Task: Object-Oriented Thinking
Done by: Nazarii Montytskyi
Chosen topic: Dormitory accommodation management
GitHub Repository: https://github.com/NazaryiMontytskyi/dormitory-management-system
Code Documentation: https://nazaryimontytskyi.github.io/dormitory-management-system/

Part 1. Functional and non-functional requirements for the chosen system

Functional requirements — the requirements which describe what exactly the system must do. In these requirements detailed functions, abilities or behaviours expand so that the developer can understand them clearly.

Functional requirements for the Dormitory accomodation management system

ID	FR-001
Name	Booking of an accommodation
Description	System must provide the user an ability to book a dormitory accomodation
Acceptance criteria	1. User provides the data about his real name, phone number (in correct format) and e-mail (in correct format) 2. User selects one of the available accomodations in the dormitory from the list displayed in the system 3. User receives the email notification about successful booking 4. If there are no available accomodations to book user must see this in the list with no available unit 5. User pays for the booking using the banking system 6. User clicks the "Book" button to book an accomodation

ID	FR-002
Name	Declining of booking
Description	System must allow user to cancel his existing booking within 24 hours
Acceptance criteria	1. User must have an existing booking, displayed in its personal account 2. User click the "Cancel" button in the list of its existing bookings in order to cancel it 3. User receives a confirmation notification to its email 4. The accomodation recently booked by the user becomes available again

ID	FR-003
Name	Editing a booking
Description	System must allow user to edit its booking
Acceptance criteria	1. User can edit an existing booking which it has 2. System must balance (calculate and back to user a rest or get extra cash from user) the difference in price between the previous and current state of booking 3. User confirms changes clicking the “Edit” button 4. User receives an email notification

ID	FR-004
Name	Payment for booking
Description	System must allow user to pay for its booking using credit card
Acceptance criteria	1. User inputs its credentials of credit card in valid format (number of card, CVV code and the term) 2. User receives a confirmation message about successful transaction to his email 3. The booking changes its status to “Paid” after the payment

ID	FR-005
Name	Supplementation of booking
Description	System must allow user to add available dormitory services to its booking
Acceptance criteria	1. User can choose available dormitory services with checkbox which will provided with the accomodation together 2. The price for booking changes according to chosen services within

ID	FR-006
Name	Creation of bookings
Description	System must allow the administrator to create a booking manualy
Acceptance criteria	1. There are an available accommodations in the system 2. Administrator creates the booking manualy inputing the user personal data, etc. (for example, speaking with user by mobile phone) 3. The booking receives a status “Booked” but not “Paid” 4. User receives the appropriate notification on its email address within a bill it must to pay

ID	FR-007
Name	Administrative cancelation of bookings
Description	System must provide the administrator to cancel existing booking
Acceptance criteria	1. There are an existing booking in the system 2. Administrator clicks the “Cancel” next to element of the list (which represents the existing booking) in order to cancel a booking 3. User receives the notification on its email about the declining of its booking

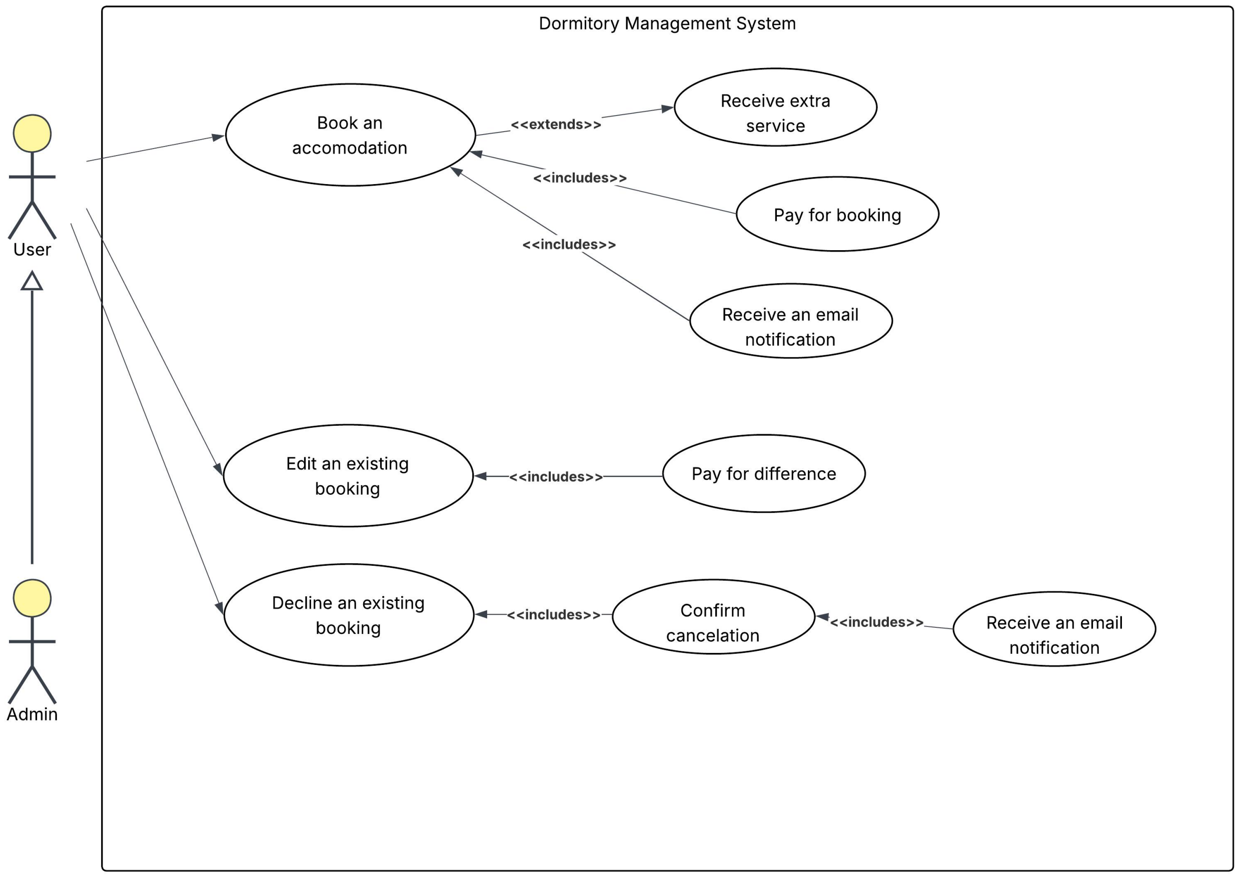
ID	FR-008
Name	Administrative editing of bookings
Description	System must provide the administrator to edit existing booking
Acceptance criteria	1. Administrator changes the booking existing in system (for example, on user’s request) 2. User receives the email notification about changes 3. User pays the difference in price if required

ID	FR-009
Name	Freeing up the accomodation
Description	The accomodation must obtain the “Available” status after the user’s guesting is over
Acceptance criteria	1. System automatically change the status of the accomodation to “Available” after the user’s period of guesting is over 2. User receives the letter with grateful regards of being a guest in the dormitory

Non-functional requirements — the requirements which describe the way system should follow. They answers us the question “How?”.

ID	**Non-functional requirement**
NFR-001	The web-pages loads less than 4 seconds in conditions of full workload
NFR-002	The system handles its work with 100 users at the time
NFR-003	The system works correctly for Android 11+ and iOS 15+
NFR-004	Users finish their tasks in the system less than 2 minutes
NFR-005	The system protects user data with the AES-256 protocol

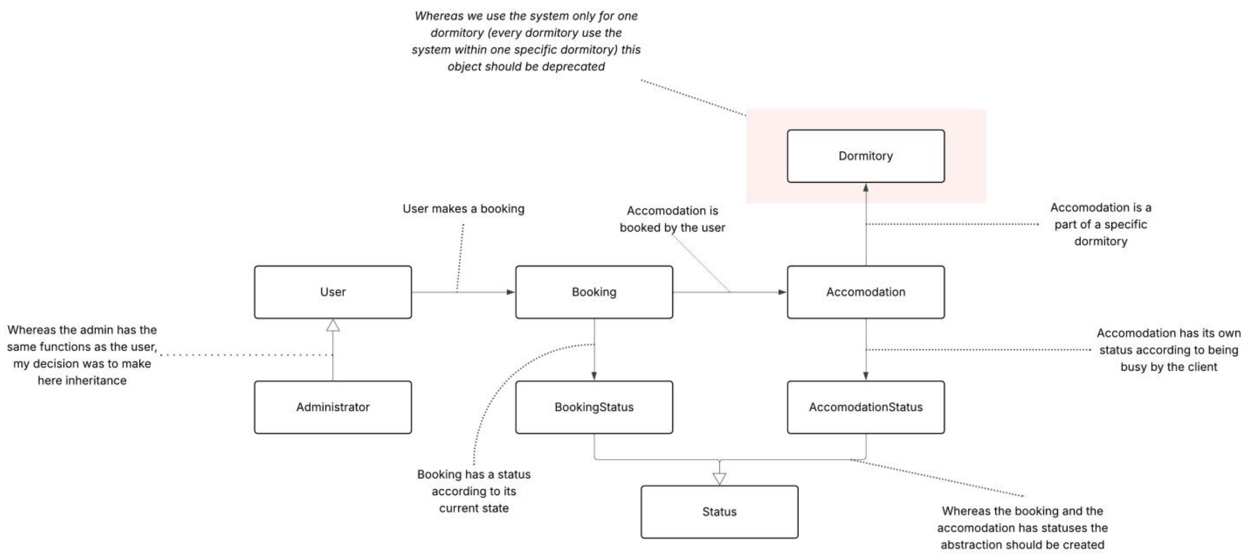
Part 2. Design use cases for the system based on the requirements



Explanation: As you can see the Admin actor hasn't any arrows going to any of the present use cases. Why did I do so? Both of the actors in the system can do the same actions. User can book an accomodation on his own, so the admin can do the same for the user. We have the same action but with different behaviour. So I decided not to duplicate the use cases but to show the admin inherits those functionalities from the user.

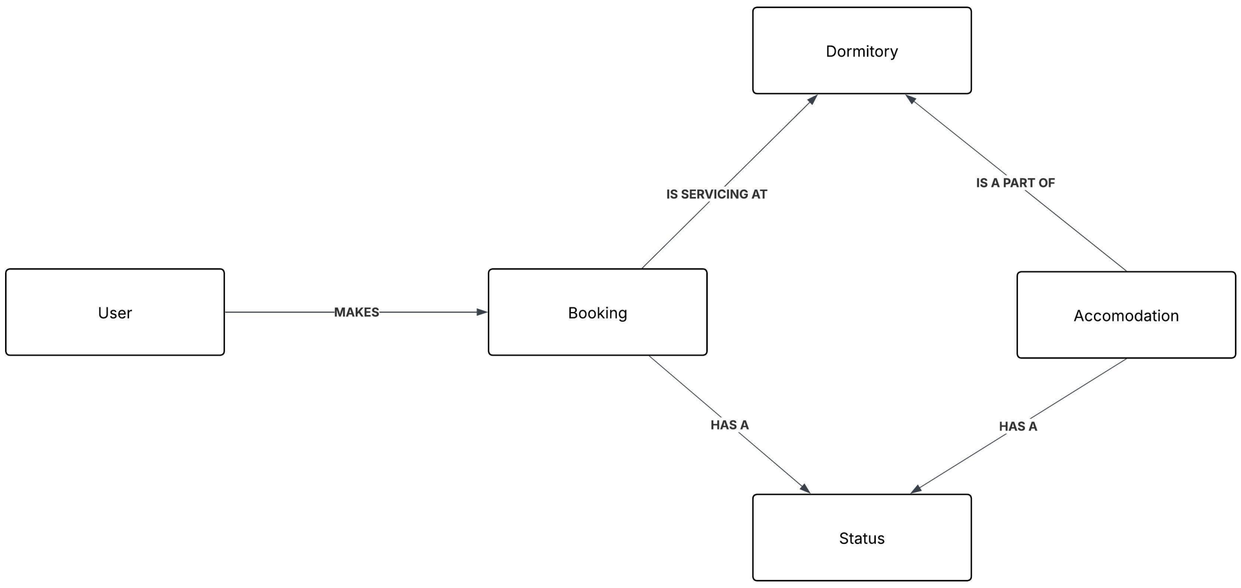
If we ask the question "Is an admin an user?" in order to build "is a" relationship, we answer definetely "Yes". We can't do the same in the reverse order. The user isn't the admin. So we can inherit the admin from the user. I decided to use DRY principle here as a guidence. Yes, the UML-diagram isn't the same as the code does. But I believe that this principle also suits here.

Part 3. Identify the objects, classes and relationships in the system


Whereas we use the system only for one dormitory (every dormitory use the system within one specific dormitory) this object should be deprecated
Dormitory
User makes a booking
Accomodation is booked by the user
Accomodation is a part of a specific dormitory
User
Booking
Accomodation
Whereas the admin has the same functions as the user, my decision was to make here inheritance
Accomodation has its own status according to being busy by the client
Administrator
BookingStatus
AccomodationStatus
Booking has a status according to its current state
Status
Whereas the booking and the accomodation has statuses the abstraction should be created

On the picture above you can see my ideas about the objects in the system. But I decided to make this chart easier. First of all, I removed Dormitory object, because it isn't necessary. Every dormitory applies the system where there aren't any other connections with other dormitories.

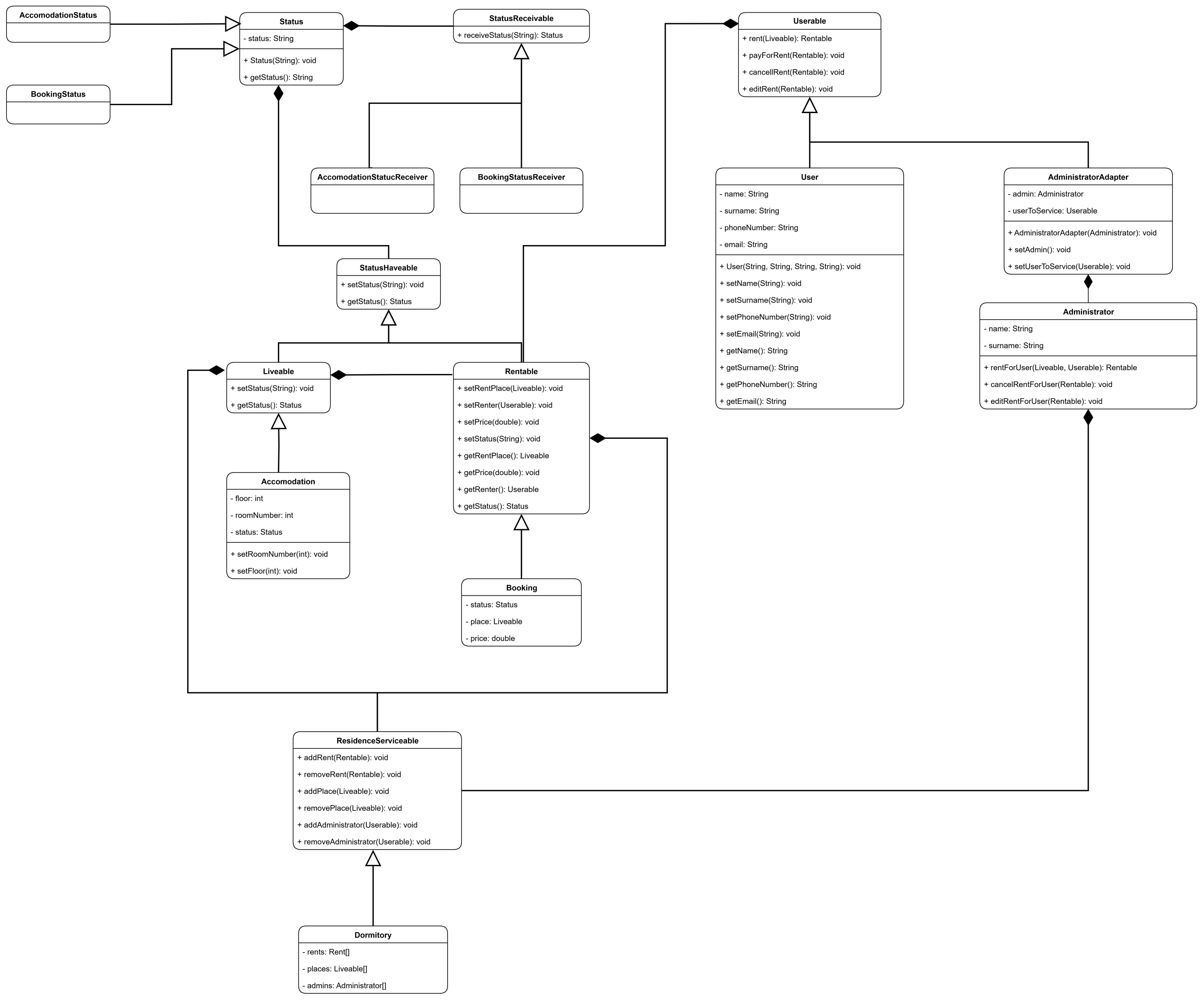
You can see here different statuses for booking and accomodations. We can just remove those two blocks and replace by the single status block. To split the logic for two different statuses types we just can use Factory pattern to receive the status we actually need for the exact object.



So, for now we have a single user. The admin is still present as an actor in the system but inherits the behaviour from user. Both the user and the admin can do the same actions but with the different behaviour. Booking and accommodation has some of the similarities. Booking has its own status depending on the user's steps in completing it. The accomodation status depends on the user's occupation of it. A dormitory here is used as an aggregation object which consist the accomodations (a set of accomodations) and the bookings (a set of bookings) which are serviced to users in this residence.

So in the next part you can see the class diagram

Part 4. Design class diagrams picturing classes, their attributes and relations in the system


AccomodationStatus
BookingStatus
Status
- status: String
+ Status(String): void
+ getStatus(): String
StatusReceivable
+ receiveStatus(String): Status
AccomodationStatucReceiver
BookingStatusReceiver
Userable
+ rent(Liveable): Rentable
+ payForRent(Rentable): void
+ cancellRent(Rentable): void
+ editRent(Rentable): void
User
- name: String
- surname: String
- phoneNumber: String
- email: String
+ User(String, String, String, String): void
+ setName(String): void
+ setSurname(String): void
+ setPhoneNumber(String): void
+ setEmail(String): void
+ getName(): String
+ getSurname(): String
+ getPhoneNumber(): String
+ getEmail(): String
AdministratorAdapter
- admin: Administrator
- userToService: Userable
+ AdministratorAdapter(Administrator): void
+ setAdmin(): void
+ setUserToService(Userable): void
Administrator
- name: String
- surname: String
+ rentForUser(Liveable, Userable): Rentable
+ cancelRentForUser(Rentable): void
+ editRentForUser(Rentable): void
StatusHaveable
+ setStatus(String): void
+ getStatus(): Status
Liveable
+ setStatus(String): void
+ getStatus(): Status
Rentable
+ setRentPlace(Liveable): void
+ setRenter(Userable): void
+ setPrice(double): void
+ setStatus(String): void
+ getRentPlace(): Liveable
+ getPrice(double): void
+ getRenter(): Userable
+ getStatus(): Status
Accomodation
- floor: int
- roomNumber: int
- status: Status
+ setRoomNumber(int): void
+ setFloor(int): void
Booking
- status: Status
- place: Liveable
- price: double
ResidenceServiceable
+ addRent(Rentable): void
+ removeRent(Rentable): void
+ addPlace(Liveable): void
+ removePlace(Liveable): void
+ addAdministrator(Userable): void
+ removeAdministrator(Userable): void
Dormitory
- rents: Rent[]
- places: Liveable[]
- admins: Administrator[]

EXPLANATION

As you can see the class diargam is so huge for the task and includes 17 classes and interfaces. When I maked those diagram I was afraid about overengineering. I also tried following all the principles mastered in the course.

By the way, I can see the violation some of the SOLID principles in my own design. So for now I will explain the diagram and then I will emphasize some of my mistakes which I admit.

Interface **Userable** is responsible to provide the logic of the user who will be serviced in the dormitory. It contains four methods which class **User** should implement. As you can see the User class has some additional fields which describes the nature of this entity.

You can see the **Administrator** class which separated from the **Userable** interface and the Userable class. What is the reason for a such decision? Yes, both the administrator and the user do the same actions. But they do so in the different ways. So the administrator need some modified methods structures than the Userable interface has. Here we can use the Adapter pattern in order to adapt administrator's actions to the Userable requirements. For that I used the **AdministratorAdapter** class.

Here is a remarkable class called **Status**. This class is used by two other entities defined in our system design: **Accomodation** and **Booking**. As we set in the functional requirements both bookings and accomodations can change their statuses depending on user's actions. So the goal of this class is to keep the value of the status. Classes **AccomodationStatus** and **BookingStatus** are responsible for defining statuses both for the accomodations and bookings.

I decided to use **the Abstract Factory** pattern to receive the statuses. Why I did so? Our statuses varies, so as we learned in the course, we should incapsulate those parts. We have the interface which defines the factory of statuses **StatusReceivable**. Also we have two implementations of these class: **AccomodationStatusReceiver** and **BookingStatusReceiver**.

We can see the interface **StatusHaveable** which is extended by the interfaces **Liveable** and **Rentable**. So both the Accomodation (which implements Liveable) and the Booking (which implements Rentable) has statuses. In order not to duplicate the code in the interfaces I decided to make an interface which two other interfaces will extend. I consider this as maintaining the DRY (Don't repeat yourself) principle.

We also have here the **ResidenceServiceable** interface which represents the business logic for the entity which can service the users in the sphere of proposing accomodation for money. This interface is the ground of the **Dormitory** class which aggregates accomodations, bookings and administrators. Accomodations are the parts of the dormitory because it uses them as the service. Bookings are the parts of the dormitory because the bookings are making in the specific dormitory. Administrators are the part of the dormitory because the dormiory needs some management.

You can say me that here are many interfaces designed. You’re right. Maybe they are too many of them. But I decided to maintain the principle that the program should be based on interfaces not on implementations.

Part 5. Java Code for the solution

I highly recommend you to follow the link and go to the GitHub Repository where you can find all of the code for the task.

Also I generated the documentation for code where I described all of the methods in the solution.

But also I will copy and paste the code here.

com.montytskyi.Bookings.Interfaces.Rentable.java

```java
package com.montytskyi.Bookings.Interfaces;

import com.montytskyi.DormitoriesPlaces.Interfaces.Liveable;
import com.montytskyi.Statuses.Implementations.Status;
import com.montytskyi.Statuses.Interfaces.StatusHaveable;
import com.montytskyi.Userables.Intrefaces.Userable;

/**
 * The interface which defines the logic of the place which can be rented by the client
 * @author Nazarii Montytskyi
 * @version 1.0
 */
public interface Rentable extends StatusHaveable{

  /**
   * The method which allows to set the rent place which can be rented by the user
   * @param place - the place which can be rented by the user
   */
  public void setRentPlace(Liveable place);

  /**
   * The method which allows to set the renter which is renting the defined place
   * @param renter - a user or a client who rents the defined place
   */
  public void setRenter(Userable renter);

  /**
   * The method which allows to set the price for the rent
   * @param price - a double value which defines the price for the rent
   */
  public void setPrice(double price);

  /**
   * The method which allows to set the status of the renting
   * @param stringLabel - a value which defines the current status of the renting
   */
  public void setStatus(String stringLabel);

  /**
   * The method which allows to get the instance of rent place
   * @return Liveable rent place
   */
  public Liveable getRentPlace();

  /**
   * The method which allows to get the price for the rent
   * @return price for the rent
   */
  public double getRentPrice();

  /**
   * The method which allows to get the instance of the renter
   * @return Userable renter of the specific place
   */
  public Userable getRenter();

  /**
   * The method which allows to get the value of current renting status
   * @return Status of the current renting
   */
  public Status getStatus();

}
```

```
com.montytskyi.Bookings.Implementations.Booking.java
package com.montytskyi.Bookings.Implementations;

import com.montytskyi.Bookings.Interfaces.Rentable;
import com.montytskyi.DormitoriesPlaces.Interfaces.Liveable;
import com.montytskyi.Statuses.Implementations.BookingStatusReceiver;
import com.montytskyi.Statuses.Implementations.Status;
import com.montytskyi.Userables.Intrefaces.Userable;

/**
 * The class which represents the booking in the dormitory. The class contains the info
about the rented place, the person who rents it, the price and the current status
 * @author Nazarii Montytskyi
 * @version 1.0
 */
public class Booking implements Rentable{

  /** The place which is rented by the client */
  private Liveable place;

  /** The client who rents the place */
  private Userable renter;

  /** The price paid for the rent */
  private double price;

  /** The current status of the renting */
  private Status bookingStatus;

  /** The empty constructor */
  public Booking(){

  }

  /**
   * The constructor with parameters without status
   * @param place - represents the place rented by the user
   * @param renter - the client who rents the place
   * @param price - the price paid for the rent
   */
  public Booking(Liveable place, Userable renter, double price){
    this.place = place;
    this.renter = renter;
    this.price = price;
  }

  /**
   * The constructor with all the parameters possible
   * @param place - represents the place rented by the user
   * @param renter - the client who rents the place
   * @param price - the price paid for the rent
   * @param bookingStatus - the current status of the booking
   */
  public Booking(Liveable place, Userable renter, double price, Status bookingStatus){
    this(place, renter, price);
    this.bookingStatus = bookingStatus;
  }

  /**
   * The setter method to define the place for renting
   * @param place - the place rented by the client
   */
  @Override
  public void setRentPlace(Liveable place) {
    this.place = place;
  }

  /**
   * The setter method to define the person who rents the place
   * @param renter - the client who rents the place
   */
  @Override
  public void setRenter(Userable renter) {
    this.renter = renter;
  }

  /**
   * The setter method which allows to set the price for the rent
   * @param price - the price for the rent
   */
  @Override
  public void setPrice(double price) {
    this.price = price;
  }

  /**
   * The setter method which allows to set the current status for the rent
   * @param stringLabel - the label which contains the information about the status
   */
  @Override
  public void setStatus(String stringLabel) {
    this.bookingStatus = (new BookingStatusReceiver()).receiveStatus(stringLabel);
  }

  /**
   * The getter method that returns the place rented by the user
   * @return place - the place rented by the user
   */
  @Override
  public Liveable getRentPlace() {
    return this.place;
  }
```

```
  /**
   * The getter method that returns the price for the rent
   * @return price - the price for the rent
   */
  @Override
  public double getRentPrice() {
    return this.price;
  }

  /**
   * The getter method that returns the renter who rents the place
   * @return renter - the client who rents the place
   */
  @Override
  public Userable getRenter() {
    return this.renter;
  }

  /**
   * The getter method that returns the status of the current renting
   * @return bookingStatus - the current status of the renting
   */
  @Override
  public Status getStatus() {
    return this.bookingStatus;
  }

}
```

com.montytskyi.DormitoriesPlaces.Interfaces.Liveable.java

```
package com.montytskyi.DormitoriesPlaces.Interfaces;

import com.montytskyi.Statuses.Interfaces.StatusHaveable;

/**
 * The interface which represents the place that can be rented by some client
 * @author Nazarii Montytskyi
 * @version 1.0
 */
public interface Liveable extends StatusHaveable {

}
```

com.montytskyi.DormitoriesPlaces.Implementations.Accomodation.java

```
package com.montytskyi.DormitoriesPlaces.Implementations;

import com.montytskyi.DormitoriesPlaces.Interfaces.Liveable;
import com.montytskyi.Statuses.Implementations.AccomodationStatusReceiver;
import com.montytskyi.Statuses.Implementations.Status;

/**
 * The class which implements Liveable interface and represents an accomodation which can
be rented by some client
 * @author Nazarii Montytskyi
 * @version 1.0
 */
public class Accomodation implements Liveable {
  /** The integer field which represents the floor on which the accommodation is located
*/
  private int floor;

  /** The integer field which represents the number of room */
  private int roomNumber;

  /** The value which represents the current status of an accommodation */
  private Status status;

  /**
   * The constructor with all the parameters possible
   * @param floor - integer value which represents the floor where the room is located
   * @param roomNumber - integer value which represents the number of room
   * @param status - the value which represents the current status of an accommodation
   */
  public Accomodation(final int floor, final int roomNumber, final Status status){
    this.floor = floor;
    this.roomNumber = roomNumber;
    this.status = status;
  }

  /**
   * The constructor with parameter which identifies the room's coordinates
   * @param floor - integer value which represents the floor where the room is located
   * @param roomNumber - integer value which represents the number of room
   */
  public Accomodation(final int floor, final int roomNumber){
    this.floor = floor;
    this.roomNumber = roomNumber;
    this.status = (new AccomodationStatusReceiver()).receiveStatus("FREE");
  }

  /**
   * Am empty class constructor
   */
  public Accomodation(){

  }

  /**
   * The setter method which is implemented from the interface and provides the ability
to set the status
   * @param statusLabel - string which contains the current status of the accommodation
   */
  @Override
  public void setStatus(String statusLabel) {
    this.status = (new AccomodationStatusReceiver()).receiveStatus(statusLabel);
  }

  /**
   * The getter method which is implemented from the interface and provides the ability
to get the current accomodation status
   * @return status - the current status of an accommodation
   */
  @Override
  public Status getStatus() {
    return this.status;
  }

  /**
```

```
     * The getter method which return the floor number
     * @return floor - the floor where the room is located
     */
    public int getFloor() {
      return floor;
    }

    /**
     * The setter method which allows to set the floor of the room
     * @param floor - the floor where the room is located
     */
    public void setFloor(int floor) {
      this.floor = floor;
    }

    /**
     * The getter method which allows to receive the number of the room
     * @return roomNumber - the value which contains the number of the room
     */
    public int getRoomNumber() {
      return roomNumber;
    }

    /**
     * The setter method which allows to set the number of the room
     * @param roomNumber - the value which contains the number of the room
     */
    public void setRoomNumber(int roomNumber) {
      this.roomNumber = roomNumber;
    }

  }
```

```
com.montytskyi.Residences.Implementations.Dormitory.java
package com.montytskyi.Residences.Implementations;

import java.util.List;

import com.montytskyi.Bookings.Interfaces.Rentable;
import com.montytskyi.DormitoriesPlaces.Interfaces.Liveable;
import com.montytskyi.Residences.Interfaces.ResidenceServiceable;
import com.montytskyi.Userables.Implementations.Administrator;

/**
 * The class which represents the Dormitory which will be managed by the system. The
Dormitory class is implemented from the ResidenceServiceable interface.
 * @author Nazarii Montytskyi
 * @version 1.0
 */
public class Dormitory implements ResidenceServiceable{

  /**
   * A constructor that receives all the parameters possible
   * @param rents - the collection of rents which is in charge of containing the
dormitory rents
   * @param admins - the collection of administrators which is responsible of containing
the dormitory administrators
   * @param places - the collection of places which is responsible of containing all the
dormitory renting places
   */
  public Dormitory(List<Rentable> rents, List<Administrator> admins, List<Liveable>
places) {
    this.rents = rents;
    this.admins = admins;
    this.places = places;
  }

  /**
   * An empty constructor of the Dormitory class
   */
  public Dormitory() {
  }

  /** A field which is responsible of containing the Dormitory rents */
  private List<Rentable> rents;

  /**
   * A field which is repsonsible of containing the Dormitory administrators
   */
  private List<Administrator> admins;

  /**
   * A filed which is responsible of containing the Dormitory rentable places
   */
  private List<Liveable> places;

  /**
   * An implemented method from the interface ResidenceServiceable which allows to add a
rent to the dormitory
   * @param rent - represents a rent made by an user
   */
  @Override
  public void addRent(Rentable rent) {
    this.rents.add(rent);
  }

  /**
   * An implemented method from the interface ResidenceServiceable which allows to remove
an existing rent from the dormitory
   * @param rent - represent a rent made by an user
   */
  @Override
  public void removeRent(Rentable rent) {
    this.rents.remove(rent);
  }

  /**
   * An implemented method from the interface ResidenceServiceable which allows to add a
rentable place to the dormitory
   * @param place - represents a place which is able to be rented by an user
   */
  @Override
  public void addPlace(Liveable place) {
    this.places.add(place);
  }

  /**
   * An implemented method from the interface ResidenceServiceable which allows to add
remove an existing in the dormitory place
   * @param place - represents a place which is able to be rented by an user
   */
  @Override
  public void removePlace(Liveable place) {
    this.places.remove(place);
  }

  /**
   * An implemented method from the interface ResidenceServiceable which allows to add an
administrator for the specific Dormitory
   */
  @Override
  public void addAdministrator(Administrator admin) {
    this.admins.add(admin);
  }

  /**
   * An implemented method from the interface ResidenceServiceable which allows to remove
an administrator from the specific Dormitory
   */
  @Override
  public void removeAdministrator(Administrator admin) {
    this.admins.remove(admin);
  }

  /**
   * A getter method which returns the list of existing rents in the dormitory
   * @return rents - list of existing rents in the dormitory
   */
  public List<Rentable> getRents() {
    return rents;
  }

  /**
   * A setter method which sets the collection of rents which will be contained in the
Dormitory
   * @param rents - list of rents in the dormitory
   */
  public void setRents(List<Rentable> rents) {
    this.rents = rents;
  }

  /**
   * A getter method which allows to get the collection of the administrators
   * @return admins - the list of existing administrator in the dormitory
   */
  public List<Administrator> getAdmins() {
    return admins;
  }

  /**
   * A setter method which allows to set the collection of the administrators for the
dormitory
   * @param admins - the list of administrators of the specific dormitory
   */
  public void setAdmins(List<Administrator> admins) {
    this.admins = admins;
  }

  /**
   * A getter method which return the list of places which are the part of the dormitory
   * @return places - the list of places in the dormitory which can be rented by some
client
```

```
   */
  public List<Liveable> getPlaces() {
    return places;
  }

  /**
   * A setter method which sets the list of places which are the part of the dormitory
   * @param places - the list of places in the dormitory which can be rented by some
client
   */
  public void setPlaces(List<Liveable> places) {
    this.places = places;
  }

}
```

```
com.montytskyi.Residences.Interfaces.ResidenceServiceable.java
package com.montytskyi.Residences.Interfaces;

import com.montytskyi.Bookings.Interfaces.Rentable;
import com.montytskyi.DormitoriesPlaces.Interfaces.Liveable;
import com.montytskyi.Userables.Implementations.Administrator;

/**
 * The interface which represents an instution which can be user for the renting
dormitories or accommodations
 * @author Nazarii Montytskyi
 * @version 1.0
 */
public interface ResidenceServiceable {

  /**
   * The method which allows to add the created rent to the specific residence
   * @param rent - a renting offer created by the specific client
   */
  public void addRent(Rentable rent);

  /**
   * The method which provides the ability to remove the rent from the specific residence
   * @param rent - a renting offer that should be removed from the dormitory
   */
  public void removeRent(Rentable rent);

  /**
   * The method which provides the ability to add an existing or recently created place
which is able to be rented to the specific residence
   * @param place - the place that will be added to an specific residence
   */
  public void addPlace(Liveable place);

  /**
   * The method that allows to remove the place from an specific residence
   * @param place - the place which is about to be removed from the specific residence
   */
  public void removePlace(Liveable place);

  /**
   * The method that allows to add an administrator for the specific residence
   * @param admin - the administrator who will be in charge of the specific residence
   */
  public void addAdministrator(Administrator admin);

  /**
   * The method that allows to remove an existing the administrator from the specific
residence
   * @param admin - the administrator that should be removed from the specific residence
   */
  public void removeAdministrator(Administrator admin);

}
```

```
com.montytskyi.Statuses.Implementations.AccomodationStatus.java
package com.montytskyi.Statuses.Implementations;

/**
 * The class which represents the status of an accommodation which can be rented by the
user
 * @author Nazarii Montytskyi
 * @version 1.0
 */
public class AccomodationStatus extends Status{

  /**
   * The constructor of the class which allows to pass the status of an accommodation
straightforwardly
   * @param accomodationStatus - which contains the information about the accomodation
status
   */
  public AccomodationStatus(String accomodationStatus){
    super(accomodationStatus);
  }
}
```

```
com.montytskyi.Statuses.Implementations.AccomodationStatusReceiver.java
package com.montytskyi.Statuses.Implementations;

import com.montytskyi.Statuses.Interfaces.StatusReceivable;

/**
 * A class which responsibility is to return the status of an accomodation according to
its occupation by the client
 * @author Nazarii Montytskyi
 * @version 1.0
 */
public class AccomodationStatusReceiver implements StatusReceivable {

  /**
   * The method which is implemented from the interface StatusReceivable. Method provides
the status of an accomodation
   * @param statusLabel - which contains the info about the status of an acommodation
   * @throws IllegalArgumentException
   * @return AccomodationStatus which can be presented as value of "FREE" or "BUSY"
   */
  @Override
  public Status receiveStatus(String statusLabel){
    switch(statusLabel){
      case "FREE":
        return new AccomodationStatus("FREE");
      case "BUSY":
        return new AccomodationStatus("BUSY");
      default:
        throw new IllegalArgumentException("The status for accomodation must have value
'FREE' or 'BUSY");
    }
  }
}


com.montytskyi.Statuses.Implementations.BookingStatus.java
package com.montytskyi.Statuses.Implementations;

/**
 * The class which represents the status of an booking made by user
 * @author Nazarii Montytskyi
 * @version 1.0
 */
public class BookingStatus extends Status{

  /**
   * The constructor which allows to set the booking status
   * @param bookingStatus - which contains the info about status of a booking which is
about to be created
   */
  public BookingStatus(String bookingStatus){
    super(bookingStatus);
  }
}


com.montytskyi.Statuses.Implementations.BookingStatusReceiver.java
package com.montytskyi.Statuses.Implementations;

import com.montytskyi.Statuses.Interfaces.StatusReceivable;

/**
 * An implementation class of the StatusReceivable interface which allows to create a
status for the booking made by the user
 * @author Nazarii Montytskyi
 * @version 1.0
 */
public class BookingStatusReceiver implements StatusReceivable{

  /**
   * A method implemented from the interface StatusReceivable which allows to receive a
specific status of a user's booking
   * @param statusLabel - a string which defines a status of a booking
   * @throws IllegalArgumentException
   * @return Booking Status of an existing user's booking
   */
  @Override
  public Status receiveStatus(String statusLabel) {
    switch (statusLabel) {
      case "NOT BOOKED":
        return new BookingStatus("NOT BOOKED");
      case "BOOKED":
        return new BookingStatus("BOOKED");
      case "PAIED":
        return new BookingStatus("PAIED");
      case "FINISHED":
        return new BookingStatus("FINISHED");
      default:
        throw new IllegalArgumentException("Booking status should receive one of the
three its statuses: 'NOT BOOKED', 'BOOKED', 'PAIED'");
    }
  }

}
```

```
com.montytskyi.Statuses.Implementations.Status.java
package com.montytskyi.Statuses.Implementations;

/**
 * The class which represents a specific status for the entities which require it
 * @author Nazarii Montytskyi
 * @version 1.0
 */
public class Status {

  /** A field which defines a status */
  private String status;

  /**
   * The constructor which allows to set a status strightforwardly
   * @param status - which is represented by the string which contains it
   */
  public Status(String status){
    this.status = status;
  }

  /**
   * The getter method of a class which allows to receive a current status
   * @return status
   */
  public String getStatus(){
    return this.status;
  }
}
```

```
com.montytskyi.Statuses.Interfaces.StatusHaveable.java
package com.montytskyi.Statuses.Interfaces;

import com.montytskyi.Statuses.Implementations.Status;

/**
 * The specific interface which allows some classes to have a status defined in the class
Status
 * @version 1.0
 * @author Nazarii Montytskyi
 */
public interface StatusHaveable {

  /**
   * A method which allows to set a specific status
   * @param statusLabel - which specifies a status
   */
  public void setStatus(String statusLabel);

  /**
```

```
   * A method which allows to get a current status
   * @return status, which is contained by the implementation class
   */
  public Status getStatus();

}


com.montytskyi.Statuses.Interfaces.StatusReceivable.java
package com.montytskyi.Statuses.Interfaces;

import com.montytskyi.Statuses.Implementations.Status;

/**
 * The interface which defines the implementations which has some status or requires it
 * @author Nazarii Montytskyi
 * @version 1.0
 */
public interface StatusReceivable {

  /**
   * The method which returns a specific status according to a label which provides an
ability to define those status
   * @param statusLabel - which contains label, which allows to define the status
   * @return Status, which is defined by the method
   */
  public Status receiveStatus(String statusLabel);

}


com.montytskyi.Userables.Interfaces.Userable.java
package com.montytskyi.Userables.Intrefaces;

import com.montytskyi.Bookings.Interfaces.Rentable;
import com.montytskyi.DormitoriesPlaces.Interfaces.Liveable;

/**
 * An interface that defines common methods to manage the users which will be use the
dormitory management system
 * @author Nazarii Montytskyi
 * @version 1.0
 */
public interface Userable {

  /**
   * @param place - the place which will be rented by the client
   * @return Rentable object that contains the information about the rent
   * @see com.montytskyi.Bookings.Interfaces.Rentable
   */
  public Rentable rent(Liveable place);

  /**
   * The method is responsible to provide the user an ability to pay the rent
   * @param currentRent - which has been already done by the user recently
   */
  public void payForRent(Rentable currentRent);

  /**
   * The method is responsible to cancel existing user's rent
   * @param currentRent - which should be cancelled
   */
  public void cancelRent(Rentable currentRent);

  /**
   * The method is responsible to edit the rent which was recently created by the user
   * @param currentRent - which should be edited
   */
  public void editRent(Rentable currentRent);

}
```

```
com.montytskyi.Userables.Implementations.Administrator.java
package com.montytskyi.Userables.Implementations;

import com.montytskyi.Bookings.Implementations.Booking;
import com.montytskyi.Bookings.Interfaces.Rentable;
import com.montytskyi.DormitoriesPlaces.Interfaces.Liveable;
import com.montytskyi.Userables.Intrefaces.Userable;

/**
 * The class which represents the manager of the dormitory management systems
 * @version 1.0
 * @author Nazarii Montytskyi
 */
public class Administrator {

  /** The name of the administrator */
  private String name;

  /** The surname of the administrator */
  private String surname;

  /**
   * An empty constructor of a class
   */
  public Administrator(){

  }

  /**
   * A constructor with all of the parameters possible which includes name and surname
   * @param name - which represents the administrator's name
   * @param surname - which represents the administrator's surname
   */
  public Administrator(String name, String surname){
    this.name = name;
    this.surname = surname;
  }

  /**
   * A method which provides the administrator an ability to make a new rent for an
existing user
   * @param place - which will be rented by a specific user
   * @param renter - which is a user who rents a specific place
   * @return Rentable, which represents a rent created by some user
   */
  public Rentable rentForUser(Liveable place, Userable renter){
    System.out.println("Administrator books a place for a user");
    return new Booking(place, renter, 200);
  }
  /**
   * A method which provides the administrator with an ability to cancel an existing rent
of an existing user
   * @param rent - represents a renting offer which recently was created by an user
   */
  public void cancelRentForUser(Rentable rent){
    System.out.println("Admin cancels the rent for the user");
    rent.setStatus("FINISHED");
  }

  /**
   * A method which provides the administrator with an ability to edit an existing rent
of an existing user
   * @param rent - which represents an existing user's rent
   */
  public void editRentForUser(Rentable rent){
    System.out.println("Admin changes rent for user");
  }

  /**
   * A getter method of the class which allows to receive a name of the administrator
   * @return name, String value which contains the name of the administrator
   */
  public String getName() {
    return name;
  }

  /**
   * A setter method of the class which allows to set a name for the specific
administrator
   * @param name - a name which will be set for the administrator
   */
  public void setName(String name) {
    this.name = name;
  }

  /**
   * A getter method of the class which allows to receive a surname of the specific
administrator
   * @return surname, the surname of the specific administrator
   */
  public String getSurname() {
    return surname;
  }

  /**
   * A setter method of the class which allows to set a surname for the specific
administrator
   * @param surname - which contains the surname of an administrator
   */
  public void setSurname(String surname) {
    this.surname = surname;
  }
}
```

```
com.montytskyi.Userables.Implementations.AdministratorAdapter.java
package com.montytskyi.Userables.Implementations;

import com.montytskyi.Bookings.Interfaces.Rentable;
import com.montytskyi.DormitoriesPlaces.Interfaces.Liveable;
import com.montytskyi.Userables.Intrefaces.Userable;

/**
 * The class which adapts the some users operations defined in 'Userable' to connect them
with administrative functions
 * @author Nazarii Montytskyi
 * @version 1.0
 */
public class AdministratorAdapter implements Userable{

  /** The administrator which abilities must be adapted to the requirements of 'Userable'
interface */
  Administrator administrator;

  /** The user which will be serviced by the administrator */
  Userable user;

  /**
   * The constructor of the adapter which applies the Administrator using dependency
injection
   * @param administrator - the administrator which will manage the system
   */
  public AdministratorAdapter(Administrator administrator){
    this.administrator = administrator;
  }

  /**
   * The method which allows to set the specific user who will be serviced by the
administrator
   * @param user , which are going to be serviced
   */
  public void setUserToService(Userable user){
    this.user = user;
  }

  /**
   * The implemented method from the interface Userable which implements renting the
place for the user
   * @param place - place, which will be rented for a while
   * @return Rentable, which represents the rent of a user
   */
  @Override
  public Rentable rent(Liveable place) {
    return this.administrator.rentForUser(place, user);
  }

  /**
   * The implmenented method from the interface Userable which implements payment process
for the user
   * @param currentRent - Rentable, which represents the current rent bill of the user
   */
  @Override
  public void payForRent(Rentable currentRent) {
    System.out.println("The administrator provides the payment ability to user's rent");
  }

  /**
   * The implemented method from the interface Userable which provides the administrator
an ability to cancel an existing user's rent
   * @param currentRent - Rentable, which represents an existing user's rent
```

```
     */
    @Override
    public void cancelRent(Rentable currentRent) {
      System.out.println("The administrator cancels user's rent");
      this.administrator.cancelRentForUser(currentRent);
    }

    /**
     * The implemented method from the interface Userable which provides the administrator
an ability to edit a rent recently created by the user
     * @param currentRent - Rentable, which represents an existing user's rent
     */
    @Override
    public void editRent(Rentable currentRent) {
      this.administrator.editRentForUser(currentRent);
    }

}


com.montytskyi.Userables.Implementations.User.java
package com.montytskyi.Userables.Implementations;

import com.montytskyi.Bookings.Implementations.Booking;
import com.montytskyi.Bookings.Interfaces.Rentable;
import com.montytskyi.DormitoriesPlaces.Interfaces.Liveable;
import com.montytskyi.Userables.Intrefaces.Userable;

/**
 * The class 'User' implements interface 'Userable' in order to implement the logic of
managing users' rents in the dormitory
 * @author Nazarii Montytskyi
 * @version 1.0
 */
public class User implements Userable{

    /** The name of user */
    private String name;

    /** The surname of user */
    private String surname;

    /** The phone number of user */
    private String phoneNumber;

    /** The email of user */
    private String email;

    /** The empty constructor of a class */
    public User(){

    }

    /**
     * The constructor which receives all parameters possible
     * @param name - the name of the user
     * @param surname - the surname of the user
     * @param phoneNumber - the phone number of the user
     * @param email - the email of the user
     */
    public User(String name, String surname, String phoneNumber, String email){
```

```
    this.name = name;
    this.surname = surname;
    this.phoneNumber = phoneNumber;
    this.email = email;
  }

  /**
   * The method implemented from the interface 'Userable' which goal is to rent the place
for the user
   * @param place - the place which will be rented by the user
   * @return Returns the rent which consists the user information and the data about the
place it's rented
   */
  @Override
  public Rentable rent(Liveable place) {
    System.out.println("User rents the place");
    Rentable newRent = new Booking(place, this, 200);
    newRent.setStatus("BOOKED");
    return newRent;
  }

  /**
   * The method implemented from the interface 'Userable' which goal is to provide a
payment for the rent
   * @param currentRent - receives a created rent where user becomes provided with the
ability to pay for the rent
   */
  @Override
  public void payForRent(Rentable currentRent) {
    currentRent.setStatus("PAIED");
    System.out.println("User pays for the rent");
  }

  /**
   * The method implemented from the interface 'Userable' which goal is to cancel user's
existing rent
   * @param currentRent - receives an existing user's rent which has been recently
created
   */
  @Override
  public void cancelRent(Rentable currentRent) {
    currentRent.setStatus("FINISHED");
    System.out.println("User cancels the rent");
  }

  /**
   * The method implemented from the interface 'Userable' which goal is to edit existing
user's rent
   * @param currentRent - receives an existing user's rent which has been recently
created
   */
  @Override
  public void editRent(Rentable currentRent) {
    System.out.println("User edits his rent");
  }

  /**
   * The getter method for the user's name
   * @return a User's name
   */
  public String getName() {
    return name;
  }

  /**
   * The setter for the user's name
   * @param name - the name of the user
   */
  public void setName(String name) {
    this.name = name;
  }

  /**
   * The getter for user's surname
   * @return the User's surname
   */
  public String getSurname() {
    return surname;
  }

  /**
   * The setter method for user's surname
   * @param surname - The user's surname
   */
  public void setSurname(String surname) {
    this.surname = surname;
  }

  /**
   * The getter method for user's phone number
   * @return the User's phone number
   */
  public String getPhoneNumber() {
    return phoneNumber;
  }

  /**
   * The setter method for user's phone number
   * @param phoneNumber - the user's phone number
   */
  public void setPhoneNumber(String phoneNumber) {
    this.phoneNumber = phoneNumber;
  }

  /**
   * The getter for user's email
   * @return the user's email
   */
  public String getEmail() {
    return email;
  }

  /**
   * The setter method for user's email
   * @return the user's email
   */
  public void setEmail(String email) {
    this.email = email;
  }

}
```