Bilkent University

Department of Computer Engineering

CS 319 - Object Oriented Programming Term Project

RSim: Railway Simulator

Design Report

Semahat Elif Dayı, Zahit Saygın Doğu, Mevlüt Geredeli, Gizem Uzuner

Instructor: Ertuğrul Kartal Tabak

Analysis/Final Report

March21, 2015

This report is submitted to the Department of Computer Engineering of Bilkent University in partial fulfillment of the requirements of the Object Oriented Programming course CS319.

Table of Contents

Introduction

Simulation is a key application of estimating the future states of the systems that are working on a structured basis. Computer aided simulators helps people to analyze complicated systems and react on the real systems using the confidence from the simulation, knowing that the system will work properly after its construction is complete or it won't collapse if some properties of the system is changed.

Railway systems can be very complex and it can be hard to estimate the effects of the design decisions or changes in the system. If there would be a convenient railway simulation software that could ease the work of the designers and maintainers of such systems. Therefore we decided to design and implement a railway simulator software called RSim, which provides a tool for designing railway systems and simulating the given system to observe how the system works. This simulator allows user to create new railway route and edit existing routes. User can add new stations, trains and edit their properties. Eventually, railway system is simulated and user can observe system in action.

Our aim while creating RSim is to show how a railway system works and enable users to build efficient railway systems or create more efficient timetables or plans by simulating and observing the effects of the changes. User can create different variation of routes and observe what will happen at the desired time. Hence, once simulated with satisfying results, the system can be constructed or changes in an existing system can be implemented.

This report contains the overview of the simulator, specifies the simulator's requirements, analysis and design of the system. This report includes the architectural patterns which will be used in our simulator software and current system that is familiar with our project. Moreover, report gives information about functional requirements, non-functional requirements, pseudo requirements, use-case models that include scenarios, use-case diagrams, object and class model, dynamic model, which are a part of the analysis stage. At the end of the report, there are mockups of user interface of the RSim software.

Current System

We have made an online search for existing railway simulator software and we have found a satisfying software called OpenTrack. Here is the explanation of the software website:

OpenTrack[2] began in the mid-1990s as a research project at the Swiss Federal Institute of Technology. The aim of the project *Object-Oriented Modeling in Railways* was to develop a catalyst for practical economic solutions to complex railway technology problems.

Today, the railway simulation tool OpenTrack is a well-established railway planning software and it is used by railways, the railway supply industry, consultancies and universities in different countries.

OpenTrack allows modeling, simulating and analyzing the following types of rail systems:

- High speed rail
- Heavy rail / Intercity rail
- Commuter rail systems
- Heavy haul freight
- Mining railway systems
- Metro / Subway / Underground systems
- Light rail (LRT)
- Tram / Streetcar systems
- People mover systems
- Rack railways / Mountain railways
- Maglev (magnetic levitation) systems (e.g. Transrapid)

OpenTrack supports the following kinds of tasks:

- Determining the requirements for a railway network's infrastructure
- Analyzing the capacity of lines and stations
- Rolling stock studies (for example, future requirements)
- Running time calculation
- Timetable construction; analyzing the robustness of timetables (single or multiple simulation runs, Monte-Carlo simulation)
- Evaluating and designing various signaling systems, such as *discrete block systems*, *short blocks*, *moving blocks*, *LZB*,*CBTC (communication-based train control)*, *ATP*, *ATO*, *ETCS Level 1*, *ETCS Level 2, ETCS Level 3* (see also: ***ERTMS***)
- Analyzing the effects of system failures (such as infrastructure or train failures) and delays
- Calculation of power and energy consumption of train services
- Simulation of railway power supply systems (using **OpenPowerNet**)

Requirements Analysis

While the existing software OpenTrack provides a vast amount of functionalities, the complexity of the system makes it hard to learn and start using it. Our software RSim won't be as complicated as OpenTrack and provide considerably less functionalities. However, the main goal of the RSim is to learn the software development stages for the team so we will implement the project.

Overview

The purpose of this project is to simulate a railway system. For simplicity there is only one line with no cross sections. User can add a new station, specify the distance between the stations, edit and remove the stations if necessary. User can also specify the capacity of the trains, the frequency of the trains and number of the trains. User should also specify the average quantity of people coming to a specific station, waiting and times for the trains for that station. After everything is specified user can advance time to see how full the trains and stations are and how the system works. Then user can modify the parameters and observe the system so that the user can achieve optimal operation strategy for the system.

We are planning to realize our design by using Java programming language. We are planning to build a simple GUI for adjusting parameters and showing the state of the system. However, since it will exceed the scope of the course adding any animations or graphical representations for the line, stations and trains will be considered to be added at the end of the implementation.

We decided to pick this topic for the project because we see high potential for expanding. If the project won't satisfy the requirements for the course, we can always add new features.

Functional Requirements

There are main functionalities of the RSim. One of them is to design and model a railway system to be able to observe simulations on it. After the modelling is done, the user can simulate the behaviour of the system. The functional requirements for RSim can be organized in three main sections the requirements related to the sections. The list of all functional requirements are given below:

- Authentication
    - User can log in to system.
    - User can open and simulate a saved system by any user.
    - User cannot edit a system saved by other users.
- Railway System Design
    - User should add new stations to the system.
    - User should define the distances between the stations.
    - User should define train specifications (such as speed, capacity...)
    - User can change the station properties at edit time.
    - User can save the layout of the stations to a file.
- Simulation

- User is not allowed to modify the properties of trains and stations during simulation time.
- User can see the system state after some defined time by the user.
- User can view the simulation as time passes.
- Controller won't let trains to be overlapped. This means, in between two adjacent stations there can be only one train at a time instant.

Non-functional Requirements

- All user inputs should be acknowledged within 1 second.

  The responsiveness of a system is important. Therefore we decided to include this non-functional requirement for user convenience.

- A system crash should not result in long term data loss.

  The progress of the activity of the user is important. We will introduce a saving machanism to achieve this goal.

- Any simulation calculation taking more than 1 minute will terminate and result in warning.

  Since the simulations can be quite complicated and the time interval that the user states can be too large, there will be a requirement that at maximum the time it takes to calculate the simulation process won’t exceed 1 minute. Otherwise the simulation will terminate and the user will be warned about the situation.

Pseudo Requirements

- The System will be implemented on Java. Because Java is one of the best object oriented programming languages and it provides nice UI API's to make platform independent applications.

Scenarios

Ertuğrul has a childhood dream of creating a virtual train system and simulation its behavior. Ertuğrul finds out there is software called Railway Simulator for this. Ertuğrul gets the program, starts it. Ertuğrul sees the user interface. Ertuğrul first creates an account for the program. After that he logs in to the system and starts editing. He first creates a track by using create track button. He creates a track initially with three stations. He decides he wants to add some more stations and adds them by using add station button. After that he creates a train dispatcher at the beginning of the track. After this he creates 3 trains using add train button. He specifies the destination and the departure hour vector of the trains. After that he is satisfied with the design of the railway system so he decides to simulate it. He pushes the simulate button and watches the simulation for hours. After a while he gets bored and decides to close the program. He stops the simulation and saves state of the railway design. After that he closes the program. After closing the program he gets curious and opens the program with another account to see what happens. He locates the design file to open it. He is able to see the design, simulate it but he notices that he can't change the design. It is because he is not logged in as his original profile.

Use-Case Model

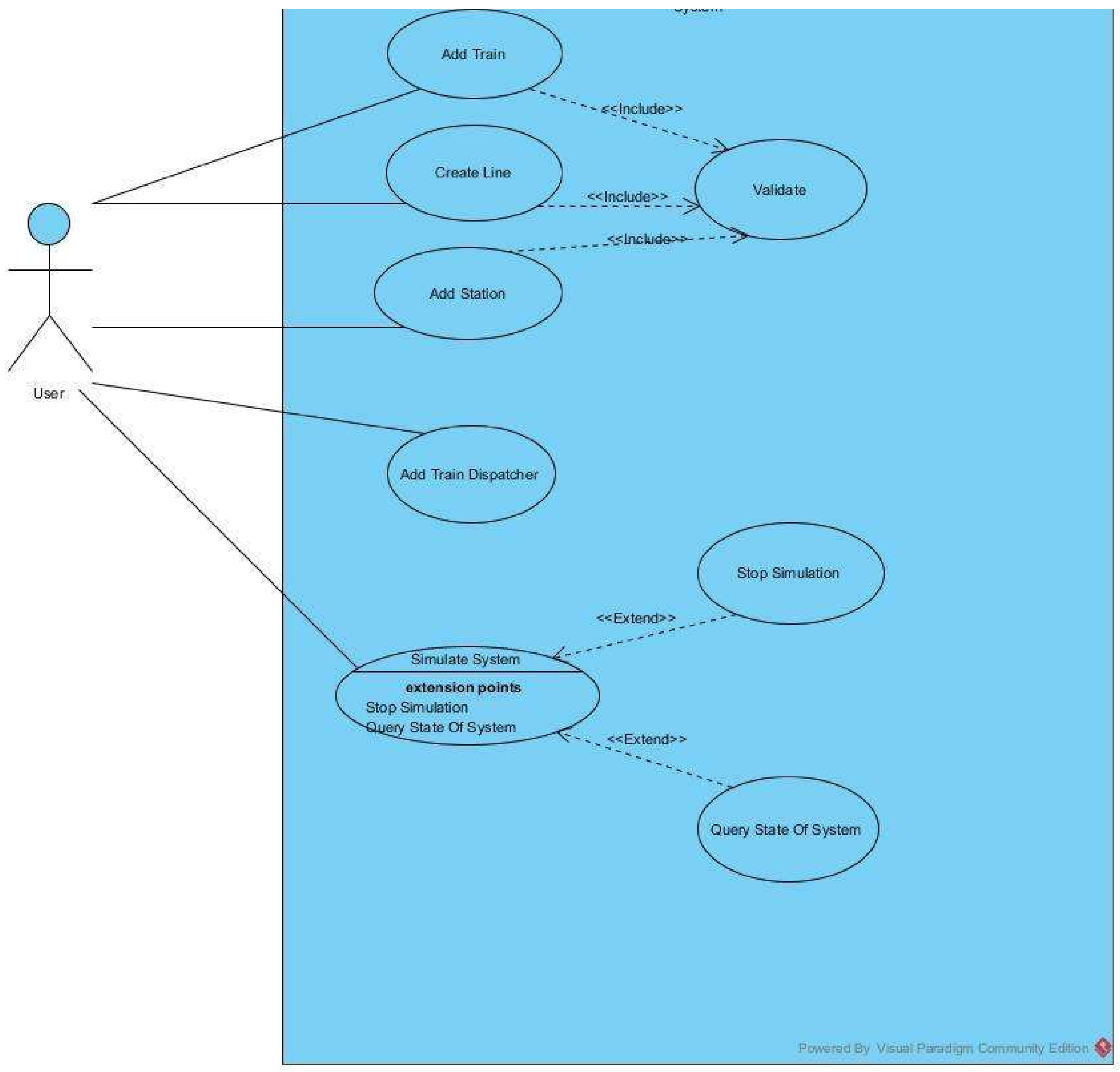


Figure 1: Use Case Diagram

In the Figure 1 use case diagram of RSim is given. The corresponding textual descriptions of the use cases are given below.

Use Cases

Adding Station

Use Case Name: Adding Station
Participating actors: User (Ertuğrul)
Entry Conditions:

- Program is running.
- There is no simulation in progress.
- User is logged in.
- There is a track existing.

Flow of Events:

- Ertuğrul presses the button for adding new station.,
- Ertuğrul specifies the properties of the station.
    - Ertuğrul can specify the name of the station.
    - Ertuğrul can specify the position of the station.
    - Ertuğrul can specify the maximum wagon capacity of the station.

If the station is overlapping with another station, or it is conflicting with the line specifications, Ertuğrul will see a warning.

If there is no warning Ertuğrul can confirm the addition of the station.

The station will be added to the line system.

Exit Condition:

- The station is added, or the station addition process is aborted.

Creating a New Track

Use Case Name: Creating a new track
Participating actors: User (Ertuğrul)
Entry Conditions:

- Program is running.
- There is no simulation in progress.
- User is logged in.
- There is no track existing.

Flow of Events:

- Ertuğrul presses the button for adding new line.
- Ertuğrul specifies the properties of the line.
    - Ertuğrul can specify the minimum distance between stations.
    - Ertuğrul can specify the maximum length of the track.
    - Ertuğrul can specify the maximum station count in the line.
    - Ertuğrul can specify the name of the line.
    - Ertuğrul can specify the name of the first station.
    - Ertuğrul can specify names and the positions of the following stations.

If any station is overlapping with another station, or it is conflicting with the line specifications, Ertuğrul will see a warning.

If there is no warning Ertuğrul can confirm the creation of the line.

The track will be added to the line.

Exit Condition:

- The new track is added or the track addition process is aborted.

Adding a Train

Use Case Name: Adding a train
Participating actors: User (Ertuğrul)
Entry Conditions:

- Program is running.
- There is no simulation in progress.
- User is logged in.
- There is a track existing with at least 2 stations.

Flow of Events:

- Ertuğrul presses the button for adding new train.
- Ertuğrul specifies the properties of the train.
  - Ertuğrul can specify which train dispatcher the train will start it's service.
  - Ertuğrul can specify the capacity of the train.
  - Ertuğrul can specify the count of the wagon.
  - Ertuğrul can specify the maximum capacity of each wagon.
  - Ertuğrul can specify the departure time table for the train.
  - Ertuğrul can specify the directions which the train will go at each departure time.

If the wagon count is more than the stations can handle, the program will give a warning.
If there is no warning Ertuğtul can confirm the addition of the train.
The train will be added to the line system.
Exit Condition:

- The train is added to the dispatching queue of the dispatcher.

Add a Train Dispatcher

Use Case Name: Add Train Dispatcher
Participating actors: User (Ertuğrul)
Entry Conditions:

- Program is running.
- There is no simulation in progress.
- User is logged in.
- There is a track existing with at least 2 stations.

Flow of Events:

- Ertuğrul presses the button for adding new traindispatcher.
- If there is a dispatcher on the station, Ertuğrul will see a warning message.
- Ertuğrul specifies the properties of the train dispatcher.
    - Ertuğrul must specify which station the train dispatcher will work at.
    - Ertuğrul can specify which way the train will move initially.

Exit Condition:

- The train dispatcher is added to the station, or the existing dispatcher is overridden.

Simulate System

Use Case Name: Simulate the System
Participating actors: User (Ertuğrul)
Entry Conditions:

- Program is running.
- There is no simulation in progress.
- User is logged in.
- There is a track existing with at least 2 stations.
- There is at least one train dispatcher on one of the stations.

Flow of Events:

- Ertuğrul presses the button for starting the simulation.
- Ertuğrul will see the simulation results in the screen.
    - Ertuğrul can adjust the speed of the simulation.
    - Ertuğrul can stop the simulation.

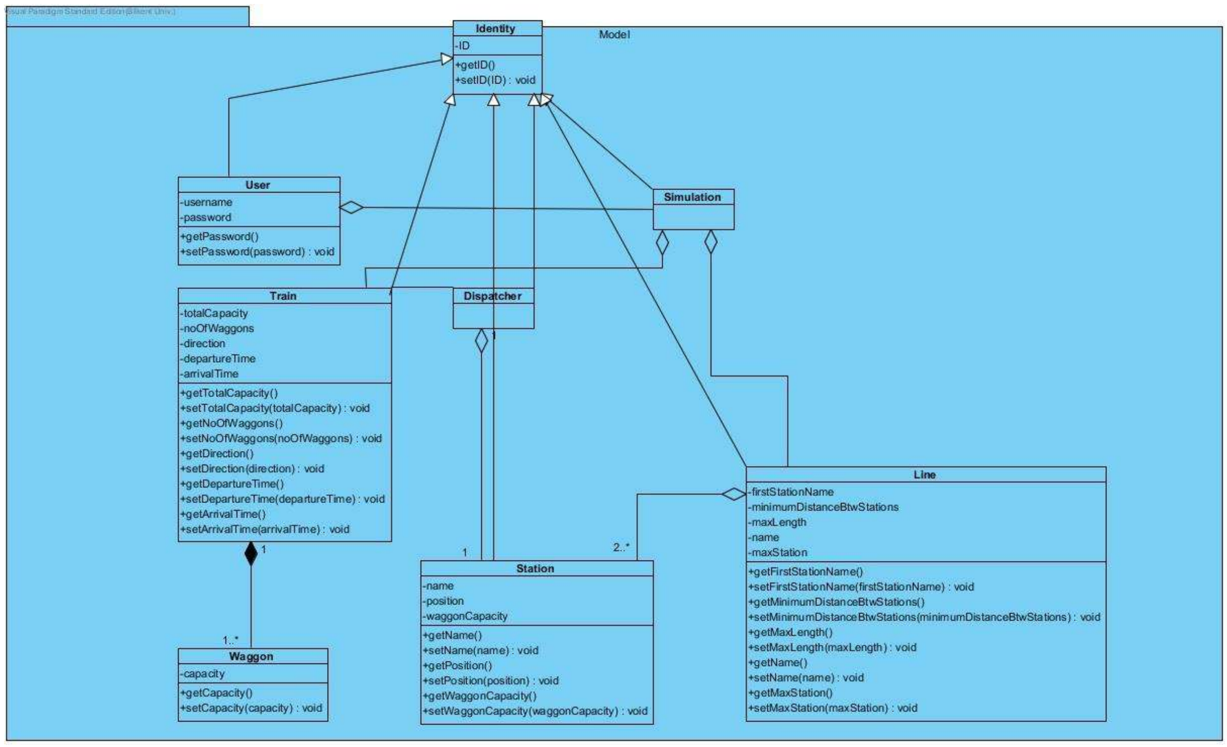
After stopping Ertuğrul can save the simulation process.
Exit Condition:

- The simulation process is either saved or nothing happens.

Analysis Models

Class Model

The initial class diagram of the system is given below. The classes are organized in a way that they will be addressing the requirements for the system. Most of the classes in this stage are the the problem domain objects that are mostly the entity objects. There are no implementation specific objects in this stage. In the following stages of the projects implementation specific objects and classes might be added to the system.



Model
Identity
-ID
+getID()
+setID(ID) : void
User
-username
-password
+getPassword()
+setPassword(password) : void
Simulation
Train
-totalCapacity
-noOfWaggons
-direction
-departureTime
-arrivalTime
+getTotalCapacity()
+setTotalCapacity(totalCapacity) : void
+getNoOfWaggons()
+setNoOfWaggons(noOfWaggons) : void
+getDirection()
+setDirection(direction) : void
+getDepartureTime()
+setDepartureTime(departureTime) : void
+getArrivalTime()
+setArrivalTime(arrivalTime) : void
Dispatcher
Line
-firstStationName
-minimumDistanceBtwStations
-maxLength
-name
-maxStation
+getFirstStationName()
+setFirstStationName(firstStationName) : void
+getMinimumDistanceBtwStations()
+setMinimumDistanceBtwStations(minimumDistanceBtwStations) : void
+getMaxLength()
+setMaxLength(maxLength) : void
+getName()
+setName(name) : void
+getMaxStation()
+setMaxStation(maxStation) : void
Station
-name
-position
-waggonCapacity
+getName()
+setName(name) : void
+getPosition()
+setPosition(position) : void
+getWaggonCapacity()
+setWaggonCapacity(waggonCapacity) : void
Waggon
-capacity
+getCapacity()
+setCapacity(capacity) : void
1
1..*
1
2..*

Figure 1 Class Diagram of Model Classes

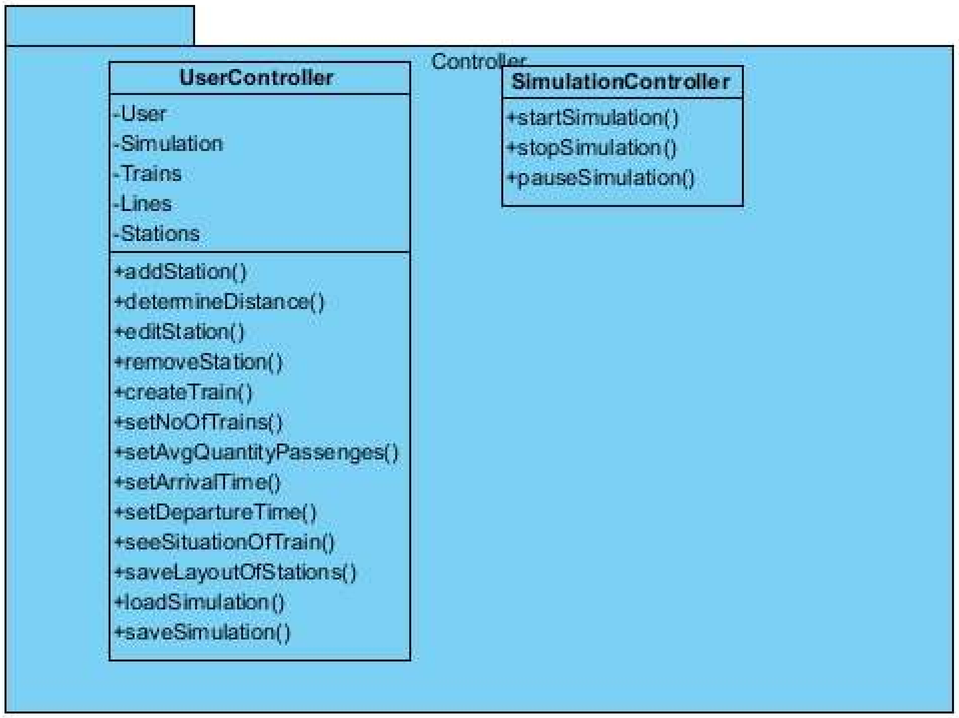


Figure 2 Class Diagram of Controller Classes

State Diagrams

New File Creation Diagram

Figure 3 represent how user can creates a new file. Simulator saves railway systems as a file. Firstly, user creates new railway system and s/he must add new station. Station is a major requirement for railway system. After adding new station, user can modify properties of station. If user wants to add more station, s/he can continue adding. However, if user is done with system design and s/he wants to simulate, s/he can run simulator and user can get results. After simulation, user have both choice which are user is allowed to edit system or save system.

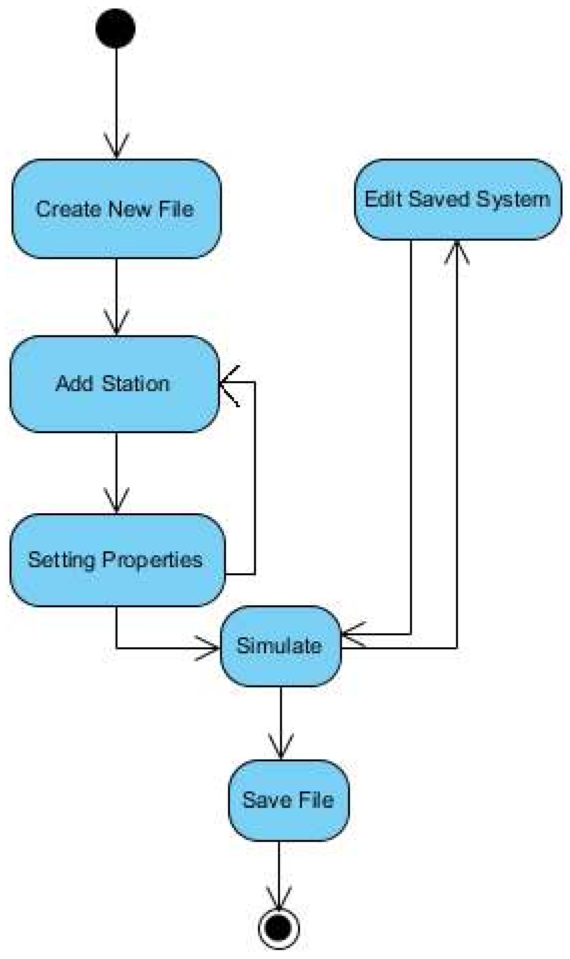


Figure 3 Create New File State Diagram

Open File Belonging Another User

In this railway simulator program, users are allowed to see another users' files. Even though they can see systems, they are not allowed to edit these system. They are only capable to see another users' systems. Initially, user must be login and s/he chooses existing file belonging another user. In Figure 4 this functionality's state diagram is given.

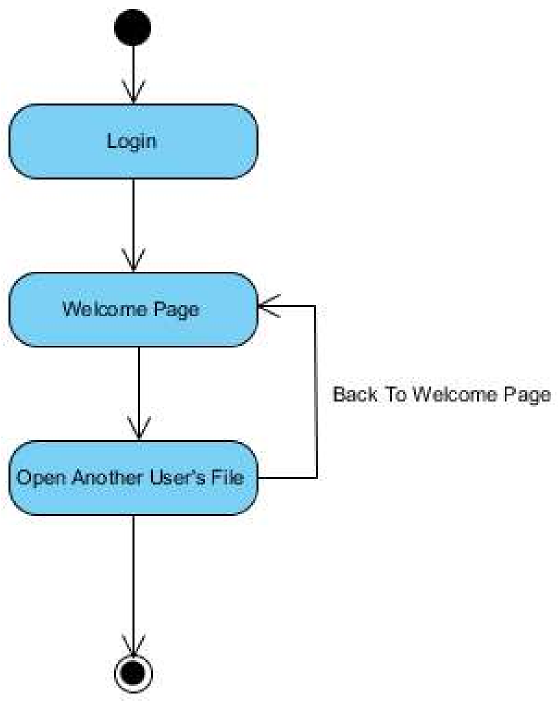


Figure 4 Open File Belonging to Other User State Diagram

Remove Desired Train

Figure 5 explains how user can remove desired train in system. At the system design stage, if user wants to remove one of the trains, s/he should reach the system that includes desired train. When the system is open, user can choose any train as his wish. Therefore, train is removed and simulator allows user to continue system design.

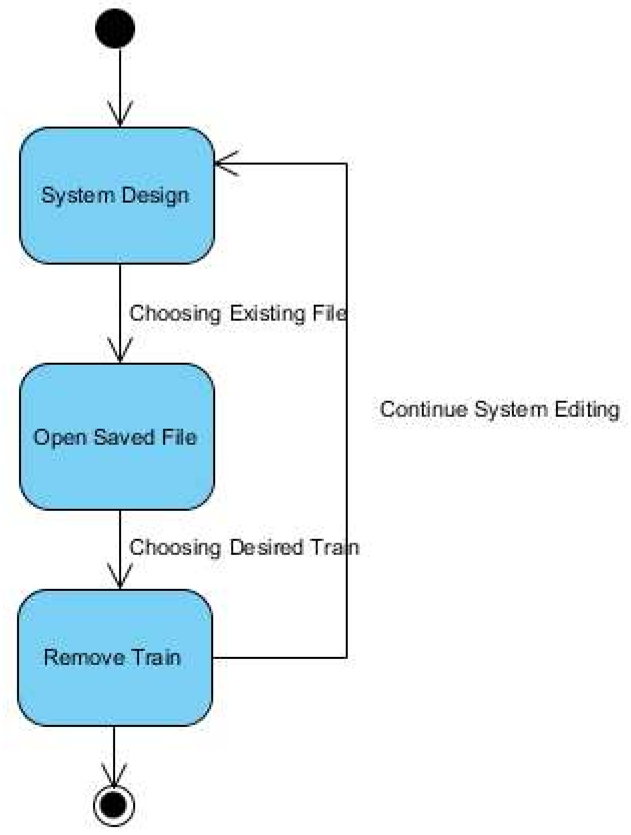


Figure 5 Remove Desired Train State Diagram

Dynamic Models

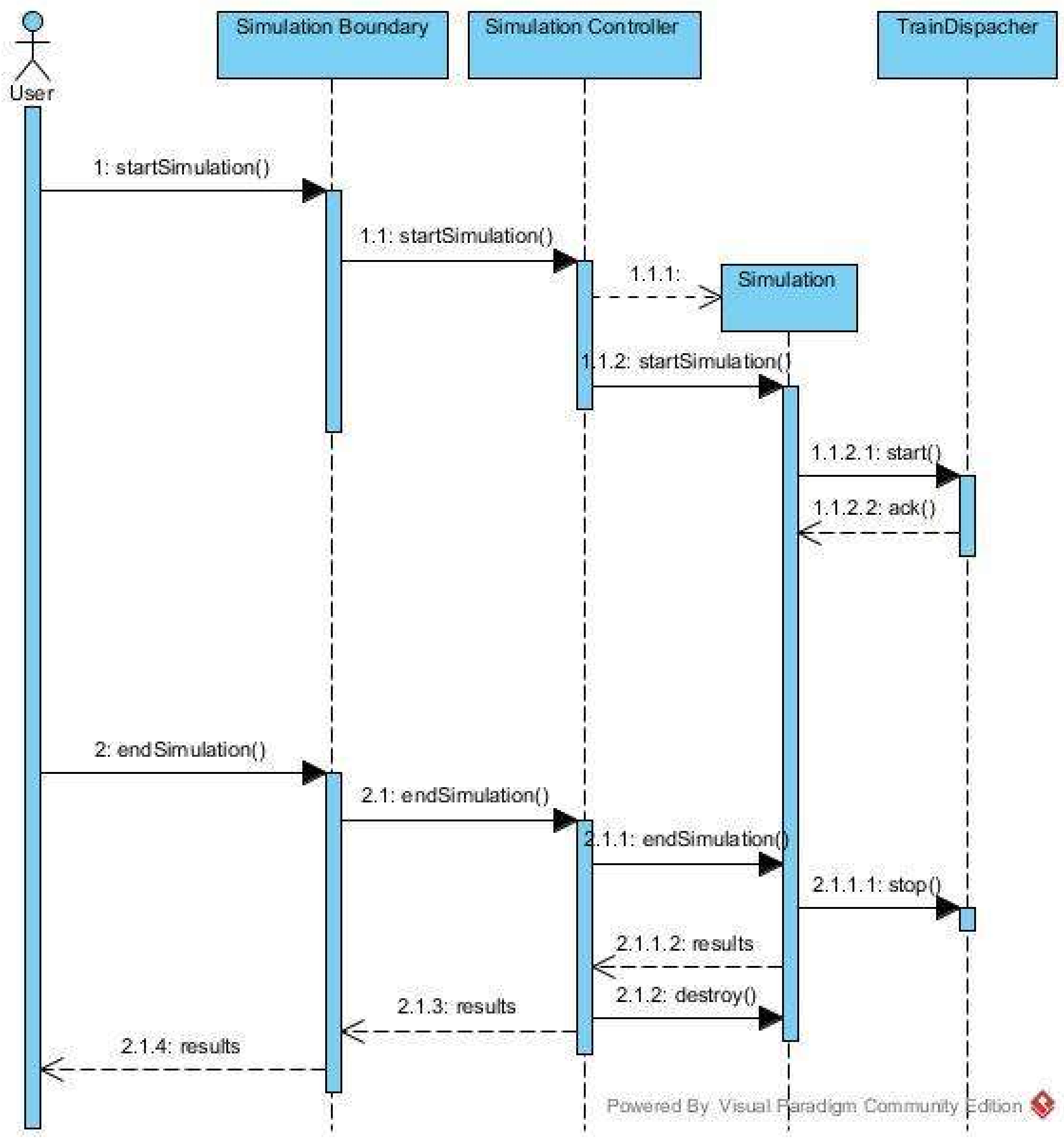


Figure 6 Sequence Diagram for Simulating the System

In Figure 6, the sequence diagram of simulating a system is given. The user pushes the button to start simulation, which is on the SimulationBoundary object. SimulationBoundary object is the View object associated with a simulation. When the button is pressed, SimulationBoundary notifies SimulationController which then creates a Simulation instance. After that the calculations are made and shown to the user until the user presses the stop button.

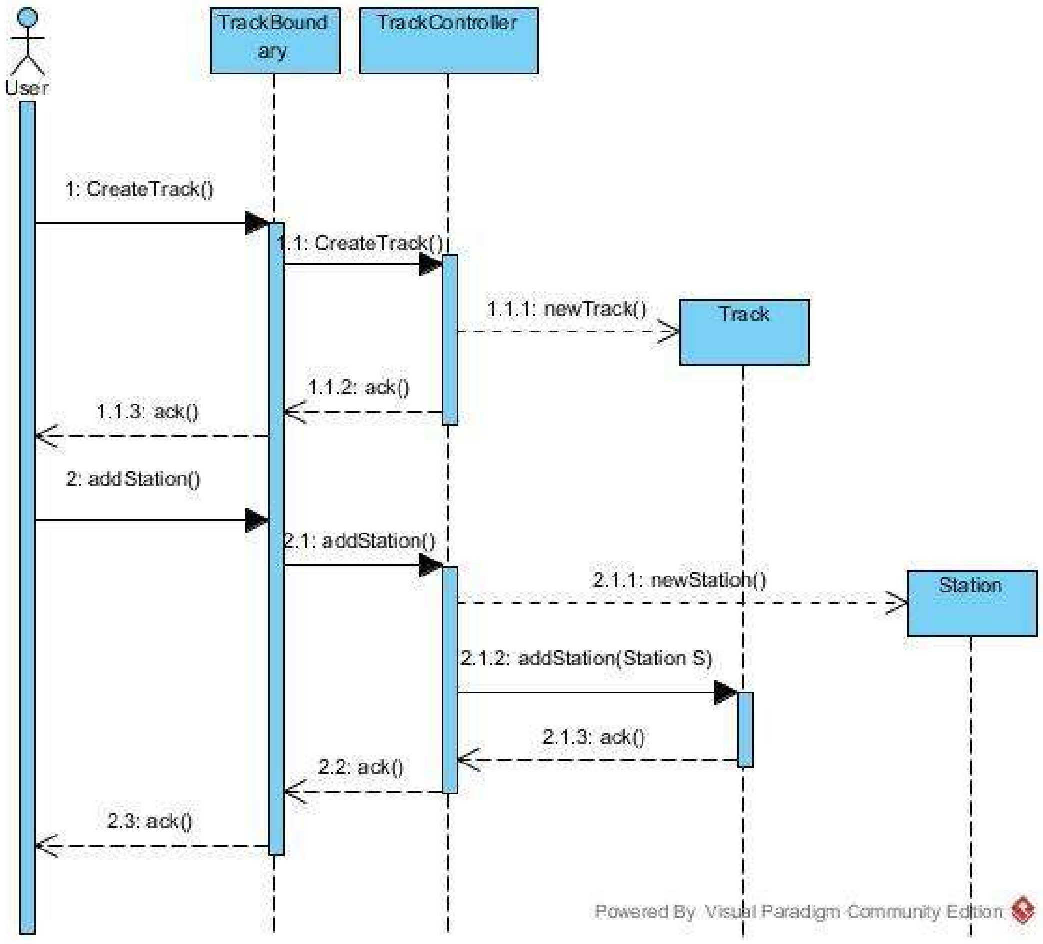


Figure 7 Sequence Diagram for Creating a Track

In Figure 7, the sequence diagram of creating and managing a track is given. The user pushes the add new track button which is on TrackBoundary which is a View object associated with the Track. After this, the TrackController is notified and it will create a new Track instance. After this the user clicks the button for adding a station to a track, then the TrackController object then it creates a new Station instance and links it to the associated Track object. The changes in the Track model will be shown by the TrackBoundary object to the user.

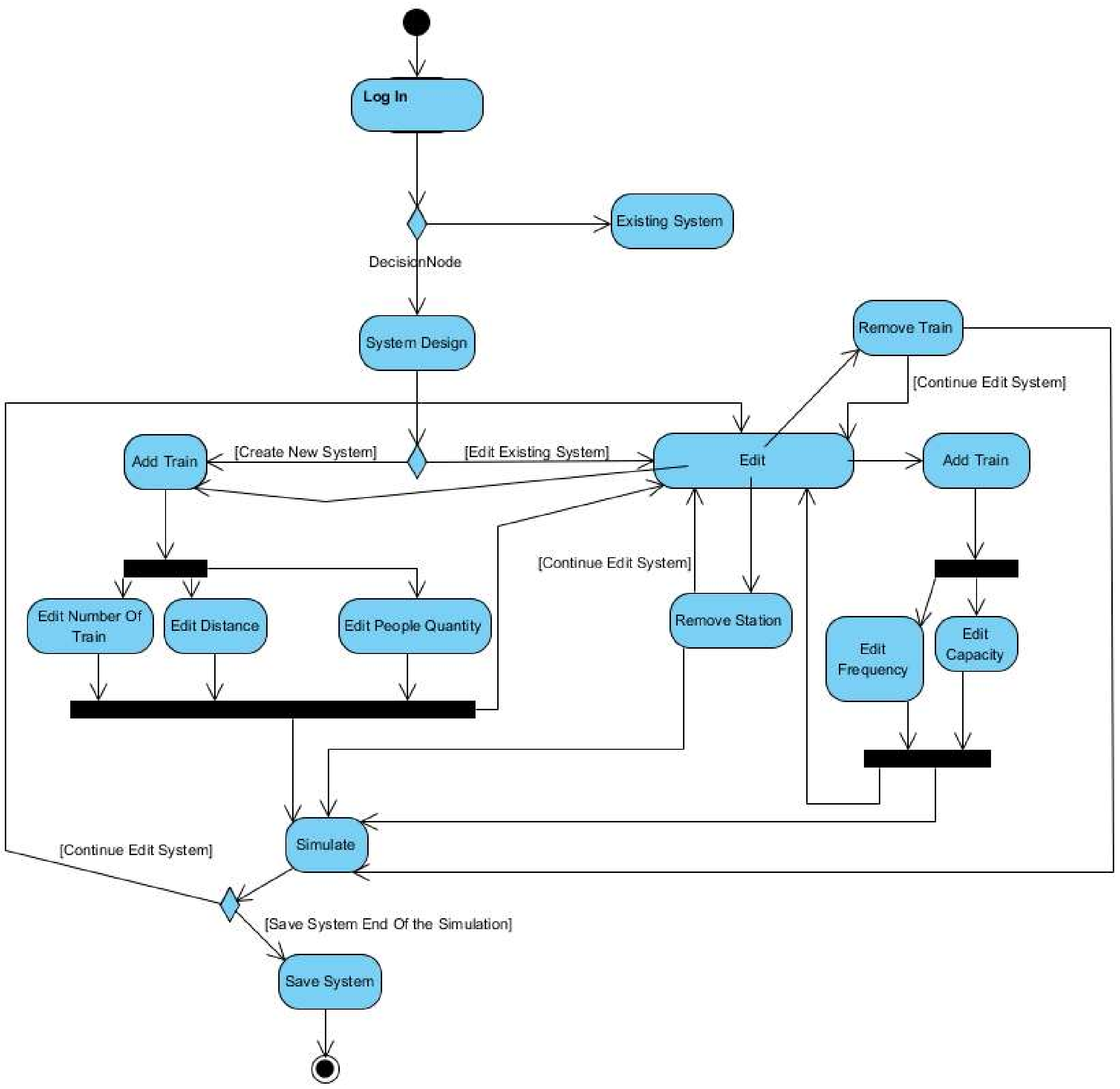
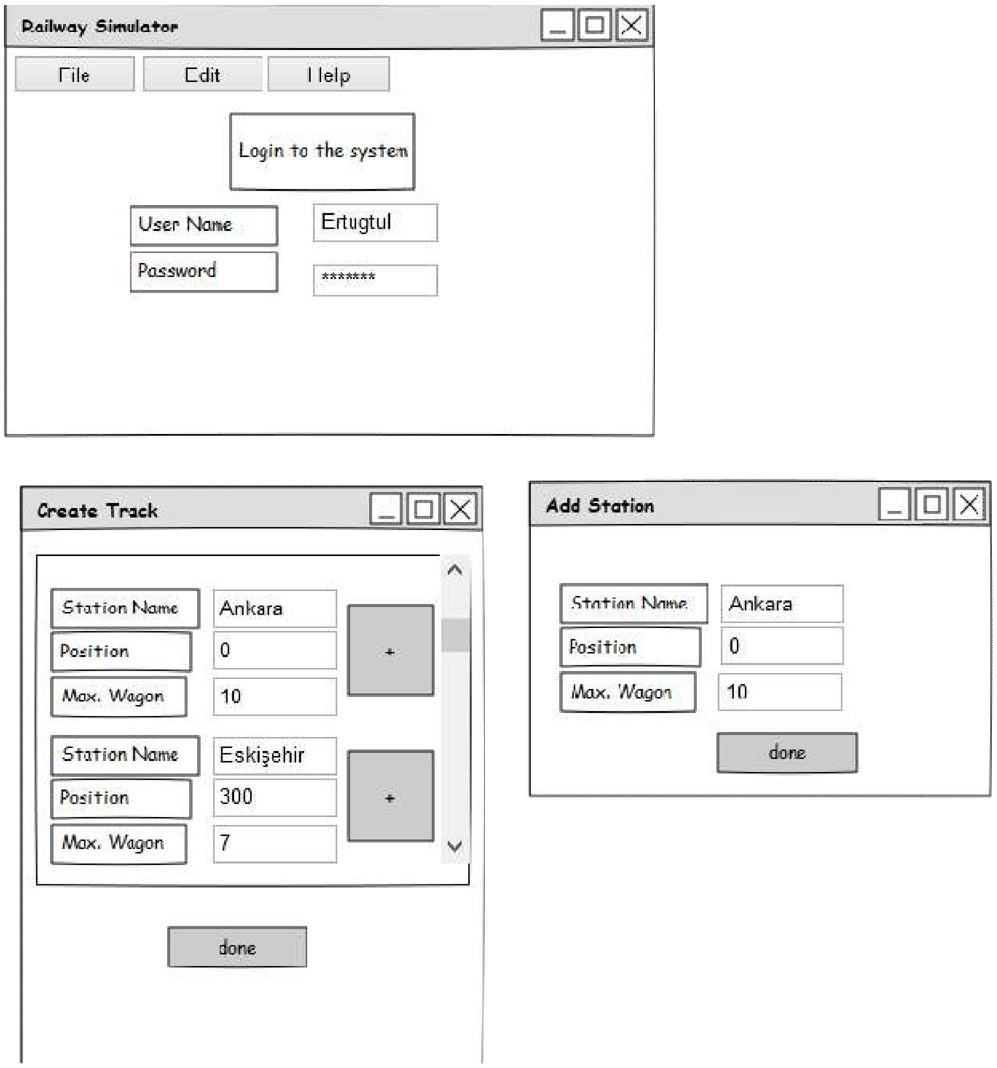


Figure 8 Activity Diagram

User Interface

User interface will be the secondary goal for the project to achieve. However, a simple design is given below in Figure 9 and Figure 10. Since the system requires authentication for a user, there should be a login screen as well as a new user sign up screen. For the functionalities that are required for RSim, there should be a create track dialog and an add station dialog. After the creation of the track and during the simulation time, there should be a general view of the system. In this stage, we consider the graphical interface as a secondary goal and therefore don't have concrete and comprehensive mock ups. During the implementation stage the look and feel of the system will be determining while developing the GUI parts in case we decide to use the GUI.

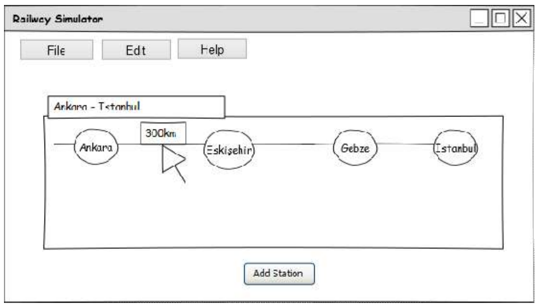


User Name	Ertugtul
Password	*******

Station Name	Ankara
Position	0
Max. Wagon	10
Station Name	Eskişehir
Position	300
Max. Wagon	7

Station Name	Ankara
Position	0
Max. Wagon	10

Figure 9 Mockup for login, add station and create track windows



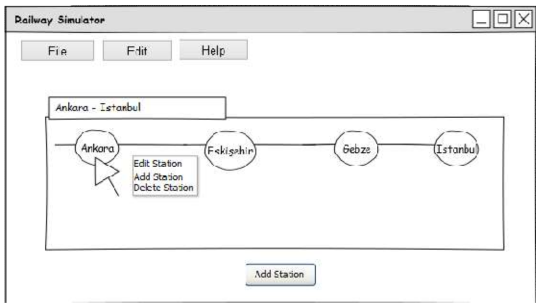


Figure 10 Mockup for General View of the System

Domain Lexicon

The problem domain of the RSim project is the railway systems. In railway system there are multiple tracks that serve as lines of transportation. There are trains that consist of waggons. Trains have their departure hours in a timetable. There might be delays in the trains so a train might be late. There are places that the trains stop and the passengers can enter to the train or leave the train. Those places are called stations. The stations can be included on only a single lines or multiple lines. The tracks and lines can be intersected at junctions. Junctions can be at a station or outside of a station. The lines can have single track as well as multiple tracks to allow two trains to go at the same line at the same time. The list of the words that are encountered on the problem domain are listed below:

- Train
- Railway
- Waggon
- Passenger
- Station
- Timetable
- Ticket
- Track
- Line
- Junction

Design

Design Goals

Describing design goals is the first and one of the most important parts of system design because it labels the qualities that our system should focus on. We determined our design goals as ease of use, good documentation, ease of learning, reliability, tradeoffs, robustness, portability and readability and defining them properly will help us to reduce complexity of the system.

Ease of Use& Ease of Learning :

Ease of usability and ease of learning are important key concept in terms of user's comfort. It is important for user to interact with the system easily without prior knowledge and external help. This design goals can be achieved by providing clear instructions and user documentation and simple user interface. For example in our railway simulation system we will build a simple GUI for adjusting parameters and showing the state of the system.

Good Documentation:

The documentation of the system should be organized well enough to make it easier for new developers to understand the system. Design and analysis reports, documentation of source code should be easily understandable by them to satisfy this design goal.

Reliability:

Reliability is one of the most important design goals. In order to satisfy relialibility requirements, system should have ability to run consistently and system crush shouldn't result in long term data loss, should adhere to boundary conditions, should define faults and tolerate them. Achieving this goal can only be possible by implementing testing process at every stage of development of the system.

Functionality:

In our project our main goal is to create non-complex railway simulation system so that user can interact with the system easily without any external help. Therefore we will try to avoid complicated functions in our system.

Robustness

Robustness is the ability of a system to handle invalid user inputs and unexpected situations. For example in our railway simulation system if user enters negtive number while specifying number of trains system should be able to handle this situation in order to satisfy this goal.

Portability

Portability is an important factor for a software because it provides usability of the same software in different environments. We determined that our system will be implemented in Java. Java Virtual Machine(JVM) plays an important role for portability because nowadays we have JVMs which are written almost for all platforms.

Readability

It is important for developers to understand the system from reading the code. If code is easy to read, it will be easy to understand which makes it easy to debug, maintain and extend.

Sub-System Decomposition

Considering the needs of the project we decided to use MVC architectural pattern. Following the MVC pattern, the subsystems are determined as model subsystem, controller subsystem and interface subsystem where the model subsystem will contain the entity objects of the system and the controller subsystem will modify the entity objects. On each change entity objects will notify the views associated with them and the view objects will update themselves. Interface subsystem will be also responsible for the user interaction where it will get the input from the user and trigger events in controller objects. In figureX the component diagram of the subsystem decomposition is given.

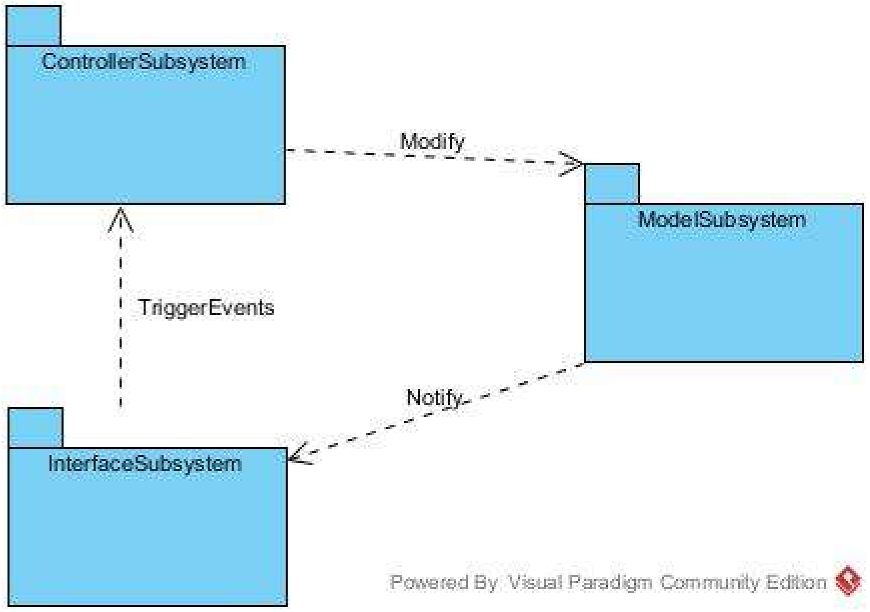


Figure 11 Component Diagram

Architectural Patterns

In order to manage with the complexity, we decided to use MVC architectural pattern. MVC (Model View Controller) architectural pattern is a strong architectural pattern where the entity objects are separated from the view objects and the responsibility for modifying the entity objects are given to the controller objects.

Inside the model subsystem, since we had many classes we decided to use meaningful packages for improving understandability of the subsystem. In Figure 12, redesigned class diagram is given.

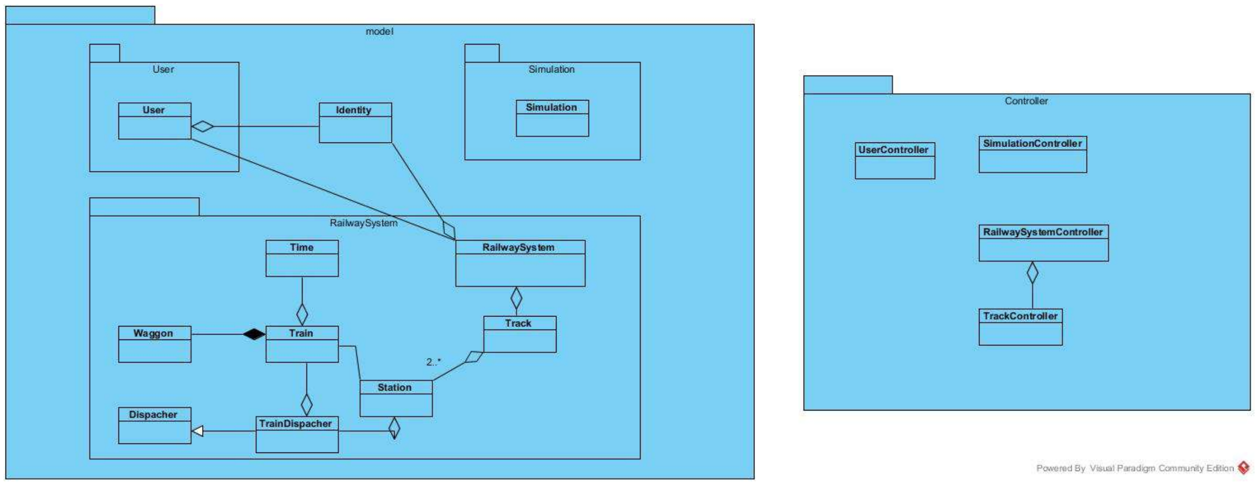


Figure 12 Class Diagram

Hardware/Software Mapping

Railway Simulator is designed to be working on a single node offline, hence there will be only one computer which the software will run on. With a decision, implementation language was chosen to be Java, hence the real software will run on a JVM which will run on top of the host machine. This way, the RSim software will be platform independent. In the Figure 13, deployment diagram is given.

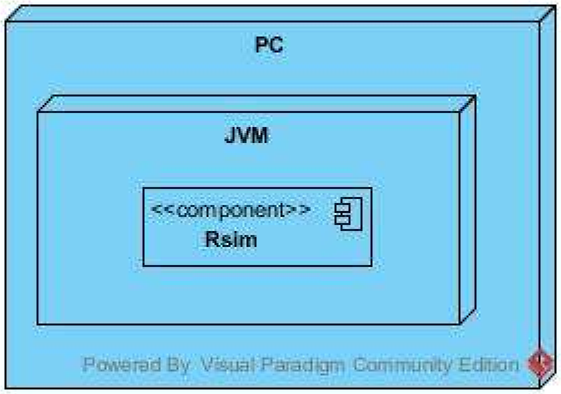


Figure 13 Deployment Diagram

Addressing Key Concerns

Persistent Data Management

In Railway System, the persistent data management will be done by object serialization. The Entity objects will be serializable and by object serialization tools that the programming language will provide; the state of the railway system, state of the simulations and all data that will need to be persistent will be stored on the disk with special file types to RSim. All of the settings and user records will be stored in a special directory that will be created on the user's home folder. On the first run of the program. Since the persistent storage functionality will be separated from other functionalities in the system migrating to another aproach such as using a database system will be easy for future developments.

Access Control and Security

In Railway System Simulator RSim, users will need to log in to the system to be able to simulate or modify the railway system design. Since the system will run on a local computer, there is not much security concerns. Users will be able to modify their own design and simulate it and assign permissions for other users. The permissions on the system will be similar to the operating system permissions. The permissions are listed below:

- Modify Design
- Read Design
- Simulate Design

Different than the operating system file permissions, these permissions won't be disjoint. For example the one who will be able to modify the design can also read the design and simulate the design. The one who is able to simulate the design can read the design but not able to modify the design. The one who is able to read the design will only be able to read the resign. The one who doesn't have the read permission won't be able to perform the other operations.

Global Software Control

In the RSim, event-driven global control flow is used. This is because, event-driven supports that our system’s flow is controlled by events, this global control flow mechanism is one of the most proper control mechanism for our project. In this simulator project, we decided to work with model view controller design patter. By using this pattern, it is provided that simulator waits for an event, which could be mouse click or keyboard inputs, updates the views. As a result, when an event occurs, detection will be made by a part and handling will be made by another part. User Interface subsystem gets the input from the user and delivers it to the Controller subsystem. Controller decides the event's type and changes it in the views related with the event. When event is occurred, different types of object decide action so that there is no main controller. This kind of design is called as decentralized design and it allows distribution of dynamic behavior to objects.

Boundary Conditions

Initialization

- When a user wants to use our simulator s/he will encounter the login page. User needs to enter his own information which are password and user's name. After entering personal page, user can create new system, see another users' systems and edit existing system which is belongs to user. This is because simulator cannot allow user to edit another users' systems.

- Having chosen the creating new system, user will encounter "add station" page. User can decide properties of station and change them as his wish. After adding station system leads user to following functionalities: adding new train, removing station and removing trains. If user wants to add new train, he can edit train's properties such as capacity, frequency. If user wants to remove trains or station, it is required to decision that which train/station will be removed.

- When user chooses editing system, user open saved file and s/he can do followings: adding new station, new train or removing station and train.

- User simulates the system when s/he finishes design of system. User is supposed to enter time interval of simulation. Eventually, user can see its result and s/he can continue to edit system or save system as a file and then finish the program.

Termination

- User can simulate the system anytime and acquire the system's result.
- After simulation, simulator provides user that s/he can save the system as a file and finish simulation.
- After simulation, if user wants to continue working on simuator, simulator provides that and user can continue edititng the system.

Failure

If user wants to modify the railway system that belongs to an another user, simulator does not allows the user. This is because, users can only see other railway systems but they cannot edit another railway systems.

Object Design

Pattern applications

Using design patterns will help us easily manage the frequently encountered problems in our system. In this section we will introduce the design patterns that will be used in the system and the reasons why we will use them.

In the Rsim project, we applied Observer as behavioral, Singleton, Builder and AbstractFactory as creational, Façade and Bridge as structural patterns.

Singleton Pattern

Singleton pattern is used when there is a requirement for a single instance of a specific object in the whole system. We will also use façade design pattern so there should be only one instance of a Façade object and all of the objects in the system will be able to get the reference of that instance by using getInstance() method. The sample code for Singleton Pattern is as follows:

```
public class ModelFacade{
       private static ModelFacade uniqueInstance = null;

       private ModelFacade(){
              //constructor
       }

       public static ModelFacade getInstance()
       {
              If (uniqueInstance == null)
                     uniqueInstance = new ModelFacade();
              return uniqueInstance;
       }

}
```

This pattern is used in many classes in our system such as the facade objets, RSim object, Simulation object etc.

Façade Pattern

Façade pattern is used to provide a clear and clean interface of the subsystems in the system. The advantage of the Façade pattern is that the internal changes in the subsystems wouldn't change the subsystem interface if the façade is not changed. In our system the facade pattern is used in most of the subsystems which are complex.

This pattern will make our implementation time longer but once the system is deployed the maintenance effort of the system will be significantly less.

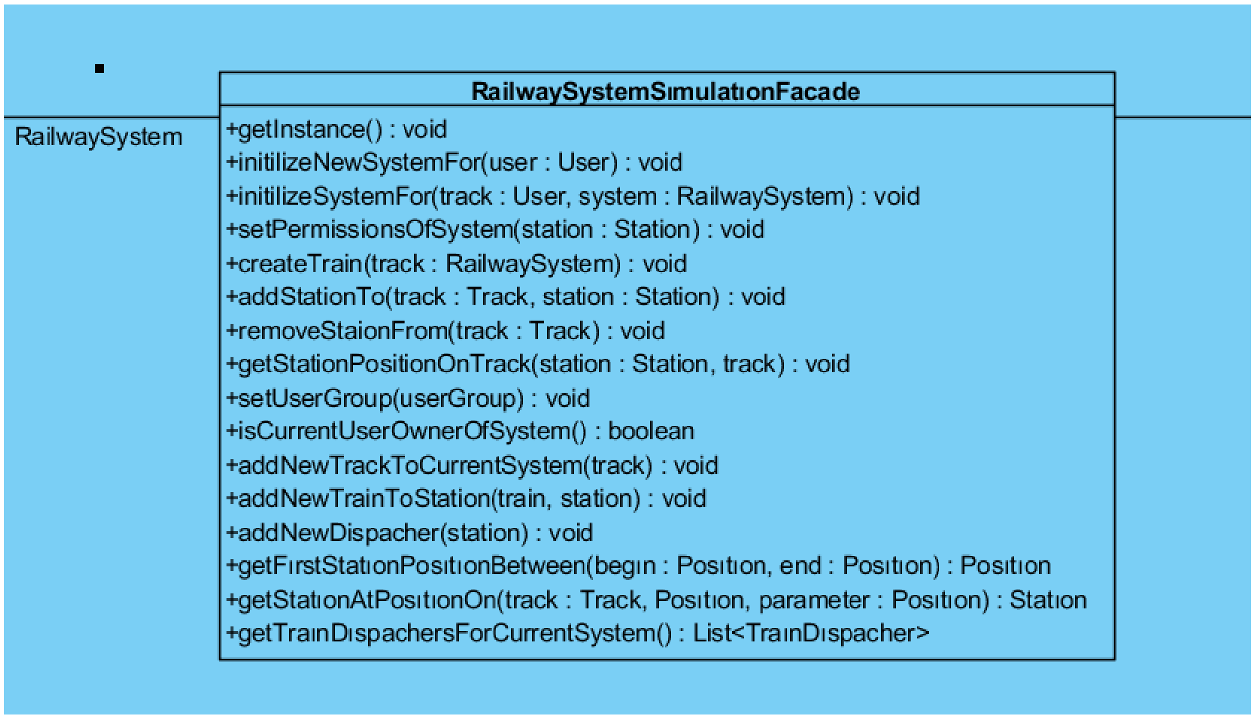

RailwaySystem
RailwaySystemSimulationFacade
+getInstance() : void
+initilizeNewSystemFor(user : User) : void
+initilizeSystemFor(track : User, system : RailwaySystem) : void
+setPermissionsOfSystem(station : Station) : void
+createTrain(track : RailwaySystem) : void
+addStationTo(track : Track, station : Station) : void
+removeStaionFrom(track : Track) : void
+getStationPositionOnTrack(station : Station, track) : void
+setUserGroup(userGroup) : void
+isCurrentUserOwnerOfSystem() : boolean
+addNewTrackToCurrentSystem(track) : void
+addNewTrainToStation(train, station) : void
+addNewDispacher(station) : void
+getFirstStationPositionBetween(begin : Position, end : Position) : Position
+getStationAtPositionOn(track : Track, Position, parameter : Position) : Station
+getTrainDispachersForCurrentSystem() : List<TrainDispacher>

Figure 14 Railway System Façade

Bridge Pattern

Bridge pattern is used to decouple the abstraction of an object with its implementation. This pattern provides flexibility and maintainability to the system, because the abstractions and the implementations can be refined and changed independently without having to change the used ones.

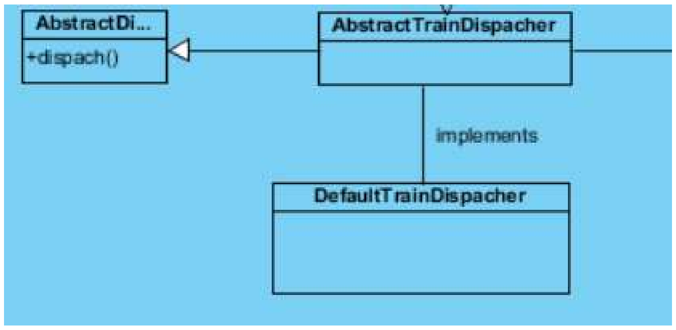


Figure 15 Bridge Pattern Applied on Dispacher

Abstract Factory Pattern

Abtract factory pattern is used to create and produce compatible objects. The identity class is a complex class to be constructed on its own. New identitiy instances has to have incremental ids, and it has to be added to the identity database upon creation. It also needs to represent the type of the object which it is the identity of. Therefore the IdentityFactory abstract class is provided and the implementations of the factories are provided to have integrity.

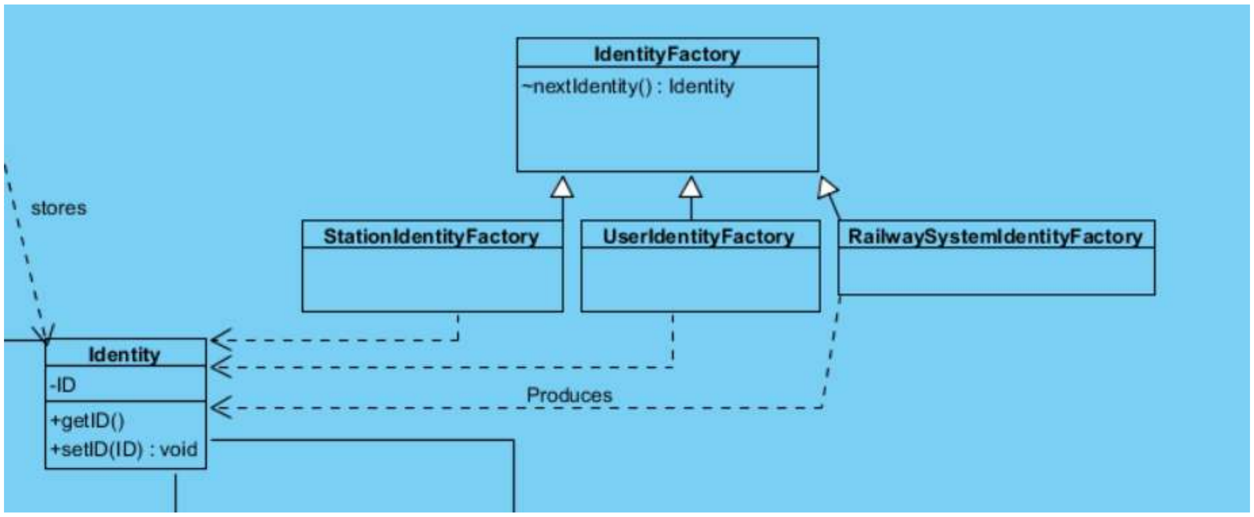
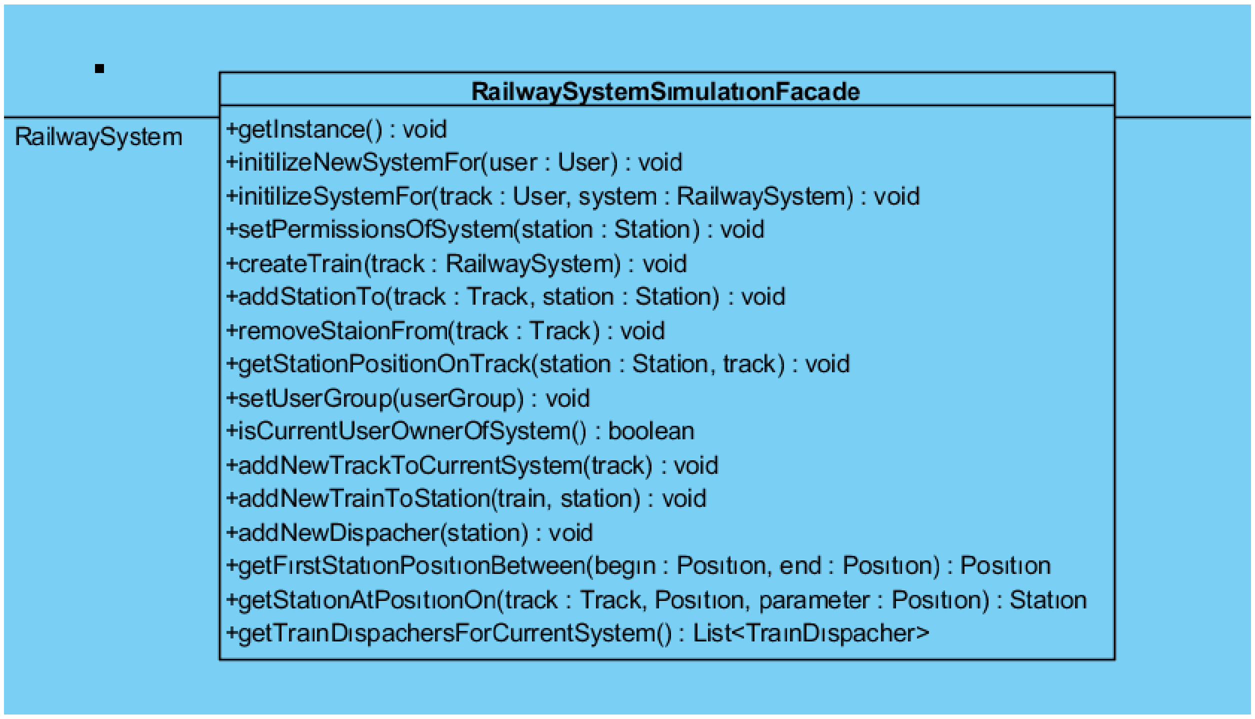


Figure 16 Abstract Factory Pattern Applied to Identity Product

Builder Pattern

Builder Pattern is used when there are lots of attributes that needs to be or can be set during the object construction. This yields lots of constructors with lots of parameters. Those kinds of constructors are called telescopic constructors and they are error prone. We used builder pattern on creation of Train objects. Train object has a lot of attributes and most of them needs to be set during the creation. To make the creation simple and error free we will implement a TrainBuilder.



```
+getInstance() : void
+initilizeNewSystemFor(user : User) : void
+initilizeSystemFor(track : User, system : RailwaySystem) : void
+setPermissionsOfSystem(station : Station) : void
+createTrain(track : RailwaySystem) : void
+addStationTo(track : Track, station : Station) : void
+removeStaionFrom(track : Track) : void
+getStationPositionOnTrack(station : Station, track) : void
+setUserGroup(userGroup) : void
+isCurrentUserOwnerOfSystem() : boolean
+addNewTrackToCurrentSystem(track) : void
+addNewTrainToStation(train, station) : void
+addNewDispacher(station) : void
+getFirstStationPositionBetween(begin : Position, end : Position) : Position
+getStationAtPositionOn(track : Track, Position, parameter : Position) : Station
+getTrainDispachersForCurrentSystem() : List<TrainDispacher>
```

Figure 17 Builder Pattern Applied on Train

Observer Pattern

Observer pattern is used when there is frequently changing objects. Observer pattern is used to introduce consistency among the objects. There are observer objects which reflects the state of the object which is being observed. There are more than one way to implement the observer pattern. In our case subject will notify the observers.
In our implementation of the observer, the view objects will be the observer objects where the model objects will be the subjects. Whenever there is a change in the model objects the views will be notified and the views will be able to reflect the changes.

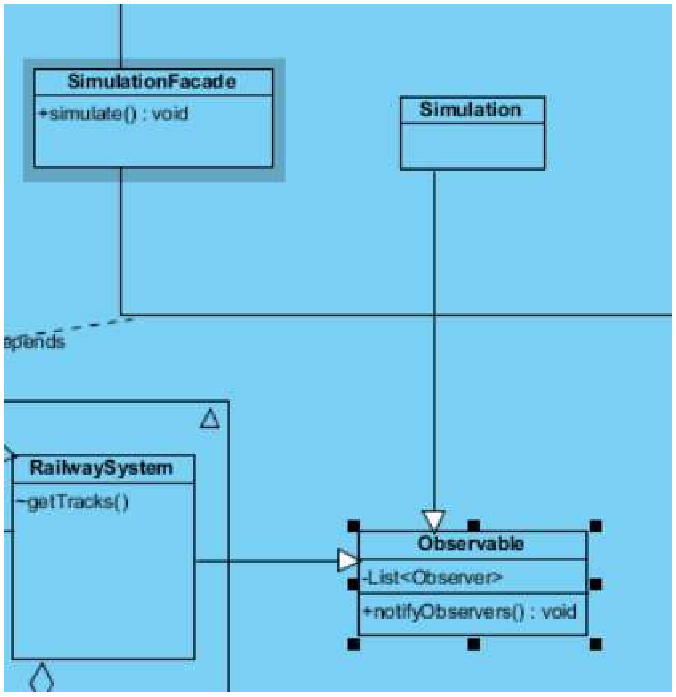


Figure 18 Observable Interface for Observable Objects

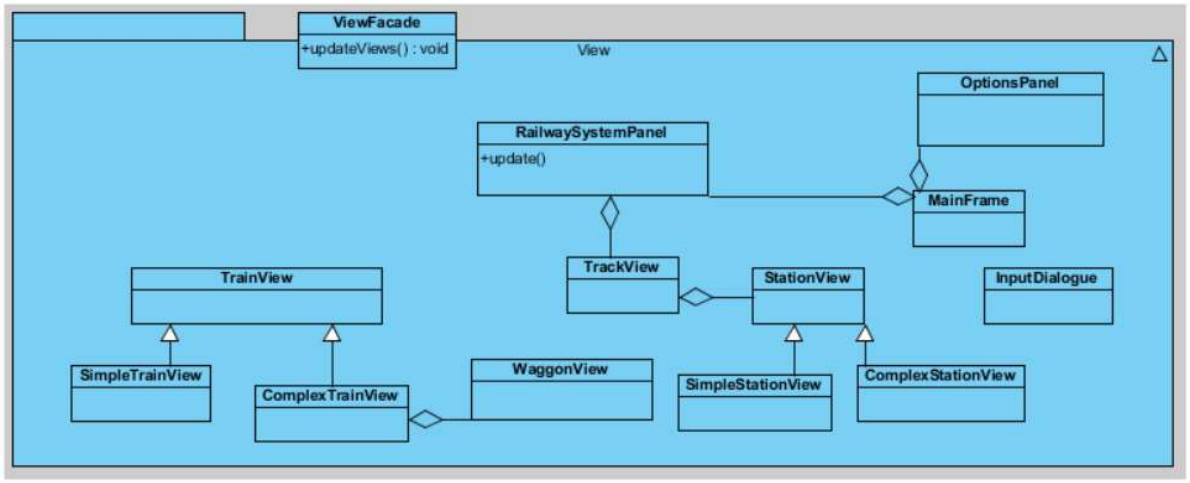


Figure 19 Observer Pattern on View Objects

Class Diagrams After Applying Patterns

After determining the design patterns that will be used, the object design is made. While making the object design the objects are put into the packages that are taxonomically related to them. The Dynamic Simulation objects are put into the Simulation Package whereas the objects which represents the static state of the railway system is put into the RailwaySystem package. The separate snapshots of the packages from the object design are given below.

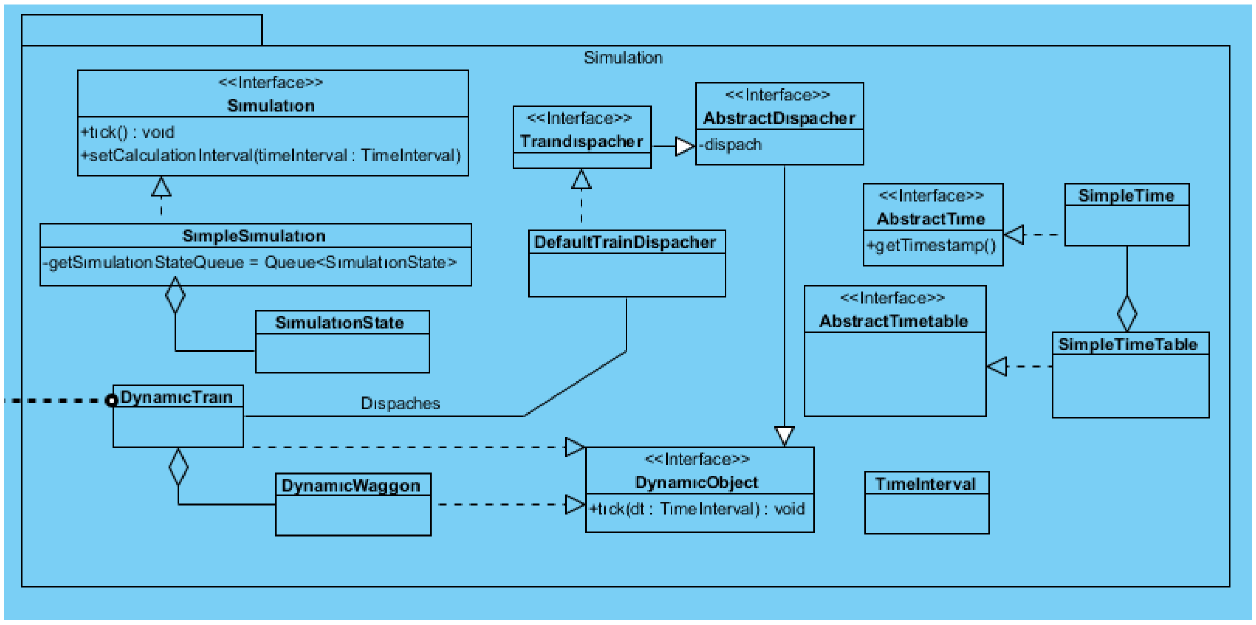


Figure 20 Simulation Package

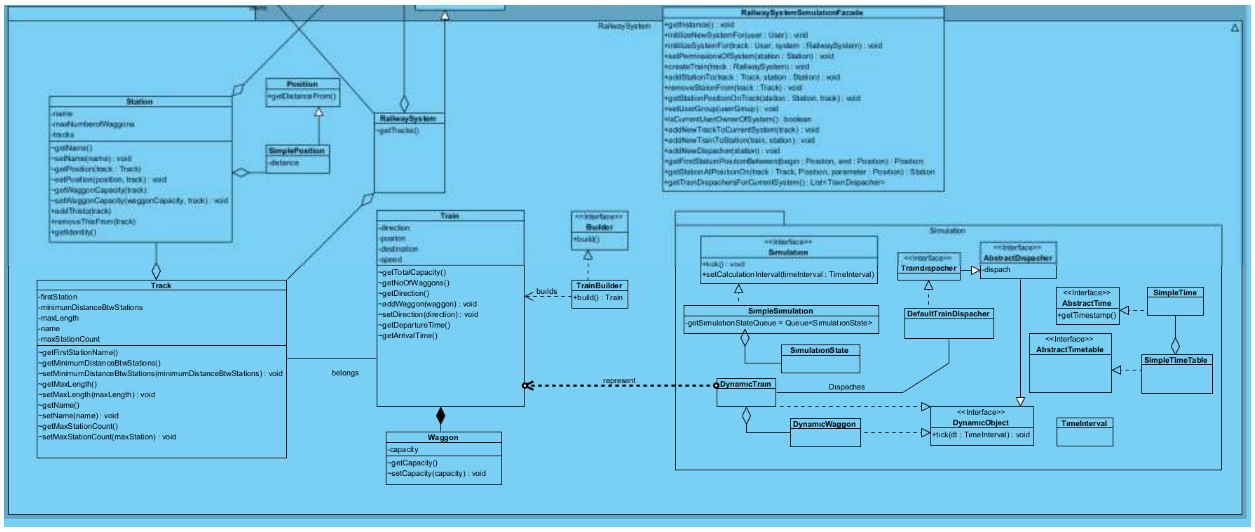


Figure 21 RailwaySystem Package

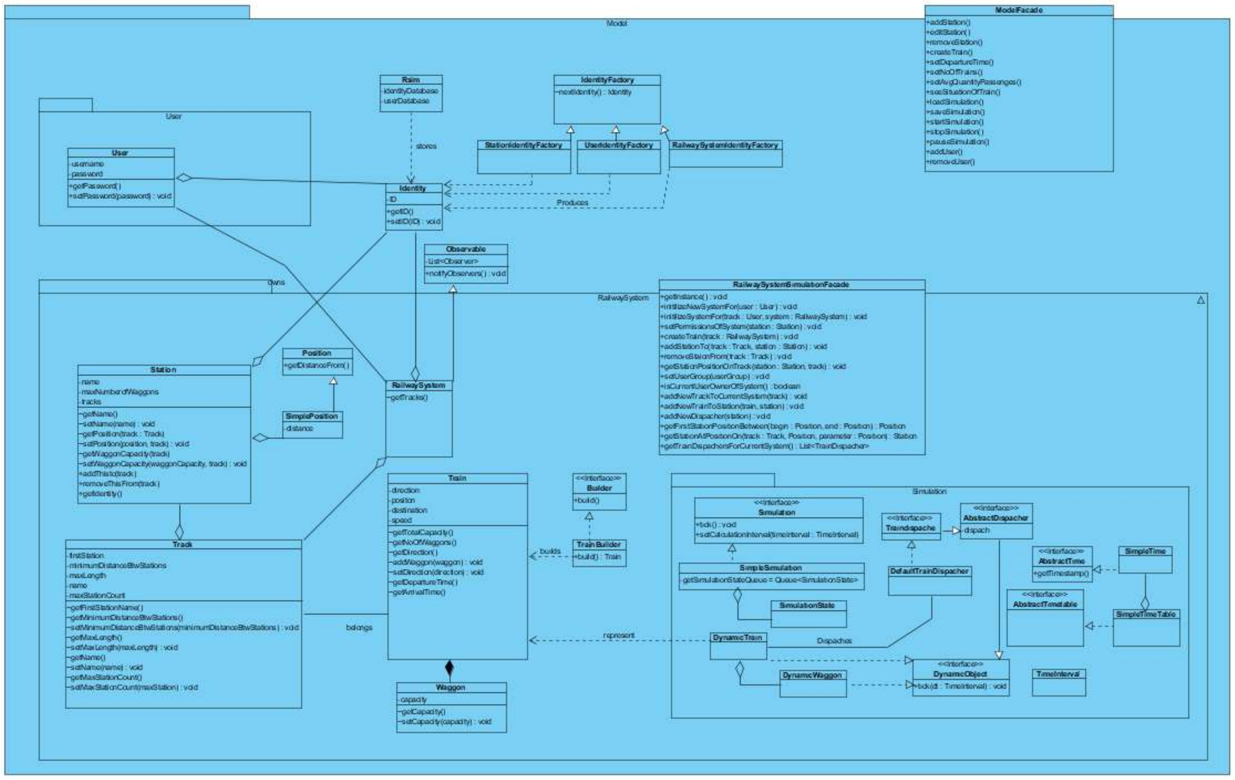


Figure 22 Model Package

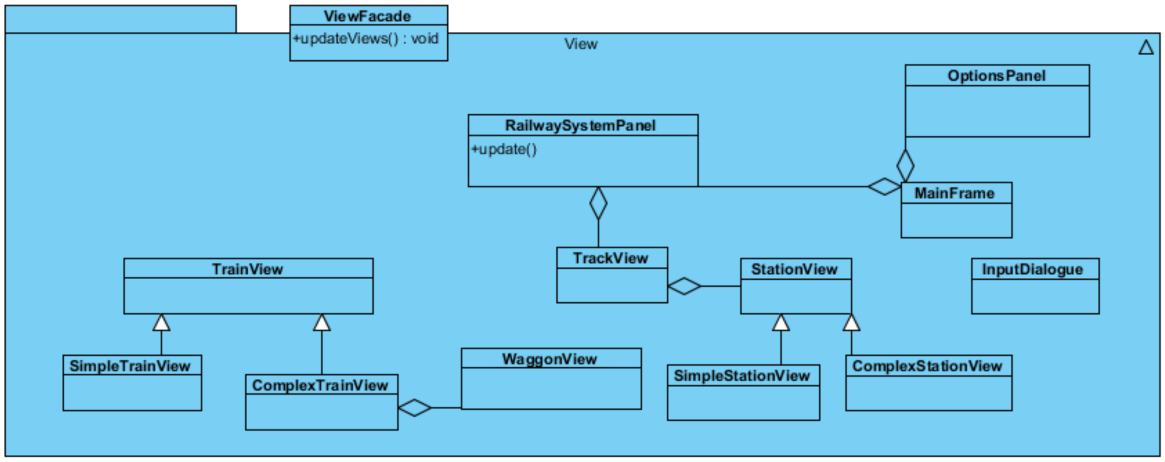


Figure 23 View Package

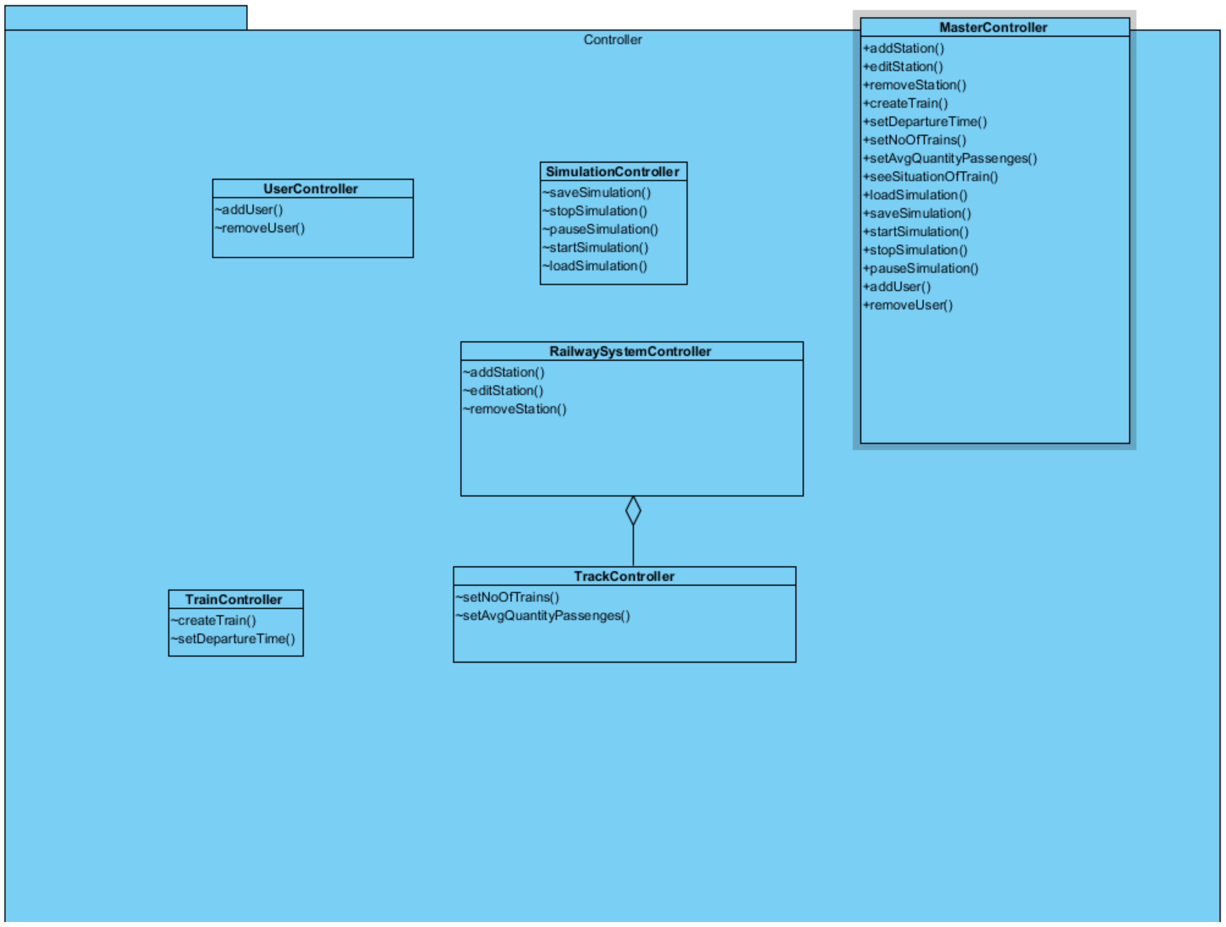


Figure 24 Controller Package

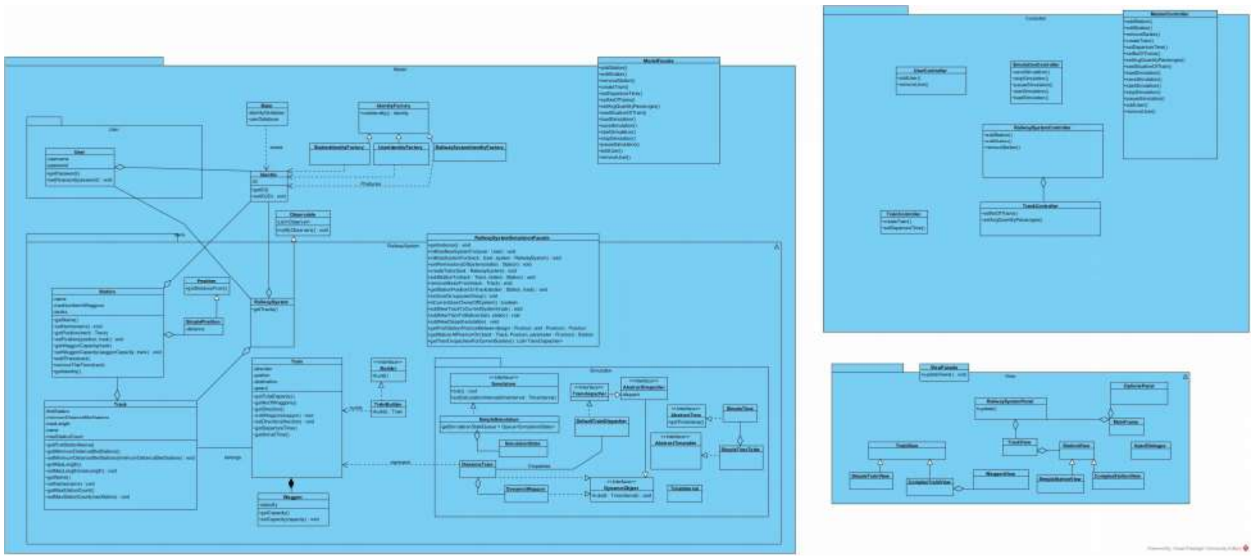


Figure 25 RSim Class Diagram

Class Interfaces

Description of the classes and their public interfaces, on the other hand their visibility, type signatures, attributes and methods are explain deeply in this section. In this part, they will be presented as following; Model Classes, View Classes and Controller classes since we have applied MVC as structural pattern.

Model Classes

ModelFacade.java:

ModelFaçade is the class that makes all the changes according to the modifications on model. It makes interface of subsystem bare. ModelFaçade class provides several public methods such as addStation(), editStation(), removeStation(), createTrain(), setDepartureTime(), setNoOfTrains(), setAvgQuantityPassenger(), seeSituationOfTrain(), loadSimulation(), saveSimulation(), startSimulation(), stopSimulation(), pauseSimulation(), addUser(), removeUser().

User.java:

User is a class that is purposed s/he will use the program. Login is a major step for user in order to access program and his profile. Therefore, this class keeps user's informations. Username and password are saved as a properties of the class. This class also includes both getPasswod() and setPassword(password) methods.

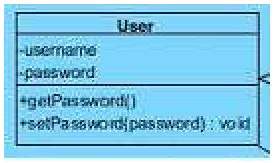


Figure 26 User Class

RailwaySystemSimulationFacade.java:

This is another facade which is included in the model facade. RailwaySystemSimulationFacade includes several methods, another package which is called as simulation and classes of railway system's objects such as train, wagon, station, track. These methods are getInstance(), initilizeNewSstemFor(user:user), initilizeSystemFor(track: User , system: RailwaySystem), setPermissionsOfSystem(station:Station), createTrain(track: RailwaySystem), addStationTo(track: Track, station: Station), removeStationFrom(track:Track), getStationPositionOnTrack(station, track), setUserGroup(userGroup), isCurrentUserOwnerOfSystem(), addNewTrackToCurrentSystem(track), addNewTrainToStation(train, track), addNewDizpacher(station), getFirstStationPositionBetween(begin, end), getStationAtPositionOn(track, position, parameter), getTrainDispacherForCurrentSystem().

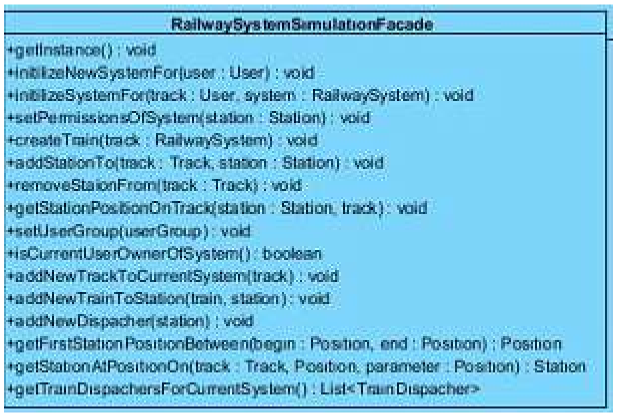

RailwaySystemSimulationFacade
+getInstance() : void
+initilizeNewSystemFor(user : User) : void
+initilizeSystemFor(track : User, system : RailwaySystem) : void
+setPermissionsOfSystem(station : Station) : void
+createTrain(track : RailwaySystem) : void
+addStationTo(track : Track, station : Station) : void
+removeStaionFrom(track : Track) : void
+getStationPositionOnTrack(station : Station, track) : void
+setUserGroup(userGroup) : void
+isCurrentUserOwnerOfSystem() : boolean
+addNewTrackToCurrentSystem(track) : void
+addNewTrainToStation(train, station) : void
+addNewDispacher(station) : void
+getFirstStationPositionBetween(begin : Position, end : Position) : Position
+getStationAtPositionOn(track : Track, Position, parameter : Position) : Station
+getTrainDispachersForCurrentSystem() : List<TrainDispacher>

Figure 27 RailwaySystemSimulationFacade Class

Station.java:

This class is a one of the major class of the system. Station is a main component of railway system that user must be add the system. This class has name, and it keeps tracks and maximum number of wagons as a properties. This class has methods which are; getName(), setName(), setPosition(), getWaggonCapacity(), setWaggonCapacity(), addThisto(), removeThisFrom(), getİdentity().

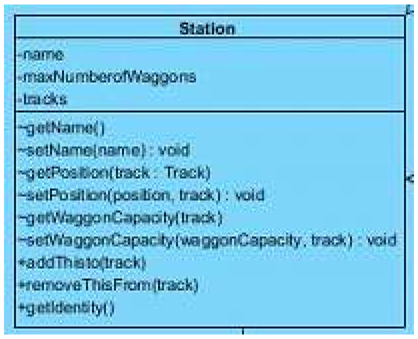

Station
-name
-maxNumberofWaggons
-tracks
~getName()
~setName(name) : void
~getPosition(track : Track)
~setPosition(position, track) : void
~getWaggonCapacity(track)
~setWaggonCapacity(waggonCapacity, track) : void
+addThisto(track)
+removeThisFrom(track)
+getIdentity()

Figure 28 Station Class

Track.java:

This class is belong to railway system. It is a requirement for existence of station since station can only exist when it is on a track. This class keeps first station, minimum distance between station, track's name, maximum station count and maximum length as a properties. Moreover it uses some methods which are getFirstStationName(), getMinimumDistanceBtwStations(), setMinimumDistanceBtwStations() getMaxLength(), setMaxLength(), getName(), setName() ,getMaxStationCount(), setMaxStationCount().

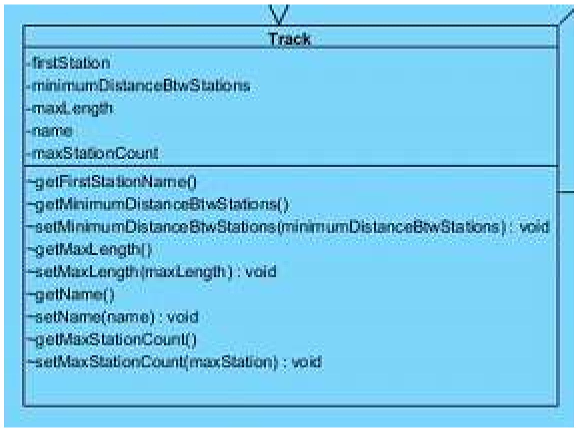

Track
-firstStation
-minimumDistanceBtwStations
-maxLength
-name
-maxStationCount
~getFirstStationName()
~getMinimumDistanceBtwStations()
~setMinimumDistanceBtwStations(minimumDistanceBtwStations) : void
~getMaxLength()
~setMaxLength(maxLength) : void
~getName()
~setName(name) : void
~getMaxStationCount()
~setMaxStationCount(maxStation) : void

Figure 29 Track Class

Position.java:

This class has a only one method which is getDistanceFrom(). This class is used parameter type for another methods. Also, it generalizes a class which is called as SimplePosition which keeps a one parameter that is a distance.

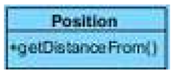


Figure 30 Position Class

RailwaySystem.java:

railway system involves multiple tracks that serve as lines of transportation. This class has only one method which is getTrack() that helps to get information about these multiple tracks.

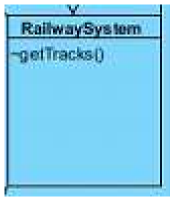


Figure 31 RailwaySystem Class

Train.java:

Train is another major object in the system. It has some requirement and obligation to must satisfy. For example, it cannot have zero number of wagons. This class keeps some data such as direction, position, destination, speed as a parameter. Also, in this class, getTotalCapacity(), getNoOfWaggons(), getDirection(), setDirection(), addWaggon(), getDepartureTime(), getArrivalTime() are used as a methods.

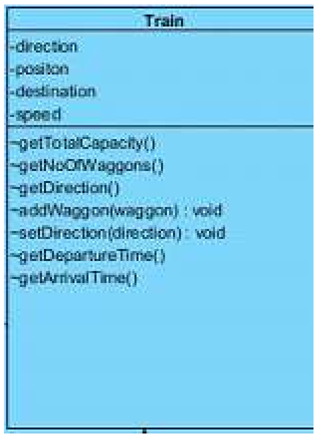


Figure 32 Train Class

Builder.java:
This is a interface of train. It builds a train by build() method.

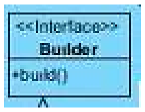


Figure 33 Builder Class

Waggon.java:
This class is a main component of train. Trains cannot exists without wagons and trains have limited wagons. Wagon class has captain so that system has some limits due to the these capacity. This class has a one parameter which is capacity and it uses both getCapacity() and setCapacity() methods.

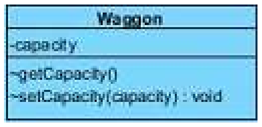


Figure 34 Waggon Class

Rsim.java:
This class implements Serializable and. It includes identityDatabase and userDatabase. This is main entity object. It is singleton design patter. This class is constructed by views and controllers hence this object construct other objects which are places in model.

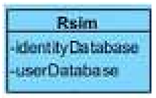


Figure 35 RSim Class

Identity.java:
It keeps id and uses two methods which are getID(), setID(). This class is capable for identification of the users. Moreover, It produces detailed identifications such ad UserIdentityFactory, RailwaySystemIdentityFactory, IdentityFactory, StationIdentityFactory.

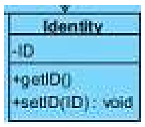


Figure 36 Identity Class

Obserable.java:

This class has a one property which is list that keeps observers and it uses notfyObserverts() method. Observer objects are capable to reflects the changing whenever changing is occurred in the model objects.

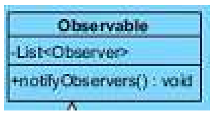


Figure 37 Observable Class

Simulation.java:

This is another interfaces which is in the simulation package. It has tick() method and setCalculationInterval() method.

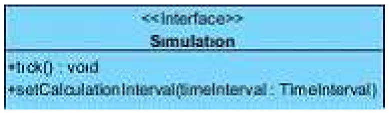


Figure 38 Simulation Class

SimpleSimulation.java:

It has queue which determines the state of the simulations. It has a property which is called getSimulationStateQueue. This is a way to simulate the railway system. This class keeps track of the dynamic objects of the simulation. It calculates the next state using the time interval specified and keeps a record of simulation state in the queue.

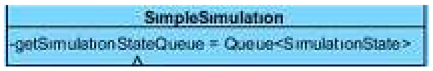


Figure 39 SimpleSimulation Class

AbstractDispacher.java:

It keeps dispahers as a parameter. This is an interface class and it is places in simulation package.

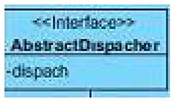


Figure 40 AbstractDispacher Class

AbstractTime.java:
It has getTimestamp() method and it is an interface class.

<<Interface>>
AbstractTime
+getTimestamp()

Figure 41 AbstractTime Class

DynamicObject.java:
This class is an interface class and it has tick() method.

Controller Classes

Controller Class
Controller Class is the class that provides the other classes with the ability to communicate with Controller classes. Controller class involve controllers which belong to other classes. This class includes following controllers: MasterController, UserController, SimulationController, RailwaySystemController, TrackController, TrainController.

MasterController.java:
This controller uses these methods which are addStation(), editStation(), removeStation(), createTrain(), setDepartureTime(), setNoOfTrains(), setAvgQuantityPassenger(), seeSituationOfTrain(), loadSimulation(), saveSimulation(), startSimulation(), stopSimulation(), pauseSimulation(), addUser(), removeUser().

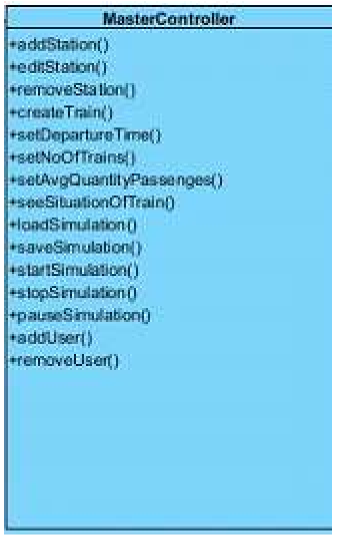


Figure 42 MasterController Class

UserController.java:

This controller uses these methods which are addUser(), removeUser().

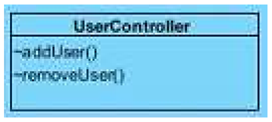


Figure 43 UserController Class

SimulationController.java:

SimulationController creates a Simulation instance when simulation is started.This controller uses these methods which are loadSimulation(), saveSimulation(), startSimulation(), stopSimulation(), pauseSimulation().

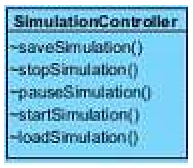


Figure 44 SimulationController Class

RailwaySystemController.java:

This controller uses these methods which are   addStation(), editStation(), removeStation().

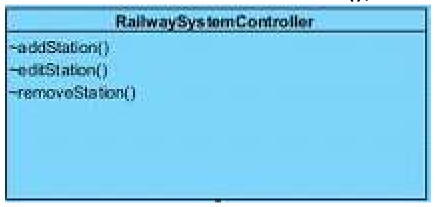


Figure 45 RailwaySystemController Class

TrackController.java:

This controller is notified when user wants to create track and click the button and it will create a new Track instance. This class uses these methods which are setNoOfTrains(), setAvgQuantityPassenger().

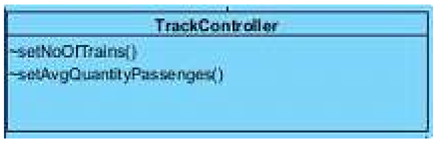


Figure 46 TrackController Class

TrainController .java:

This controller uses these methods which are createTrain(), setDepartureTime().

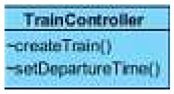


Figure 47 TrainController Class

View Classes

ViewFacade.java:

ViewFacade is the class that provides the other classes with the ability to communicate with View classes**.** In this class, following are included: RailwaySystemPanel which has update method, TrainView, SimpleTrainView, ComplexTrainview, WaggonView, TrackView, StatioView, SimpleStationView, ComplexStationView, MainFrame, InpuDialoue, OptionsPanel. Also, this facade has updateViews() method.

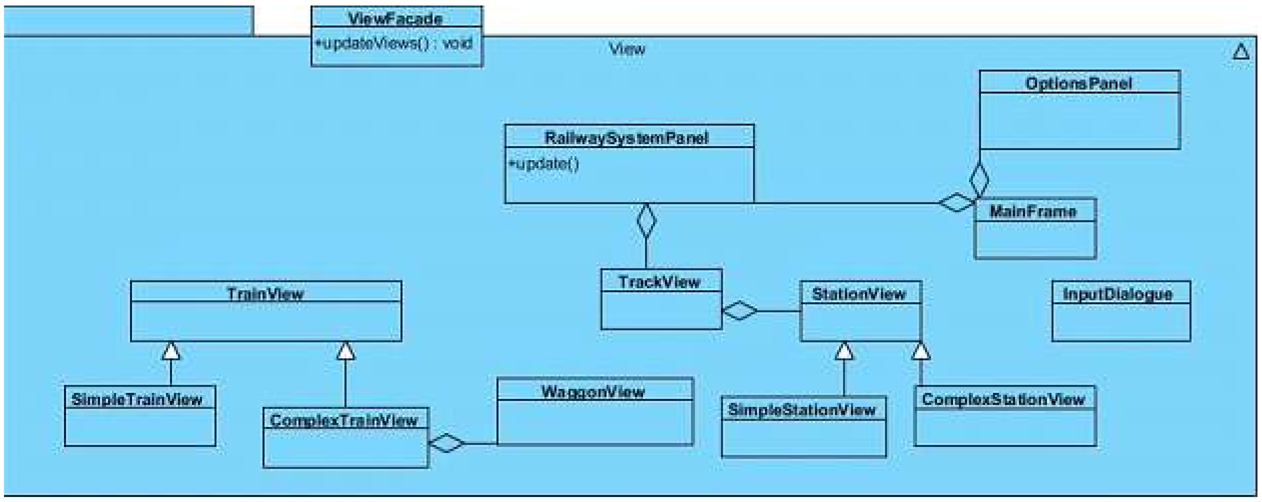


Figure 48 View Package

Specifying Contracts

Contracts are used to state the dynamic behavior of the objects. It is used for declariung the constraints on an object or a method. Using the contracts, we can know when we can call a method and what will happen after the method is invoked. We can also know that a spesific field of a class will always satisfy a condition.

Here are some of the contracts in RailwaySimulator system is defined in OCL:

1. **context Station inv: self->tracks.size() > 1**

   The stations can't exist without being on a track.

2. **context Train inv: self>wagoons != null**

   The waggons list must not be null in any of the trains.

3. **context Train inv: maxNumberOfWaggons > 1**

   The capacity of a wagon must be an integer which is greater than 0.

4. **context Wagon inv: capacity > 0**

   The capacity of a wagon must be an integer which is greater than 0.

5. **context Wagon::setCapacity(capacity:int) post: self->capacity = capacity**

   This contracts mean that inputted integer constant will be set to the capacity of the wagon.

6. **Context Station::getName() post: result = name**

   This contract represents the retrieval of the name of the Station object. Eventually, result will be equal to name of the Station object.

7. **context Station::setName(name:String) post: self->name = name**

   This contracts mean that inputted String will be set to the name of the Station object.

8. **context Station::getPosition(track:Track)**
   pre: track != null && track.stationList.contains(self)

   This contract means that to call getPosition method from a Station, the Station muts be on the track that is specified.

9. **context Station::setPosition(position:Position, track:Track) post: self->position = Position**

   This contract represents the setting the position of the Station on the given track.

10. **context Station::addThisTo(track)**
    pre: not track.stationList.contains(self) and
    not track.stationList.contains(self)
    post: self->tracks.contains(track) and
    track.stationList.contains(self)

    This contract means looking at first the condition that whether the station which is wanted to be added to the given track is contained in the list of stations , then eventually if it does not exist , it is added, and both entries are set in the lists.

11. **context Station::removeThisFrom(track)**
    pre: **self->tracks.contains(track) and**
        track.stationList.contains(self)
    post: **not track.stationList.contains(self) and**
        not track.stationList.contains(self)

This contract means looking at first the condition that whether the station which is wanted to be removed from the given track is contained in the list of stations , then eventually if it exists , it is removed from the track, and both entries are set in the lists.

12. **contextUser::setPassword(password:long)**
    pre: isValid(password)
    post: self.password = password

This contract means that given that the password is valid, after calling the setter method for password the user's password will be changed.

13. **context RailwaySystemFacade::getInstance()**
    post: not null, unique RailsaySystem.instance

This contract is a representation of singleton pattern. The static method get instance can ge called from anywhere and the singleton pattern ensures that there is only one unique instance of RailwaySystemFacade object.

14. **context RailwaySystemFacade::initializeNewSystemFor(user:User)**
    pre: user.getID() != null
    post: self->currentUser = user

This contract ensures that the setting of the currentUser on the RailwaySystemFacade instance is properly done.

15. **context RailwaySystemFacade::initilizeSystemFor(user:User, system:RailwaySystem)**
    post: self->currentUser = user and self->currentSystem = system

This contract means that the initilization of the RailwaySystemFacade will have an effect on the object as specified.

16. **context RailwaySystemFacade::addStationTo(track:Track, station:Station)**
    pre: isOnCurrentSystem(track)
    post: station.addThisTo(track, position)

    This contract means looking at first the condition that whether the station which is wanted to be added to the given track is contained in the list of stations , then eventually if it does not exist , it is added, and both entries are set in the lists.

17. **context RailwaySystemFacade:: removeStationFrom(track:Track, station:Station)**
    pre: isOnCurrentSystem(track)
    post: station.removeThisFrom(track)

    This contract means looking at first the condition that whether the station which is wanted to be removed from the given track is contained in the list of stations , then eventually if it exists , it is removed from the track, and both entries are set in the lists.

18. **contextRailwaySystemFacade::getStationPositionOnTrack(track:Track, station:Station)**
    pre: isOnCurrentSystem(track)

    This contract means that to get the position of the station on a given track, the station must exist in that track.

19. **context RailwaySystemFacade::getStationsMaxNumWaggonsOnTrack(track:Track, station:Station)**
    pre: isOnCurrentSystem(track)

    This contract means that to get the max number of wagons of the station on a given track, the station must exist in that track.

20. **context RailwaySystemFacade::setUserGroup(group:UserGroup)**
    post: currentSystem.setGroup(group)

    This contract means that after the call of the setUserGroup on RailwaysystemFacade object the results of currentSystem.setGroup(group) call will apply.

21. **context RailwaySystemFacade::addNewTrackToCurrentSystem()**
    post: self->currentSystem.getTracks().size() == @pre-> currentSystem.getTracks().size() +1

    This contract means that after the call, the number of tracks in the current system will increase by 1.

22. **context RailwaySystemFacade::addNewTrackToCurrentSystem(Station :station)**
pre: station != null
post: self->currentSystem.getTracks().size() == @pre-> currentSystem.getTracks().size() +1 and self->currentSystem.getLastTrack().getFirstStation().equals(station)

This contract means that after the call, the number of tracks in the current system will increase by 1.

23. **context RailwaySystemFacade::isOnCurrentSystem(track:Track)**
pre : self->currentSystem != null

This contract means that the current system must be set before making this query.

24. **context RailwaySystemFacade::isOnCurrentSystem(station:Station)**
pre : self->currentSystem != null

This contract means that the current system must be set before making this query.

25. **context RailwaySystemFacade::addNewTrainTo(station:Station)**
pre: isOnCurrentSystem(station:Station)

This contract means that the current system must include the station that we will add a train to.

26. **context RailwaySystemFacade::addTrainTo(station:Station, train:Train)**
pre: isOnCurrentSystem(station:Station)

This contract means that the current system must include the station that we will add a train to.

Conclusions and Lessons Learned

The railway simulator software called RSim is a tool that allows user to design railway system and simulate it to observe how it works. In this system user can create new railway route, edit existing routes, add new stations, trains and edit their properties. At the end of the simulation process user can observe system in action. Our aim while creating RSim is to show how a railway system works and enable users to build efficient railway systems or create more efficient timetables or plans by simulating and observing the effects of the changes. User can create different variation of routes and observe what will happen at the desired time. Hence, once simulated with satisfying results, the system can be constructed or changes in an existing system can be implemented.

In particular this is system is developed in three major stages. First one is analysis process. In this process, we have described our problem and considered solutions to these problem. At the beginning, we have defined functional and non-functional requirements. After requirements definition, we created use-case diagram and use-cases that will handle our requirements. Then we created class diagram according to classes which will be used in the project. Lastly, dynamic models of the program was created which are sequence diagram, state charts and activity diagram. These diagrams helped us to analysis project easily and shows that how system works, what kinds of functionalities are provided to user.

Secondly, we started system design and define the main systematic behavior of our simulator. We first defined our design goals in order to reduce complexity of our system. Then we divide our system into subsystems and we finished the decomposition of it. It is decided that, simulator is applied model view controller design pattern so modularity of the system will be increased. Lastly we derive the map of software-hardware architecture and addressed our key concerns.

Consequently, in this level of development of program, we design major structure of simulator. In our object design, we used five design patterns; façade, observer, bridge, singleton and abstract factory. After this, we derive the class interfaces and specified contracts using OCL.

Lessons Learned

In analysis design we learned how to describe a problem and derive appropriate solutions for that problem using functional requirements. Also we learned how to use tools like Visual Paradigm which help us to create diagrams such as use case, class and activity. In design report, we had to consider lots of different implementation types. First, among different kind of patterns, we decided to use MVC pattern. Then, according to our class diagram, we draw our package diagrams by using Visual Paradigm. In order to make a proper project we critically thought about our design and changed it in case of need. After design report, we continued with object design. In object design process, we learned how to use OCL's by specifying contracts with it. We chose the patterns that are the most proper ones for our system and by applying those patterns we developed our initial design and we learned how to draw class interfaces. The project required a great deal of team work, we needed to show a better application of work design, in time reporting, synchronization among team and communication. So we learned how to work with different people in a restricted time period.

References

1. Object-Oriented Software Engineering, Using UML, Patterns, and Java, 2nd Edition, by Bernd Bruegge and Allen H. Dutoit, Prentice-Hall, 2004, ISBN: 0-13-047110-0.
2. "OpenTrack Railway Technology." - Railway Simulation. Web. 23 Mar. 2015. <http://www.opentrack.ch/opentrack/opentrack_e/opentrack_e.html#Events>.