AFRICAN UNIVERSITY OF SCIENCE AND TECHNOLOGY

ABUJA

COMPUTER SCIENCE DEPARTMENT

MSc COMPUTER SCIENCE THESIS

A MODEL-VIEW-CONTROLLER BASED PLATFORM FOR THE MODELING AND SIMULATION OF THE HOIST SCHEDULING PROBLEM

Author:
ELEONU HENRY CHIKA
Student ID: **40065**

Supervisor:
PROF. MAMADOU KABA TRAORÉ

December, 2010

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Prof. Mamadou Kaba Traoré, for his assistance in the past one year and throughout the period of research on this thesis. I am also grateful to all the faculty of computer science department for the knowledge they have imparted to me.

I would like to take this opportunity to thank the management and staff of African University of Science and Technology Abuja for there immense support throughout the period of the thesis. You have been very instrumental to the realization of this thesis. I can not thank you enough.

Finally, I would like to thank my family and friends who have supported me during the course of this dissertation. Without their assistance this work would not have been possible.

ABSTRACT

The hoist scheduling problem is a critical issue in the design and control of many manufacturing processes. When the hoist number and tank numbers are very large, finding an optimal schedule is very hard. As a result of this, a lot of scheduling algorithms have been developed, and thus created a need to evaluate these algorithms. This calls for a cheap and efficient way of evaluating different hoist scheduling algorithms. To address this issue, I propose a generic simulator which will be a visual tool that will be developed with Java technology. The result of this work will help to reduce cost and also help to guarantee product quality in production lines.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS ____ *II*

ABSTRACT ____ *III*

TABLE OF CONTENTS ____ *IV*

TABLE OF FIGURES ____ *VI*

LIST OF CODE LISTINGS ____ *VII*

1. INTRODUCTION ____ *1*

1.1 PROBLEM DEFINITION AND MOTIVATION ____ **1**

1.2 THESIS ____ **1**

2. HOIST SCHEDULING PROBLEM ____ *3*

2.1 INDUSTRIAL IMPORTANCE ____ **3**

2.2 HSP SOLUTIONS ____ **5**

2.3 NEED FOR EVALUATION TOOL ____ **5**

2.4 EXISTING TOOLS ____ **6**

3. MODEL-VIEW-CONTROLLER ____ *7*

3.1 DESIGN PATTERNS ____ **7**

3.2 MVC PATTERN ____ **8**

3.3 WHY MVC ____ **10**

3.4 PROBLEM WITH SWING APPLICATION ____ **10**

4. DESIGN AND IMPLEMENTATION OF SIMULATOR ____ *12*

4.1 PROBLEM PARAMETER AND DESCRIPTION ____ **12**

4.2 MVC STRATEGY ____ **13**

4.2.1 Use-Case Diagram ____ 14

4.2.2 Sequence Diagram ____ 15

4.3 THE VIEW ____ **16**

4.3.1 UML Class Diagram for the View ____ 16

4.4 THE CONTROLLER ____ **19**

4.5 THE MODEL ____ **21**

4.6 MVC DESIGN ____ **23**

4.7 USER INTERFACE ____ **25**

4.8 INPUTS AND OUTPUTS OF THE SIMULATOR ____ **30**

4.8.1 Inputs to the System ____ 30

4.8.2 Outputs to the System ____ 32

4.9 IMPLEMENTATION ____ **35**

4.9.1 Thread Implementation ____ 36

4.9.1 MVC Implementation ____ 42

4.9.2 BIRT Report Implementation ____ 47

5. TESTING AND RESULTS ____ *50*

5.1 TESTING AND RESULTS ____ **50**
5.1.1 First Test ____ 51
5.1.2 Second Test ____ 54
5.1.3 Third Test ____ 57

5.2 EVALUATION AND FINDINGS ____ **59**
5.2.1 Evaluation ____ 59
5.2.2 Findings ____ 60

6. CONCLUSIONS AND FURTHER RESEARCH ____ ***62***

7. REFERENCES ____ ***63***

TABLE OF FIGURES

Figure 1 A Hoist System ____ *4*
Figure 2 MVC of Swing Applications ____ *11*
Figure 3 MVC of Simulator ____ *13*
Figure 4 Use Case Diagram of System ____ *14*
Figure 5 High-Level Sequence Diagram of Simulator ____ *15*
Figure 6 Class Diagram of View ____ *18*
Figure 7 Class Diagram for the Controllers ____ *20*
Figure 8 Class Diagram for the Model ____ *23*
Figure 9 Class Diagram showing MVC Relationships ____ *24*
Figure 10 Screen shot for the Main Window ____ *25*
Figure 11 Screen shot of Main window with two carriers ____ *26*
Figure 12 Screen shot for physical configuration ____ *26*
Figure 13 Screen Shot for treatment window ____ *27*
Figure 14 Screen shot for Initial Load Window ____ *29*
Figure 15 Screen shot for Initial States window ____ *29*
Figure 16 Screen Shot for Hoist Moves Window ____ *30*
Figure 17 Treatment Text File ____ *31*
Figure 18 hoist moves text file ____ *32*
Figure 19 Bath Report ____ *33*

LIST OF CODE LISTINGS

Listing 1: thread implementation in Simulation class……36

Listing 2: thread implementation in Hoist class…….37

Listing 3: thread implementation in Tank class…….40

Listing 4: InitialStatesController class……...42

Listing 5: event listener for Ok button in InitialStatesDialog class…...46

Listing 6: update of hoist's Y position in view from model……..…47

Listing 7: hoist image update…….47

Listing 8: execution of good bath report……48

1. INTRODUCTION

1.1 PROBLEM DEFINITION AND MOTIVATION

The purpose of this work is to build a simulator which will have a graphical user interface that can evaluate hoist scheduling algorithms. The Model-View-Controller (MVC) design pattern will be adopted. The simulator will simulate hoist moves of different hoist scheduling algorithms, and also evaluate these algorithms. It also report violations of imposed constraints and also compares the algorithms to find which gives optimal scheduling. The simulator will be implemented with Java.

The hoist scheduling problem (HSP) is encountered in many production lines in many industries. This problem has been proven to be NP complete problem. Consequently many heuristic algorithms have been proposed by many researchers to solve this problem. Problem arises on the scheduling algorithm to adopt in an automated hoist system. As a result of the numerous algorithms, there is need to have a visual tool to explore, evaluate and compare these algorithms.

1.2 THESIS

I am proposing a visual tool (simulator) that can be use to create visual simulation that can evaluate different hoist scheduling algorithms. I am proposing that moves computed from hoist scheduling algorithm should be used as input to this simulator, so as to make the evaluation of the algorithm easy and less expensive. The moves can be in the form of a text file.

We are also adopting a Model-View-Controller (MVC) architectural design pattern for this simulator.

2. HOIST SCHEDULING PROBLEM

2.1 INDUSTRIAL IMPORTANCE

The Hoist Scheduling Problem (HSP) deals with the scheduling of hoist that move product between tanks in electroplating facilities that perform chemical surface treatments. Electroplating lines are totally automated manufacturing systems that are used to cover parts with a coat of metal. They are encountered in many industries: spectacle trade, mechanics, jewelry, household electrical appliances, and printed circuit boards. Hoist system is also use in electroplating processes for the production of floppy disks, computer hard drives, communication network connectors and switches, Aerospace parts (airplane parts) etc. In particular, they allow the protection of parts from corrosion or give them some aesthetic properties. Such a process is important as it may constitute a bottleneck in the production cycle. The hoist system is used in chemical processing, food processing, metal and pharmaceutical production (Manier and Bloch., 2003). Integrated circuit (IC) manufacturing system is one of the most important new generation industries. The IC manufacturers try to increase their profits by improving the process technology and manufacturing efficiency. To do this, the hoist system is employed for material handling. Material handling hoists are commonly used in smelting and plating processes to transport materials between workstations.

There are time windows constraints (a minimum and a maximum values) for the time spend at each tank. This time window is a quality constraint which must be adhered to ensure the quality of the products in the

production line. In such production lines, any violation of the time window constraint will result in the product being discarded because standard must be adhered to. When ever there is a violation of any of these standards, the company involved will incur losses. As no inventories are allowed, this soak time tolerance is the only source of flexibility. Tanks and hoists can only process one job at a time. We also assume that one hoist must do all moves.

The control of the hoist's movements with respect to those constraints is known as the Hoist Scheduling Problem (HSP) (Lamothe et al, 1994).

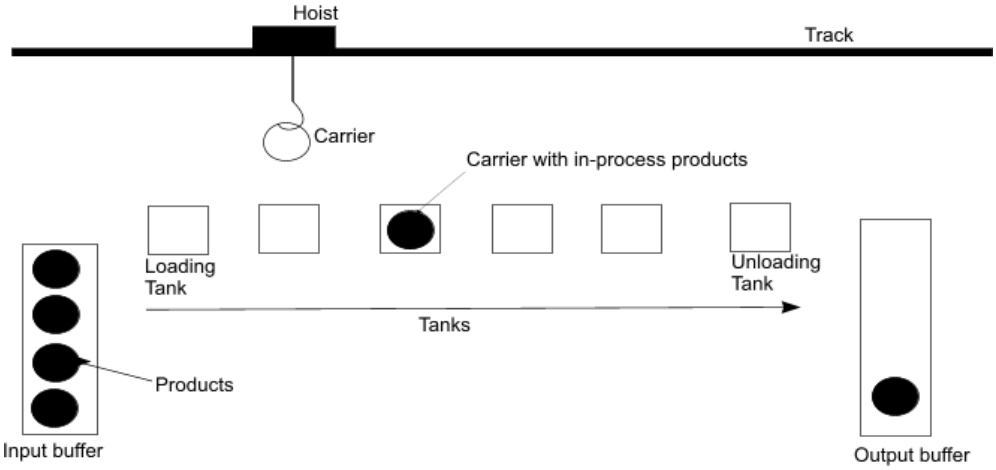

Figure 1 A Hoist System

The *scope* of the study is in the area of single hoist cyclic scheduling with multiple tanks. An example of this system is shown in Figure 1. The objective is to minimize the total production time and also have fewer products that are defective. Efficient scheduling of such hoists can improve production throughput dramatically. In many production lines, violations of the constraints can lead to defective job, which will be discarded, thus negative cost implication to the company. Thus the need for efficient hoist scheduling algorithms becomes very important. This has lead to many hoist scheduling algorithms being proposed.

As the number of tanks increase and the number of carriers that can be in the system at any point in time also increases it becomes very difficult to schedule the hoist to do all the moves, at the same time obeying all the time window constraints and other constraints that may be imposed on the system. Generally, the problem to determine the scheduling for operations done by a hoist with the objective to optimize the productivity appears as a NP-complete problem (Lei and Wang, 1989). The generic hoist scheduling problem is NP-hard and arises from automated manufacturing lines (Riera and Yorke-Smith, 2002).

2.2 HSP SOLUTIONS

Solutions to HSP are mainly heuristic algorithm solution. The first solutions to the hoist scheduling problem used mathematical programming (Phillips and Unger, 1976). Later, artificial intelligence techniques in the forms of local search and constraint logic programming (CLP) were applied (Baptiste et al., 1994; Lam, 1997). More recently, a hybrid technique that combines MIP and CLP has been developed (Rodoˇek and Wallace, 1998) and many other algorithms that I have not mentioned. So a problem arises on the hoist scheduling algorithm to adopt in an automated hoist system, and hence a need for an evaluation tool to evaluate the numerous hoist scheduling algorithms that may be an option for an automated hoist system.

2.3 NEED FOR EVALUATION TOOL

The material-handling operations (i.e., the operations to move jobs between stages or tanks) are performed by a computer-programmed robot. The programs that run on these computers are based on some of these hoist scheduling algorithms. These numerous algorithms needs to

be evaluated, to enable a company choose the best option that will help it to maximize profit. This calls for the need of a visual interactive tool that will simulate the hoist system and evaluate different moves that result from different algorithms.

My aim is to model, design and develop a simulator (visual tool) that can be use to simulate some of these hoist system classes and also evaluate some of the heuristic algorithms that have been proposed as solutions to the hoist scheduling problems of these hoist systems.

2.4 EXISTING TOOLS

No Visual tool has been identified to carry out the evaluations of hoist scheduling algorithms. To the best of my knowledge, this will be the first of its kind to be used.

3. MODEL-VIEW-CONTROLLER

3.1 DESIGN PATTERNS

A pattern for software architecture describes a particular recurring design problem that arises in specific design contexts, and presents a well proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate. (Buschmann et al., 1996). Three categories of patterns defined by (Buschmann et al.)

• Architectural patterns

• Design patterns

• Idioms

Architectural Patterns

Architectural Pattern is a high-level structure for software systems that contains a set of predefined sub-systems. The responsibilities of each sub-system are defined and detail the relationships between sub-systems. The Model View-Controller pattern falls under this category.

Design Patterns

Design Pattern is a Mid-level construct which is Implementation independent and Designed for 'micro-architectures' – somewhere between sub-system and individual components.

A design pattern is a documented best practice or core of a solution that has been applied successfully in multiple environments to solve a problem that recurs in a specific set of situations. Some of the characteristics of design patterns are:

- Help improve the quality of the software in terms of the software being reusable, maintainable, extensible, etc.
- Patterns are logical in nature.

- Pattern descriptions are usually independent of programming language or implementation details.
- Patterns are more generic in nature and can be used in almost any kind of application.
- A design pattern does not exist in the form of a software component on its own. It needs to be implemented explicitly each time it is used.
- Patterns provide a way to do “good” design and are used to help design frameworks.

(Kuchana, 2004)

Most design patterns also make software more modifiable. The reason for this is that they are time-tested solutions. Therefore, they have evolved into structures that can handle change more readily than what often first comes to mind as a solution.

The design patterns in this work are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context, as defined by Gamma and others (Gamma et al, 1995). In this work, I am adopting a Model-View-Controller (MVC) design pattern.

3.2 MVC PATTERN

One of the frequently cited frameworks was the Model-View-Controller framework for Smalltalk (Krasner and Pope, 1988), which divided the user interface problem into three parts. The parts were referred to as a data model which contains the application object or computational parts

of the program, the view, which presented the user interface, and the controller, which interacted between the user and the view. Before MVC, user interface designs tended to lump these objects together. MVC decouples them to increase flexibility and reuse

Model-View-Controller (MVC) is a classic design pattern often used by applications that need the ability to maintain multiple views of the same data. The MVC pattern hinges on a clean separation of objects into one of three categories — models for maintaining data, views for displaying all or a portion of the data, and controllers for handling events that affect the model or view(s). MVC is the pattern we are adopting for this thesis.

View: it renders the contents of the model; it is responsible for mapping graphics onto a device. A view typically has a one to one correspondence with a display surface and knows how to render to it. A view attaches to a model and renders its contents to the display surface.

Controller: Serves as a bridge between the View and the Model. A controller offers facilities to change the state of the model. The controller interprets the mouse and keyboard inputs from the user, commanding the model and/or the view to change as appropriate. It is the piece that manages user interaction with the model. It provides the mechanism by which changes are made to the state of the model. Based on the user interactions and the outcome of the model actions, the controller responds by selecting an appropriate view.

Model: The model is the piece that represents the state and low-level behaviour of the component. It manages the state and conducts all transformations on that state. The model has no specific knowledge of

either its controllers or its views. The view is the piece that manages the visual display of the state represented by the model. A model can have more than one view. The model represents real world objects.

3.3 WHY MVC

It allows for modular separation of function. Software developed using this design pattern is easier to maintain. It can allow change of Interface from Swing to three dimensions OpenGL or to another graphic API. The model can also be changed from UML model to DEVS (atomic and coupled) model without changing the interface or controller. The MVC pattern can open up new levels of robustness, code reuse, and organization.

3.4 PROBLEM WITH SWING APPLICATION

In Applying MVC to swing applications, splitting the controller from the view didn't work well in practical terms because the view and controller parts of a component required a tight coupling (for example, it was very difficult to write a generic controller that didn't know specifics about the view). So Sun Microsystems collapsed these two entities into a single UI (user-interface) object (Fowler, 2010). Figure 2 shows how swing MVC is structured in swing. In this Simulator, I have been able to provide logically separation at the application level amongst the model view and controller.

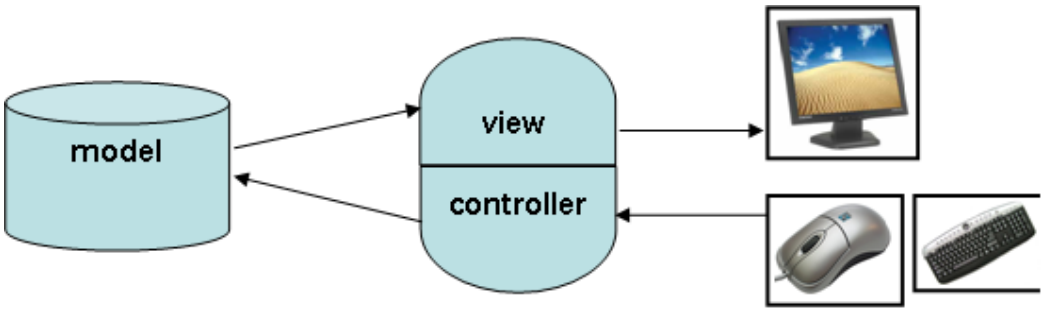

Figure 2 MVC of Swing Applications

4. DESIGN AND IMPLEMENTATION OF SIMULATOR

4.1 PROBLEM PARAMETER AND DESCRIPTION

The hoist system to be modeled has the following characteristics:

- ✓ it has a single hoist
- ✓ it has multi -tank(multi -stage)
- ✓ No-wait or the no-wait-in-process constraints, this means that the products must be taken to the next tank immediately it is removed from a tank, without waiting.
- ✓ multi-carrier(multi-barrel)
- ✓ multi-product, different products can be processed in the in the system
- ✓ all the product in a carrier are identical and each product type has require the same processing sequence
- ✓ processing time in a tank is within upper and lower bound(time window constraint)
- ✓ single line
- ✓ Multifunction tanks.
- ✓ There is no storage for the carriers near the facility or at the load/unload stations. Then empty carriers must remain on the line and be moved from tank to tank so as to prevent them from interfering with loaded carriers.

The hoist can carry only one carrier at a time. Each tank can only take one carrier at a time, and each carrier can take many products to be treated. The first tank serves as the loading tank where products are loaded to the carrier. Empty carriers have to be taken to the first tank to be loaded. The last tank is the unloading tank. Full carriers must be taken to the last tank as the last stage of treatment of the products, for the carriers to be unloaded. It has both input and output buffer. The robot will travel to the specified tank/location, wait if necessary, lift the job, travel to the next tank on the route and then release the job. After that, the hoist is ready for the next scheduled material handling operation. The hoist

should have enough time to travel between the starts of successive operations, which is called the *traveling time* constraints.

4.2 MVC STRATEGY

The architectural design pattern adopted is the model-view-controller design. Figure 3 shows this pattern.

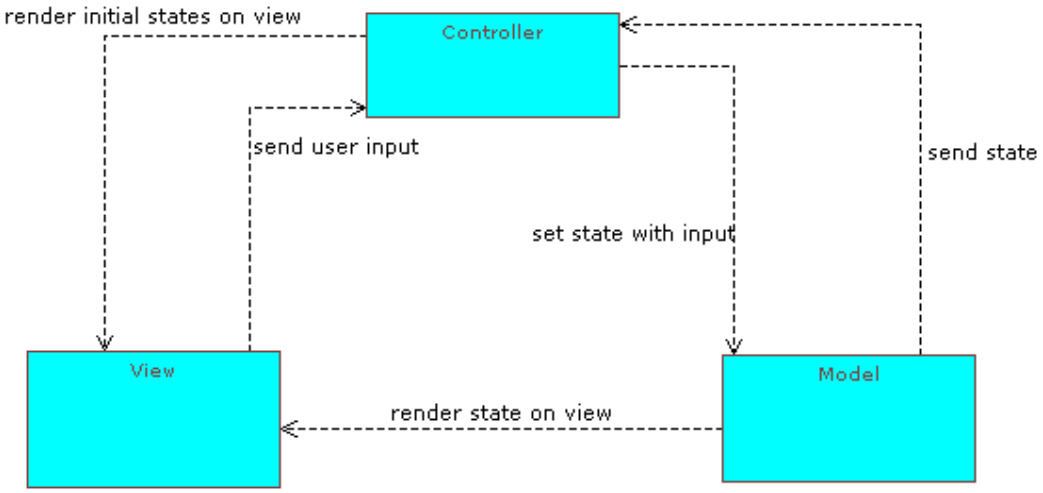

Figure 3 MVC of Simulator

The MVC design pattern strategy has enabled me to separate the problem into three separate problems, the view, the model and the controller. This has enabled me to focus on one subsystem at a time, there by avoiding the confusions involved in handling the whole system as one. Much effort on MVC is directed towards web applications, and not much is done on swing standalone application. This thesis focused on how to use MVC in swing applications, and devising techniques on how to achieve this. My aim is to make a clear separation between Model, view and controller.

Adoption of MVC has forced me to isolate the model, view, and controller components, it really did make me rethink my application and

put a lot of thought into the architecture. If I can handle the extra up-front work and the additional complexity, the MVC pattern can open up new levels of robustness, code reuse, and organization.

4.2.1 Use-Case Diagram

The use-cases triggered by the primary actor, user, and the secondary actor, simulator, is shown in figure 4

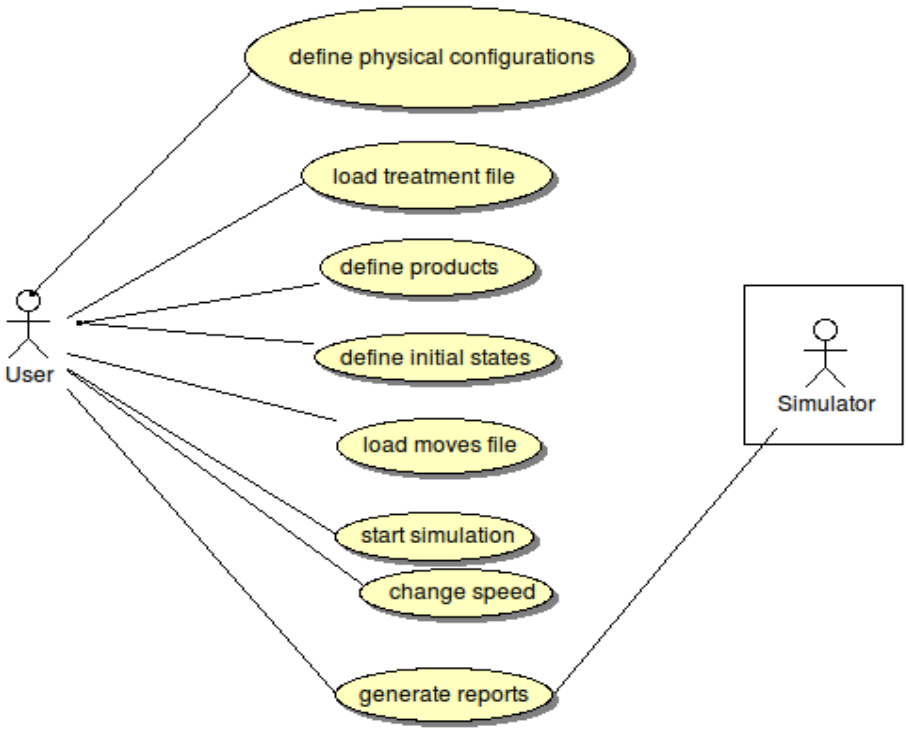

Figure 4 Use Case Diagram of System

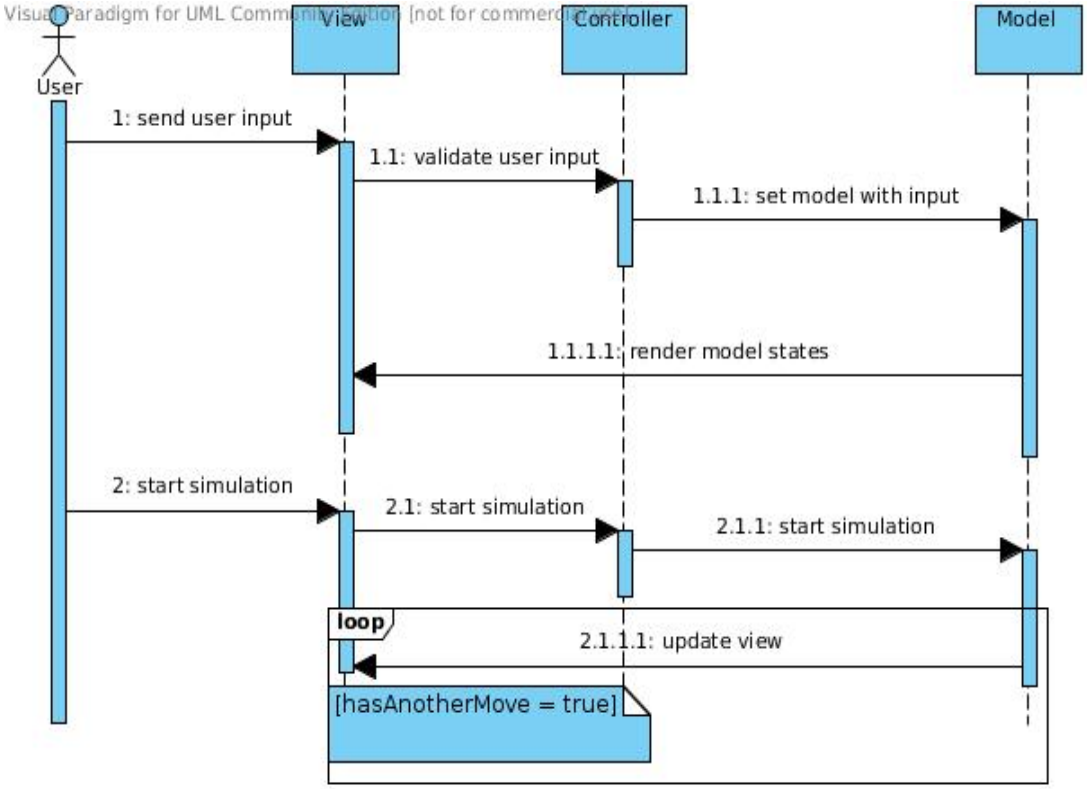

Figure 5 High-Level Sequence Diagram of Simulator

4.2.2 Sequence Diagram

Figure 5, shows a high level sequence diagram of the hoist system simulator. The system is made up of three subsystems, the View, the Controller and the Model. The user can only interact with the system from the view which is the user interface as we have said before. The view takes user inputs and then passes it on to the controller which validates the inputs and then passes it on to the model, thus changing the state of the model. The model responds by passing its states to the controller, which then passes it on to the view to be rendered. When simulation starts, the model continuously changes its states and continuously updates the view.

4.3 THE VIEW

The view renders the contents of the model. It specifies exactly how the model data should be presented. If the model data changes, the view must update its presentation as needed. This is achieved by the model calling on the view to change. The setter methods in the controllers which set the reference of the GUI components from the view in the controllers is called in the view. The methods that execute user actions in the controller are also called from the event listener of the GUI components in the view.

4.3.1 UML Class Diagram for the View

UML class diagram was use to model the view of the simulator. The Class diagram for the view is shown in Figure 6. Some of the classes in the view subsystem are shown below.

Canvas:

This class extends the java swing's JFrame and implements the interface, Graphical and it forms the main window of the simulator. From this main window, other child windows of the system can be launched.

HoistDisplay:

It shows a visual display of the hoist on the Canvas. This class is a subclass of the JLabel and also implements the interface Graphical which has the draw method which draws this object if there is a change in state of the hoist from the model. The hoist icon is rendered in this JLabel.

TankDisplay:

This shows a visual display of the tank on the Canvas. This class is a subclass of the JLabel and also implements the interface Graphical which has the draw method which draws this object if there is a change in state of the tank from the model. The tank icons are rendered in this JLabel.

TreatmentDialog, PhysicalConfigDialog, HoistMovesDialog, InitialLoadDialog and InitialStatesDialog, all extends the JDialog class, and they are responsible for taking the user input. They form the child windows of the Canvas.

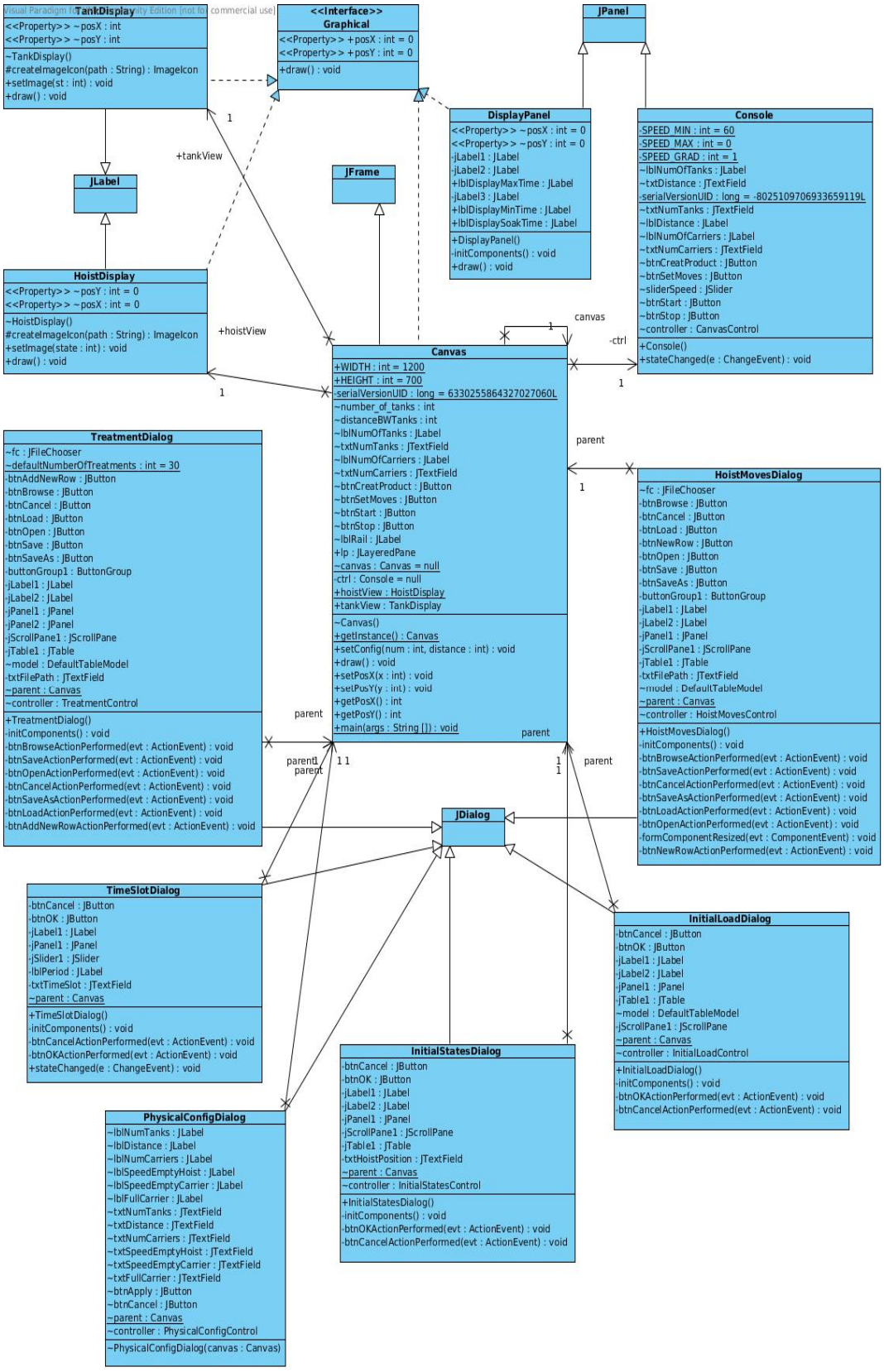

Figure 6 Class Diagram of View

4.4 THE CONTROLLER

We intend to have a controller class whose job is to contain interactions of the user interface components and to interact with the model. The class does not inherit from any swing class. The controller will keep references to user interface components that will be updated or modified in some way and also provide a set of methods that other components can directly call in their event handler. In my opinion, only one instance of the controller class is ever needed and the controller made a singleton class. Some setter classes also need to initialize the references to the interface elements that the controller need to interact with. The controller also has other methods that execute user actions and subsequently send message to the model. It also has methods that send message to the view.

The controller classes serve as a bridge between the model and the view. Figure 7 shows the controller classes of the hoist system simulator. For instance, the TreatmentControl class serves as bridge between the HoistSystem class in the model with the TreatmentDialog class in the view.

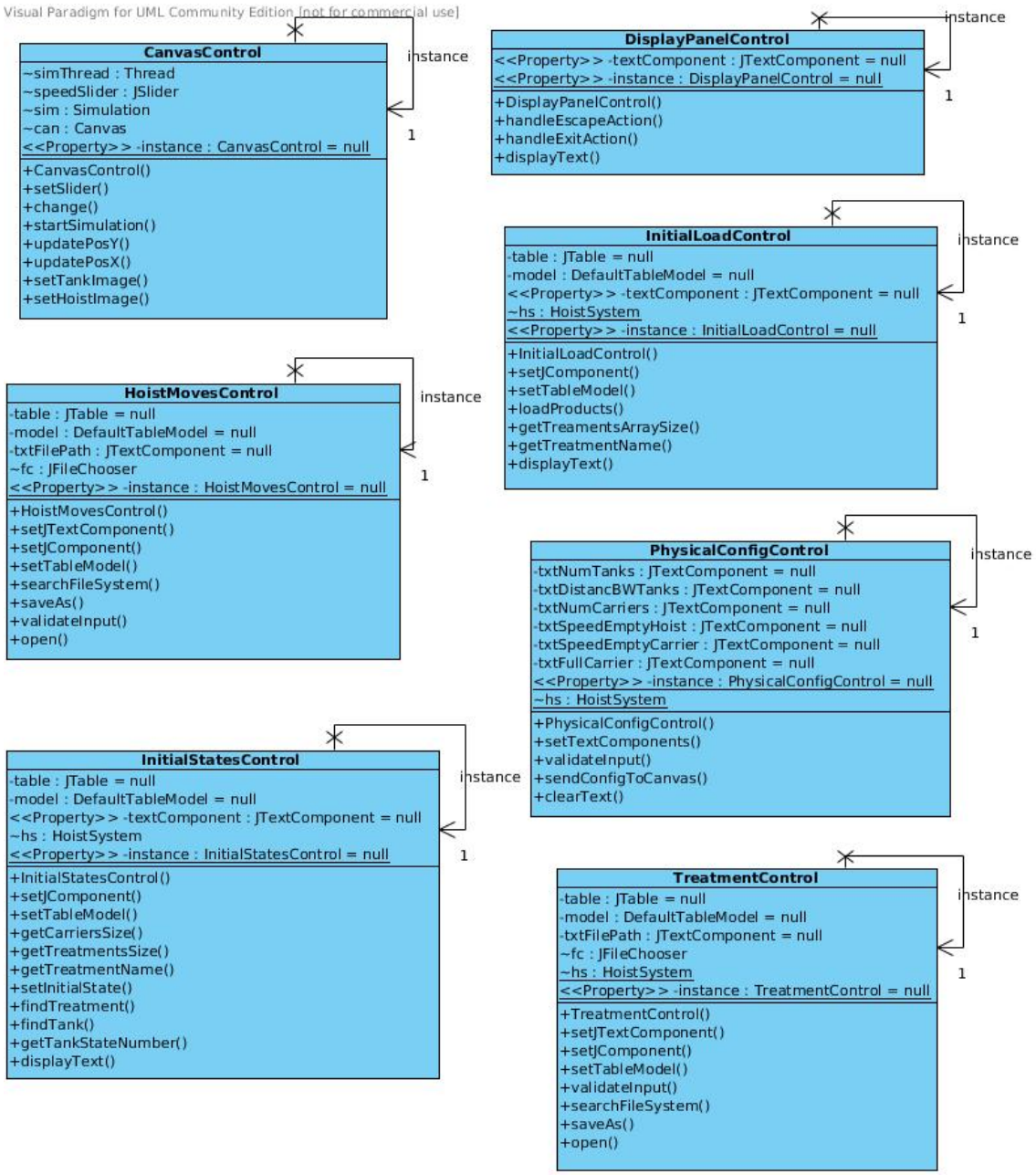

Figure 7 Class Diagram for the Controllers

4.5 THE MODEL

The model represents data and the rule that govern the access to and update to this data. The model stores internal states that affect the drawing of the view. The model represents the real world objects of the hoist system, based on how the hoist system looks and function. I was able to decompose the system into many classes that naturally represented the physical component of a hoist system. The model is the work horse of the system as it contains the logic of the simulator. The class diagram of the model is shown in figure 8. Some of the classes are listed below:

HoistSystem:

This serves as the class where variables, arrays and method that are common to the whole system in defined. The model can only be accessed from outside of it through this class. This class stores the states that are common to the whole hoist system.

Hoist:

This is the class where variables and methods representing the properties and actions of the hoist are defined.

Tank:

This represents the properties and actions of the tank or nodes of the system.

Carrier:

The carrier represents the actions and properties of the container which is moved by the hoist. The container holds the products.

Product:

The class product has the properties and actions of the product that is being treated.

HoistState:

This is an enumeration type which has the states that a hoist can be. I defined three states that the hoist can assume. The states are:

- ✓ EMPTY HOIST, this is the state when the hoist is not carrying anything.
- ✓ EMPTY CARRIER, this is the state when the hoist is carrying and empty carrier.
- ✓ FULL CARRIER, this is the state when the hoist is carrying a carrier with products in it.

TankState:

This is an enumeration type which has the states that a tank can be. The state defined for the tank is:

- ✓ EMPTY TANK, this is when the tank has no carrier in it.
- ✓ EMPTY CARRIER, the tank has an empty carrier in it.
- ✓ FULL CARRIER, the tank has a carrier with products in it.

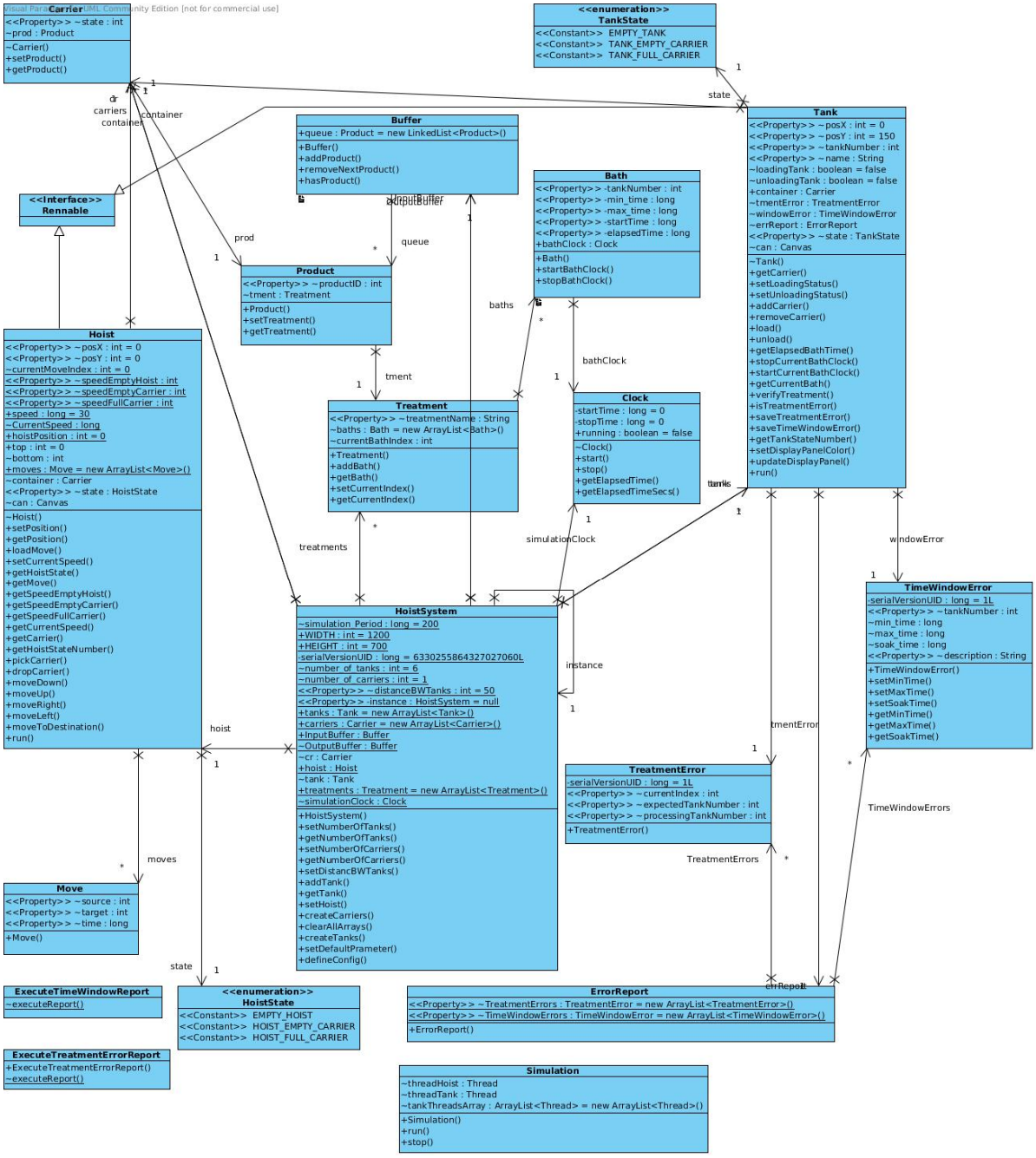

Figure 8 Class Diagram for the Model

4.6 MVC DESIGN

Figure 9, shows the relationships amongst classes across subsystems. This helps us to understand better how the subsystems are related to one

another. Only classes that have relationships with classes in another subsystem appeared in this diagram.

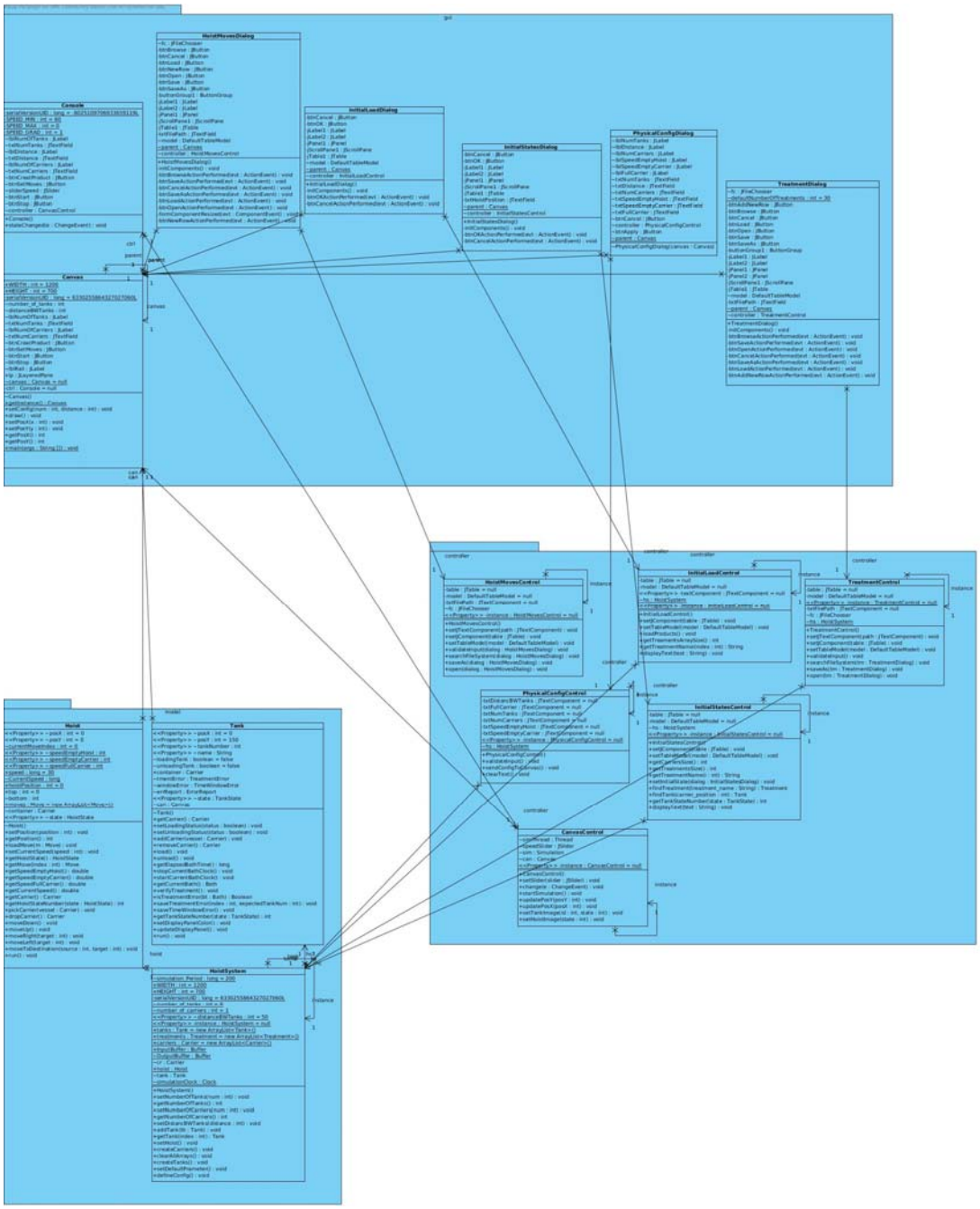

Figure 9 Class Diagram showing MVC Relationships

4.7 USER INTERFACE

The main window: This is the interface which has the canvas on which the simulation takes place. On the canvas, the track, hoist, tanks and carriers are displayed. It has menus from where other child windows could be launched. A screen shot of this window is shown in figure 10. Figure 11 also shows the main window with two carriers with one of the carriers with a product being processed.

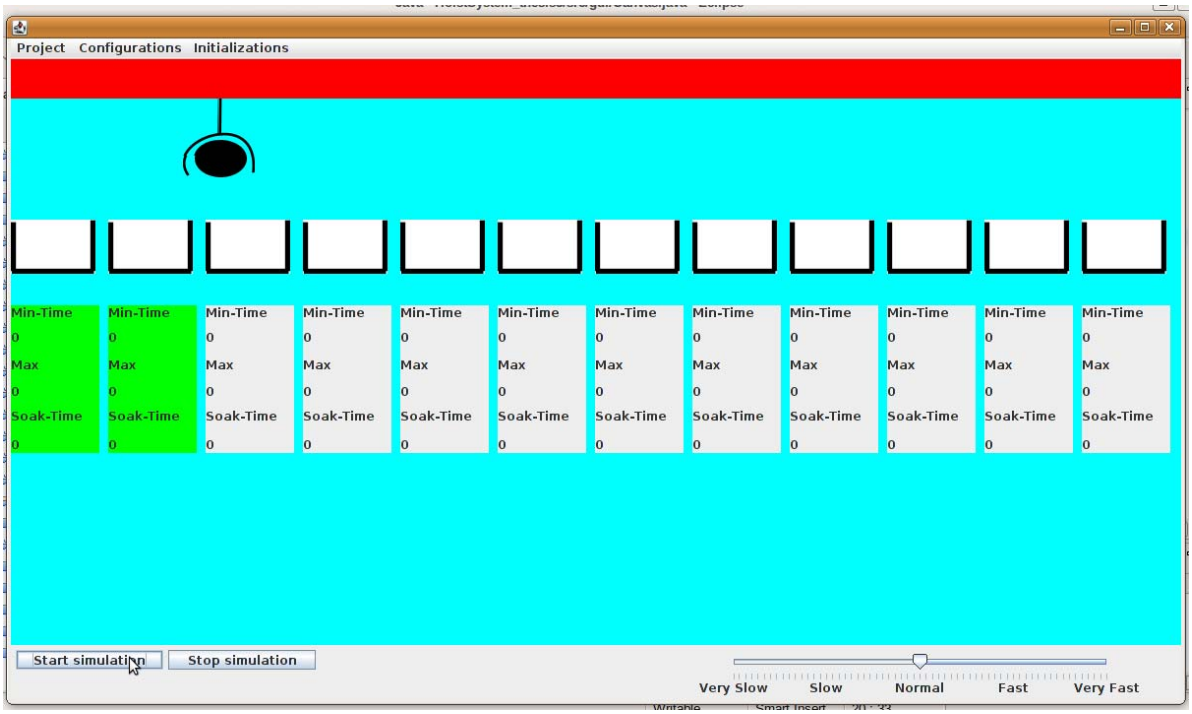

Figure 10 Screen shot for the Main Window

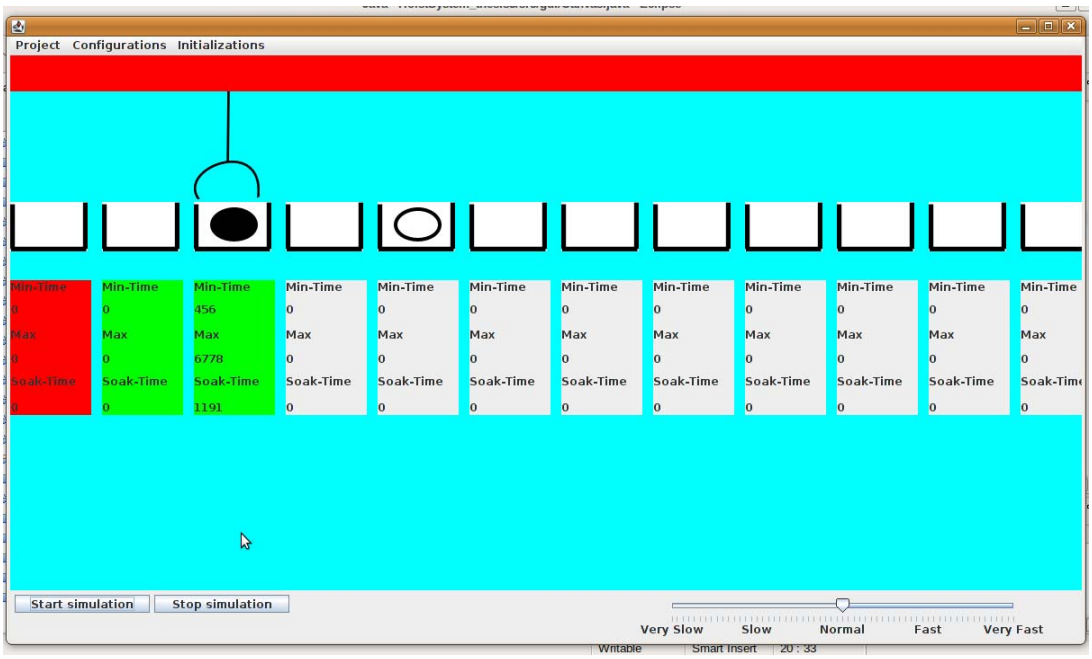

Figure 11 Screen shot of Main window with two carriers

Physical configurations: this is the interface where physical configurations of the hoist system are defined. Physical parameters such as the number of tanks needed in system, the distance between the tanks, and the number of carriers in the system and the speed of the hoist are also defined in this window. The screen shot for this window is shown in figure 12

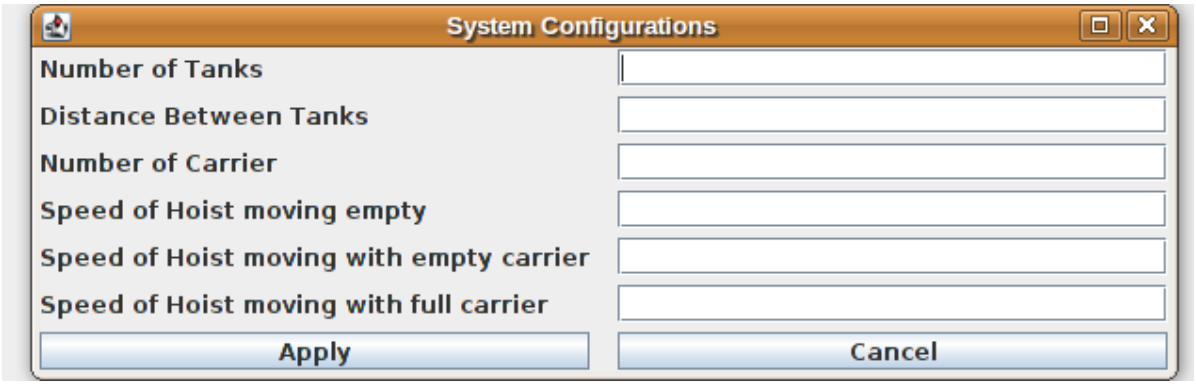

Figure 12 Screen shot for physical configuration

Treatment window: this is the window for defining and loading treatments. A treatment is the sequence of processing that a product will undergo. A treatment has the information on time window constraint (i.e. the minimum and maximum time that a product can be soaked in a tank) on each tank that a product will be processed. Treatments can be defined on the grid provided or loaded from a text file. These text files contain the information about treatment that a product will undergo in a production line. A screen shot of this window is shown in figure 13

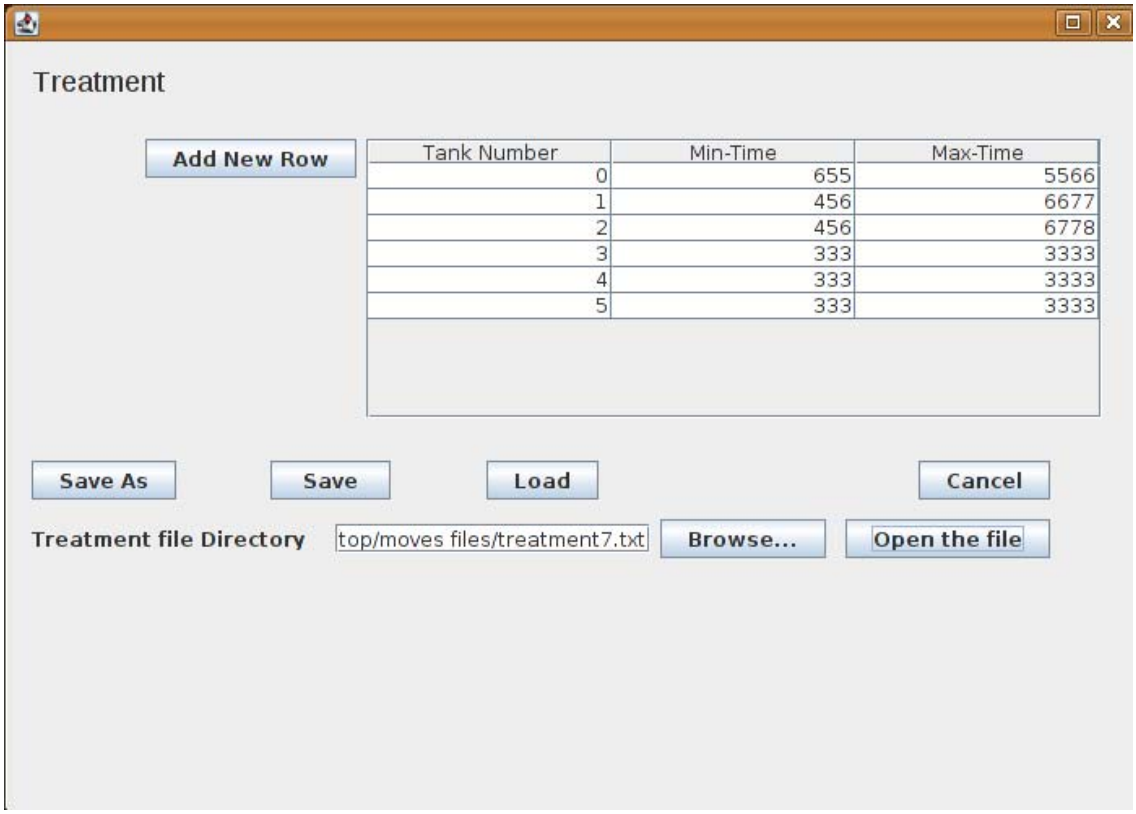

Figure 13 Screen Shot for treatment window

Initial load: this is the window where the number of products undergoing a particular treatment is defined and also the products are loaded to the input buffer from this window. The screen shot for this window is shown in figure 14

Initial States: this is the window where the initial states of the hoist and tanks are defined before simulation starts. The initial position of the hoist and carriers are defined. Initial treatments, treatment levels and also elapsed time on a treatment level are also assigned to the carrier here. The screen shot for the initial states window is shown in figure 15

Hoist Moves: this is the window where hoist moves are defined or loaded from a text file. The text file contains information on the movements of a hoist. The moves file has columns for source, target (destination) and pick time. A row in the moves file is a move, which has the tank number of the source tank, the number of destination tank, and the time the carrier will be picked from the source. The moves provided by this file determine how the movement of the hoist is going to be like. The source and destination (target) are tank numbers to be visited by the hoist. The hoist picks a carrier from a source and drops it at the target. The screen shot for hoist moves window is shown in figure 16

I am proposing that moves computed from hoist scheduling algorithm should be used as input to this simulator, so as to make the evaluation of the algorithm easy and less expensive.

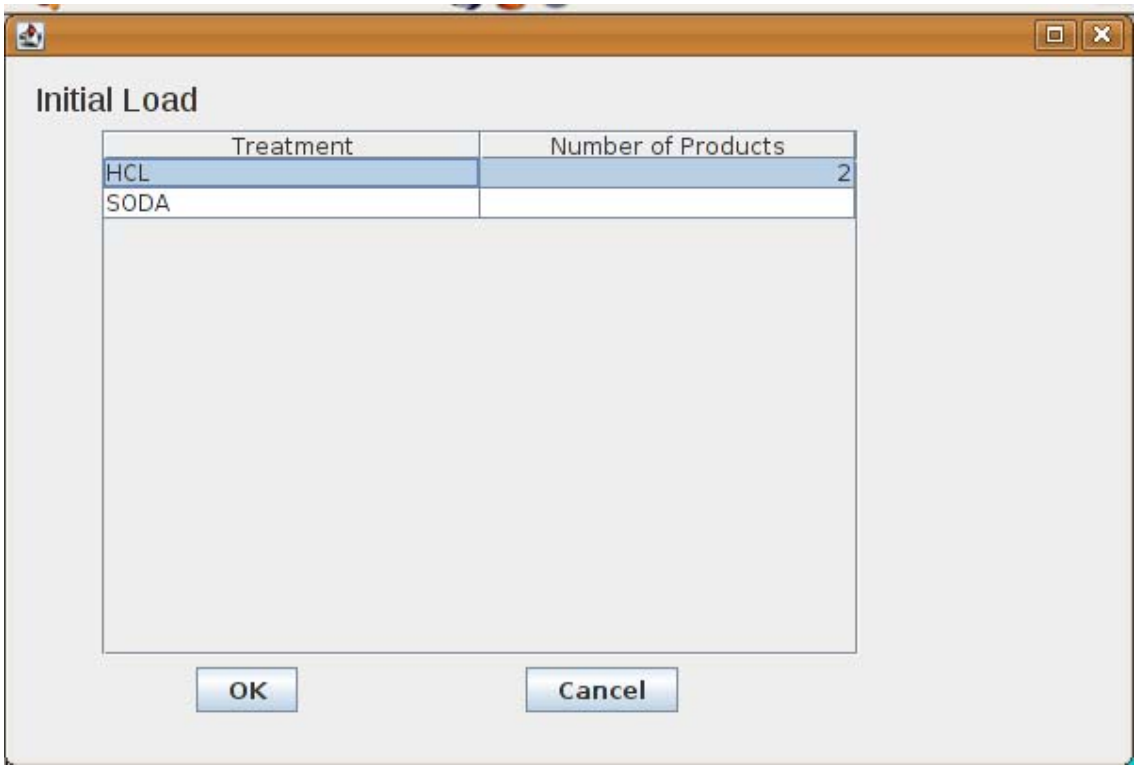

Figure 14 Screen shot for Initial Load Window

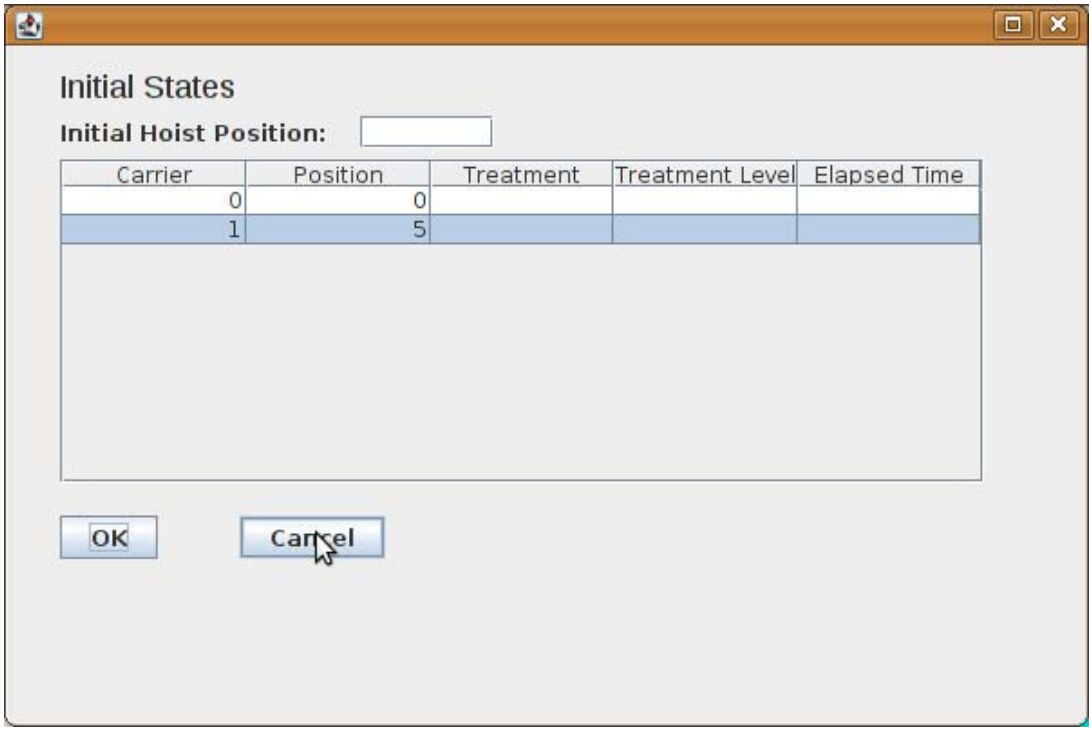

Figure 15 Screen shot for Initial States window

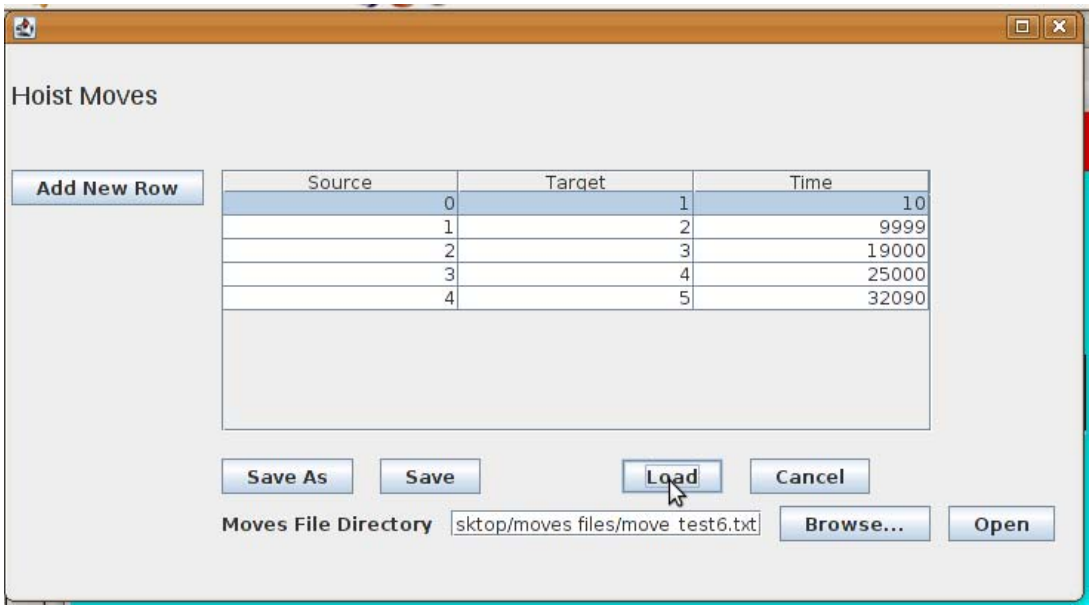

Figure 16 Screen Shot for Hoist Moves Window

4.8 INPUTS AND OUTPUTS OF THE SIMULATOR

4.8.1 Inputs to the System

The inputs to the system are text files. The treatments, the hoist moves are two important inputs that come in a text file format. An example text file for treatment is shown in the figure 17 below. The file is made up of three row, the first columns is the tank number, which represents the tanks where the baths are supposed to take place. The second column is the minimum bath time, which is the minimum time that a bath is supposed to take place. The third column is the maximum bath time, which is the maximum bath time for the product in the tank identified by the tank number. The minimum and maximum bath times are time window constraints imposed on the system. The time is specified in

milliseconds. The comma (,) character is use as a delimiter to separate data items in a row in the file.

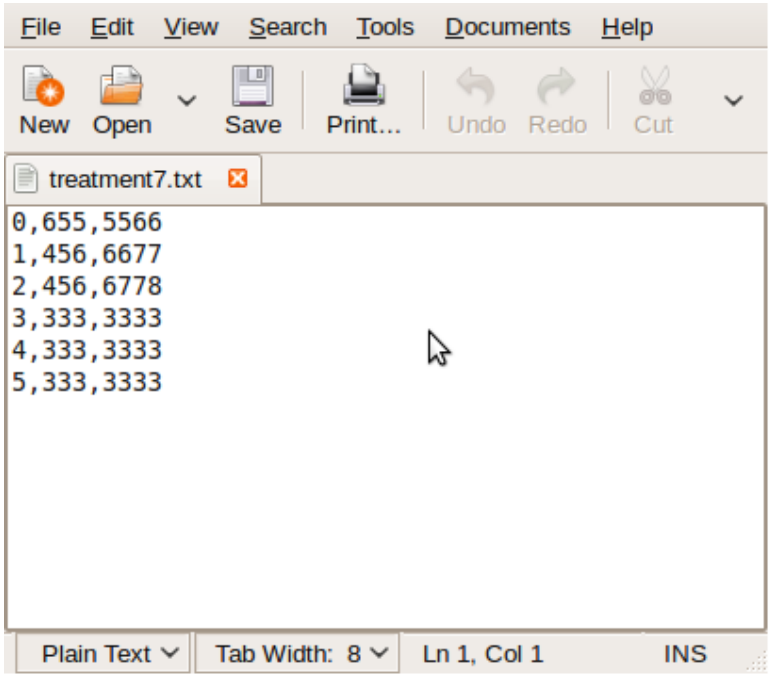

Figure 17 Treatment Text File

The second input to be explained is the hoist moves text file. An example text file of the hoist moves is shown in figure 18. This file is made up of three columns, the first is the source, the second is the target (destination) and the last is the pick time. A row in the file represents a move. A hoist move takes place when a carrier is moved from a source tank to a target tank. The source column in the file is represented by the tank number of the source tank while the target is represented by the tank number of the target tank. The third column represents the pick time, which is the time that he hoist will pick the carrier from the source tank.

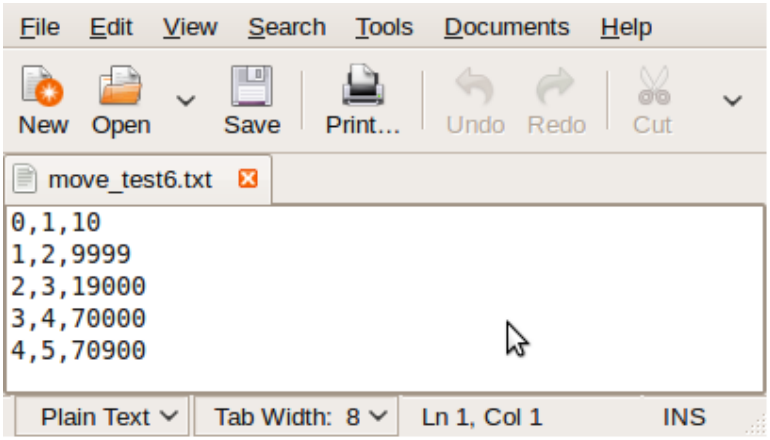

Figure 18 hoist moves text file

4.8.2 Outputs to the System

The output of the simulator can be seen in the panel that displays under each tank and also the final report that is generated after the simulation has concluded. The panel that appears under each tank shows for any bath taking place in that tank, the minimum bath time, the maximum bath time and also the soak time. If the soak time is below the minimum time, the colour is orange, if it is within the time window, the colour is green, and if it is above maximum time, the colour is red. If the product is being soaked in the wrong tank, the colour becomes black. This visually gives the user an instant report on the situation of the bath in the tank. This display panel is shown in figure 19.

Figure 19 Bath Report

A second set of output of this simulator is the good bath report, time window report and the treatment report that is generated after a simulation has concluded. With these reports, evaluations can be made about the simulation run. The table below shows a time window violation report from a simulation, which gives a detailed report on the time window violations during the simulation. This is very important, because it is a measure of how accurate the moves supplied to the simulator is, and consequently how accurate the algorithm that generated the moves is. The total number of unsuccessful baths is a very important parameter in evaluating moves and also the algorithms that generated those moves.

Time Window Violation

Tank Number	Product Name	Minimum Time(milliseconds)	Maximum Time(milliseconds)	Soak Time	Description
0	T2-1	454	7345	7795	soak time is above
2	T1-1	456	7778	7878	soak time is above
2	T2-1	234	6778	7789	soak time is above
5	T2-1	334	7456	7770	soak time is above
8	T2-1	341	7229	7606	soak time is above

Number of Violations: 5

Another important report is the treatment error report which shows the baths that are not being done in the right tank as specified by the treatment. This means the processing sequence specified by the treatment in the product is not strictly followed. The table below shows a treatment error report. It shows that one of the baths was not done in the right tank. The total number of bad treatments is a very important parameter use in comparing moves, and consequently algorithms which generated these moves. Another very important report is the good bath report, which shows all baths that were successful in a simulation run. This a very important report, which also shows the total number of good baths, which is a very important parameter used in measuring the performance of a hoist scheduling algorithm.

TREATMENT ERROR

Current Index	Product Name	Expected Tank Number	Processing Tank Number
3	T1-1	5	3
9	T1-1	3	4

Treatment Error Count: 2

4.9 IMPLEMENTATION

The implementation is Java based, adopting the Model View Controller (MVC) architectural design pattern. The simulator is made up of three packages, the gui, which serves as the view and the model, which is the model and the controller which is the controller. Two libraries (Application programming interface (API's)) that were use amongst others are swing and Business Intelligence and Reporting Tools (BIRT).

The Swing classes (part of the Java™ Foundation Classes software) implement a set of components for building graphical user interfaces (GUIs) and adding rich graphics functionality and interactivity to Java applications. The GUI components used for developing the simulator are part of this swing API. Graphical components ranging from buttons, tables, text fields etc. are all from swing.

The Business Intelligence and Reporting Tools (BIRT) API is an open source API that provides reporting and business intelligence capabilities for rich client and web applications, especially those based on Java and Java EE. BIRT is a top level API developed and maintained by Eclipse Foundation, an independent not-for-profit consortium of software industry vendors and an open source community. BIRT was used to implement the time window violation report and the treatment error report.

The eclipse and netbeans Integrated Development Environment (IDE) where use as the development environments. The GUI of some of the child windows were developed under netbeans, while the other parts of the software were developed on the eclipse IDE.

4.9.1 Thread Implementation

It is not possible to implement a simulator of this sort without utilizing Java's multi-threading capabilities. It enabled many tasks to be carried out concurrently. The java code in Listing 1 shows a class, Simulation. Simulation implements the Runnable interface. In the run method of this class, the threads for hoist and each tank is started.

Listing 1: thread implementation in Simulation class

```
package model;

public class Simulation implements Runnable {
	Thread threadHoist;
	Thread threadTank;
	public Simulation() {
		// TODO Auto-generated constructor stub
	}
	@Override
	public void run() {
		HoistSystem.simulationClock.start();
		//start thread for each tank
```

```
                for(int i = 0; i < HoistSystem.tanks.size(); i++){
                        threadTank = new Thread(HoistSystem.tanks.get(i));
                        threadTank.start();
                }
                //start thread for hoist
                threadHoist = new Thread(HoistSystem.hoist);
                threadHoist.start();
        }

}
```

The Java code in Listing 2 show the java class Hoist. This class has method and attributes that represent the action and properties respectively of a real hoist. It implements the Runnable interface. The execution of the moves in the moves arraylist of the Hoist is done in the run method. A for loop is used to run through the moves arraylist, so as to execute all move object in the arraylist. Since this can take a very long time, thus it is implemented by making it run in a separate thread, so that it will not take up the processor time completely for the period of its execution.

Listing 2: thread implementation in Hoist class

```
package model;

import gui.Canvas;
import gui.TankDisplay;
import java.util.ArrayList;
import org.eclipse.birt.report.engine.api.EngineException;


public class Hoist implements Runnable{
        public static ArrayList<Move> moves = new ArrayList<Move>();
        int posX =0;
        int posY =0;

        public void moveDown(){
                while(true){
                        posY = posY + 1;
                        can.hoistView.setPosY(posY);
                        if(posY == bottom){
                                break;
                        }
                        try {
                                Thread.sleep(speed);
                        } catch (InterruptedException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                        }
```

```
		}
	}

	public void moveUp(){
		while(true){
			posY = posY - 1;
			can.hoistView.setPosY(posY);
			if(posY == top){
				break;
			}
			try {
				Thread.sleep(speed);
			} catch (InterruptedException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			}
		}
	}

	public void moveRight(int target){
			TankDisplay tankGUI = (TankDisplay) can.lp.getComponent(target);
				can.hoistView.setPosX(tankGUI.getPosX());
	}

	public void moveLeft(int target){
			TankDisplay tankGUI = (TankDisplay) can.lp.getComponent(target);

				can.hoistView.setPosX(tankGUI.getPosX());
	}

	public void moveToDestination(int source, int target){
		//find direction to go
		if(target < source){
			if(getPosY() > top){ //if hoist is bellow top
				moveUp();
			}
			moveLeft(target);
			hoistPosition = target;
			moveDown();
		}else if(target > source){
			if(getPosY() > top){ //if hoist is bellow top
				moveUp();
			}
			moveRight(target);
			hoistPosition = target;
			moveDown();
		}else if(target == source){
			if(getPosY() < bottom){
				moveDown();
			}
		}
	}

	@Override
	public void run() {
		int source;
		int target;
		long pick_time;
		System.out.println("simulation time " +
HoistSystem.simulationClock.getElapsedTime());
```

```
//loops through all the moves in the moves arraylist
                    for(int i=0; i < moves.size(); i++){
                                Move move = moves.get(i);
                                source = move.getSource();
                                target = move.getTarget();
                                pick_time = move.getTime();
                                moveToDestination(hoistPosition, source);
                                System.out.println("this move is source " + source + " target " + target + "
pick time " + pick_time);
                                while(true){
                                            if(HoistSystem.simulationClock.getElapsedTime() >= pick_time){


          pickCarrier(HoistSystem.tanks.get(source).removeCarrier());
                                                        break;
                                            }
                                }

                                moveToDestination(source, target);
                                if(container != null){
                                            HoistSystem.tanks.get(target).addCarrier(dropCarrier());
                                }
                    }
                    try {
                                ExecuteTimeWindowReport.executeReport();
                                ExecuteGoodBathReport.executeReport();
                                ExecuteTreatmentErrorReport.executeReport();
                    } catch (EngineException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                    }
          }//end of run method
}
```

The Java code in listing 3, show how thread is implemented in a Tank. The run method in this class is responsible for loading a carrier with products from the input buffer immediately the carrier is dropped in the first (loading) tank. Another thread is also implementation in the setDisplayPanelColor() method. This thread is started in the inner class in this method when a carrier with a product is dropped in a tank for processing. This thread is responsible for changing the color of the display panel, and also continuously update the soak time as the product processes in the tank.

Listing 3: thread implementation in Tank class

```
package model;

import java.awt.Color;
import javax.swing.JOptionPane;
import gui.Canvas;
import gui.DisplayPanel;
import gui.TankDisplay;

public class Tank implements Runnable{

	Tank(){
		errReport = new ErrorReport();
		can = Canvas.getInstance();
	}


	public void load(){
		container.setProduct(HoistSystem.InputBuffer.removeNextProduct());
		container.getProduct().getTreatment().setCurrentIndex(0);
		startCurrentBathClock();
		setState(TankState.TANK_FULL_CARRIER);
		int stateNum = getTankStateNumber(getState());
		TankDisplay tankGUI = (TankDisplay) can.lp.getComponent(tankNumber);
		tankGUI.setImage(stateNum);
		setDisplayPanelColor();
		System.out.println("the current bath index in loading method " +
container.getProduct().getTreatment().getCurrentIndex());
		verifyTreatment();
	}

	public void setDisplayPanelColor(){
		new Thread(new Runnable() {
			public void run() {

				Bath bt = getCurrentBath();
				HoistSystem hs = HoistSystem.getInstance();
				DisplayPanel panel = (DisplayPanel)
can.lp.getComponent(tankNumber + hs.number_of_tanks);
				panel.setVisible(true);
				panel.setBackground(Color.ORANGE);
				panel.setOpaque(true);


	panel.lblDisplayMinTime.setText(String.valueOf(bt.getMin_time()));

	panel.lblDisplayMaxTime.setText(String.valueOf(bt.getMax_time()));

	panel.lblDisplaySoakTime.setText(String.valueOf(bt.getElapsedTime()));
				//panel.repaint();
				panel.draw();


				while(true){
					long lastElapsedTime = bt.getElapsedTime();
					if(bt.getTankNumber() == tankNumber){
						if(bt.getElapsedTime() > bt.getMin_time() &&
bt.getElapsedTime() < bt.getMax_time()){
							panel.setBackground(Color.GREEN);
```

```
                                        }else if(bt.getElapsedTime() >
bt.getMax_time()){
                                                panel.setBackground(Color.RED);
                                        }
                                }else{
                                        panel.setBackground(Color.BLACK);
                                }


        panel.lblDisplaySoakTime.setText(String.valueOf(bt.getElapsedTime()));
                                System.out.println("the elapsed time in " + tankNumber +
" is " + bt.getElapsedTime());
                                //panel.repaint();
                                panel.draw();


                                if(bt.bathClock.running == false){
                                        panel.setBackground(Color.CYAN);
                                        panel.setVisible(false);
                                        break;
                                }

                        }
                        panel.lblDisplayMinTime.setText(String.valueOf(0));
                        panel.lblDisplayMaxTime.setText(String.valueOf(0));
                        panel.lblDisplaySoakTime.setText(String.valueOf(0));
                        //panel.repaint();
                        panel.draw();
                }
        }).start();
    }
    @Override
    public void run() {
        if(loadingTank == true){
            while(true){
                if(container != null){
                    if(container.getProduct() == null){
                        boolean anyProduct =
    HoistSystem.InputBuffer.hasProduct();
                        if(anyProduct == true){
                            if(container.getProduct() == null){
                                load();
                            }


                        }
                    }
                }
            }
        }
    }

}
```

4.9.1 MVC Implementation

Implementation of the Controller

The controller classes contain the interaction of the user interface components and also method that send message from the model to the view. The controller keeps track of references to user interface components and also provides methods that the components can directly call in their event handler. Since only one instance of the controller is ever needed in this simulator, the singleton design pattern is adopted. A static getInstance() method is use to get the one instance of the controller from any part of he application. The java code in Listing 4 shows the InitialStatesController class. Setter methods like setJComponent(), setTableModel() are used to initialize the reference(s) to the GUI element(s) that the controller needs to interact with. Some methods need to be present that send message to the model, such as setInitialState(InitialStatesDialog) and others that sent message to the view. All the other controllers follow the same design as explained above.

Listing 4: InitialStatesController class

```
package controller;

import gui.Canvas;
import gui.InitialStatesDialog;
import gui.TankDisplay;
import javax.swing.JOptionPane;
import javax.swing.JTable;
import javax.swing.table.DefaultTableModel;
import javax.swing.text.JTextComponent;
import model.Bath;
import model.Carrier;
import model.HoistSystem;
import model.Product;
import model.Tank;
import model.TankState;
import model.Treatment;

public class InitialStatesControl {

        /* ===== singleton ===== */

        private static  InitialStatesControl instance = null;
```

```
/* ===== controller instance fields===== */

	/* GUI components we want to call methods on ... */
	private JTable table = null;
	private DefaultTableModel model = null;
	private JTextComponent textComponent = null;
	HoistSystem hs;
	public  InitialStatesControl() {
		// TODO Auto-generated constructor stub
		hs = HoistSystem.getInstance();
	}

	public static  InitialStatesControl getInstance() {
		if (instance==null) {
			instance = new  InitialStatesControl();
		}
		return instance;
	}

	public void setJComponent(JTable table) {
		this.table = table;
	}
	public void setTableModel(DefaultTableModel model){
		this.model = model;
	}
	public void setTextComponent(JTextComponent textComponent) {
		this.textComponent = textComponent;
	}
	public int getCarriersSize(){
		return hs.carriers.size();
	}
	public int getTreatmentsSize(){
		return hs.treatments.size();
	}
	public String getTreatmentName(int i){
		return hs.treatments.get(i).getTreatmentName();
	}
	/* actions we want to provide to GUI event handlers ... */

	public void setInitialState(InitialStatesDialog dialog) {
		Canvas can = Canvas.getInstance();
		int posX =0;
	int hposition = 0;

	try{
		hposition = Integer.parseInt(textComponent.getText());
	}catch(NumberFormatException e){}
	hs.hoist.hoistPosition = hposition;
	for(int i=0;i<hs.tanks.size();i++){
		if(hposition == i){
			posX = (hposition * hs.getDistanceBWTanks()) + (hposition * 90);
			hs.hoist.setPosX(posX);
			hs.hoist.setPosY(hs.hoist.top);

			can.hoistView.setPosX(hs.hoist.getPosX());
			can.hoistView.setPosY(hs.hoist.getPosY());
			can.hoistView.draw();
			//sys.hoist.setImage(HoistState.HOIST_EMPTY_CARRIER);
			break;
```

```
                }
            }
            int carrier_position = 0;
            Carrier cr;
            String treatment_name = null;
            int treatment_level = 0;
            long elapsed_time = 0;
            for(int i=0; i<HoistSystem.tanks.size();i++){
                hs.tanks.get(i).container = null;
                hs.tanks.get(i).setState(TankState.EMPTY_TANK);

                int state = getTankStateNumber(hs.tanks.get(i).getState());
                TankDisplay tankGUI = (TankDisplay) can.lp.getComponent(i);
                tankGUI.setImage(state);
            }
            end:
            for(int row_index =0; row_index<table.getRowCount(); row_index ++){
            for(int col_index=0; col_index<table.getColumnCount(); col_index ++){
                Object cellVar = table.getModel().getValueAt(row_index, col_index);
                if(col_index == 0){
                    int carrier_no = (Integer) cellVar;
                }else if(col_index == 1){
                    if(cellVar == null){
                        JOptionPane.showMessageDialog(dialog, "position can not be
null.");
                        break end;
                    }
                    carrier_position = (Integer) cellVar;
                      //double d = Double.valueOf(str).doubleValue();
                }else if(col_index == 2){
                    if(cellVar == null){
                        cr = HoistSystem.carriers.get(row_index);

                        Tank tk = findTank(carrier_position);
                        tk.addCarrier(cr);
                        tk.container.setProduct(null);
                        continue end;
                    }
                     treatment_name = cellVar.toString();
                     //double d = Double.valueOf(str).doubleValue();
                }else if(col_index == 3){
                    if(cellVar == null){
                        JOptionPane.showMessageDialog(dialog, "treatment level for
applied treatment can not be null.");
                        break end;
                    }
                    treatment_level = (Integer) cellVar;
                }else{
                    if(cellVar == null){
                        JOptionPane.showMessageDialog(dialog, "elapsed time for applied
treatment can not be null.");
                        break end;
                    }
                    String strTime = cellVar.toString();
                    elapsed_time = Long.valueOf(strTime).longValue();
                }
            }
            //
            cr = HoistSystem.carriers.get(row_index);
            Product pd = new Product();
```

```
                        pd.setProductID(0);
                        Treatment tmt = findTreatment(treatment_name);
                        tmt.setCurrentIndex(treatment_level);
                        Bath bath = tmt.getBath(treatment_level);
                        bath.setElapsedTime(elapsed_time);
                        pd.setTreatment(tmt);
                        cr.setProduct(pd);
                        Tank tk = findTank(carrier_position);
                        tk.addCarrier(cr);
                        //TankState state = TankState.TANK_FULL_CARRIER;
                        //tk.setImage(state);
    }
        }

        public Treatment findTreatment(String treatment_name){
                Treatment tt = new Treatment();
                for(int i=0;i<HoistSystem.treatments.size();i++){

if(HoistSystem.treatments.get(i).getTreatmentName().equals(treatment_name)){
                                tt = HoistSystem.treatments.get(i);
                        }
                }
                return tt;
        }

        public Tank findTank(int carrier_position){
                Tank tk = null;
                for(int i=0;i<HoistSystem.tanks.size();i++){
                        if( HoistSystem.tanks.get(i).getTankNumber() == carrier_position){
                                tk = HoistSystem.tanks.get(i);
                        }
                }
                return tk;
        }

        public int getTankStateNumber(TankState state){
                int num;
                if(state == TankState.EMPTY_TANK){
                        num = 0;
                }else if(state == TankState.TANK_EMPTY_CARRIER){
                        num = 1;
                }else{
                        num = 2;
                }
                return num;
        }
}
```

Implementation of the View

The view has the GUI components that the user interacts with. The code in listing 5, show the event listener for the ok button in the InitialStatesDialog class. This class is the interface from where users can

input parameters for initial states of the system before simulation starts. The setInitialStates() method of the InitilaStatesControl class is called to handle user input. All the GUI components with event listener, are implemented this way. The setter methods in the controller are called in the view to set the GUI component reference in the controller. This is shown in Listing 5.

Listing 5: event listener for the OK button for the InitialStatesDialog

```
//pass a reference of jTable1 to InitialStatesController class
controller.setJComponent(jTable1);

btnOK.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent evt) {
           btnOKActionPerformed(evt);
        }
    });

private void btnOKActionPerformed(java.awt.event.ActionEvent evt) {
          controller.setInitialState(this);
          dispose();
}//GEN-LAST:event_btnOKActionPerformed
```

Implementation of the Model

The model is use to model the real world hoist system objects. As the states of the model objects change, they have to notify the view to update. We are going to explain one of the hoist system objects, the hoist, which is represented in the model with the Hoist class. The hoist can pick a carrier, it can drop a carrier, and it can move up, move down, move right and also move left. Listing 6 shows how the hoist updates the view when its Y-positon changes. The update takes place in moveDown() method. Listing 7 shows how the hoist updates the view when the hoist drops a carrier.

Listing 6: a code snippet that shows how the Y-position is updated in view from the Hoist

```
public class Hoist implements Runnable{
	Canvas can;

	Hoist(){
		can = Canvas.getInstance();
	}
	public void moveDown(){
		while(true){
			posY = posY + 1; //change Y position
			can.hoistView.setPosY(posY);//update Y position in view
			if(posY == bottom){
				break;
			}
			try {
				Thread.sleep(speed);
			} catch (InterruptedException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			}
		}
	}
}
```

Listing 7: how hoist image is updated

```
	public Carrier dropCarrier(){
		Carrier vs;
		vs = container;
		container = null;
		setState(HoistState.EMPTY_HOIST);//change hoist state
		int stateNum = getHoistStateNumber(getState());
		can.hoistView.setImage(stateNum);//update the hoist image in view
		return vs;
	}
```

in both listing 6 and 7, hoistView is the JPanel that displays the hoist.

4.9.2 BIRT Report Implementation

Because we wanted to generate report from the simulation, we used the Helios version of Eclipse IDE for Java and Report Developers. This

version has a BIRT report designer, which make it easy to design a report.

First the different reports were designed using this report designer. To run the report, a birt-runtime-2_6_1 Report Engine is used. The report is generated as a pdf document.

To run the report in eclipse, we need to add the jar files in birt-runtime-2_6_1 in the Java build path of the project. First of all we have to download birt-runtime-2_6_1.zip from www.eclipse.org, and then unpack it in a directory in your machine. Then go to the directory where the .zip file was unpacked, and then go to /birt-runtime-2_6_1/ReportEngine/lib, then add all the jars found there. We then write a class to run the report. Listing 8 shows the class that executes the good bath report. In this class, we set the EngineConfig's birt home to the ReportEngine directory as show in Listing 8. We set EngineConfig's setLogConfig() method to a directory of our choice. We use IReportEngine's openReportDesign() method to open the report we designed earlier from the directory that it was save. We then use setOutputFileName() method of PDFRenderOption to specify the directory to output the pdf file.

Listing 8: execution of good bath report

```
package model;

import org.eclipse.birt.core.framework.Platform;
import org.eclipse.birt.report.engine.api.EngineConfig;
import org.eclipse.birt.report.engine.api.EngineException;
import org.eclipse.birt.report.engine.api.IReportEngine;
import org.eclipse.birt.report.engine.api.IReportEngineFactory;
import org.eclipse.birt.report.engine.api.IReportRunnable;
import org.eclipse.birt.report.engine.api.IRunAndRenderTask;
import org.eclipse.birt.report.engine.api.PDFRenderOption;

public class ExecuteGoodBathReport {

	public ExecuteGoodBathReport() {
		// TODO Auto-generated constructor stub
```

```
	}
	static void executeReport() throws EngineException{

		IReportEngine engine=null;
		EngineConfig config = null;

		try{
			config = new EngineConfig( );
			config.setBIRTHome("/home/celeonu/birt-runtime-2_6_1/ReportEngine");
			config.setLogConfig("/home/celeonu/BIRT-2_3_2/logs",
java.util.logging.Level.FINEST);
			Platform.startup( config );
			IReportEngineFactory factory = (IReportEngineFactory)
Platform.createFactoryObject(IReportEngineFactory.EXTENSION_REPORT_ENGINE_FACTORY);
			engine = factory.createReportEngine( config );

			IReportRunnable design = null;
			//Open the report design
			design =
engine.openReportDesign("/home/celeonu/workspace/HoistSystem_thesis4/src/reports/goodTimeWind
ow_report.rptdesign");
			IRunAndRenderTask task = engine.createRunAndRenderTask(design);

			task.setParameterValue("ordParam", (new Integer(10101)));
			task.validateParameters();

			PDFRenderOption options = new PDFRenderOption();
			//options.setOption( IPDFRenderOption.FIT_TO_PAGE, new Boolean(true)
);

	options.setOutputFileName("/home/celeonu/Desktop/good_bath_report.pdf");
			options.setOutputFormat("pdf");

			task.setRenderOption(options);
			task.run();
			task.close();
			engine.destroy();
		}catch( Exception ex){
			ex.printStackTrace();
		}
		finally
		{
		     Platform.shutdown( );
		}

		}

}
```

5. TESTING AND RESULTS

5.1 TESTING AND RESULTS

The total number of good baths within the time windows is a parameter that will be used to measure the performance of an algorithm. Three different sets of moves will be use to test the simulator with two sets of treatments. A system with 12 tanks and 2 carriers will be use for this test. The simulation starts with one carrier in the first tank while the second carrier in the last tank.

Table 1 and Table 2 show the treatments that will be used for the tests.

Table 1: Treatment 1 (T1)

Tank Number	Minimum Time(milliseconds)	Maximum Time(milliseconds)
0	655	5566
1	456	7677
2	456	7778
5	333	8333
4	333	8333
5	333	9333
6	1234	23444
7	123	9456
1	345	8785
3	567	7894
10	123	8376
11	453	8679

Table 2: Treatment 2 (T2)

Tank Number	Minimum Time(milliseconds)	Maximum Time(milliseconds)
0	454	7345
1	456	8754
2	234	6778
3	222	8456
4	337	8333
5	334	7456
6	123	9342
7	445	9567
8	341	7229
9	567	9667
10	556	10055
11	451	7445

5.1.1 First Test

The moves file to be use is shown in the table below.

Source Tank	Target Tank	Pick Time(milliseconds)
0	1	100
11	0	200
1	2	500
0	1	1500
2	3	2500
1	2	3500
3	4	4500
2	3	5700
4	5	6700
3	4	7666
5	6	8333
4	5	9333
6	7	14000
5	6	15000
7	1	16000
6	7	17000

1	4	18000
7	8	33555
4	10	38000
8	9	40999
10	11	41000
9	10	51344
11	2	52233
10	11	53000

After the simulation run, the following reports were generated

Good Bath

Tank Number	Product Name	Minimum Time(milliseconds)	Maximum Time(milliseconds)	Soak Time
0	T1-1	655	5566	1346
1	T1-1	456	7677	7466
1	T2-1	456	8754	8013
4	T1-1	333	8333	7994
3	T2-1	222	8456	7833
5	T1-1	333	9333	7780
4	T2-1	337	8333	7813
6	T1-1	1234	23444	7784
7	T1-1	123	9456	7764
6	T2-1	123	9342	7726
1	T1-1	345	8785	7671
7	T2-1	445	9567	7746
10	T1-1	123	8376	7683

9	T2-1	567	9667	8223
10	T2-1	556	10055	7543

Number of Good Baths: 15

Time Window Violation

Tank Number	Product Name	Minimum Time(milliseconds)	Maximum Time(milliseconds)	Soak Time	Description
0	T2-1	454	7345	7795	soak time is above
2	T1-1	456	7778	7878	soak time is above
2	T2-1	234	6778	7789	soak time is above
5	T2-1	334	7456	7770	soak time is above
8	T2-1	341	7229	7606	soak time is above

Number of Violations: 5

TREATMENT ERROR

Current Index	Product Name	Expected Tank Number	Processing Tank Number
3	T1-1	5	3
9	T1-1	3	4

Treatment Error Count: 2

5.1.2 Second Test

The moves file to be use for this test is shown in the table below.

Source Tank	Target Tank	Pick Time
0	1	100
11	0	200
1	2	500
0	1	1500
2	5	1900
1	2	2500
5	4	4500
2	3	5700
4	5	6700
3	4	7666
5	6	8333
4	5	8433
6	7	12000
5	6	13000
7	1	16000
6	7	17000
1	3	18000
7	8	33555
3	10	38000
8	9	40000
10	11	41000
9	10	51344
11	2	52233
10	11	53000

The reports generated for this test are shown below.

Good Bath

Tank Number	Product Name	Minimum Time(milliseconds)	Maximum Time(milliseconds)	Soak Time
0	T1-1	655	5566	1407
2	T1-1	456	7778	7591
1	T2-1	456	8754	7599
5	T1-1	333	8333	7761
4	T1-1	333	8333	8096
3	T2-1	222	8456	7886
5	T1-1	333	9333	7751
4	T2-1	337	8333	7798
6	T1-1	1234	23444	7897
7	T1-1	123	9456	7794
6	T2-1	123	9342	7906
1	T1-1	345	8785	7807
7	T2-1	445	9567	7671
3	T1-1	567	7894	7863
10	T1-1	123	8376	7742
9	T2-1	567	9667	8223
10	T2-1	556	10055	7632

Number of Good Baths: 17

Time Window Violation

Tank Number	Product Name	Minimum Time(milliseconds)	Maximum Time(milliseconds)	Soak Time	Description
1	T1-1	456	7677	7953	soak time is above
0	T2-1	454	7345	8014	soak time is above
2	T2-1	234	6778	8032	soak time is above
5	T2-1	334	7456	7649	soak time is above
8	T2-1	341	7229	7933	soak time is above

Number of Violations: 5

TREATMENT ERROR

Current Index	Product Name	Expected Tank Number	Processing Tank Number

Treatment Error Count 0

5.1.3 Third Test

The moves file to be use for this test is shown in the table below.

Source Tank	Target Tank	Pick Time
0	1	100
11	0	200
1	2	300
0	1	400
2	5	500
1	2	900
5	4	1500
2	3	2700
4	5	3700
3	4	4666
5	6	5333
4	5	5433
6	7	7000
5	6	8600
7	1	9000
6	7	10000
1	3	11000
7	8	12555
3	10	13000
8	9	14000
10	11	20000
9	10	21344
11	2	22233
10	11	31000

The reports generated for this test is shown below

Good Bath

Tank Number	Product Name	Minimum Time(milliseconds)	Maximum Time(milliseconds)	Soak Time
0	T1-1	655	5566	1889
1	T1-1	456	7677	7660
2	T1-1	456	7778	7633

1	T2-1	456	8754	7881
5	T1-1	333	8333	7726
4	T1-1	333	8333	7676
3	T2-1	222	8456	7728
5	T1-1	333	9333	7910
4	T2-1	337	8333	7845
6	T1-1	1234	23444	7897
7	T1-1	123	9456	7813
6	T2-1	123	9342	7818
1	T1-1	345	8785	7796
7	T2-1	445	9567	7994
3	T1-1	567	7894	7843
10	T1-1	123	8376	7665
9	T2-1	567	9667	8391
10	T2-1	556	10055	7761

Number of Good Baths: 18

Time Window Violation

Tank Number	Product Name	Minimum Time(milliseconds)	Maximum Time(milliseconds)	Soak Time	Description
0	T2-1	454	7345	7713	soak time is above
2	T2-1	234	6778	7653	soak time is above

5	T2-1	334	7456	7930	soak time is above
8	T2-1	341	7229	7774	soak time is above

Number of Violations: 4

TREATMENT ERROR

Current Index	Product Name	Expected Tank Number	Processing Tank Number

Treatment Error Count 0

5.2 EVALUATION AND FINDINGS

5.2.1 Evaluation

The total number of good baths within time windows is a parameter that will be used to measure the performance of an algorithm. From the three test performed in section 5.1, we can see that the first test had 2 treatment error, which shows that two baths were done in the wrong tank. Treatment error is a very critical error which should be avoided. This is an indicator that the moves are not good enough and that the algorithms that produced the moves does not give a good scheduling.

The second test does not have any treatment error and the total number of good baths is 17, while the total number of time window violations is 5. This moves produced a better scheduling than the moves from the first test.

The third test did not produce any treatment error, and the total number of good baths is 18 and a total number of time window violations of 4. Since the total number of good bath are higher for the third test when compared to the second test, we can say that the algorithm for the last test is the best algorithm for the hoist system parameters we have adopted for the test.

5.2.2 Findings

We have been able to show that given a set of moves based on some hoist scheduling algorithm, it is possible to use a these moves as input to a visual simulator that can simulate the hoist system operations and thus enable the evaluation of the algorithm.

We were able to identify finite states that the hoist and tank in a hoist system can be in any point in time. These finite states where derived by grouping the infinite states into finite states that could be used to model and then implement the simulator. These states can be adopted by those who want to model the hoist system. These states can also be adopted while modeling the hoist system using the Discrete Event systems specification **(**DEVS) formalism or also while modeling with DEVS graphical notations like DEVS Driven Modeling Language (DDML) (Traoré, 2009).

We were able to device a programming technique that enabled the separation of the controller from the view at the application level by initializing the reference of the GUI components in the controller class. Making a separation between the controller and the view has always been a very difficult task in swing application, but this technique has proved to

be very effective. This will make it very easy to modify the view, using swing or other APIs. This technique will create a new insight in software engineering and most especially in the application of MVC pattern in swing applications.

Because of the MVC design pattern adopted for building the simulator, it will be possible to change the view and use a view that is based on a 3D interface based on the OpenGL graphic API. The model can also be change by adopting a framework such as simStudio or DEVSJAVA, which are java implementations of DEVS formalism. This will be possible since the hoist system is a Discrete Event System.

6. CONCLUSIONS AND FURTHER RESEARCH

The result of my thesis will interest those who are seeking to develop and test algorithms for the HSP and also manufacturing outfits that make use of the hoist systems.

The MVC pattern adopted has proved to be a very good design pattern for the development of the hoist scheduling simulator. My design can be used as building blocks for more sophisticated hoist system simulators in the future. Clearly, the work in my thesis will be a valuable asset to many people who are seeking to evaluate their HSP algorithms.

Further work can be done on this simulator with less work, because of the MVC design pattern adopted for this simulator, since the view, model and controller are neatly separated. Work can be done on only the view to have a GUI that has three dimensional (3D) capabilities, based on OpenGL API. Work can also be done on the model to adopt a Discrete Event Specification (DEVS) modelling. A Java based API for DEVS such as SimStudio can be use for this implementation. Further work could also be done to develop a version of this simulator that can run on the web.

7. REFERENCES

(Lamothe et al, 1994) Lamothe, J., Correge, M., Delmas, J. hoist scheduling problem in a real time context, 11th International Conference on Analysis and Optimization of Systems Discrete Event Systems

(Lei and Wang, 1989) Lei, L. and Wang, T. J. A proof: the cyclic hoist scheduling problem is NP-complete, Working paper #89-0016, Rutgers University.

(Riera and Yorke-Smith, 2002) Riera, D. and Yorke-Smith, N. *An Improved Hybrid Model for the Generic Hoist Scheduling Problem*. Annals of Operations Research 115, 173-191, September 2002.

(Phillips and Unger, 1976) Phillips, L.W., Unger, P. S. Mathematical Programming Solution of a Hoist Scheduling Program, AIIE Transactions, 1976, 8, n. 2, 219-225.

(Baptiste et al., 1994) Baptiste, P., Legeard, B., Manier, M. A. A Scheduling Problem Optimisation Solved with Constraint Logic Programming, In: Proc. of the Intl. Conf. on Parallel Architectures and Compilation Techniques

(Lam, 1997) Lam, K. A Heuristic Method for Multiple Hoist Scheduling Problems by using Simulated Annealing and Local Search'. Working Paper (1997)

(Rodoˇek and Wallace, 1998) Rodoˇek, R. and Wallace, M. A Generic Model and Hybrid Algorithm for Hoist Scheduling Problems , In Proc. 4th Int. Conf. on Principles and Practice of Constraint Programming (CP98). Springer-Verlag, LNCS 1520

(Maria, 1997) Anu Maria, Introduction to Modeling and Simulation, *Proceedings of the 1997 Winter Simulation Conference*

(Fishman, 2001) Fishman, G. S., Discrete-Event Simulation modelling' programming and analysis, Springer Series in Operations Research

(Wainer, 2009) Wainer, G. A., Discrete-Event Modeling and Simulation, a practitioner's approach. Taylor & Francis Group.

(Gamma et al, 1995) Gamma, E., Helm, R., Johnson, R. and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, 1995.

(Krasner and Pope, 1988) Krasner, G.E. and Pope, S.T. (1988). A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object Oriented Programming*, 1(3):26-49.

(Traoré, 2009)Traoré, M. K., A Graphical Notation for DEVS, SpringSim '09 Proceedings of the 2009 Spring Simulation Multiconference

(Buschmann et al., 1996) F. Buschmann, R. Meunier, H. Rohnert, P.Sommerlad, M. Stal, Pattern-Oriented Software Architecture A System

of Patterns, John Wiley and Sons Ltd, Chichester, UK, 1996 ISBN 0-471-95869-7

(Kuchana, 2004) Kuchana, P., Software architecture design patterns in Java, Auerbach Publications.

(Manier and Bloch., 2003) M. Manier and C Bloch, A Classification for Hoist Scheduling Problems, The International Journal of Flexible Manufacturing Systems, 15, 37–55, 2003

(Zeigler et al, 2000)Zeigler, B. P., H. Praehofer, and T. G. Kim. 2000. Theory of modeling and simulation, 2nd. ed. New York: Academic Press.

(Fowler, 2010), Amy Fowler, A Swing Architecture Overview, an Article in Sun Developer Network (SDN), http://java.sun.com/products/jfc/tsc/articles/architecture/