservation.
- Allow users to amend a reservation while they're making it.
- Facilitate transparent extension.
- Allow business to make Reservify more specific to them by setting parameters.
- Allow users to make reservations.

We can see this functionality clearly in the use case diagram depicted in *Figure 1.1.*

## 1.2 Use case diagram

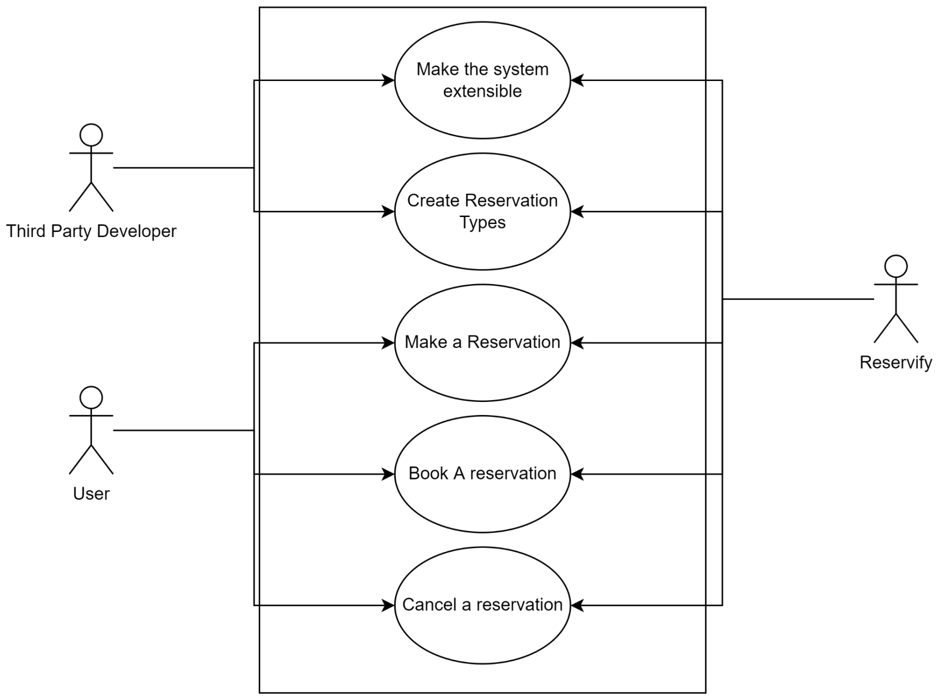


Figure 1.1: use case diagram

### 1.2.1 Use Case Descriptions

#### 1.2.1.1 Make the system extensible

The system should be extensible so that it can be used by developers in order to create their own reservation system tailored to their needs while Reservify provides the foundational functionality e.g. gym reservation.

#### 1.2.1.2 Create Reservation Types

Third party developers must be able to create different types of Reservations i.e. a gym can create a Gym Class reservation or a Pool session reservation, and specify the requirements needed to make that reservation.

#### 1.2.1.3 Make a Reservation

Users must be able to fill out details for a given reservation created by the third party developer.

#### 1.2.1.4 Book a Reservation

Users must be able to book a reservation with the system in order to confirm the reservation.

#### 1.2.1.5 Cancel a Reservation

Users must be able to cancel a reservation to notify the system that they cannot attend.

## 1.3 Quality attributes

To ensure the creation of a high quality system, we focused on setting and upkeeping the following quality attributes:

### 1.3.1. Security

Security is of utmost importance to us due to the fact that our system is designed to interact with customers while handling sensitive information. We adhered to security guidelines and ensure that we avoid security breaches such as:

- **Sensitive Data Leakage**: by the use of encryption tools to ensure that all sensitive information is safe against a data breach.
- **Insufficient Monitoring**: by the use of the Interceptor design pattern, which produces continuous logs throughout the system's lifetime. The logs are stored and can be used to trace back any errors and flaws within the system.
- **Vulnerable Components**: we ensure that any external APIs or Frameworks we use are continuously scanned for security metrics.
- **Code Base Flaws**: We use an automated security scan to catch and solve coding flaws throughout the development process such as code complexity, code duplication, code style, performance, error proneness and unused code.

### 1.3.2. Extensibility

We ensure that our system is extensible and can be easily used and modified by our clients by adhering to:

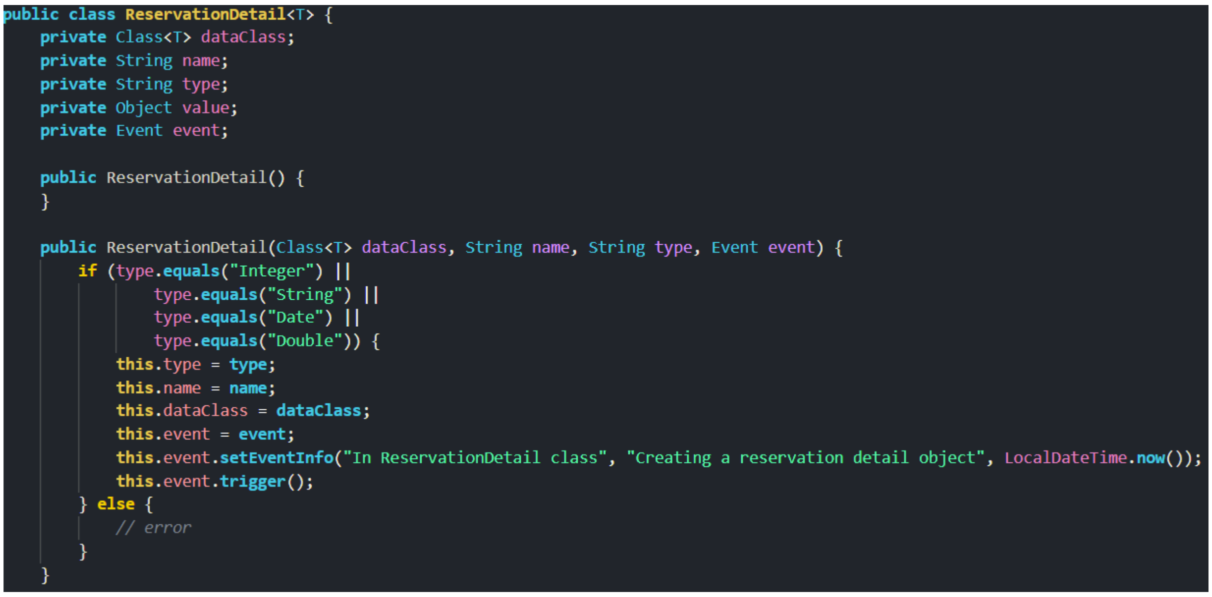
- Keeping the system architecture components loosely coupled with each other: ensuring that no components are strongly associated with another where there is no need for such relationships.
- Keeping the system modular: Using modules which allow us to separate the system functionalities into concrete modules which can be replicated or reused.

### 1.3.3. Reliability

The reliability of the system is ensured by the number of automated tools we have built within our development process. The automated code metrics analysis, code security scans, unit testing ensure our clients that our system is reliable and any small errors during development can be caught, assessed and fixed before the system is readily available for users to interact with. The use of frequent logging tools and monitoring also keep our system transparent and reliable.

### 1.3.4 SOLID

1. **Single Responsibility Principle:** We made sure that our classes were kept short and easy to read and our methods were responsible for one task only as seen in *Figure 1.2*. Here we can see the Reservation Details class which contains exactly what we expect: details of a reservation.



```java
public class ReservationDetail<T> {
    private Class<T> dataClass;
    private String name;
    private String type;
    private Object value;
    private Event event;

    public ReservationDetail() {
    }

    public ReservationDetail(Class<T> dataClass, String name, String type, Event event) {
        if (type.equals("Integer") ||
                type.equals("String") ||
                type.equals("Date") ||
                type.equals("Double")) {
            this.type = type;
            this.name = name;
            this.dataClass = dataClass;
            this.event = event;
            this.event.setEventInfo("In ReservationDetail class", "Creating a reservation detail object", LocalDateTime.now());
            this.event.trigger();
        } else {
            // error
        }
    }
```

Figure 1.2: reservation details SRP

2. **Open Closed Principle:** The factory pattern allows the system to be closed for modification but it can extend and create new products and concrete factories so we can depend on abstraction and not on concrete implementations as seen in Chapter 5.6. The Prototype pattern also contributes to this.

3. **Liskov Substitution Principle:** Defines that objects of a superclass should be replaceable with objects of its subclasses without breaking the application. This can be seen in Booking where we can have Early Booking functioning without Booking itself as Booking separates the implementation from Reservation. The Early Booking behaves the same way as the Booking.

4. **Interface Segregation Principle:** States that a client should not be exposed to methods it does not need. We use several interfaces throughout Reservify in order to keep users away from methods that are unnecessary for their development. This is where the Facade pattern comes in. It abstracts the user entirely from the methods in the class e.g. `createReservation(Booking booking, String name, Double price), initInterceptor()` etc.

5. **Dependency Inversion Principle:** Tells us that high-level modules should not depend on low-level modules and should depend on abstractions instead. Since we use the Three Tier Architecture, the Data is not allowed to make upward calls to Logic (layer above it), if it did it should not expect an answer from it, and the Business logic is not allowed to make upward calls to the Application (see Chapter 3.1).
   High level modules in our Logic layer, are easily reusable and unaffected by changes in low level modules such as the ones in our Data layer. The same goes for modules in the Application not being immensely affected by the Logic layer [4]. The modules do not make calls to modules in layers above it, only vice versa, unless there is some error-handling (see *Figure 1.3*).

### 1.3.5 The Law of Demeter

This law states that a module should not know about the innards of the objects it manipulates and it should only communicate with close classes (avoid long call chains). We consciously avoided these long chain calls throughout the project as it is unpleasant to retrieve a value as:

```
if(reservation.getReservationID().getUserID().getUserName());
```

Our method calls are kept short to avoid this issue for less dependencies throughout the code.

## 1.4 Tactics for supporting architectural use cases

Based on Bass et. al's “insert book”, an architectural pattern is discussed to have a relationship between a context, problem, solution.

1. **Context:** A recurring problem → setting up a reservation from scratch. The system needs to be clear and well documented so modules can be maintained and developed independently without affecting the rest of the system.

2. **Problem:** The software must be designed so that the modules can evolve and be developed independently with little interaction (loosely coupled, high cohesion).
3. **Solution:** In order to separate each of the concerns, we can use the layered pattern that can be seen in more detail in Chapter 3.1. This will allow us to have layers where modules are grouped in and offer a cohesive set of services e.g. createReservation, cancelReservation etc. in the Business Logic layer. The relations between the layers are unidirectional. Each layer is not able to use the preceding layer above it.

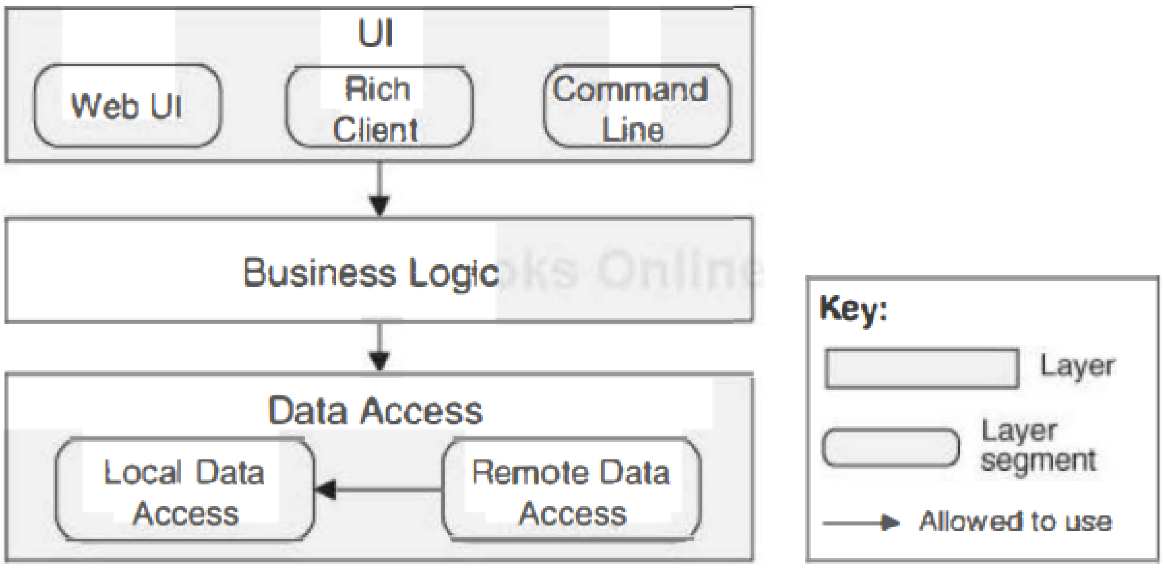


Figure 1.3: layered design with segmented layers [4]

# 2. Design patterns

## 2.1 Interceptor pattern

The interceptor is a behavioural design pattern that is used to intercept a request made to a system by allowing services to be added transparently to the framework and then triggering those services automatically on the occurrence of particular events. What's cool about the interceptor is that once we have it set up and built into our framework it allows us to offer our clients a way to change our processing cycle to best suit their needs, while maintaining encapsulation and not giving them access to our source code.
The interceptor is incorporated into the system to facilitate the requirement of transparent extension as the dispatcher and context object will allow clients to create and register their own concrete interceptors.
The interceptor is also implemented to facilitate a logging feature which will intercept between states and output to the console information on where in the code the execution currently is.

## 2.2 Bridge pattern

The Bridge is a structural design pattern that allows decoupling of classes within the code thus falling in line with SOLID principles. It does this through separating an abstraction from its implementation so the two can vary independently.
After a user makes a reservation, a booking will be generated along with the reservation as a confirmation mechanism for the owner. It will use the information given during the reservation and pass it in a database as a Booking object.

## 2.3 Command pattern

The Command pattern is a behavioural pattern which is used for encapsulating a request as an object and is used for providing clients with specific parameterised requests that they can adapt to their products.
The Command Pattern is used in the system to interact with a system user. Reservify has a list of Commands stored in its UIToolKit, such as MakeReservation, CancelReservation, ChangeReservation, each inheriting from the Command interface. These commands ensure the execution of user requests by interacting with other key classes in the system, such as concrete interceptors, memento objects and more.
The Command pattern is also frequently used for implementing Undo operations, and Reservify also implements an UndoCommand, which supports undoing any previous Commands left by a user.

## 2.4 Factory pattern

The factory pattern is a creational design pattern which acts as an interface for creating objects. We use the factory pattern in the Reservation package. Our factory pattern looks different to traditional factory patterns, as there are no objects that we can create when we begin. The third party developer creates their own reservation object like a gym class or pool session. They use the createReservation method from Reservify which creates a new object and adds it to the reservation factory. This adds the object to an array list of reservations. To instantiate a new object from the factory, the third party developer can select a reservation they want to create by indexing it through a number or by name.

## 2.5 Memento

The Memento is a behavioural design pattern that is used to capture and externalise an object's state so that it can later be restored. What's cool about Memento is that it doesn't violate encapsulation while doing this.
Memento is implemented to support the requirement to allow users to amend their reservation details while making a reservation. A reservation is made up of various details such as first name, last name, etc. As the user enters a detail in the terminal and moves on to the next, the memento implementation allows the user to return to the previous detail and change it.

## 2.6 Visitor

The main purpose of the Visitor is to allow the definition of a new operation without altering an existing object structure. For example, we have implemented a CurrencyConverterVisitor which calls a currency converter API in order to convert the current Euro currency to Dollars or Pounds. This can be used during the reservation in order to give users the option to convert the price to another currency.
As well as providing extra functionality in the system, the Visitor issues respect for OCP, thus keeping the system loosely coupled. It allows for clients of the system to add additional methods to all of the classes since we do not want to implement the logic in each class as that would explode the common logic in multiple classes and in multiple places.

## 2.7 Independently Researched Patterns

### 2.7.1 Prototype

The prototype pattern is a creational design pattern that lets you make an exact copy of an object. If you have an object that you want to copy exactly, it is difficult as you have to go through all the fields recursively of the original object, and copy the values. Furthermore fields may not be visible. The prototype pattern supports deep cloning in the class that wants to support being copied by implementing the cloneable interface. This is used in our project as when the third party developer creates a reservation type, we want to make copies of this object with the inputted user details saved. This means to create the object we simply call the clone method of the class, otherwise we have to instantiate the reservation object again, we lose the values stored in the previous object as we have created a new object, and have to set those values again in the new object. The prototype pattern is also used with the Memento design pattern. To store the current state of the reservation object we simply have to call the clone method and store it in the memento class. This allows for the saving and restoring of states from reservation to be implemented very neatly.

### 2.7.2 Facade

The Facade pattern is a structural design pattern implemented in the Reservify class. This class abstracts the functionality of the backend system, and only exposes methods that the developer needs to utilise the system. For example when we create a reservation we have to create the reservation object and then add it to the reservation factory. The third party developer doesn’t need to know about this. All they need to know is how to create a reservation. So we abstract those two lines of code into a method for the developer to consume.

# 3. System architecture

## 3.1 Three Tier Architecture

The three tier architecture is a software application architecture that organises applications into three logical and physical computation tiers where we have the following tiers (see *Figure 3.1*):

1. *Presentation:* User Interface
2. *Logic:* Where data is processed
3. *Database:* Where data associated with the application is stored and managed.

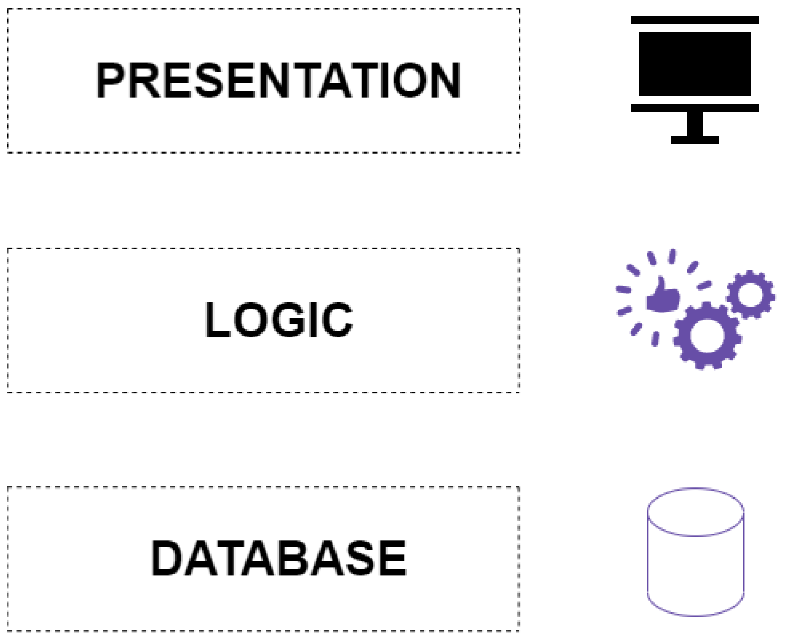


Figure 3.1: three tier architecture

We can see our system architecture including the classes split into these three tiers in *Figure 3.2* below. In this case, since App3 is public, the packages imported to it will be separated from the private logic in the rest of the classes in the logic layer.

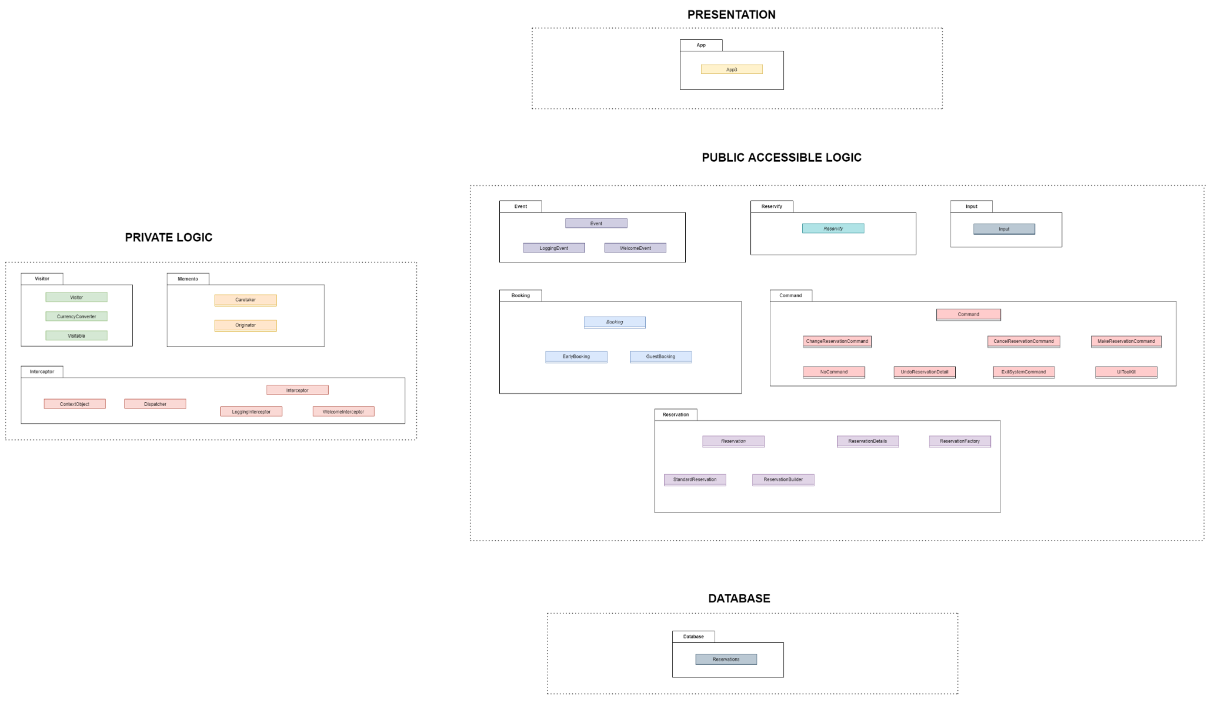


Figure 3.2: three tier architecture for reservify

# 4. Structural and behavioural diagrams

## 4.1 Class diagram

A class diagram gives us a blueprint-like overview of Reservify during the system design process. It allows us to show the objects that make up the system as well as the relationships between the objects and describe what they do and what services they provide as seen in *Figure 4.1*.

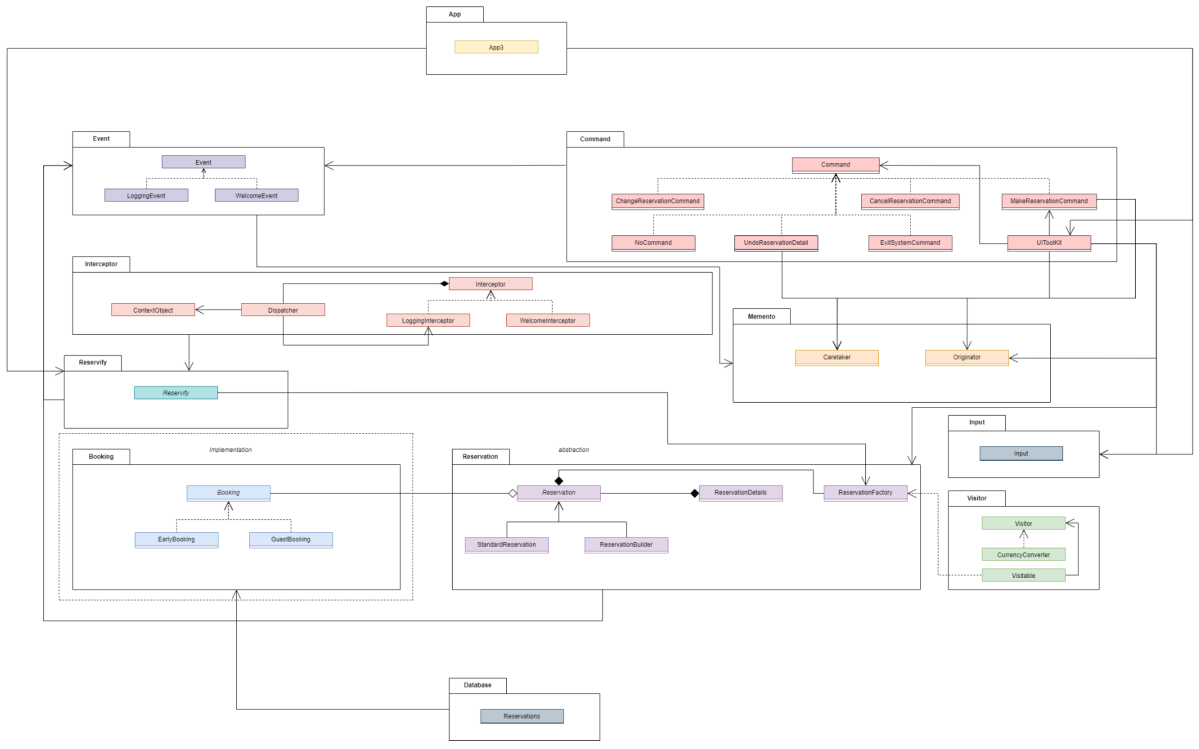


Figure 4.1: class diagram for reservify

Due to the complexity of the code, the methods are not included in this class diagram as it would be difficult to view.

## 4.2 Sequence diagram

For the sequence diagram we decided to diagram the interceptor execution and demonstrate the chain of responsibility that we have implemented in our dispatcher.

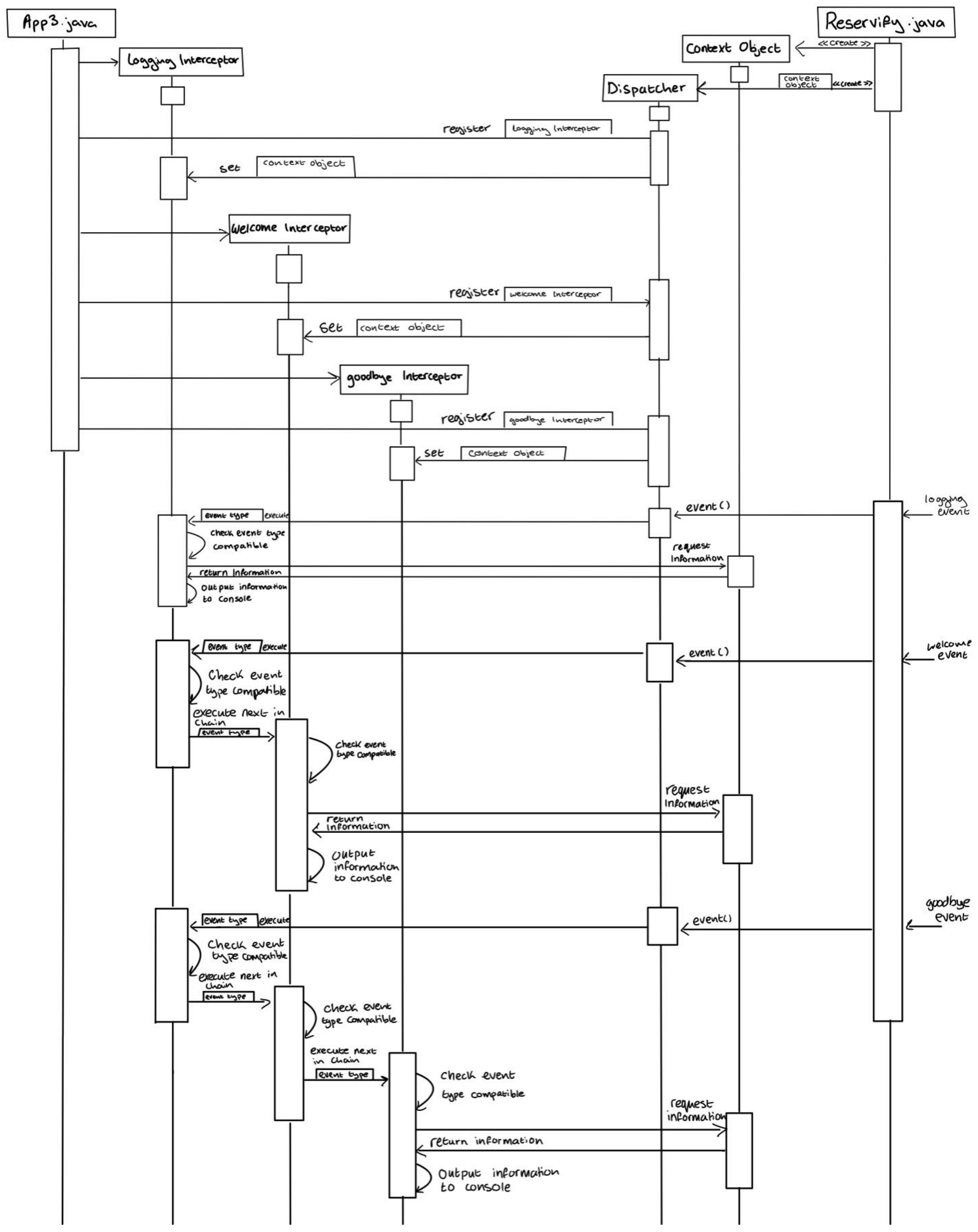


Figure 4.2: sequence diagram

# 5. Code fragments

## 5.1 Interceptor Implementation

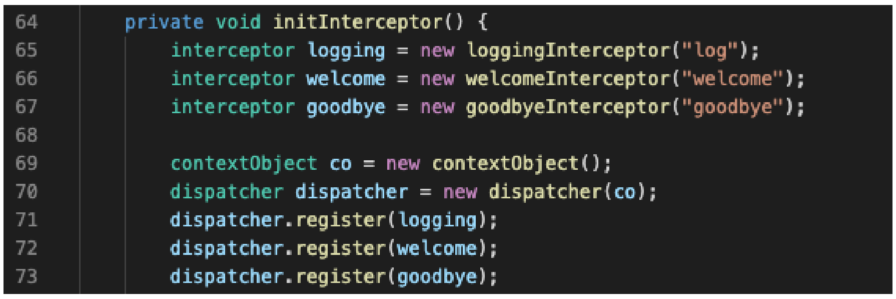
Two aspects of our implementation of the interceptor that we believe to be interesting are the process of the interceptor registration and our use of the Chain of Responsibility pattern in the dispatcher.

### 5.1.1 Interceptor Registration

During the process of designing our interceptor implementation we were faced with two options that would dictate aspects of the registration process:

- Pass the context object per registration.
- Pass the context object per event.

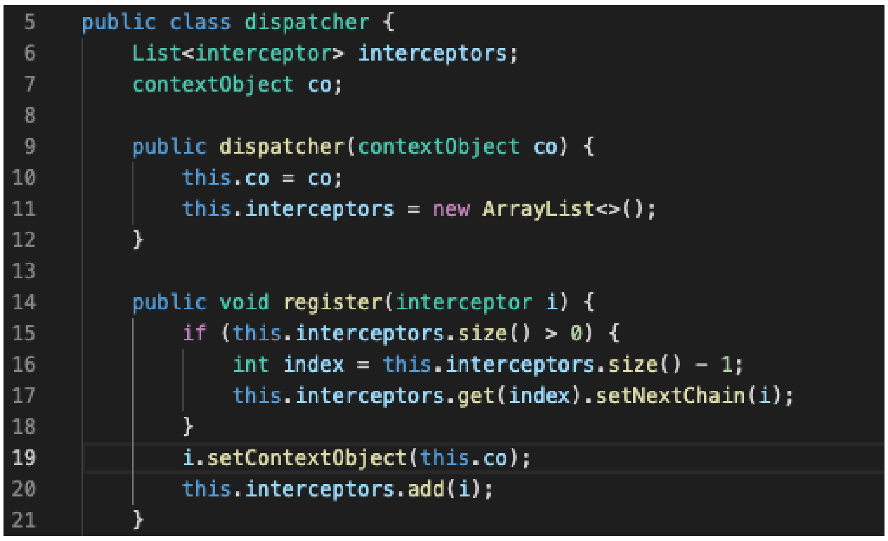
We decided to pass the context object per registration as we had only one context object interface and there is less overhead that way.



```java
    private void initInterceptor() {
        interceptor logging = new loggingInterceptor("log");
        interceptor welcome = new welcomeInterceptor("welcome");
        interceptor goodbye = new goodbyeInterceptor("goodbye");

        contextObject co = new contextObject();
        dispatcher dispatcher = new dispatcher(co);
        dispatcher.register(logging);
        dispatcher.register(welcome);
        dispatcher.register(goodbye);
```

Figure 5.1: initInterceptor method

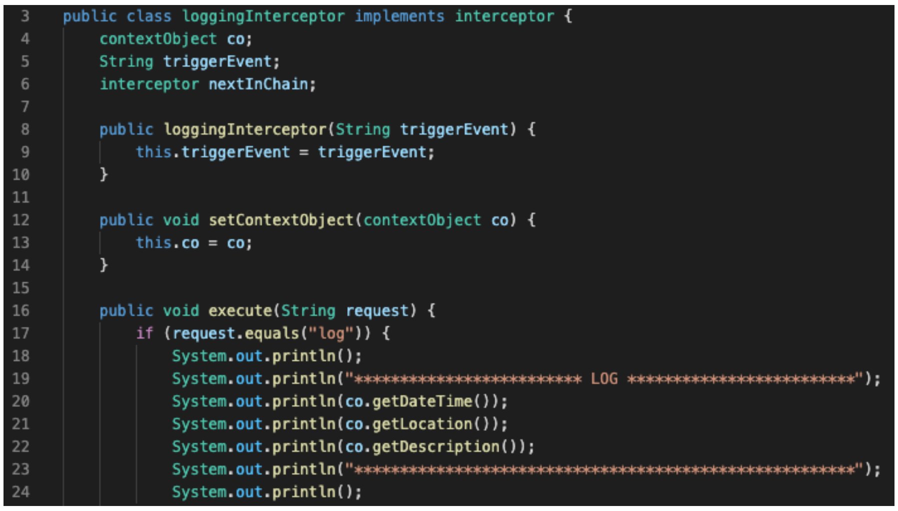
In the *Figure 5.1* above we can see the concrete interceptors, the context object, and the dispatcher getting instantiated. Notice how the context object is passed a parameter to the dispatcher constructor. The register method of the dispatcher is then executed and accepts the instance of the concrete interceptor as a parameter.



```java
public class dispatcher {
    List<interceptor> interceptors;
    contextObject co;

    public dispatcher(contextObject co) {
        this.co = co;
        this.interceptors = new ArrayList<>();
    }

    public void register(interceptor i) {
        if (this.interceptors.size() > 0) {
            int index = this.interceptors.size() - 1;
            this.interceptors.get(index).setNextChain(i);
        }
        i.setContextObject(this.co);
        this.interceptors.add(i);
    }
```

Figure 5.2: dispatcher

As can be seen in *Figure 5.2* above, the dispatcher accepts the context object and saves the instance in the variable “co”. Once the register method is executed, it first checks to see if we have already registered an interceptor before this one (this is to facilitate the implementation of Chain of Responsibility which is discussed in the section 5.1.2 below). Second it sets the context object of the concrete interceptor to the instance saved in the dispatcher. This is the context object the concrete interceptor will use going forward (as can be seen in *Figure 5.3* below). Finally it appends the concrete interceptor to the list of interceptors.



```java
public class loggingInterceptor implements interceptor {
    contextObject co;
    String triggerEvent;
    interceptor nextInChain;

    public loggingInterceptor(String triggerEvent) {
        this.triggerEvent = triggerEvent;
    }

    public void setContextObject(contextObject co) {
        this.co = co;
    }

    public void execute(String request) {
        if (request.equals("log")) {
            System.out.println();
            System.out.println("************************* LOG *************************");
            System.out.println(co.getDateTime());
            System.out.println(co.getLocation());
            System.out.println(co.getDescription());
            System.out.println("*******************************************************");
            System.out.println();
```

Figure 5.3: example of a concrete interceptor

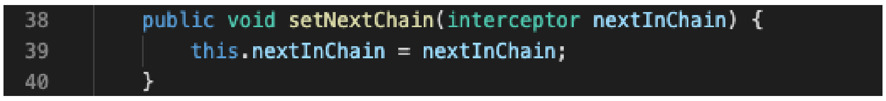
## 5.1.2 Chain of Responsibility

Another design decision of the interceptor pattern is how the dispatcher is going to choose the interceptor to execute in response to an event. Some choices are:

- Take a FIFO or LIFO approach.
- Associate a priority with each concrete interceptor.
- Implement a Chain of Responsibility.

We decided to go with Chain of Responsibility which is a behavioural design pattern that lets you pass requests along a chain of handlers[1].

In *Figure 5.2* above you can see the process of setting the next in chain of a concrete interceptor upon registration. First we check to see if this is the first interceptor we are registering in which case it wouldn’t be the next in chain for any interceptor. If it's not the first interceptor we get the last interceptor in the list, set this new interceptor as its nextInChain (method can be seen in *Figure 5.4*), and then finally add this new interceptor to the list.

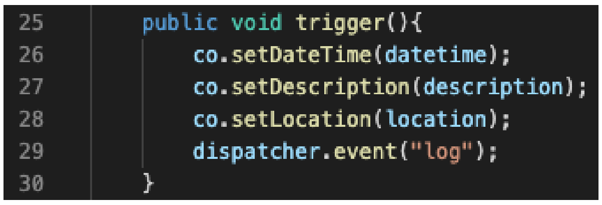


```java
    public void setNextChain(interceptor nextInChain) {
        this.nextInChain = nextInChain;
    }
```

Figure 5.4: implementation of setting the next in chain interceptor

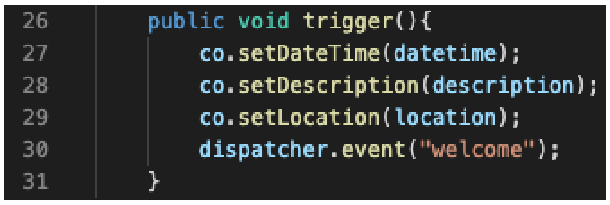
When an event is triggered the dispatcher executes the first interceptor in the list, that interceptor then checks if it can deal with the event and if it can’t it executes the interceptor next chain, and so on until an interceptor can deal with the event. A string called the “trigger” is used to allow interceptors to determine whether they can deal with this event or not.

Coding fragments of these implementations can be seen in *Figures 5.5, 5.6, 5.7, 5.8, 5.9, 5.10,* and *5.11.*



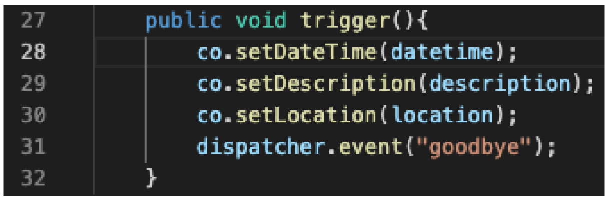
```java
public void trigger(){
    co.setDateTime(datetime);
    co.setDescription(description);
    co.setLocation(location);
    dispatcher.event("log");
}
```

Figure 5.5: example trigger method for log event



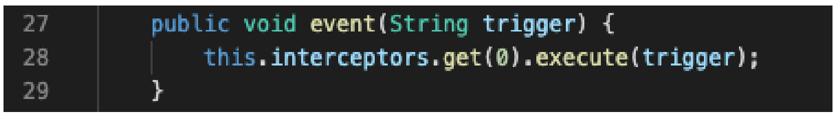
```java
public void trigger(){
    co.setDateTime(datetime);
    co.setDescription(description);
    co.setLocation(location);
    dispatcher.event("welcome");
}
```

Figure 5.6: example trigger method for welcome event



```java
public void trigger(){
    co.setDateTime(datetime);
    co.setDescription(description);
    co.setLocation(location);
    dispatcher.event("goodbye");
}
```

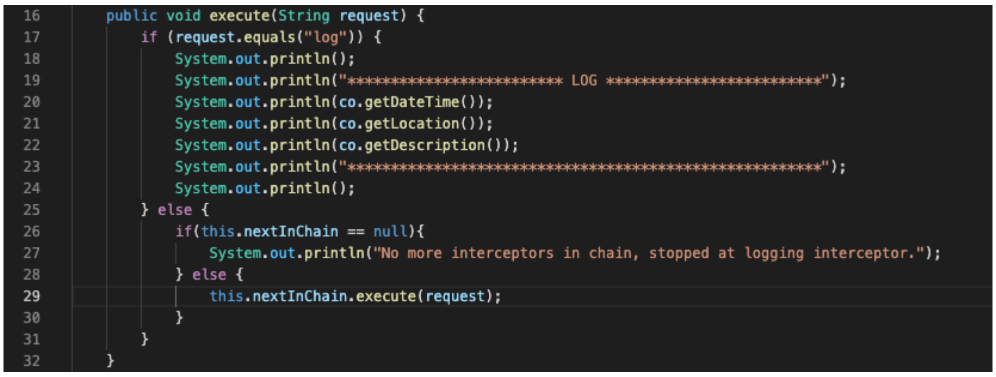
Figure 5.7: example trigger method for goodbye event



```java
public void event(String trigger) {
    this.interceptors.get(0).execute(trigger);
}
```

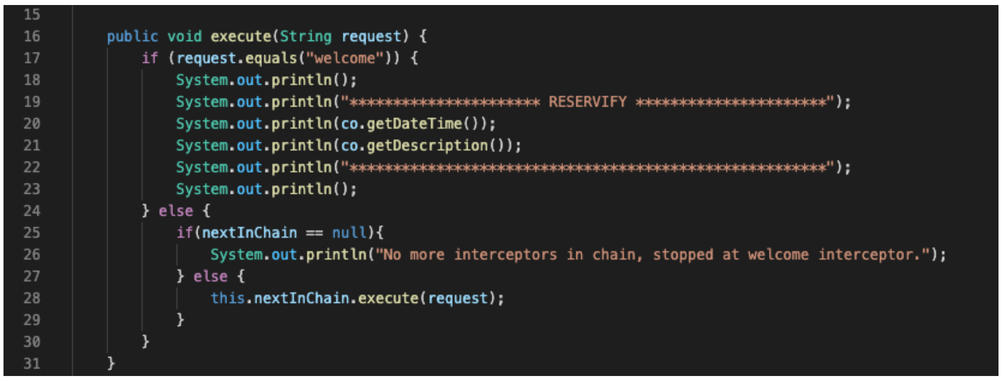
Figure 5.8: event method triggering first interceptor in list

To show the implementation of the chain of responsibility we created three interceptors: logging, welcome, and goodbye. Logging is executed throughout various parts of the code to log information to the console, welcome is executed at the start to welcome the user, and goodbye is executed when the user is exiting the system. Welcome and goodbye were implemented purely for demonstrative purposes. Their implementation and example outputs can be seen in *Figures 5.9, 5.10, 5.11, 5.12, 5.13,* and *5.14*.



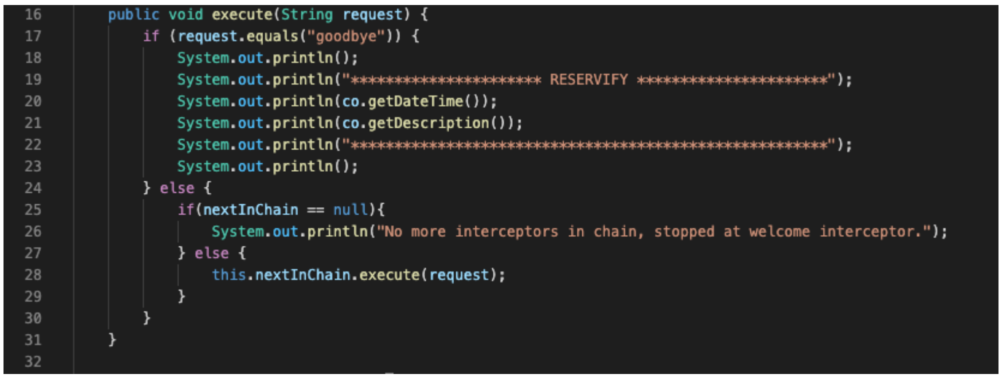
```java
public void execute(String request) {
    if (request.equals("log")) {
        System.out.println();
        System.out.println("************************* LOG *************************");
        System.out.println(co.getDateTime());
        System.out.println(co.getLocation());
        System.out.println(co.getDescription());
        System.out.println("*******************************************************");
        System.out.println();
    } else {
        if(this.nextInChain == null){
            System.out.println("No more interceptors in chain, stopped at logging interceptor.");
        } else {
            this.nextInChain.execute(request);
        }
    }
}
```

Figure 5.9: execute method of logging interceptor



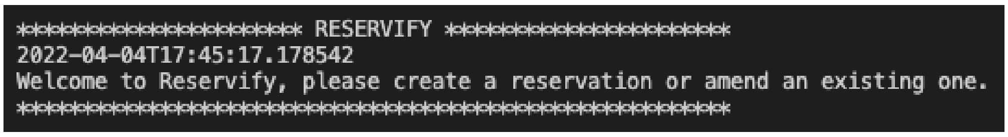
```
    public void execute(String request) {
        if (request.equals("welcome")) {
            System.out.println();
            System.out.println("********************** RESERVIFY **********************");
            System.out.println(co.getDateTime());
            System.out.println(co.getDescription());
            System.out.println("*******************************************************");
            System.out.println();
        } else {
            if(nextInChain == null){
                System.out.println("No more interceptors in chain, stopped at welcome interceptor.");
            } else {
                this.nextInChain.execute(request);
            }
        }
    }
```

Figure 5.10: execute method of welcome interceptor



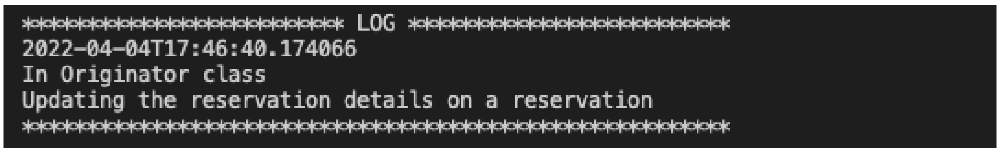
```
    public void execute(String request) {
        if (request.equals("goodbye")) {
            System.out.println();
            System.out.println("********************** RESERVIFY **********************");
            System.out.println(co.getDateTime());
            System.out.println(co.getDescription());
            System.out.println("*******************************************************");
            System.out.println();
        } else {
            if(nextInChain == null){
                System.out.println("No more interceptors in chain, stopped at welcome interceptor.");
            } else {
                this.nextInChain.execute(request);
            }
        }
    }
```

Figure 5.11: execute method of goodbye interceptor



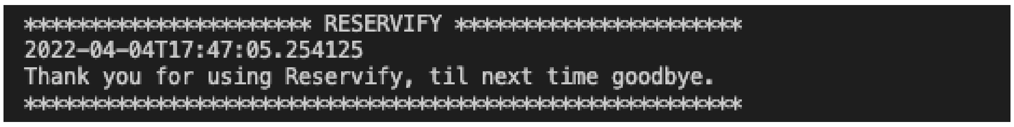
```
********************** RESERVIFY **********************
2022-04-04T17:45:17.178542
Welcome to Reservify, please create a reservation or amend an existing one.
*******************************************************
```

Figure 5.12: output of welcome interceptor



```
************************* LOG *************************
2022-04-04T17:46:40.174066
In Originator class
Updating the reservation details on a reservation
*******************************************************
```

Figure 5.13: output of logging interceptor



```
********************** RESERVIFY **********************
2022-04-04T17:47:05.254125
Thank you for using Reservify, til next time goodbye.
*******************************************************
```

Figure 5.14: output of goodbye interceptor

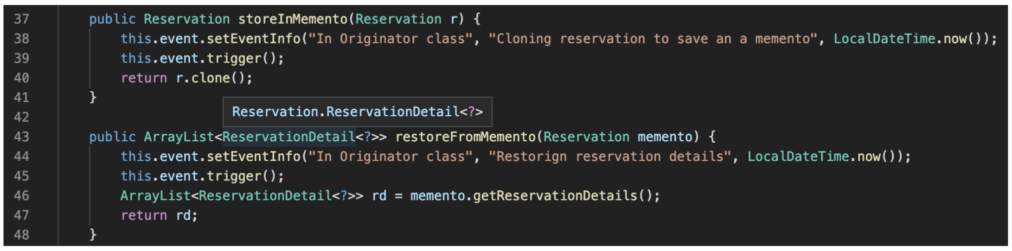
## 5.2 Memento Implementation

Three aspects we believe to be interesting in our Memento implementation are where we chose to use it, incorporating it into command, and making use of the prototype pattern inside memento.

### 5.2.1 Memento Usage

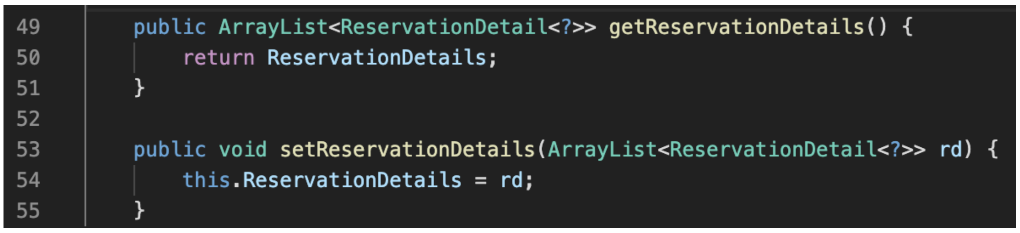
We chose to use the Memento design pattern for our reservation object when a user is making a reservation. When a user makes a reservation a number of reservation details are requested from the user such as first name, last name, age, etc. so we implemented the memento to keep track of reservation objects for each time a reservation detail is added to a reservation. This way we can allow the user to go back and change a reservation detail without having to start a new reservation.

We have a Memento interface which is a tag interface and is implemented by Reservation class. A caretaker to set and get memento. And an originator which contains and sets the reservation details. The implementation of these classes can be seen in the Figures below.



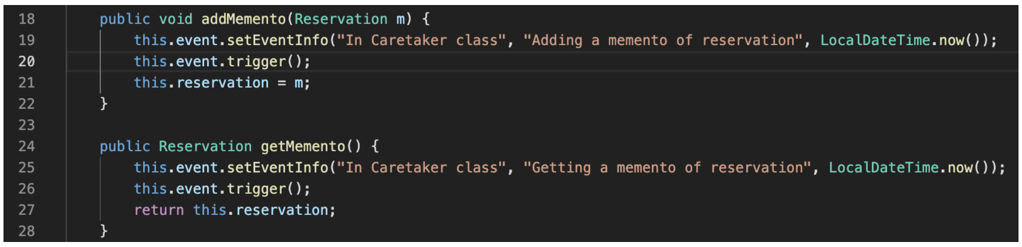
```
    public Reservation storeInMemento(Reservation r) {
        this.event.setEventInfo("In Originator class", "Cloning reservation to save an a memento", LocalDateTime.now());
        this.event.trigger();
        return r.clone();
    }
                              Reservation.ReservationDetail<?>
    public ArrayList<ReservationDetail<?>> restoreFromMemento(Reservation memento) {
        this.event.setEventInfo("In Originator class", "Restorign reservation details", LocalDateTime.now());
        this.event.trigger();
        ArrayList<ReservationDetail<?>> rd = memento.getReservationDetails();
        return rd;
    }
```

Figure 5.15: key code elements from originator class



```
    public ArrayList<ReservationDetail<?>> getReservationDetails() {
        return ReservationDetails;
    }

    public void setReservationDetails(ArrayList<ReservationDetail<?>> rd) {
        this.ReservationDetails = rd;
    }
```

Figure 5.16: key code elements from Reservation class which implements Momento

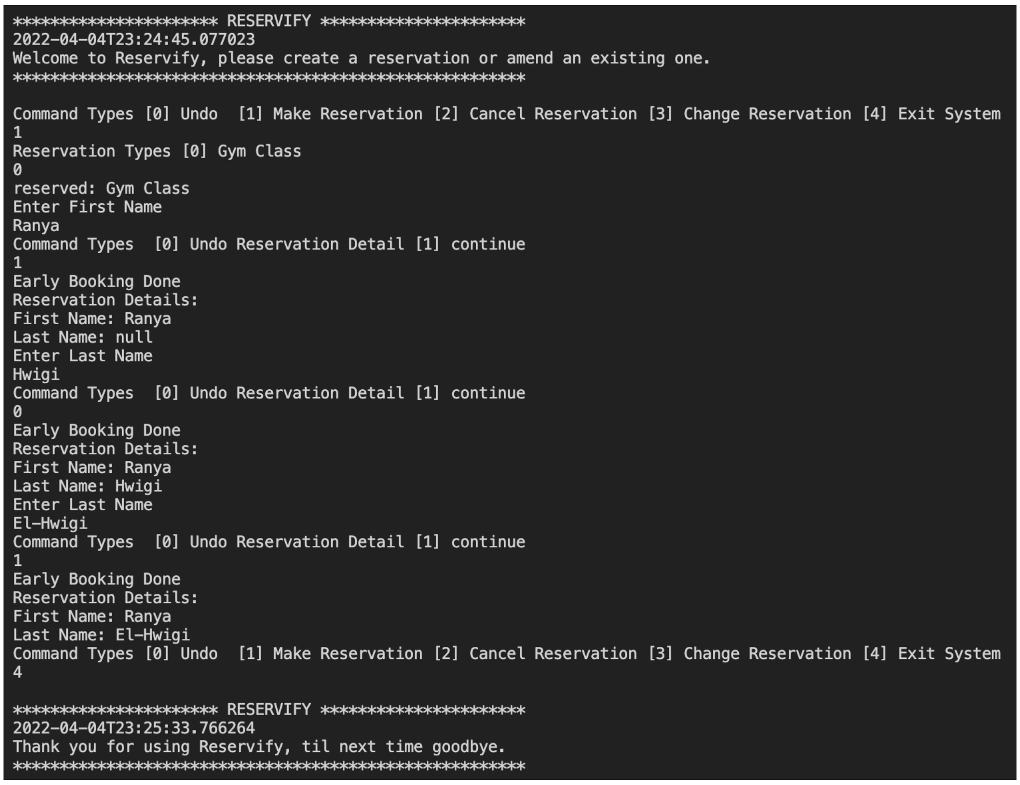


```
    public void addMemento(Reservation m) {
        this.event.setEventInfo("In Caretaker class", "Adding a memento of reservation", LocalDateTime.now());
        this.event.trigger();
        this.reservation = m;
    }

    public Reservation getMemento() {
        this.event.setEventInfo("In Caretaker class", "Getting a memento of reservation", LocalDateTime.now());
        this.event.trigger();
        return this.reservation;
    }
```

Figure 5.17: key code elements from caretaker class

### 5.2.2 Integration of Memento with Command

We implemented the memento within our implementation of the Command design pattern. Within the command package we implemented an UndoReservationDetail command and within MakeReservationCommand we use caretaker to update and store the memento. When a user first starts the system they are presented with multiple commands, among them is “Make Reservation”. Once they choose that command they are asked to enter some details. This is where the memento command comes in, once they enter a detail they are asked whether they would like “Undo reservation detail” or “Continue”. If they choose to “Undo reservation detail” we go restore the reservation object to the memento, If they choose to “Continue” we update the memento to include the detail added.

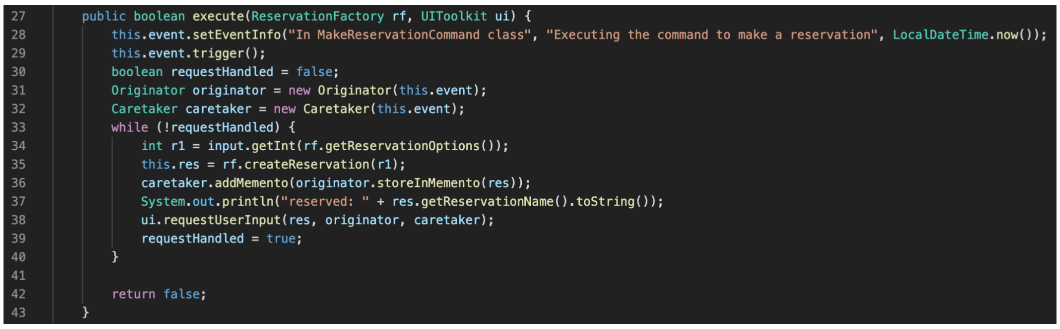


```
********************** RESERVIFY **********************
2022-04-04T23:24:45.077023
Welcome to Reservify, please create a reservation or amend an existing one.
*******************************************************

Command Types [0] Undo  [1] Make Reservation [2] Cancel Reservation [3] Change Reservation [4] Exit System
1
Reservation Types [0] Gym Class
0
reserved: Gym Class
Enter First Name
Ranya
Command Types  [0] Undo Reservation Detail [1] continue
1
Early Booking Done
Reservation Details:
First Name: Ranya
Last Name: null
Enter Last Name
Hwigi
Command Types  [0] Undo Reservation Detail [1] continue
0
Early Booking Done
Reservation Details:
First Name: Ranya
Last Name: Hwigi
Enter Last Name
El-Hwigi
Command Types  [0] Undo Reservation Detail [1] continue
1
Early Booking Done
Reservation Details:
First Name: Ranya
Last Name: El-Hwigi
Command Types [0] Undo  [1] Make Reservation [2] Cancel Reservation [3] Change Reservation [4] Exit System
4

********************** RESERVIFY **********************
2022-04-04T23:25:33.766264
Thank you for using Reservify, til next time goodbye.
*******************************************************
```

Figure 5.18: example process of undoing reservation details

The implementation of memento inside command goes as follows:

1. Once “Make Reservation” is chosen its execute method is executed. Inside the execute method we instantiate our caretaker, originator, and Reservation (our memento). We add a memento which is the reservation with no reservations details set yet.

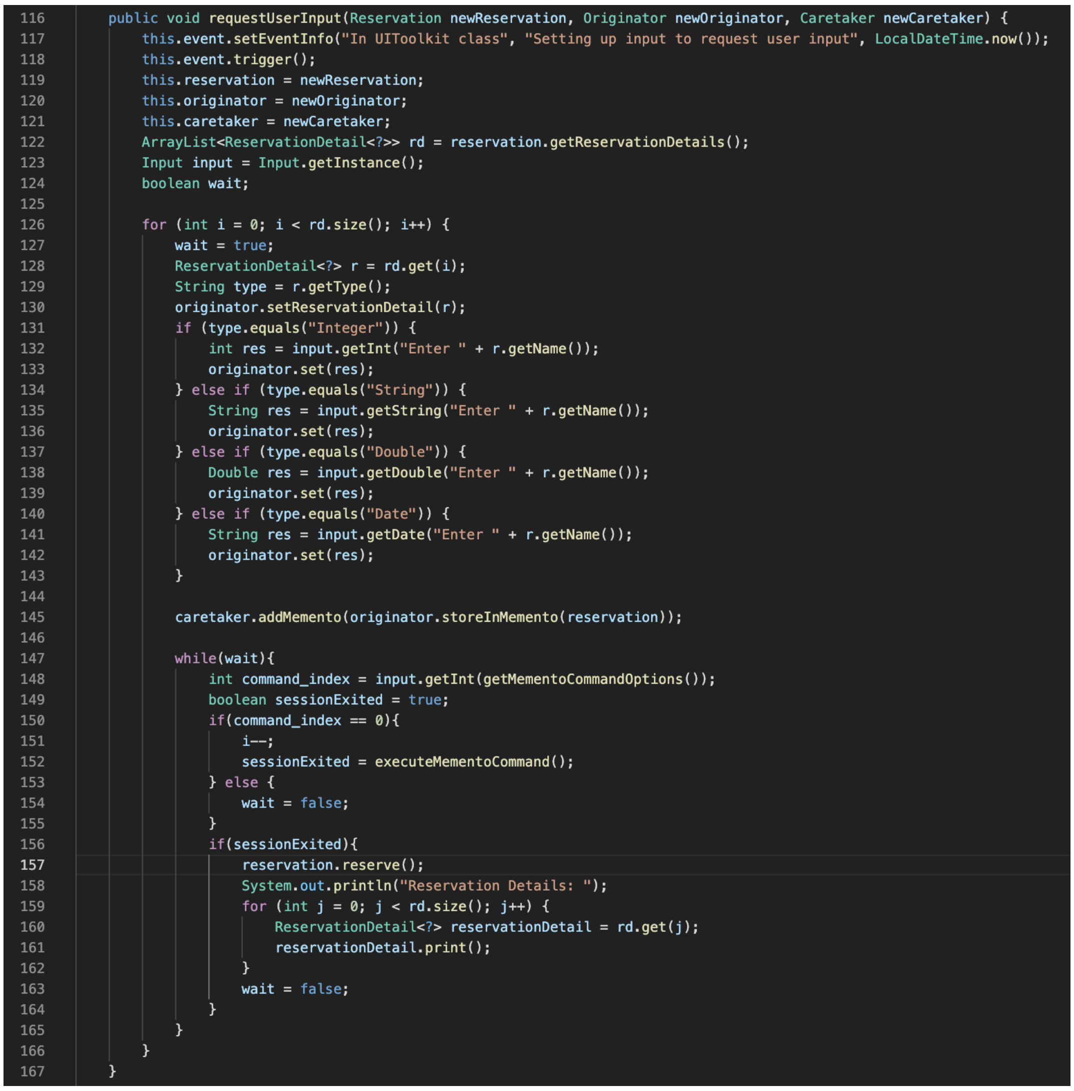


```java
    public boolean execute(ReservationFactory rf, UIToolkit ui) {
        this.event.setEventInfo("In MakeReservationCommand class", "Executing the command to make a reservation", LocalDateTime.now());
        this.event.trigger();
        boolean requestHandled = false;
        Originator originator = new Originator(this.event);
        Caretaker caretaker = new Caretaker(this.event);
        while (!requestHandled) {
            int r1 = input.getInt(rf.getReservationOptions());
            this.res = rf.createReservation(r1);
            caretaker.addMemento(originator.storeInMemento(res));
            System.out.println("reserved: " + res.getReservationName().toString());
            ui.requestUserInput(res, originator, caretaker);
            requestHandled = true;
        }

        return false;
    }
```

Figure 5.19: execute method of MakeReservationCommand.

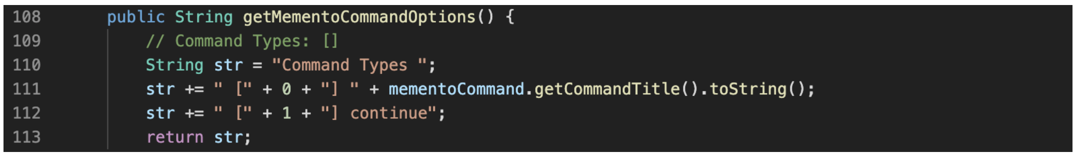
2. After a reservation is made we move on to requesting input from the user. At first, using a for loop to loop through the list of reservation details, the current reservation detail we’re requesting, for example first name, is retrieved. We set the originator to let it know this is the reservation detail we’re getting this time around and then once its input we set the value of the reservation detail in the originator. Next we use caretaker to set the memento which accepts a clone of the originator. Finally we enter a while loop which outputs to the user its current options “Undo reservation detail” to revert back and fill in that detail again, or “Continue” to move on to the next reservation detail. If the user chooses to undo the reservation detail we decrement

the for loop so we go back to the previous reservation detail and execute the memento command UndoReservationDetail.



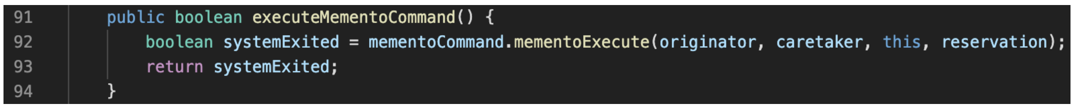
```java
public void requestUserInput(Reservation newReservation, Originator newOriginator, Caretaker newCaretaker) {
    this.event.setEventInfo("In UIToolkit class", "Setting up input to request user input", LocalDateTime.now());
    this.event.trigger();
    this.reservation = newReservation;
    this.originator = newOriginator;
    this.caretaker = newCaretaker;
    ArrayList<ReservationDetail<?>> rd = reservation.getReservationDetails();
    Input input = Input.getInstance();
    boolean wait;

    for (int i = 0; i < rd.size(); i++) {
        wait = true;
        ReservationDetail<?> r = rd.get(i);
        String type = r.getType();
        originator.setReservationDetail(r);
        if (type.equals("Integer")) {
            int res = input.getInt("Enter " + r.getName());
            originator.set(res);
        } else if (type.equals("String")) {
            String res = input.getString("Enter " + r.getName());
            originator.set(res);
        } else if (type.equals("Double")) {
            Double res = input.getDouble("Enter " + r.getName());
            originator.set(res);
        } else if (type.equals("Date")) {
            String res = input.getDate("Enter " + r.getName());
            originator.set(res);
        }

        caretaker.addMemento(originator.storeInMemento(reservation));

        while(wait){
            int command_index = input.getInt(getMementoCommandOptions());
            boolean sessionExited = true;
            if(command_index == 0){
                i--;
                sessionExited = executeMementoCommand();
            } else {
                wait = false;
            }
            if(sessionExited){
                reservation.reserve();
                System.out.println("Reservation Details: ");
                for (int j = 0; j < rd.size(); j++) {
                    ReservationDetail<?> reservationDetail = rd.get(j);
                    reservationDetail.print();
                }
                wait = false;
            }
        }
    }
}
```

Figure 5.20: requesting user input in UIToolkit class.



```java
public String getMementoCommandOptions() {
    // Command Types: []
    String str = "Command Types ";
    str += " [" + 0 + "] " + mementoCommand.getCommandTitle().toString();
    str += " [" + 1 + "] continue";
    return str;
```

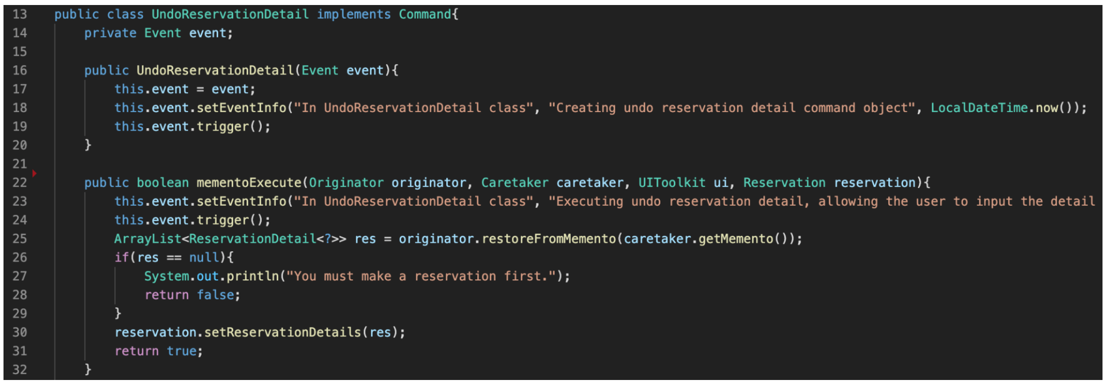
Figure 5.21: method for outputting memento command options.



```java
public boolean executeMementoCommand() {
    boolean systemExited = mementoCommand.mementoExecute(originator, caretaker, this, reservation);
    return systemExited;
}
```

Figure 5.22: method for executing memento command.

3. If the user would like to undo the reservation detail they have entered, inside the UndoReservationDetail class we restore from memento and reset the reservation details of the reservation to those of the memento.

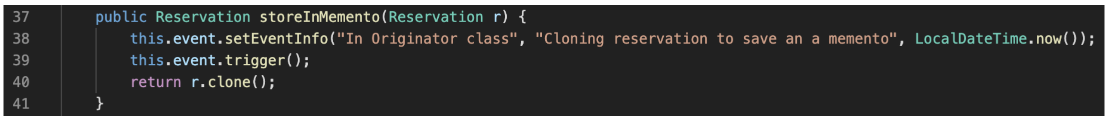


```java
public class UndoReservationDetail implements Command{
    private Event event;

    public UndoReservationDetail(Event event){
        this.event = event;
        this.event.setEventInfo("In UndoReservationDetail class", "Creating undo reservation detail command object", LocalDateTime.now());
        this.event.trigger();
    }

    public boolean mementoExecute(Originator originator, Caretaker caretaker, UIToolkit ui, Reservation reservation){
        this.event.setEventInfo("In UndoReservationDetail class", "Executing undo reservation detail, allowing the user to input the detail
        this.event.trigger();
        ArrayList<ReservationDetail<?>> res = originator.restoreFromMemento(caretaker.getMemento());
        if(res == null){
            System.out.println("You must make a reservation first.");
            return false;
        }
        reservation.setReservationDetails(res);
        return true;
    }
```

Figure 5.23: code fragments from UndoReservationDetail command class.

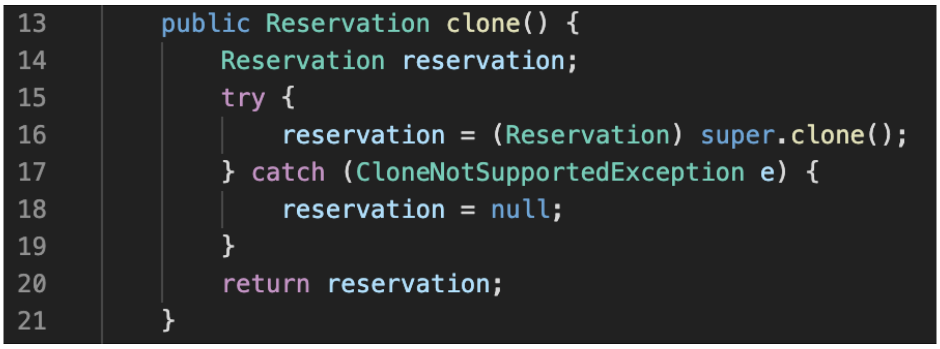
## 5.2.3 Integration of Memento with Prototype

Another design pattern we used alongside Memento was Prototype. As part of Memento we would be copying and resetting variables for the memento. Instead of getters and setters we decided to use Prototype and clone the object as the memento.



```java
    public Reservation storeInMemento(Reservation r) {
        this.event.setEventInfo("In Originator class", "Cloning reservation to save an a memento", LocalDateTime.now());
        this.event.trigger();
        return r.clone();
    }
```

Figure 5.24: use of Prototype clone inside of Originator.



```java
    public Reservation clone() {
        Reservation reservation;
        try {
            reservation = (Reservation) super.clone();
        } catch (CloneNotSupportedException e) {
            reservation = null;
        }
        return reservation;
    }
```

Figure 5.25: implementation of clone method.

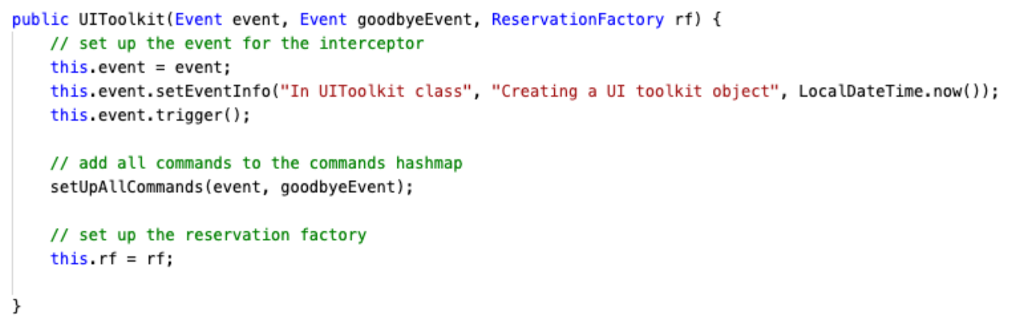
## 5.3. Command Implementation

The Command Pattern is used in the System to allow for user-system interaction. The main components of the Pattern implemented are Command and UIToolKit.

### 5.3.1. UI Tool Kit

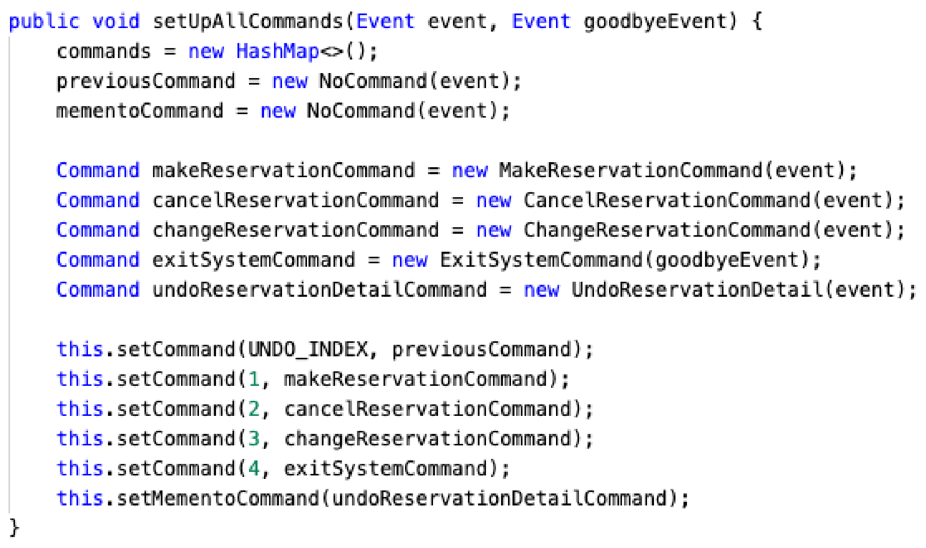
In order to ensure efficiency, all commands within the system should be stored in one place where they can be executed from. For this, the UIToolKit.java class was implemented.

The UIToolKit stores all commands in a HashMap, which maps each command to an integer. When a user requests a command to be executed, the matching slot of the command gets looked up in the UIToolKit's command map.



```java
public UIToolkit(Event event, Event goodbyeEvent, ReservationFactory rf) {
    // set up the event for the interceptor
    this.event = event;
    this.event.setEventInfo("In UIToolkit class", "Creating a UI toolkit object", LocalDateTime.now());
    this.event.trigger();

    // add all commands to the commands hashmap
    setUpAllCommands(event, goodbyeEvent);

    // set up the reservation factory
    this.rf = rf;

}
```

Figure 5.26: *the constructor of the ToolKit*



```java
public void setUpAllCommands(Event event, Event goodbyeEvent) {
    commands = new HashMap<>();
    previousCommand = new NoCommand(event);
    mementoCommand = new NoCommand(event);

    Command makeReservationCommand = new MakeReservationCommand(event);
    Command cancelReservationCommand = new CancelReservationCommand(event);
    Command changeReservationCommand = new ChangeReservationCommand(event);
    Command exitSystemCommand = new ExitSystemCommand(goodbyeEvent);
    Command undoReservationDetailCommand = new UndoReservationDetail(event);

    this.setCommand(UNDO_INDEX, previousCommand);
    this.setCommand(1, makeReservationCommand);
    this.setCommand(2, cancelReservationCommand);
    this.setCommand(3, changeReservationCommand);
    this.setCommand(4, exitSystemCommand);
    this.setMementoCommand(undoReservationDetailCommand);
}
```

Figure 5.27: *concrete commands being stored in the toolkit*

### 5.3.2. Undoing Commands

One of the strengths of the Command pattern is the ability to undo any previous commands, which the pattern does by always keeping track of the latest command.

The undo command gets mapped to the UIToolKit at a specific slot specified at the beginning of the class, 0 for the current case. At each execution request, the system checks whether the current slot maps to the undo button index and if it does, it calls to undo the previous command.

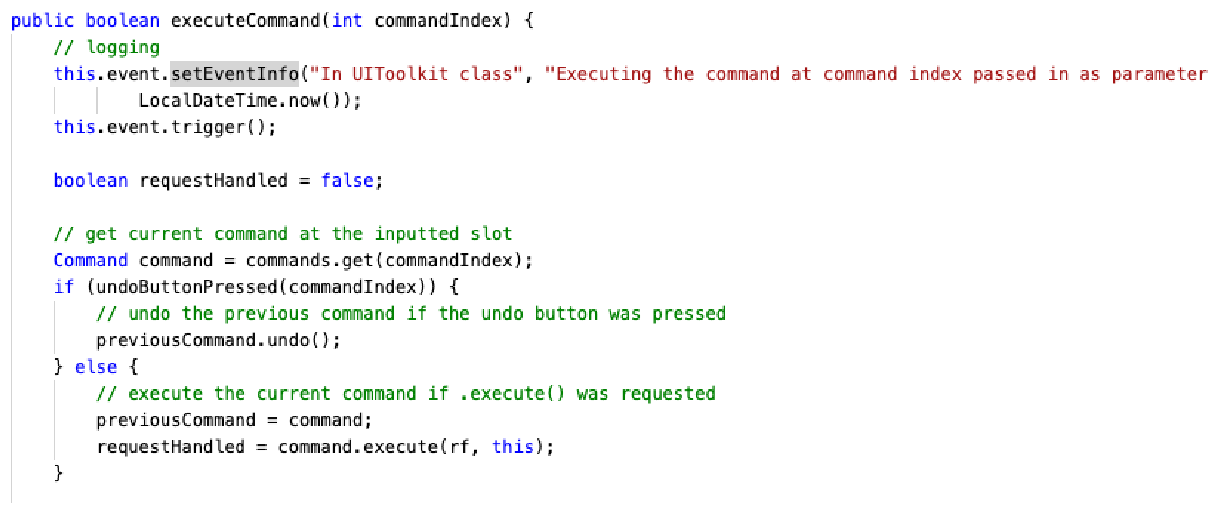

```java
public boolean executeCommand(int commandIndex) {
    // logging
    this.event.setEventInfo("In UIToolkit class", "Executing the command at command index passed in as parameter
            LocalDateTime.now());
    this.event.trigger();

    boolean requestHandled = false;

    // get current command at the inputted slot
    Command command = commands.get(commandIndex);
    if (undoButtonPressed(commandIndex)) {
        // undo the previous command if the undo button was pressed
        previousCommand.undo();
    } else {
        // execute the current command if .execute() was requested
        previousCommand = command;
        requestHandled = command.execute(rf, this);
    }
```

Figure 5.28: setting event info

When constructing the toolkit, the previousCommand gets set to a NoCommand object, which is an empty class that does nothing at each .execute() and .undo() call and acts only as a placeholder.

### 5.3.3. Command Classes

#### 5.3.3.1. The Command Interface

The command interface is the superclass for all the concrete commands and ensures that all concrete commands implement all the necessary methods such as .execute() and .undo().

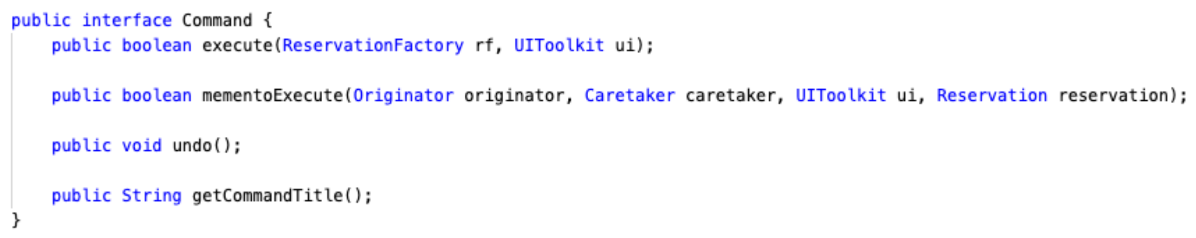
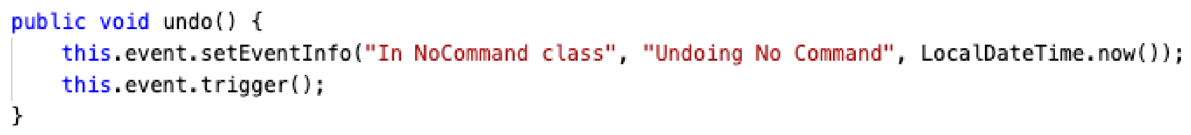

```java
public interface Command {
    public boolean execute(ReservationFactory rf, UIToolkit ui);

    public boolean mementoExecute(Originator originator, Caretaker caretaker, UIToolkit ui, Reservation reservation);

    public void undo();

    public String getCommandTitle();
}
```

Figure 5.29: execute and undo commands

### 5.3.3.1. NoCommand

No Command is a class that can be used anytime a mock command needs to be constructed or a slot in the command toolkit needs to be kept by a placeholder. This class was used to keep the place of the previousCommand when setting up a UIToolKit. The alternative here is to set the previousCommand to a null object, which would result in an extra conditional statement every time an undo button is pressed to ensure that the previous command is not null. NoCommand helps avoid the redundancy of that methodology.



```java
public void undo() {
    this.event.setEventInfo("In NoCommand class", "Undoing No Command", LocalDateTime.now());
    this.event.trigger();
}
```

Figure 5.30: *the undo command in the NoCommand class does nothing executes logs with the Interceptor*

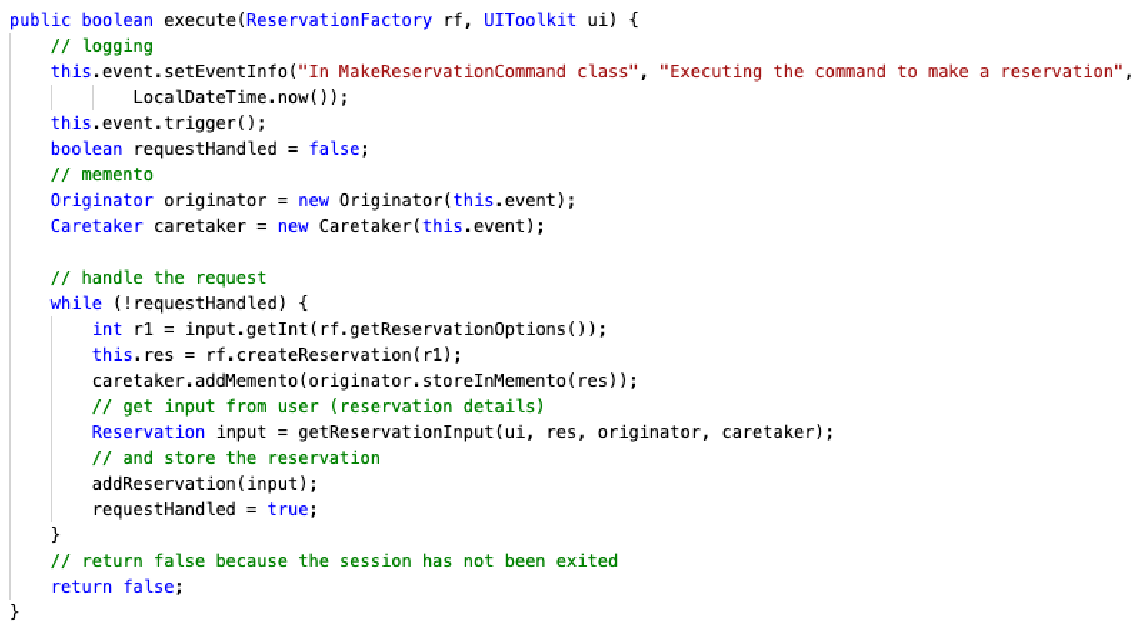
### 5.3.3.2 Concrete Commands

The existing concrete commands implemented for Reservify are:

- CancelReservationCommand
- ChangeReservationCommand
- ExitSystemCommand
- MakeReservationCommand

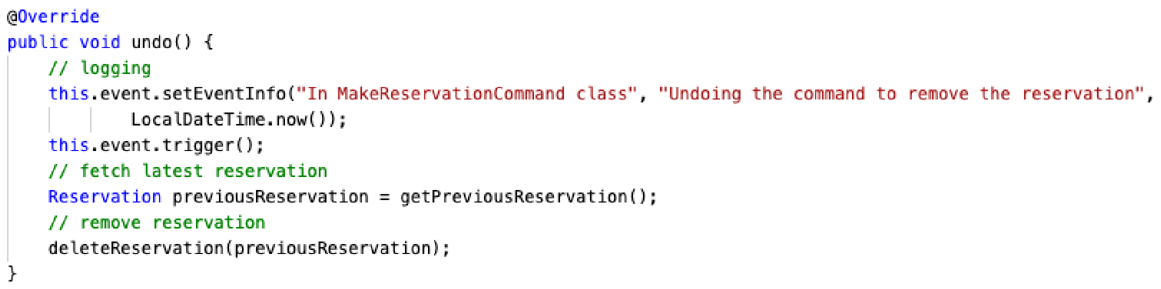
#### 5.3.3.2.1 MakeReservationCommand

The MakeReservationCommand is responsible for adding new reservations to the system.



```java
public boolean execute(ReservationFactory rf, UIToolkit ui) {
    // logging
    this.event.setEventInfo("In MakeReservationCommand class", "Executing the command to make a reservation",
            LocalDateTime.now());
    this.event.trigger();
    boolean requestHandled = false;
    // memento
    Originator originator = new Originator(this.event);
    Caretaker caretaker = new Caretaker(this.event);

    // handle the request
    while (!requestHandled) {
        int r1 = input.getInt(rf.getReservationOptions());
        this.res = rf.createReservation(r1);
        caretaker.addMemento(originator.storeInMemento(res));
        // get input from user (reservation details)
        Reservation input = getReservationInput(ui, res, originator, caretaker);
        // and store the reservation
        addReservation(input);
        requestHandled = true;
    }
    // return false because the session has not been exited
    return false;
}
```

Figure 5.31: *execute method implemented in the MakeReservationCommand, where a new reservation gets collected from the user and added to the database*

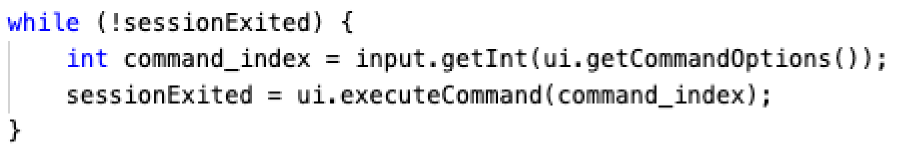


```
@Override
public void undo() {
    // logging
    this.event.setEventInfo("In MakeReservationCommand class", "Undoing the command to remove the reservation",
            LocalDateTime.now());
    this.event.trigger();
    // fetch latest reservation
    Reservation previousReservation = getPreviousReservation();
    // remove reservation
    deleteReservation(previousReservation);
}
```

Figure 5.32: *the undo method implemented in the MakeReservationCommand which fetches the latest reservation and deletes it from the database*

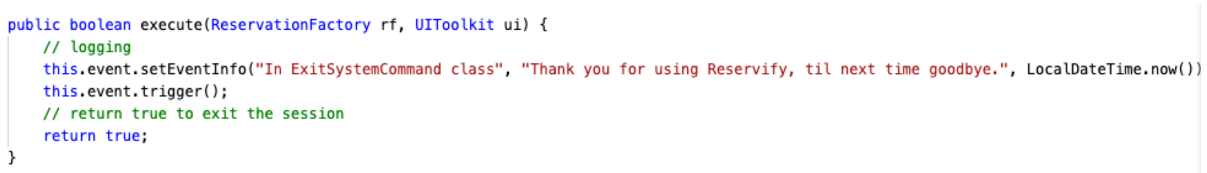
### 5.3.3.2.2 ExitSystemCommand

The exit system command gets called by a user when they are finished with the ongoing session. This command does perform any actions inside the undo() method as it can never be called. For execute(), it lets the system know that it can stop receiving requests and finish execution.



```
while (!sessionExited) {
    int command_index = input.getInt(ui.getCommandOptions());
    sessionExited = ui.executeCommand(command_index);
}
```

Figure 5.33: *receiving user requests inside App.java*



```
public boolean execute(ReservationFactory rf, UIToolkit ui) {
    // logging
    this.event.setEventInfo("In ExitSystemCommand class", "Thank you for using Reservify, til next time goodbye.", LocalDateTime.now())
    this.event.trigger();
    // return true to exit the session
    return true;
}
```

Figure 5.34: *the ExitSystemCommand returns true when its execute() method gets called, which means that the session has been terminated by the user*

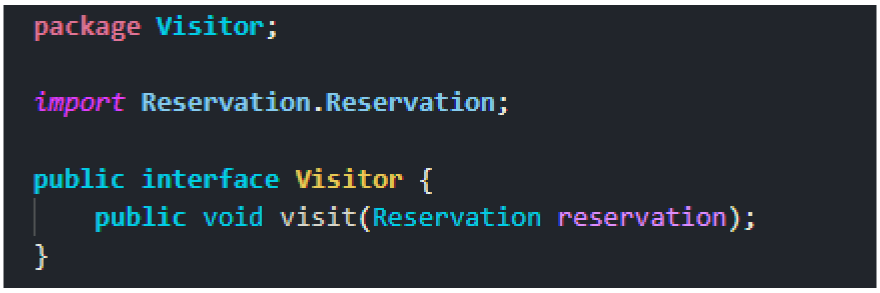
## 5.4 Visitor

The Visitor pattern lets us define new operations without introduction of modifications to an existing object structure. Without bloating the existing code, we can simply add a currency converter algorithm to the reservation class from which it is entirely separated. This gives way to many other capabilities to be implemented in the future e.g. Tax for a new way of taxing, Specific types of discounts etc.

### 5.4.1 Currency Converter

#### 5.4.1.1 The Visitor Interface

The visitor class contains a method in which includes the class we will be 'visiting' in the parameters as seen in *Figure 5.35*.

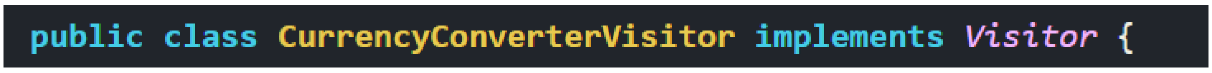


```java
package Visitor;

import Reservation.Reservation;

public interface Visitor {
    public void visit(Reservation reservation);
}
```

Figure 5.35: Visitor interface

#### 5.4.1.2 The CurrencyConverterVisitor class

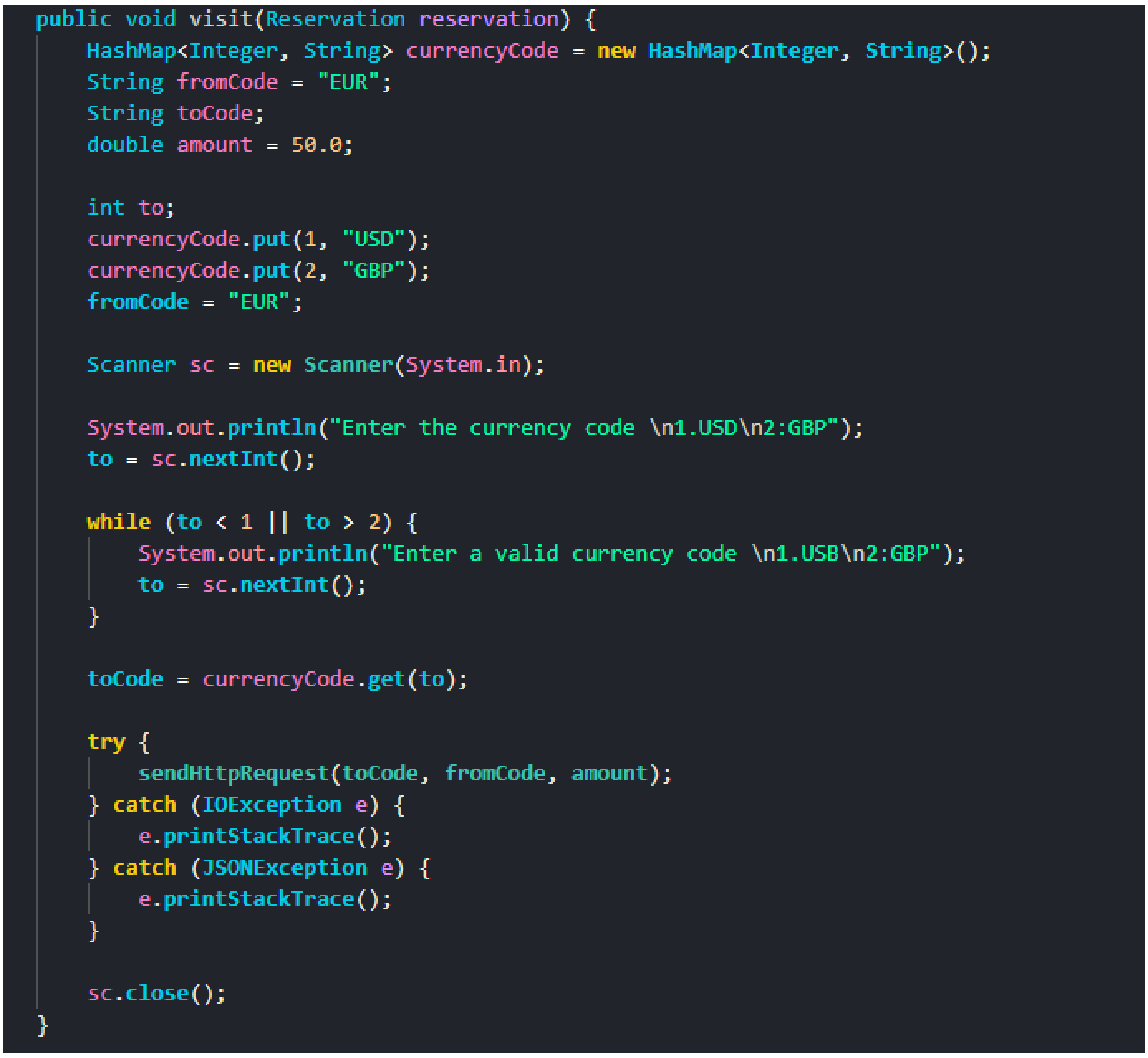
In the CurrencyConverterVisitor, we implement the visitor class above as seen in *Figure 5.36*.



```java
public class CurrencyConverterVisitor implements Visitor {
```

Figure 5.36: CurrencyConverterVisitor implementing Visitor

#### 5.4.1.2.1 Visit Command



```java
public void visit(Reservation reservation) {
    HashMap<Integer, String> currencyCode = new HashMap<Integer, String>();
    String fromCode = "EUR";
    String toCode;
    double amount = 50.0;

    int to;
    currencyCode.put(1, "USD");
    currencyCode.put(2, "GBP");
    fromCode = "EUR";

    Scanner sc = new Scanner(System.in);

    System.out.println("Enter the currency code \n1.USD\n2:GBP");
    to = sc.nextInt();

    while (to < 1 || to > 2) {
        System.out.println("Enter a valid currency code \n1.USB\n2:GBP");
        to = sc.nextInt();
    }

    toCode = currencyCode.get(to);

    try {
        sendHttpRequest(toCode, fromCode, amount);
    } catch (IOException e) {
        e.printStackTrace();
    } catch (JSONException e) {
        e.printStackTrace();
    }

    sc.close();
}
```

Figure 5.37: visit method in CurrencyConverter

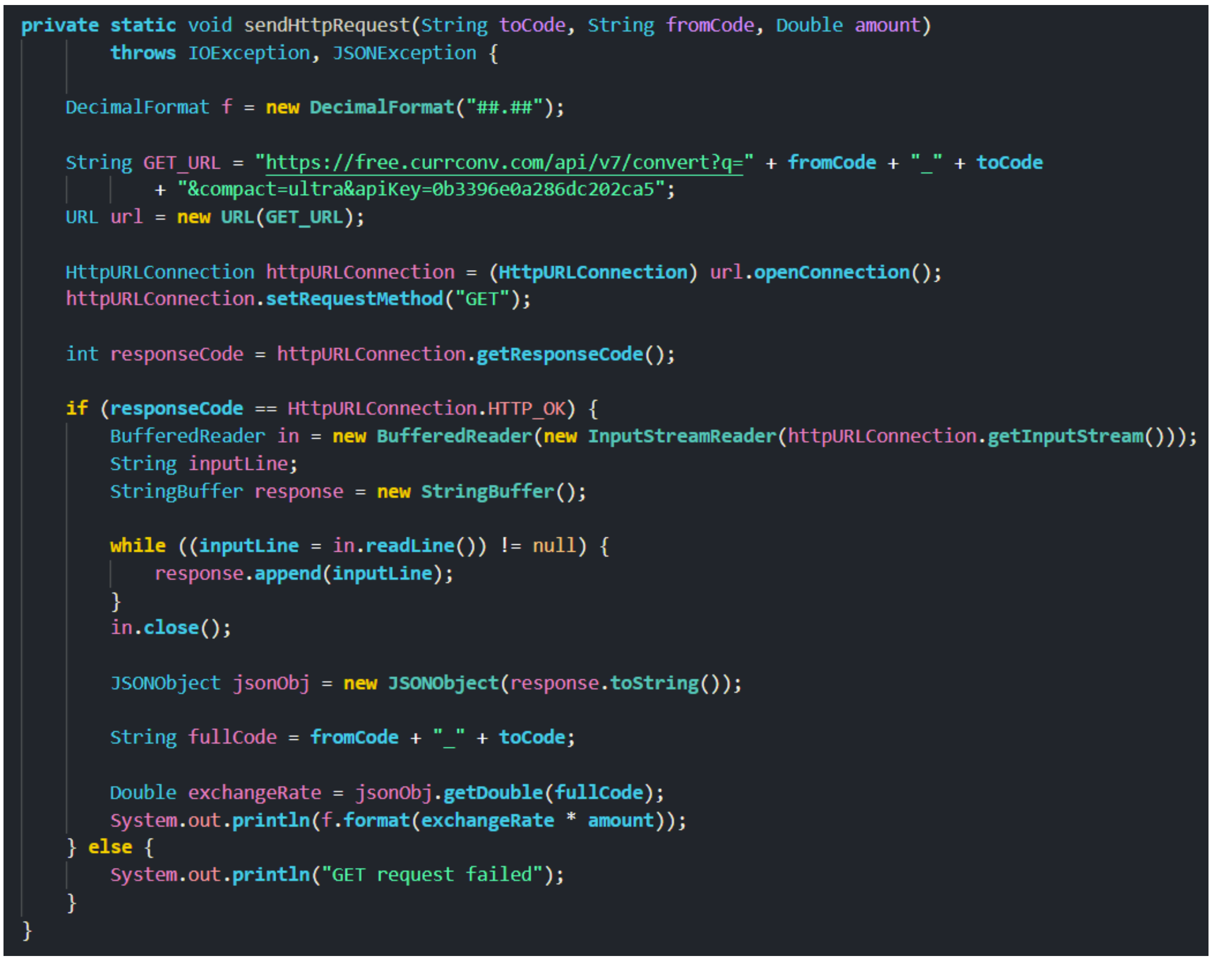
This visit method (see *Figure 5.37*) is overridden from the Visitor class and this is where we implement the Currency Conversion. Here, we can see that we have a “from” currency code and a “to” currency code where the ‘from’ code is always Euro. These represent the currency we are converting from and to, for example:

- EUR
- USD
- GBP

We create a hashmap in order to represent each of the codes with a specific number. For example, if we enter 1, the currency code is USD and we will convert from EUR to USD in that case.

A check to ensure that the user has entered a 1 or a 2 is included. A try catch is used in order to catch any exceptions that occur with the HTTP request to a currency converter API.

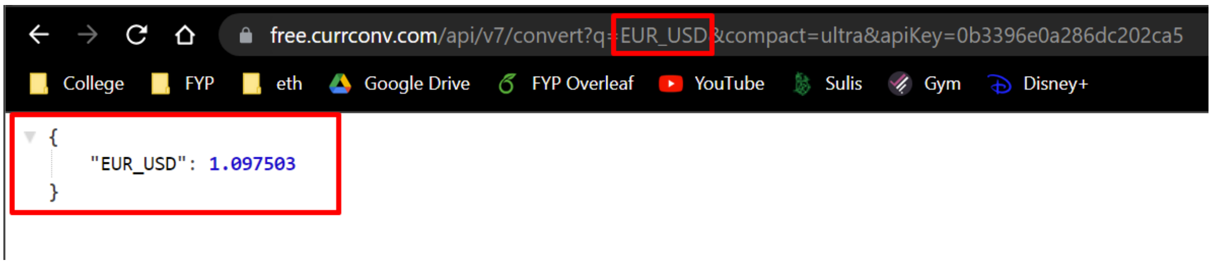
### 5.4.1.2.2 Send Http Request method



```
private static void sendHttpRequest(String toCode, String fromCode, Double amount)
        throws IOException, JSONException {

    DecimalFormat f = new DecimalFormat("##.##");

    String GET_URL = "https://free.currconv.com/api/v7/convert?q=" + fromCode + "_" + toCode
            + "&compact=ultra&apiKey=0b3396e0a286dc202ca5";
    URL url = new URL(GET_URL);

    HttpURLConnection httpURLConnection = (HttpURLConnection) url.openConnection();
    httpURLConnection.setRequestMethod("GET");

    int responseCode = httpURLConnection.getResponseCode();

    if (responseCode == HttpURLConnection.HTTP_OK) {
        BufferedReader in = new BufferedReader(new InputStreamReader(httpURLConnection.getInputStream()));
        String inputLine;
        StringBuffer response = new StringBuffer();

        while ((inputLine = in.readLine()) != null) {
            response.append(inputLine);
        }
        in.close();

        JSONObject jsonObj = new JSONObject(response.toString());

        String fullCode = fromCode + "_" + toCode;

        Double exchangeRate = jsonObj.getDouble(fullCode);
        System.out.println(f.format(exchangeRate * amount));
    } else {
        System.out.println("GET request failed");
    }
}
```

Figure 5.38: HttpRequest method

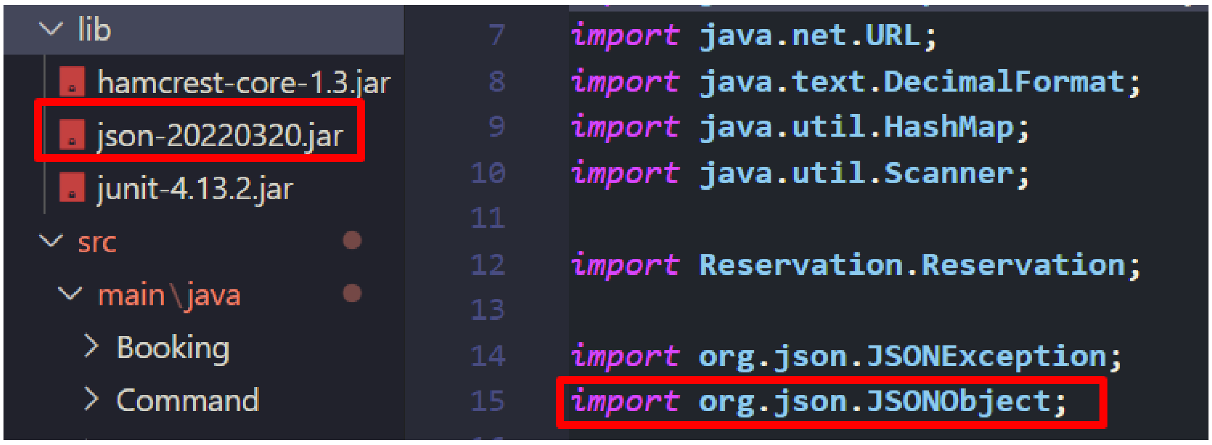
This method (see *Figure 5.38*) sends a Http request to a currency converter API that returns a JSON object in which we must read. We add the URL API that we require and replace the to and from codes based on what the user typed into the command line (see *Figure 5.39*). A Http URL connection allows us to open the URL and set the request method to GET since we are retrieving information and not writing to the server.


free.currconv.com/api/v7/convert?q=EUR_USD&compact=ultra&apiKey=0b3396e0a286dc202ca5
College
FYP
eth
Google Drive
FYP Overleaf
YouTube
Sulis
Gym
Disney+

```
{
    "EUR_USD": 1.097503
}
```

Figure 5.39: calling the API in browser

After this, we need to check to ensure that our response code received from the Http request is “ok”, hence the if statement in *Figure 5.38*. Now, we can read the information from the response. We do this using a BufferedReader. While we read the response, we add it to the Buffer and we can parse the response from the request. Since the data we are receiving is in JSON format, we will have to read the JSON through adding an external JSON jar dependency package file (see *Figure 5.40*) that allows us to create a JSONObject for the response.



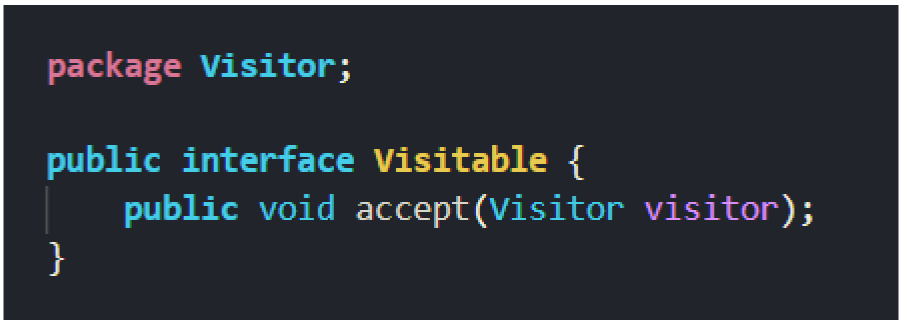
```java
import java.net.URL;
import java.text.DecimalFormat;
import java.util.HashMap;
import java.util.Scanner;

import Reservation.Reservation;

import org.json.JSONException;
import org.json.JSONObject;
```

Figure 5.40: importing necessary jar file for JSON Objects

We get the 'EUR_USD' value in the JSON file value by getting the double from the JSON object that is named as such e.g. 1.09703 in the URL checking case. This will allow us to multiply the amount by the exchange rate and return the value to the user. In the case that we do not get a HttpStatus = OK, then we need to alert the system that the GET request did not work.

### 5.4.1.3 The Visitable Interface

The visitable interface (see *Figure 5.41*) provides us with the ability to allow the class we intend to visit.

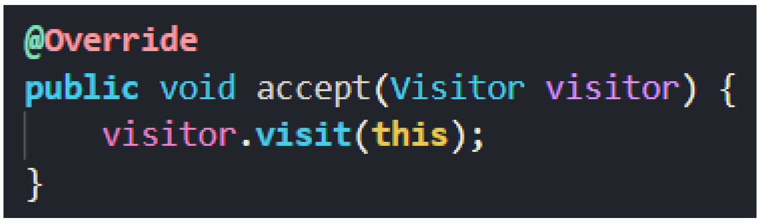


```java
package Visitor;

public interface Visitable {
    public void accept(Visitor visitor);
}
```

Figure 5.41: Visitable interface

#### 5.4.1.3.1 Accept Command

The Visitor class is implemented in the ReservationFactory where the accept method is implemented (see *Figure 5.42*).



```java
@Override
public void accept(Visitor visitor) {
    visitor.visit(this);
}
```

Figure 5.42: accepting the Visitor in ReservationFactory

## 5.5 Bridge

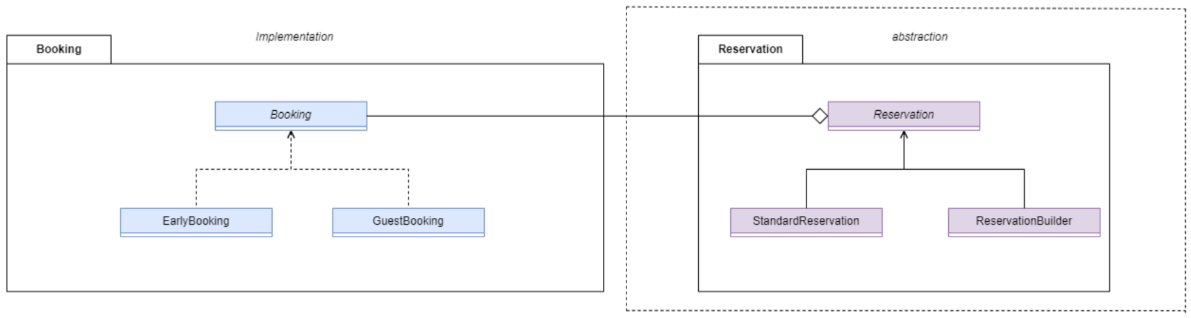


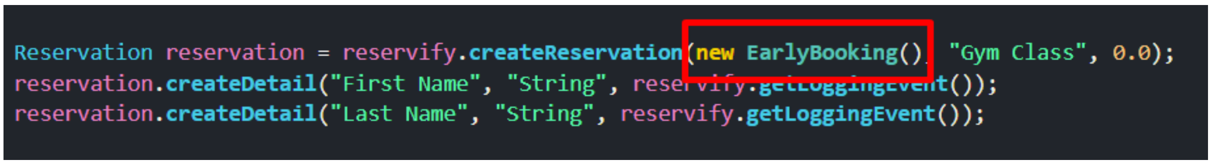
Figure 5.43: diagram of bridge in reservify

### 5.5.2 Booking

#### 5.5.2.1 EarlyBooking

Booking is our Implementation for the Reservation abstraction as seen in *Figure 5.43*. It allows us to decouple the Booking functionality into the Reservation code without bloating the Reservation code itself. The Reservation does not need to know what is in Booking. However, booking will be made up of components defined in Reservation. We can use Booking to store reservation information as a 'Booking' for future reference.

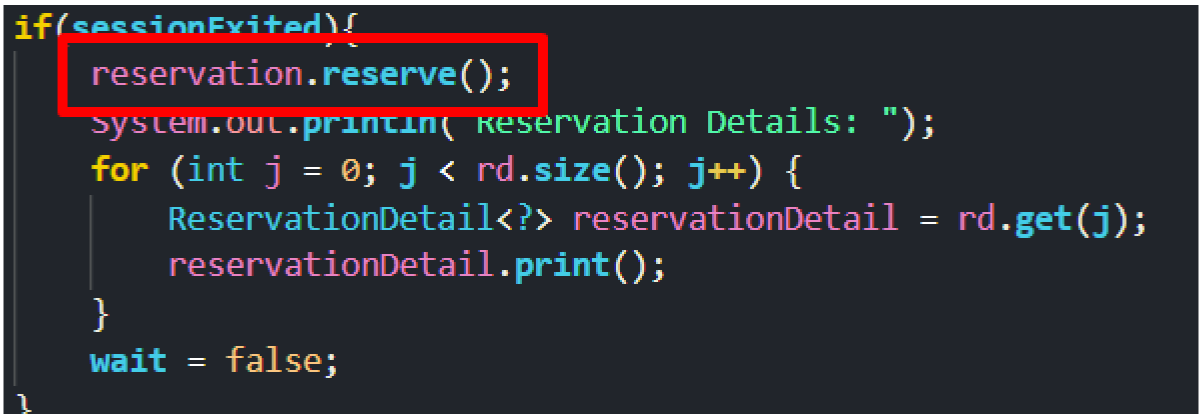
The early booking, for example, is created when we create a reservation in the App3 class ( (see *Figure 5.44*).



```java
Reservation reservation = reservify.createReservation(new EarlyBooking(), "Gym Class", 0.0);
reservation.createDetail("First Name", "String", reservify.getLoggingEvent());
reservation.createDetail("Last Name", "String", reservify.getLoggingEvent());
```

Figure 5.44: creating Early Booking in App3

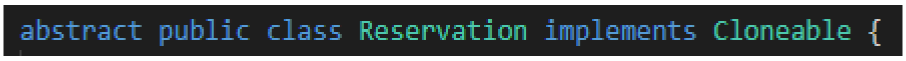
As soon as the session is over, we can save the book and make a reservation in the UIToolKit class (see *Figure 5.45*).

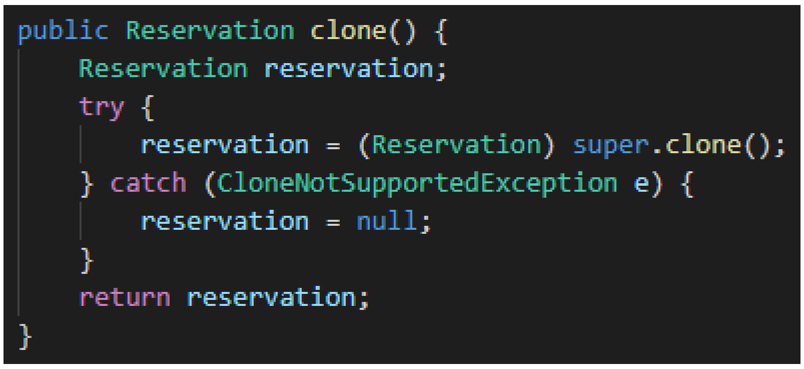


```java
if(sessionExited){
    reservation.reserve();
    System.out.println("Reservation Details: ");
    for (int j = 0; j < rd.size(); j++) {
        ReservationDetail<?> reservationDetail = rd.get(j);
        reservationDetail.print();
    }
    wait = false;
}
```

Figure 5.45: abstract reserve method called when reservation is made

## 5.5 Prototype


```java
abstract public class Reservation implements Cloneable {
```
Figure 5.46: Reservation implementing the Cloneable interface

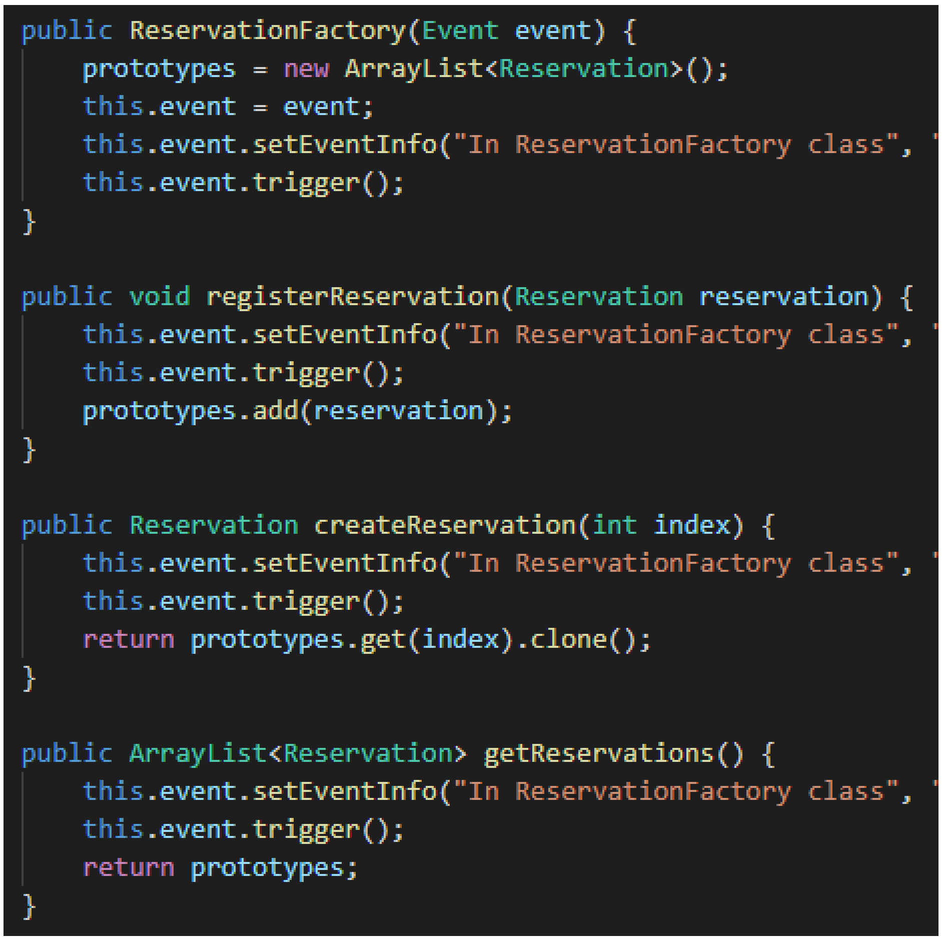

```java
public Reservation clone() {
    Reservation reservation;
    try {
        reservation = (Reservation) super.clone();
    } catch (CloneNotSupportedException e) {
        reservation = null;
    }
    return reservation;
}
```
Figure 5.47: clone method

The prototype pattern was created by implementing the cloneable interface, along with the clone method. The clone method works on all subclasses to support the creation of objects through cloning.

## 5.6 Factory

For the reservation factory, you first register a reservation to the factory by calling the registerReservation method and pass in a reservation. This reservation can now be instantiated by using the createReservation method, and passing in an index. The getReservations method returns a list of reservations that are available to be created by the factory.

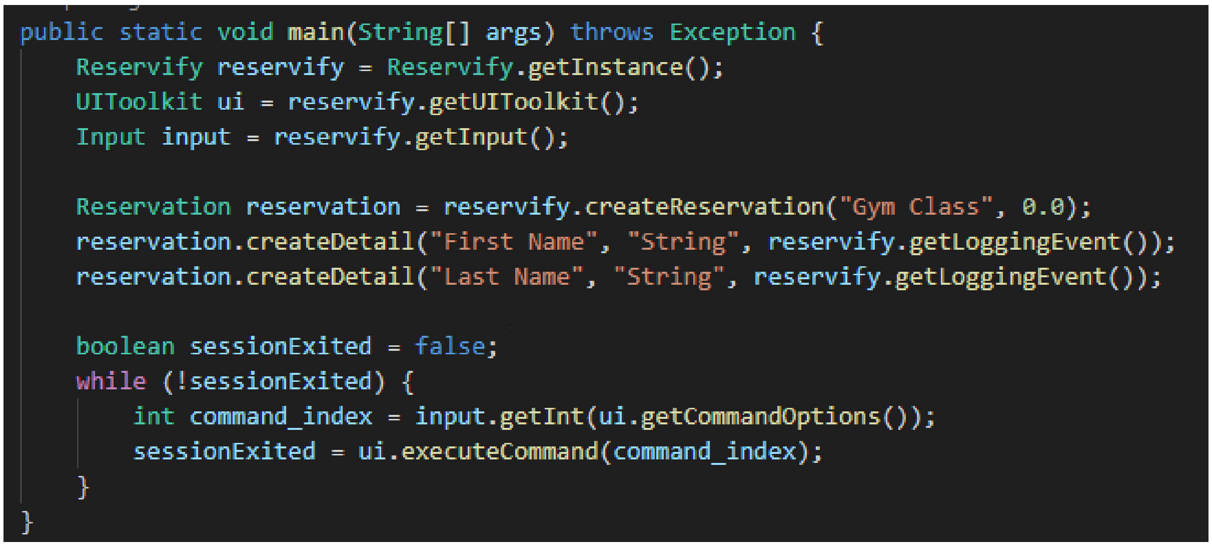
## 5.6 Facade



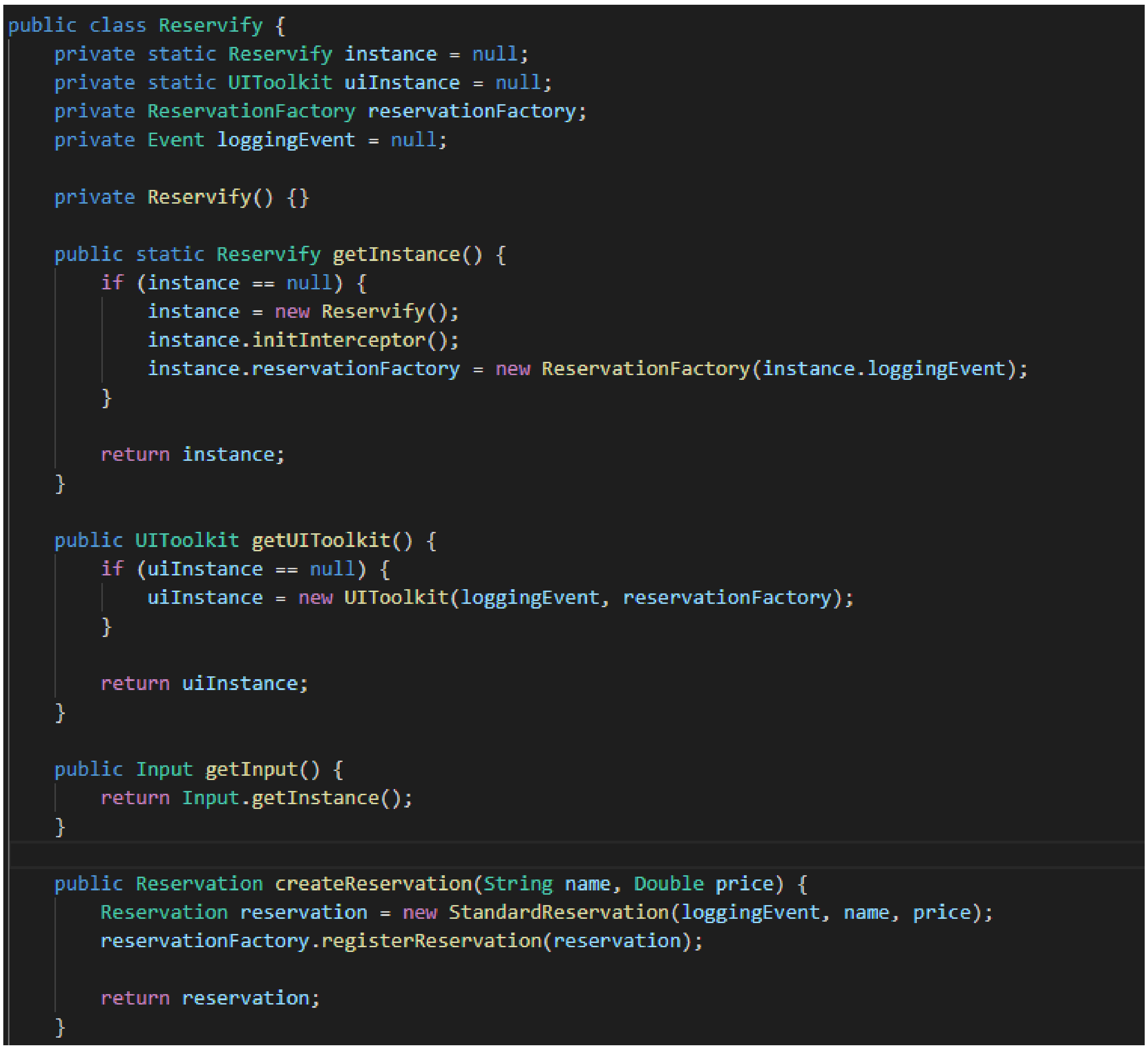
```java
public ReservationFactory(Event event) {
    prototypes = new ArrayList<Reservation>();
    this.event = event;
    this.event.setEventInfo("In ReservationFactory class",
    this.event.trigger();
}

public void registerReservation(Reservation reservation) {
    this.event.setEventInfo("In ReservationFactory class",
    this.event.trigger();
    prototypes.add(reservation);
}

public Reservation createReservation(int index) {
    this.event.setEventInfo("In ReservationFactory class",
    this.event.trigger();
    return prototypes.get(index).clone();
}

public ArrayList<Reservation> getReservations() {
    this.event.setEventInfo("In ReservationFactory class",
    this.event.trigger();
    return prototypes;
}
```

Figure 5.48: Reservation factory methods to register, create and get reservations

The facade pattern exposes only the methods that are needed for the third party developers to use the system. This means we can hide all the complexity of the code, and make it the code needed for developers less bloated.



```java
public static void main(String[] args) throws Exception {
    Reservify reservify = Reservify.getInstance();
    UIToolkit ui = reservify.getUIToolkit();
    Input input = reservify.getInput();

    Reservation reservation = reservify.createReservation("Gym Class", 0.0);
    reservation.createDetail("First Name", "String", reservify.getLoggingEvent());
    reservation.createDetail("Last Name", "String", reservify.getLoggingEvent());

    boolean sessionExited = false;
    while (!sessionExited) {
        int command_index = input.getInt(ui.getCommandOptions());
        sessionExited = ui.executeCommand(command_index);
    }
}
```



```java
public class Reservify {
    private static Reservify instance = null;
    private static UIToolkit uiInstance = null;
    private ReservationFactory reservationFactory;
    private Event loggingEvent = null;

    private Reservify() {}

    public static Reservify getInstance() {
        if (instance == null) {
            instance = new Reservify();
            instance.initInterceptor();
            instance.reservationFactory = new ReservationFactory(instance.loggingEvent);
        }

        return instance;
    }

    public UIToolkit getUIToolkit() {
        if (uiInstance == null) {
            uiInstance = new UIToolkit(loggingEvent, reservationFactory);
        }

        return uiInstance;
    }

    public Input getInput() {
        return Input.getInstance();
    }

    public Reservation createReservation(String name, Double price) {
        Reservation reservation = new StandardReservation(loggingEvent, name, price);
        reservationFactory.registerReservation(reservation);

        return reservation;
    }
```

Figure 5.49: Reservify class

The code to implement the system is miniscule, powerful and easy to understand for the developer.

# 6. WOW factor

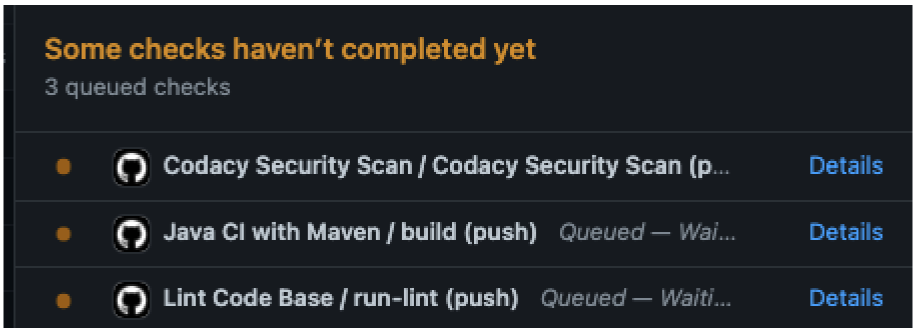
## 6.1. Code Evaluation Tool: Super-Linter

Automated Linting flows have allowed us to ensure a high quality of code in our previous projects and was one of the first additions to Reservify. Super-Linter is a GitHub Actions workflow which is a combination of various linters and helps validate source code in a few different programming languages. Linters are especially useful in group work settings as they allow the team to:

- build rules and guidelines and stick to them in terms of code structure
- prevent code that breaks the system to be pushed onto the main branch
- simplify code reviews: smaller issues get caught and pointed out by Linter instead of another group member

The result of each Linter run is visible in the repository. The Linter outputs the logs about the issues it caught and suggests a solution when possible. In case of a Linter run failure, each team member focuses on fixing their commits instead of moving onto a different coding task.

Overall the Super-Linter became a simple but useful tool throughout the development process and allowed the team to stay productive while keeping the code base error-free and cohesive in terms of code structure.

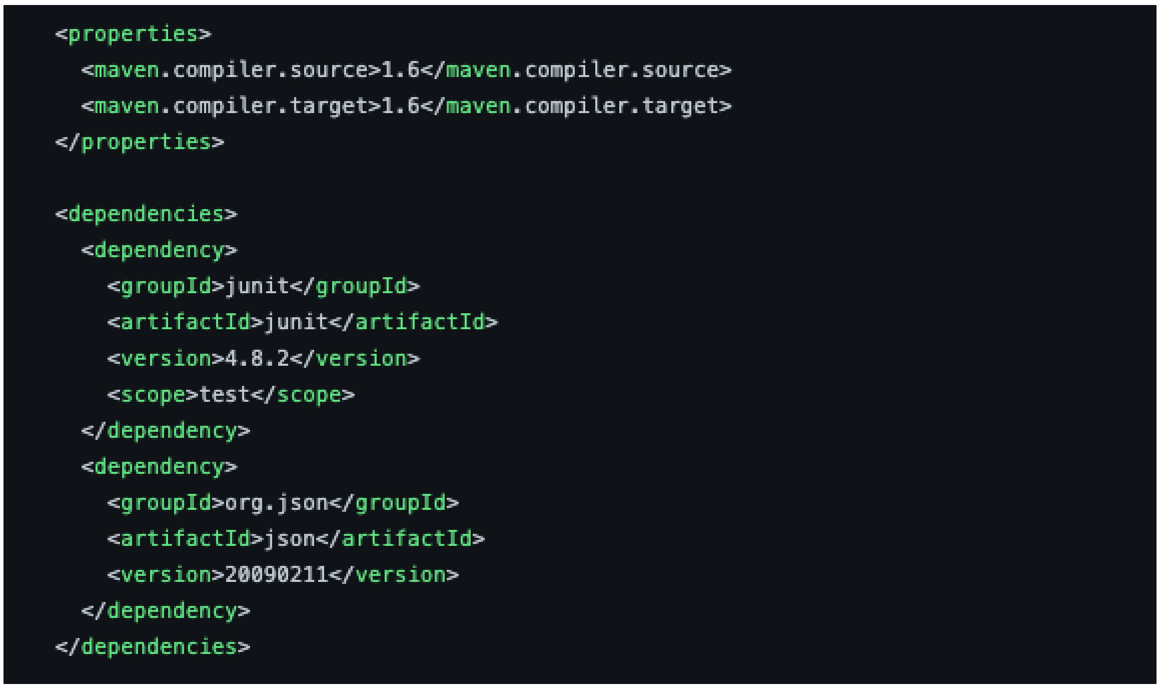


Some checks haven't completed yet
3 queued checks

- Codacy Security Scan / Codacy Security Scan (p... — Details
- Java CI with Maven / build (push) *Queued — Wai...* — Details
- Lint Code Base / run-lint (push) *Queued — Waiti...* — Details

Figure 6.1: *3 workflows run at each push to the main branch*

## 6.2. DevOps: Automated Project Building and Testing with Maven



```xml
<properties>
  <maven.compiler.source>1.6</maven.compiler.source>
  <maven.compiler.target>1.6</maven.compiler.target>
</properties>

<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.8.2</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.json</groupId>
    <artifactId>json</artifactId>
    <version>20090211</version>
  </dependency>
</dependencies>
```

Figure 6.2: *maven pom.xml file setup*

Maven is an open source software management and code comprehension tool developed by Apache. It allows developers to automate code building, testing and more with the use of its POM file. We began working with maven at the midpoint of the development process. To ensure a successful Maven run, the code needed to have a specific Maven-appropriate structure.

- /src/ directory containing all source code
- /lib/ directory containing .jar libraries
- /target/ containing test classes for Maven in binary form

The refactoring of our code base caused some small road blocks and code errors but we managed to fix them in a couple of coding sessions.

As a result, we added a Maven workflow to our GitHub repository which builds the project on a Windows operating system at each push to the main branch. This ensures that no broken code can be committed to the main branch and that our code works across multiple platforms since some team members were using a different operating system.

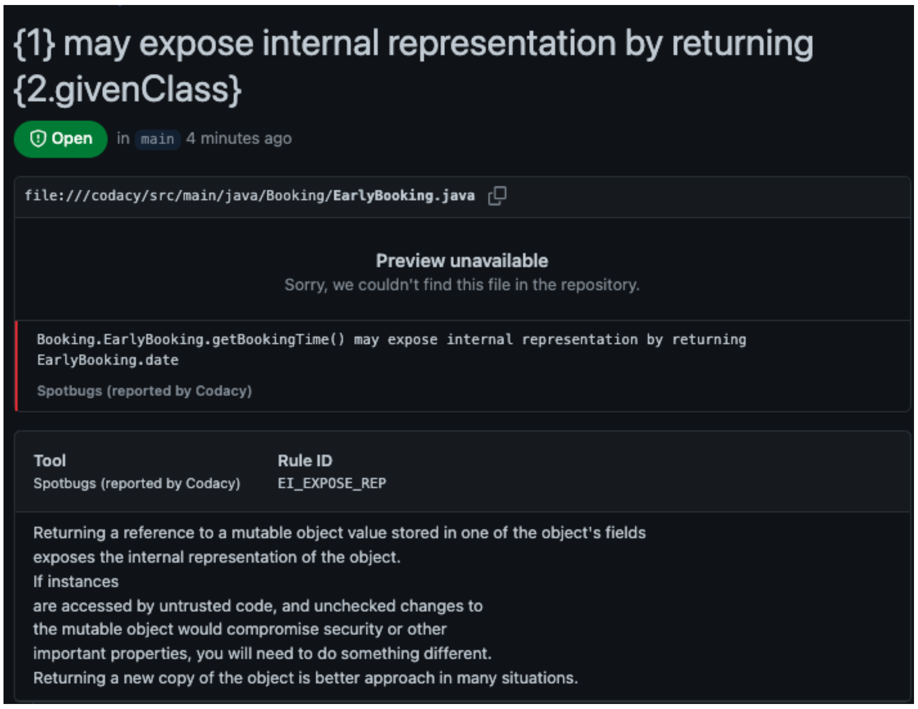
We also found another use for our Maven workflow and directed Maven to our testing directory, where Maven is able to run all the JUnit tests at each commit to the main branch. This ensures that no new commits break any existing tests.

## 7.3. Security: The Codacy Security Scan

Codacy is a security scanning tool which we incorporated into our system by setting up the Codacy Analysis CLI tool using GitHub Actions. This tool runs on every commit to the main branch and analyses the security of the system.

Out of many different security tools, we opted for Codacy due to the fact that:

- It is able to recognize project flaws such as: code complexity, code duplication, code style, performance, error proneness, unused code and more.
- Codacy is able to scan code in over 30 programming languages, and automatically recognises which programming language each file in the coding directory uses and changes its checking methodology accordingly
- Displays the security scan result in detail under the Security tab on the GitHub repository page and opens an issue automatically. This allows the team to have a list of ongoing security issues and solve them as the project progresses.



{1} may expose internal representation by returning {2.givenClass}

Open in main 4 minutes ago

file:///codacy/src/main/java/Booking/EarlyBooking.java

Preview unavailable
Sorry, we couldn't find this file in the repository.

```
Booking.EarlyBooking.getBookingTime() may expose internal representation by returning
EarlyBooking.date
```

Spotbugs (reported by Codacy)

| Tool | Rule ID |
|---|---|
| Spotbugs (reported by Codacy) | EI_EXPOSE_REP |

Returning a reference to a mutable object value stored in one of the object's fields
exposes the internal representation of the object.
If instances
are accessed by untrusted code, and unchecked changes to
the mutable object would compromise security or other
important properties, you will need to do something different.
Returning a new copy of the object is better approach in many situations.

Figure 7.3: *a GitHub issue opened automatically by Codacy when finding a security flaw*

For the project to work with Codacy, we set up a Security policy which lets people know which versions of our project support automated Security scans.

# 8. Testing

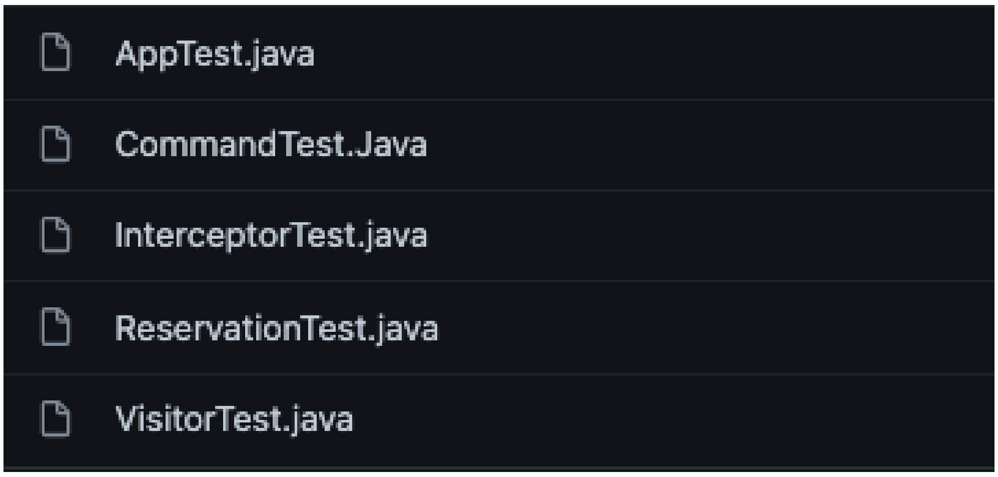

AppTest.java
CommandTest.Java
InterceptorTest.java
ReservationTest.java
VisitorTest.java

Figure 8.1: *the files in the tests directory*

Reservify is a system implemented in Java, therefore uses the JUnit testing methodology, which allows software developers to test complex and scalable systems in detail.

Units are located in the target/test_classes directory and contain test classes for the different design and architectural patterns used in the system to ensure that any additional changes do not break any core system behaviour. Within the classes, user-system interaction is simulated and then tested.

To ensure that testing could be carried out without the need of manually triggering tests, we set up a Maven, which is a well known software management and comprehension tool and allows for automated project building and software testing. The Maven workflow set up by us runs the test classes located in the test folder and notifies the team in case of test or build failure. This allows us to continuously develop our system without the need of manual testing calls.

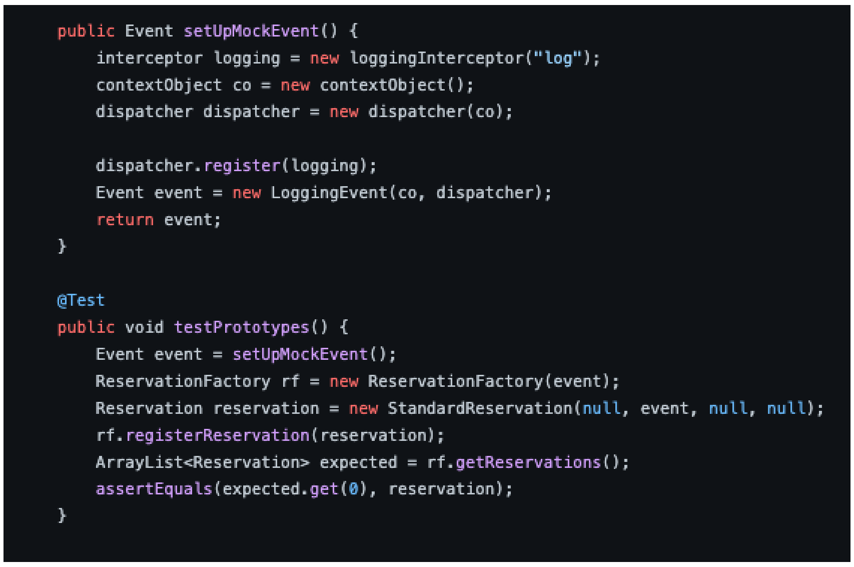

```java
public Event setUpMockEvent() {
    interceptor logging = new loggingInterceptor("log");
    contextObject co = new contextObject();
    dispatcher dispatcher = new dispatcher(co);

    dispatcher.register(logging);
    Event event = new LoggingEvent(co, dispatcher);
    return event;
}

@Test
public void testPrototypes() {
    Event event = setUpMockEvent();
    ReservationFactory rf = new ReservationFactory(event);
    Reservation reservation = new StandardReservation(null, event, null, null);
    rf.registerReservation(reservation);
    ArrayList<Reservation> expected = rf.getReservations();
    assertEquals(expected.get(0), reservation);
}
```

Figure 8.2: *an example of tests for the Reservation class. Which makes sure that Reservations get correctly registered using the Prototype design pattern.*

# 9. Problems encountered

## 9.1. Maven Integration Issues

We began the Maven setup at the midpoint of the project. Throughout the implementation process, we had all been using VSCode as our default IDEs. Maven's integration with this IDE can sometimes be flawed, which we did not find out until later. As a result, we came across many instances where our project did not build on either of our machines. Through research, we found people with similar problems online[2] and found that VSCode's functionality of clearing the workspace worked to get our build up and running again.

This integration also caused issues with importing external dependencies such as the jar file used in order to create JSON objects to use APIs while it worked prior to the integration. It eventually was resolved given the research mentioned above.

# 10. Evaluation and Critique

## 10.1. Initial Idea and Approach

Our initial approach was to build the most stripped down version of a Reservation system as possible. This system would be applied to concrete cases and extended by our clients. Although we ended up creating the same exact system in the, our initial approach was not ideal.

Due to the fact that we had no concrete use cases or requirements, it was very difficult for us to fully integrate all the functionalities that we had in mind. Portions of our code were left unfinished due to the concept not being fully developed.

We decided to take a new approach when we realised that the speed of our development was halted, and began by focusing on a concrete use case. We focused on a library system, where a user would be able to reserve a room, a laptop and a book. This gave us a few ideas:

- There exist bookable items for the system (i.e. Book, Laptop, Room)
- We need information from the user when booking (Name, Time, Date, etc.)
- All the classes need to outputting logs for easier debugging

From these ideas, we built a functioning library system, and afterwards, stripped it down to become a generic reservation system. We kept all the vital information and got rid of all the library-specific parts of code.

From this experience we realised that although our initial idea was good and achievable, beginning from an extremely generic subset of use cases was not enough for us to fully develop our system. Instead, we focused on a specific case and expanded from there.

## 10.2. Patterns Chosen

For the implementation of our project we used the functionalities of the following design patterns:

| Creational | Structural | Behavioural |
|---|---|---|
| Factory | Bridge | Command |
| Prototype | Facade | Interceptor |
| | | Memento |
| | | Visitor |

To choose the correct types of design patterns for the problems at hand, we first began by identifying the desirable structure and behaviour of our system. We then researched the different pros and cons of each design pattern and identified parts of our system where these functionalities would be of use.

This approach was different from our approach in the previous project (auctionly), where we began by writing code first and refactoring it with design patterns later. We found in the previous case that replacing existing code with design patterns was time consuming and often caused in breaking other code in place. Therefore we changed our ways and focused even more intently on design and architectural patterns for this system.

To implement the *command pattern*, we began thinking about the types of requests that can be going into and out of our system, which we could then encapsulate using the Command Pattern. Because we were implementing a booking system, we wanted to have a minimal text based user interface, where a user would be letting the system know what actions they wanted to take. The straightforward approach here would be the use of if statements, which match the user's input to the system's functionality, but we identified that this would be the perfect place to avoid overloading the system with conditional logic and instead using a Command pattern.

The pattern helped us structure the way requests would move around the system and the kind of information the system needed for each of the requests to be fulfilled with ease. The definition of a command interface allowed us to add or remove new commands in the system whenever there was a need to do so. Overall, we are happy with the choice of the command pattern to drive our user interface as it avoided using logic branches, gave us a definition of a command and worked well with the other parts of the system.

We wanted to equip our system with logging functionality, which would allow us to see how our driver moved from one part of the system to another. We knew that the *interceptor pattern* is one of the most useful patterns for implementing logging and began by setting up logging.

The logging interceptor became a very helpful part of our system as it seamlessly integrated with the other parts and allowed us to keep track of the systems behaviour at all times. Later we added another concrete interceptor, the Goodbye Interceptor which had a different functionality but was easy to implement due to the interceptor interface we had in place.

Since we had already implemented the undo functionality for user requests in the Command pattern, we decided to implement it with *the Memento pattern* as well to compare and contrast the two. Instead of replacing Command undo, we used Memento to undo the details of each reservation i.e. the user entered their name and wants to undo this operation.

Undoing "larger" requests with Command and "smaller" details with Memento worked really well for us as it allowed us to differentiate between core system requests and the sub-requests between them. We did not find either to be better at implementing Undo and are happy with which parts of the system each of them help out with.

We tried to look for an easy way to flexibly create user forms for our system, where each client would be able to specify what types of information they needed from their users (i.e. String data types of name, integer data types of age). We looked for the best design pattern to use that would allow us to implement this functionality and came across the *prototype pattern*. This pattern is not usually used in this way but we found that it worked very seamlessly with our system and gave us the exact functionality we needed.

When we noticed that our Reservation package was becoming bloated with several classes and different functionalities, we decided to use the Bridge pattern to create a separation between the abstraction (Reservation) and the implementation (Booking) of our project. This was quite difficult to implement due to the fact that we were halfway through the implementation and it needed quite a bit of refactoring. However it allowed us to create a clear separation between the two and we can now use the Booking package for additional functionalities that we should not bloat Reservation with.

We found a use for the *Visitor pattern* when we decided to implement a currency converter functionality for our system. This pattern seamlessly integrates with the rest of the classes and easily converts currencies from one to another. This gave us a deep understanding on the importance and strength of the visitor pattern and gave us an idea of any other functionalities to add to our system.

Lastly, we implemented the *Facade pattern* to tie everything together and created the Reservify class. Which includes all the public (client ready) functionalities that each client can utilise to create their own system based on our implementation. It abstracts all the internal code from the client and only exposes the core functionalities needed to support the clients.

We found that researching design patterns at the start of the implementation and finding fitting functionalities for them within our system worked really well for us and we got to learn more about them as we progressed while working on this project.

# 10. References

[1] Refactoring.guru. n.d. *Chain of Responsibility*. [online] Available at: <https://refactoring.guru/design-patterns/chain-of-responsibility> [Accessed 5 April 2022].

[2] GitHub. 2020. *Working Maven project reports multiple errors in VSCode and will not build · Issue #1232 · redhat-developer/vscode-java*. [online] Available at: <https://github.com/redhat-developer/vscode-java/issues/1232> [Accessed 5 April 2022].

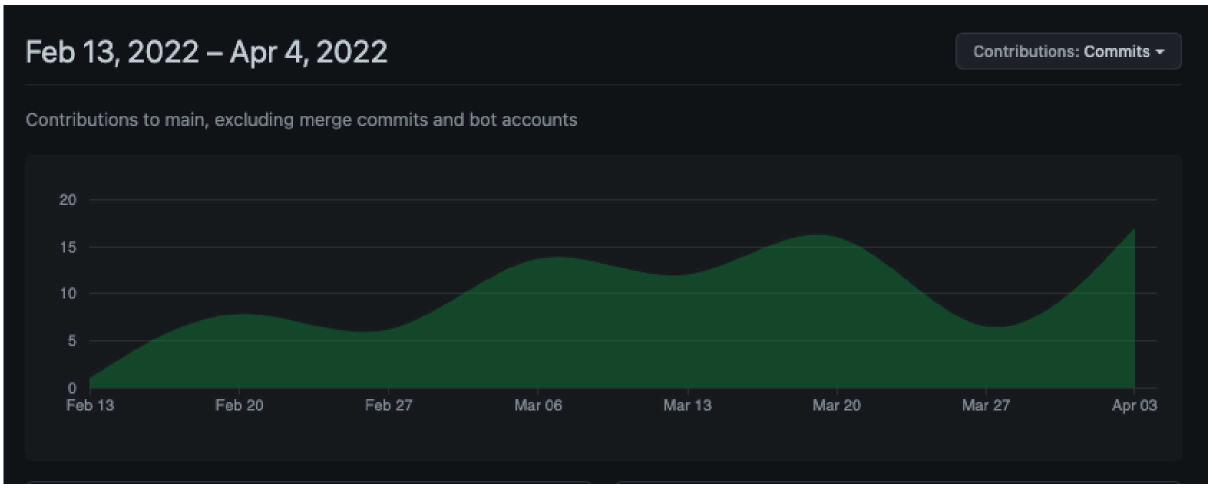
[3] Martin, R., 2009. *Clean code. A handbook of agile software craftsmanship*. Boston: Pearson Education.

[4] Kazman, R., Bass, L. and Clements, P., 2013. *Software Architecture in Practice, Second Edition*. 3rd ed. Addison-Wesley.

## 11. Contribution of each team member

| Package | Class | Ranya | John | Nutsa | Erona | Lines |
|---|---|---|---|---|---|---|
| Booking | Booking | 0 | 0 | 0 | 26 | 26 |
| | EarlyBooking | 0 | 0 | 0 | 43 | 43 |
| | GuestBooking | 0 | 0 | 0 | 49 | 49 |
| Command | CancelReservationCommand | 0 | 0 | 42 | 0 | 42 |
| | ChangeReservationCommand | 0 | 0 | 45 | 0 | 45 |
| | Command | 0 | 0 | 16 | 0 | 16 |
| | ExitSystemCommand | 0 | 0 | 41 | 0 | 41 |
| | MakeReservationCommand | 46 | 0 | 45 | 0 | 91 |
| | NoCommand | 0 | 0 | 41 | 0 | 41 |
| | UIToolKit | 30 | 102 | 40 | 14 | 186 |
| | UndoReservationDetail | 42 | 0 | 0 | 0 | 42 |
| Event | Event | 10 | 0 | 0 | 0 | 10 |
| | GoodbyeEvent | 33 | 0 | 0 | 0 | 33 |
| | LoggingEvent | 31 | 0 | 0 | 0 | 31 |
| | WelcomeEvent | 32 | 0 | 0 | 0 | 32 |
| Input | Input | 0 | 62 | 0 | 0 | 62 |
| Interceptor | ContextObject | 39 | 0 | 0 | 0 | 39 |
| | Dispatcher | 31 | 0 | 0 | 0 | 31 |
| | GoodbyeInterceptor | 41 | 0 | 0 | 0 | 41 |
| | Interceptor | 11 | 0 | 0 | 0 | 11 |
| | LoggingInterceptor | 46 | 0 | 0 | 0 | 46 |
| | WelcomeInterceptor | 41 | 0 | 0 | 0 | 41 |
| Memento | Caretaker | 29 | 0 | 0 | 0 | 29 |
| | Memento | 6 | 0 | 0 | 0 | 6 |
| | Originator | 49 | 0 | 0 | 0 | 49 |
| Reservation | Reservation | 19 | 19 | 0 | 19 | 56 |
| | ReservationBuilder | 0 | 0 | 0 | 0 | 0 |
| | ReservatioNDetail | 0 | 58 | 0 | 0 | 58 |

| | | | | | | |
|---|---|---|---|---|---|---|
| | ReservationFactory | 0 | 82 | 0 | 0 | 82 |
| | ReservationType-ENUM | 0 | 0 | 0 | 11 | 11 |
| | StandardReservation | 0 | 25 | 0 | 25 | 49 |
| Reservify | Reservify | 0 | 41 | 0 | 41 | 82 |
| Visitor | CurrencyConverterVisitor | 0 | 0 | 0 | 124 | 124 |
| | DiscountVisitor | 0 | 0 | 0 | 13 | 13 |
| | Visitor | 0 | 0 | 0 | 5 | 5 |
| | Visitable | 0 | 0 | 0 | 7 | 7 |
| App | App1 | 0 | 29.5 | 0 | 29.5 | 59 |
| | App2 | 15 | 15 | 15 | 15 | 60 |
| | App3 | 0 | 12 | 0 | 12 | 23 |
| Tests | AppTest | 0 | 0 | 35 | 0 | 35 |
| | CommandTest | 0 | 0 | 25 | 26 | 51 |
| | InterceptorTest | 0 | 0 | 18 | 0 | 18 |
| | ReservationTest | 0 | 21 | 21 | 0 | 42 |
| Workflows | Maven | 0 | 0 | 27 | 0 | 27 |
| | Super-Linter | 0 | 0 | 30 | 0 | 30 |
| | Codacy | 0 | 0 | 60 | 0 | 60 |
| | Pom | 0 | 0 | 25 | 25 | 50 |
| **Totals** | | 551 | 466.5 | 526 | 484.5 | 2025 |


Feb 13, 2022 – Apr 4, 2022

Contributions to main, excluding merge commits and bot accounts