THE DEER ALLEY HOTEL 2 BOOKING SYSTEM

Date: 9 May 2016

Atanas Latinov, student number 239841
Daniel Hamarik, student number 239857
Ivan Stoyanov, student number 240247
Stefan Aleksiev, student number 240278

Supervisors:

Henrik Kronborg Pedersen

Mikkel Bayard Rasmussen

TABLE OF CONTENTS

1. Abstract....3

2. SCRUM....4

2.1 Product Backlog....4

2.2 Burndown chart....5

3. Design patterns....9

3.1 Factory pattern....9

3.2 Iterator pattern....10

3.3 Model–view–controller (MVC)....11

3.4 Singleton pattern....12

4. Client/Server connection....12

1. Abstract

Our customer is the proprietor of a hotel with a long history and traditions that stretch back to the 15th century. His desire is to replace the old ledger that is used to contain the guests' information and bookings with an automated booking system as he sees that avoiding technology might hold back his business.

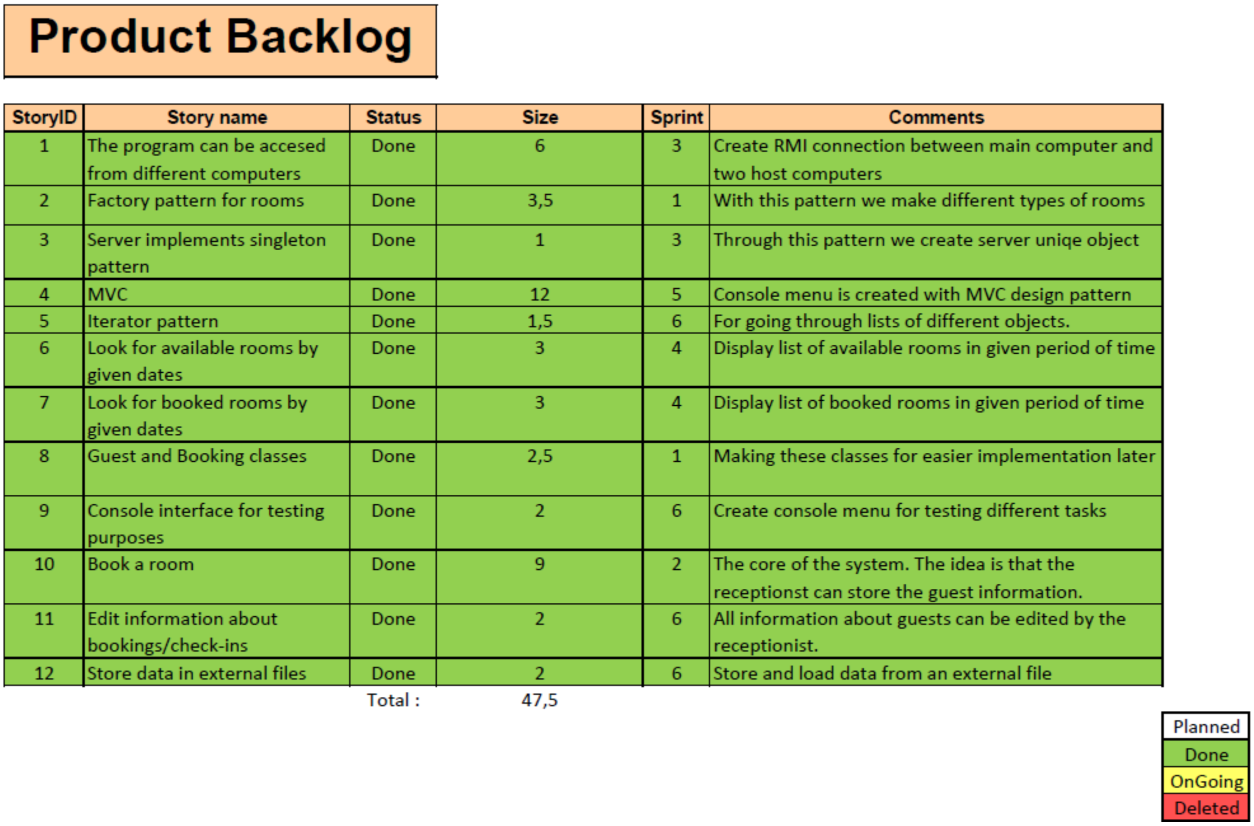
The system enables its users to book rooms, check in and check out guests, as well as save that information and then be able to display it depending on the criteria selected by the user. Furthermore, editing reservations and adding expenses is possible and an automated system for making rooms free is created. The core of this project is the extension of the aforementioned functionalities, which consists of it being accessible from two other systems on which the user can get a list of either the available or booked rooms.

The completion of the sprints well within the set time frame and the conclusions drawn from the release burndown chart have given us more insight into the complex process that is software development and have shown how beneficial it is to use a product development framework. The tests on the main components of the program have proven that it is a viable software product that satisfies all the requirements and demands and show that the program is capable of facilitating the room and guest management and providing assistance in the work of the receptionist.

2. SCRUM

The following section introduces the product backlog as well as detailed information about every sprint that the team had to complete. The burndown chart gives an overview of the work done for the duration of the project and also compares it with the ideal work done/time ratio.

2.1 Product Backlog

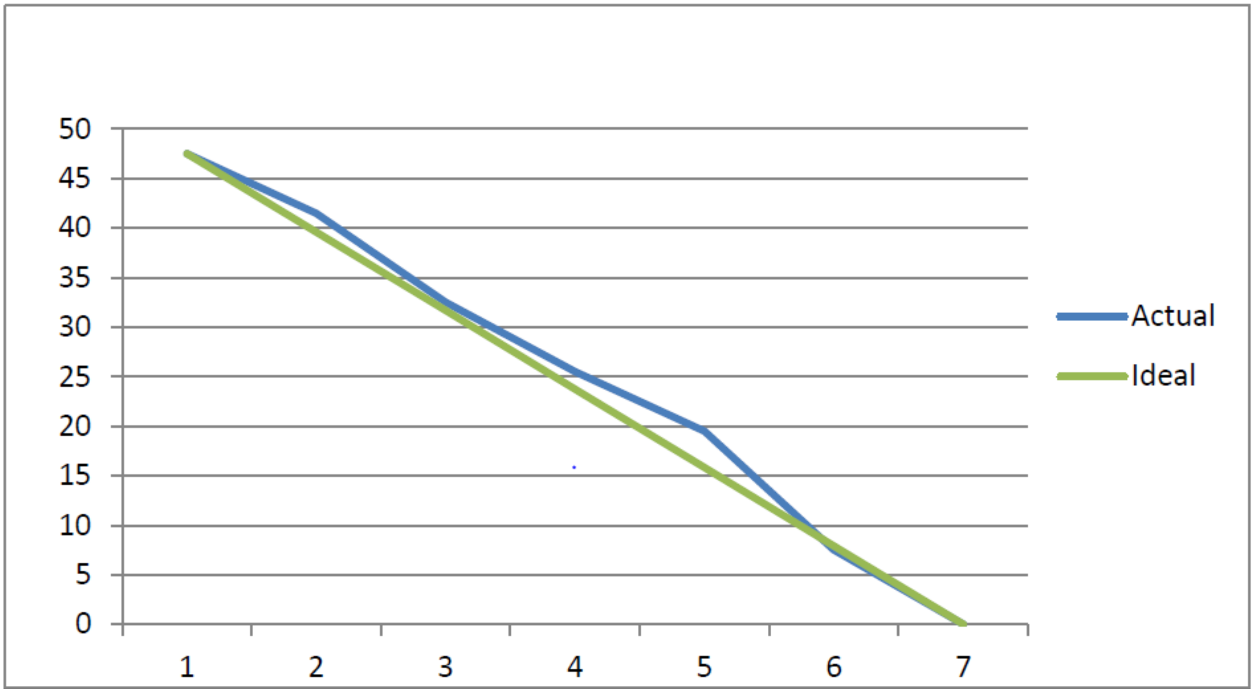


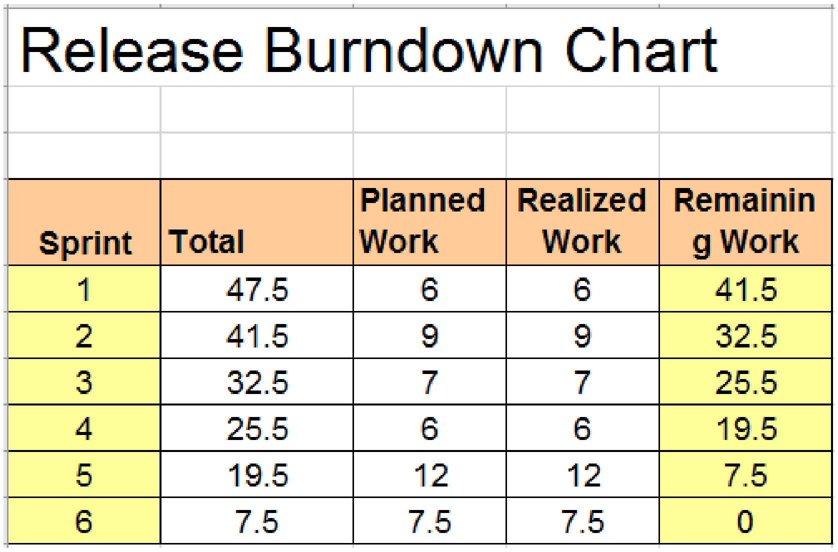
Product Backlog

StoryID	Story name	Status	Size	Sprint	Comments
1	The program can be accesed from different computers	Done	6	3	Create RMI connection between main computer and two host computers
2	Factory pattern for rooms	Done	3,5	1	With this pattern we make different types of rooms
3	Server implements singleton pattern	Done	1	3	Through this pattern we create server uniqe object
4	MVC	Done	12	5	Console menu is created with MVC design pattern
5	Iterator pattern	Done	1,5	6	For going through lists of different objects.
6	Look for available rooms by given dates	Done	3	4	Display list of available rooms in given period of time
7	Look for booked rooms by given dates	Done	3	4	Display list of booked rooms in given period of time
8	Guest and Booking classes	Done	2,5	1	Making these classes for easier implementation later
9	Console interface for testing purposes	Done	2	6	Create console menu for testing different tasks
10	Book a room	Done	9	2	The core of the system. The idea is that the receptionst can store the guest information.
11	Edit information about bookings/check-ins	Done	2	6	All information about guests can be edited by the receptionist.
12	Store data in external files	Done	2	6	Store and load data from an external file
		Total :	47,5		

Planned
Done
OnGoing
Deleted

2.2 Burndown chart

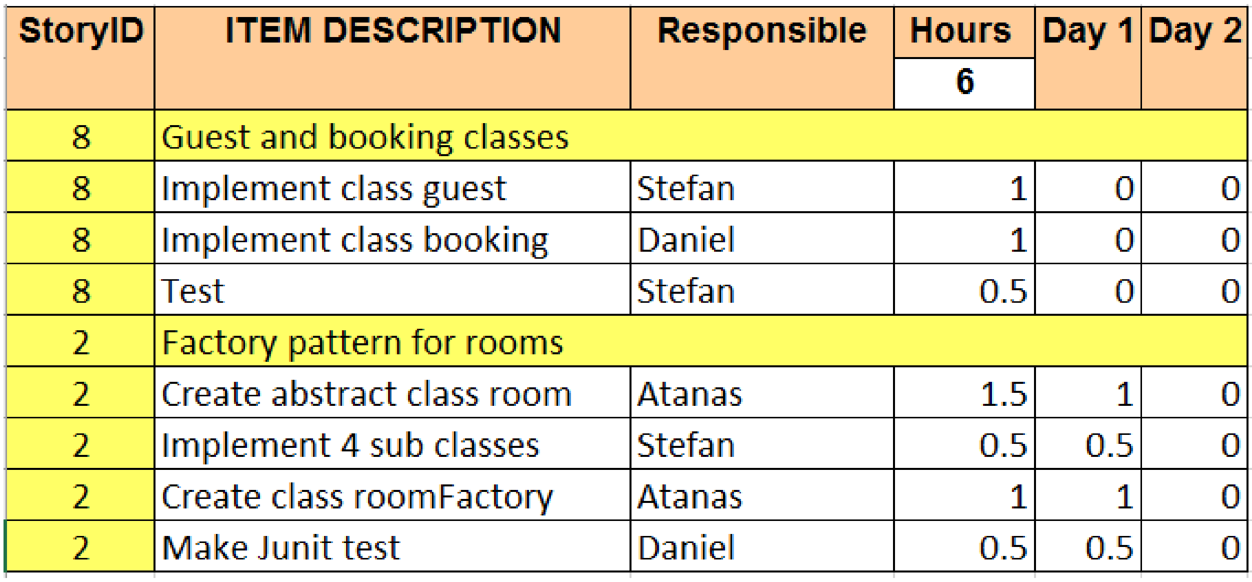




Release Burndown Chart

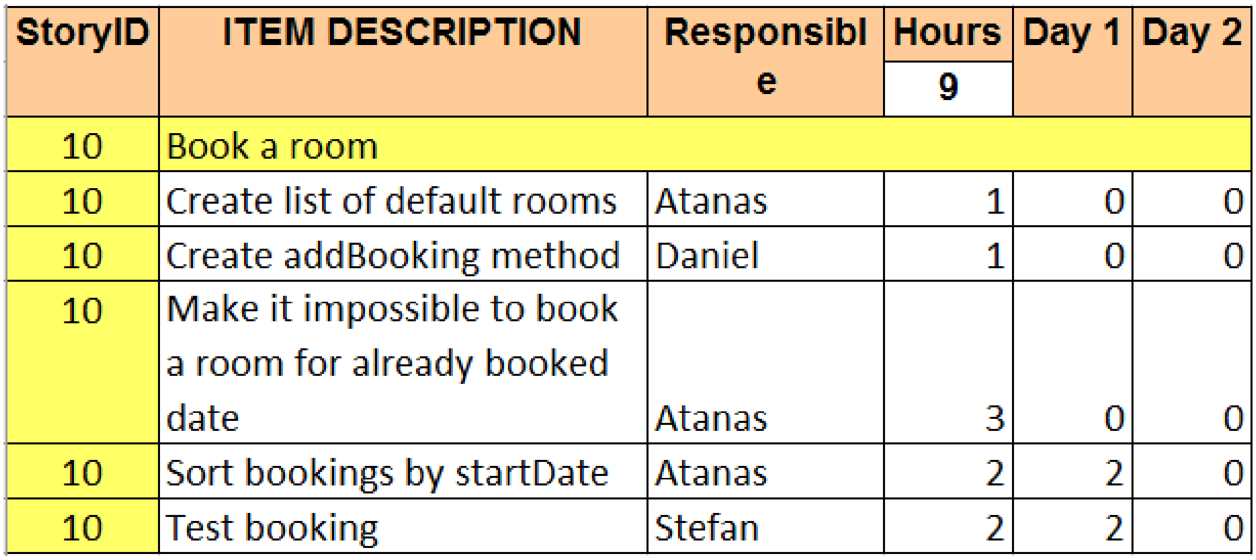
Sprint	Total	Planned Work	Realized Work	Remaining Work
1	47.5	6	6	41.5
2	41.5	9	9	32.5
3	32.5	7	7	25.5
4	25.5	6	6	19.5
5	19.5	12	12	7.5
6	7.5	7.5	7.5	0

Sprint 1



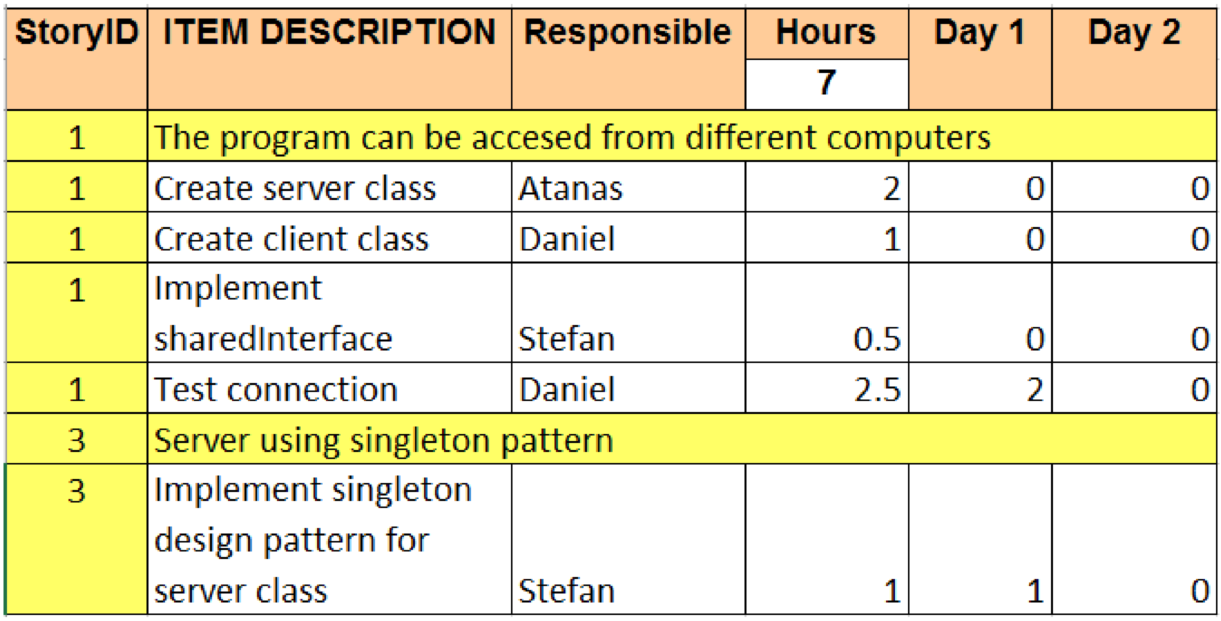
StoryID	ITEM DESCRIPTION	Responsible	Hours	Day 1	Day 2
			6		
8	Guest and booking classes				
8	Implement class guest	Stefan	1	0	0
8	Implement class booking	Daniel	1	0	0
8	Test	Stefan	0.5	0	0
2	Factory pattern for rooms				
2	Create abstract class room	Atanas	1.5	1	0
2	Implement 4 sub classes	Stefan	0.5	0.5	0
2	Create class roomFactory	Atanas	1	1	0
2	Make Junit test	Daniel	0.5	0.5	0

Sprint 2



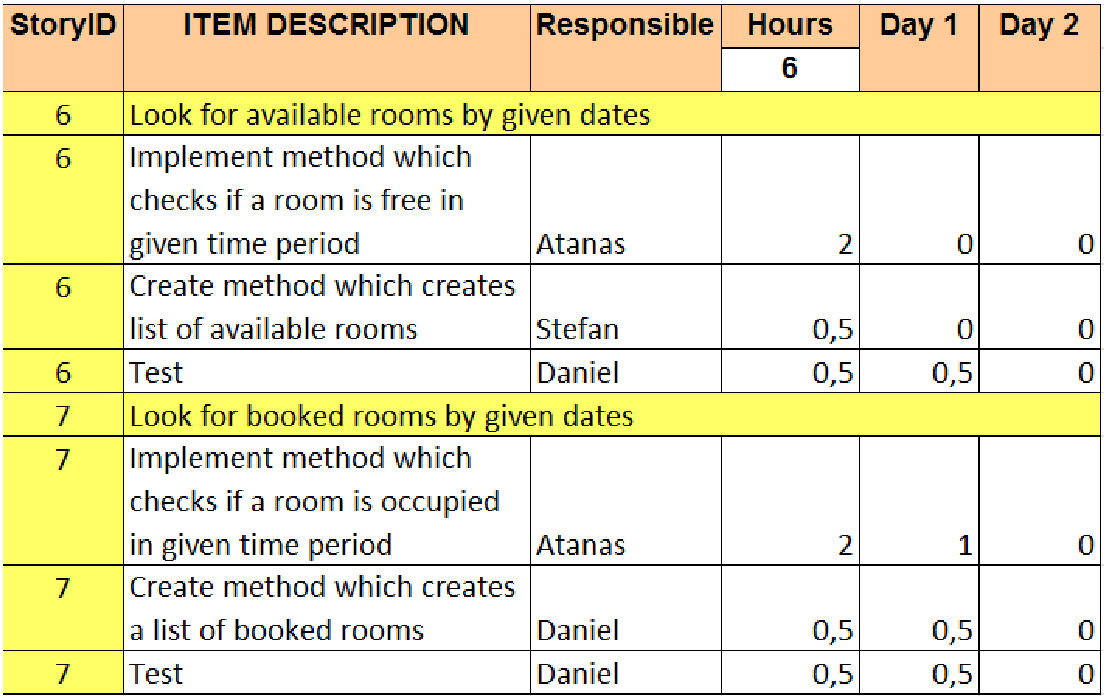
StoryID	ITEM DESCRIPTION	Responsible	Hours	Day 1	Day 2
			9		
10	Book a room				
10	Create list of default rooms	Atanas	1	0	0
10	Create addBooking method	Daniel	1	0	0
10	Make it impossible to book a room for already booked date	Atanas	3	0	0
10	Sort bookings by startDate	Atanas	2	2	0
10	Test booking	Stefan	2	2	0

Sprint 3



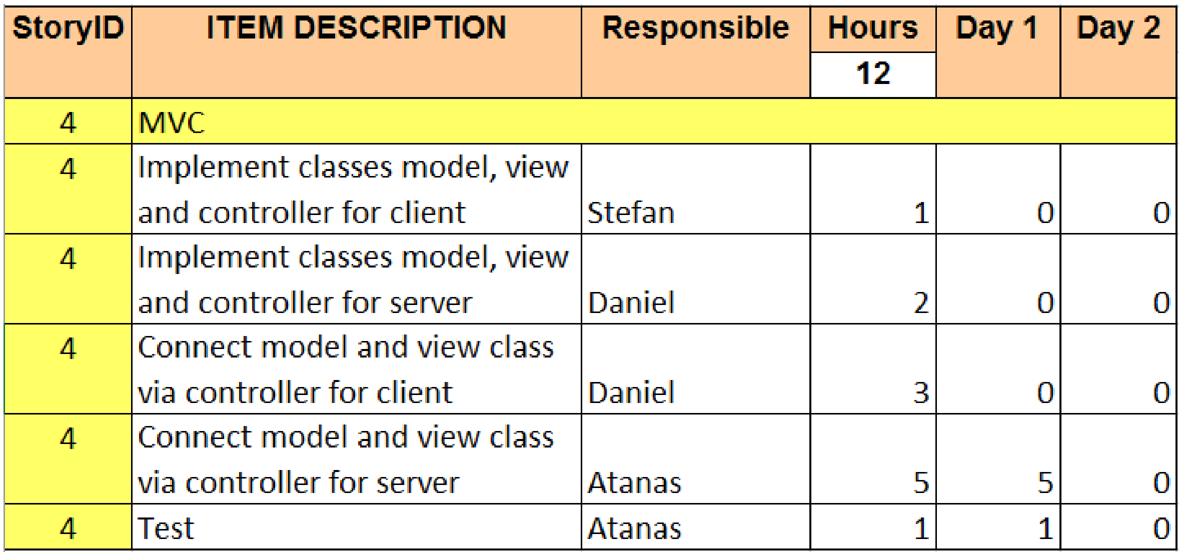
StoryID	ITEM DESCRIPTION	Responsible	Hours	Day 1	Day 2
			7		
1	The program can be accesed from different computers				
1	Create server class	Atanas	2	0	0
1	Create client class	Daniel	1	0	0
1	Implement sharedInterface	Stefan	0.5	0	0
1	Test connection	Daniel	2.5	2	0
3	Server using singleton pattern				
3	Implement singleton design pattern for server class	Stefan	1	1	0

Sprint 4



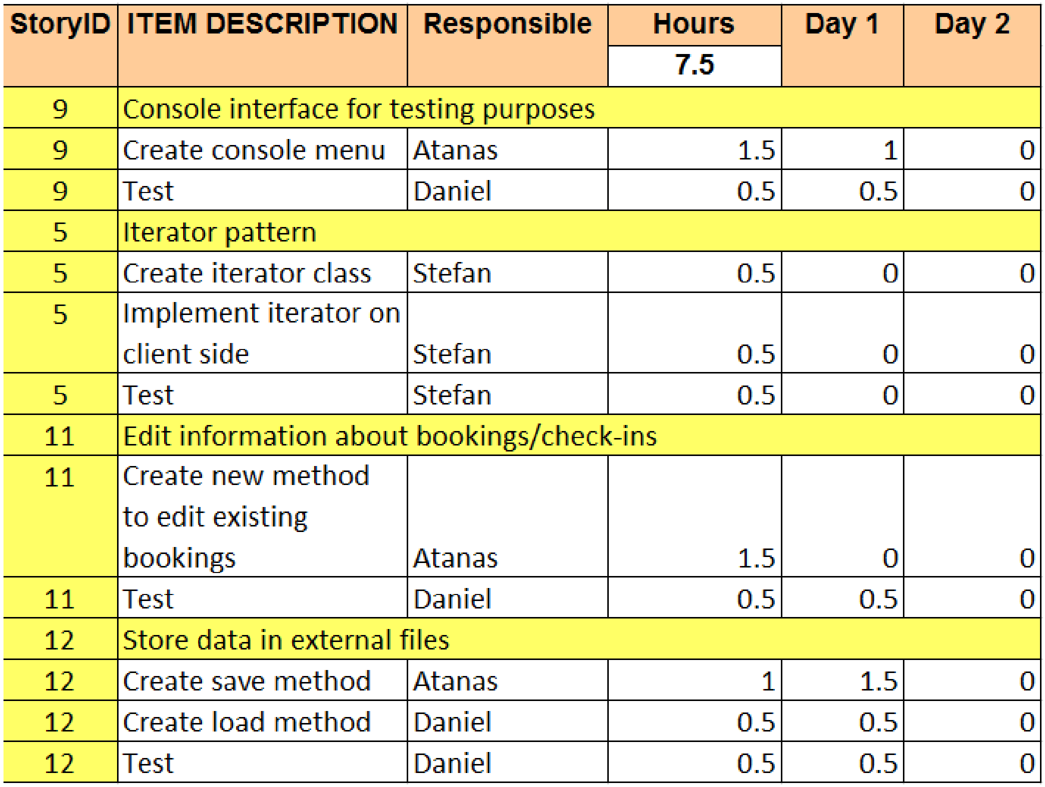
StoryID	ITEM DESCRIPTION	Responsible	Hours	Day 1	Day 2
			6		
6	Look for available rooms by given dates				
6	Implement method which checks if a room is free in given time period	Atanas	2	0	0
6	Create method which creates list of available rooms	Stefan	0,5	0	0
6	Test	Daniel	0,5	0,5	0
7	Look for booked rooms by given dates				
7	Implement method which checks if a room is occupied in given time period	Atanas	2	1	0
7	Create method which creates a list of booked rooms	Daniel	0,5	0,5	0
7	Test	Daniel	0,5	0,5	0

Sprint 5



StoryID	ITEM DESCRIPTION	Responsible	Hours	Day 1	Day 2
			12		
4	MVC				
4	Implement classes model, view and controller for client	Stefan	1	0	0
4	Implement classes model, view and controller for server	Daniel	2	0	0
4	Connect model and view class via controller for client	Daniel	3	0	0
4	Connect model and view class via controller for server	Atanas	5	5	0
4	Test	Atanas	1	1	0

Sprint 6

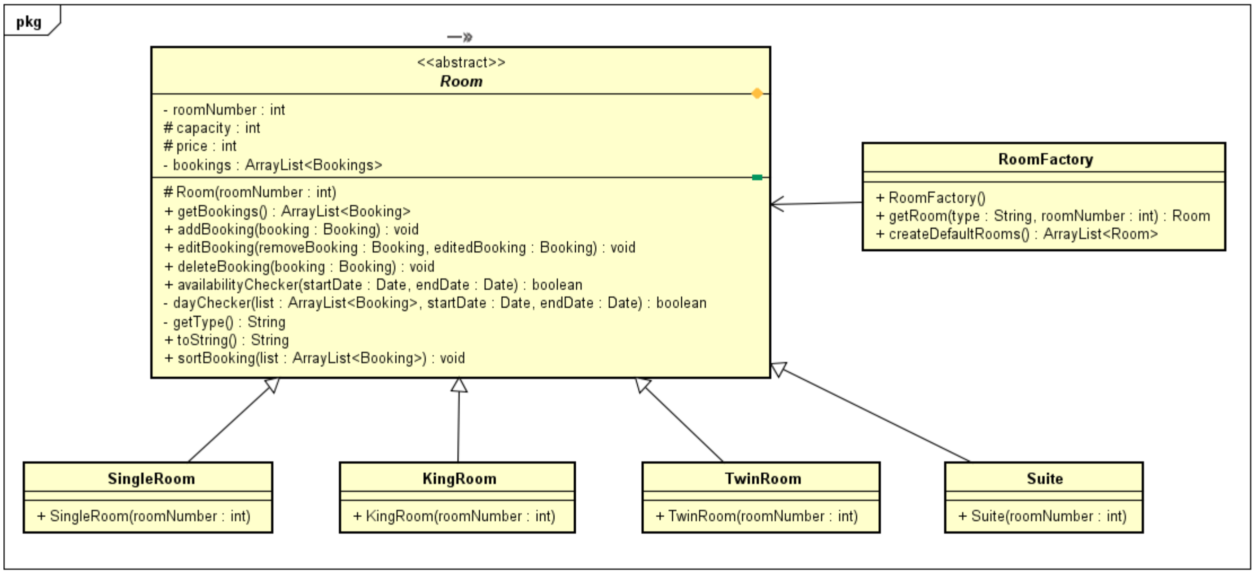


StoryID	ITEM DESCRIPTION	Responsible	Hours	Day 1	Day 2
			7.5		
9	Console interface for testing purposes				
9	Create console menu	Atanas	1.5	1	0
9	Test	Daniel	0.5	0.5	0
5	Iterator pattern				
5	Create iterator class	Stefan	0.5	0	0
5	Implement iterator on client side	Stefan	0.5	0	0
5	Test	Stefan	0.5	0	0
11	Edit information about bookings/check-ins				
11	Create new method to edit existing bookings	Atanas	1.5	0	0
11	Test	Daniel	0.5	0.5	0
12	Store data in external files				
12	Create save method	Atanas	1	1.5	0
12	Create load method	Daniel	0.5	0.5	0
12	Test	Daniel	0.5	0.5	0

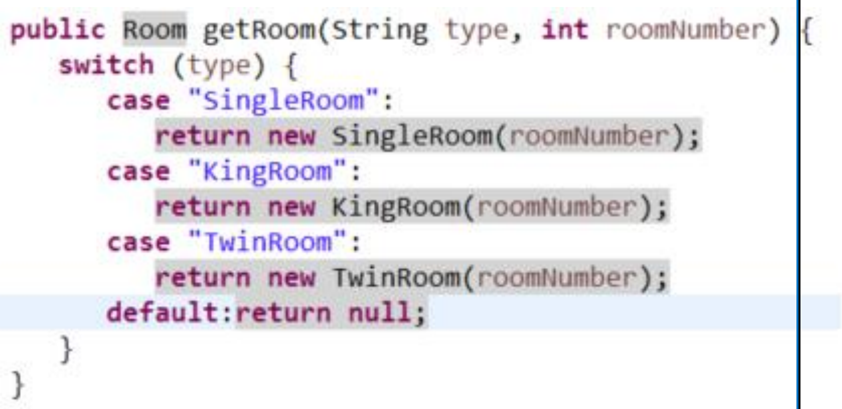
3. Design patterns

This section introduces the main patterns implemented in the system.

3.1 Factory pattern


pkg
<<abstract>>
Room
- roomNumber : int
# capacity : int
# price : int
- bookings : ArrayList<Bookings>
# Room(roomNumber : int)
+ getBookings() : ArrayList<Booking>
+ addBooking(booking : Booking) : void
+ editBooking(removeBooking : Booking, editedBooking : Booking) : void
+ deleteBooking(booking : Booking) : void
+ availabilityChecker(startDate : Date, endDate : Date) : boolean
- dayChecker(list : ArrayList<Booking>, startDate : Date, endDate : Date) : boolean
- getType() : String
+ toString() : String
+ sortBooking(list : ArrayList<Booking>) : void
RoomFactory
+ RoomFactory()
+ getRoom(type : String, roomNumber : int) : Room
+ createDefaultRooms() : ArrayList<Room>
SingleRoom
+ SingleRoom(roomNumber : int)
KingRoom
+ KingRoom(roomNumber : int)
TwinRoom
+ TwinRoom(roomNumber : int)
Suite
+ Suite(roomNumber : int)

Factory design pattern provides one of the best ways to create objects with the creation of the pure fabrication object called RoomFactory that handles all the room creations.

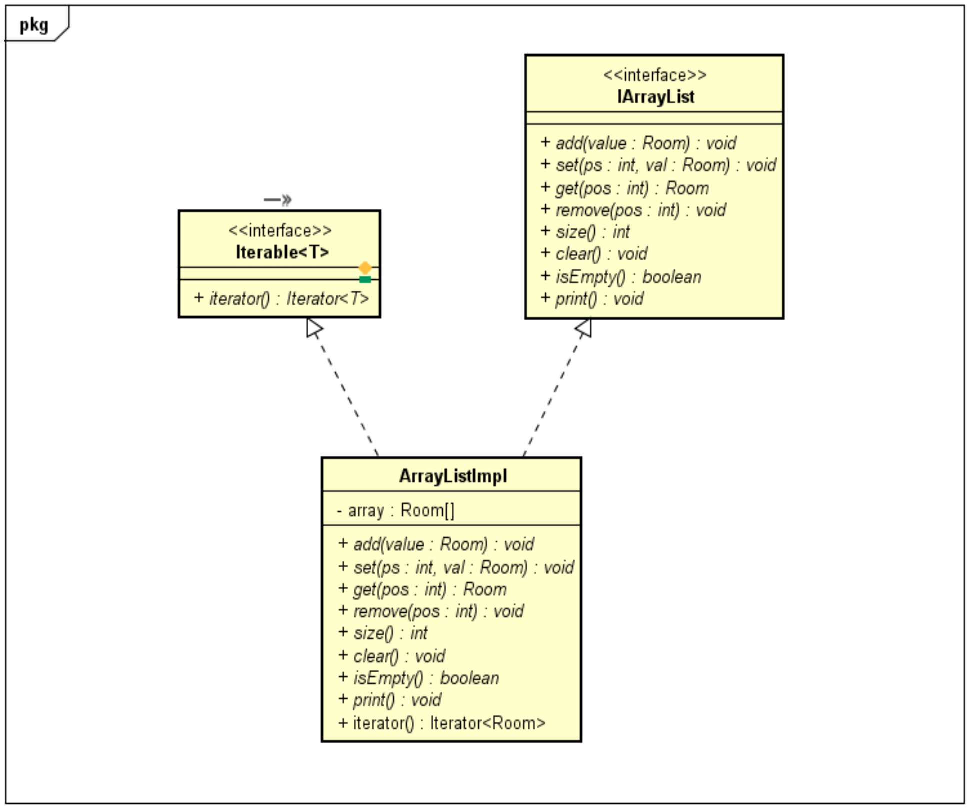


```
public Room getRoom(String type, int roomNumber) {
    switch (type) {
        case "SingleRoom":
            return new SingleRoom(roomNumber);
        case "KingRoom":
            return new KingRoom(roomNumber);
        case "TwinRoom":
            return new TwinRoom(roomNumber);
        default:return null;
    }
}
```

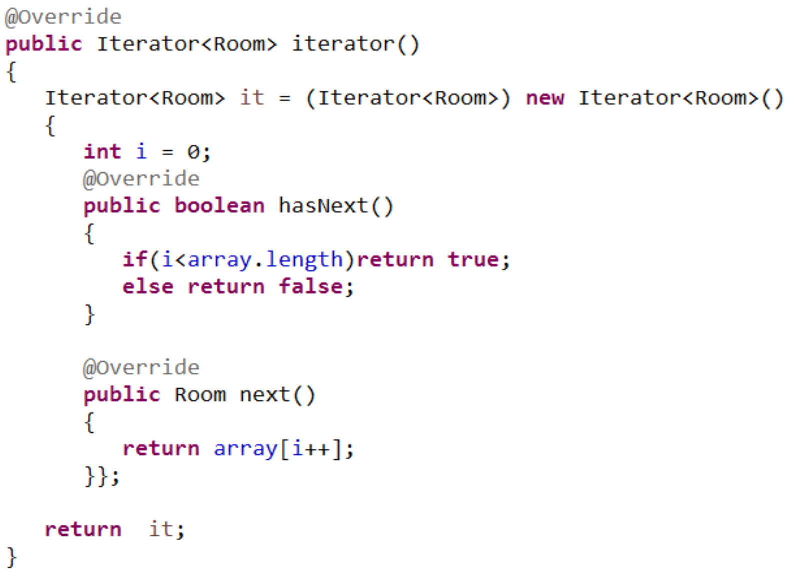
RoomFactory object has several advantages:

- Separate the responsibility of complex creation of Room objects
- Hide potentially complex creation logic
- Allow object caching or recycling

3.2 Iterator pattern



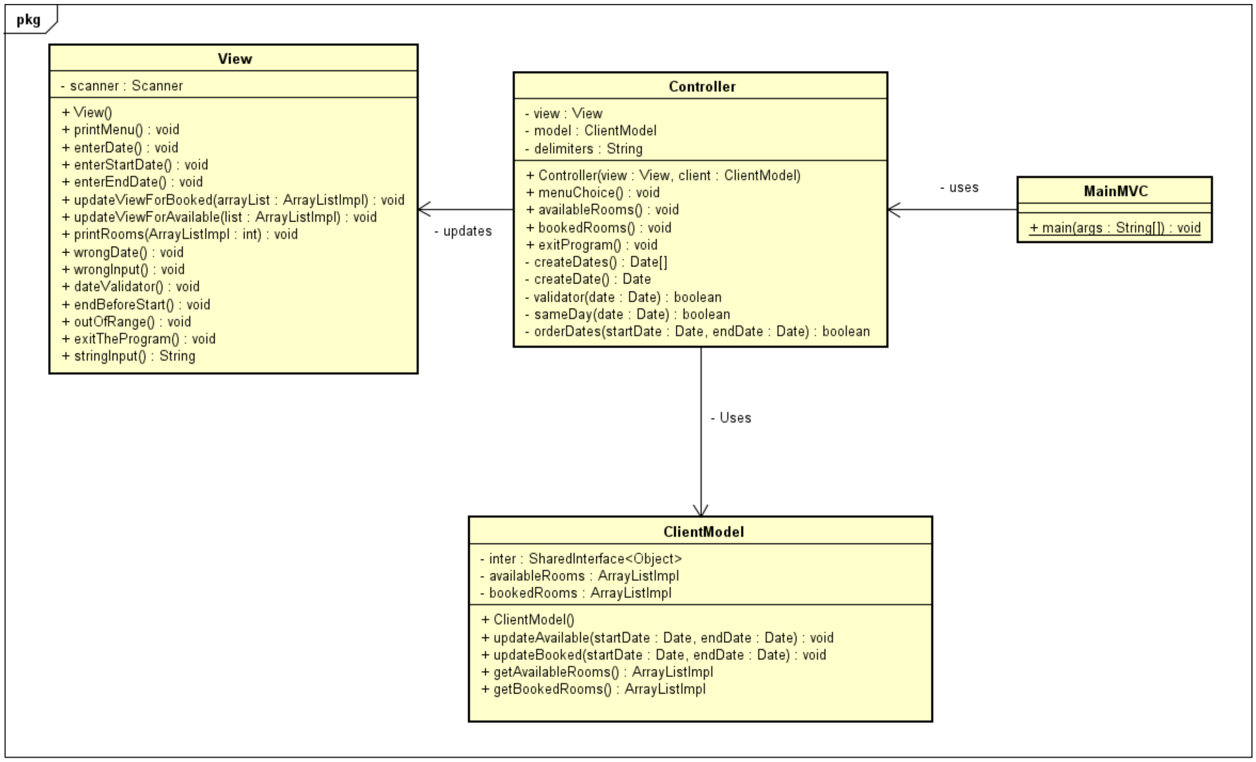
In our project we have created a collection called ArrayListImpl which implements the Iterable<T> interface. We have overridden the methods hasNext() and next(), so we can access the elements of a collection object in a sequential manner without any need to know its underlying representation.



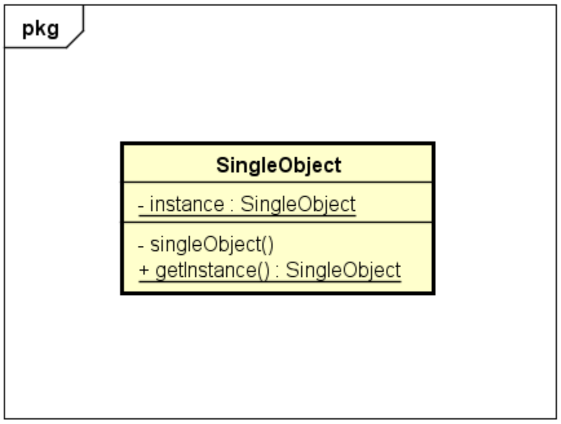
```
@Override
public Iterator<Room> iterator()
{
   Iterator<Room> it = (Iterator<Room>) new Iterator<Room>()
   {
      int i = 0;
      @Override
      public boolean hasNext()
      {
         if(i<array.length)return true;
         else return false;
      }

      @Override
      public Room next()
      {
         return array[i++];
      }};

   return  it;
}
```

3.3 Model–view–controller (MVC)

The Model-View-Controller Pattern is used to separate the application's concerns. In our case the View is the output representation of information on the console, it generates new output to the user based on changes in the model. The Model directly manages the data, logic and rules of the system, it stores data that is retrieved according to commands from the controller and displayed in the view. A controller sends commands to the model to update information and it sends commands to the view to change the view's presentation of the data.


pkg
View
- scanner : Scanner
+ View()
+ printMenu() : void
+ enterDate() : void
+ enterStartDate() : void
+ enterEndDate() : void
+ updateViewForBooked(arrayList : ArrayListImpl) : void
+ updateViewForAvailable(list : ArrayListImpl) : void
+ printRooms(ArrayListImpl : int) : void
+ wrongDate() : void
+ wrongInput() : void
+ dateValidator() : void
+ endBeforeStart() : void
+ outOfRange() : void
+ exitTheProgram() : void
+ stringInput() : String
- updates
Controller
- view : View
- model : ClientModel
- delimiters : String
+ Controller(view : View, client : ClientModel)
+ menuChoice() : void
+ availableRooms() : void
+ bookedRooms() : void
+ exitProgram() : void
- createDates() : Date[]
- createDate() : Date
- validator(date : Date) : boolean
- sameDay(date : Date) : boolean
- orderDates(startDate : Date, endDate : Date) : boolean
- uses
MainMVC
+ main(args : String[]) : void
- Uses
ClientModel
- inter : SharedInterface<Object>
- availableRooms : ArrayListImpl
- bookedRooms : ArrayListImpl
+ ClientModel()
+ updateAvailable(startDate : Date, endDate : Date) : void
+ updateBooked(startDate : Date, endDate : Date) : void
+ getAvailableRooms() : ArrayListImpl
+ getBookedRooms() : ArrayListImpl

3.4 Singleton pattern

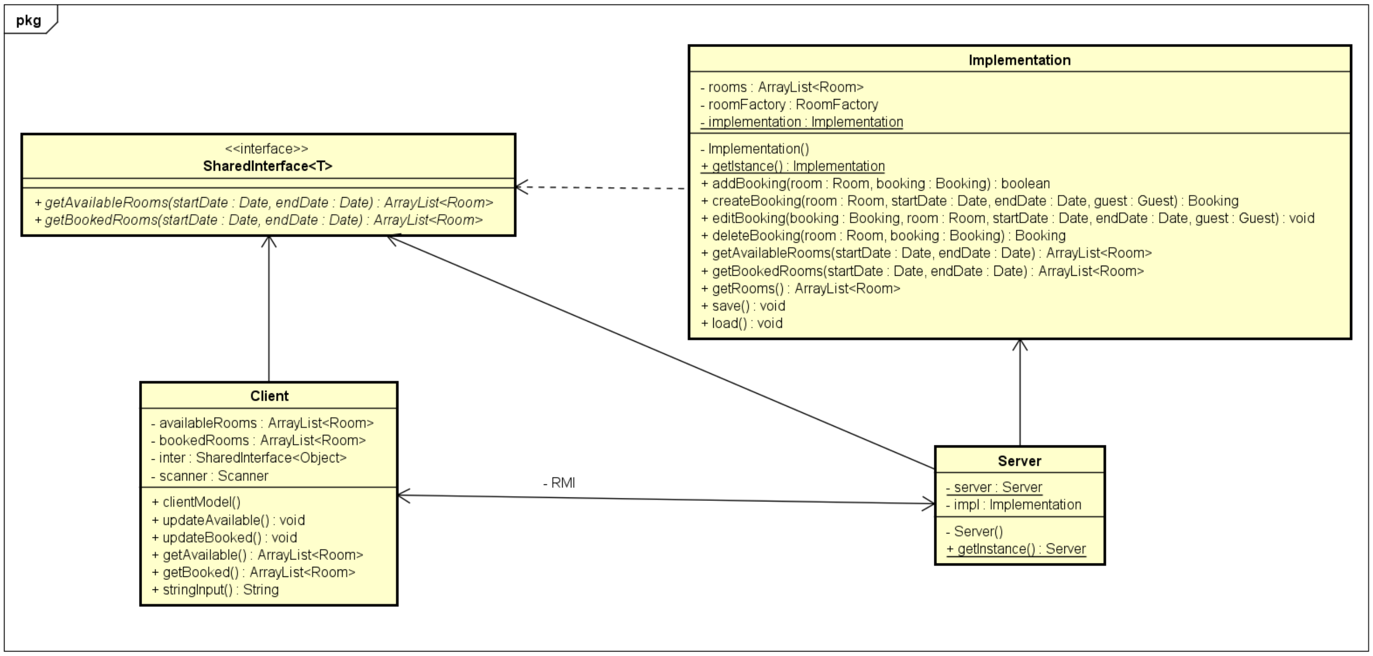


In our project only one instance of Implementation class is needed because it is the core of our program. It operates with unique data which is loaded and saved from an external file. On the other hand methods from the Implementation class may need to be called from various places in our code. One solution is to pass an Implementation instance around as parameter to all classes which use its methods. This is possible but inconvenient so in this situation we find the singleton design pattern as very useful. With singleton design pattern there is always exactly one instance of a class allowed. With this approach, we have global visibility to this single instance, via the static getInstance method of the class.

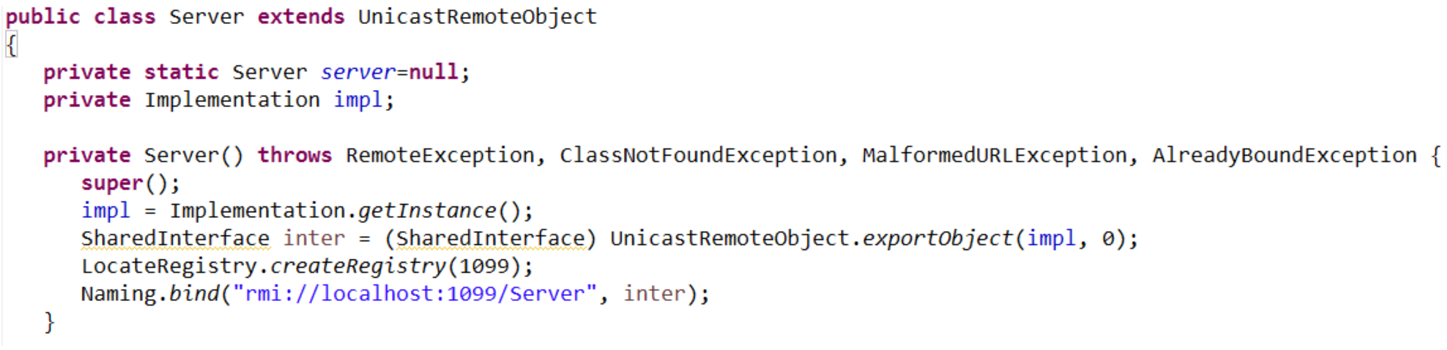
4. Client/Server connection

4.1 RMI Remote method invocation

The Java Remote Method Invocation (RMI) system allows an object running in one Java virtual machine to invoke methods on an object running in another Java virtual machine.

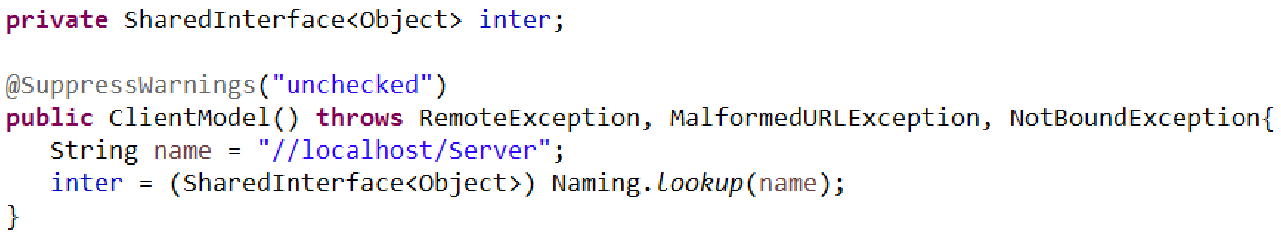

pkg
<<interface>>
SharedInterface<T>
+ getAvailableRooms(startDate : Date, endDate : Date) : ArrayList<Room>
+ getBookedRooms(startDate : Date, endDate : Date) : ArrayList<Room>
Implementation
- rooms : ArrayList<Room>
- roomFactory : RoomFactory
- implementation : Implementation
- Implementation()
+ getIstance() : Implementation
+ addBooking(room : Room, booking : Booking) : boolean
+ createBooking(room : Room, startDate : Date, endDate : Date, guest : Guest) : Booking
+ editBooking(booking : Booking, room : Room, startDate : Date, endDate : Date, guest : Guest) : void
+ deleteBooking(room : Room, booking : Booking) : Booking
+ getAvailableRooms(startDate : Date, endDate : Date) : ArrayList<Room>
+ getBookedRooms(startDate : Date, endDate : Date) : ArrayList<Room>
+ getRooms() : ArrayList<Room>
+ save() : void
+ load() : void
Client
- availableRooms : ArrayList<Room>
- bookedRooms : ArrayList<Room>
- inter : SharedInterface<Object>
- scanner : Scanner
+ clientModel()
+ updateAvailable() : void
+ updateBooked() : void
+ getAvailable() : ArrayList<Room>
+ getBooked() : ArrayList<Room>
+ stringInput() : String
- RMI
Server
- server : Server
- impl : Implementation
- Server()
+ getInstance() : Server

Server :



```
public class Server extends UnicastRemoteObject
{
   private static Server server=null;
   private Implementation impl;

   private Server() throws RemoteException, ClassNotFoundException, MalformedURLException, AlreadyBoundException {
      super();
      impl = Implementation.getInstance();
      SharedInterface inter = (SharedInterface) UnicastRemoteObject.exportObject(impl, 0);
      LocateRegistry.createRegistry(1099);
      Naming.bind("rmi://localhost:1099/Server", inter);
   }
```

Client :



```
private SharedInterface<Object> inter;

@SuppressWarnings("unchecked")
public ClientModel() throws RemoteException, MalformedURLException, NotBoundException{
   String name = "//localhost/Server";
   inter = (SharedInterface<Object>) Naming.lookup(name);
}
```