Design Patterns

SEM-08b

December 16, 2021

1 Design pattern 1: Strategy

1.1 Introduction

In our application a user can see their schedule consisting of their bookings. A user may need to sort their schedule of bookings using various sorting methods. The first design pattern we have selected to implement is the strategy pattern. This pattern is effective in our design architecture, as it allows the schedule of bookings to be sorted in one of many sorting methods. This provides the user freedom within our application, while further improving the efficiency for users.

1.2 Implementation

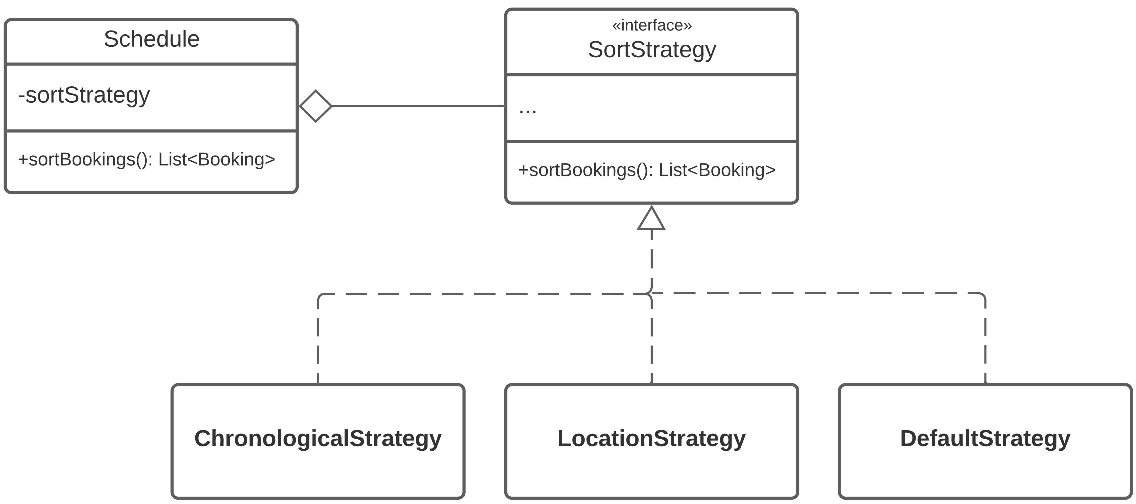

Figure 1.1: Strategy Class Diagram

This design pattern is used to sort the list of bookings for each user. The Booking class has an attribute which stores the owner of the booking and each user would want to access the bookings that they have made. Hence, the schedule class contains a list of bookings made by the same booking owner and the strategy decides how to sort them. Figure 3.1 shows a snippet of the schedule class and Figure 3.2 shows a snippet of the SortStrategy interface (see Appendix A). By default the schedule is made to only contain future bookings.

The schedule class contains a list of bookings and a sortStrategy object of type SortStrategy. When the user requests a sorted list of bookings, a new schedule object with a corresponding sortStrategy is created. The method of sorting corresponding to the schedule is then applied to the list of bookings.

Having the sortStrategy as a field in the schedule class, allows us to have schedules with different sorting techniques without implementing the schedule multiple times.

The idea of the SortStrategy interface is simple. It only contains one method that each child class has to override. By implementing this interface, and consequently the sortBookings method, we create new sorting strategies. These new strategies can be used in the Schedule class.

The process of adding new sorting strategies has been made simple. This is due to the SortStrategy interface. Implementing a new sort strategy just consists of building off of the interface, hence overriding its methods.

1.2.1 Default Strategy

This strategy is used when the user has no preference on how they want to view the bookings. This is the default ordering of the bookings.

The list of bookings is sorted in ascending order in terms of the ID of the booking.

1.2.2 Chronological Strategy

Users might want to see their earliest bookings first. We implemented this sorting strategy so users can see the bookings in chronological order. This allows users to view their bookings from earliest to latest.

The list of bookings will be sorted from the first to last booking based on the date. Furthermore, if the dates of two bookings are on the same day, the start time of the booking is considered.

1.2.3 Location Strategy

If the user wants to know which bookings are taking place next to each other they can use this sorting strategy.

This has been implemented to sort bookings according to the building and room they are taking place in. This is useful . If two bookings are happening in the same building, they are sorted according to room number.

1.3 Possible extensions

Looking at the future of our application, we can optionally add more sort strategies to further provide the users with more freedom.

There is also a possibility to implement a schedule for each room or building. Henceforth, new sorting strategies can be added, which are more suited towards these new schedules.

2 Design pattern 2: Chain of responsibility

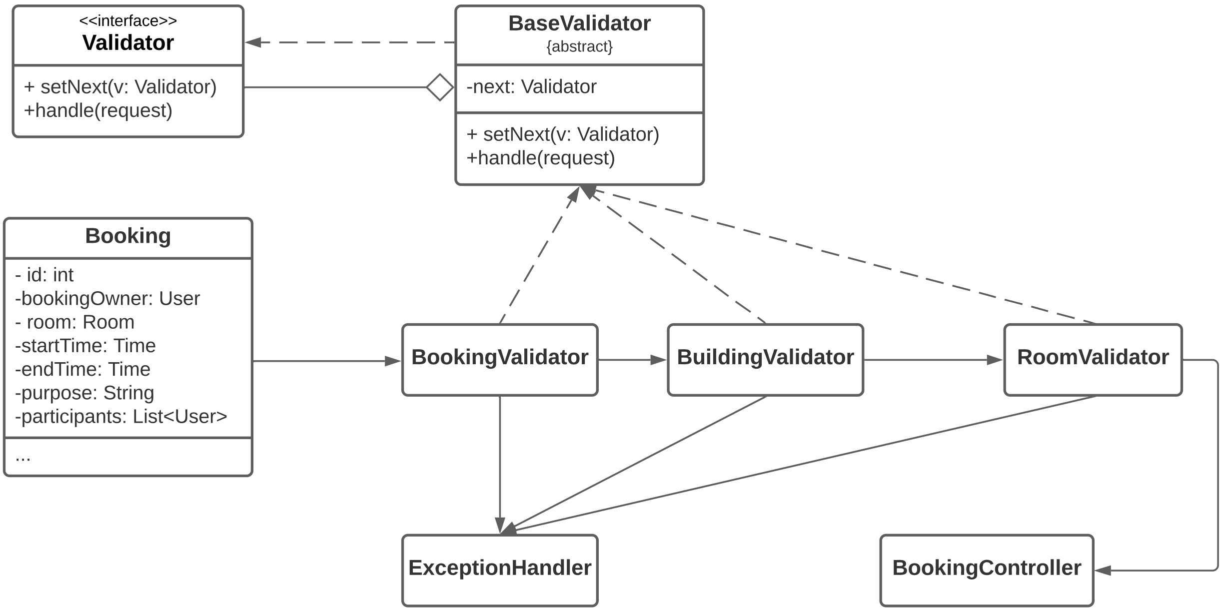

Figure 2.1: Chain of Responsibility Class Diagram

2.1 Introduction

Our system allows a user to make bookings by specifying details such as the room it will take place in, the purpose, a list of participants, the start and end times. To make sure these bookings are valid, they must undergo a set of checks. For this purpose, we decided to implement these checks via the chain of responsibility design pattern. This pattern is effective in our design architecture, as it ensures the validity of a booking before storing it in the database. This prevents users from accidentally creating an invalid booking as well as unnecessary database storage.

2.2 Implementation

This design pattern makes use of three handlers to check the validity of a booking, in the following order:

- BookingValidator: Checks if the date and time of the booking are not before the current date and time and if the start time is not after the end time. Also checks if the specified room and building exist. If all these conditions are met, the next validator is called. Otherwise, the chain breaks and an exception is thrown, resulting in the booking being discarded.
- BuildingValidator: Checks if the building is open during the specified times. If it is, the next validator is called. Otherwise, the chain breaks and an exception is thrown, resulting in the booking being discarded.
- RoomValidator: Checks if the specified room is not occupied during the specified times. If this last check passes, it means the user input is valid, so a new booking is created and stored. Otherwise, an exception is thrown and the booking is discarded.

After the user makes a booking, the input is first passed through all validators. If all the checks are successful, the booking is created and stored in the database. Otherwise, if any of the three handlers fails, the booking is discarded and an exception is thrown, letting the user know what went wrong.

2.3 Possible extensions

In the future, if there would be more constraints on creating a booking, those could be easily added as checks in one of the validators or new handlers could be created and added to the chain to ensure a booking meets the requirements before it is stored.

3 Appendix A

```java
public class Schedule {

    List<Booking> bookings;
    SortStrategy sortStrategy;

    public Schedule(List<Booking> bookings, SortStrategy sortStrategy) {
        this.bookings = bookings;
        this.sortStrategy = sortStrategy;
    }

    public Schedule(SortStrategy sortStrategy) {
        bookings = new ArrayList<>();
        this.sortStrategy = sortStrategy;
    }

    public void addBooking(Booking booking) {
        bookings.add(booking);
    }

    public List<Booking> sortBookings() {
        return this.sortStrategy.sortBookings(bookings);
    }
```

Figure 3.1: Schedule class

```java
public interface SortStrategy {

    List<Booking> sortBookings(List<Booking> bookings);

}
```

Figure 3.2: Sort Strategy Interface

```java
public class ChronologicalSortStrategy implements SortStrategy {

    @Override
    public List<Booking> sortBookings(List<Booking> bookings) {
        bookings.sort(new DateComparator());
        return bookings;
    }

    protected class DateComparator implements Comparator {
        @Override
        public int compare(Object o1, Object o2) {
            Booking b1 = (Booking) o1;
            Booking b2 = (Booking) o2;
            LocalDate dateB1 = b1.getDate();
            LocalDate dateB2 = b2.getDate();

            if (dateB1.isBefore(dateB2)) {
                return -1;
            }
            if (dateB1.isAfter(dateB2)) {
                return 1;
            } else if (dateB1.isEqual(dateB2)) {
                LocalTime timeB1 = b1.getStartTime();
                LocalTime timeB2 = b2.getStartTime();
                if (timeB1.isBefore(timeB2)) {
                    return -1;
                }
                if (timeB1.isAfter(timeB2)) {
                    return 1;
                }
            }
            return 0;
        }
    }
}
```

Figure 3.3: Chronological Strategy

```java
public class LocationStrategy implements SortStrategy {

    @Override
    public List<Booking> sortBookings(List<Booking> bookings) {
        bookings.sort(new LocationComparator());
        return bookings;
    }

    protected class LocationComparator implements Comparator {

        @Override
        public int compare(Object o1, Object o2) {
            Booking b1 = (Booking) o1;
            Booking b2 = (Booking) o2;

            int building1 = b1.getBuilding();
            int building2 = b2.getBuilding();

            if (building1 < building2) {
                return -1;
            }
            if (building2 < building1) {
                return 1;
            }
            if (building1 == building2) {
                int room1 = b1.getRoom();
                int room2 = b2.getRoom();

                if (room1 < room2) {
                    return -1;
                }
                if (room2 < room1) {
                    return 1;
                }
            }
            return 0;
        }
    }
}
```

Figure 3.4: Location Strategy

4 Appendix B

```java
public interface Validator {

    void setNext(Validator handler);

    boolean handle(Booking booking) throws InvalidBookingException,
        InvalidRoomException, BuildingNotOpenException;
}
```

Figure 4.1: validator

```java
public abstract class BaseValidator implements Validator {

    private transient Validator next;

    public void setNext(Validator validator) { this.next = validator; }

    protected boolean checkNext(Booking booking) throws InvalidBookingException,
        InvalidRoomException, BuildingNotOpenException {
        if (next == null) {
            return true;
        }
        return next.handle(booking);
    }


}
```

Figure 4.2: baseValidator

```java
public class BookingValidator extends BaseValidator {

    private transient BuildingController buildingController = new BuildingController();
    private transient RoomController roomController = new RoomController();

    @Override
    public boolean handle(Booking booking) throws InvalidBookingException,
        InvalidRoomException, BuildingNotOpenException {

        if (booking.getDate().compareTo(LocalDate.now()) < 0) {
            throw new InvalidBookingException("Date of booking is in the past");
        } else if (booking.getDate().compareTo(LocalDate.now()) == 0
            && booking.getStartTime().compareTo(LocalTime.now()) <= 0) {
            throw  new InvalidBookingException("Booking start time is before current time");
        } else if (buildingController.getBuilding(booking.getBuilding()) == null) {
            throw new InvalidBookingException("Building does not exist");
        } else if (roomController.getRoom(booking.getRoom()) == null) {
            throw new InvalidBookingException("Room does not exist");
        } else if(booking.getStartTime().compareTo(booking.getEndTime()) >= 0) {
            throw new InvalidBookingException("Start time is after end time");
        }

        return super.checkNext(booking);
    }
}
```

Figure 4.3: bookingValidator

```java
public class BuildingValidator extends BaseValidator {

    private transient BuildingController buildingController = new BuildingController();

    @Override
    public boolean handle(Booking booking) throws BuildingNotOpenException,
        InvalidBookingException, InvalidRoomException {
        Building building = buildingController.getBuilding(booking.getBuilding());
        if (booking.getStartTime().compareTo(building.getOpeningTime()) < 0
            || booking.getEndTime().compareTo(building.getClosingTime()) > 0) {
            throw new BuildingNotOpenException("Building is not open during this interval");
        }
        return super.checkNext(booking);
    }
}
```

Figure 4.4: buildingValidator

```java
public class RoomValidator extends BaseValidator {

    private transient BookingController bookingController = new BookingController();

    /**
     * Method for checking if two bookings overlap.
     *
     * @param booking the booking we check for validity
     * @param other a booking from the database
     * @return true if the bookings overlap, false otherwise
     */
    public boolean bookingsOverlap(Booking booking, Booking other) {
        if (booking.getRoom() == other.getRoom()
            && booking.getDate().equals(other.getDate())) {
            if ((booking.getEndTime().compareTo(other.getStartTime()) >= 0
                && booking.getEndTime().compareTo(other.getEndTime()) > 0)
                || (booking.getStartTime().compareTo(other.getStartTime()) >= 0
                && booking.getStartTime().compareTo(other.getEndTime()) < 0)) {
                return true;
            }
        }
        return false;
    }

    @Override
    public boolean handle(Booking newBooking) throws InvalidRoomException,
        InvalidBookingException, BuildingNotOpenException {
        List<Booking> bookings = bookingController.getBookings();
        for (Booking booking : bookings) {
            if (bookingsOverlap(newBooking, booking)) {
                throw new InvalidRoomException("The room is not available during this interval");
            }
        }
        return super.checkNext(newBooking);
    }

}
```

Figure 4.5: roomValidator