SDJ2 Course Assignment

Spring 2017

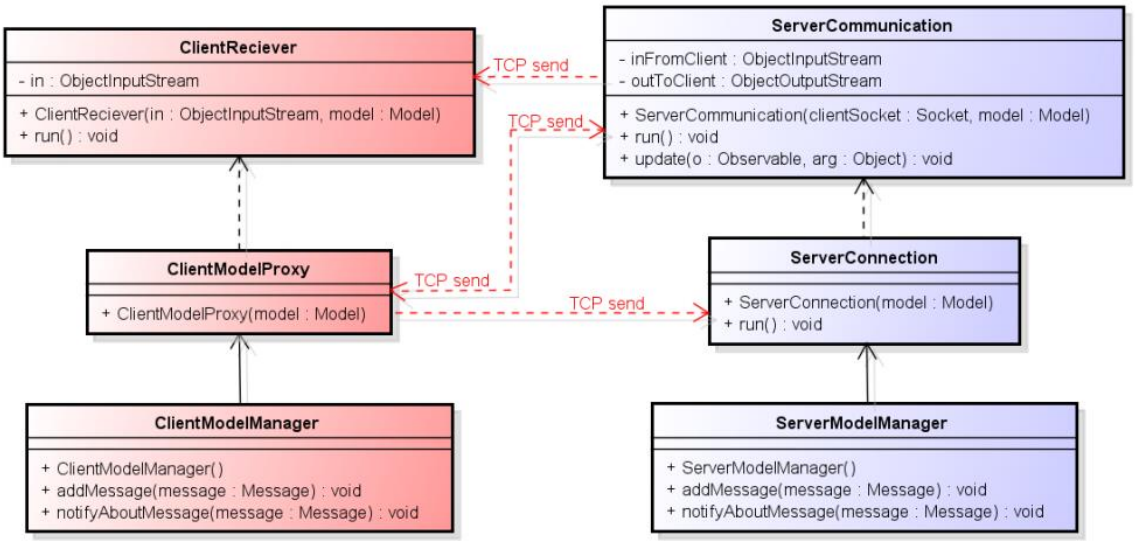
- Andreea Buturca (253691)
- Marek Lowy (253652)
- Martin Janosik (253934)
- Krzysztof Majcher (253784)
- Stela Brziakova (254198)

Abstract

- VIA Bus is a company located in Horsens, Denmark with Trip Driver as the manager. At VIA Bus, you can either rent a bus with a chauffeur – driving to a destination of your choice, or travel by bus to one of the predefined locations, i.e. a few European countries and some Danish sites and events. VIA Bus needs a system to keep track of the tours, chauffeurs, bus routes and customers, and therefore the manager Trip Driver decided to contact us to create it. The system is based on Java and meets the requirements of the customer. The system is user friendly, can be used by any employee of the company, it contains menus, tabs, lists and buttons exactly as Trip Driver required. System is connected through a network which enables the clients to get current trips the second they are added. After it is added the client computer is able to search through them using date intervals. There are several options in the main menu. The main feature of the system is to create and manage trips. User can create the trip picking up the bus, chauffeur, date and destination.

Client-Server Architecture

Example of UML diagram in client-server architecture (chat application):



General description of Client-Server:

- Client-server architecture (client/server) is a network architecture in which each computer or process on the network is either a client or a server. Servers are powerful computers or processes dedicated to managing disk drives (file servers), printers (print servers), or network traffic (network servers). Clients are PCs or workstations on which users run applications. Clients rely on servers for resources, such as files, devices, and even processing power.

Client-Server using Java sockets:

- Server:
  - On server side the port number needs to be specified. Following is to create welcome socked. “ServerSocket welcomeSocket = new ServerSocket(PORT);”. Welcome socket is used to “welcome” clients who try to connect to the server. When the client connects to the server, the connection controller is created “Socket connectionSocket = welcomeSocket.accept();”. Connection controller is used to create ObjectStreams. Those send and receive objects from client.
  - // create output stream attached to the socket ObjectOutputStream outToClient = new ObjectOutputStream(connectionSocket.getOutputStream());

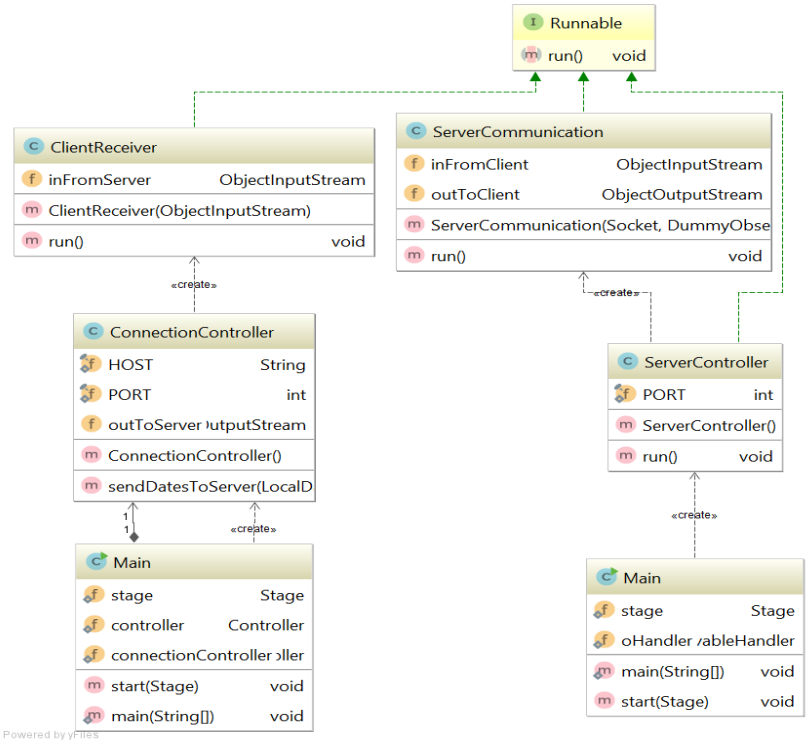
- // create input stream attached to the socket ObjectInputStream inFromClient = new ObjectInputStream((connectionSocket.getInputStream()));
- Reading an sending objects from clients is done by writeObject() and readObject methods from ObjectStreams.

- Client:
  - Client is done in similar way. To create connection it needs to know port number used on server and servers IP address. "Socket clientSocket = new Socket(HOST, PORT);". After the client socket is created, ObjectStreams are created in similiar way as in server, only using clientSocket.

Client-Server according to UML diagram:

- Class server connection is responsible for starting the server and waiting for the clients to connect. After the client connects to the server, the instance of serverCommunication class is created with connectionSocket.
- Class serverCommunication is responsible for creating object streams and communication with client.
- Class clientModelProxy is responsible for connecting to the server and creating clientReceiver.
- Class clientReceiver is receiving objects from server and passing them to the client system.

Our implementation of Client-Server:



Description of our implementation:

- Server:
  - After the main class creates serverController the welcome socket is created and sever controller waits for clients to connect.
  - When client connects, serverCommunication is created. After that the current trip list is sent to the client and then the class waits for dates from client side which are then passed to the different part of the system to be handled.
  - Other parts of the system responsible for sending objects to clients are DataHandler and DummyObserver.
- Client:
  - The connection controller connects to the server and creates object streams and client receiver.
  - Client receiver is expecting ProxyTripList object from list. When it is received, the list is displayed in client GUI.

Code examples:

- Class ServerController

```
public class ServerController implements Runnable {

    private static final int PORT = 6666;

    public void run() {
        int count = 1;
        try {
            ServerSocket welcomeSocket = new ServerSocket(PORT);
            System.out.println("Server started");
            DummyObserver observer = new DummyObserver();
            while (true) {
                Socket connectionSocket = welcomeSocket.accept();
                ServerCommunication c = new ServerCommunication(connectionSocket,
observer);
                new Thread(c, "Communication " + count).start();
                count++;
            }
        } catch (Exception e) {
            e.printStackTrace();
        }

    }
}
```

- Run method from class ServerCommunication.

```
-     public void run() {
        try {
            outToClient.writeObject(DataHandler.getInstance().getTrips());
            outToClient.reset();
        } catch (IOException e) {
            e.printStackTrace();
```

```
        }
        while (true) {
            try {
                LocalDate[] dates = (LocalDate[]) inFromClient.readObject();
                DataHandler.getInstance().getInDates(dates, outToClient);
                inFromClient.reset();
            } catch (IOException e) {
                //
            } catch (ClassNotFoundException e) {
                //
            }
        }

    }
```

- Contructor of ConnectionController

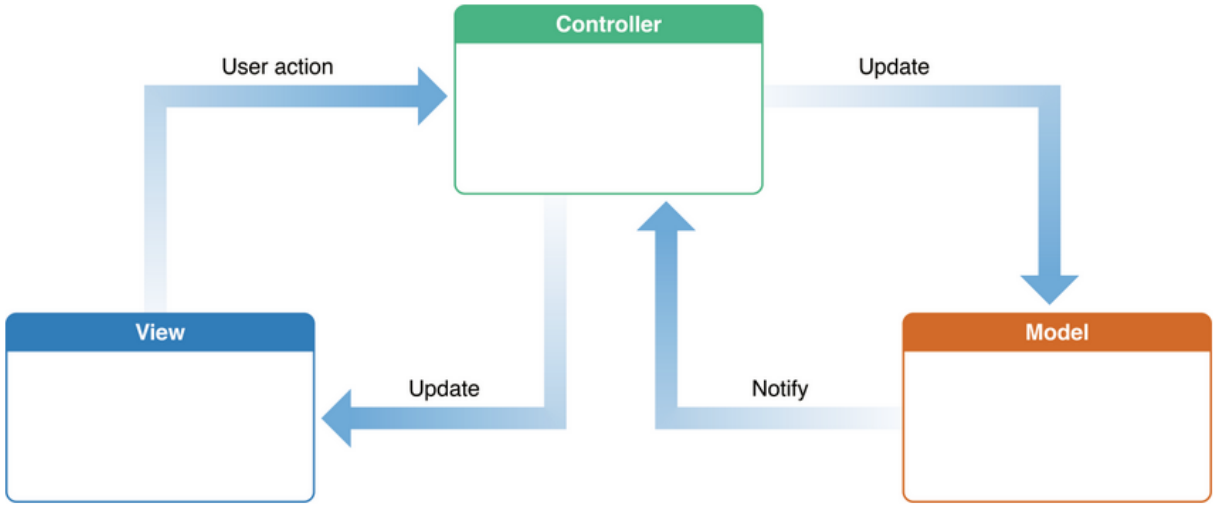
```
public ConnectionController() throws IOException {
    try {
        Socket socket = new Socket(HOST, PORT);
        outToServer = new ObjectOutputStream(socket.getOutputStream());
        ObjectInputStream inFromServer = new
ObjectInputStream(socket.getInputStream());
        ClientReceiver reciever = new ClientReceiver(inFromServer);
        new Thread(reciever, "Reciever").start();
    } catch (Exception e) {
        e.printStackTrace();
    }

}
```

- Run method from ClientReceiver

```
-     @Override
    public void run() {
        while (true) {
            try {
                ProxyTripList trips = (ProxyTripList) inFromServer.readObject();
                Main.controller.showList(trips);
            } catch (IOException | ClassNotFoundException e) {
                e.printStackTrace();
            }
        }
    }
```

Model-View-Controller design pattern

General diagram for MVC design pattern:

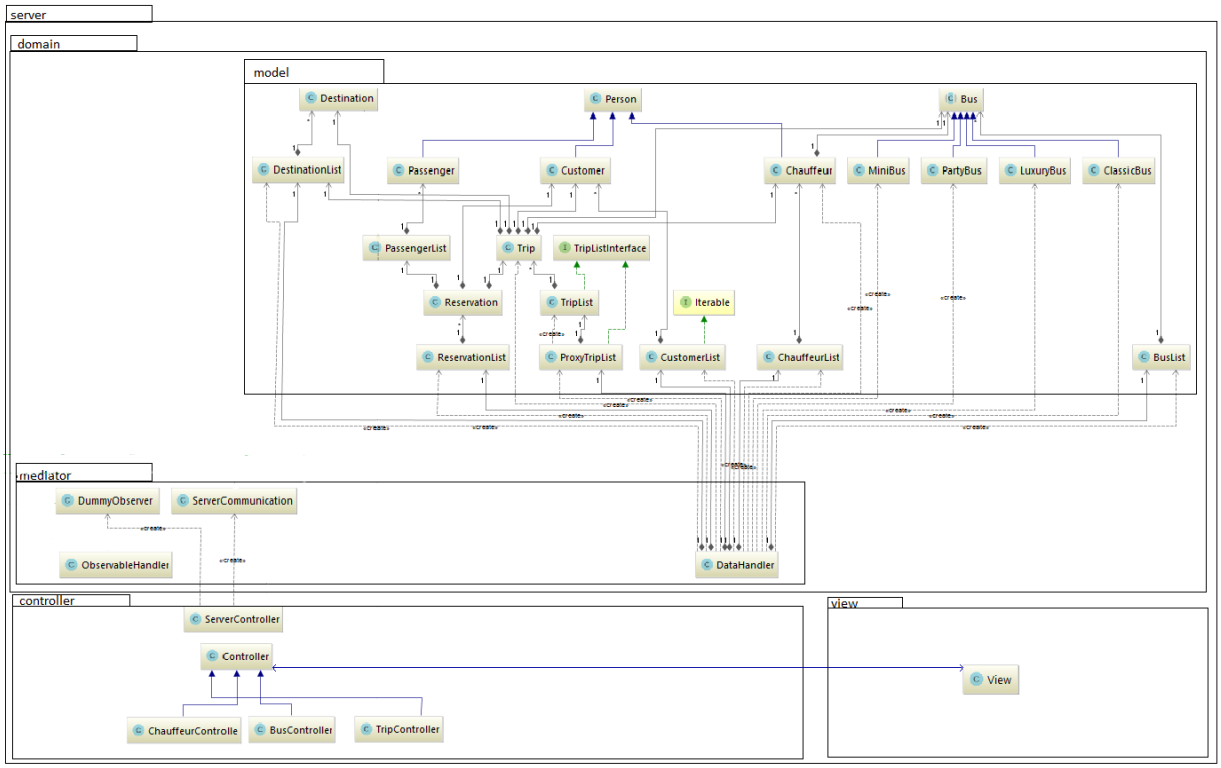


MVC description:

- The Model-View-Controller (MVC) design pattern assigns objects in an application one of three roles: model, view, or controller. The pattern defines not only the roles objects play in the application, it defines the way objects communicate with each other.
- **Controller**
    - Controllers act as an interface between Model and View components to process all the logic and requests, manipulate data using the Model component and interact with the Views to show the final output.
- **View**
    - Views are responsible for displaying all or a portion of the data to the user.
- **Model**

- Model corresponds to all the data-related logic that the user works with. This can represent either the data that is being transferred between the View and Controller components or any other logic-related data.

Our implementation:



- **Controller** (BusController)

```
private void loadList() {
    busListview.getSelectionModel().setSelectionMode(SelectionMode.MULTIPLE);
    ObservableList<Bus> items = DataHandler.getInstance().getObservableListOfBuses();
    busListview.setItems(items);
}

/**
 * Deletes selected buses from list of buses.
 */

public void deleteBus(ActionEvent actionEvent) throws FileNotFoundException, ParseException {
    ObservableList<Bus> selected;
    selected = busListview.getSelectionModel().getSelectedItems();
    for (Bus aSelected : selected) {
        DataHandler.getInstance().removeFromBuslist(aSelected);
    }

    loadList();
    DataHandler.getInstance().save();
}
}
    }
```

- BusController delegates work to DataHandler from mediator package in order to get all data it needs to display something in GUI, or to create new objects of classes in model package.

- DataHandler class:

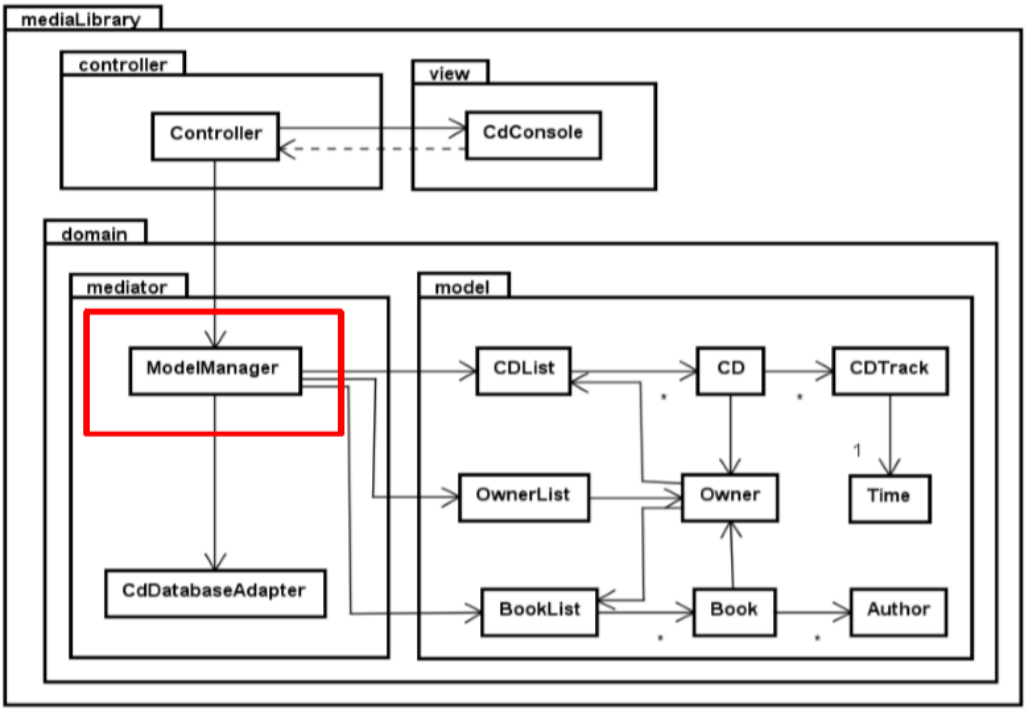
```
public void removeFromBuslist(Bus bus) {
    busList.removeBus(bus);
}

public ObservableList getObservableListOfBuses() {
    ObservableList<Bus> items = FXCollections.observableArrayList();
    for (Bus bus : busList.getArrayBuses()) {
        items.add(bus);
    }
    return items;
}
```

- DataHandler solves the tasks by accessing the model and it contains all the necessary methods to mediate work from controller classes to model classes.

- The package View contains all fxml files for all different screens in the system. Those do not have access to anything else in the system, and are controlled by corresponding classes in the Controller package.

Façade

An example of UML class diagram for façade design pattern:



The overall purpose for the Façade design pattern:

- Facade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system. This type of design pattern comes under structural pattern as this pattern adds an interface to existing system to hide its complexities.

Description of example UML diagram:

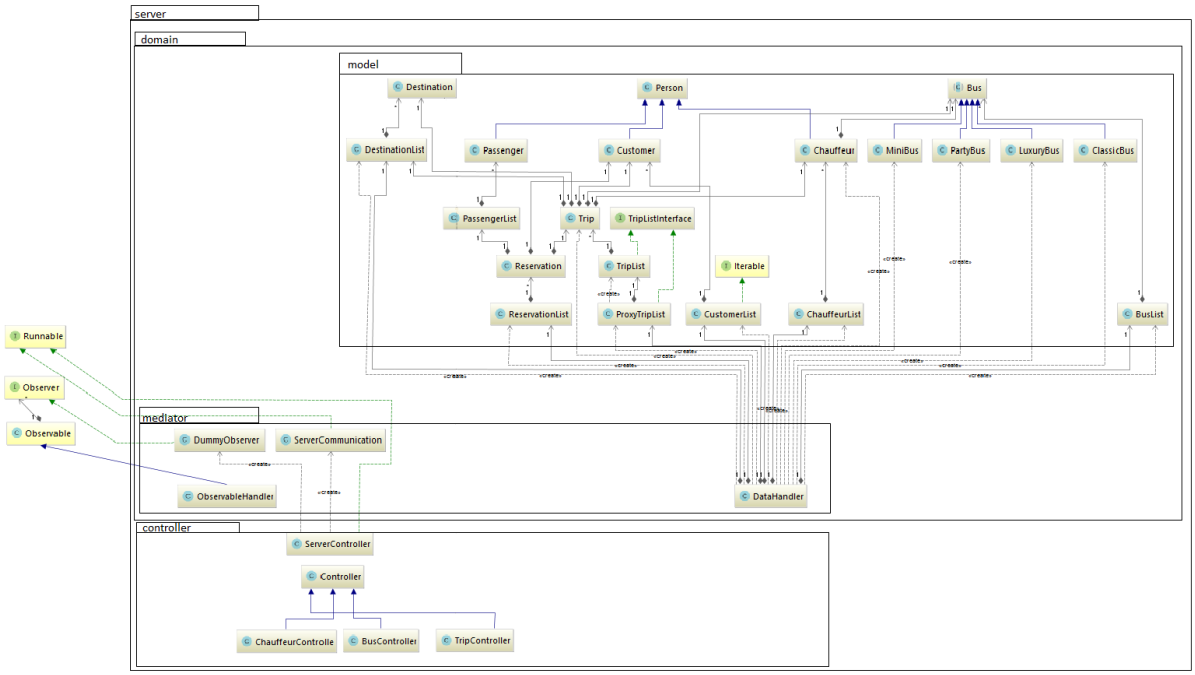
- The ModelManager is the façade for model having model state (and additional actions).
- The intent of façade is to provide an interface with simplified methods required by client which delegate calls to methods of existing system classes.

- Facade shows how to make a single object represent an entire subsystem.

Usage of façade design pattern:

- When client wants to access data from the model, the model manager method is called and model manager does the job of accessing model data.
- Façade design pattern does not require to have interface for façade class.

Our implementation:



- In our system, the data is controlled through one class called DataHandler. The user is able to access all the data through this class. It doesn't use an interface like it is shown on the original diagram. When the client wants to access the list of tours from his computer, The DataHandler class allows the user to get their list of information in one call by creating an object of type DataHandler.

Code example:

- Method load from data handler loads data from file and stores them inside

```
public void load() {
    String filename = "mainData.bin";
    ObjectInputStream in = null;
    try {
        File file = new File(filename);
        FileInputStream fis = new FileInputStream(file);
        in = new ObjectInputStream(fis);
        trips = (ProxyTripList) in.readObject();
        busList = (BusList) in.readObject();
        chauffeurList = (ChauffeurList) in.readObject();
        customerList = (CustomerList) in.readObject();
        reservationList = (ReservationList) in.readObject();
        destinationList = (DestinationList) in.readObject();
    } catch (ClassCastException | IOException | ClassNotFoundException e) {
        testCreate();
    } finally {
        try {
            if (in != null) {
                in.close();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- User can call getter methods to get model data

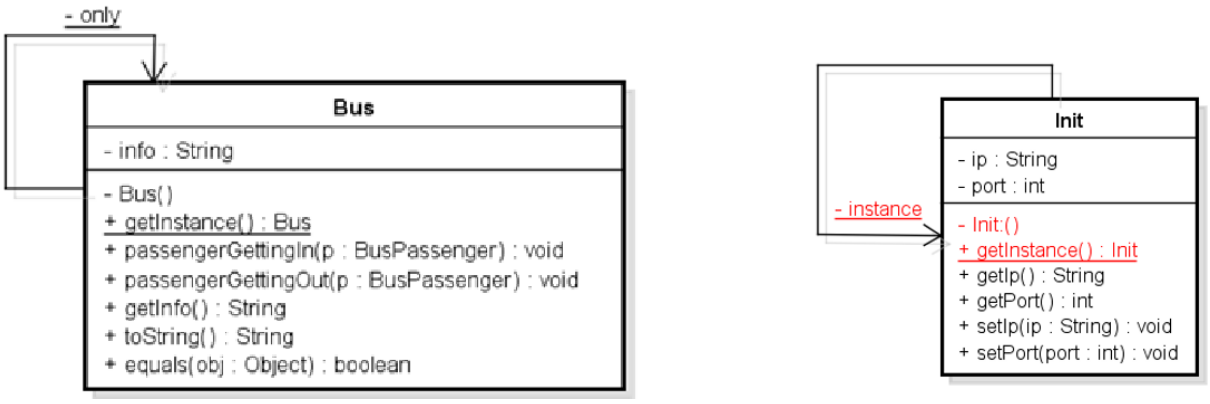
```
public ProxyTripList getTrips() {
    return trips;
}
```

- Or controllers call methods to get ObservableLists<> in order to display data in GUI

```
public ObservableList getObservableListOfBuses() {
    ObservableList<Bus> items = FXCollections.observableArrayList();
    for (Bus bus : busList.getArrayBuses()) {
        items.add(bus);
    }
    return items;
}
```

Singleton design pattern

General UML class diagram for singleton design pattern:



The overall purpose for the singleton design pattern:

- Singleton involves a single class which is responsible to create an object while making sure that only single object gets created. This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.

Criteria for singleton:

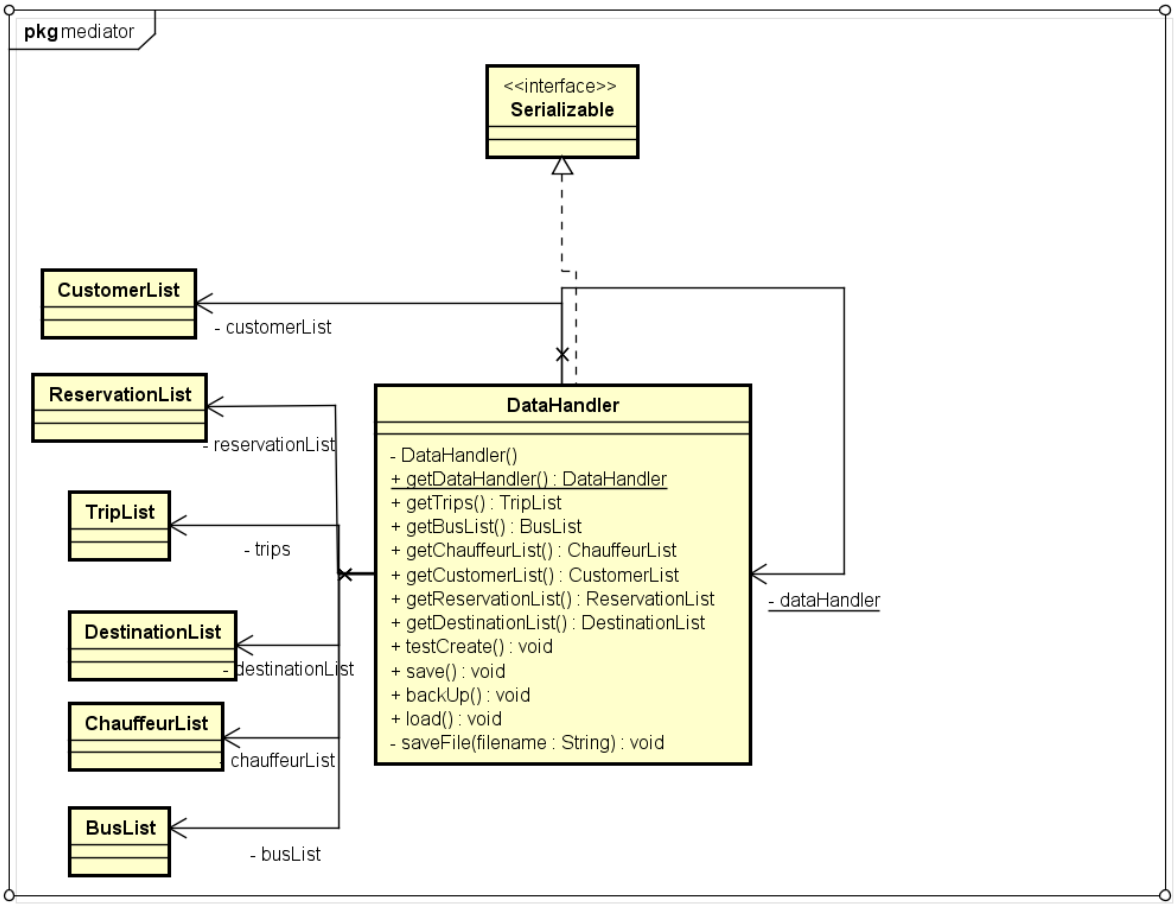
- private static Singleton instance - Private static class variable of the same type as the class that is the only instance of the class.
- private Singleton() – Private constructor to restrict instantiation of the class from other classes.
- public static Singleton getInstance() – Public static class method returning an instance of the class, this method has to ensure that only one instance is create. This is the global access point for outer world to get the instance of the singleton class.

Usage of singleton design pattern:

- Singleton pattern is mostly used in multi-threaded and database applications. It is used in logging, caching, thread pools, configuration settings etc.

- To use the singleton class, we need to have static member of class, private constructor and static factory method.
    - **Static member:** It gets memory only once because of static, it contains the instance of the Singleton class.
    - **Private constructor:** It will prevent to instantiate the Singleton class from outside the class.
    - **Static method:** This provides the global point of access to the Singleton object and returns the instance to the caller.

Our implementation:



Code example:

```
public class DataHandler implements Serializable {

    public static final char CLASSIC = 'c';
    public static final char LUXURY = 'l';
    public static final char PARTY = 'p';
    public static final char MINI = 'm';
    private static DataHandler dataHandler;
    private ProxyTripList trips;
    private BusList busList;
    private ChauffeurList chauffeurList;
    private CustomerList customerList;
    private ReservationList reservationList;
    private DestinationList destinationList;


    private DataHandler() {

    }

    public static DataHandler getInstance() {
        if (dataHandler == null) {
            dataHandler = new DataHandler();
        }
        return dataHandler;
    }
```

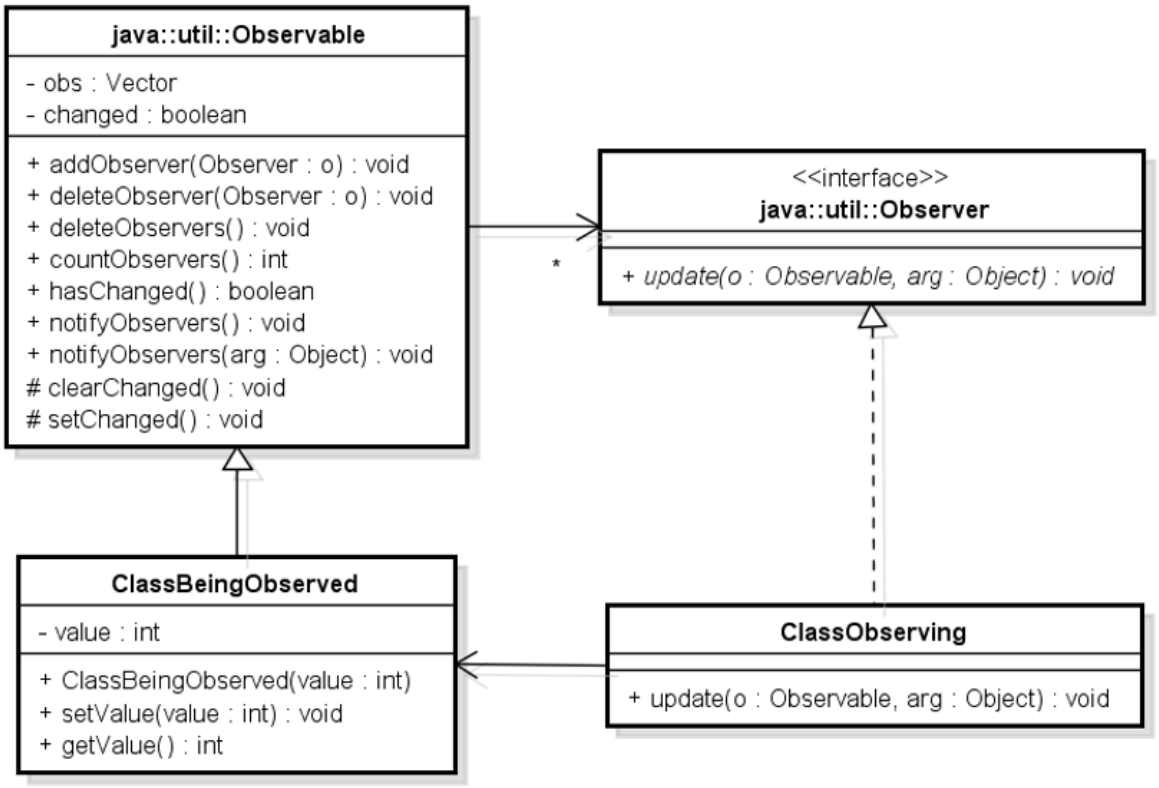
- We have inserted printout statement in the getInstance() method in which we are printing hashcode of datahandler object to the console. By doing this we can easily see if we are using only one instance of datahandler in the whole project.
- After running project and trying several parts of program, we got following output in the console:

```
    jar;C:\src\SDJ_Course\out\production\SDJ
    Main
 2 735164
 3 Server started
 4 735164
 5 735164
 6 735164
 7 735164
 8 735164
 9 735164
10
11 Process finished with exit code 0
12
```

- Number 735164 is the hashcode of the datahandler instance. Since the number is the same in all printouts, we consider test as success, we conclude that we have implemented singleton design pattern correctly.

Observer design pattern

General UML diagram for observer design pattern:



General purpose of observer design pattern:

- Observer automatically updates observing objects when one observable object changes state.
- One-to-many dependency is defined. All observing objects are notified when observed object changes its state.
- General algorithm of observer:
- 1) Subscribe to a service
- 2) Getting a message every time there is an update
- 3) Act upon the update

- In Java:
- 1) addObserver(observer)
- 2) setChanged() and notifyObservers(message)
- 3) observers implement method update(…)

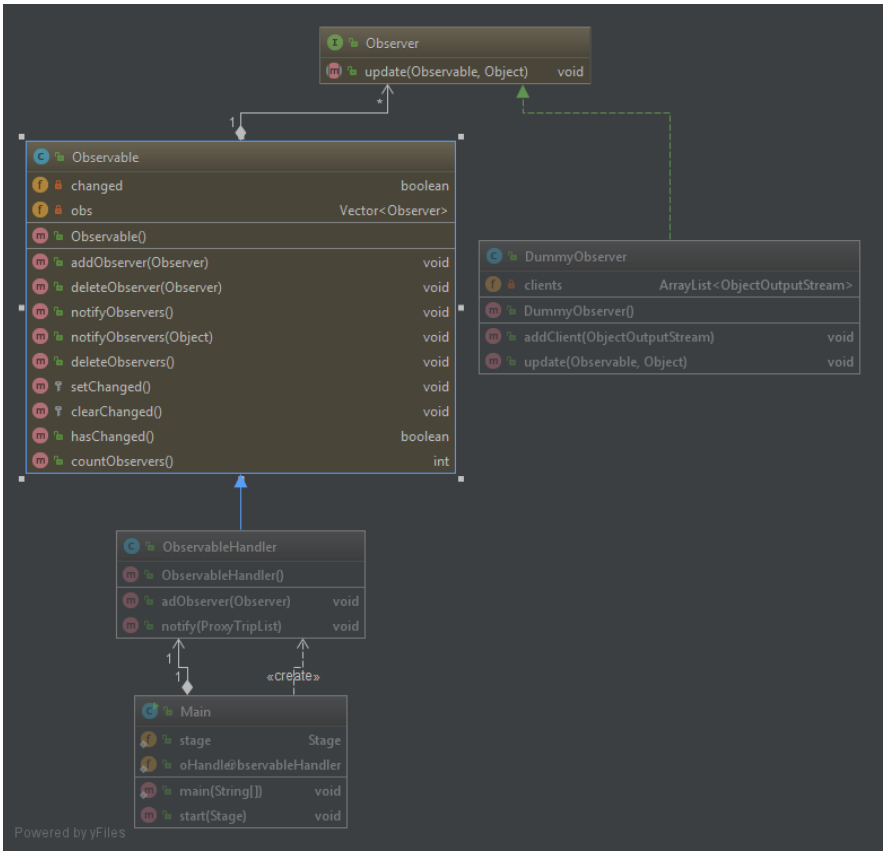
Description of UML diagram:

The pattern has two parts: a subject being observed and observers observing state change in the subject. The subject keeps a list of observers and when the subject's state changes the observers are being notified making a loop and calling method Update for each observer.

The Observer is an abstract interface with an abstract method Update to be implemented in the subclass ClassObserving. It is up to the ClassObserving what to do when a subject calls the update method but as indicated in the diagram this should get the updated state value.

The subject/observable part is the abstract class Subject containing a list of observers and with methods to add an observer to the list, remove an observer from the list and notify all observers, respectively. The subclass ClassBeingObserved handles the logic of the subject without direct information about the observers and with methods changing the state simply calling method Notify in its superclass Subject. The observable side is thereby nicely divided into a general part handling observers with methods independent on the actual subject being observed and a general part with specific information about the subject state.

Our implementation:



Code examples:

- ObservableHandler class

```
public class ObservableHandler extends Observable {

    public void addObserver(Observer ob){
        super.addObserver(ob);
    }

    public void notify(ProxyTripList trips){
        super.setChanged();
        super.notifyObservers(trips);
    }
}
```

- Update method from DummyObserver class

```
@Override
public void update(Observable o, Object arg) {
    for (int i=0;i<clients.size();i++){
        try {
            clients.get(i).writeObject(arg);
            clients.get(i).reset();
        } catch (IOException e) {
            //
        }
    }
}
```

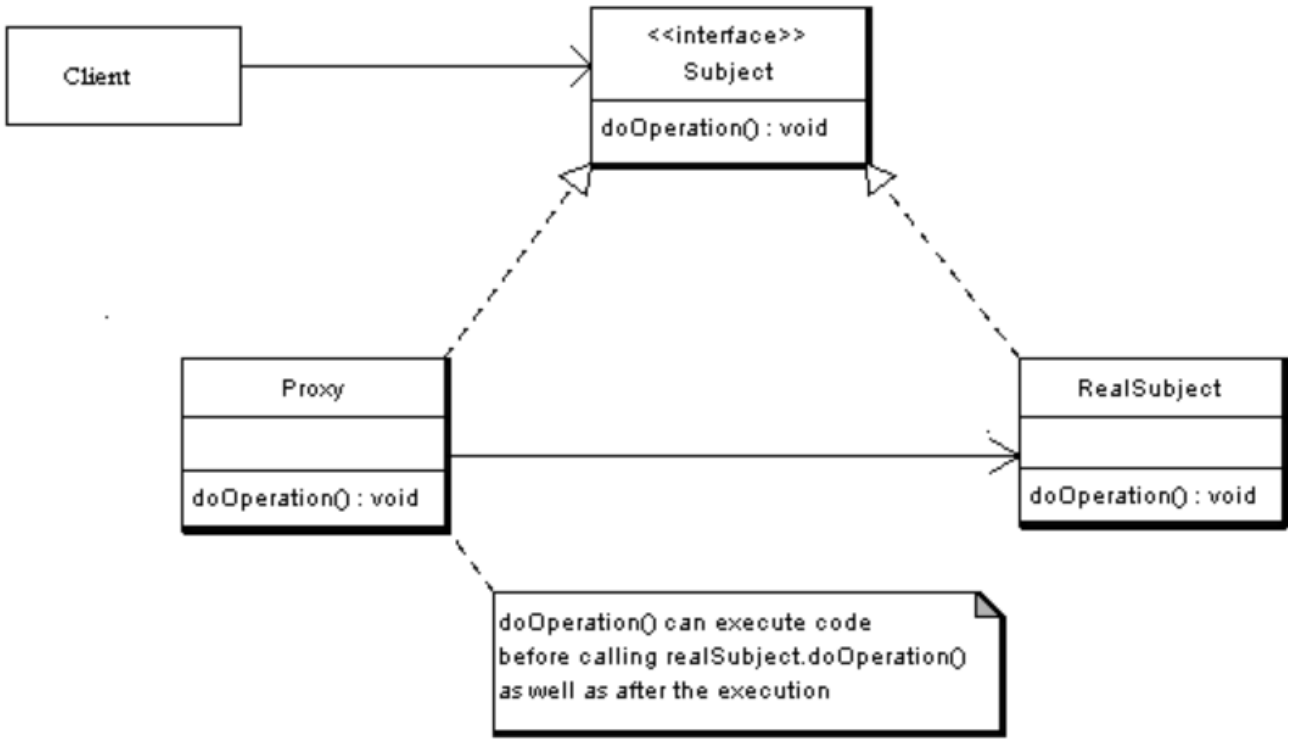
- Method addTrip from DataHandler responsible for triggering state change in observable class

```
public void addTrip(Bus bus, Chauffeur chauffeur, Destination pickUp,
Destination destination, int distance, LocalDate startDatePicker, String
fieldStartTime, LocalDate endDatePicker, String fieldEndTime, int price,
boolean food, boolean accommodation, boolean ticket) {
    Trip trip = new Trip(bus, chauffeur, pickUp, destination, distance,
startDatePicker, fieldStartTime, endDatePicker, fieldEndTime, price);

    if (food) {
        trip.setFood(true);
    }
    if (accommodation) {
        trip.setAccommodation(true);
    }
    if (ticket) {
        trip.setTickets(true);
    }
    trips.add(trip);

    Main.oHandler.notify(trips);
}
```

- DummyObserver is observing class which contains all clients connected to the server. Upon update it send updated trip list (arg) to all of them.

Proxy

General UML diagram for proxy design pattern:



Overall purpose for proxy design pattern:

- Proxy provides a placeholder for another object to control access to it. A wrapper to delegate work to a real subject.
- It can be a thread safe collection delegating work to another collection in synchronized methods, or a client object with access to a model on another computer.

General UML diagram description:

- The Subject is interface implemented by the RealSubject, representing its services. The interface must be implemented by the proxy as well, so that the proxy can be used in any location where the RealSubject is used.
- The proxy has an instance variable from RealSubject that allows the Proxy to access it. Implements the same interface implemented by the RealSubject, so that the Proxy can be

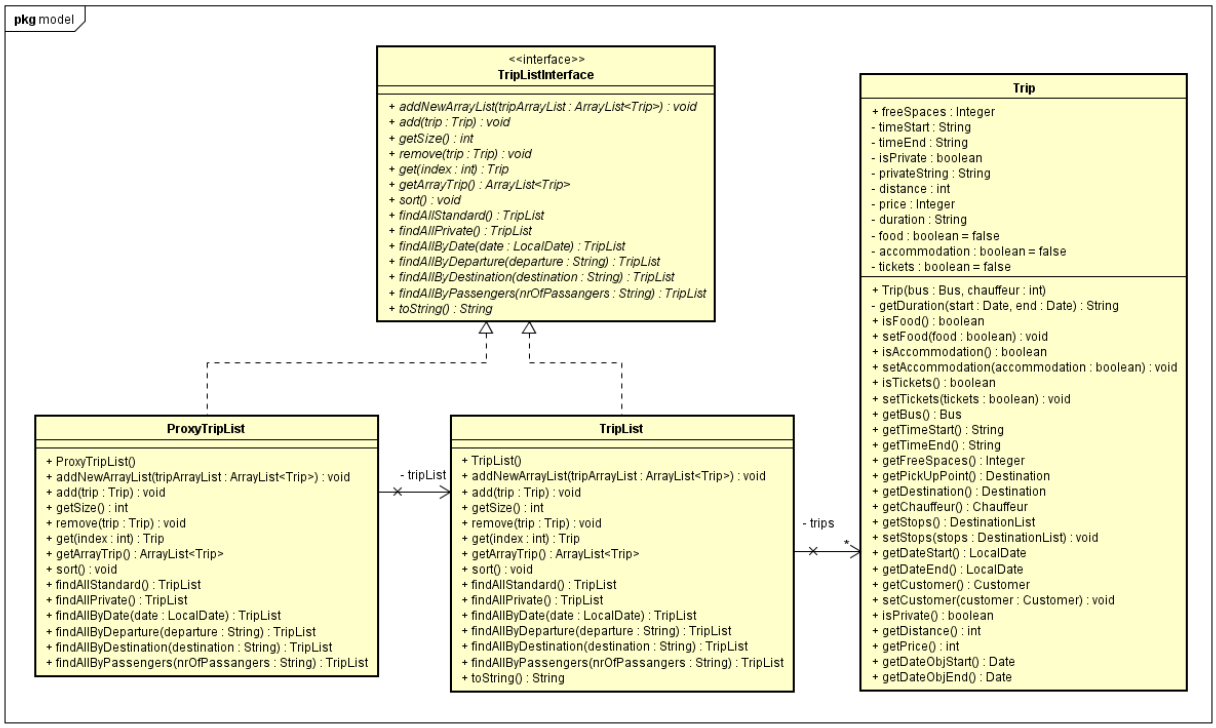
substituted for the RealSubject. Controls access to the RealSubject and may be responsible for its creation and deletion.

- RealSubject is the real object that the proxy represents.

Usage of proxy design pattern:

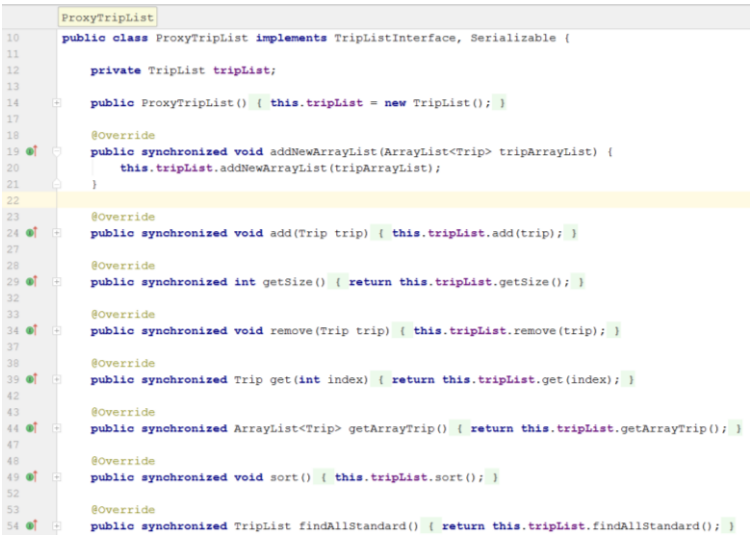
- A client obtains a reference to a Proxy, then the client handles the proxy in the same way it handles RealSubject and therefor invoking the method doSomething(). At that point the proxy can do different things prior to invoking RealSubject - doSomething() method. The client might create a RealSubject object at that point, perform initialization, check permissions of the client to invoke the method, and then invoke the method on the object. The client can also do additional tasks after invoking the doSomething() method, such as incrementing the number of references to the object.

Our implementation:



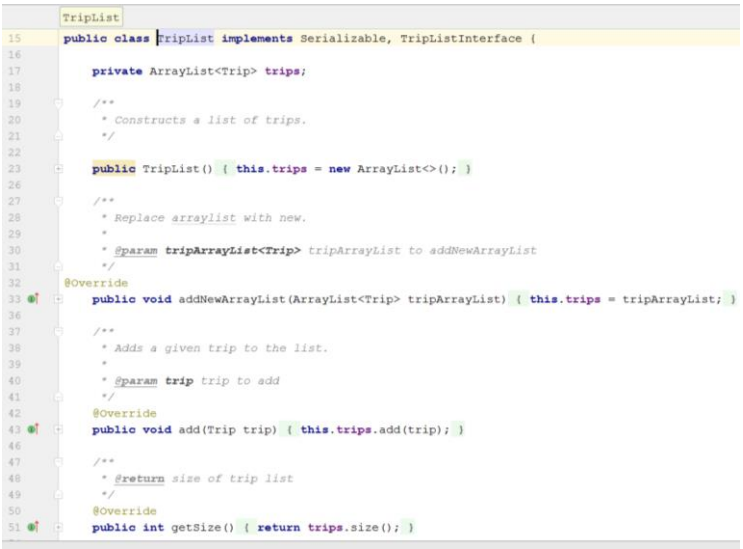
Code examples:

- Proxy class



```
ProxyTripList
public class ProxyTripList implements TripListInterface, Serializable {

    private TripList tripList;

    public ProxyTripList() { this.tripList = new TripList(); }

    @Override
    public synchronized void addNewArrayList(ArrayList<Trip> tripArrayList) {
        this.tripList.addNewArrayList(tripArrayList);
    }

    @Override
    public synchronized void add(Trip trip) { this.tripList.add(trip); }

    @Override
    public synchronized int getSize() { return this.tripList.getSize(); }

    @Override
    public synchronized void remove(Trip trip) { this.tripList.remove(trip); }

    @Override
    public synchronized Trip get(int index) { return this.tripList.get(index); }

    @Override
    public synchronized ArrayList<Trip> getArrayTrip() { return this.tripList.getArrayTrip(); }

    @Override
    public synchronized void sort() { this.tripList.sort(); }

    @Override
    public synchronized TripList findAllStandard() { return this.tripList.findAllStandard(); }
```

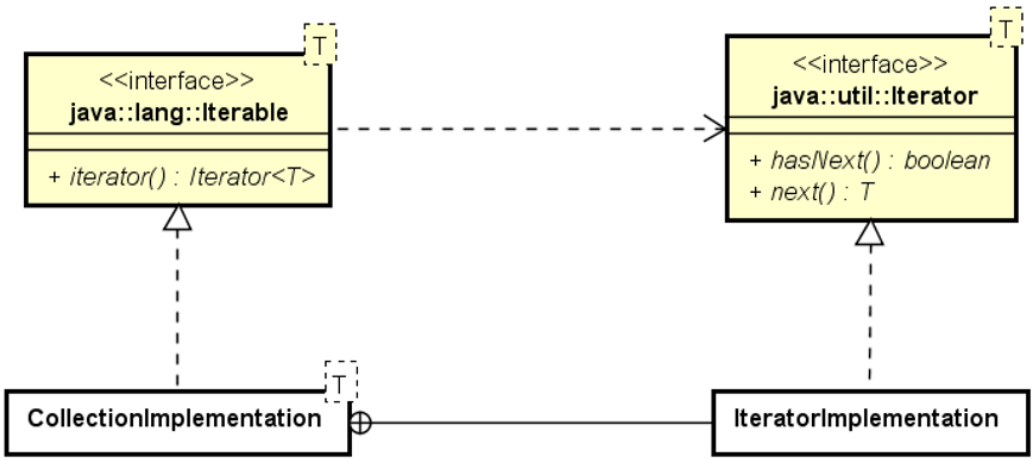
- real class



```
TripList
public class TripList implements Serializable, TripListInterface {

    private ArrayList<Trip> trips;

    /**
     * Constructs a list of trips.
     */

    public TripList() { this.trips = new ArrayList<>(); }

    /**
     * Replace arraylist with new.
     *
     * @param tripArrayList<Trip> tripArrayList to addNewArrayList
     */
@Override
    public void addNewArrayList(ArrayList<Trip> tripArrayList) { this.trips = tripArrayList; }

    /**
     * Adds a given trip to the list.
     *
     * @param trip trip to add
     */
    @Override
    public void add(Trip trip) { this.trips.add(trip); }

    /**
     * @return size of trip list
     */
    @Override
    public int getSize() { return trips.size(); }
```

Iterator design pattern

General UML diagram for iterator design pattern:



Overall purpose for the Iterator design pattern:

- An iterator is an object that provides the means to iterate over a collection. It provides methods that allow the user to acquire and use each element in a collection in turn. Most collections provide one or more ways to iterate over their elements.

Description of general UML diagram:

- The Iterator interface is defined in the Java standard class library. The two primary abstract methods defined in the Iterator interface are:
  - hasNext, which returns true if there are more elements in the iteration.
  - next, which returns the next element in the iteration.
- There is no assumption about the order in which an Iterator object delivers the elements from the collection. In the case of a list, there is a linear order to the elements, so the iterator would likely follow that order. In other cases, an iterator may follow a different order that makes sense for that collection.

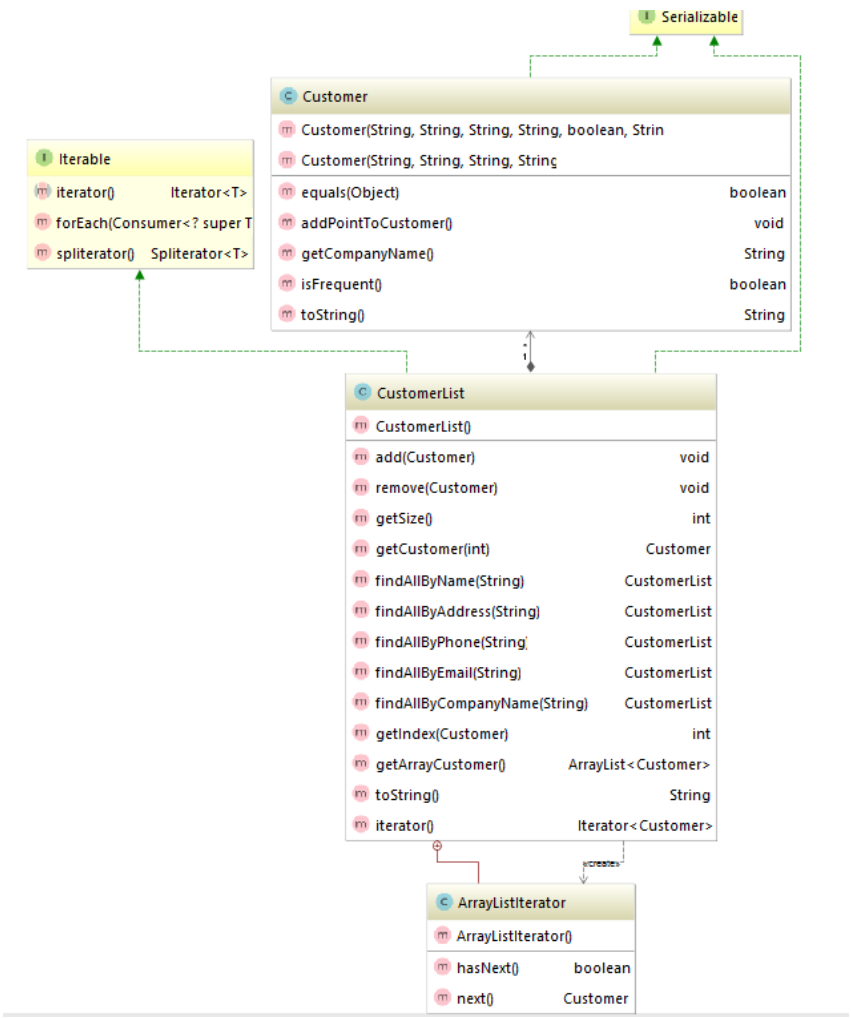
Usage of iterator design pattern:

- Including an Iterator method in the collection, it makes the collection Iterable or in other words, it implements the Iterable interface. The iterator method returns an object that implements the Iterator. The user can then interact with that object, using the hasNext and next methods, to access the elements in the list.

Remarks for iterator design pattern:

- Iterator method boolean hasNext() - returns true if the iteration has more elements. (In other words, returns true if next would return an element rather than throwing an exception.)
- Iterator method E next() - returns the next element in the iteration. Throws: – NoSuchElementException-iteration has no more elements.

Our implementation:



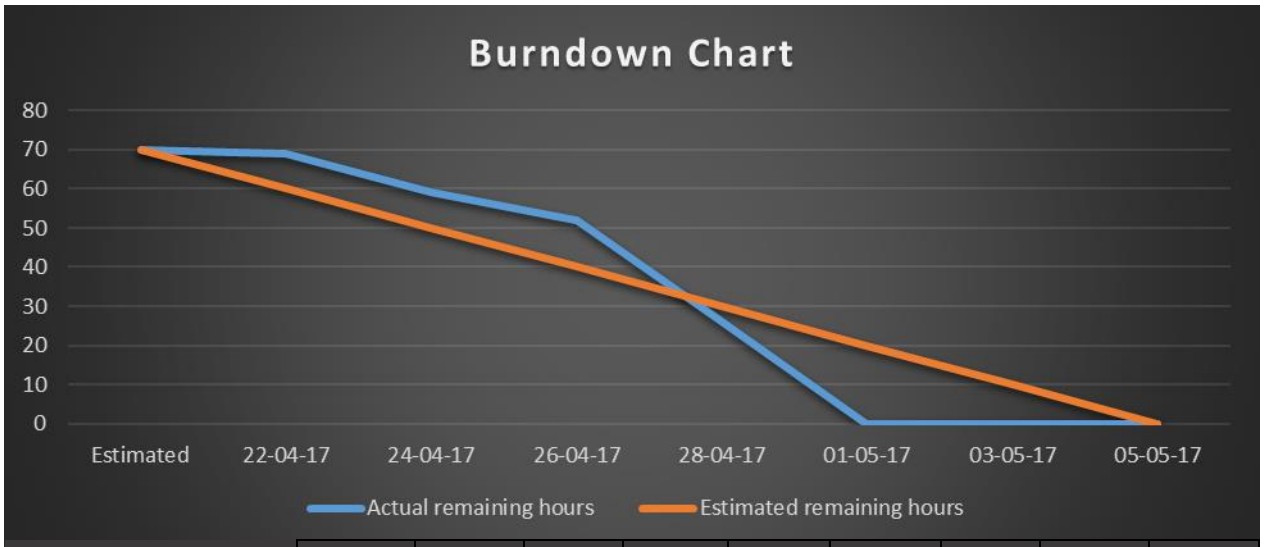
Code examples:

```
@Override
public Iterator<Customer> iterator() {
    return new ArrayListIterator();
}

private class ArrayListIterator implements Iterator<Customer>
{
    private int currentIndex;

    public ArrayListIterator()
    {
        this.currentIndex = 0;
    }

    @Override
    public boolean hasNext()
    {
        return currentIndex < getSize();
    }

    @Override
    public Customer next()
    {
        if (!hasNext())
            throw new IllegalStateException("Element not found");
        Customer customer = customers.get(currentIndex);
        currentIndex++;
        return customer;
    }
```

Scrum and UP usage

Product backlog and burndown chart

ID	Priority	Estimated time	Story
1	Critical	8	As user I want to connect to server
2	Critical	2	After connecting client to server, current trip list is displayed
3	Critical	18	Customer wants observer design pattern
4	Critical	5	Customer wants proxy design pattern
5	Critical	5	Customer wants iterator design pattern
6	Critical	5	Customer wants façade design pattern
7	Critical	5	Customer wants singleton design pattern
8	Critical	4	Customer require to test proxy
9	Critical	9	Customer wants MVC design pattern
10	Low	1	Different GUI on client computers
11	Medium	3	As client user I want to see a list of tours in a given date interval
12	Low	3	Display triplist in GUI on update
13	Low	2	Customer wants clean code without redundant methods and useless sysouts.



	Estimated	22-04-17	24-04-17	26-04-17	28-04-17	01-05-17	03-05-17	05-05-17	Total hours
Server connection	8					8			8
Show trip list when connected	2				2				2
Façade design pattern	5			5					5
Iterator design pattern	5				5				5
Proxy design pattern	5					5			5
Observer design pattern	18					18			18
MVC design pattern	9						9		9
Client GUI	1		1						1
Display triplist in GUI on update	3					3			3
Singleton design pattern	5			5					5
Display triplist in date interval	3						3		3
Junit test for proxy	4						4		4
Clean code	2						2		2
Actual remaining hours	70	69	59	52	26	0	0	0	
Estimated remaining hours	70	60	50	40	30	20	10	0	

Sprint Review – 22.04

100% completed work:

- Server and client classes made
- Server running
- Client able to connect, but doing nothing else
- GUI for client (decision to make it made on sprint planning) made.

Changes and new requests:

- Showing current trip list in GUI task was added to backlog.
- Decided to wait with documentation for completed client-server code.
- We decided that we will change class DataHandler into singleton.

Product owner decided to work fully with GUI, even if it means more work.

Sprint Retrospective – 22.04

PB	ID	Task title	Responsible	Estimated	Status
1	1	Design Client-Server architecture	All	1	Done
1	1	Implement server classes	Martin	2	Done
1	1	Implement client side	Martin	2	In progress
1	1	Test connecting to the server	Marek	1	Done
1	1	Document Client-Server architecture	Krzysztof	2	In progress
10	1	Create client side GUI	Marek	1	Done

This sprint started the Elaboration phase of UP.

In the first sprint we have decided to work on client-server architecture from our product backlog. Tasks were split between team members as shown in table. We have fully implemented the server side and the client side needed some improvements. The documentation for the client-server has been started. The GUI for client side has been created in this sprint too.

Sprint Retrospective – 24.04

PB	ID	Task title	Responsible	Estimated	Status
3	1	Implement Observer design pattern	Marek, Stela	3	In progress
3	2	Test Observer design pattern	Marek, Stela	2	In progress
6	3	Implement Façade design pattern	Martin	2	Done
6	4	Test Façade design pattern	Krzysztof	1	Done
6	5	Documennt Façade design pattern	Andreea	2	Done
7	6	Implement Singleton design pattern	Andreea	2	Done
7	7	Test Singleton design pattern	Andreea	1	Done
7	8	Document Singleton design pattern	Andreea	2	Done

This sprint started Construction phase of UP.

Tasks for this sprint has been planned according to the chart. The observer design pattern implementation has started; however, no working result was made. Façade design pattern has been implemented and tested. Documentation of façade followed. Static class DataHandler was changed to singleton design pattern, which was fully implemented, tested and documented during the sprint.

During this sprint we have found out that we are in need of new product owner, since Martin didn't have enough time courtesy of personal reasons. Team met before sprint review and we came to decision that the new product owner will be Andreea.

Sprint Review – 24.04

100% completed work:

- Façade deign pattern was implemented, tested and documented.
- Datahandler class was changed to singleton design pattern.
- Singleton was both tested and documented.

Changes and new requests:

- Decision to implement observer design pattern correctly, but not pretty.

After discussion with product owner it was decided that having correct and working observer design pattern is more important than having clear and easily understandable code and structure.

Sprint Review – 26.04

100% completed work:

- Final implementation and testing of observer.
- Sending trips to client upon connecting.
- Iterator design pattern finished.

Changes and new requests:

- By doing the MVC documentation we have found out that MVC from last project was implemented incorrectly and it needs to be redone.
- We got stuck upon showing any trip lists in GUI and we couldn't find the mistake.

Product owner decided to not care too much about showing list in GUI and show them in console for now. Also, product owner decided to update all classes which don't follow MVC rules.

Sprint Retrospective – 26.04

PB	ID	Task title	Responsible	Estimated	Status
3	1	Implement Observer design pattern	Marek, Stela	7	Done
3	2	Test Observer design pattern	Marek, Stela	3	Done
3	3	Document Observer design pattern	Stela	4	Done
2	4	Send trip list to client when connected	Martin	2	Done
12	5	Display updated trip list on client GUI	Marek	3	In progress
5	6	Implement Iterator design pattern	Andreea	2	Done
5	7	Test Iterator design pattern	Andreea	1	Done
5	8	Document Iterator design pattern	Andreea	2	Done
9	9	General documenntation of MVC	Krzysztof	2	In progress

The observer design pattern was fully implemented. By comparing observer classes in our project with the official ones we came to conclusion that the observer design pattern was designed and implemented properly. Documentation for observer was also finished. After connecting client to server, the current trip list form the server is sent to client. The implementation for showing new trip list after creating new trip on server has started. Iterator design pattern has been implemented, tested and documented. Documentation for MVC design pattern was created and started.

Sprint Review – 28.04

100% completed work:

- Packages for MVC.
- Rewrite and fix all classes which did not follow MVC rules.
- Testing of MVC design pattern.
- All documentations were updated to follow newly made MVC.
- Implementation, testing and documentation of proxy design pattern.

On the sprint planning we found out that when we change packages and rewrite a lot of code we will need to update all our previously made documentations. Product owner decided to do it in this sprint.

Sprint Retrospective – 28.04

PB	ID	Task title	Responsible	Estimated	Status
9	1	Create packages for MVC design pattern	Krzysztof	1	Done
9	2	Update controller classes to folow MVC	Marek	4	Done
9	3	Test MVC design pattern	Stela	1	Done
9	4	Document MVC design pattern	Krzysztof	3	In progress
4	5	Implement Proxy design pattern	Andreea	2	Done
4	6	Test Proxy design pattern	Andreea	1	Done
4	7	Document Proxy design pattern	Andreea	2	Done
0	8	Update previous documentations to follow MVC	All	5	Done
3	9	Fix sending and displaying updated trip list	Martin	3	Not done

In this sprint we have created packages for MVC design pattern. We had to rewrite a lot of code which we have copied from SEP1 and was not following MVC rules. After that we tested MVC by applying rules on our pattern and trying to find any mistakes. Documentation for MVC was not finished however. Andreea also implemented a proxy design pattern, which was tested and documented afterwards. All previously made documentations were updated to follow new code or packages. We have planned to fix sending and displaying trips in GUI but we didn’t manage to complete this task.

Sprint Review – 01.05

100% completed work:

- All list are displayed in GUI
- Junit test for proxy design pattern.
- Client-Server documentation
- MVC documentation
- Cleaning the code

Changes and updates:

- Cleaning the code task added to product backlog

During this sprint we have finally finished the documentation. After discussion with product owner we decided that it would be nice to run inspection on the code and clean it little bit. Martin with Stela did the Junit testing for proxy.

Sprint Retrospective – 01.05

PB	ID	Task title	Responsible	Estimated	Status
11	1	Implement showing trips in given date interval	Marek	3	Done
8	2	Junit test for proxy design pattern	Martin, Stela	4	Done
11	3	Fix displaying trips in date interval in GUI	Martin	2	Done
1	4	Finnish Client-Server documentation	Krzysztof, Marek	2	Done
9	5	Finnish MVC documentation	Martin, Stela	2	Done
13	6	Delete reduntant methods and sysouts, clean code	Andreea	2	Done

This sprint finished the Construction phase of UP.

During this sprint we have finished all the coding tasks. We have created Junit test to test proxy design pattern. We have also finished client-server and MVC documentations. After all this was finished we ran inspection on the code using which we have found and deleted all reduntant methods and sysouts from the code.

Sprint Review – 03.05

The system was fully finished during the sprint. All parts were tested and approved.

The discussion was held in team during which we have discussed what to do now, since we are ahead of schedule and finished. The decision to take last day (sprint 05.05) off and to upload project before deadline was made.

Sprint Retrospective – 03.05

PB	ID	Task title	Responsible	Estimated	Status
		Final touches			
		Final documentation			

This sprint started and ended the Transition phase of UP.

During this sprint we did final documentation for project, final tests and touches.