Yasser Almohammad

Simulation of a Railway Switching Yard

Intro:

In this assignment we'll have a look at the basic data structures used, the packages constructed and the classes, how they work and their relations with each other.
More discussion will follow about the graphical simulation.
Followed by calculation of Complexity of the most significant algorithms, then few samples of the final application we've got.

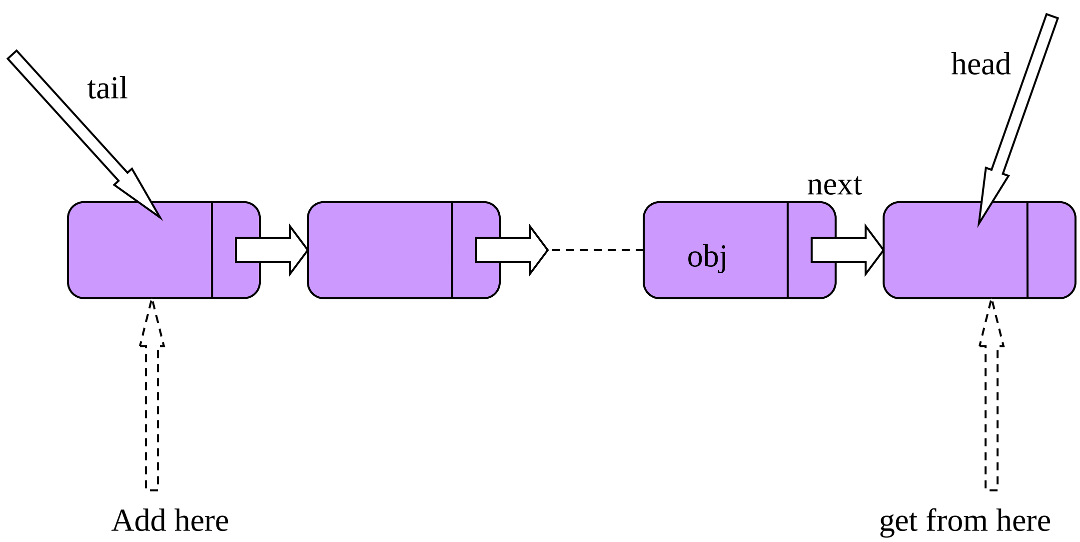
The programming language used in this assignment was Java, because it's very flexible modular, and can help build systems in a relatively acceptable time, and the whole application was OO.
A documentation of the whole project and every method is also available, since it can be generated automatically in java by including the proper commented tags and running javadoc tool, so much of the methods explained bellow are taken directly from the documentation, some features of JDK 1.5 are used and so the JDK 1.5 is required to run the final project.

Basic Data Structures:

These are the stack and queue, which are the main structures that the whole program is running according to their rules.

Two classes were made for his purpose which are: **MyQueue**, and **MyStack**.

MyQueue class:
elements are added in the tail side and de-queued on the head side, so it looks like this:



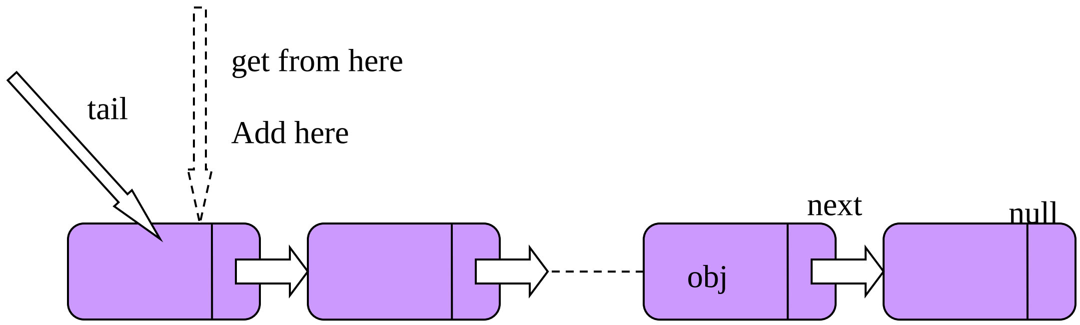
Element of this queue are of Node type, which is a class that holds an object value and a next value of the next element, thus allowing the value to be of any type, as for constructing this Node object, it's hidden from the programmer by including it's code inside the queue method

Field Summary	
protected Node	**head**
protected Node	**tail**
Constructor Summary	
MyQueue()	

Method Summary	
java.lang.Object	dequeue() remove from head
void	destroyStack() method not required in Java, since the GC does it's work just fine but it's added for formality of the probelm
int	getLength() we maintain a count of the stack elements
boolean	isEmpty() check content availability
java.lang.Object	peek() just get head without removing it
java.lang.Object	queue(java.lang.Object obj) add to tail
java.lang.String	toString() nicely concatenate it's internal element toString returns

As stated above, objects queued are objects of any type, and how the queue manages it's elements is irrelevant to the programmer, it could be using an array, a linked list or what ever.

The second class is **MyStack** :



Elements are added and removed from top: last in first out.
[like MyQueue]Element of this stack are of Node type, which is a class that holds an object value and a next value of the next element, thus allowing the value to be of any type, as for constructing this Node object, it's hidden from the programmer by including it's code inside the push method, and extract it back when requested back

Field Summary	
protected Node	**head**

Constructor Summary
MyStack()

Method Summary	
void	destroyStack() method not required in Java, since the GC does it's work just fine but it's added for formality of the probelm
int	getLength() count of elements
boolean	isEmpty() check content availability
java.lang.Object	peek() just see it without poping it
java.lang.Object	pop() pop the top out of the stack
java.lang.Object	push(java.lang.Object obj) on the top, push the passed object
java.lang.String	toString() nicely concatenate it's elements

So the implementation of both MyStack and MyQueue are a linked list implementations.

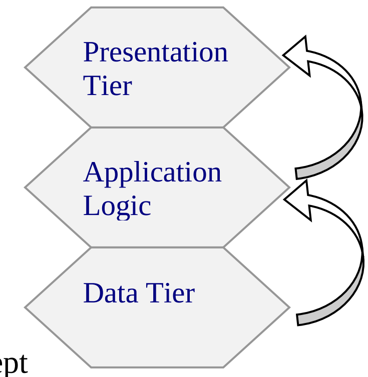
Packages, Classes, and how they work with each other:

The project was divided into three main packages:

Packages
myUtils
railwaySwitchingCore
railwayswitchingyardsim

Each has a different isolated task from the other, we'll come to mention each package and it's classes and how each one does it's work to do the switching simulation task,

we can say that our work depends on three tier mechanism: the data tier is formed of one class that is responsible for loading the simulation data into the application logic tier that does it's work and present data to the user.



The first two packages above contains the logic, and the Second package contains a class for data loading
The final package is the UI classes only.

There is nothing to say about the final package classes, except that they are the main program that populates a frame to view the final output, but we'll talk about the first two package that truly matters.

Package myUtils	Class Summary
CarQueue	
CarStack	
MyQueue	Already discussed
MySortedList	Used in the second assignment
MyStack	Already discussed
Node	Already discussed

Class CarStack

```
java.lang.Object
  └myUtils.MyStack
      └myUtils.CarStack
```

The graphical Car data, this class extends MyStack class, so it will have a stack functionality but for of TrainCar objects, some methods are overridden to provide additional info related to graphical context.
each car will be drawn by itself, drawing will happen from left to right

Field Summary	
(package private) int	**carHeight**
(package private) int	**carWidth**
java.awt.Color	**color**
(package private) int	**height**
(package private) static int	**margin**
(package private) int	**width**
(package private) int	**x**
(package private) int	**y**

Method Summary	
void	**draw**(java.awt.Graphics2D g) just draw a Rectangle for this stack place then forward draw command to it's stack elements [cars]
TrainCar	**pop**() updates the coordinates
TrainCar	**push**(TrainCar car) coordinate data are inherited from the head each insertion propagates change through the whole stack
void	**updateCoords**() after each change to the stack we call this internally so we update coordinates of each car, update propagation.

So for example: the method's pop implementation in this class overrides that of the parent by updating the coordinates of the remaining elements in the class, so it will be like this:

```
Super.pop();
updateCoords(…);
```

so we call the parent implementation and put the additions we need.

As seen above the two methods are overridden and new two are created: the draw and updateCoords();

Class CarQueue

```
java.lang.Object
  └myUtils.MyQueue
      └myUtils.CarQueue
```

Graphical Car data.
drawing will be from right to left [from head to tail]

this is a queue with special added functionality to draw cars and have a graphical representation.

Method Summary	
`TrainCar`	`dequeue()` as a car is dequeued, coordinates are updated
`void`	`draw(java.awt.Graphics2D g)` we draw the queue then forward the drawing command to it's cars
`TrainCar`	`queue(TrainCar car)` coordinate data are inherited from the tail
`void`	`updateCoords()` updates each car coordinates after each change

Just like CarStack, two methods are overridden and new two are created as explained before.

It was a good idea to do things this way: creating a general purpose structures then extending them and overriding the required methods, it made the job much easier especially that similar classes are needed in the two assignments.

Now we move to the second package:

Package railwaySwitchingCore

Class Summary	
LogMsg	Logging messages
Track	A sub train track
TrainCar	One train car
TrainDataLoader	Loader for the train file
TrainSwitching	The controller of the simulation
YardCoords	Predefined coordinates of the main graphical entities

We'll discuss every one of the above classes:

Class LogMsg:

it logs messages to both a TextArea field if available also to the standard output. the sole rule of this class is to output logging messages in any way designed in the logMsg method, which here is to the standard output and to a TextArea object if available, this method could be changed to log to a file too, thus happens without the need to change anything else in the code.

Field Summary	
static javax.swing.JTextArea	textArea
Constructor Summary	
LogMsg()	
Method Summary	
static void	logMsg(java.lang.Object obj)
static void	logMsg(java.lang.String str)

Class Track:

the track class will hold a stack and a track info, plus additional billing info

Field Summary	
java.lang.String	destCity
java.lang.String	name

Constructor Summary
Track()
Track(java.lang.String trackName, java.lang.String dest) build a track from a name and a destined city

Method Summary	
void	addCar(TrainCar car) adds a car to this track, updates total weight and bill info
void	closeTrain() empty method to issue a total billing command and close related resources

Class TrainCar

A train car object will have all necessary information to tell it's identity and information, it is also responsible for drawing itself over a graphical surface within a limited space defined by the graphical fields mentioned bellow.

Field Summary	
java.lang.String	cargo
int	carNum
java.lang.String	destination
int	height graphical height

int	miles
java.lang.String	origin
int	weight
int	width graphical width
int	x graphical x pos
int	y graphical y pos

Constructor Summary
TrainCar()
TrainCar(int carNum, java.lang.String cargo, java.lang.String origin, java.lang.String destination, int weight, int miles)

Method Summary	
void	draw(java.awt.Graphics2D g) a car is responsible for drawing itself
java.lang.String	toString() formats the car info in a proper way to display

Class TrainDataLoader:

we'll use this class static method to load train data from an input file, the static method accepts a file path and returns a CarQueue that represents the main train.
The loader handles errors and reports them when they happen.

Constructor Summary
TrainDataLoader()

Method Summary	
static CarQueue	loadTrainInfo(java.lang.String filePath) passing an input file path we construct the CarQueue objecy

To read from a file we simply create a BufferedReader instance out of a FileReader instance:

```
reader=new BufferedReader(new FileReader(filePath));
while( line=reader.readLine() !=null){
        tokenize the line
        build a train car
        get tokens
        parse the read tokens and update the car info
        add the car to the queue
}
//Close resource
```

Class YardCoords

class designated to calculate the main object coordinates in the yard, as stated bellow these objects are the main train, the subtrain and the transfer train.
These coordinates are rectangular shapes that represent the space in which a matching object should draw itself inside.
These coordinates are relative, so no matter on which surface the target was drawn it will get a fine coordinates.

Field Summary	
static java.awt.Rectangle	mainTrain
static java.awt.Rectangle[]	subTrain
static java.awt.Rectangle	transferTrain

Constructor Summary
YardCoords()

Method Summary	
static void	calcCoords(int width, int height)

Class TrainSwitching:

our core class that maintains everything attached together to do the task of this project the class will hold the main train info, a transfer train, and 4 subtrains.
it also has the graphical information to draw against.

Nested Class Summary	
(package private) class	TrainSwitching.SwitchingThread

Field Summary	
boolean	enableSwitching [set true to stop simulation]
int	height height of the image to create
static java.awt.image.BufferedImage	image draw on it's graphics context
java.awt.Graphics2D	img the Graphical context of the image
(package private) CarQueue	mainTrain
long	step
(package private) CarStack[]	subTrains
java.awt.Graphics2D	targetGraphics on which we flip the image

	onto later to display onscreen for example
(package private) CarStack	transfer temp stack to transfer coupled cars
int	width width of the image to create

Constructor Summary
TrainSwitching(java.lang.String filePath, int width, int height, java.awt.Graphics2D g) passing a file name to load data from and width, height of the target window and a graphics device to drawn against the main train is loaded and becomes ready to switch

Method Summary	
void	billTrains() go through all sub trains and make an accounting file for each, this should be the final method to get called.
void	switchTrains() this method initiates a new thread to do the switching thing the thread do the switching and calls the rendering
void	viewMainTrain() views the main train content on the standard output and in a logging text area
void	viewSubTrains() views the sub trains content on the standard output

The methods are straight forward, for example:
To view the main train content just call: mainTrain.toString() to get a string representation of the train and it's content, since all elements overrides the toString() method.
Also billTrains() just call each train bill() method and display the result.

```
Create a File object
Create a PrintWriter object of the File object to allow type writting
Create a StringBuffer to append Strings to

Foreach(train:subTrain){
        train.calcBill();
        create a Formatter object to write on a StringBuffer a formatter Strings like:
        StringBuffer str = new StringBuffer();
        Formatter formatter = new Formatter(str);
        formatter.format(" Car number : %d\r\n Cargo : %s \r\n…",car.carNum,
            car.origin…);
}
...
Then write info to the PrintWriter

Close resources
```

The Formatter Object is introduced in java 1.5 and allows a C like string formatting with precision and length determination of arguments, for example:
fobj.format("%6d",44332.32303)→ 44332.3

now a call to switchTrain() method initiates a new Thread that does the switching and commits the drawing on screen, thus no blocking for the UI happens, next is an explanation of this thread:

Class TrainSwitching.SwitchingThread

```
java.lang.Object
  └java.lang.Thread
      └railwaySwitchingCore.TrainSwitching.SwitchingThread
```

an internal class to do the switching job, without locking the main UI

Field Summary	
(package private) java.awt.image.AffineTransformOp	op needed for the image

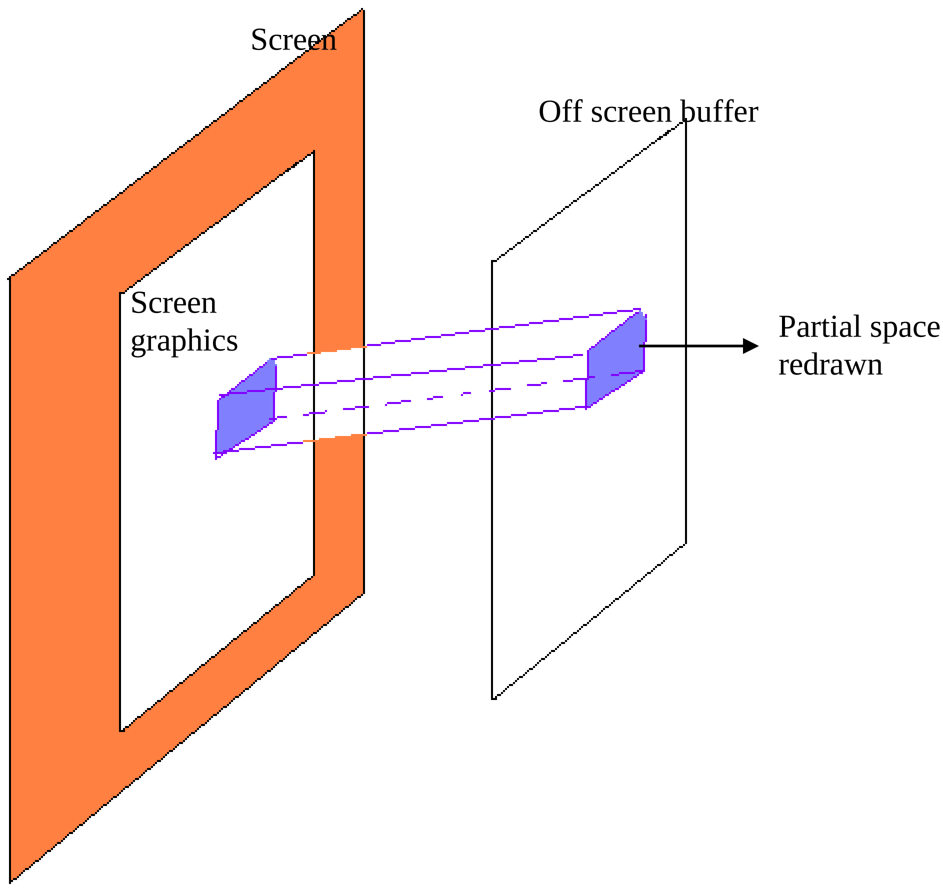
Method Summary	
void	commitScene() all changes to the image draw is commited to the screen immediately
void	drawYard(java.awt.Graphics2D g, int width, int height) initially we draw the yard it self, with no cars
void	run() begin the simulation, called upon starting the thread
void	step() one step by sleeping for a while and updating the figure drawn

Mechanism of switching is done as stated in the assignment script by using the help of the transfer stack.

Now we'll have more in depth discussion of the drawing mechanism used:

Graphics:
drawing is done using double buffering technique, with partial changes taken into account:
So all drawing is done off screen, and flipped back onto screen at once, so no direct change to the graphic device happens only when copying off-screen onto screen.
When a small region changes in the figure, that space only gets changed, and not the whole scene is drawn again, for example: if the sub train was changed by adding a new train, only the sub train gets redrawn.
so drawing can't be any smoother.

In java we use a BufferedImage object for this purpose.

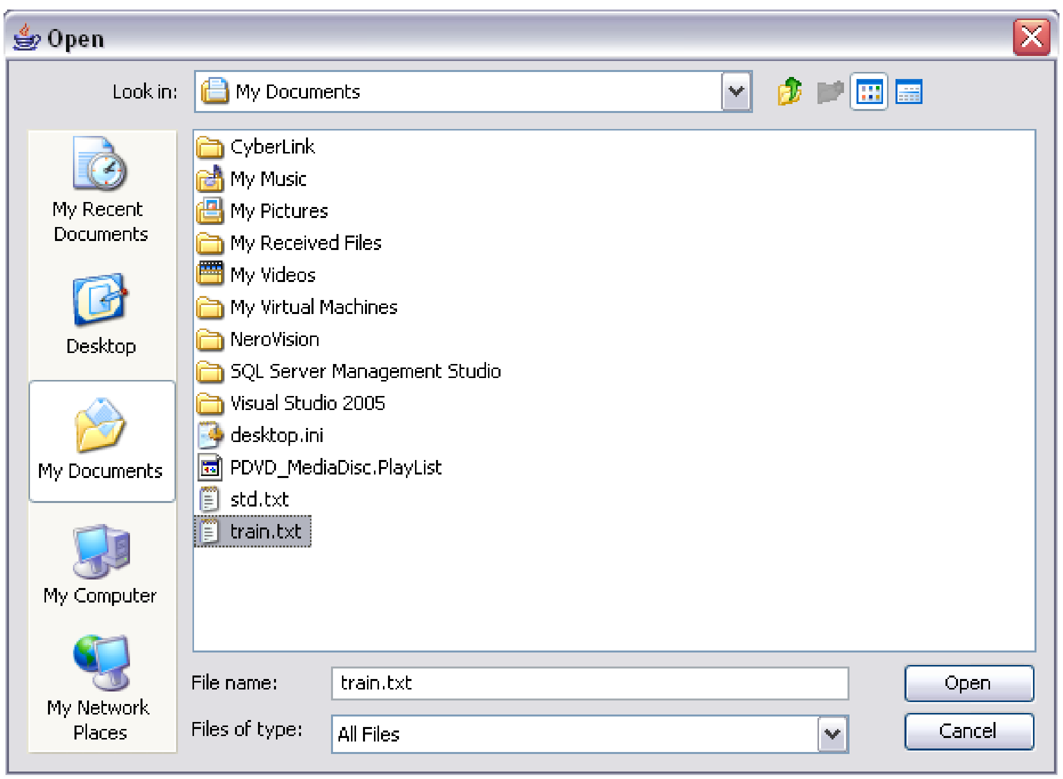


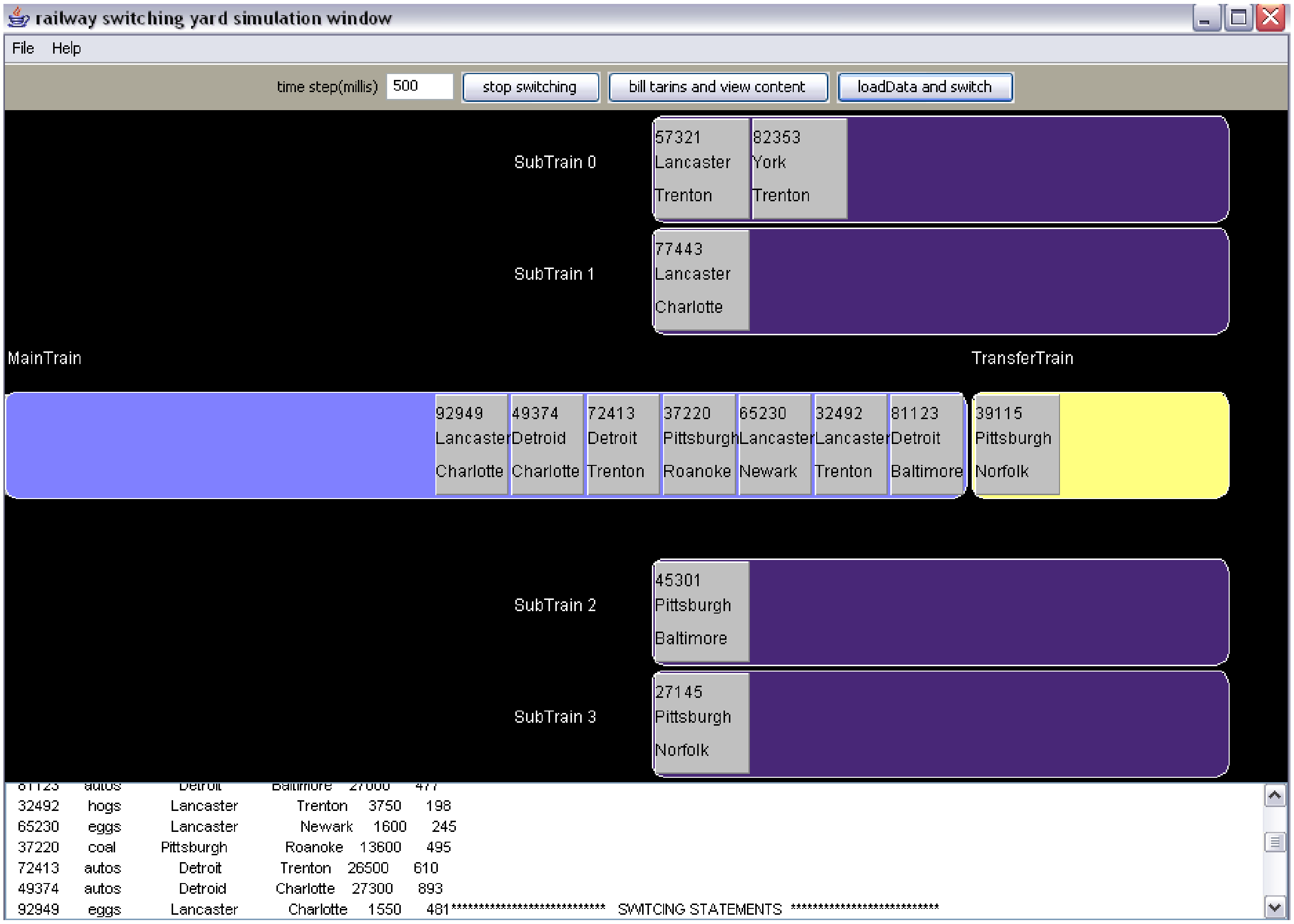
Every object is responsible for drawing itself, within a space handed to it by it's logical parent for example:

A car know how to draw it self, and it's handed it's own coordinates by the parent object which could be a the sub train stack or the main train.

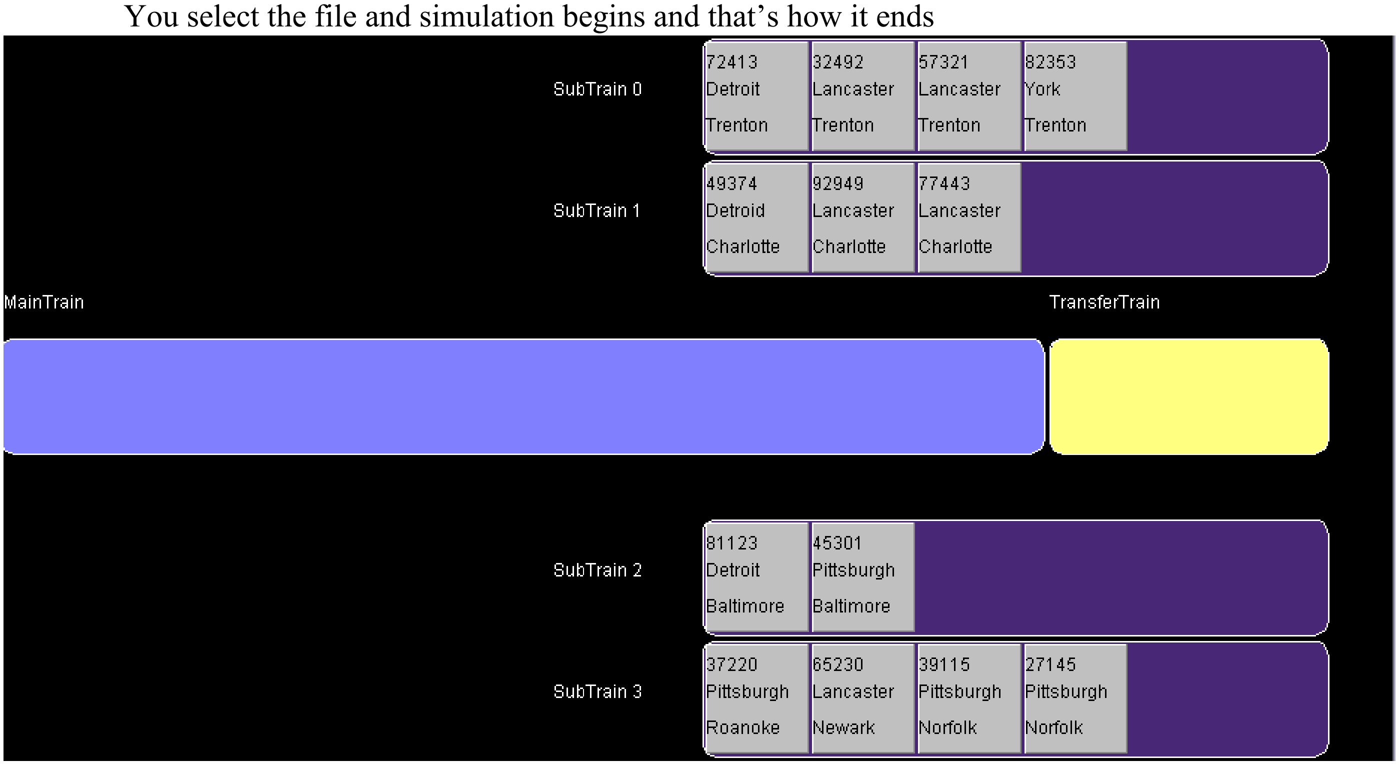
Drawing is done using steps inside a single thread, and thus prevents the UI from locking as we said before, and is done every step, which can be initially controlled by the user by setting it's value.

Samples and tests:
The main window initially looks like this:





You select the file and simulation begins and that's how it ends



Sample switching statements and an accounting file for the track1

```
uncoupling between cars 45301 and 57321 and back car 45301 onto track 3
uncoupling between cars 82353 and 77443 and back them onto track 1
uncoupling between cars 77443 and 39115 and back car 77443 onto track 2
uncoupling between cars 27145 and 81123 and back them onto track 4
uncoupling between cars 81123 and 32492 and back car 81123 onto track 3
uncoupling between cars 32492 and 65230 and back car 32492 onto track 1
uncoupling between cars 65230 and 37220 and back car 65230 onto track 4
uncoupling between cars 37220 and 72413 and back car 37220 onto track 4
uncoupling between cars 72413 and 49374 and back car 72413 onto track 1
Back remaining cars on track 2
```

```
Track 1 : A total of 4 cars bound for Trenton:
 Car number : 72413
      Cargo : autos
     Origin : Detroit
Destination : Trenton
     Wieght : 26500 pounds
   Distance : 610 miles
       Cost : $8082

 Car number : 32492
      Cargo : hogs
     Origin : Lancaster
Destination : Trenton
     Wieght : 3750 pounds
   Distance : 198 miles
       Cost : $371

 Car number : 57321
      Cargo : cattle
     Origin : Lancaster
Destination : Trenton
     Wieght : 5000 pounds
   Distance : 198 miles
       Cost : $495

 Car number : 82353
      Cargo : hogs
     Origin : York
Destination : Trenton
     Wieght : 3500 pounds
   Distance : 181 miles
       Cost : $316

Total weight:19.375 tons
 Total Bill : $9264.0
```

Complexity of algorithms:

Method name	Complexity	algortihm
MyStack method		
MyQueue method		

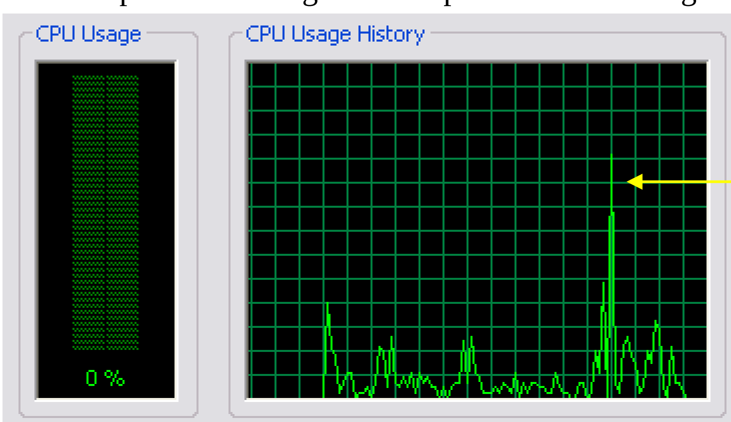
CarStack.updateCoords(…)	O(n)	While(more elements){
		Elem.updateCoords(…)
		}

CarQueue.updateCoords(…)	O(n)	While(more elements){
		Elem.updateCoords(…)
		}

SwitchingThread.run()	2n max => O(N)	While(!mainTrain.isEmpty()){
		.
		.
		.
		While(!transfer.isEmpty()){
		Push(pop)
		}
		}

All methods here are linear methods and complexity is either 1 or O(n) max, since all methods depend on direct loops or on the stack-queue functions.

The drawing as we do it, takes a fraction of the CPU processing time that varies depending on the time step, which normally takes less that 5-10% of the CPU speed you can see upon monitoring the CPU performance during 5 different simulations.



This peek happens when an showing the OpenDialog window and beginning the simulation (5 peeks), during the remaining of the simulation the speed almost settles

Test1[3 tests]:
2steps/sec

Test2[3 tests]:
60-20steps/sec