Danmarks Tekniske Universitet

02115 Java Programming

DTU Informatics

A simple Cinema Seats Booking System

Fariha Nazmul

Student ID: s094747

10.12.10

Table of Contents

1. Introduction

The report is based on the Java implementation of a simple Cinema Seats Booking System that can register seat reservations for persons going to cinema shows. According to the problem description, the model of our system has to handle two different kinds of users:

a. The customers who are looking for a screening of a movie on a certain day and time and who wants to book some tickets.
b. The system administrator who sets up the system and adds new film shows to the cinema theatres.

The model of the system should have the following properties:

The office or an administrator sets up the available cinema theatres each having a given capacity of seats. Setting up the plans for showing films in the cinemas is also a job for the office.

The audience communicates with the office. And each booking is identified by a unique booking number to which other booking information is attached.

The booking information consists of the person name, the film name, the cinema name, the day, and the location of the booked seats (row numbers and seat numbers).

A booking is always done one week in advance. One can only book for a day within a week. The system only handles one week at a time.

For the simplification of the model, it is assumed that one film is screened at one cinema hall on a particular day and information on the hours of the films shown is left out. The system does not consider the charging for booked seats. Also no authentication or password is required for getting access to use the system.

The Cinema System has both a command line based user interface and a GUI. In the command line based user interface commands are written one by one and then read, parsed and executed. In the graphical interface the user (both administrator and customer) can perform their valid actions using buttons and menus. The user can also see the list of films that are being showed and the list of cinema halls that are showing films.

2. Analysis and Design

This section discusses the analysis and design phase of the Cinema Seats Booking System. First of all, the classes are discovered from the project description emphasizing on loose coupling of the classes. After that the class design is basically influenced by the responsibility driven design and localization of changes.

2.1 Discovering Classes

The Cinema System model can be analyzed and compartmentalized into smaller elements and then each of these elements can be designed as a class in the system. For example, each cinema has a theatre. A theatre has a number of rows and each row is made up of a number of seats. So if we consider each of these elements as an individual class then the design of the model becomes much clear and simple and loosely coupled. Again the booking information of each booking has some fields that are general for each booking. So a class can be designed to hold single booking information which has the appropriate properties as its attributes. Then each screening of a film has to maintain the information of the film, the cinema where it will be screened and the day of the show. Therefore a class can be designed to hold all these information of a particular film show. By following this approach we can find the elementary classes of the system.

Once we have designed the elementary blocks of the system, then we need to create collections of them. These collections actually act as the data bank of the overall Cinema Booking System. As each collection also has some actions of its own to perform on it, designing each collection along with its actions as an individual class makes handling of the data bank easier. Also creating individual classes for each collection gives the freedom to change to another collection type. In that case changing only the collection class will maintain the actual functionality of the system. Considering this we can create collection classes for all cinemas, film shows and bookings named as CinemaRepository, FilmShowRepository and BookingRepository respectively.

After discovering the collection classes, we need to handle the overall operations of the cinema booking system which can act as the engine of the whole model. The CinemaSystemEngine class can be designed to create and maintain all the collections of the system i.e. CinemaRepository, FilmShowRepository and BookingRepository and perform the desired operations on them.

Now to handle the IO operations on the system, we need to design a class that only executes these functions. These compartmentalization of the IO operations helps to maintain the functionality of all the IO based actions at one place.

As we have discovered all the classes that are necessary for storing data and performing the operations on the system, we need two classes to represent the command line interface and graphical interface of the system.

2.2 Class Design

This section describes and discusses the data representation of the classes and algorithms for performing the desired results on the system.

2.2.1 Classes for Enumeration

User: The Cinema Booking System has two types of users i.e. administrator and customer. Administrator has the right to add a cinema hall in the system and then add new films to the existing cinema halls. The administrator can also save the state of the whole system in a file and later can load the entire system from the file. Choosing the type of user interface and closing the system are also administrative operations. On the other hand, the customer can reserve any number of seats available for a particular film show. For this purpose, both the customer and administrator can search for the show days and show place of any film shows. For this purpose an enumeration class **User** has been created with the values User.ADMIN, User.CUSTOM, User.USER and User.UNKNOWN representing admin mode, custom mode, no mode and unknown mode respectively. Each enum can also be represented by a string.

CommandWord: The system has a defined set of commands that can be executed. This enumeration class is designed to hold all the commands of the system associated with their string representation and the number of parameters needed for them.

Day: The cinema system manages the booking of seven days. So an enumeration class is created to hold the information of each day along with a string description and integer representation.

2.2.2 Classes to use enumeration classes

Day Parameters & User Parameters: These classes are used to make the enumeration classes usable. Each class contains a HashMap that contains all the valid enum values along with their string representation and methods to get the corresponding enum value given the string representation of it as input. The HashMap is indexed by the string representation of the enum and thus enables to get the corresponding enum value from user input very easily and in short time.

CommandWords: This class has the same functionality as the other two user classes of enum for CommandWord. But it also has an extra functionality of choosing the legal mode of a command. One HashMap “validAllCommands” maintains the string value and corresponding enum value for each command. There are methods to get the enum values if the string is given as input and another function to check any string if it’s a valid command of the system. Another HashMap is used in this class named “legalUserForCommand” to maintain the list of valid users for each command. The HashMap is indexed by a CommandWord. There is a method to check if a command is legal to be executed for a givne user mode. This structure enables us to extend to system with any new commands very easily and without changing any other classes. To add any new command to the system in future, one needs to add the enum of the command in the CommandWord class and add an entry in the HashMap "legalUserForCommand" with the list of legal users.

2.2.3 Classes for the Elementary blocks

BookingStatus: This class is used to maintain the booking status of each seat for each day of the week. It has an array of Boolean values and a method to set the state of a slot to any given value and a method to get the state of a slot. The size of the array is a parameter of the constructor. So it gives the option to extend the system to hold reservation for any number of days.

Seat: The Seat class holds the information of a single seat in a row of a theatre. It has the attribute row number, seat number and an object of class BookingStatus to keep the booking information of that particular seat i.e. whether it is booked or not.

Row: This class holds information of a single row of a theatre. It has the field row number, length of the row and an ArrayList holding all the seat objects of that row. This class can book seats residing in that row.

Theatre: This class keeps the information related to each theatre in a cinema hall i.e. name of the theatre, number or rows, length of each row and all the row objects of that theatre.

Cinema: The Cinema class is designed to hold the name of the cinema hall and a single theatre object. But this can be easily extended to hold many theatres for a cinema hall.

FilmShow: Keeping only the film name in a class is not worthy enough. That is why I have created a class named FilmShow to keep information of a particular show of the system. It has the attributes a film name, a cinema name and the day of the show. This structure also helps to prevent the addition of a film to a cinema hall on the same day.

BookingInformation : Each booking of the system is identified by a unique booking number and it contains the information of the film show, reserved seats and the customer. This class is created to maintain the information of a single booking. This helps us to create a collection of booking information and search in that collection for any existing booking or customer.

2.2.4 Classes for Collections

Instead of having the collections of Cinema, FilmShow and BookingInformation in the SystemEngine, I have designed individual collection classes for them. This structure helps to maintain the functionality of the each collection to be bounded within a single class and thus enables us to change that particular collection type or functionality by modifying that collection class only.

CinemaRepository: To maintain the information of all the existing Cinemas in the system, I have chosen an ArrayList of type Cinema. This enables to add as many Cinema as possible to the system and also helps to check if a Cinema already exists in the system This class has the methods to add a Cinema in the system and to book seats for a film show that is being shown in this Cinema hall. For booking seats I've followed the algorithm or sequential booking i.e. once all the seats of a row is exhausted then only seats from the second row is given to the customer.

FilmShowRepository:This class holds the collection of film shows and performs the operations related to film show. It can add a film show in the system and check when and where a film is showing. It can also check if a existing cinema is free or not.

BookingRepository: This class is the collection class for the booking information. In this class I have chosen a HashMap that is indexed by a booking number and keeps the corresponding booking information with the booking number. This class can add a booking information in the system and find and show any booking given a booking number or a customer's name.

2.2.5 Classes for System Engine

CinemaSystemEngine*:* This class is the main engine of the whole cinema booking system. All the core functionalities of the system is implemented here and it holds the data bank of the system also. This class holds all the objects of the collection classes. Thus this is the main database of the system. When the administrator wants to save the state of the system, he can save an object of this engine because all the required data objects are maintained in this class. Again the administrator can load the system data by loading an object of this system engine. So only one instance of this engine is created in the main system and this system is passed as parameter to the interface classes and all the functionalities of the system in performed on this object i.e. this class contains implementations for all the available commands in the system. This class also maintains the mode of the current user and checks for every command if the current user has the authority to run the desired command.

FileHandler: This class contains all the IO operations of the system. This takes the current object of the cinema system engine as parameter and works on it. It implements the load and save command execution of the system and also the addFrom command execution.

2.2.6 Classes for Command Line Interface

To implement the command line interface of the system, we need to design some classes that can read each command with its parameters, tokenize them into parts and perform actions accordingly. For this purpose, we have to designed the command forms, the property of each command and their input representations. The class named CommandWord is designed to hold the properties of the commands. CommandWords class works on the previously mentioned class to check the validity of each command and uses the user-defined exception classes InvalidCommandException and UnrecongnizedCommandException when the input commands are invalid or unknown.

Command: This class holds the information of a single command i.e. it has a field for the CommandWord and a list for the parameters of the command. Since different command has different numbers or parameter an ArrayList is used hold the parameter list.

Parser: A parser is also designed to take input from the console and parse them to appropriate categories of commands to execute them. This parser structure is influenced by the examples of the course book.

CommandLineInterface: In the end a class needs to be designed to run the parser and perform the desired operations on the existing cinema system and CommandLineInterface class is used for that purpose. This function uses the parser object to get input from the user as an instance of Command and then checks the validity of the command and throws appropriate exceptions. If the command is valid and user mode is legal, then this class chooses the appropriate function call depending on the input command and executes it.

2.2.7 Classes for Graphical Interface

DialogBox: This is a helping class for the graphical user interface and it creates the dialog box with appropriate number of input fields depending on the action to be performed on the system and returns the user inputs as a list of parameters to the main frame of the GUI. This is created as an individual class to avoid code duplication and loose coupling of the classes.

GraphicalInterface: This is the class that creates the GUI of the system and performs the actions accordingly. In this class I have created the frame and desired menu items and buttons to perform the functionalities of the Cinema Booking System. There are buttons for each of the desired actions i.e. commands in command line interface and clicking that button creates a dialog box to get the desired user inputs. After getting the input it calls the methods defined in the CinemaSystemEngine to perform the actions. Here I have created to listPanels to show the currently existing films and cinemas in the sytem. In the file menu I have added the IO operations of the system as menu items.

2.2.8 Class for Cinema System

CinemaSystem: This is the main class of the system that contains the main function of the cinema booking system. This main function takes a string argument. If no argument is passed to the main function, it will start running the command line interface of the system. On the other hand, if "GUI" is given as parameter, it will run the graphical interface of the system. So it is up to the administrator to choose any form of interface and run the system.

2.3 Overview of Class Design

The following diagram represents the overview of the overall designed system.

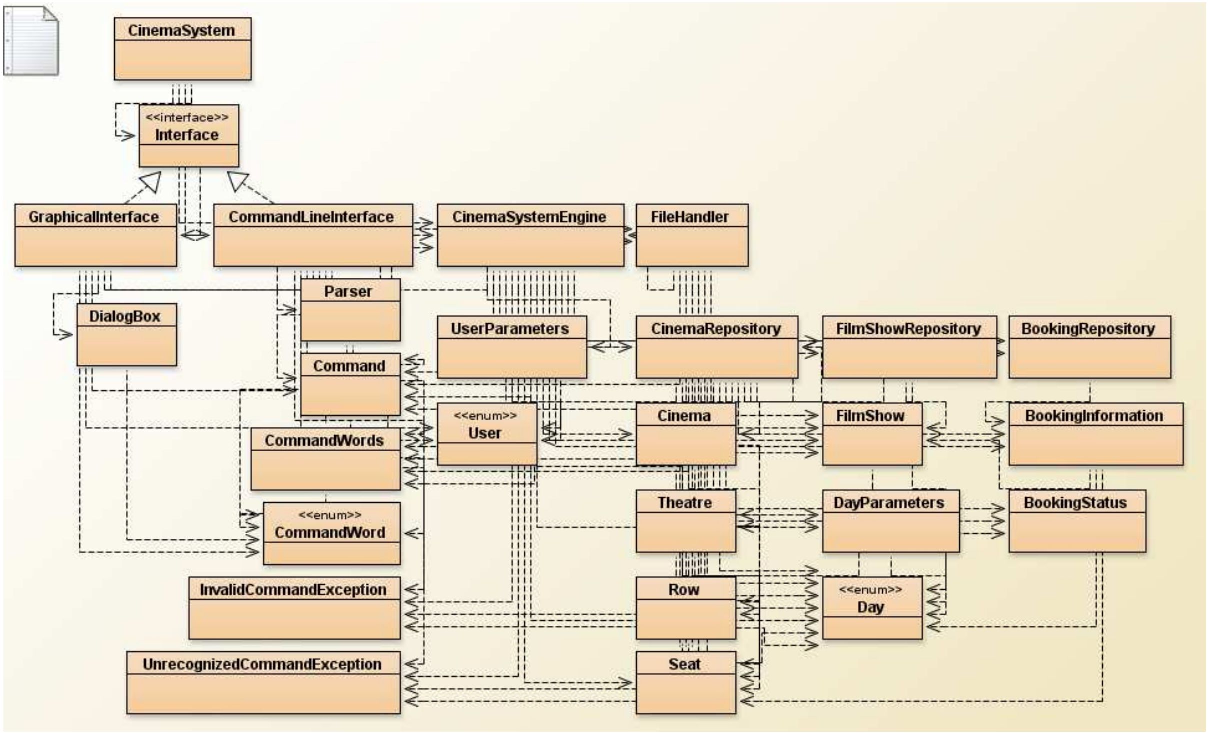


Figure 1: Overall project diagram

2.4 The Outline Implementation

In this section a short overview of the class design is presented with emphasis on data fields and public methods of each class. In the class diagram, the data fields of a class that are associated with another class are not shown in the "attribute" box. Instead of that, each associated attribute is shown as connecting elements of the two associated classes. Here is class diagram of the CinemaSystemEngine and its components.

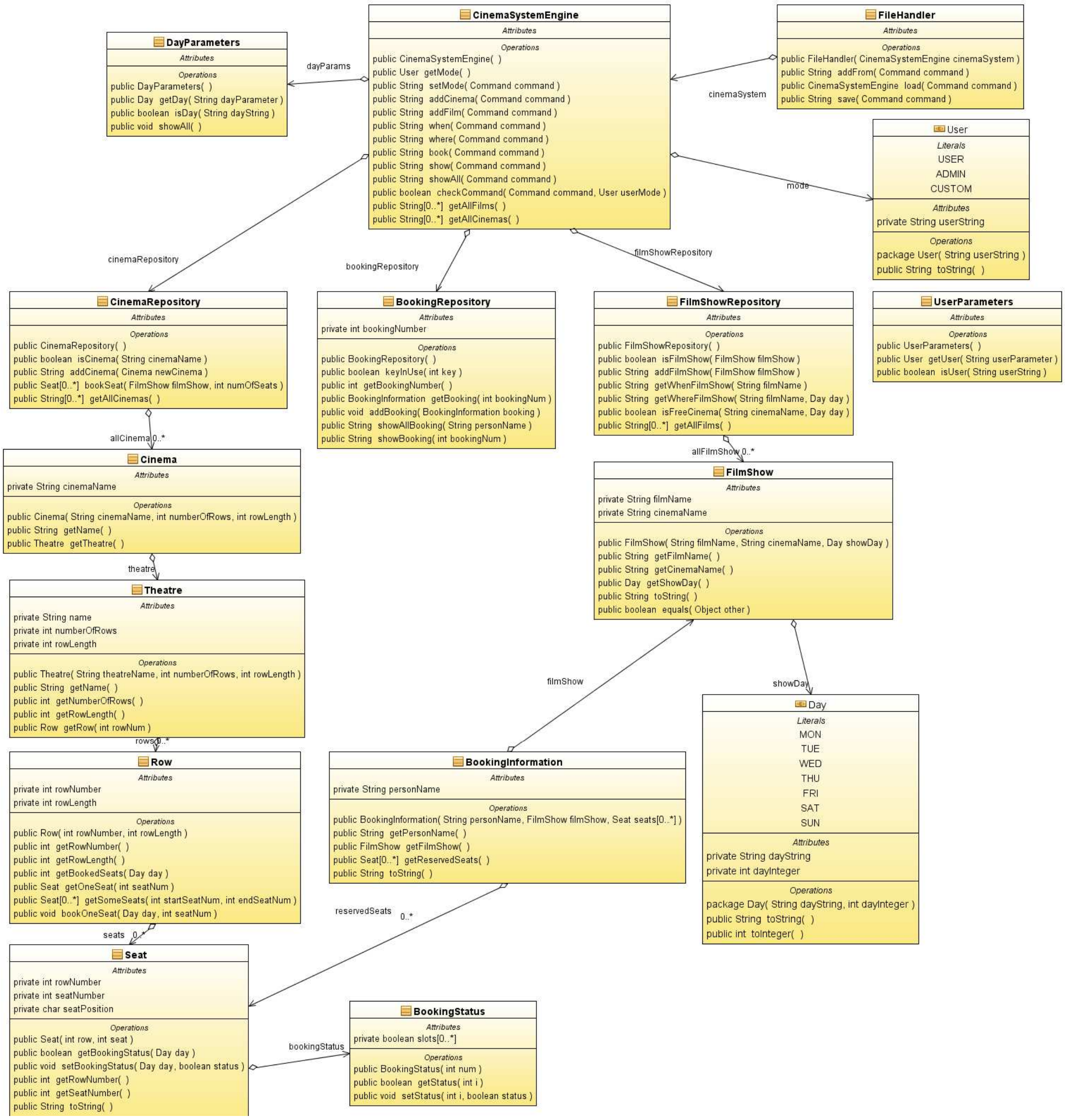

CinemaSystemEngine
Attributes
Operations
public CinemaSystemEngine( )
public User getMode( )
public String setMode(Command command)
public String addCinema(Command command)
public String addFilm(Command command)
public String when(Command command)
public String where(Command command)
public String book(Command command)
public String show(Command command)
public String showAll(Command command)
public boolean checkCommand(Command command, User userMode)
public String[0..*] getAllFilms( )
public String[0..*] getAllCinemas( )
FileHandler
Attributes
Operations
public FileHandler(CinemaSystemEngine cinemaSystem)
public String addFrom(Command command)
public CinemaSystemEngine load(Command command)
public String save(Command command)
cinemaSystem
DayParameters
Attributes
Operations
public DayParameters( )
public Day getDay(String dayParameter)
public boolean isDay(String dayString)
public void showAll( )
dayParams
User
Literals
USER
ADMIN
CUSTOM
Attributes
private String userString
Operations
package User(String userString)
public String toString( )
mode
cinemaRepository
bookingRepository
filmShowRepository
CinemaRepository
Attributes
Operations
public CinemaRepository( )
public boolean isCinema(String cinemaName)
public String addCinema(Cinema newCinema)
public Seat[0..*] bookSeat(FilmShow filmShow, int numOfSeats)
public String[0..*] getAllCinemas( )
BookingRepository
Attributes
private int bookingNumber
Operations
public BookingRepository( )
public boolean keyInUse(int key)
public int getBookingNumber( )
public BookingInformation getBooking(int bookingNum)
public void addBooking(BookingInformation booking)
public String showAllBooking(String personName)
public String showBooking(int bookingNum)
FilmShowRepository
Attributes
Operations
public FilmShowRepository( )
public boolean isFilmShow(FilmShow filmShow)
public String addFilmShow(FilmShow filmShow)
public String getWhenFilmShow(String filmName)
public String getWhereFilmShow(String filmName, Day day)
public boolean isFreeCinema(String cinemaName, Day day)
public String[0..*] getAllFilms( )
UserParameters
Attributes
Operations
public UserParameters( )
public User getUser(String userParameter)
public boolean isUser(String userString)
allCinema 0..*
Cinema
Attributes
private String cinemaName
Operations
public Cinema(String cinemaName, int numberOfRows, int rowLength)
public String getName( )
public Theatre getTheatre( )
allFilmShow 0..*
FilmShow
Attributes
private String filmName
private String cinemaName
Operations
public FilmShow(String filmName, String cinemaName, Day showDay)
public String getFilmName( )
public String getCinemaName( )
public Day getShowDay( )
public String toString( )
public boolean equals(Object other)
theatre
Theatre
Attributes
private String name
private int numberOfRows
private int rowLength
Operations
public Theatre(String theatreName, int numberOfRows, int rowLength)
public String getName( )
public int getNumberOfRows( )
public int getRowLength( )
public Row getRow(int rowNum)
filmShow
showDay
Day
Literals
MON
TUE
WED
THU
FRI
SAT
SUN
Attributes
private String dayString
private int dayInteger
Operations
package Day(String dayString, int dayInteger)
public String toString( )
public int toInteger( )
rows 0..*
Row
Attributes
private int rowNumber
private int rowLength
Operations
public Row(int rowNumber, int rowLength)
public int getRowNumber( )
public int getRowLength( )
public int getBookedSeats(Day day)
public Seat getOneSeat(int seatNum)
public Seat[0..*] getSomeSeats(int startSeatNum, int endSeatNum)
public void bookOneSeat(Day day, int seatNum)
BookingInformation
Attributes
private String personName
Operations
public BookingInformation(String personName, FilmShow filmShow, Seat seats[0..*])
public String getPersonName( )
public FilmShow getFilmShow( )
public Seat[0..*] getReservedSeats( )
public String toString( )
reservedSeats 0..*
seats 0..*
Seat
Attributes
private int rowNumber
private int seatNumber
private char seatPosition
Operations
public Seat(int row, int seat)
public boolean getBookingStatus(Day day)
public void setBookingStatus(Day day, boolean status)
public int getRowNumber( )
public int getSeatNumber( )
public String toString( )
bookingStatus
BookingStatus
Attributes
private boolean slots[0..*]
Operations
public BookingStatus(int num)
public boolean getStatus(int i)
public void setStatus(int i, boolean status)

Figure 2: Part of the Class diagram handling the Cinema System Engine and IO operations

The next class diagram shows the system maintaining the command line interface and graphical interface of the Cinema Booking System.

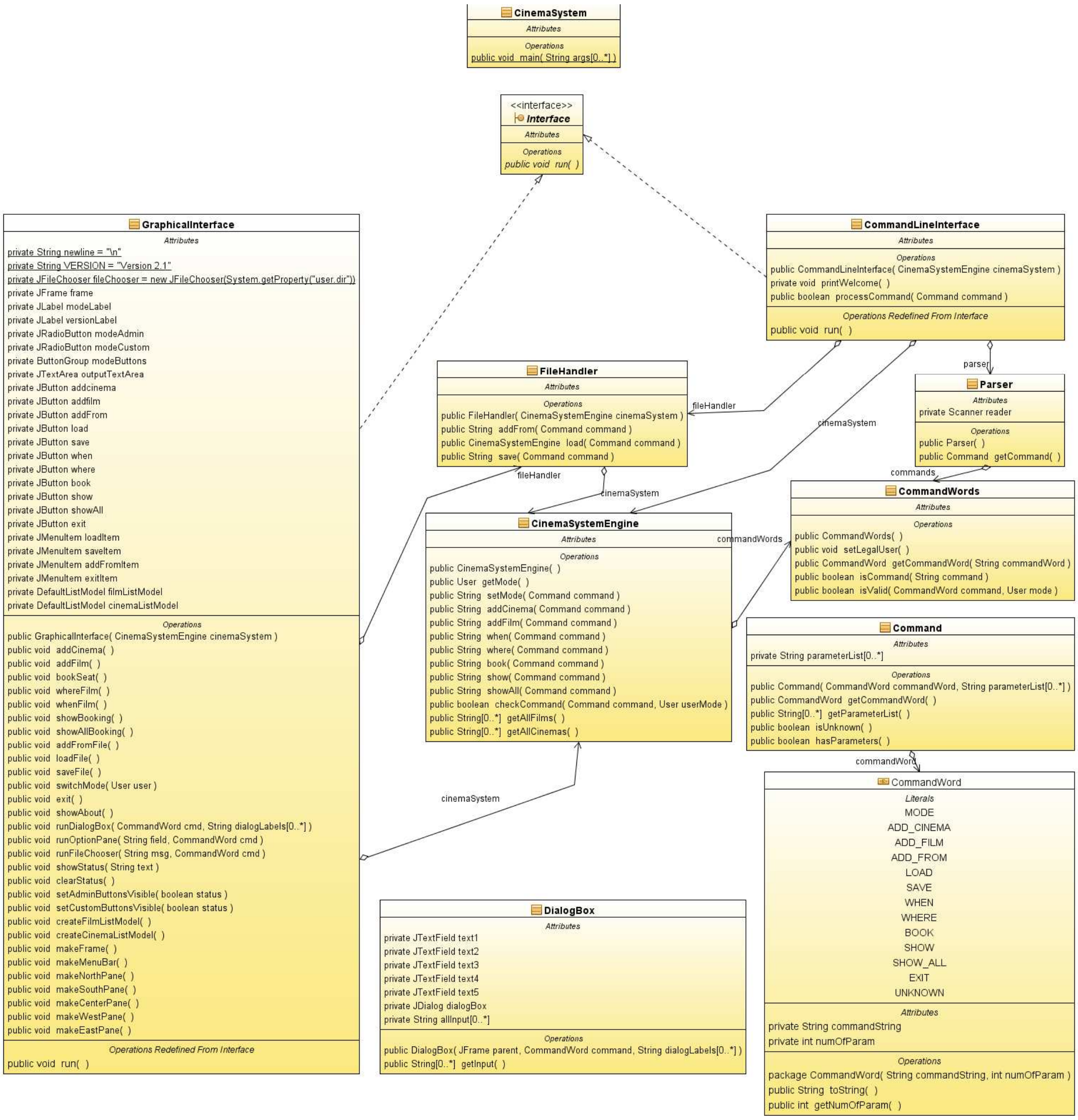

CinemaSystem
Attributes
Operations
public void main(String args[0..*])
<<interface>>
Interface
Attributes
Operations
public void run( )
GraphicalInterface
Attributes
private String newline = "\n"
private String VERSION = "Version 2.1"
private JFileChooser fileChooser = new JFileChooser(System.getProperty("user.dir"))
private JFrame frame
private JLabel modeLabel
private JLabel versionLabel
private JRadioButton modeAdmin
private JRadioButton modeCustom
private ButtonGroup modeButtons
private JTextArea outputTextArea
private JButton addcinema
private JButton addfilm
private JButton addFrom
private JButton load
private JButton save
private JButton when
private JButton where
private JButton book
private JButton show
private JButton showAll
private JButton exit
private JMenuItem loadItem
private JMenuItem saveItem
private JMenuItem addFromItem
private JMenuItem exitItem
private DefaultListModel filmListModel
private DefaultListModel cinemaListModel
Operations
public GraphicalInterface(CinemaSystemEngine cinemaSystem)
public void addCinema( )
public void addFilm( )
public void bookSeat( )
public void whereFilm( )
public void whenFilm( )
public void showBooking( )
public void showAllBooking( )
public void addFromFile( )
public void loadFile( )
public void saveFile( )
public void switchMode(User user)
public void exit( )
public void showAbout( )
public void runDialogBox(CommandWord cmd, String dialogLabels[0..*])
public void runOptionPane(String field, CommandWord cmd)
public void runFileChooser(String msg, CommandWord cmd)
public void showStatus(String text)
public void clearStatus( )
public void setAdminButtonsVisible(boolean status)
public void setCustomButtonsVisible(boolean status)
public void createFilmListModel( )
public void createCinemaListModel( )
public void makeFrame( )
public void makeMenuBar( )
public void makeNorthPane( )
public void makeSouthPane( )
public void makeCenterPane( )
public void makeWestPane( )
public void makeEastPane( )
Operations Redefined From Interface
public void run( )
CommandLineInterface
Attributes
Operations
public CommandLineInterface(CinemaSystemEngine cinemaSystem)
private void printWelcome( )
public boolean processCommand(Command command)
Operations Redefined From Interface
public void run( )
parser
fileHandler
cinemaSystem
FileHandler
Attributes
Operations
public FileHandler(CinemaSystemEngine cinemaSystem)
public String addFrom(Command command)
public CinemaSystemEngine load(Command command)
public String save(Command command)
fileHandler
cinemaSystem
Parser
Attributes
private Scanner reader
Operations
public Parser( )
public Command getCommand( )
commands
CommandWords
Attributes
Operations
public CommandWords( )
public void setLegalUser( )
public CommandWord getCommandWord(String commandWord)
public boolean isCommand(String command)
public boolean isValid(CommandWord command, User mode)
CinemaSystemEngine
Attributes
Operations
public CinemaSystemEngine( )
public User getMode( )
public String setMode(Command command)
public String addCinema(Command command)
public String addFilm(Command command)
public String when(Command command)
public String where(Command command)
public String book(Command command)
public String show(Command command)
public String showAll(Command command)
public boolean checkCommand(Command command, User userMode)
public String[0..*] getAllFilms( )
public String[0..*] getAllCinemas( )
commandWords
Command
Attributes
private String parameterList[0..*]
Operations
public Command(CommandWord commandWord, String parameterList[0..*])
public CommandWord getCommandWord( )
public String[0..*] getParameterList( )
public boolean isUnknown( )
public boolean hasParameters( )
commandWord
cinemaSystem
CommandWord
Literals
MODE
ADD_CINEMA
ADD_FILM
ADD_FROM
LOAD
SAVE
WHEN
WHERE
BOOK
SHOW
SHOW_ALL
EXIT
UNKNOWN
Attributes
private String commandString
private int numOfParam
Operations
package CommandWord(String commandString, int numOfParam)
public String toString( )
public int getNumOfParam( )
DialogBox
Attributes
private JTextField text1
private JTextField text2
private JTextField text3
private JTextField text4
private JTextField text5
private JDialog dialogBox
private String allInput[0..*]
Operations
public DialogBox(JFrame parent, CommandWord command, String dialogLabels[0..*])
public String[0..*] getInput( )

Figure 3: Part of the class diagram handling the command line and graphical interface

3. From Design to Implementation

This section presents a bridging from design to source code by stating some superior matters on the implementation.

The cinema system provides two types of user interfaces. Each of the user interface class has a common method "run" to start the interface. The choice of the interface to run is dependent on the user input. So it is a good idea to create a Java interface "Interface" that has a method definition "void run()" and then each of the user interface class implements this interface. This interface helps us to create one instance of the interface in the system and assign appropriate subclass object to it depending on the user input. Moreover we know, Java allows a class to implement any number of interfaces.

To handle the errors occurring in the system for various reasons I have defined two exception classes named InvalidCommandException and UnrecongnizedCommandException. This exception classes are used when a user inputs an invalid command or a command that is not legal to run on the existing user mode. Another Java exception IllegalArgumentException is used when the parameters for the commands in UI or actions in GUI are not defined correctly. IOException is used for any kind of file. Also errors while trying to execute commands in wrong sequence is also handled in the system. By throwing appropriate exceptions and then catching them in the system, made the system safe and invulnerable to inappropriate user input.

While implementing the GUI, actions are detected by adding actionlistener to the buttons. But all the methods of the main CinemaSystemEngin takes as input a CommandWord and its list of parameters and executed to function. To use the same system model for both the interfaces I have used the same structure of input in the GUI while calling these functions. After invocation of an actionlistener, I have created the corresponding Command object with the CommandWord and input list and called the function in the System Engine. Even if in future there is no command line interface of the system, we will only need the enum class CommandWord for the GUI.

Another important factor of implementation was to avoid any kind of code duplication. For this purpose I have created the DialogBox class that can create different kinds of dialog box depending on the user input. I have also created some methods namely runOptionPane(), runFileChooser() to avoid duplication of code as these methods are used by many of the actionlistener methods.

4. Testing

Two types of testing have been done on the implemented system. First of all, structural testing or white-box testing of the three main collection classes has been performed. Since most of the functionalities of the system are performed on the collection classes, it is necessary to test each and every method of these classes. Collection classes perform their methods based on the elementary classes and the enumeration classes. So successful structural testing of the collection classes also ensures the successful performance of the elementary classes. Structural tests of these collection classes determines that almost all branches of the program's choice and loop constructs have been tested and they work accordingly.

Once the structural testing of the collection classes are done, then we go one step up in the hierarchy and test the CinemaSystemEngine for each of the methods it implements. CinemaSystemEngine performs operation on the collection classes. But it also does some additional checking of command validation, legal mode validation and correct argument validation. Structural tests on this class can ensure that all of those validations are done successfully.

Now we also need to test the actual functionalities of all the functions of the system. To perfrom the functional testing I have taken the same approach as the structural testing i.e. first I tested the collection classes and then the CinemaSystemEngine. By checking the functionalities of these classes it can be ensured that the system solved the problem specified in the assignment text.

Testing each of the elementary classes is not so necessary since the collection classes are using these classes to perform their methods. If the elementary classes were not working properly both in the sense of structural and functional, then the testing on the collection classes would have failed. Same reasoning can be made for the other enum classes and classes and user of enum classes.

5. Summary

In summary, I have designed and implemented the CinemaSystemEngine that performs all the functionalities mentioned in the given problem description. For providing the user interface of the system I have designed and created a command line based interface and a graphical interface of the Cinema Booking System.

In order to achieve this, I have to design the class structures of the system couple of times. The key factors while designing the system were to design a loosely coupled structure, to utilize the benefits of object oriented programming as much as possible, to make the design easily extendable in future and to make it safe and invulnerable.

The command line interface of the system supports all the commands specified in the system description and two different user modes. It also checks the validity of the command, legal mode of executing a command, validity of the parameters of each command. It also verifies the execution of legal sequence of commands.

The graphical user interface of the system is also implemented fully according to the specification and supports all the functionality of the system. The GUI takes the same parameters as input from the user as the command line interface. But in some situations it could have been made more automated by providing more exposure to the existing data. For example, to check when a particular film is showing, both interfaces take the name of the film as input from the user. But in the GUI it could have been done by giving an option to the user to choose a film from a list of existing films and show the result accordingly.

The design and implementation of the system is highly influenced by following the guidelines and examples of the textbook “Objects First with Java - A practical Introduction Using BlueJ”. The overall implementation of the Cinema Booking System is done in BlueJ.

6. User's Guide

How to Run the Program

- To start the Graphical Interface, a string parameter "GUI" is passed to the main() method of the Class CinemaSystem and executed.
- Running the main() method of the Class CinemaSystem without any argument will start the Command Line Based Interface.

The Set of Commands for Command Line Interface

The commands are not case sensitive.

The set of commands given below are valid for both the administrator and the customers.

Command	**Parameters**	**Description**
`mode`	*user*	The parameter *user* is either `ADMIN` or `CUSTOM`. The command allows for switching between administrator and customer mode.
`when`	*film name*	The command lists the names of days on which the film identified by *film name* is shown. All cinemas are searched.
`where`	*film name* *day name*	The command lists the names of all cinemas showing the film identified by the specified *film name* on the day specified by *day name.*
`show`	*booking number*	Lists booking information corresponding to the stated *booking number.*
`showAll`	*person name*	Lists all bookings done by the specified *person name.*

The set of commands given below are directed towards the administrator.

Command	**Parameters**	**Description**
`addCinema`	*cinema name* *rows* *rowLength*	The command adds a cinema hall for booking. The parameters are: The *cinema name* identifying the cinema, the number of *rows* of seats in the cinema (≤ 30), and the number of seats of each row (≤ 20) given by *rowLength*.
`addFilm`	*film name* *cinema name* *day names*	The command defines the name of a film, the name of the cinema showing the film, and the days of the week on which the film will be shown in the given cinema. The parameters are: The *film name* identifying the film to be played in the cinema given by *cinema name* on the days stated by *day names*. The *day names* is a list of days, e.g. `Mon Thu Sat`
`addFrom`	*file name*	The command adds from a file cinema halls and films defined for booking. . The parameter *file name* identifies the file holding informations on the cinemas and films to be added: Each line of the file holds either an `addCinema` command or an `addFilm` command
`load`	*file name*	The command initializes the system with the state previously stored in a file by use of the `save` command below. The parameter *file name* identifies the file holding information on the defined cinemas, films and bookings done in a previous run of the program where the state has been saved to that file
`save`	*file name*	The command saves the state of the system in a file. The parameter *file name* identifies the file which should hold information on the defined cinemas, films and bookings done in the actual run of the program
`exit`		Stops running the program.

The set of commands given below are directed towards the customers.

Command	Parameters	Description
book	*person name* *film name* *cinema name* *day name* *number of seats*	The command books a number of seats in a cinema hall for a film shown in that cinema on the specified day. The result of the booking is given by returning a unique booking number used to identify the booking just done taken from the sequence of booking numbers. The parameters are: The *person name* to identify the person doing the booking, the *film name* identifying the film to be played in the cinema given by *cinema name* on the day stated by *day name* and last the *number of seats* to be booked

Sample Program Execution in Command Line Interface

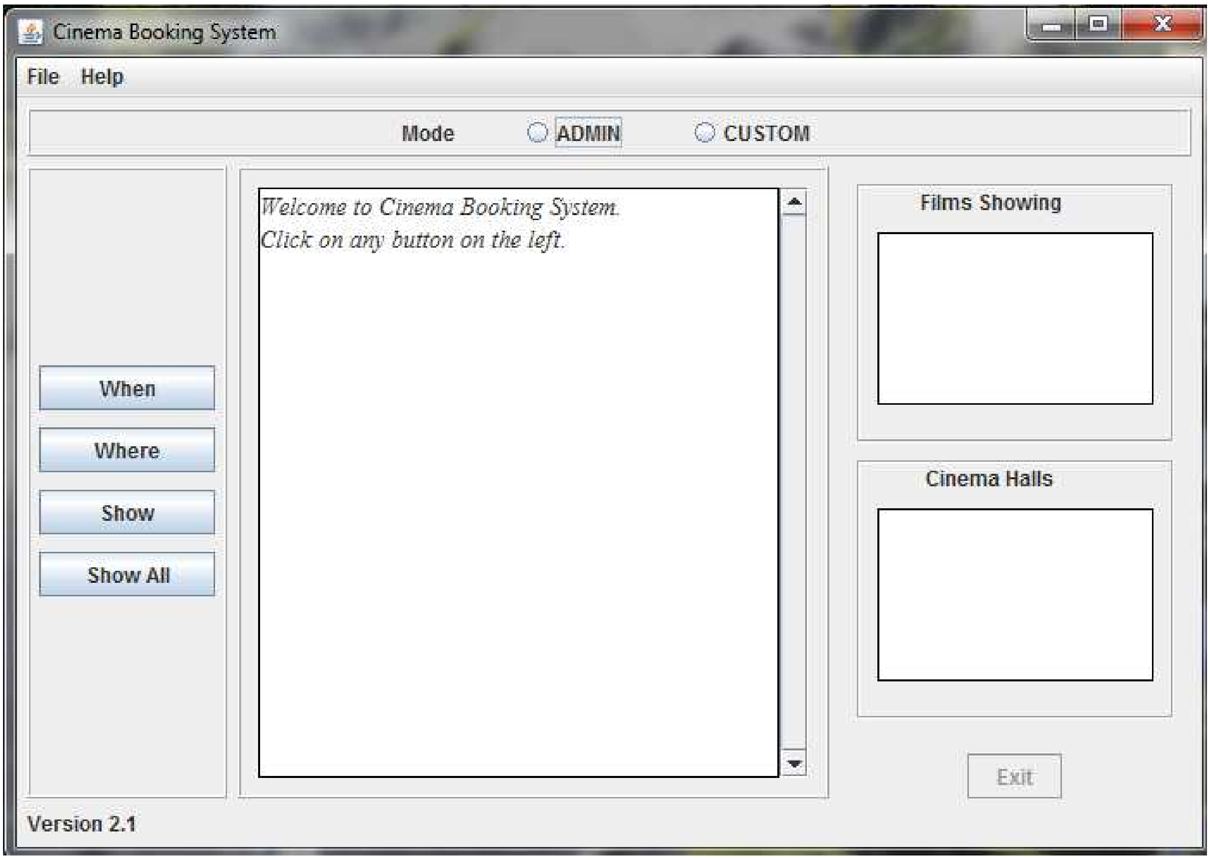
```
Welcome to the Cinema Seats Booking System!
Please enter your command...

>mode ADMIN                                //command to change mode
Mode changed to ADMIN               //status of change of mode
>addFrom addFromFile.txt            //addFrom filename
CINEMA_HALL_1 added to the system.
CINEMA_HALL_2 added to the system.
FILM_1 added to CINEMA_HALL_1 on SUN.
FILM_1 added to CINEMA_HALL_1 on MON.
FILM_2 added to CINEMA_HALL_2 on WED.
FILM_2 added to CINEMA_HALL_2 on TUE.
>addCinema Cinema_Hall_3 25 20             //addCinema
CINEMA_HALL_3 added to the system.
>addFilm Film_3 Cinema_Hall_3 Thu Fri    //addFilm
FILM_3 added to CINEMA_HALL_3 on FRI.
FILM_3 added to CINEMA_HALL_3 on THU.
>mode CUSTOM                        //command to change mode
Mode changed to CUSTOM
>when Film_2                        //when command
TUE WED
>where Film_2 Tue                          //where command
CINEMA_HALL_2
>Book Fariha_Nazmul Film_2 Cinema_Hall_2 Tue 2       //book command
Successfully done.                                           //status
Booking Number: 1                                                    //booking information
Person Name: FARIHA_NAZMUL
Film Name: FILM_2
Cinema Name: CINEMA_HALL_2
Day of Show: TUE
Reserved Seats: 1A 1B
```

```
>show 1                             //show booking_number
Booking Number: 1                   //booking information
Person Name: FARIHA_NAZMUL
Film Name: FILM_2
Cinema Name: CINEMA_HALL_2
Day of Show: TUE
Reserved Seats: 1A 1B
>showAll Fariha_Nazmul        //showAll
Booking Number: 1                   //booking information
Person Name: FARIHA_NAZMUL
Film Name: FILM_2
Cinema Name: CINEMA_HALL_2
Day of Show: TUE
Reserved Seats: 1A 1B
>mode ADMIN                         //change of mode
Mode changed to ADMIN
>save Data.txt                //save system
Saved to file: Data.txt
>Load Data.txt                //load system
Successfully loaded.
>exit                               //exit system
Thank you. Goodbye!
```

Visual Representation of the GUI

Here is the screenshot of the GUI to show how it looks:



Welcome to Cinema Booking System.
Click on any button on the left.

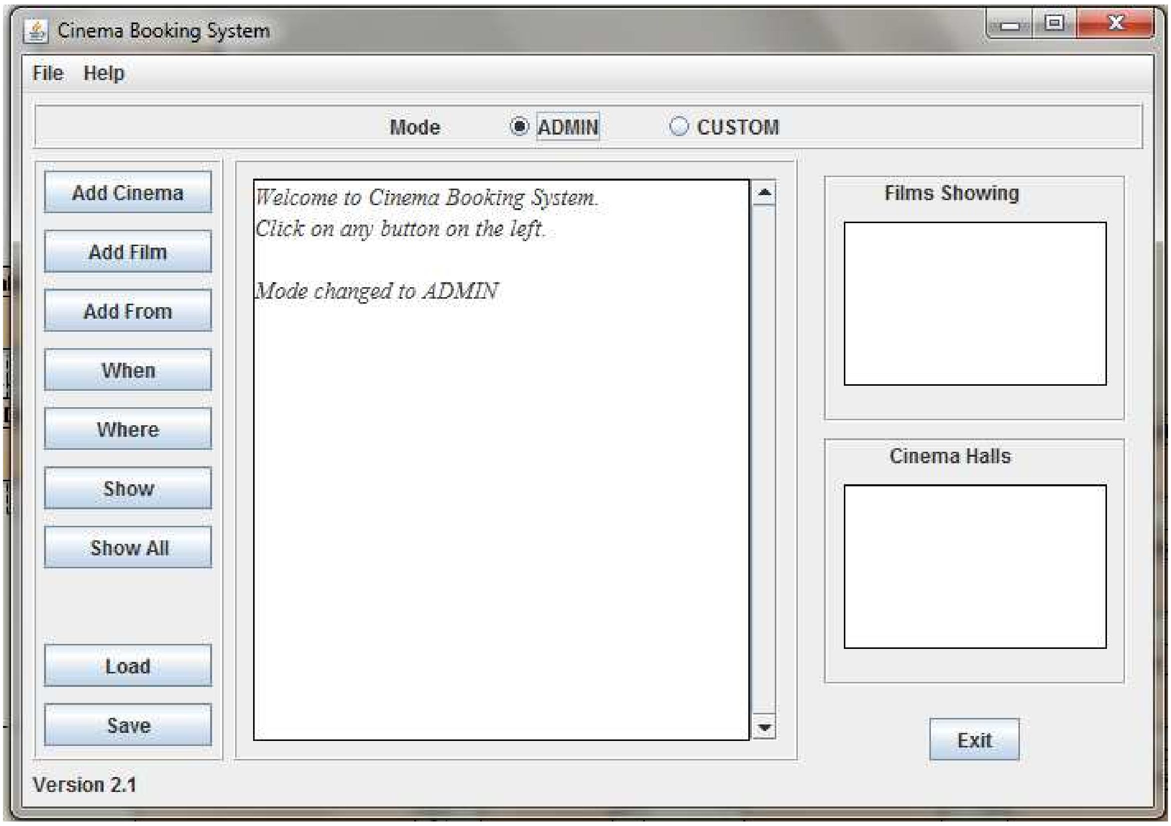
Figure 4: GUI screenshot

- On the top panel, user has the option to switch between two modes. Depending upon the modes the buttons and menu items are shown.
- First when the system is run, there is no mode specified. So only the common actions that can be performed by both users are shown as active.
- On the left panel, the buttons represent each of the functions that can be performed on the system. They are the same as the commands mentioned above. Clicking on any button will open a popup window with text fields to collect user input. If the user gives correct inputs and clicks OK button only then it will be executed.
- On the right panel, the films that are currently showing and are in the database of the system are shown. Also the existing Cinema Halls names are shown in the bottom list.
- In the middle panel, is the output terminal. Here the user can see the status of performing each action and also the output of the actions.
- On the left top, there is the file menu. Only the administrator can perform the File menu operations. Help menu is accessible to both user and it only shows information about the system version.
- On the bottom right, there is the Exit button. Only administrator has the right to exit the application.

Sample Program Execution in GUI

If the ADMIN mode is selected than the operations that are valid for ADMIN are shown on the screen. The ADMIN can then choose to run any of the operations. In the following two screenshots I have shown the GUI when ADMIN mode is chosen and the ADMIN runs the addCinema and load operation.

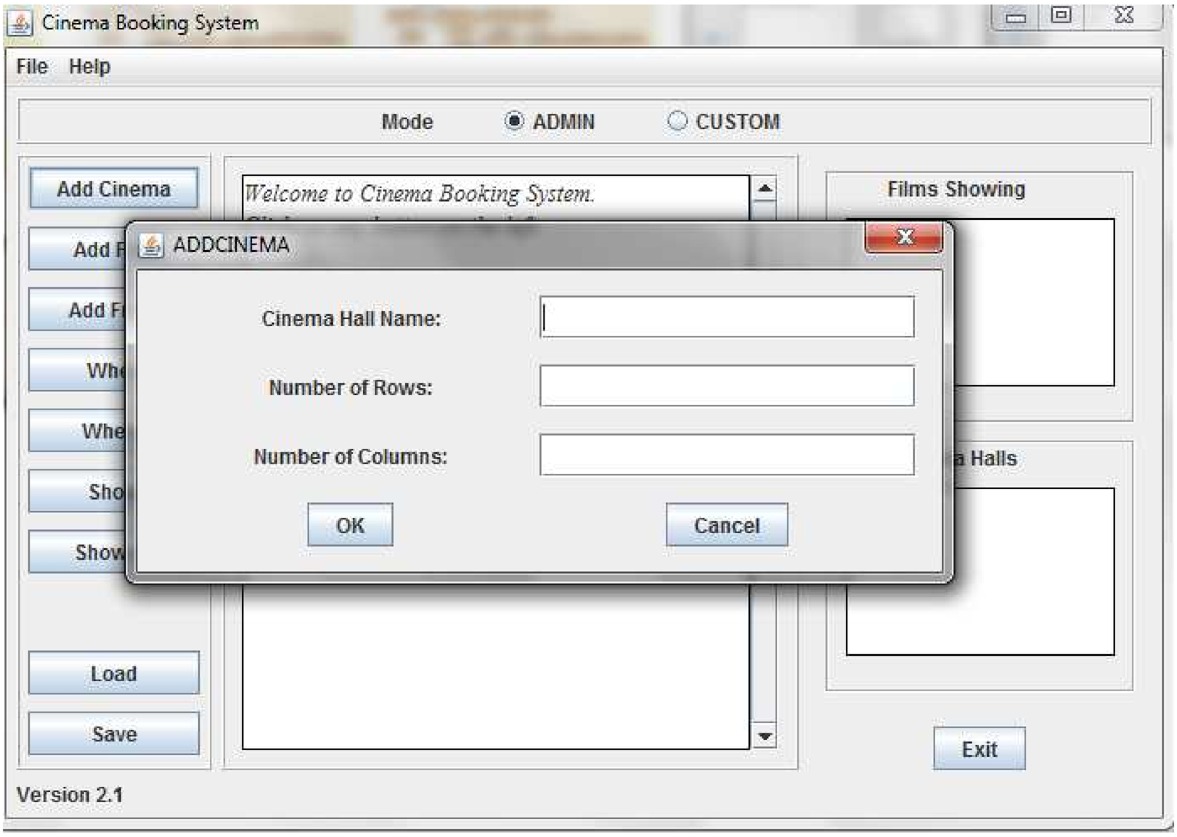
Then the Figures show when CUSTOM mode is selected and the CUSTOM runs the book operation.



Welcome to Cinema Booking System.
Click on any button on the left.

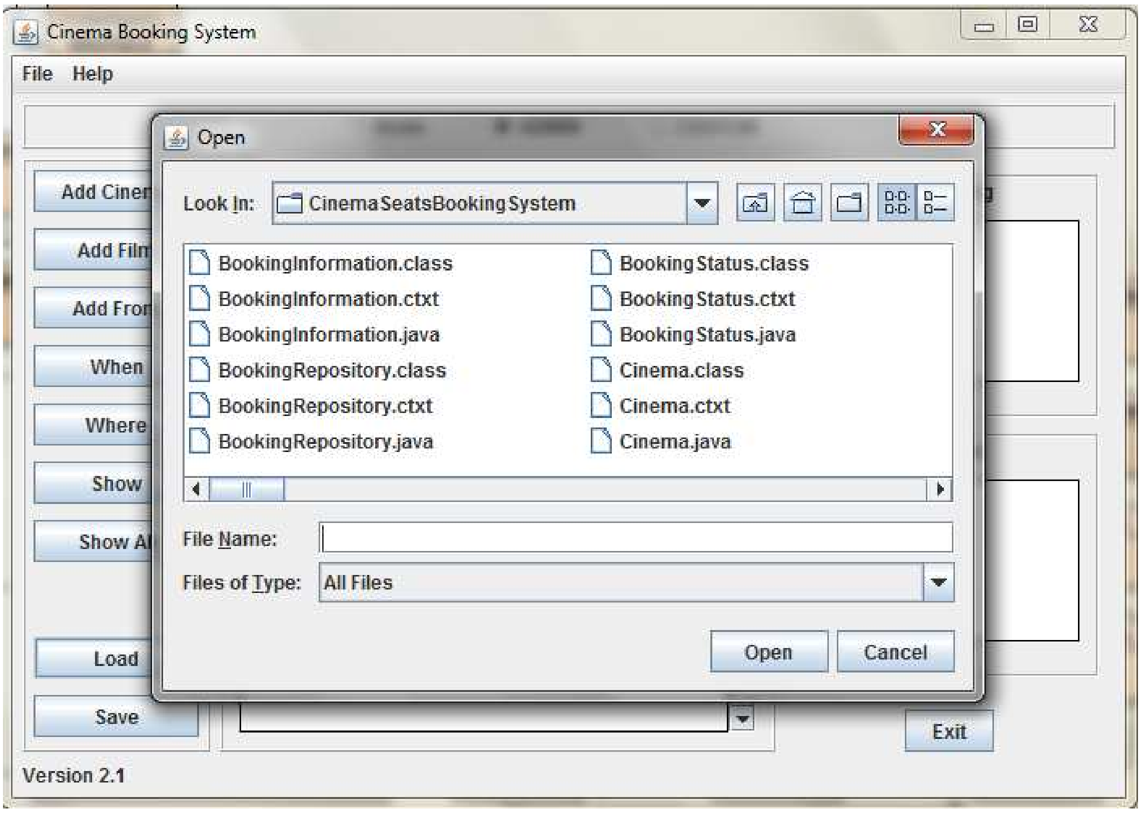
Mode changed to ADMIN

Figure 5: ADMIN mode seletcted



Welcome to Cinema Booking System.

Figure 6: ADMIN runs the addCinema operation



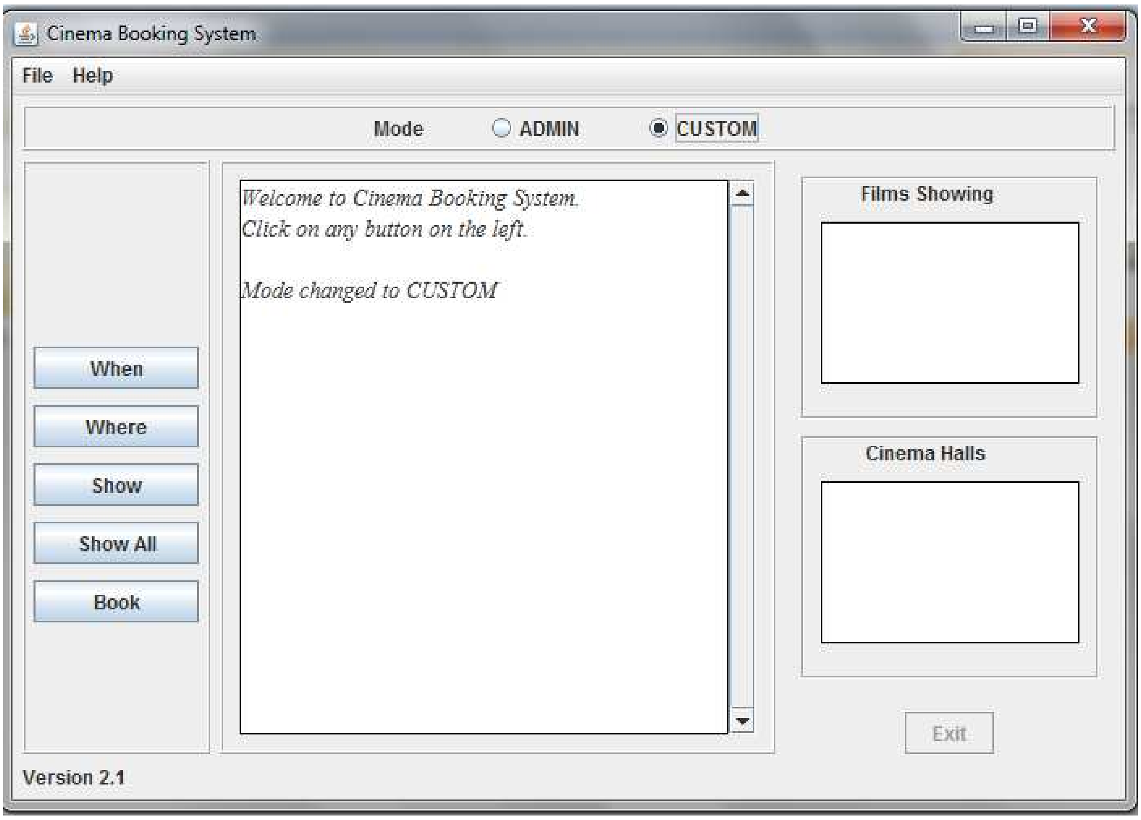
Look In: CinemaSeatsBookingSystem

BookingInformation.class
BookingInformation.ctxt
BookingInformation.java
BookingRepository.class
BookingRepository.ctxt
BookingRepository.java
BookingStatus.class
BookingStatus.ctxt
BookingStatus.java
Cinema.class
Cinema.ctxt
Cinema.java

File Name:

Files of Type: All Files

Figure 7: ADMIN runs the Load operation

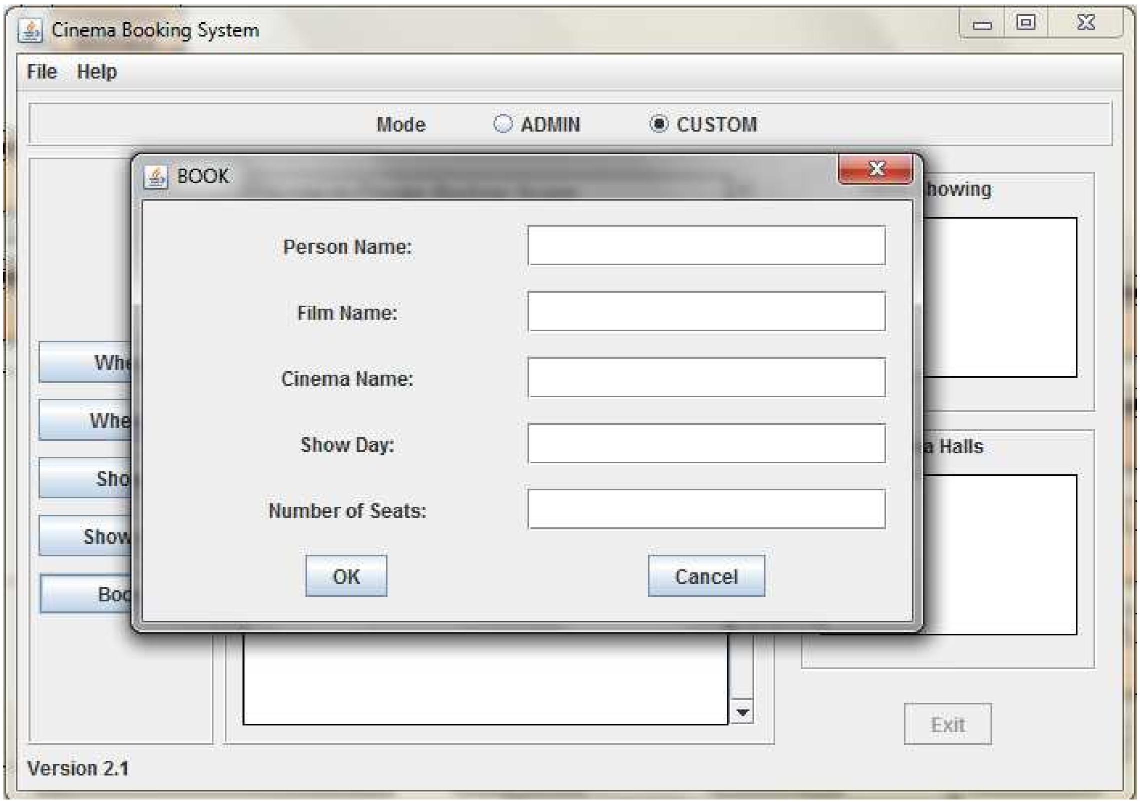


Mode ○ ADMIN ● CUSTOM

Welcome to Cinema Booking System.
Click on any button on the left.

Mode changed to CUSTOM

Figure 8: CUSTOM mode is selected



Cinema Booking System

File Help

Mode ○ ADMIN ● CUSTOM

BOOK

Person Name:

Film Name:

Cinema Name:

Show Day:

Number of Seats:

Figure 9: CUSTOM runs book operation

Appendix A
Program Listing

CinemaSystem.java

```
/**
 * Provides the interfaces of a simple Cinema Seats Booking System.
 *
 * @author Fariha Nazmul
 * @version 01.11.10
*/
public class CinemaSystem
{
   /**
    *The main function to run the command line interface
    * @param args String "GUI" to run the GUI interface,
    *              Runs the Console if no parameter is given
    */

   public static void main(String[] args){
      //the cinema system engine
      CinemaSystemEngine cinemaSystem = new CinemaSystemEngine();

      Interface interaction;

      if(args.length!=0 && args[0].equalsIgnoreCase("GUI"))
     //starts the GUI
         interaction = new GraphicalInterface(cinemaSystem);
      else
         //starts the command line interface
         interaction = new CommandLineInterface(cinemaSystem);
      interaction.run();
   }

}
```

Interface.java

```
/**
 * The interface to be extended by any class wishing to
 * provide any form of user interface of the system.
 *
 */

/**
 *
 * @author Fariha Nazmul
 * @version 01.11.10
 */
```

```
public interface Interface {  void run();}
```

CommandLineInterface.java

```
import java.io.IOException;

/**
 * Provides a command line interface to the Cinema Booking System.
 * Different commands provide access to the data in the system.
 *
 * @author Fariha Nazmul
 * @version 01.11.10
 */
public class CommandLineInterface implements Interface

{
    // The cinema booking system to be viewed and manipulated.
    private CinemaSystemEngine cinemaSystem;
    // A parser for handling user commands.
    private Parser parser;
    // A file handler for the IO operations
    private FileHandler fileHandler;

    /**
     * Constructor for objects of class CommandLineInterface
     * @param cinemaSystem The system to be used.
     *
     */
    public CommandLineInterface(CinemaSystemEngine cinemaSystem)
    {
        this.cinemaSystem = cinemaSystem;
        parser = new Parser();
        fileHandler = new FileHandler(this.cinemaSystem);
    }


    /**
     * Reads a series of commands from the user to interact
     * with the cinema booking system.
     * Stops only when the admin types 'exit'.
     */
    public void run()
    {
        printWelcome();
        // Enters the main command loop.
        //Here we repeatedly read commands and
        // execute them until admin wants to exit.

        boolean finished = false;
        while (! finished) {
```

```
    System.out.print(">");
    Command command = parser.getCommand();

    try {
      finished = processCommand(command);
    } catch (IllegalArgumentException ex) {
      System.out.println(ex.getMessage());
    } catch (UnrecognizedCommandException ex) {
      System.out.println(ex.toString());
    } catch (InvalidCommandException ex) {
      System.out.println(ex.toString());
    } catch (ClassNotFoundException ex) {
      System.out.println(ex.toString());
    } catch (IOException ex) {
      System.out.println(ex.toString());
    }
  }
  System.out.println("Thank you. Goodbye!");
}

 /**
 * Print out the opening message for the user.
 */
private void printWelcome()
{
  System.out.println();
  System.out.println("Welcome to the Cinema Seats Booking System!");
  System.out.println("Please enter your command...");
  System.out.println();
}


/**
 * Given a command, processes the command
 * and performs actions accordingly.
 * @param command The command to be processed.
 * @return true if the command quits the system, false otherwise.
 */
public boolean processCommand(Command command)
    throws IllegalArgumentException, UnrecognizedCommandException,
    InvalidCommandException, IOException, ClassNotFoundException
{
  boolean wantToExit = false;

  //check if the command is valid and
  //also the user mode is legal to run the command
  if(cinemaSystem.checkCommand(command, cinemaSystem.getMode())){

    CommandWord commandWord = command.getCommandWord();
    //choose the functions according to the commands
```

```
            switch(commandWord){
                case MODE:
                    System.out.println(cinemaSystem.setMode(command));
                    break;
                case ADD_CINEMA:
                    System.out.println(cinemaSystem.addCinema(command));
                    break;
                case ADD_FILM:
                    System.out.println(cinemaSystem.addFilm(command));
                    break;
                case ADD_FROM:
                    System.out.println(fileHandler.addFrom(command));
                    break;
                case LOAD:
                {this.cinemaSystem = fileHandler.load(command);
                 System.out.println("Successfully loaded.");
                 break;}
                case SAVE:
                    System.out.println(fileHandler.save(command));
                    break;
                case WHEN:
                    System.out.println(cinemaSystem.when(command));
                    break;
                case WHERE:
                    System.out.println(cinemaSystem.where(command));
                    break;
                case BOOK:
                    System.out.println(cinemaSystem.book(command));
                    break;
                case SHOW:
                    System.out.println(cinemaSystem.show(command));
                    break;
                case SHOW_ALL:
                    System.out.println(cinemaSystem.showAll(command));
                    break;
                case EXIT:
                    wantToExit = true;break;
                default: break;
            }
            // else command not recognised.
        }
        return wantToExit;
    }

}
```

Parser.java

```
import java.io.Serializable;
import java.util.Scanner;
import java.util.ArrayList;

/**
 * This parser reads user input and tries to interpret it as a valid command.
 * Each time it reads a line from the terminal and
 * tries to interpret the line as a command with two parts i.e.
 * a valid commandString and a list of parameters.
 * It returns the command as an object of class Command.
 *
 * The parser has a set of known command words. It checks user input against
 * the known commands, and if the input is not one of the known commands, it
 * returns a command object that is marked as an unknown command.
 *
 * @author  Fariha Nazmul
 * @version 01.11.10
 */

public class Parser implements Serializable
{
    private CommandWords commands;  // holds all valid command words
    private Scanner reader;         // source of command input

    /**
     * Creates a parser to read from the terminal window.
     */
    public Parser()
    {
        commands = new CommandWords();
        reader = new Scanner(System.in);
    }


    /**
     * Returns the next command from the user.
     * @return The next command from the user.
     */
    public Command getCommand()
    {
        String inputLine;   // will hold the full input line
        String commandString = null;
        ArrayList<String> parameterList = new ArrayList<String>();

        inputLine = reader.nextLine();

        // Finds the parts of the command in the line.
        Scanner tokenized = new Scanner(inputLine);
        if(tokenized.hasNext()) {
            // the commad string
            commandString = tokenized.next().toLowerCase();
            // the list of parameters
            while(tokenized.hasNext()) {
              parameterList.add(tokenized.next());
            }
        }

        return new Command(commands.getCommandWord(commandString),parameterList);
    }

}
```

Command.java

```
import java.io.Serializable;
import java.util.ArrayList;
/**
 * This class holds information about a command issued by the user.
 * Each command has two parts:
 *     a CommandWord and a list of parameters
 * If the command had only one word, then the parameter list is empty.
 *
 * @author  Fariha Nazmul
 * @version 01.11.10
 */

public class Command implements Serializable
{
    private CommandWord commandWord;
    private ArrayList<String> parameterList;

    /**
     * Create a command object.
     * @param commandWord The CommandWord.
     * @param parameterList The list of parameters passed to the command.
     * It can also be empty.
     */
    public Command(CommandWord commandWord, ArrayList<String> parameterList)
    {
        this.commandWord = commandWord;
        this.parameterList = new ArrayList<String>();
        this.parameterList = parameterList;
    }

    /**
     * Returns   The command word of this command.
     * @return  The command word.
     */
    public CommandWord getCommandWord()
    {
        return commandWord;
    }

    /**
     * Returns the list of parameters of the command.
     * @return The parameter list of this command.
     */
    public ArrayList<String> getParameterList()
    {
        return parameterList;
    }

    /**
     * Checks if the command is Unknown.
     * @return true if this command is unknown, false otherwise.
     */
    public boolean isUnknown()
    {
        return (commandWord == CommandWord.UNKNOWN);
    }

    /**
     * Checks if the command has any parameters.
     * @return true if the parameter list is not empty, false otherwise.
     */
    public boolean hasParameters()
    {
        return (parameterList.size() != 0);
    }
}
```

CommandWords.java

```
import java.io.Serializable;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;

/**
 * This class holds an enumeration of all command words
 * known to the Cinema Booking System.
 * It is used to recognise commands as they are typed in.
 * It also to checks the legal mode of that command.
 *
 * @author  Fariha Nazmul
 * @version 01.11.10
 */

public class CommandWords implements Serializable
{
   // Mappings between a command word and the CommandWord
   // associated with it in different user mode.
   private HashMap<String, CommandWord> validAllCommands;
   // A mapping between a command and its valid users
   private HashMap<CommandWord, ArrayList<User>> legalUserForCommand;

   /**
    * Constructor - initialises the command words.
    */
   public CommandWords()
   {
      validAllCommands = new HashMap<String, CommandWord>();

      for(CommandWord command : CommandWord.values()) {
         if(command != CommandWord.UNKNOWN) {
            validAllCommands.put(command.toString(), command);
         }
      }
      // set the mapping of command with its legal mode
      this.setLegalUser();
   }


   /**
    * Creates the mapping of command and its legal mode.
    */
   public void setLegalUser(){

      legalUserForCommand = new HashMap<CommandWord, ArrayList<User>>();
      //initialize all commands with empty user
      for (CommandWord command: CommandWord.values())
         legalUserForCommand.put(command, new ArrayList<User>());

      // set mapping for each command
      legalUserForCommand.get(CommandWord.ADD_CINEMA).add(User.ADMIN);
      legalUserForCommand.get(CommandWord.ADD_FILM).add(User.ADMIN);
      legalUserForCommand.get(CommandWord.ADD_FROM).add(User.ADMIN);
      legalUserForCommand.get(CommandWord.LOAD).add(User.ADMIN);
      legalUserForCommand.get(CommandWord.SAVE).add(User.ADMIN);
      legalUserForCommand.get(CommandWord.EXIT).add(User.ADMIN);
      legalUserForCommand.get(CommandWord.BOOK).add(User.CUSTOM);
      legalUserForCommand.get(CommandWord.MODE).addAll(
            Arrays.asList(User.values()));
      legalUserForCommand.get(CommandWord.WHEN).addAll(
            Arrays.asList(User.values()));
      legalUserForCommand.get(CommandWord.WHERE).addAll(
            Arrays.asList(User.values()));
      legalUserForCommand.get(CommandWord.SHOW).addAll(
            Arrays.asList(User.values()));
      legalUserForCommand.get(CommandWord.SHOW_ALL).addAll(
            Arrays.asList(User.values()));
   }

   /**
    * Finds the CommandWord associated with a command word.
    * @param commandWord The word to look up.
    * @return The CommandWord correspondng to commandWord, or UNKNOWN
    *         if it is not a valid command word.
    */
   public CommandWord getCommandWord(String commandWord)
   {
      CommandWord command = validAllCommands.get(commandWord);
      if(command != null) {
         return command;
      }
      else {
         return CommandWord.UNKNOWN;
      }
   }

   /**
    * Checks whether a given String is a valid command word for the system.
    * @param command The command string
    * @return true if it is, false otherwise.
    */
   public boolean isCommand(String command)
   {
      return validAllCommands.containsKey(command);
   }

    /**
    * Checks whether a given command is a valid
    * command word for a given user mode in the system.
    * @param command   The command word
    * @param mode      The mode of the user
    * @return true if it is, false otherwise.
    */

   public boolean isValid(CommandWord command, User mode)
   {
      if(legalUserForCommand.containsKey(command))
         return legalUserForCommand.get(command).contains(mode);
      else
         return false;
   }
}
```

CommandWord.java

```
import java.io.Serializable;

/**
 * Representations for all the valid command words in the Cinema Booking System
 * along with their number of parameters.
 * Each command word is associated with a string in a particular language
 * and the minimum number of parameters needed for that command.
 *
 * @author Fariha Nazmul
 * @version 01.11.10
 */
public enum CommandWord implements Serializable
{
    // A value for each command word along with its
    // corresponding user interface string and number of parameters.

    MODE("mode", 1),
    ADD_CINEMA("addcinema", 3),
    ADD_FILM("addfilm", 3),
    ADD_FROM("addfrom", 1),
    LOAD("load", 1),
    SAVE("save", 1),
    WHEN("when", 1),
    WHERE("where", 2),
    BOOK("book", 5),
    SHOW("show", 1),
    SHOW_ALL("showall", 1),
    EXIT("exit", 0),
    UNKNOWN("unknown",0);

    // The command as string.
    private String commandString;
    // The number of parameters
    private int numOfParam;

    /**
     * Initialise with the corresponding command word and num of parameter.
     * @param commandWord The command string.
     * @param numOfParam The valid number of paramters for the command.
     */

    CommandWord(String commandString, int numOfParam)
    {
        this.commandString = commandString;
        this.numOfParam = numOfParam;
    }
    /**
     * Returns the string representation of the command word.
     * @return The command word as a string.
     */
    public String toString()
    {
        return commandString;
    }


    /**
     * Returns the number of parameters for the command.
     * @return The num of parameters needed for the command.
     */
    public int getNumOfParam()
    {
        return numOfParam;
    }

}
```

CinemaSystemEngine.java

```
import java.io.Serializable;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

/**
 * Provides the engine of a simple Cinema Seats Booking System.
 *
 * @author Fariha Nazmul
 * @version 01.11.10
 */
public class CinemaSystemEngine implements Serializable {

    private User mode;              //the user mode
    private DayParameters dayParams;    //the mapping of enumeration of days
    private CommandWords commandWords;  //the mapping of enumeration of commands
    private CinemaRepository cinemaRepository;     //the collection of cinemas
    private FilmShowRepository filmShowRepository; //the collection of filmShows
    private BookingRepository bookingRepository;  //the collection of bookings

    /**
     * Allows the user to interact with the cinema booking system.
     */
```

```
public CinemaSystemEngine() {
   mode = User.USER;
   dayParams = new DayParameters();
   commandWords = new CommandWords();
   cinemaRepository = new CinemaRepository();
   filmShowRepository = new FilmShowRepository();
   bookingRepository = new BookingRepository();
}

/**
 * Returns the mode of the user.
 * @return The user mode
 */
public User getMode() {
   return mode;
}

/**
 * Sets the mode of the user.
 * @param command The command
 * @return The string representing the status of mode change
 * @throws IllegalArgumentException when proper parameter is not passed
 */
public String setMode(Command command) throws IllegalArgumentException {
   String param = null;
   User user = User.USER;
   UserParameters allUsers = new UserParameters();

   param = command.getParameterList().get(0).toLowerCase();
   user = allUsers.getUser(param);

   if ((param != null) && allUsers.isUser(param)) {
      this.mode = user;
      return "Mode changed to " + mode.toString().toUpperCase();
   } else {
      throw new IllegalArgumentException("Correct mode is not " +
            "specified in MODE command.");
   }
}

/**
 * Adds a Cinema in the Cinema Repositoy.
 * @param command The command
 * @return The string representing the status of adding cinema
 * @throws NullPointerException when null is passed
 * @throws IllegalArgumentException when proper parameter is not passed
 */
public String addCinema(Command command) throws NullPointerException,
      IllegalArgumentException {
   String cinemaName = null;
```

```
  int rowNum = 0, rowLength = 0;

  //getting all parameters
  cinemaName = command.getParameterList().get(0).toLowerCase();
  rowNum = Integer.parseInt(command.getParameterList().get(1));
  rowLength = Integer.parseInt(command.getParameterList().get(2));

  //checking each parameter
  if (cinemaName != null && rowNum > 0 && rowNum <= 30 && rowLength > 0
      && rowLength <= 20) {
    Cinema tempCinema = new Cinema(cinemaName, rowNum, rowLength);
    return cinemaRepository.addCinema(tempCinema);
  } else {
    throw new IllegalArgumentException("Correct argument not " +
        "specified in addCinema command.");
  }
}

/**
 * Adds a Film show in the FilmShow Repository.
 * @param command The command
 * @return The string representing the status of adding films show
 * @throws NullPointerException when null is passed
 * @throws IllegalArgumentException when proper parameter is not passed
 */
public String addFilm(Command command) throws NullPointerException,
    IllegalArgumentException {
  Set<Day> days = new HashSet<Day>();
  String filmName = null, cinemaName = null;

  StringBuffer result = new StringBuffer();

  //getting all parameters
  filmName = command.getParameterList().get(0).toLowerCase();
  cinemaName = command.getParameterList().get(1).toLowerCase();
  for (int i = 2; i < command.getParameterList().size(); i++) {
   if(dayParams.isDay(command.getParameterList().get(i).toLowerCase())){
     days.add(dayParams.getDay(
         command.getParameterList().get(i).toLowerCase()));
   }
  }

  //checking each parameter
  if (filmName != null && cinemaName != null && days.size() != 0) {
    if (cinemaRepository.isCinema(cinemaName)) {
      for (Iterator it = days.iterator(); it.hasNext();) {
        FilmShow filmShow = new FilmShow(filmName,
            cinemaName, ((Day) (it.next())));
        result.append(filmShowRepository.addFilmShow(filmShow));
        result.append("\n");
      }
    } else {
      result.append("No Cinema Found:" + "\n" +
          cinemaName.toUpperCase() + " does not exist.");
    }
    return result.toString();
  } else {
    throw new IllegalArgumentException("Correct argument not " +
        "specified in addFilm command.");
  }
}

/**
 * Returns the days when a film is shown.
 * @param command The command
 * @return The string representing the days
 * @throws NullPointerException when null is passed
 * @throws IllegalArgumentException when proper parameter is not passed
 */
public String when(Command command) throws NullPointerException,
    IllegalArgumentException {
  String filmName = null;

  filmName = command.getParameterList().get(0).toLowerCase();

  if (filmName != null) {
    return filmShowRepository.getWhenFilmShow(filmName);
  } else {
    throw new IllegalArgumentException("Correct argument not " +
        "specified in WHEN command.");
  }
}

/** Returns the cinema where a film is shown on a day.
 * @param command The command
 * @return The string representing the cinemas
 * @throws NullPointerException when null is passed
 * @throws IllegalArgumentException when proper parameter is not passed
 */
public String where(Command command) throws NullPointerException,
    IllegalArgumentException {
  String filmName = null;
  Day day = null;

  filmName = command.getParameterList().get(0).toLowerCase();
  if (dayParams.isDay(command.getParameterList().get(1).toLowerCase())) {
    day = dayParams.getDay(
        command.getParameterList().get(1).toLowerCase());
  }
  if (filmName != null && day != null) {
    return filmShowRepository.getWhereFilmShow(filmName, day);
  } else {
    throw new IllegalArgumentException("Correct argument not " +
        "specified in WHERE command.");
  }
}

/** Books the seats for a certain film show.
 * @param command The command
 * @return The string representing the status of booking
 * @throws NullPointerException when null is passed
 * @throws IllegalArgumentException when proper parameter is not passed
 */
public String book(Command command) throws IllegalArgumentException {
  String personName = null, filmName = null, cinemaName = null;
  Day showDay = null;
  int numOfSeats = 0;

  //getting all the parameters
  personName = command.getParameterList().get(0).toLowerCase();
  filmName = command.getParameterList().get(1).toLowerCase();
  cinemaName = command.getParameterList().get(2).toLowerCase();
  numOfSeats = Integer.parseInt(command.getParameterList().get(4));
  if (dayParams.isDay(command.getParameterList().get(3).toLowerCase())) {
    showDay = dayParams.getDay(
        command.getParameterList().get(3).toLowerCase());
  }

  //checking each parameter
  if (personName != null && filmName != null && cinemaName != null
      && showDay != null && numOfSeats > 0) {
    FilmShow filmShow = new FilmShow(filmName, cinemaName, showDay);
    if (filmShowRepository.isFilmShow(filmShow)) {
      ArrayList<Seat> seats =
          cinemaRepository.bookSeat(filmShow,numOfSeats);
      if (seats != null) {
        //add booking info
        bookingRepository.addBooking(new BookingInformation(
            personName, filmShow, seats));
        return new String("Successfully done."
            + bookingRepository.showBooking(
              bookingRepository.getBookingNumber()));
      } else {
        return new String("Not enough free seats are available.");
      }
    } else {
      return new String("No show Exists: " + filmName.toUpperCase() +
          " " + cinemaName.toUpperCase() + " "
          + showDay.toString().toUpperCase());
    }

  } else {
    throw new IllegalArgumentException("Correct argument not " +
        "specified in BOOK command.");
  }

}

/** Shows a booking information given a booking number.
 * @param command The command
 * @return The string representing the booking information
 * @throws NullPointerException when null is passed
 * @throws IllegalArgumentException when proper parameter is not passed
 */
public String show(Command command)
{
  int bookingNum = 0;

  bookingNum = Integer.parseInt(command.getParameterList().get(0));
  return bookingRepository.showBooking(bookingNum);
}

/** Shows all booking information given a person name.
 * @param command The command
 * @return The string representing all the booking informations
 * @throws NullPointerException when null is passed
 * @throws IllegalArgumentException when proper parameter is not passed
 */
public String showAll(Command command) {
  String personName = null;

  personName = command.getParameterList().get(0).toLowerCase();
  return bookingRepository.showAllBooking(personName);
}

/**
 * Checks the legal mode of the command against the current user mode.
 * @param command The string of the command to be checked
 * @param userMode The mode of the user
 * @return true if the command is valid for the mode,
 * or throws exception otherwise.
 */
public boolean checkCommand(Command command, User userMode)
    throws UnrecognizedCommandException, InvalidCommandException,
    IllegalArgumentException {

  // checks the command is a valid command of the system
  if (!commandWords.isCommand(command.getCommandWord().toString())) {
    throw new UnrecognizedCommandException(
            command.getCommandWord().toString());
  }

  // checks the legal mode of command
  if (!commandWords.isValid(command.getCommandWord(), userMode)) {
    throw new InvalidCommandException(
        command.getCommandWord().toString());
  }

  // checks the number of parameters
  if (command.getParameterList().size() <
            command.getCommandWord().getNumOfParam()) {
    throw new IllegalArgumentException(
        "Incorrect number of arguments for command: "
        + command.getCommandWord().toString().toUpperCase());
  }

  return true;
}

/**
 * Returns the list of all film names in the system.
 * @return the list of films
 */
public ArrayList<String> getAllFilms(){

  ArrayList<String> filmList = new ArrayList<String>();
  filmList = filmShowRepository.getAllFilms();
  return filmList;
}

/**
 * Returns the list of all cinema names in the system.
 * @return the list of cinemas
 */
public ArrayList<String> getAllCinemas(){

  ArrayList<String> cinemaList = new ArrayList<String>();
  cinemaList = cinemaRepository.getAllCinemas();
  return cinemaList;
}
}
```

FileHandler.java

```
import java.io.*;
import java.util.ArrayList;
import java.util.Scanner;

/**
 * Provides the file-handling operations on an CinemaSystemEngine.
 *
 * @author Fariha Nazmul
 * @version 01.11.10
 */
public class FileHandler implements Serializable {
  // The system on which i/o operations are performed.

  private CinemaSystemEngine cinemaSystem;

  /**
   * Constructor for objects of class FileHandler.
   * @param cinemaSystem The cinemaSystem to use.
   */
  public FileHandler(CinemaSystemEngine cinemaSystem) {
    this.cinemaSystem = cinemaSystem;
  }

  /**
   * Reads and runs addCinema and addFilm commands from a text file.
   * The file is assumed to contain one command with parameters
   * separated by space in each line.
   * @param command The command with its parameters
   * @throws IOException On input failure.
   * @throws IllegalArgumentException On incorrect parameters of the command
   * @throws InvalidCommandException  On passing command not valid in the mode
   * @throws UnrecognizedCommandException On passing Unknown command
   */
  public String addFrom(Command command)
        throws IOException, IllegalArgumentException,
        InvalidCommandException, UnrecognizedCommandException {
    String inputLine;
    Command commandInFile;
    ArrayList<String> parameterList;
    Scanner reader, tokenized;

    StringBuffer result = new StringBuffer();

    CommandWords commands = new CommandWords();
    String fileName = null, commandString = null;
    if (command.getParameterList().size()
          == command.getCommandWord().getNumOfParam()) {
      fileName = command.getParameterList().get(0);
    }
    if (fileName != null) {
      reader = new Scanner(new File(fileName));

      do {
        inputLine = reader.nextLine();
        // Find the parts of the command in the line.
        tokenized = new Scanner(inputLine);
        if (tokenized.hasNext()) //the command string
        {
          commandString = tokenized.next().toLowerCase();
        }
        parameterList = new ArrayList<String>();
        while (tokenized.hasNext()) {    //the parameters
          parameterList.add(tokenized.next());
        }
        commandInFile = new Command(commands.getCommandWord(
              commandString), parameterList);

        //execute the commands
        if (cinemaSystem.checkCommand(commandInFile,
              cinemaSystem.getMode())) {
          switch (commandInFile.getCommandWord()) {
            case ADD_CINEMA:
              result.append(cinemaSystem.addCinema(commandInFile));
              result.append("\n");
              break;
            case ADD_FILM:
              result.append(cinemaSystem.addFilm(commandInFile));
              result.append("\n");
              break;
            default:
              break;
          }
        } else {
          throw new InvalidCommandException(
              commandInFile.getCommandWord().toString());
        }
      } while (reader.hasNextLine());
      reader.close();
    } else {
      throw new IllegalArgumentException("Correct argument not " +
          "specified in addFrom command.");
    }
    return result.toString();
  }
```

```
/**
 * Reads the binary version of an Cinema Seat Booking System from given file.
 * @param command The command with its parameters
 * @return The cinema system engine object.
 * @throws IllegalArgumentException On incorrect parameters of the command
 * @throws UnrecognizedCommandException On passing Unknown command
 */
public CinemaSystemEngine load(Command command) throws IOException,
        ClassNotFoundException, IllegalArgumentException {
    String sourceFile = null;

    if (command.getParameterList().size()
            == command.getCommandWord().getNumOfParam()) {
        sourceFile = command.getParameterList().get(0);
    }
    if (sourceFile != null) {
        File source = new File(sourceFile);
        ObjectInputStream is = new ObjectInputStream(
                new FileInputStream(source));
        CinemaSystemEngine savedSystem =(CinemaSystemEngine)is.readObject();
        is.close();
        return savedSystem;

    } else {
        throw new IllegalArgumentException("Correct argument not " +
                "specified in LOAD command.");
    }
}

/**
 * Saves a binary version of the CinemaSystemEngine to the given file.
 * @param command The command with its parameters
 * @throws IOException If the saving process fails for any reason.
 * @throws IllegalArgumentException On incorrect parameters of the command
 */
public String save(Command command)
        throws IOException, IllegalArgumentException {
    String destinationFile = null;

    if (command.getParameterList().size()
            == command.getCommandWord().getNumOfParam()) {
        destinationFile = command.getParameterList().get(0);
    }
    if (destinationFile != null) {
        File destination = new File(destinationFile);
        ObjectOutputStream os = new ObjectOutputStream(
                new FileOutputStream(destination));
        os.writeObject(cinemaSystem);
        os.close();
```

```
            return new String("Saved to file: " + destinationFile);
        } else {
            throw new IllegalArgumentException("Correct argument not " +
                    "specified in SAVE command.");
        }
    }
}
```

FilmShowRepository.java

```
import java.io.Serializable;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

/**
 * A class to maintain an arbitrary number of film shows.
 *
 * @author Fariha Nazmul
 * @version 01.11.10
 */
public class FilmShowRepository implements Serializable
{
    // Storage for an arbitrary number of film shows.
    private ArrayList<FilmShow> allFilmShow;

    /**
     * Performs initialization for the film show collection.
     */
    public FilmShowRepository()
    {
        allFilmShow = new ArrayList<FilmShow>();
    }

    /**
     * Checks if a filmshow already exists.
     * @param filmShow  The film show object
     * @return true if the film show exists, else false
     */
    public boolean isFilmShow(FilmShow filmShow)
    {
        for(Iterator it = allFilmShow.iterator(); it.hasNext();){
                if(((FilmShow)(it.next())).equals(filmShow))
                    return true;
       }
       return false;
    }
```

```
/**
 * Adds a new film show to the collection.
 * @param filmShow The film show to be added
 * @return Status of the add operation
 */
public String addFilmShow(FilmShow filmShow) throws NullPointerException
{
   StringBuffer result = new StringBuffer();
   if(filmShow != null){
      //if the cinema hall is not free
      if(!this.isFreeCinema(
            filmShow.getCinemaName(),filmShow.getShowDay())){
         result.append("Cinema Hall Not Available on "
               + filmShow.getShowDay().toString().toUpperCase() +": ");
         result.append("\n" + filmShow.getCinemaName().toUpperCase()
               + " is already occupied on "
               + filmShow.getShowDay().toString().toUpperCase()+ ".");
         return result.toString();
      }
      else{   //cinema hall is free
         allFilmShow.add(filmShow);
         result.append(filmShow.getFilmName().toUpperCase()+" added to "
               + filmShow.getCinemaName().toUpperCase() + " on "
               + filmShow.getShowDay().toString().toUpperCase()+ ".");
      }
   }
   else{
      throw new NullPointerException(
            "Null value provided with addFilmShow Command.");
   }
   return result.toString();
}


/**
 * Looks up a film name and returns the
 * corresponding days the film is shown.
 * @param filmName The name of the film to be looked up.
 * @return The days on which the film is shown.
 */
public String getWhenFilmShow(String filmName) throws NullPointerException
{
   boolean foundFilm = false;
   Set<Day> matches = new HashSet<Day>();
   StringBuffer result = new StringBuffer();

   if(filmName != null){
      for(Iterator it = allFilmShow.iterator(); it.hasNext();){
         FilmShow tempFilmShow = ((FilmShow)(it.next()));
```

```
        if(tempFilmShow.getFilmName().equals(filmName)){
          matches.add(tempFilmShow.getShowDay());
          foundFilm = true;
        }
      }
      if(foundFilm){
        for(Iterator it = matches.iterator(); it.hasNext();){
          result.append(((Day)(it.next())).toString().toUpperCase());
          result.append(" ");
        }
      }
      else{
        result.append("No Film Show Found:" );
        result.append("\nThere is no show for the film: "
            + filmName.toUpperCase() + ".");
      }

    }
    else{
      throw new NullPointerException(
          "No film provided with WHEN command.");
    }
    return result.toString();
  }

  /**
   * Looks up a film name and the day of show and returns the
   * corresponding cinemas where the film is shown.
   * @param filmName The name of the film to be looked up.
   * @param day The day on which the show is
   * @return The cinemas on which the film is shown on given day.
   */
  public String getWhereFilmShow(String filmName, Day day)
      throws NullPointerException
  {
    boolean foundFilm = false;
    Set<String> matches = new HashSet<String>();
    StringBuffer result = new StringBuffer();

    if(filmName != null){
      for(Iterator it = allFilmShow.iterator(); it.hasNext();){
        FilmShow tempFilmShow = ((FilmShow)(it.next()));
        if(tempFilmShow.getFilmName().equals(filmName)
            && tempFilmShow.getShowDay().toString().equals(
            day.toString())){
          matches.add(tempFilmShow.getCinemaName());
          foundFilm = true;
        }
      }
      if(foundFilm){
         for(Iterator it = matches.iterator(); it.hasNext();){
            result.append(((String)(it.next())).toUpperCase());
            result.append(" ");
         }
      }
      else{
         result.append("No Film Show Found:" );
         result.append("\nThere is no show for the film: "
               + filmName.toUpperCase()
               + " on " + day.toString().toUpperCase() + "." );
      }
   }
   else{
      //result.append("\nIllegal Argument: No film provided" );
      throw new NullPointerException(
            "No film provided with WHERE command.");
   }
   return result.toString();
}

/**
 * Checks if a Cinema is free on a particular day
 * @param cinemaName The name of the cinema
 * @param day The day to look up
 * @return true if its free, else false
 */
public boolean isFreeCinema(String cinemaName, Day day)
{

   for(Iterator it = allFilmShow.iterator(); it.hasNext();){
      FilmShow tempFilmShow = ((FilmShow)(it.next()));
      if(tempFilmShow.getCinemaName().equals(cinemaName)
           && tempFilmShow.getShowDay().toString().equals(
           day.toString())){
            return false;
      }
   }
   return true;
}

/**
 * Returns the list of all film names in the system.
 * @return the list of films
 */
public ArrayList<String> getAllFilms(){

   ArrayList<String> filmList = new ArrayList<String>();
   for(FilmShow filmShow: allFilmShow){
      if(!filmList.contains(filmShow.getFilmName()))
         filmList.add(filmShow.getFilmName());
```

```
        }
        return filmList;
    }

}
```

FilmShow.java

```
import java.io.Serializable;


/**
 * A class to hold the information of a single filmshow
 * A film show has three attributes:
 *      The name of the film,
 *      The name of the cinema,
 *      The name of the day.
 * @author Fariha Nazmul
 * @version 01.11.10
 */
public class FilmShow implements Serializable
{
    private String filmName;    // the name of the film
    private String cinemaName;  // the name of the cinema
    private Day showDay;        // the day of the show

    /**
     * Constructor for objects of class FilmShow
     * @param filmName    the name of the film
     * @param cinemaName      the name of the cinema
     * @param showDay       the day of the show
     */
    public FilmShow(String filmName, String cinemaName, Day showDay )
    {
        // initialise instance variables
        this.filmName = filmName;
        this.cinemaName = cinemaName;
        this.showDay = showDay;
    }

    /**
     * Returns the name of the film.
     * @return     the name of the film
     */
    public String getFilmName()
    {
        return filmName;
    }
```

```
   /**
    * Returns the name of the cinema.
   * @return     the name of the cinema
   */
  public String getCinemaName()
  {
    return cinemaName;
  }

   /**
    * Returns the day of the show.
   * @return     the day of the show
   */
  public Day getShowDay()
  {
    return showDay;
  }

   /**
    * Returns the string representation of the film show.
   * @return     the string representation of the film show
   */
  public String toString()
  {
    String stringRep = "Film Name: " + filmName.toUpperCase()
        + "\n" + "Cinema Name: "+ cinemaName.toUpperCase()
        + "\n"+ "Day of Show: "+ showDay.toString().toUpperCase();
    return stringRep;
  }

  /**
   * Implements content equality for FilmShows.
   * @return true if this FilmShow matches the other,
   *       false otherwise.
   */
  public boolean equals(Object other)
  {
    if(other instanceof FilmShow) {
      FilmShow otherFilmShow = (FilmShow) other;
      return filmName.equals(otherFilmShow.getFilmName()) &&
          cinemaName.equals(otherFilmShow.getCinemaName()) &&
          showDay.toString().equals(
          otherFilmShow.getShowDay().toString());
    }
    else {
      return false;
    }
  }
}
```

BookingRepository.java

```
import java.io.Serializable;
import java.util.HashMap;
import java.util.Iterator;
/**
 * A class to maintain an arbitrary number of booking informations.
 * Each booking is indexed by the booking number.
 *
 * @author Fariha Nazmul
 * @version 01.11.10
 */
public class BookingRepository implements Serializable
{
   // Storage for an arbitrary number of booking.
   private HashMap<Integer, BookingInformation> allBooking;
   // A unique booking number for each booking
   private int bookingNumber;

   /**
    * Performs initialization for the booking collection.
    */
   public BookingRepository()
   {
      allBooking = new HashMap<Integer, BookingInformation>();
      bookingNumber = 0;
   }

   /**
    * Returns whether or not the current booking number is in use.
    * @param key The number to be looked up.
    * @return true if the key is in use, false otherwise.
    */
   public boolean keyInUse(int key)
   {
      return allBooking.containsKey(new Integer (key));
   }

   /**
    * Returns the last booking number.
    * @return The last booking number
    */
   public int getBookingNumber()
   {
      return bookingNumber;
   }

   /**
    * Looks up a booking number and returns the
    * corresponding booking information.
    * @param bookingNum The booking number to be looked up.
    * @return The booking information corresponding to the number.
    */
   public BookingInformation getBooking(int bookingNum)
   {
      if(keyInUse(bookingNum))
         return allBooking.get(new Integer(bookingNum));
      else
         return null;
   }

   /**
    * Adds a new booking information to the collection.
    * @param booking The booking information associated
    *                with next booking number
    */
   public void addBooking(BookingInformation booking)
   {
      bookingNumber++;
      if(booking != null)
         allBooking.put(new Integer(bookingNumber), booking);
   }


   /**
    * Searches for all booking informations stored under a person
    * with the given name.
    * @param personName The name of the person
    * @return A String representing all the bookings found.
    */
   public String showAllBooking(String personName)
   {
      boolean foundPerson = false;
      StringBuffer matches = new StringBuffer();
      if(personName != null) {
         // Find the bookings that are made by the person.
         Iterator<Integer> it = allBooking.keySet().iterator();
         while(it.hasNext()) {
            Integer key = it.next();
            BookingInformation booking = allBooking.get(key);
            if(booking.getPersonName().equals(personName)) {
               //at least one booking is found
               foundPerson = true;
               matches.append(this.showBooking(key));
            }
         }
      }
      if(!foundPerson)    //if there is no match
```

```
            matches.append("Person Not Found: " + personName.toUpperCase());

        return matches.toString();
    }

    /**
     * The booking information associated with the booking number.
     * @param bookingNum The booking number
     * @return The booking information
     */
     public String showBooking(int bookingNum)
     {
        StringBuffer booking = new StringBuffer();
        if(keyInUse(bookingNum)){
            booking.append("\nBooking Number: " + bookingNum);
            booking.append(((BookingInformation)(allBooking.get(
                    new Integer(bookingNum)))).toString());
        }
        else{
            booking.append("Booking Not Found: " +  bookingNum);
        }
        return booking.toString();
    }

}
```

BookingInformation.java

```
import java.io.Serializable;
import java.util.ArrayList;
/**
 * A class to maintain the information of each booking.
 * Each booking holds the following information:
 *     Name of the person,
 *     Name of the film,
 *     Name of the cinema,
 *     Day of the show,
 *     Number of seats reserved
 *
 * @author Fariha Nazmul
 * @version 01.11.10
 */
public class BookingInformation implements Serializable
{
    // The name of the person booking the seats
    private String personName;
    // The filmShow consisting the information of the show
    //i.e. film name, cinema name and day
    private FilmShow filmShow;
    // The reserved seats
    private ArrayList<Seat> reservedSeats;

    /**
     * Constructor for objects of class BookingInformation
     * @param personName Name of the person
     * @param filmShow Information of the filmShow
     * @param seats The list of reserved seats
     */

    public BookingInformation(String personName, FilmShow filmShow,
            ArrayList<Seat> seats)
    {
        this.personName = personName;
        this.filmShow = filmShow;
        this.reservedSeats = (ArrayList<Seat>) seats;
    }

    /**
     * Returns the name of the person making the booking
     * @return     the name of the person
     */
    public String getPersonName()
    {
        return this.personName;
    }

    /**
     * Returns the information of the film show
     * @return     the film show
     */
    public FilmShow getFilmShow()
    {
        return this.filmShow;
    }

    /**
     * Returns the list of reserved seats
     * @return     the list of seats
     */
    public ArrayList<Seat> getReservedSeats()
    {
        return this.reservedSeats;
    }

    /**
     * A String representation of the booking information
     * @return     the string representation of booking information
     */
    public String toString()
```

```
   {
      StringBuffer strBuffer = new StringBuffer();
      strBuffer.append("\n"+ "Person Name: " + personName.toUpperCase() +
         "\n"+ filmShow.toString()+ "\n" + "Reserved Seats: ");
      for(Seat s: reservedSeats)
      {
         strBuffer.append(s.toString() + " ");

      }
      return strBuffer.toString();
   }
}
```

CinemaRepository.java

```
import java.io.Serializable;
import java.util.ArrayList;
import java.util.Iterator;

/**
 * A class to maintain an arbitrary number of cinemas.
 *
 * @author Fariha Nazmul
 * @version 01.11.10
 */
public class CinemaRepository implements Serializable {
   // Storage for an arbitrary number of cinemas.

   private ArrayList<Cinema> allCinema;

   /**
    * Perform any initialization for the cinema collection.
    */
   public CinemaRepository() {
      allCinema = new ArrayList<Cinema>();
   }

   /**
    * Checks if a Ciname is already created.
    * @param cinemaName The name of the cinema
    * @return true if the Cinema exists, else false
    */
   public boolean isCinema(String cinemaName) {
      for (Iterator it = allCinema.iterator(); it.hasNext();) {
         if (((Cinema) (it.next())).getName().equals(cinemaName)) {
            return true;
         }
      }
      return false;
   }
```

```
/**
 * Adds a new cinema to the collection.
 * @param newCinema The cinema to be added
 * @return Status of the add operation
 */
public String addCinema(Cinema newCinema) throws NullPointerException {
   StringBuffer result = new StringBuffer();
   if (newCinema != null) {
      //checks if the cinema already exists
      if (this.isCinema(newCinema.getName())) {
         result.append("Cinema Hall Exists: ");
         result.append("\n" + newCinema.getName().toUpperCase()
               + " already exists in the system.");
         return result.toString();
      } else {
         allCinema.add(newCinema);
         result.append(newCinema.getName().toUpperCase()
               + " added to the system.");
      }
   } else {
      throw new NullPointerException("No cinema provided with " +
            "addCinema command.");
   }

   return result.toString();
}

/**
 * Books number of seats for a certain film show.
 * @param filmShow The information of the filmshow
 * @param numOfSeats Number of seats to be booked
 * @return The list of reserved books
 */
public ArrayList<Seat> bookSeat(FilmShow filmShow, int numOfSeats) {
   int availableSeats = 0, bookedSeats = 0, theatreIndex = 0;
   Theatre theatre = null;
   ArrayList<Seat> seats = new ArrayList<Seat>();
   for (int i = 0; i < allCinema.size(); i++) {
      if (allCinema.get(i).getName().equals(filmShow.getCinemaName())) {
         theatre = allCinema.get(i).getTheatre();
         theatreIndex = i;
      }
   }
   //check first if all the seats are available or not
   for (int i = 1; i <= theatre.getNumberOfRows(); i++) {
      bookedSeats=theatre.getRow(i).getBookedSeats(filmShow.getShowDay());
      bookedSeats++;
      for (int j = availableSeats; j < numOfSeats
            && bookedSeats <= theatre.getRowLength(); j++) {
                seats.add(theatre.getRow(i).getOneSeat(bookedSeats));
                availableSeats++;
                bookedSeats++;
            }
            if (availableSeats >= numOfSeats) {
                break;
            }
        }
        //book the seats if they are availabe
        if (availableSeats >= numOfSeats) {
            for (int i = 0; i < seats.size(); i++) {
                allCinema.get(theatreIndex).getTheatre().getRow(
                        seats.get(i).getRowNumber()).bookOneSeat(
                          filmShow.getShowDay(), seats.get(i).getSeatNumber());
            }
        } //seats are not available
        else {
            seats = null;
        }
        return seats;
    }

    /**
     * Returns the list of all cinema names in the system.
     * @return the list of cinemas
     */
    public ArrayList<String> getAllCinemas() {

        ArrayList<String> cinemaList = new ArrayList<String>();
        for (Cinema cinema : allCinema) {
            if (!cinemaList.contains(cinema.getName())) {
                cinemaList.add(cinema.getName());
            }
        }
        return cinemaList;
    }
}
```

Cinema.java

```
import java.io.Serializable;
/**
 * A class to hold the information of the Cinema.
 * A cinema can have many theatres.
 * For simplicity, currently it holds only one thetre.
 * But with some modification, it can add more theatres
 *
 * @author Fariha Nazmul
 * @version 01.11.10
 */
```

```
public class Cinema implements Serializable
{
   // The name of the Cinema
   private String cinemaName;
   // The theatre inside the cinema
   private Theatre theatre;
   //private ArrayList<Theatre> theatres;

   /**
    * Constructor for objects of class Cinema.
    * @param cinemaName    The name of the cinema
    * @param numberOfRows  The number of rows in cinema
    * @param rowLength     The number seats in each row
    */
   public Cinema(String cinemaName, int numberOfRows, int rowLength)
   {
      this.cinemaName = cinemaName;
      this.theatre = new Theatre(cinemaName, numberOfRows, rowLength);
      //addTheatre(theatreName, numberOfRows, rowLength)
   }

   /**
    * Returns the name of the cinema.
    * @return     the name
    */
   public String getName()
   {
      return  this.cinemaName;
   }

   /**
    * Returns the theatre inside the cinema.
    * @return     the theatre
    */
   public Theatre getTheatre()
   {
      return this.theatre;
   }

   /* public void addTheatre(String theatreName, int numberOfRows, int rowLength)
   {
      Theatre tempTheatre = new Theatre(theatreName, numberOfRows, rowLength);
      this.theatres.add(tempTheatre);
   }*/

}
```

Theatre.java

```
import java.io.Serializable;
import java.util.ArrayList;

/**
 * A class to maintain the information of a theatre.
 *
 * @author Fariha Nazmul
 * @version 01.11.10
 */
public class Theatre implements Serializable
{
    private String name;    //the name of the theatre
    private int numberOfRows;       //number of rows
    private int rowLength;          //number of seats in a row
    private ArrayList<Row> rows;    //list of rows

    /**
     * Constructor for objects of class Theatre.
     */
    public Theatre(String theatreName, int numberOfRows, int rowLength)
    {
        this.name = theatreName;
        this.numberOfRows = numberOfRows;
        this.rowLength = rowLength;
        this.rows = new ArrayList<Row>();

        for(int i = 1; i <= numberOfRows; i++)
        {
            Row tempRow = new Row(i, rowLength);
            rows.add(tempRow);
        }
    }

    /**
     * Returns the name of the theatre.
     * @return     the name of the theatre
     */
    public String getName()
    {
        return  this.name;
    }

    /**
     * Returns the number of rows.
     * @return     the number of rows of the theatre
     */
    public int getNumberOfRows()
    {
        return  this.numberOfRows;
    }

    /**
     * Returns the length of the row.
     * @return     the length of the row of the theatre
     */
    public int getRowLength()
    {
        return  this.rowLength;
    }

    /**
     * Returns a row object.
     * @param rowNum The number of the row
     * @return       The row object
     */
    public Row getRow(int rowNum)
    {
        return rows.get(rowNum-1);
    }

}
```

Row.java

```
import java.io.Serializable;
import java.util.ArrayList;

/**
 * A class to maintain a row of seats in a cinema hall.
 * Stores collection of seats.
 * Can find seats by number.
 * Requests for reservations.
 *
 * @author Fariha Nazmul
 * @version 01.11.10
 */
public class Row implements Serializable
{
    // the number corresponding to this row
    private int rowNumber;
    // the number of seats in this row
    private int rowLength;
    // the list of seats in the row
    private ArrayList<Seat> seats;
    /**
     * Constructor for objects of class Row
     */
    public Row(int rowNumber, int rowLength)
    {
        this.rowNumber = rowNumber;
        this.rowLength = rowLength;
        this.seats = new ArrayList<Seat>();

        for(int i = 1; i <= rowLength; i++){
            Seat tempSeat = new Seat(rowNumber, i);
            seats.add(tempSeat);
        }
    }

    /**
     * Returns the row number.
     * @return     the row number
     */
    public int getRowNumber()
    {
        return rowNumber;
    }

    /**
     * Returns the row length.
     * @return     the number of seats in the row
     */
    public int getRowLength()
    {
        return rowLength;
    }

    /**
     * Looks up for the number of seats booked on a particular day.
     * @param day the day to look up
     * @return     the number of booked seats
     */
    public int getBookedSeats(Day day)
    {
        int bookedSeats = 0;
        for(int i = 0; i < rowLength; i++){
            if(seats.get(i).getBookingStatus(day))
                bookedSeats++;
        }
        return bookedSeats;
    }

    /**
     * Returns a single seat.
     *
     * @param  seatNum    The number of the seat
     * @return     the indexed seat
     */
    public Seat getOneSeat(int seatNum)
    {
        return seats.get(seatNum-1);
    }

    /**
     * Returns a list of seats.
     *
     * @param  startSeatNum    The first seat number of the list
     * @param  endSeatNum      The last seat number of the list
     * @return     the list of seats
     */
    public ArrayList<Seat> getSomeSeats(int startSeatNum, int endSeatNum)
    {
        ArrayList<Seat> someSeats = new ArrayList<Seat>();
        for(int i = startSeatNum-1; i < endSeatNum; i++ ){
            someSeats.add(seats.get(i));
        }
        return someSeats;
    }

    /**
     * Books a single seat on a day.
     * @param day       the day to book
     * @param seatNum   the seat number
     */
    public void bookOneSeat(Day day, int seatNum)
    {
        seats.get(seatNum-1).setBookingStatus(day, true);
    }

}
```

Seat.java

```
import java.io.Serializable;


/**
 * A class to maintain the information of each seat in a Cinema Hall.
 * A Seat can accept booking, stores the booked status
 * and also stores booking information
 *
 * @author Fariha Nazmul
 * @version 01.11.10
 */
public class Seat implements Serializable
{
    private int rowNumber;      //the row number
    private int seatNumber;     //the seat number
    private char seatPosition;  //seat representation
    private BookingStatus bookingStatus;    //status of the seats of each day


    /**
     * Constructor for objects of class Seat.
     */
    public Seat(int row, int seat)
    {
       rowNumber = row;
       seatNumber = seat;
       seatPosition = (char) ('A' + seat - 1);
       bookingStatus = new BookingStatus(Day.values().length);
    }

    /**
     * Checks the booking status of the seat.
     * @param day  the day to check the status
     * @return     true if the seat is booked, else false
     */
    public boolean getBookingStatus(Day day)
    {
       return bookingStatus.getStatus(day.toInteger());
    }

    /**
     * Set the booking status of the seat to given input.
     * @param  day      The day to book
     * @param  status   booking status of the seat
     */
    public void setBookingStatus(Day day, boolean status)
    {
       bookingStatus.setStatus(day.toInteger(), status);
    }

    /**
     * Returns the row number of the seat.
     * @return     the row number
     */
    public int getRowNumber()
    {
       return rowNumber;
    }

    /**
     * Returns the seat number.
     * @return     the seat number
     */
    public int getSeatNumber()
    {
```

```
    return seatNumber;
  }

  /**
   * A string representation of the seat.
   * @return      the string representation of the seat
   */

  public String toString()
  {
    Integer row = new Integer(rowNumber);
    return row.toString()+seatPosition;
  }

  /**
   * Implements content equality for seats.
   * @return true if this Seat matches the other,
   *       false otherwise.
   */

  public boolean equals(Object other)
  {
    if(other instanceof Seat) {
      Seat otherSeat = (Seat) other;
      return rowNumber == otherSeat.getRowNumber() &&
           seatNumber == otherSeat.getSeatNumber();
    }
    else {
      return false;
    }
  }
}
```

BookingStatus.java

```
import java.io.Serializable;

/**
 * A class to maintain the booking status for day.
 * The list is indexed by the day
 *
 * @author Fariha Nazmul
 * @version 01.11.10
 */
public class BookingStatus implements Serializable
{
  // A list of booleans to hold the booking status
  private boolean[] slots;
```

```
    /**
     * Constructor for objects of class BookingStatus
     * @param num The number of elements in the list
     */
    public BookingStatus(int num)
    {
        // initialise the list with false i.e. available
        slots = new boolean[num];
        for(int i = 0; i < num; i++)
            slots[i] = false;
    }

    /**
     * Return the status of the slot i
     *
     * @param  i   Index number of the slot to be looked up
     * @return     the status of the slot
     */
    public boolean getStatus(int i)
    {
        return slots[i];
    }

    /**
     * Set the status of the slot i
     *
     * @param  i       Index number of the slot to set
     * @param  status   The status to set
     */
    public void setStatus(int i, boolean status)
    {
        this.slots[i] = status;
    }
}
```

Day.java

```
import java.io.Serializable;


/**
 * Enumeration class Day
 * Representations for all the days of a week
 *
 * @author Fariha Nazmul
 * @version 01.11.10
 */
public enum Day implements Serializable
{
   // A value for each day along with its
   // corresponding user interface string
   // and a integer that helps to index day arrays.

    MON("mon", 0),
    TUE("tue", 1),
    WED("wed", 2),
    THU("thu", 3),
    FRI("fri", 4),
    SAT("sat", 5),
    SUN("sun", 6);

   // The day string.
   private String dayString;
   // The corresponding integer
   private int dayInteger;

   /**
    * Initialise with the corresponding day string.
    * @param dayString  The first three letters of the day string.
    * @param dayInteger The integer value of the day.
    */
   Day(String dayString, int dayInteger)
   {
      this.dayString = dayString;
      this.dayInteger = dayInteger;
   }

   /**
    * Returns string representation of the Day.
    * @return day as a string.
    */
   public String toString()
   {
      return dayString;
   }

   /**
    * Returns corresponding integer value of the Day.
    * @return day as an Integer.
    */
   public int toInteger()
   {
      return dayInteger;
   }
}
```

DayParameters.java

```
import java.io.Serializable;
import java.util.HashMap;

/**
 * A class to recognise Days as they are typed in.
 *
 * @author Fariha Nazmul
 * @version 01.11.10
*/
public class DayParameters implements Serializable
{

   // A mapping between a day parameter string
   //and the Day associated with it.
   private HashMap<String, Day> dayParameters;

   /**
    * Constructor - initialises the day strings.
    */
   public DayParameters()
   {
      this.dayParameters = new HashMap<String, Day>();
      for(Day day : Day.values()) {
         this.dayParameters.put(day.toString(), day);
      }
   }

   /**
    * Finds the Day associated with a day parameter.
    * @param dayParameter The word to look up.
    * @return The Day correspondng to dayParameter
    */
   public Day getDay(String dayParameter)
   {
      Day day = dayParameters.get(dayParameter);
      return day;

   }

   /**
    * Checks whether a given String is a valid day parameter.
    * @param dayString The day string
    * @return true if it is, false otherwise.
    */
   public boolean isDay(String dayString)
   {
      return dayParameters.containsKey(dayString);
    }

    /**
     * Prints all days to System.out.
     */
    public void showAll()
    {
        for(String day : dayParameters.keySet()) {
            System.out.print(day + "  ");
        }
        System.out.println();
    }

}
```

User.java

```
import java.io.Serializable;

/**
 * Enumeration class User
 * Representations for all users.
 *
 * @author Fariha Nazmul
 * @version 01.11.10
 */
public enum User implements Serializable
{
    // A value for each user in the system along with its
    // corresponding user interface string


    USER("user"), ADMIN("admin"), CUSTOM("custom");


    // The user string.
    private String userString;

    /**
     * Initialzse with the corresponding user string.
     * @param userString  The string representation of an user.
     */
    User(String userString)
    {
        this.userString = userString;
    }



    /**
```

```
     * @return user as a string.
     */
    public String toString()
    {
        return userString;
    }
}
```

UserParameters.java

```
import java.io.Serializable;
import java.util.HashMap;

/**
 * A class to recognise Users as they are typed in.
 *
 * @author Fariha Nazmul
 * @version 01.11.10
 */
public class UserParameters implements Serializable
{

    // A mapping between a user parameter string
    //and the User associated with it.
    private HashMap<String, User> userParameters;


    /**
     * Constructor - initialises the user strings.
     */
    public UserParameters()
    {
        this.userParameters = new HashMap<String, User>();
        for(User user : User.values()) {
            this.userParameters.put(user.toString(), user);
        }
    }

    /**
     * Finds the User associated with a user parameter.
     * @param userParameter The word to look up.
     * @return The User correspondng to userParameter
     */
    public User getUser(String userParameter)
    {
        User user = userParameters.get(userParameter);
        return user;
    }

    /**
```

```
     * Checks whether a given String is a valid user parameter.
     * @param userString The user string
     * @return true if it is, false otherwise.
     */
    public boolean isUser(String userString)
    {
        if(userParameters.get(userString) != User.USER)
            return userParameters.containsKey(userString);
        else
            return false;
    }
}
```

GraphicalInterface.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;

import java.io.File;

import java.util.List;
import java.util.ArrayList;

/**
 * Provides the GUI of the Cinema Booking System.
 * Different buttons provide access to the data in the system.
 * It builds and displays the application GUI and
 * initialises all other components.
 *
 *
 * @author Fariha Nazmul
 * @version 01.11.10
 */
public class GraphicalInterface implements Interface {
    // static fields:
    private final static String newline = "\n";
    private static final String VERSION = "Version 2.1";
    private static JFileChooser fileChooser =
            new JFileChooser(System.getProperty("user.dir"));

    // The cinema booking system to be viewed and manipulated.
    private CinemaSystemEngine cinemaSystem;
    // A file handler for the IO operations
    private FileHandler fileHandler;

    // fields for swing components:
    private JFrame frame;
```

```
  private JLabel modeLabel;
  private JLabel versionLabel;
  private JRadioButton modeAdmin;
  private JRadioButton modeCustom;
  private ButtonGroup modeButtons;
  private JTextArea outputTextArea;
  private JButton addcinema;
  private JButton addfilm;
  private JButton addFrom;
  private JButton load;
  private JButton save;
  private JButton when;
  private JButton where;
  private JButton book;
  private JButton show;
  private JButton showAll;
  private JButton exit;
  private JMenuItem loadItem;
  private JMenuItem saveItem;
  private JMenuItem addFromItem;
  private JMenuItem exitItem;
  private DefaultListModel filmListModel;
  private DefaultListModel cinemaListModel;

  /**
   * Creates an GraphicalInterface.
   */
  public GraphicalInterface(CinemaSystemEngine cinemaSystem) {
    this.cinemaSystem = cinemaSystem;
    this.fileHandler = new FileHandler(this.cinemaSystem);

    filmListModel = new DefaultListModel();
    cinemaListModel = new DefaultListModel();
    createFilmListModel();
    createCinemaListModel();

    makeFrame();
  }

  /**
   * Displays the GUI on screen.
   */
  public void run(){
    frame.setVisible(true);
  }

  // menu and button functions

 /**
 * AddCinema function: adds cinema in the application.
 */
  public void addCinema(){
     String[] dialogLabels = {"Cinema Hall Name:\n", "Number of Rows:\n",
                      "Number of Columns:\n"};
     runDialogBox(CommandWord.ADD_CINEMA, dialogLabels);
     createCinemaListModel();
     frame.pack();
  }

 /**
 * AddFilm function: adds films in the application.
 */
  public void addFilm(){
     String[] dialogLabels = {"Film Name:\n", "Cinema Hall name:\n",
                      "Days (Separated by Comma):\n"};
     runDialogBox(CommandWord.ADD_FILM, dialogLabels);
     createFilmListModel();
     frame.pack();
  }

 /**
 * Book function: books seats for a film show.
 */
  public void bookSeat(){
     String[] dialogLabels = {"Person Name:\n", "Film Name:\n",
                      "Cinema Name:\n", "Show Day:\n",
                      "Number of Seats:\n",};
     runDialogBox(CommandWord.BOOK, dialogLabels);
     frame.pack();
  }

 /**
  * Where function: finds the cinema hall where a film is showing.
  */
  public void whereFilm(){
     String[] dialogLabels = {"Film name:\n","Day:\n"};
     runDialogBox(CommandWord.WHERE, dialogLabels);
     frame.pack();
  }

 /**
 * When function: finds when a film is showing.
 */
  public void whenFilm(){
     String field = "Film name";
     runOptionPane(field, CommandWord.WHEN);
  }

  /**
  * Show function: shows the booking for the booking number.
```

```
*/
public void showBooking(){
   String field = "Booking Number";
   runOptionPane(field, CommandWord.SHOW);
}

 /**
 * ShowAll function: shows all the bookings made by a person.
 */
public void showAllBooking(){
   String field = "Person's name";
   runOptionPane(field, CommandWord.SHOW_ALL);
}

/**
 * AddFrom function: opens a file chooser to select a file,
 * and then reads and executes commands from the chosen file.
 */
public void addFromFile() {
   runFileChooser(newline, CommandWord.ADD_FROM);
   createCinemaListModel();
   createFilmListModel();
   frame.pack();
}

/**
 * Load function: opens a file chooser to select a file,
 * and then loads the system from the chosen file.
 */
public void loadFile() {
   runFileChooser(newline, CommandWord.LOAD);
   createCinemaListModel();
   createFilmListModel();
   frame.pack();
}

/**
 * Save function: saves the current system to a file.
 */
public void saveFile() {
   runFileChooser(newline, CommandWord.SAVE);
   frame.pack();
}

/**
 * Mode function: changes the user mode of the application.
 */
public void switchMode(User user){
   String status = "";
   ArrayList<String> commandParameter = new ArrayList<String>();
```

```
        commandParameter.add(user.toString());
        Command command = new Command(CommandWord.MODE, commandParameter);
        try{
            status = cinemaSystem.setMode(command);
        } catch(Exception ex){
            status = ex.getMessage();
        }
        clearStatus();
        showStatus(status + newline);
        frame.pack();
    }

    /**
     * Exit function: quits the application.
     */
    public void exit() {
        System.exit(0);
    }

    /**
     * 'About' function: shows the 'about' box.
     */
    public void showAbout() {
        JOptionPane.showMessageDialog(frame,
                "Cinema Booking System\n" + VERSION,
                "About Cinema Booking System",
                JOptionPane.INFORMATION_MESSAGE);
    }

    // support methods

    /**
     * Given a command and dialogLabels, creates a dialog box
     * to get user input and then executes the command with input.
     * Used for commands that need multiple user inputs.
     */
    public void runDialogBox(CommandWord cmd, String[] dialogLabels){
        String status = null;
        ArrayList<String> commandParameter =
                new DialogBox(frame, cmd, dialogLabels).getInput();
        if(commandParameter.contains("") || commandParameter.isEmpty()) return;
        Command command = new Command(cmd, commandParameter);
        try{
            switch(cmd){
                case ADD_CINEMA: status = cinemaSystem.addCinema(command);break;
                case ADD_FILM: status = cinemaSystem.addFilm(command);break;
                case BOOK: status = cinemaSystem.book(command);break;
                case WHERE: status = cinemaSystem.where(command);break;
            }
    }catch(Exception ex){
       status = ex.getMessage();
    }
    showStatus(status+newline);
 }

 /**
  * Given a field name and command, creates a JOptionPane
  * to get user input and then executes the command with input.
  * Used for commands that need single user input.
  */
 public void runOptionPane(String field, CommandWord cmd){
    String input = (String)JOptionPane.showInputDialog(
                        frame,
                        "Please Enter the " + field + ":\n",
                        cmd.toString().toUpperCase(),
                        JOptionPane.PLAIN_MESSAGE,
                        null, null, "");
    if ((input == null) || (input.length() == 0)) {
                return;}
    ArrayList<String> commandParameter = new ArrayList<String>();
    commandParameter.add(input.trim());
    Command command = new Command(cmd, commandParameter);
    switch(cmd){
       case WHEN: showStatus(cinemaSystem.when(command)+newline);break;
       case SHOW: showStatus(cinemaSystem.show(command)+newline);break;
       case SHOW_ALL: showStatus(cinemaSystem.showAll(command)+newline);
                 break;
    }
 }

  /**
  * Given a msg and a command, creates a FileChooser
  * to get user specified file name and then executes the command
  * with input.
  * Used for commands that need file operation.
  */
 public void runFileChooser(String msg, CommandWord cmd){
    int returnVal = 1;
    String status = null;
    switch(cmd){
       case ADD_FROM: returnVal = fileChooser.showOpenDialog(frame);break;
       case LOAD: returnVal = fileChooser.showOpenDialog(frame);break;
       case SAVE: returnVal = fileChooser.showSaveDialog(frame);break;
    }
    if (returnVal != JFileChooser.APPROVE_OPTION) {
       return;  // cancelled
    }
    File selectedFile = fileChooser.getSelectedFile();
    if(selectedFile.getName()!= null){
       ArrayList<String> commandParameter = new ArrayList<String>();
       commandParameter.add(selectedFile.getName());
       Command command = new Command(cmd, commandParameter);
       try {
         switch(cmd){
           case ADD_FROM: status = fileHandler.addFrom(command);;break;
           case LOAD: this.cinemaSystem = fileHandler.load(command);
                 status = "File Load Successful";break;
           case SAVE: status = fileHandler.save(command);;break;
           }
       } catch (Exception ex) {
         JOptionPane.showMessageDialog(frame,
                ex.getMessage() +newline + cmd.toString().toUpperCase()
                + " File Operation Not Successful.",
                cmd.toString().toUpperCase()+"File Error",
                JOptionPane.ERROR_MESSAGE);
         return;
       }
    }
    showStatus(status+ newline);
 }

 /**
  * Shows a message in the middle pane.
  * @param text The status message.
  */
 public void showStatus(String text) {
    outputTextArea.append(text + newline);
    outputTextArea.setCaretPosition(
         outputTextArea.getDocument().getLength());
 }

 /**
  * Clears the text area.
  */
 public void clearStatus() {
    outputTextArea.setText("");
    outputTextArea.append("Welcome to Cinema Booking System."+ newline);
    outputTextArea.append("Click on any button on the left."
         + newline+ newline);
    outputTextArea.setCaretPosition(
         outputTextArea.getDocument().getLength());
 }

 /**
  * Enables or disables toolbar buttons and menu items for ADMIN mode.
  * @param status  'true' to enable, 'false' to disable.
  */
 public void setAdminButtonsVisible(boolean status) {
    addcinema.setVisible(status);
    addfilm.setVisible(status);
    addFrom.setVisible(status);
    load.setVisible(status);
    save.setVisible(status);
    exit.setEnabled(status);

    loadItem.setEnabled(status);
    saveItem.setEnabled(status);
    addFromItem.setEnabled(status);
    exitItem.setEnabled(status);
    //commandLineItem.setEnabled(status);
 }

 /**
  * Enables or disables toolbar buttons and menu items for CUSTOM mode.
  * @param status  'true' to enable, 'false' to disable.
  */
 public void setCustomButtonsVisible(boolean status) {
    book.setVisible(status);
 }

 /**
  * Creates a list model with all the existing film names.
  */
 public void createFilmListModel() {
    List<String> filmList = cinemaSystem.getAllFilms();
    filmListModel.clear();
    for (String filmName : filmList) {
       filmListModel.addElement(filmName);
    }
 }

 /**
  * Creates a list model with all the existing cinema names.
  */
 public void createCinemaListModel() {
    List<String> cinemaList = cinemaSystem.getAllCinemas();
    cinemaListModel.clear();
    for (String cinemaName : cinemaList) {
       cinemaListModel.addElement(cinemaName);
    }
 }

 // Swing stuff to build the frame and all its components and menus

 /**
  * Creates the Swing frame and its content.
  */
 public void makeFrame() {
    // Create the fram
    frame = new JFrame("Cinema Booking System");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JPanel contentPane = (JPanel) frame.getContentPane();
    contentPane.setBorder(new EmptyBorder(6, 6, 6, 6));
    // Specify the layout manager with spacing
    contentPane.setLayout(new BorderLayout(6, 6));

    makeMenuBar();

    makeWestPane();
    makeEastPane();
    makeNorthPane();
    makeSouthPane();
    makeCenterPane();

    // building is done - arrange the components
    if (cinemaSystem.getMode() == User.USER) {
       setAdminButtonsVisible(false);
       setCustomButtonsVisible(false);
    } else if (cinemaSystem.getMode() == User.ADMIN) {
       setAdminButtonsVisible(true);
       setCustomButtonsVisible(false);
    } else if (cinemaSystem.getMode() == User.CUSTOM) {
       setAdminButtonsVisible(false);
       setCustomButtonsVisible(true);
    }

    frame.pack();
    // place the frame at the center of the screen and show
    Dimension d = Toolkit.getDefaultToolkit().getScreenSize();
    frame.setLocation(d.width / 2 - frame.getWidth() / 2,
            d.height / 2 - frame.getHeight() / 2);
    //frame.setVisible(true);
 }

 /**
  * Creates the main frame's menu bar.
  */
 public void makeMenuBar() {
    final int SHORTCUT_MASK =
            Toolkit.getDefaultToolkit().getMenuShortcutKeyMask();

    JMenuBar menubar = new JMenuBar();
    frame.setJMenuBar(menubar);
    JMenu menu;

    // create the File menu
    menu = new JMenu("File");
    menubar.add(menu);
```

```
addFromItem = new JMenuItem("AddFrom");
addFromItem.setAccelerator(KeyStroke.getKeyStroke(
      KeyEvent.VK_A, SHORTCUT_MASK));
addFromItem.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) { addFromFile(); }
   });
menu.add(addFromItem);
menu.addSeparator();

loadItem = new JMenuItem("Load");
loadItem.setAccelerator(KeyStroke.getKeyStroke(
      KeyEvent.VK_L, SHORTCUT_MASK));
loadItem.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
      loadFile();
   }
});
menu.add(loadItem);

saveItem = new JMenuItem("Save");
saveItem.setAccelerator(KeyStroke.getKeyStroke(
      KeyEvent.VK_S, SHORTCUT_MASK));
saveItem.addActionListener(new ActionListener() {

   public void actionPerformed(ActionEvent e) {
      saveFile();
   }
});
menu.add(saveItem);
menu.addSeparator();

exitItem = new JMenuItem("Exit");
exitItem.setAccelerator(KeyStroke.getKeyStroke(
      KeyEvent.VK_Q, SHORTCUT_MASK));
exitItem.addActionListener(new ActionListener() {

   public void actionPerformed(ActionEvent e) {
      exit();
   }
});
menu.add(exitItem);

// create the Help menu
menu = new JMenu("Help");
menubar.add(menu);
JMenuItem item = new JMenuItem("About CinemaBookingSystem...");
item.addActionListener(new ActionListener() {

   public void actionPerformed(ActionEvent e) {
```

```
            showAbout();
        }
    });
    menu.add(item);

}

/**
 * Creates the main frame's North Panel.
 */
public void makeNorthPane() {

    // Create the mode selection buttons on the NORTH
    modeLabel = new JLabel("Mode ");
    modeAdmin = new JRadioButton("ADMIN");
    modeAdmin.setToolTipText("Change to ADMIN mode");
    modeCustom = new JRadioButton("CUSTOM");
    modeCustom.setToolTipText("Change to CUSTOM mode");

    // add the radio buttons to a button group
    modeButtons = new ButtonGroup();
    modeButtons.add(modeAdmin);
    modeButtons.add(modeCustom);

    // add action listener to the radio buttons
    modeAdmin.addActionListener(new ActionListener() {

        public void actionPerformed(ActionEvent e) {
            setAdminButtonsVisible(true);
            setCustomButtonsVisible(false);
            switchMode(User.ADMIN);
        }
    });
    modeCustom.addActionListener(new ActionListener() {

        public void actionPerformed(ActionEvent e) {
            setAdminButtonsVisible(false);
            setCustomButtonsVisible(true);
            switchMode(User.CUSTOM);
        }
    });

    // Add mode buttons into panel with flow layout for spacing
    JPanel flowMode = new JPanel();
    flowMode.setLayout(new FlowLayout(FlowLayout.CENTER, 35, 0));
    flowMode.setBorder(
            BorderFactory.createEtchedBorder(EtchedBorder.RAISED));
    flowMode.add(modeLabel);
    flowMode.add(modeAdmin);
    flowMode.add(modeCustom);
    JPanel contentPane = (JPanel) frame.getContentPane();
    contentPane.add(flowMode, BorderLayout.NORTH);
  }

  /**
   * Creates the main frame's South Panel.
   */
  public void makeSouthPane() {
    // Create the version label at the bottom
    versionLabel = new JLabel(VERSION);
    JPanel contentPane = (JPanel) frame.getContentPane();
    contentPane.add(versionLabel, BorderLayout.SOUTH);
  }

  /**
   * Creates the main frame's Center Panel.
   */
  public void makeCenterPane() {

    // Create the output panel in the CENTER
    JPanel middlePane = new JPanel();
    middlePane.setLayout(new BoxLayout(middlePane, BoxLayout.Y_AXIS));
    middlePane.setBorder(
        BorderFactory.createEtchedBorder(EtchedBorder.RAISED));

    outputTextArea = new JTextArea(12, 30);
    outputTextArea.setToolTipText("Output Terminal");
    outputTextArea.setBorder(BorderFactory.createLineBorder(Color.black));
    outputTextArea.setEditable(false);
    outputTextArea.setFont(new Font("Serif", Font.ITALIC, 14));
    outputTextArea.setLineWrap(true);
    outputTextArea.setWrapStyleWord(true);
    clearStatus();
    JScrollPane outputScrollPane = new JScrollPane(outputTextArea);
    outputScrollPane.setVerticalScrollBarPolicy(
        JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
    outputScrollPane.setBorder(
        BorderFactory.createEmptyBorder(10, 10, 10, 10));
    middlePane.add(outputScrollPane);

    JPanel contentPane = (JPanel) frame.getContentPane();
    contentPane.add(middlePane, BorderLayout.CENTER);
  }

  /**
   * Creates the main frame's West Panel.
   */
  public void makeWestPane() {

    // Create the toolbar with the buttons
```

```
JPanel toolbar = new JPanel();
toolbar.setLayout(new GridLayout(0, 1, 0, 10));

addcinema = new JButton("Add Cinema");
addcinema.setToolTipText("Click to add a cinema");
addcinema.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) { addCinema();}
   });
toolbar.add(addcinema);

addfilm = new JButton("Add Film");
addfilm.setToolTipText("Click to add a film");
addfilm.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) { addFilm(); }
   });
toolbar.add(addfilm);

addFrom = new JButton("Add From");
addFrom.setToolTipText("Click to add from a file");
addFrom.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) { addFromFile(); }
   });
toolbar.add(addFrom);

when = new JButton("When");
when.setToolTipText("Click to find a show day");
when.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) { whenFilm(); }
   });
toolbar.add(when);

where = new JButton("Where");
where.setToolTipText("Click to find a show hall");
where.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) { whereFilm(); }
   });
toolbar.add(where);

show = new JButton("Show");
show.setToolTipText("Click to find a booking");
show.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) { showBooking(); }
   });
toolbar.add(show);

showAll = new JButton("Show All");
showAll.setToolTipText("Click to find all booking");
showAll.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) { showAllBooking(); }
   });
```

```
        toolbar.add(showAll);


        book = new JButton("Book");
        book.setToolTipText("Click to book seats");
        book.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) { bookSeat(); }
            });
        toolbar.add(book);

        load = new JButton("Load");
        load.setToolTipText("Click to load a system");
        load.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) { loadFile(); }
            });
        toolbar.add(load);

        save = new JButton("Save");
        save.setToolTipText("Click to save the system");
        save.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) { saveFile(); }
            });
        toolbar.add(save);

        // Add toolbar into panel with flow layout for spacing
        JPanel flow = new JPanel();
        flow.setLayout(new FlowLayout(FlowLayout.CENTER));
        flow.setBorder(BorderFactory.createEtchedBorder(EtchedBorder.RAISED));
        flow.add(toolbar);

        JPanel contentPane = (JPanel) frame.getContentPane();
        contentPane.add(flow, BorderLayout.WEST);
    }

    /**
     * Creates the main frame's East Panel.
     */
    public void makeEastPane() {

        // Create the east pane with the lists and exit button on EAST
        JPanel eastPanel = new JPanel();
        eastPanel.setLayout(new BoxLayout(eastPanel, BoxLayout.Y_AXIS));

        // Create the list pane with the lists
        JPanel listPanel = new JPanel();
        listPanel.setLayout(new GridLayout(0, 1, 0, 10));
        listPanel.setBorder(BorderFactory.createEmptyBorder(10, 10, 20, 10));

        // Panel for the film list
        JPanel filmListPanel = new JPanel();
```

```
filmListPanel.setLayout(new BoxLayout(filmListPanel, BoxLayout.Y_AXIS));
filmListPanel.setBorder(
        BorderFactory.createEtchedBorder(EtchedBorder.LOWERED));
JLabel filmLabel = new JLabel("Films Showing");
filmListPanel.add(filmLabel);

JList allFilms = new JList(filmListModel);
allFilms.setBorder(BorderFactory.createLineBorder(Color.black));
allFilms.setLayoutOrientation(JList.VERTICAL);
allFilms.setVisibleRowCount(5);
JScrollPane filmListScroller = new JScrollPane(allFilms);
filmListScroller.setPreferredSize(new Dimension(180, 130));
filmListScroller.setBorder(
        BorderFactory.createEmptyBorder(10, 10, 20, 10));
filmListPanel.add(filmListScroller);
listPanel.add(filmListPanel);

// Panel for the cinema list
JPanel cinemaListPanel = new JPanel();
cinemaListPanel.setLayout(
        new BoxLayout(cinemaListPanel, BoxLayout.Y_AXIS));
cinemaListPanel.setBorder(
        BorderFactory.createEtchedBorder(EtchedBorder.LOWERED));
JLabel cinemaLabel = new JLabel("Cinema Halls");
cinemaListPanel.add(cinemaLabel);
JList allCinemas = new JList(cinemaListModel);
allCinemas.setBorder(BorderFactory.createLineBorder(Color.black));
allCinemas.setLayoutOrientation(JList.VERTICAL);
allCinemas.setVisibleRowCount(5);
JScrollPane cinemaListScroller = new JScrollPane(allCinemas);
cinemaListScroller.setPreferredSize(new Dimension(180, 130));
cinemaListScroller.setBorder(
        BorderFactory.createEmptyBorder(10, 10, 20, 10));
cinemaListPanel.add(cinemaListScroller);
listPanel.add(cinemaListPanel);

eastPanel.add(listPanel);

// exit button
exit = new JButton("Exit");
exit.setToolTipText("Click to EXIT");
exit.addActionListener(new ActionListener() {

    public void actionPerformed(ActionEvent e) {
        exit();
    }
});
JPanel flowExit = new JPanel();
flowExit.setLayout(new FlowLayout(FlowLayout.CENTER, 0, 0));
flowExit.add(exit);
```

```
        eastPanel.add(flowExit);

        // Add to contentpane
        JPanel contentPane = (JPanel) frame.getContentPane();
        contentPane.add(eastPanel, BorderLayout.EAST);
    }
}
```

DialogBox.java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.ArrayList;
import java.util.StringTokenizer;

/**
 * Provides a class used for GUI of the Cinema Booking System.
 * Creates the Dialog box with multiple input fields.
 * The function getInput returns all the user input as a list.
 *
 *
 * @author Fariha Nazmul
 * @version 01.11.10
 */
public class DialogBox {

    private JTextField text1;
    private JTextField text2;
    private JTextField text3;
    private JTextField text4;
    private JTextField text5;
    private JDialog dialogBox;
    private ArrayList<String> allInput;

    /**
     * Constructor - initialises the Dialog Box.
     */
    public DialogBox(JFrame parent, final CommandWord command,
            String[] dialogLabels) {

        JLabel label1, label2, label3, label4, label5;
        JPanel panel, p;

        allInput = new ArrayList<String>();
        final int numOfElements = dialogLabels.length;
        panel = new JPanel(new GridLayout(numOfElements + 1, 2, 0, 15));
        panel.setBorder(BorderFactory.createEmptyBorder(15, 15, 15, 15));
```

```
label1 = new JLabel(dialogLabels[0]);
label2 = new JLabel(dialogLabels[1]);
text1 = new JTextField(20);
text2 = new JTextField(20);
p = new JPanel(new FlowLayout(FlowLayout.CENTER) );
p.add(label1);
panel.add(p);
panel.add(text1);

p = new JPanel(new FlowLayout(FlowLayout.CENTER) );
p.add(label2);
panel.add(p);
panel.add(text2);
if (numOfElements == 3) {
  label3 = new JLabel(dialogLabels[2]);
  text3 = new JTextField(20);

  p = new JPanel(new FlowLayout(FlowLayout.CENTER) );
  p.add(label3);
  panel.add(p);
  panel.add(text3);

} else if (numOfElements == 5) {
  label3 = new JLabel(dialogLabels[2]);
  label4 = new JLabel(dialogLabels[3]);
  label5 = new JLabel(dialogLabels[4]);
  text3 = new JTextField(20);
  text4 = new JTextField(20);
  text5 = new JTextField(20);


  p = new JPanel(new FlowLayout(FlowLayout.CENTER) );
  p.add(label3);
  panel.add(p);
  panel.add(text3);
  p = new JPanel(new FlowLayout(FlowLayout.CENTER) );
  p.add(label4);
  panel.add(p);
  panel.add(text4);
  p = new JPanel(new FlowLayout(FlowLayout.CENTER) );
  p.add(label5);
  panel.add(p);
  panel.add(text5);
}

JButton OKButton = new JButton("OK");
JButton CancelButton = new JButton("Cancel");
dialogBox = new JDialog(parent);

JPanel buttonPanel1 = new JPanel();
```

```
buttonPanel1.setLayout(new FlowLayout(FlowLayout.CENTER, 10, 0));
buttonPanel1.add(OKButton);
JPanel buttonPanel2 = new JPanel();
buttonPanel2.setLayout(new FlowLayout(FlowLayout.CENTER, 10, 0));
buttonPanel2.add(CancelButton);

panel.add(buttonPanel1);
panel.add(buttonPanel2);

dialogBox.getContentPane().add(panel, BorderLayout.CENTER);
dialogBox.setTitle(command.toString().toUpperCase());
dialogBox.setModal(true);
dialogBox.pack();
dialogBox.setPreferredSize(new Dimension(200, 250));

OKButton.addActionListener(new ActionListener() {

  public void actionPerformed(ActionEvent ae) {

    allInput.add(text1.getText());
    allInput.add(text2.getText());
    if (numOfElements == 3) {
      if (command == CommandWord.ADD_FILM) {
        StringTokenizer days =
            new StringTokenizer(text3.getText(), ", ");
        while (days.hasMoreTokens()) {
          allInput.add(days.nextToken());
        }
      } else {
        allInput.add(text3.getText());
      }
    } else if (numOfElements == 5) {
      allInput.add(text3.getText());
      allInput.add(text4.getText());
      allInput.add(text5.getText());
    }
    dialogBox.dispose();
  }
});

CancelButton.addActionListener(new ActionListener() {

  public void actionPerformed(ActionEvent ae) {
    dialogBox.dispose();
  }
});

// place the frame at the center of the screen and show
Dimension d = Toolkit.getDefaultToolkit().getScreenSize();
dialogBox.setLocation(d.width / 2 - dialogBox.getWidth() / 2,
            d.height / 2 - dialogBox.getHeight() / 2);

        dialogBox.setVisible(true);
    }


    /**
     * Creates a list of all the user inputs.
     */
    public ArrayList<String> getInput() {
        return allInput;
    }
}
```

UnrecognizedCommandExceiption.java

```
import java.io.Serializable;


/**
 * Captures an input that is not recognized as any valid command for the system.
 *
 * @author Fariha Nazmul
 * @version 01.11.10
 */
public class UnrecognizedCommandException
        extends Exception implements Serializable
{
    // The inputCommand that is unrecognized as a command.
    private String inputCommand;

    /**
     * Stores the details in error.
     * @param inputCommand The input that is unrecognized.
     */
    public UnrecognizedCommandException(String inputCommand)
    {
        this.inputCommand = inputCommand;
    }

    /**
     * Returns the command in error.
     * @return The key in error.
     */
    public String getInputCommand()
    {
        return inputCommand;
    }

    /**
```

```
     * Returns the string representation of the error.
     * @return A diagnostic string containing the input in error.
     */
    public String toString()
    {
        return "Unrecognized command: " + inputCommand.toUpperCase();
    }

    public String getMessage(){
        return "Unrecognized command: " + inputCommand.toUpperCase();
    }
}
```

InvalidCommandException.java

```
import java.io.Serializable;

/**
 * Captures an input that is not considered as a valid command
 * of the system against the legal mode to execute.
 *
 * @author Fariha Nazmul
 * @version 01.11.10
 */
public class InvalidCommandException extends Exception implements Serializable
{
    // The inputCommand that is invalid to run in the running mode.
    private String inputCommand;

    /**
     * Stores the details in error.
     * @param inputCommand The input that is invalid.
     */
    public InvalidCommandException(String inputCommand)
    {
        this.inputCommand = inputCommand;
    }

    /**
     * Returns the command in error.
     * @return The key in error.
     */
    public String getInputCommand()
    {
        return inputCommand;
    }

    /**
     * Returns the string representation of the error.
     * @return A diagnostic string containing the input in error.
     */
    public String toString()
    {
       return "Invalid mode to run the command: " + inputCommand.toUpperCase()
             + "\nAn error occurred while attempting " +
             "to run a privileged command.";
    }
}
```

Appendix B

Listing of the file catalogues on the writable CD attached to the report

- A catalogue **source** with all the Java files and the same files packed into one **.jar** file
- A catalogue **doc** with the complete javadoc documentation of all classes
- A BlueJ project “CinemaSeatsBookingSystem” with the complete solution of the system