**Multiplayer Race Car Project**

**ABSTRACT**

Car Race invites players to hop behind the wheel of a miniature race car and then compete with another car on the race track with style. Three, two, ones and zeros aren't scary on their own, but when they're combined into race track as the countdown to begin race, it certainly start beating the heart of everyone. Racetrack is one of the most lucrative video game in the world now a day. With a multi-client environment, most of the race track allows multiple players to compete. In this project, with the help of socket programming, thread implementation, marshalization we have made a program with 1 server that can handle up to 50 clients. We have used Java platform, Eclipse IDE and MAC operating system to develop the overall project. java.awt toolkit provides us with a very nice graphical environment. In the first, phase we have developed the race track, designed the car, set acceleration and given direction to the car and in the later phase we had developed the program for client server communication using socket programming. We then created one server and clients which allow the cars to be controlled and viewed on different workstations. We have implemented 360 degree rotation of the car, collision detection, acceleration of the car, sound, game start and end notification for all the users.

**Table of contents**

1. Introduction

1.1 Racetrack Client Side

1.2 Racetrack Server Side

2. Brief Description of the program

2.1.1 Establishing the Socket:

2.1.2 Collision Detection:

2.1.3 Sound for Collision:

3. Connection and the Protocols

3.1 Broadcasting

3.2 Marshalizing and Unmarshalizing the packet

3.3 Threads

4. System Design

4.1 Flow Diagram.

4.2 Class Diagram

4.3 Sequence Diagram

5. Conclusion

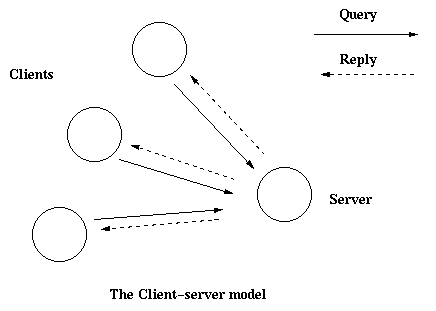
6. References

7. Screen shots of the program

**Chapter 1**

**Introduction**

Racetrack is one of the most lucrative video game in the world now a day. Because of the technological advancement, people of these days not only sit on their laptop and drive the car, they can race against each other over the network. To implement this more advanced technology and get a real taste of making a modern game, we have been assigned to design a race track. To implement the race track we have used the client-server method where a server is a computer system that selectively shares its resources; a client is a computer or computer program that initiates contact with a server in order to make use of a resource. Clients and servers exchange messages in a request-response messaging pattern: The client sends a request, and the server returns a response. To communicate, the computers must have a common language, and they must follow rules so that both the client and the server know what to expect.



**Figure1: Client- Server model**

**1.1 RacetrackClient Side**

In the client side we mainly used three packages, they are named as: controller, view, and model. Each and every package has their own classes for the connection and moving the objects.



**Figure2:** Some of the important Classes and Packages Client Side

* Class Carcontroller, controls the direction to the car, we use action ‘java.awt.event.KeyEvent’ for the movement of the car. Up key is pressed to go forward, down key is for the backward movement, left and right keys are pressed to go left and right direction.



Up key

For revolving the car in 360 degree rotation

Left key Right key

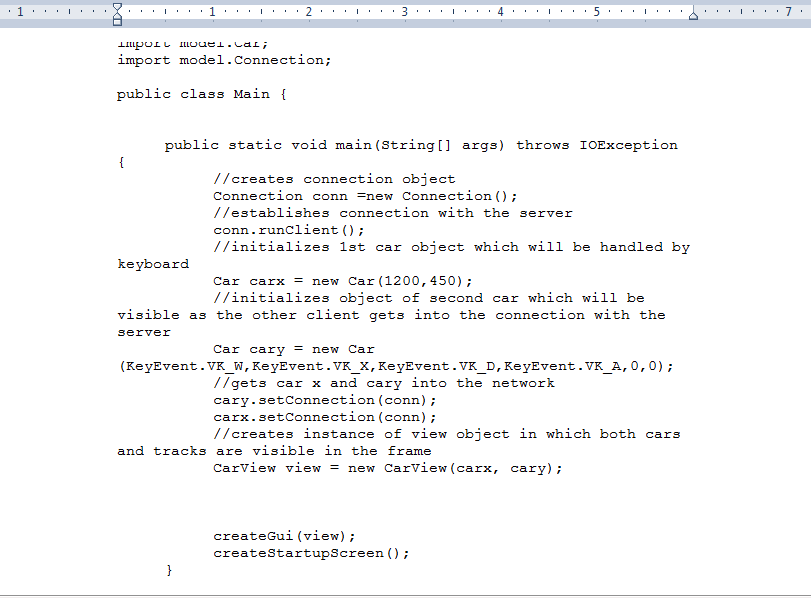
Down key

**Figure3: The Action Key Listener**

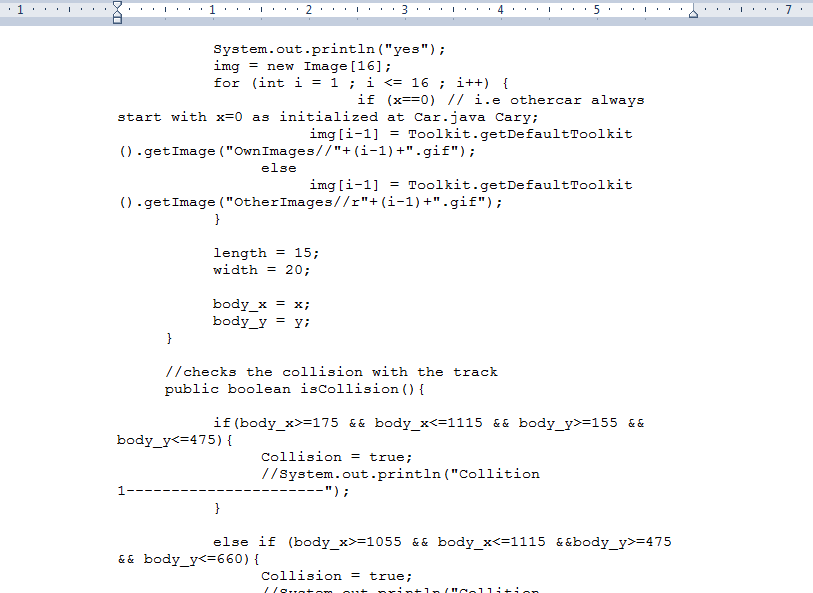
****

**Figure4: Code for the Controller class**

* Class Main is responsible for invoking and calling all the necessary functions and creating the calls ‘Car’, it also creates the construction for program. We use JFrame here instead of applet for the main class. In the Main class there is another class Connection used the TCP/IP socket for connection with the server.

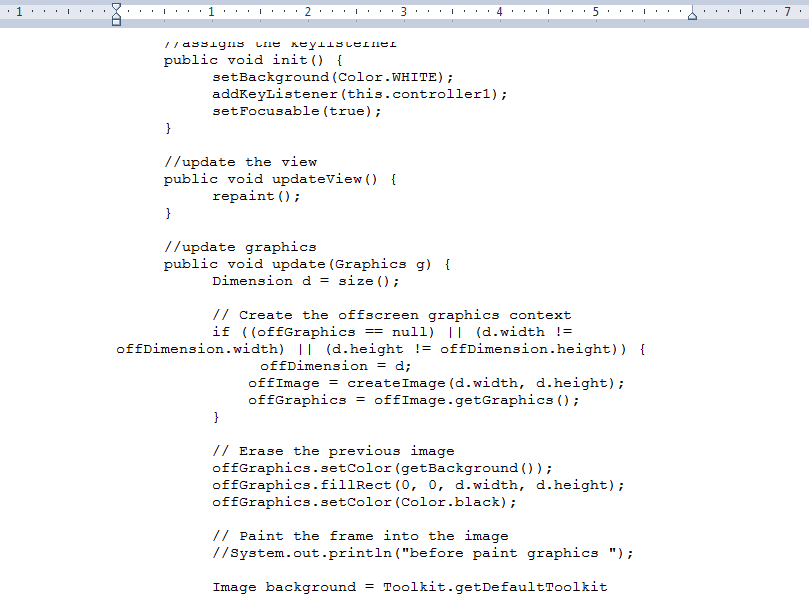
**Figure5: Code for the Main class**

* Car Class is pretty much responsible for the designing all the details about the car. It also makes sure that the car is in the right track. It also detects the collision.

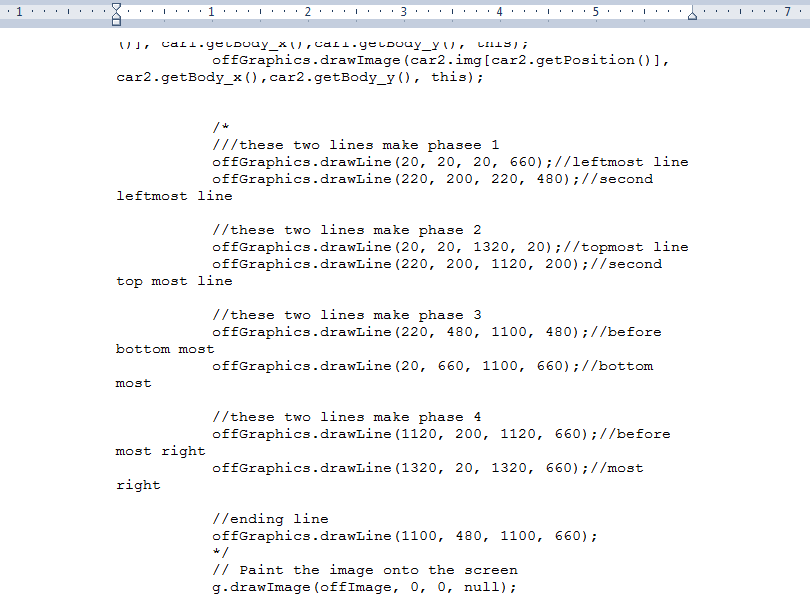


**Figure6: Code for the Car class**

* Car View is the class responsible for the graphical portion of the car. It also draws the track very nicely.



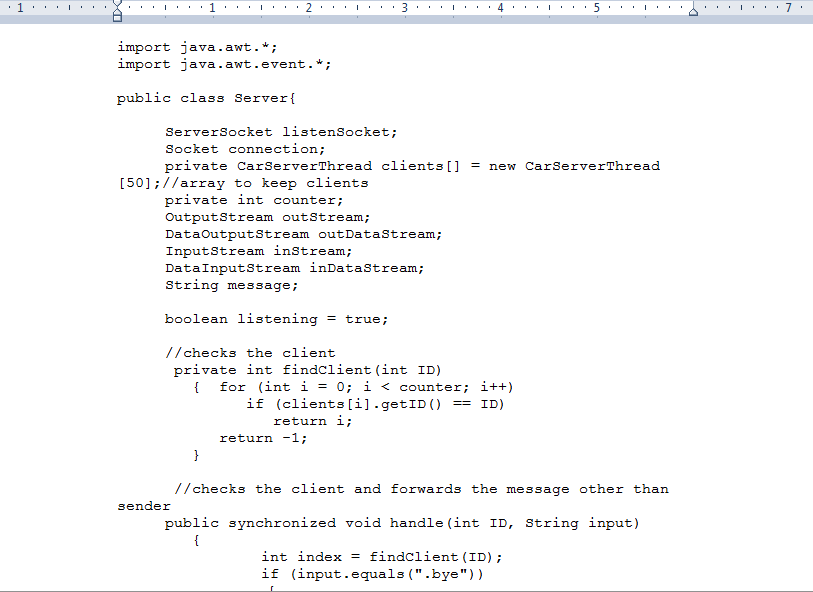
**Figure7: Code for the Car-view class**



**Figure8: Code for the Car-view class to draw track**

**1.2 Racetrack Server Side**

The server side is mainly consist of the all the thread and it does synchronization. We have used two clients which will work with the server, but our program supports up to fifty clients. We use TCP/IP socket to connect the client portion with the server end and server side was responsible for to synchronous the clients with the server. In server end use another portion for all the threading. An entire server is created for the client connection and synchronization.

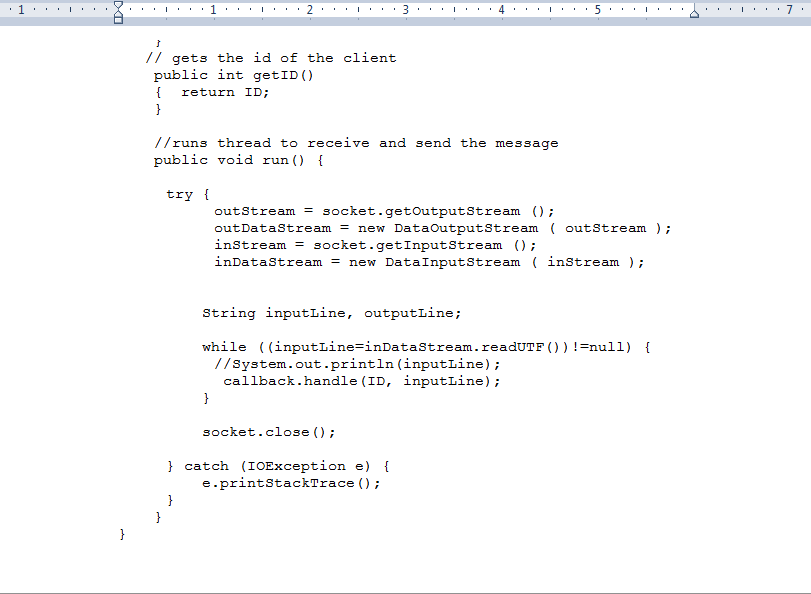


**Figure9: Code for the server**

**Chapter 2**

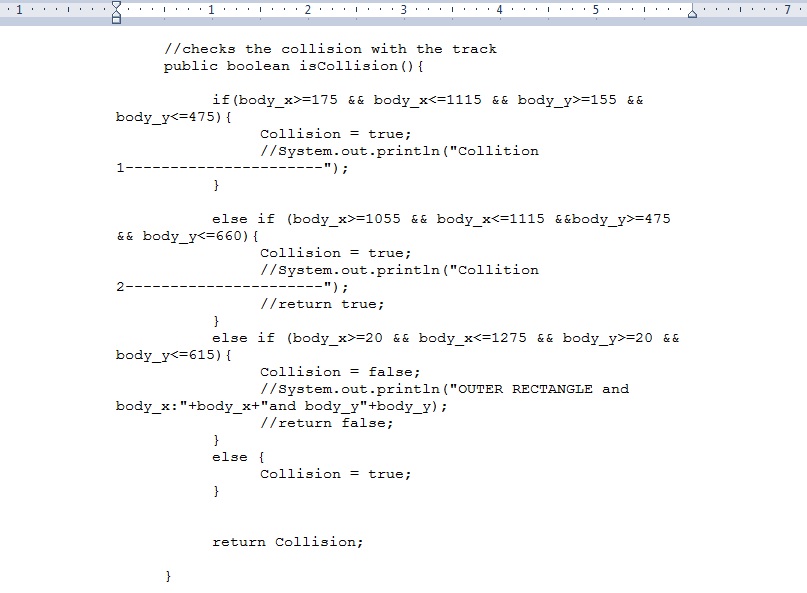
**2.1 Brief Description of the program:** This program is made of clients with one sever. The program Racetrack is a basically car game where many players can run their car in a given track. There some objects and side lines are provided to make sure that the car will not get out of the track. To make it more realistic we add sound and the collision detection in the edge of the track, start and end indication feature to our race track project. To maintain concurrency we have used multiple threads. We use TCP/IP socket for the interface and connection. Some features of the program are described below:

**2.1.1 Establishing the Socket:**Implementing the socket is the main backbone to establish the connection between the client and the server part. We first connected the client in the local machine and later we put the IP address of the server instead of local machine. Necessary exceptions are checked in the client end specially I/O exceptions. At first the data was send in bit stream but later it was marshaled and was as a packet or an objects.



**Figure10: Code for Establishing the Socket**

**2.1.2Collision Detection:**One of the main objectives of the race track is to detect the collision with the edge. The car cannot be out of the track so we implemented the collision detection to get rid of this. We fixed the coordinate (X,Y) so that the car will be inside the track.



**Figure11: Code for Collision detection**

****

**Figure12: Shows the collision**

**2.1.3 Sound for Collision:**We have implemented the sound when and edge is detected. The main purpose of the sound just is to make the game more is live from the user point of view. An audio file was kept in a directory and then it was retrieved the collision happens.

****

**Figure 13: Code for Collision Sound**

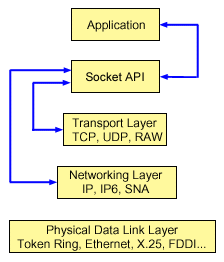
**Chapter 3**

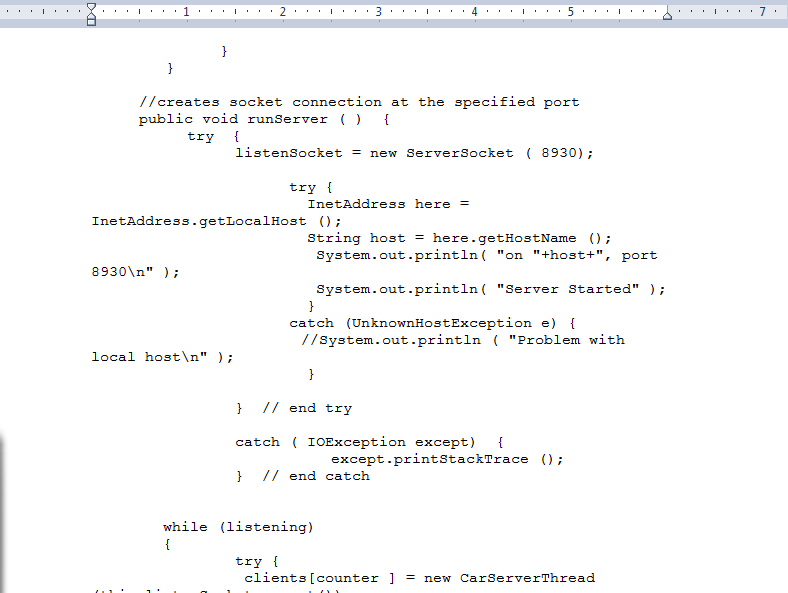
**3.1 Connection and the Protocols:** Using TCP sockets in the application allows us to broadcast the data to all the hosts on the network. Sockets are commonly used for client/server interaction. A socket has a typical flow of events.

This is a typical flow of events for a connection-oriented socket:

1. The socket() API creates an endpoint for communications and returns a socket descriptor that represents the endpoint.
2. When an application has a socket descriptor, it can bind a unique name to the socket. Servers must bind a name to be accessible from the network.
3. The listen() API indicates a willingness to accept client connection requests. When a listen() API is issued for a socket, that socket cannot actively initiate connection requests. The listen() API is issued after a socket is allocated with a socket() API and the bind() API binds a name to the socket. A listen() API must be issued before an accept() API is issued.
4. The client application uses a connect() API on a stream socket to establish a connection to the server.
5. The server application uses the accept() API to accept a client connection request. The server must issue the bind() and listen() APIs successfully before it can issue an accept() API.
6. When a connection is established between stream sockets (between client and server), you can use any of the socket API data transfer APIs. Clients and servers have many data transfer APIs from which to choose, such as send(), recv(), read(), write(), and others.

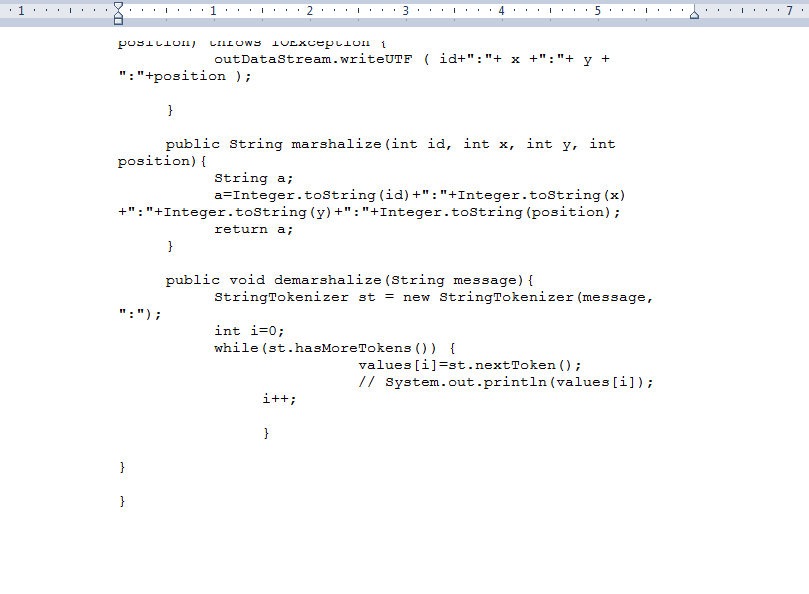
When a server or client wants to stop operations, it issues a close() API to release any system resources acquired by the socket.





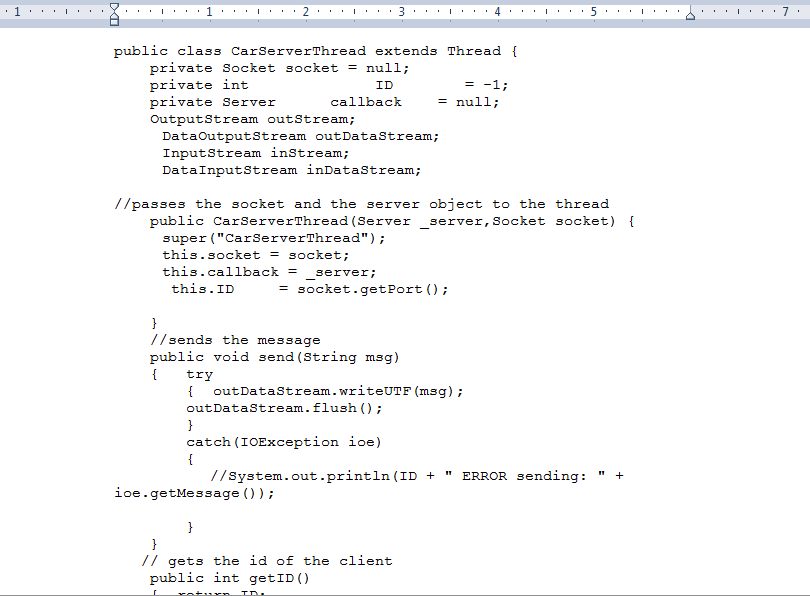
**Figure14: Socket connection at the specified port**

**3.2 Marshaling and Unmarshaling the packet:** In the earlier version of the code data was sent by using the bit stream but one of the challenging phase was to implement the marshaling and unmarshaling the packet. A sample of marshaling the packet is given below:



**Figure15:Code for marshaling and unmarshaling the message**

**3.3 Threads:** In this kind of client server program the most important part is managing the thread. As this is multi clients program therefore we have to use threads in different parts of the code a demo of the code is given below:

****

**Figure16: Code for threading in the server side**

**Chapter 4**

**4.1Flow Diagram:** The flowchart for the client and server communication is as follows:



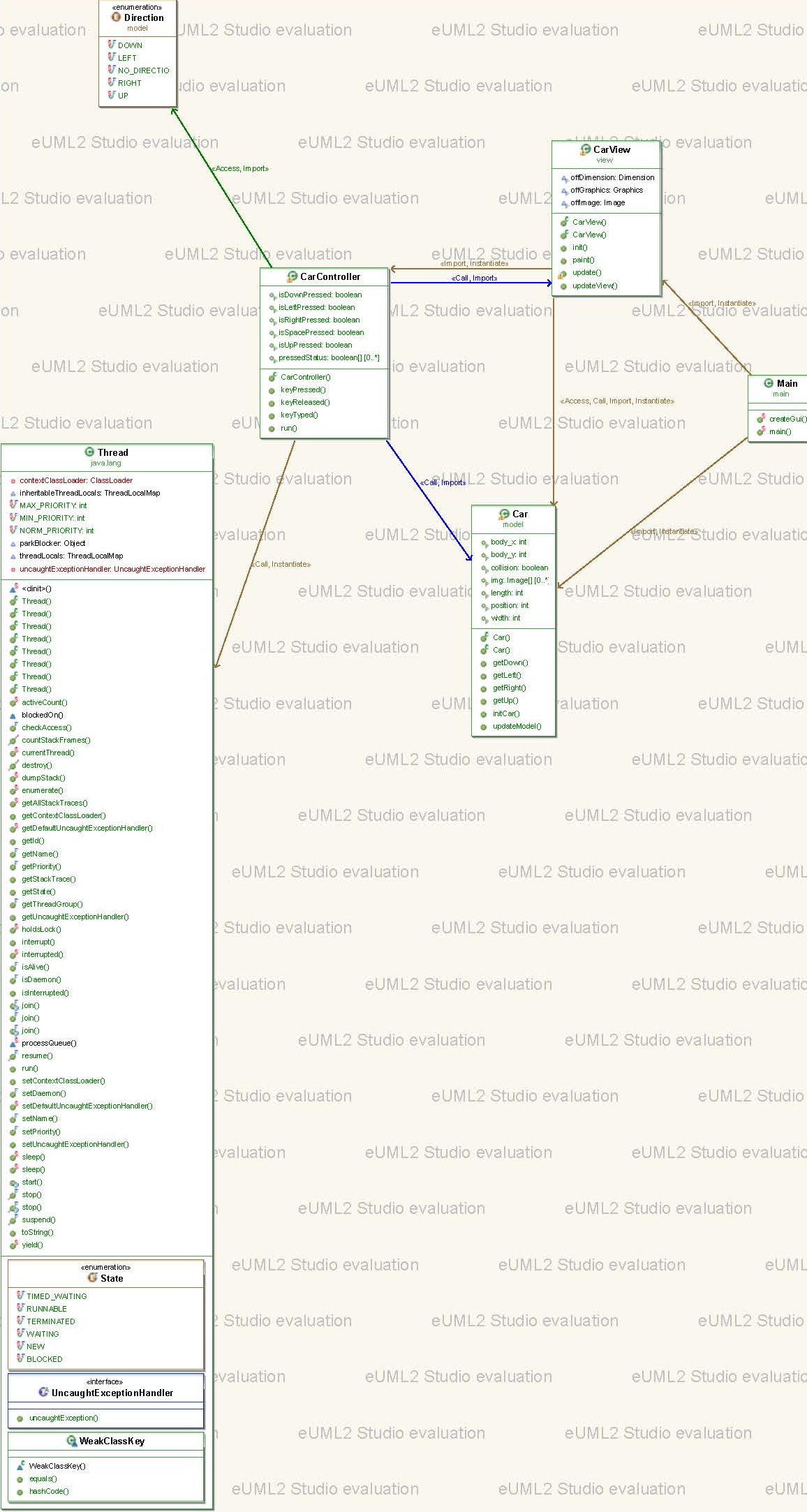
**Figure17: Flow diagram of the race track program**

**4.2 Sequence Diagram:** The sequence diagram is manly consists of all the time space and program flow the major flow of the program has been given below:



**Figure18: Sequence diagram of the race track program**

**4.3 Class Diagram:** The major classes and the thread are given in the following class diagram



****

Due to huge size of the Class diagram we have to compress file. Click the attachedpdf file for details

**Figure19: Class diagram of the race track program**

**Chapter 5**

**Conclusion**

Definition of the distributed system says that it’s the collection of independent computers that appear to its users as a single coherent system. To successfully implement the concept we need to know different architecture of the distributed system, need to know the concept of thread, client, server, code migration etc. This project gives us the primary knowledge of each of those concepts. We also implemented multi-client concept here. Due to the availability of time we were not able to add much graphics features. In future we will try to add some more features like collision detection between cars, time clock to count race time and so on.

**Chapter 6.**

**References:** The following website we followed to do the assignment

<http://java.sun.com/developer/technicalArticles/ALT/sockets/>

<http://www.dreamincode.net/forums/topic/28410-application-to-japplet-and-reverse/>

<http://stackoverflow.com/questions/26305/how-can-i-play-sound-in-java>  
<http://www.javaworld.com/javaworld/jw-12-1996/jw-12-sockets.html>

<https://moodle.cs.ualberta.ca/mod/resource/view.php?id=4565>

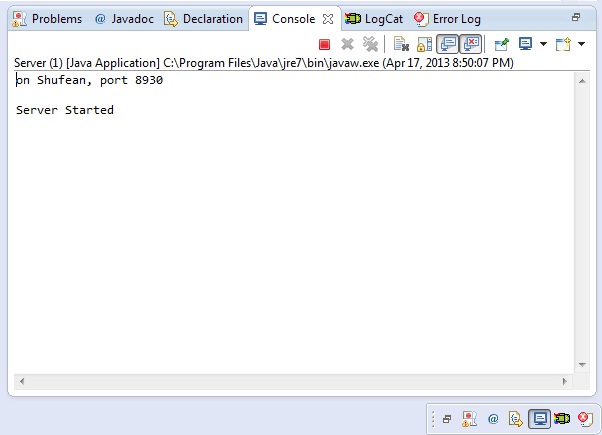
<http://www.zzrose.com/tech/pmr_sweSimpleMVCSwingExample.html>

<http://java.sun.com/developer/onlineTraining/Programming/BasicJava2/Code/SocketThrdServer.java><http://www.oracle.com/technetwork/java/socket-140484.html>

http://www.dreamincode.net/forums/topic/260214-collision-check-logic-question/

**Chapter 7**

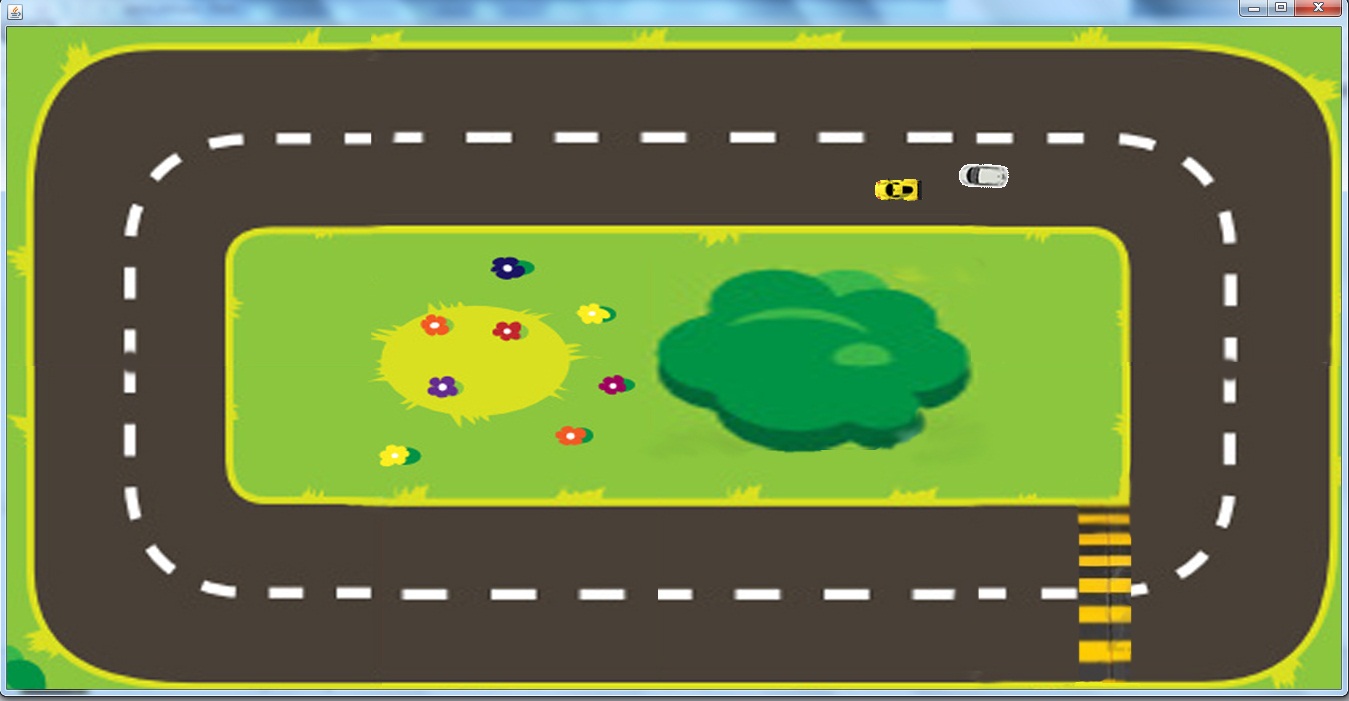
**Screen shots of the program**

****

**Figure: Shows the server program is running**



**Figure: Shows the start of the race**

****

**Figure: Shows the complete racetrack with two racing cars**

****

**Figure: Shows the end of the race**