SOFTWARE ENGINEERING FOUNDATIONS

FLORIAN BERGMANN
PERSON ID: H00020398

Assessment One: Cabin Manager

[October 25, 2010 at 23:14]

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

Part I

CREATION OF A CABIN MANAGER

1 INTRODUCTION

This report shall provide detailed information about the implementation of the cabin manager application.

1.1 ATTRIBUTES

Apart from the mandatory attributes, the following attributes were chosen to be implemented as well:

SIZE: The area of available space in the cabin.

CONDITION: The condition the cabin is in. Only the following values are allowed: `PERFECT`, `GOOD`, `FAIR`, `BAD`, `IN_SHAMBLES`, `UNKNOWN`. The limitation to these attributes is achieved by utilizing an `enumeration`.

1.2 COST CALCULATION

The calculation of a cabin's cost for one night, are calculated according to the following formula:

```
BASIC_COST + CONDITION_COST + FACILITIES_COST + SIZE_COST + (BED_TO_ROOM_RATIO
* BED_TO_ROOM_MULTIPLIER)
```

`BASIC_COST` and `BED_TO_ROOM_MULTIPLIER` are constants that can be set in the `cabin` class.

1.3 FREQUENCY REPORTS

The frequency-report provided outputs the number of cabins of a certain condition.

E.g. if two cabins are of the condition "`IN_SHAMBLES`" and one is of the condition "`GOOD`" the output would be as follows:

BAD	FAIR	GOOD	IN_SHAMBLES	PERFECT
0	0	1	2	0

1.4 STATUS REPORT

Even though not well-designed the application should meet the specification fully as all requirements were implemented and tested to function even in the case of incorrect input.

2 DIAGRAMS

2.1 CLASS DIAGRAM

The provided class diagram does not display `getters` and `setters` even though they are present for every non-final attribute present in the diagram.

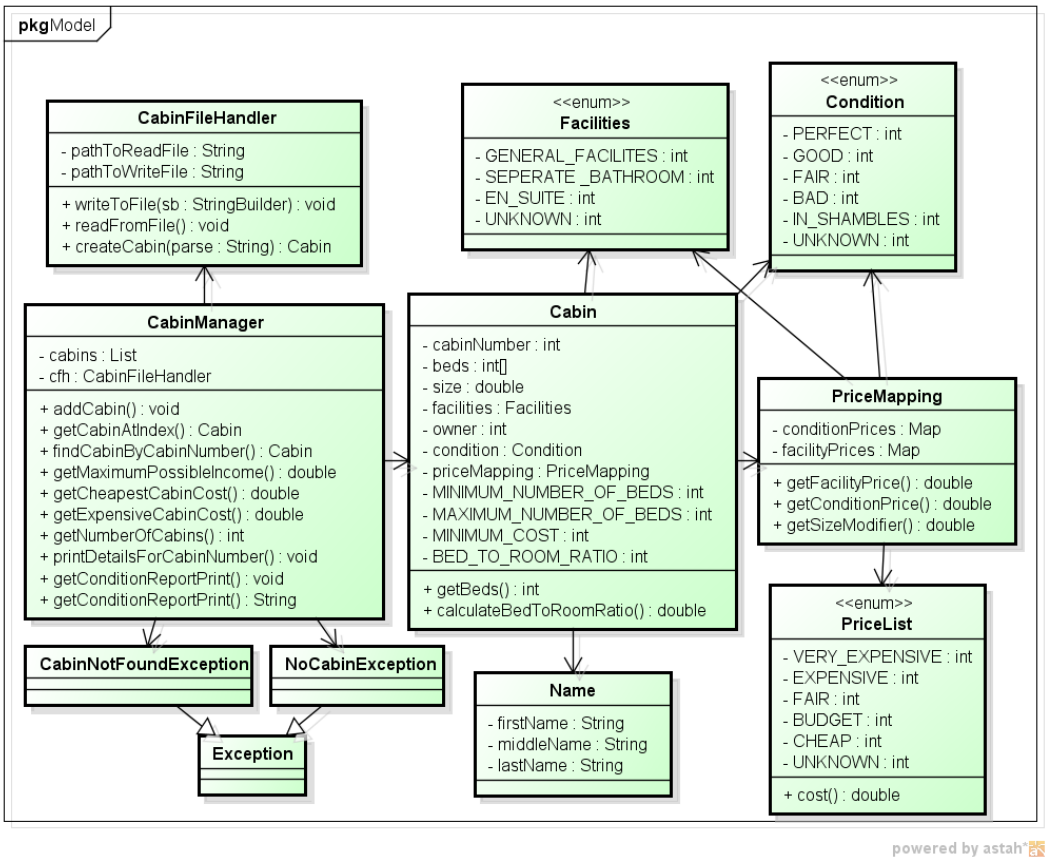

Figure 1: Class diagram of cabin manager

2.2 SEQUENCE DIAGRAM

The provided sequence diagram shows how the frequency report is generated and the printed to a file.

Even though the real implementation prints all other details as well (cabin details, overview) these have been left out of the sequence diagram to improve its clarity.

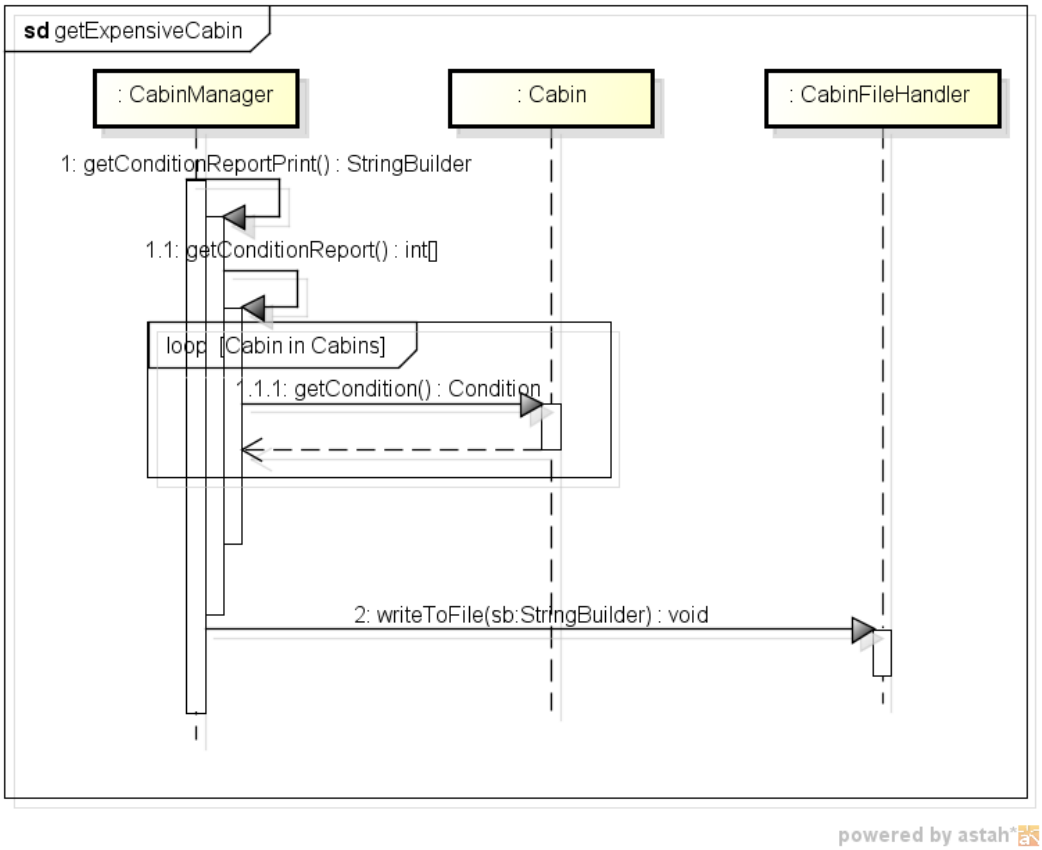

Figure 2: Sequence diagram of the printing of the frequency report

3 SOURCE CODE

Package: uk.heriotwatt.sef.model

Listing 1: uk.heriotwatt.sef.model.Cabin.java

```
package uk.heriotwatt.sef.model;

/**
 * Stores values associated with a cabin.
 *
 * @author fhb2
 *
 */
public class Cabin {

        public int cabinNumber;
        public int[] beds;
        public double size;
        public Facilities facilities;
        public Name owner;
        public Condition condition;

        private final int MINIMUM_NUMBER_OF_BEDS = 2;
        private final int MAXIMUM_NUMBER_OF_BEDS = 8;
        private final double BASIC_COST = 10;
        public final int BED_TO_ROOM_RATIO_MULTIPLIER = 5;

        private PriceMapping data;

        public Cabin() {
                this.data = new PriceMapping();
        }

        public Cabin(int cabinNumber, int[] numberOfBeds, double size,
                        Facilities facilities, Name owner, Condition condition
                            ) {
                super();
                this.cabinNumber = cabinNumber;
                this.beds = numberOfBeds;
                this.size = size;
                this.facilities = facilities;
```

```
        this.owner = owner;
        this.condition = condition;
        this.data = new PriceMapping();
}

/*
 * Getters and setters
 */

/**
 * @return The number of cabins stored in the manager.
 */
public int getCabinNumber() {
        return cabinNumber;
}

/**
 * Sets the cabin number.
 *
 * @param cabinNumber Number to be set.
 */
public void setCabinNumber(int cabinNumber) {
        this.cabinNumber = cabinNumber;
}

/**
 * @return Array of beds.
 */
public int[] getNumberOfBeds() {
        return beds;
}

/**
 * Sets the beds.
 *
 * @param numberOfBeds The new array of beds.
 */
public void setNumberOfBeds(int[] numberOfBeds) {
        if (numberOfBeds.length > 0) {
                int bedsInArray = this.calculateNumberOfBeds(
                    numberOfBeds);
                if (bedsInArray >= MINIMUM_NUMBER_OF_BEDS
                                && bedsInArray <=
                                    MAXIMUM_NUMBER_OF_BEDS) {
                        this.beds = numberOfBeds;
                } else {
                        throw new IllegalArgumentException(
```

```
                                        String.format(
                                                "Only between %d and %d beds can be placed in a cabin.",
                                                MINIMUM_NUMBER_OF_BEDS, MAXIMUM_NUMBER_OF_BEDS));
                }
        } else {
                throw new IllegalArgumentException(
                                "The number of beds must be greater than 0.");
        }
}

/**
 * @return The facilities of the cabin.
 */
public Facilities getFacilities() {
        return facilities;
}

/**
 * Attempts to set the facilites of the cabin.
 *
 * @param facilities
 */
public void setFacilities(Facilities facilities) {
        this.facilities = facilities;
}

/**
 * @return The owner of the cabin.
 */
public Name getOwner() {
        return owner;
}

/**
 * Sets the owner of the cabin.
 *
 * @param owner
 */
```

```
public void setOwner(Name owner) {
        this.owner = owner;
}


/**
 * @return The size of the cabin.
 */
public double getSize() {
        return size;
}

/**
 * Sets the size of the cabin.
 *
 * @param size The new size (must be bigger than 0)
 */
public void setSize(double size) {
        if (size >= 0) {
                this.size = size;
        } else {
                throw new IllegalArgumentException("Size must be
                    positive.");
        }
}

/**
 * The cost is calculated based on different factors:
 * - The condition.
 * - The facilities.
 * - The size.
 * - The beds/rooms present (The less beds per room the more expensive
    ).
 * The values associated with the first three are stored in {@link
    PriceMapping}
 *
 * @return The cost if the cabin.
 */
public double getCost() {
        double cost = BASIC_COST;

        double conditionModifier = this.data.getConditionPrice(this.
            condition);
        double faciltiesModifier = this.data.getFacilityPrice(this.
            facilities);
        double sizeModifier = this.data.getSizeModifier(this.size);
        double bedToRoomRatio = this.calculateRoomToBedRatio();
```

```
        cost = BASIC_COST + conditionModifier + faciltiesModifier
                        + sizeModifier
                        + (BED_TO_ROOM_RATIO_MULTIPLIER *
                            bedToRoomRatio);

        return cost;
}

/**
 * @return The condition.
 */
public Condition getCondition() {
        return condition;
}

/**
 * Sets the condition of the cabin.
 *
 * @param condition New condition to be set.
 */
public void setCondition(Condition condition) {
        this.condition = condition;
}

/**
 * @return The number of beds in the cabin.
 */
public int getBeds()
{
        return this.calculateNumberOfBeds(this.beds);
}

/**
 * Calculates the room to bed ratio.
 * @return The room to bed ratio.
 */
public double calculateRoomToBedRatio() {
        int rooms = this.getNumberOfBeds().length;
        int beds = this.calculateNumberOfBeds(this.beds);
        double bedToRoomRatio = rooms / beds;
        return bedToRoomRatio;
}

private int calculateNumberOfBeds(int[] numberOfBeds) {
        int result = 0;
        for (int i : numberOfBeds) {
```

```
                        result += i;
                }
                return result;
        }
}
```

Listing 2: uk.heriotwatt.sef.model.CabinManager.java

```
package uk.heriotwatt.sef.model;

import java.util.ArrayList;
import java.util.Formatter;
import java.util.List;
import java.util.Locale;

public class CabinManager {

        private List<Cabin> cabins;

        private CabinFileHandler cfh;

        public CabinManager(CabinFileHandler cfh) {
                this.cabins = new ArrayList<Cabin>();
                this.cfh = cfh;
                this.cabins = this.cfh.readFromFile();
        }

        public void addCabin(Cabin cab) {
                this.cabins.add(cab);
        }

        public Cabin getCabinAtIndex(int index) {
                if (index < this.getNumberOfCabins()) {
                        return this.cabins.get(index);
                } else {
                        throw new IndexOutOfBoundsException();
                }
        }

        /**
         * Attempts to find a cabin with the provided cabinNumber in the cabin
             -List.
         *
         * @param cabinNumber
         *            The cabinnumber of the cabin to be returned
         * @return The first cabin in the list with the corresponding
             cabinnumber.
```

```
     * @throws CabinNotFoundException
     *             If no cabin with the provided number could be found.
     */
    public Cabin findCabinByCabinNumber(int cabinNumber)
                    throws CabinNotFoundException {
            Cabin cabinFound = null;
            for (Cabin cabin : this.cabins) {
                    if (cabin.cabinNumber == cabinNumber) {
                            cabinFound = cabin;
                            break;
                    }
            }
            if (cabinFound != null) {
                    return cabinFound;
            } else {
                    throw new CabinNotFoundException(String.format(
                                    "The cabin with number %d was not in
                                        the list.",
                                    cabinNumber));
            }
    }

    /**
     * Returns the maximum possible income that could be achieved.
         Therefore the
     * cost for all cabins are added up.
     *
     * @return The added cost of all cabins.
     */
    public double getMaximumPossibleIncome() {
            double result = 0;
            for (Cabin cabin : this.cabins) {
                    result += cabin.getCost();
            }
            return result;
    }

    /**
     * Returns the cost for the cheapest cabin.
     *
     * @return The cost of the cheapest cabin.
     * @throws NoCabinsException
     */
    public double getCheapestCabinCost() throws NoCabinsException {
            // TODO: Empty array;
            if (this.cabins.size() > 0) {
                    Cabin cheapestCab = null;
```

```
                for (Cabin cab : this.cabins) {
                        if (cheapestCab == null) {
                                cheapestCab = cab;
                        }
                        if (cab.getCost() < cheapestCab.getCost()) {
                                cheapestCab = cab;
                        }
                }
                return cheapestCab.getCost();
        } else {
                throw new NoCabinsException(
                                "There are no cabins present. Insert
                                    cabins first.");
        }

}

/**
 * Returns the cost for the most expensive cabin.
 *
 * @return The cost of the most expensive cabin.
 * @throws NoCabinsException
 */
public double getExpensiveCabinCost() throws NoCabinsException {
        if (this.cabins.size() > 0) {
                Cabin expensiveCab = null;
                for (Cabin cab : this.cabins) {
                        if (expensiveCab == null) {
                                expensiveCab = cab;
                        }
                        if (cab.getCost() > expensiveCab.getCost()) {
                                expensiveCab = cab;
                        }
                }
                return expensiveCab.getCost();
        } else {
                throw new NoCabinsException(
                                "There are no cabins present. Insert
                                    cabins first.");
        }
}

/**
 * Returns the number of cabins currently registered in the list.
 *
 * @return The number of cabins.
 */
```

```
public int getNumberOfCabins() {
        return this.cabins.size();
}

/**
 * Prints the details of a specific cabin that is specified by its
    cabin
 * number.
 *
 * @param cabinNumber
 *            The cabinnumber of the cabin whose details should be
    printed.
 */
public void printDetailsForCabinNumber(int cabinNumber) {
        try {
                Cabin cab = this.findCabinByCabinNumber(cabinNumber);
                this.printCabDetails(cab);
        } catch (CabinNotFoundException e) {
                System.out
                                .println(String
                                                .format("Could not
                                                    find the cabin for
                                                     number %d. No
                                                    details printed.",
                                                            cabinNumber
                                                                ))
                                                                ;
        }
}

/**
 * Prints the details for all cabins to the standard output.
 */
public void printAllCabins() {
        StringBuilder sb = getAllCabinDetails();
        System.out.println(sb.toString());
}

/**
 * Acquires all information from the reports and prints the to a file.
 */
public void printReportsToFile() {
        String printString = "";
        printString += "OVERVIEW OF CABIN DETAILS:\n\n";
        StringBuilder sb = getAllCabinDetails();
        printString += sb.toString();
        printString += "SINGLE CABIN INFORMATION: \n\n";
```

```
        for (Cabin cab : this.cabins) {
                StringBuilder db = this.getCabinDetails(cab);
                printString += db.toString();
        }
        try {
                printString += "MOST EXPENSIVE CABIN: "
                                + this.getExpensiveCabinCost() + "\n\n
                                    ";
                printString += "CHEAPEEST CABIN: " + this.
                    getCheapestCabinCost()
                                + "\n\n";
        } catch (NoCabinsException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }
        printString += "MAXIMUM INCOME PER NIGHT: "
                        + this.getMaximumPossibleIncome() + "\n\n";
        printString += "CONDITION REPORT: \n\n";
        printString += this.getConditionReportPrint().toString();
        cfh.writeToFile(printString);
}

/**
 * Returns the details about all cabins
 *
 * @return A stringbuilder with formatted output.
 */
private StringBuilder getAllCabinDetails() {
        StringBuilder sb = new StringBuilder();
        Formatter formatter = new Formatter(sb, Locale.UK);
        formatter.format("%1$10s | %2$10s | %3$20s | %4$5s %n", "
            NUMBER",
                        "OWNER", "FACILITIES", "BEDS");
        for (Cabin cab : this.cabins) {
                // TODO Return the initials of the owner.
                formatter.format("%1$10d | %2$10s | %3$20s | %4$5d %n"
                    ,
                                cab.getCabinNumber(), cab.getOwner().
                                    getInitials(), cab
                                                .getFacilities().
                                                    toString().
                                                    toLowerCase(),
                                cab.getBeds());
        }
        formatter.format("%n");
        return sb;
}
```

```
/**
 * Prints the details of one cabin.
 *
 * @param cab
 *            The cabin which details should be printed.
 */
public void printCabDetails(Cabin cab) {
        StringBuilder sb = getCabinDetails(cab);
        System.out.println(sb.toString());
}

/**
 * Returns a formatted condition report.
 *
 * @return A strinbuilder containing the formatted condition report.
 */
public StringBuilder getConditionReportPrint() {
        int[] conRep = getConditionReport();
        StringBuilder sb = new StringBuilder();
        Formatter formatter = new Formatter(sb, Locale.UK);
        formatter.format("%1$15s | %2$15s | %3$15s | %4$15s | %5$15s %
            n",
                        Condition.BAD.toString(), Condition.FAIR.
                            toString(),
                        Condition.GOOD.toString(), Condition.
                            IN_SHAMBLES.toString(),
                        Condition.PERFECT.toString(), Condition.
                            UNKNOWN.toString());
        formatter.format("%1$15d | %2$15d | %3$15d | %4$15d | %5$15d %
            n",
                        conRep[0], conRep[1], conRep[2], conRep[3],
                            conRep[4],
                        conRep[5]);
        formatter.format("%n");
        return sb;
}

/**
 * Returns the values of the condition report.
 *
 * @return String array containing the number of cabins of a certain
 *         condition.
 */
public int[] getConditionReport() {
        int size = Condition.values().length;
        int[] frequencyOfConditions = new int[size];
```

```
                for (Cabin cabin : this.cabins) {
                        switch (cabin.condition) {
                        case BAD:
                                frequencyOfConditions[0]++;
                                break;
                        case FAIR:
                                frequencyOfConditions[1]++;
                                break;
                        case GOOD:
                                frequencyOfConditions[2]++;
                                break;
                        case IN_SHAMBLES:
                                frequencyOfConditions[3]++;
                                break;
                        case PERFECT:
                                frequencyOfConditions[4]++;
                                break;
                        case UNKNOWN:
                                frequencyOfConditions[5]++;
                                break;
                        default:
                                break;
                        }
                }
                return frequencyOfConditions;
        }

        /**
         * Returns the details of one cabin.
         *
         * @param cab
         *            The cabin which details should be returned
         * @return A stringbuilder with formatted output.
         */
        private StringBuilder getCabinDetails(Cabin cab) {
                StringBuilder sb = new StringBuilder();
                Formatter formatter = new Formatter(sb, Locale.UK);
                formatter
                                .format("%1$10s | %2$15s | %3$20s | %4$15s |
                                    %5$5s | %6$5s | %7$5s | %8$5s %n",
                                                "NUMBER", "OWNER", "FACILITIES
                                                    ", "CONDITION", "BEDS",
                                                "ROOMS", "SIZE", "COST");
                formatter
                                .format("%1$10d | %2$15s | %3$20s | %4$15s |
                                    %5$5d | %6$5d | %7$5.2f | %8$5.2f %n",
```

```
                                                cab.getCabinNumber(), cab.
                                                    getOwner()
                                                                .
                                                                    getFirstAndLastName
                                                                    (), cab.
                                                                    getFacilities
                                                                    ()
                                                                .toString().
                                                                    toLowerCase
                                                                    (), cab.
                                                                    getCondition
                                                                    ()
                                                                .toString().
                                                                    toLowerCase
                                                                    (), cab.
                                                                    getBeds(),
                                                                     cab
                                                                .
                                                                    getNumberOfBeds
                                                                    ().length,
                                                                     cab.
                                                                    getSize(),
                                                                     cab
                                                                .getCost());
                formatter.format("%n");
                return sb;
        }

}
```

Listing 3: uk.heriotwatt.sef.model.Condition.java

```
package uk.heriotwatt.sef.model;

/**
 * Stores the different possibilities of conditions.
 *
 * @author fhb2
 *
 */
public enum Condition {

        PERFECT, GOOD, FAIR, BAD, IN_SHAMBLES, UNKNOWN

}
```

Listing 4: uk.heriotwatt.sef.model.Facilities.java

```java
package uk.heriotwatt.sef.model;

/**
 * Stores the different possibilities of facilities.
 *
 * @author fhb2
 *
 */
public enum Facilities {

        GENERAL_FACILITES, SEPERATE_BATHROOM, EN_SUITE, UNKNOWN

}
```

Listing 5: uk.heriotwatt.sef.model.PriceList.java

```java
package uk.heriotwatt.sef.model;

/**
 * Stores the price modifiers for certain discrete price categories.
 *
 * @author fhb2
 *
 */
public enum PriceList {

        VERY_EXPENSIVE(10.0), EXPENSIVE(7.5), FAIR(5.0), BUDGET(2.5), CHEAP
            (1.0), UNKNOWN(
                        0.0);

        private final double cost;

        private PriceList(double cost) {
                this.cost = cost;
        }

        public double cost() {
                return this.cost;
        }

}
```

Listing 6: uk.heriotwatt.sef.model.PriceMapping.java

```java
package uk.heriotwatt.sef.model;
```

```
import java.util.HashMap;
import java.util.Map;

/**
 * Class to seperate the pricing mapping from the information of the cabin.
 *
 * @author florian
 *
 */
public class PriceMapping {

        private Map<Condition, Double> conditionPrices;
        private Map<Facilities, Double> facilityPrices;

        public PriceMapping() {
                this.initializeConditionPriceMapping();
                this.initializeFacilityPriceMapping();
        }

        public Map<Condition, Double> getConditionPrices() {
                return conditionPrices;
        }

        public Map<Facilities, Double> getFacilityPrices() {
                return facilityPrices;
        }

        /*
         * Getters and setters
         */

        public double getFacilityPrice(Facilities facilities) {
                return facilityPrices.get(facilities);
        }

        public double getConditionPrice(Condition condition) {
                return conditionPrices.get(condition);
        }

        /**
         * Adds Condition - Price pairs to a map. Will be used in the getCost
             ()
         * method.
         */
        private void initializeConditionPriceMapping() {
                this.conditionPrices = new HashMap<Condition, Double>();
                this.conditionPrices.put(Condition.PERFECT,
```

```
                        PriceList.VERY_EXPENSIVE.cost());
        this.conditionPrices.put(Condition.GOOD, PriceList.EXPENSIVE.
            cost());
        this.conditionPrices.put(Condition.FAIR, PriceList.FAIR.cost()
            );
        this.conditionPrices.put(Condition.BAD, PriceList.BUDGET.cost
            ());
        this.conditionPrices.put(Condition.IN_SHAMBLES, PriceList.
            CHEAP.cost());
}

/**
 * Adds Facilities - Price pairs to a map. Will be used in the getCost
    ()
 * method.
 */
private void initializeFacilityPriceMapping() {
        this.facilityPrices = new HashMap<Facilities, Double>();
        this.facilityPrices.put(Facilities.EN_SUITE,
                        PriceList.VERY_EXPENSIVE.cost());
        this.facilityPrices.put(Facilities.SEPERATE_BATHROOM,
                        PriceList.FAIR.cost());
        this.facilityPrices.put(Facilities.GENERAL_FACILITES,
                        PriceList.BUDGET.cost());
}

/**
 * Return the size modifier that can be used to calculate a price for
    a
 * cabin.
 *
 * @param size
 *            The size of the cabin.
 * @return The size modifier according to the provided size of a room.
 */
public double getSizeModifier(double size) {
        if (size < 20) {
                return PriceList.BUDGET.cost();
        } else if (size >= 20 && size < 30) {
                return PriceList.CHEAP.cost();
        } else if (size >= 30 && size < 40) {
                return PriceList.FAIR.cost();
        } else if (size >= 40 && size < 50) {
                return PriceList.EXPENSIVE.cost();
        } else {
                return PriceList.VERY_EXPENSIVE.cost();
        }
```

```
        }
}
```

Listing 7: uk.heriotwatt.sef.model.CabinFileHandler.java

```
package uk.heriotwatt.sef.model;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Date;
import java.util.LinkedList;
import java.util.List;
import java.util.Scanner;

public class CabinFileHandler {

        private String pathToReadFile;
        private String pathToReportFile;

        public CabinFileHandler(String pathReadFile, String pathWriteFile) {
                this.pathToReadFile = pathReadFile;
                this.pathToReportFile = pathWriteFile;
        }

        public void writeToFile(String sb) {
                try {
                        File file = new File(pathToReportFile);
                        PrintWriter pw = new PrintWriter(file);
                        pw.write(sb);
                        pw.flush();
                        pw.close();
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }

        public List<Cabin> readFromFile() {
                // TODO Ignore the comments in a file. (Denoted by #)
                List<Cabin> cabinList = new LinkedList<Cabin>();
                try {
                        File file = new File(this.pathToReadFile);
                        Scanner scanner = new Scanner(file);
                        while (scanner.hasNext()) {
                                String nextLine = scanner.nextLine();
                                if (nextLine.trim().startsWith("#")) {
```

```
                                System.out.println("Ignoring a
                                    commented out line.");
                        } else {
                                try {
                                        Cabin cabin = this.createCabin
                                            (nextLine);
                                        cabinList.add(cabin);
                                } catch (IllegalArgumentException e) {
                                        // TODO: handle exception
                                }
                        }
                }
        } catch (FileNotFoundException e) {
                e.printStackTrace();
        } catch (IOException e) {
                e.printStackTrace();
        }
        return cabinList;
}

public Cabin createCabin(String nextLine) {
        try {
                Cabin cabin = null;
                String[] splitList = nextLine.split(",");
                int cabinNumber = Integer.parseInt(splitList[0]);
                double size = Double.parseDouble(splitList[1]);
                Facilities facilities = Facilities.valueOf(splitList
                    [2]);
                Condition condition = Condition.valueOf(splitList[3]);
                Name name = new Name(splitList[4], splitList[5],
                    splitList[6]);
                int[] beds = new int[splitList.length - 7];
                for (int i = 7; i < splitList.length; i++) {
                        beds[i - 7] = Integer.parseInt(splitList[i]);
                }
                cabin = new Cabin(cabinNumber, beds, size, facilities,
                     name,
                                condition);
                return cabin;
        } catch (NumberFormatException e) {
                System.out.println("There was an error when parsing a
                    number!");
                e.printStackTrace();
                throw new IllegalArgumentException("Parsing failed.");
        } catch (IllegalArgumentException e) {
                System.out
```

```
                                        .println("A provided argument was not
                                            the expected type.");
                        e.printStackTrace();
                        throw new IllegalArgumentException("Parsing failed.");
                }
        }

}
```

Listing 8: uk.heriotwatt.sef.model.CabinNotFoundException.java

```
package uk.heriotwatt.sef.model;

public class CabinNotFoundException extends Exception {

        /**
         * Generated serialVersionUID to allow serialisation.
         */
        private static final long serialVersionUID = -7740644730079198039L;

        public CabinNotFoundException(String msg) {
                super(msg);
        }
}
```

Listing 9: uk.heriotwatt.sef.model.NoCabinsException.java

```
package uk.heriotwatt.sef.model;

public class NoCabinsException extends Exception {

        /**
         * Generated serialVersionUID to allow serialisation.
         */
        private static final long serialVersionUID = 2274177224545932291L;

        public NoCabinsException(String msg) {
                super(msg);
        }
}
```

Listing 10: uk.heriotwatt.sef.model.Name.java

```
package uk.heriotwatt.sef.model;

//First Name class
```

```
//F21SF - Monica
public class Name {
        private String firstName;
        private String middleName;
        private String lastName;

        // constructor to create object with first, middle and last name
        // if there isn't a middle name, that parameter could be an empty
            String
        public Name(String fName, String mName, String lName) {
                firstName = fName;
                middleName = mName;
                lastName = lName;
        }

        // returns the first name
        public String getFirstName() {
                return firstName;
        }

        // returns the last name
        public String getLastName() {
                return lastName;
        }

        // change the last name to the value provided in the parameter
        public void setLastName(String ln) {
                lastName = ln;
        }

        // returns the first name then a space then the last name
        public String getFirstAndLastName() {
                return firstName + " " + lastName;
        }

        // returns the last name followed by a comma and a space
        // then the first name
        public String getLastCommaFirst() {
                return lastName + ", " + firstName;
        }

        // returns name in the format initial, period, space, lastname
        public String getInitPeriodLast() {
                return firstName.charAt(0) + ". " + lastName;
        }

        public String getInitials() {
```

```
                return firstName.charAt(0) + ". " + lastName.charAt(0) + ".";
        }
}
```

4

EXAMPLE OUTPUT

The output printed to file will look like the following[1]:

Listing 11: Example output-file

```
OVERVIEW OF CABIN DETAILS:

    NUMBER |      OWNER |           FACILITIES |  BEDS
         1 |      J. S. |             en_suite |     2
         4 |      C. N. |    general_facilites |     4
         5 |      S. G. |             en_suite |     3
         6 |      U. O. |    general_facilites |     6
         7 |      P. N. |    seperate_bathroom |     8
         8 |      R. C. |    seperate_bathroom |     4
         9 |      B. W. |    general_facilites |     4
        10 |      G. J. |             en_suite |     8

SINGLE CABIN INFORMATION:

    NUMBER |           OWNER |           FACILITIES |       CONDITION |  BEDS
        | ROOMS |  SIZE |  COST
         1 |        John Sly |             en_suite |         perfect |     2
             |     2 | 10.00 | 37.50

    NUMBER |           OWNER |           FACILITIES |       CONDITION |  BEDS
        | ROOMS |  SIZE |  COST
         4 | Conchobar Nessa |    general_facilites |            good |     4
             |     3 | 35.00 | 25.00

    NUMBER |           OWNER |           FACILITIES |       CONDITION |  BEDS
        | ROOMS |  SIZE |  COST
         5 |        Sie Gurd |             en_suite |            fair |     3
             |     1 | 45.00 | 32.50

    NUMBER |           OWNER |           FACILITIES |       CONDITION |  BEDS
        | ROOMS |  SIZE |  COST
         6 | Uthuce Odysseus |    general_facilites |     in_shambles |     6
             |     3 | 55.00 | 23.50
```

1 Unfortunately the page size forces line breaks.

```
    NUMBER |           OWNER |           FACILITIES |       CONDITION |  BEDS
        | ROOMS |  SIZE |  COST
         7 |  Perseus Nestor |    seperate_bathroom |         perfect |     8
             |     4 | 39.00 | 30.00

    NUMBER |           OWNER |           FACILITIES |       CONDITION |  BEDS
        | ROOMS |  SIZE |  COST
         8 |   Robert Cotton |    seperate_bathroom |            fair |     4
             |     3 | 38.00 | 25.00

    NUMBER |           OWNER |           FACILITIES |       CONDITION |  BEDS
        | ROOMS |  SIZE |  COST
         9 |        Beo Wulf |    general_facilites |             bad |     4
             |     2 | 38.50 | 20.00

    NUMBER |           OWNER |           FACILITIES |       CONDITION |  BEDS
        | ROOMS |  SIZE |  COST
        10 |  Grimur Jonsson |            en_suite |            good |     8
            |     3 | 27.00 | 28.50

MOST EXPENSIVE CABIN: 37.5

CHEAPEEST CABIN: 20.0

MAXIMUM INCOME PER NIGHT: 222.0

CONDITION REPORT:

           BAD |           FAIR |           GOOD |     IN_SHAMBLES |
                      PERFECT
             1 |              2 |              2 |               1 |
                               2
```

5 TESTING REPORT

The application is delivered with a set of test cases that all pass: they are printed in Appendix A.

A short list of noteworthy test cases shall be provided as an overview:

FILE HANDLING:
- Create new Cabin from valid input string.
- Create new cabin from invalid input string (throws exception).

CABIN:
- Set number of beds (too few, too many, valid numbers).
- Test cost calculation algorithm.

CABIN MANAGER:
- Add cabin.
- Get cabin (with and without any cabins).
- Find cabin by number (cabin is in list and cabin is not in list).
- Get cheapest and most expensive cabin.

Part II

APPENDIX

APPENDIX

Package: uk.heriotwatt.sef.model.tests

Listing 12: uk.heriotwatt.sef.model.tests.CabinFileHandlerTests.java

```
package uk.heriotwatt.sef.model.tests;

import junit.framework.Assert;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import uk.heriotwatt.sef.model.Cabin;
import uk.heriotwatt.sef.model.CabinFileHandler;

public class CabinFileHandlerTests {

        private CabinFileHandler fileHandler;

        @Before
        public void setUp() throws Exception {
                this.fileHandler = new CabinFileHandler("", "");
        }

        @After
        public void tearDown() throws Exception {
        }

        @Test
        public void testCreateCabin()
        {
                String toParse = "1,10,EN_SUITE,IN_SHAMBLES,John,Jack,
                    MasterMind,2,2,2,2";
                Cabin cabin = fileHandler.createCabin(toParse);
                Assert.assertNotNull(cabin);
        }

        @Test(expected=IllegalArgumentException.class)
        public void testCreateCabinFaultyValues()
        {
```

```
                String toParse = "1,10,EN_SITE,IN_SHAMBLES,John,Jack,
                    MasterMind,2,2,2,2";
                Cabin cabin = fileHandler.createCabin(toParse);
        }
}
```

Listing 13: uk.heriotwatt.sef.model.tests.CabinManagerTests.java

```
/**
 *
 */
package uk.heriotwatt.sef.model.tests;

import static org.junit.Assert.*;

import java.util.LinkedList;

import junit.framework.Assert;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import static org.mockito.Mockito.*;

import uk.heriotwatt.sef.model.Cabin;
import uk.heriotwatt.sef.model.CabinFileHandler;
import uk.heriotwatt.sef.model.CabinManager;
import uk.heriotwatt.sef.model.CabinNotFoundException;
import uk.heriotwatt.sef.model.Condition;
import uk.heriotwatt.sef.model.Facilities;
import uk.heriotwatt.sef.model.Name;
import uk.heriotwatt.sef.model.NoCabinsException;

/**
 * @author Florian Bergmann
 *
 */
public class CabinManagerTests {

        private CabinManager cabMan;
        private Cabin mockin;
        private Cabin mockin2;

        /**
         * @throws java.lang.Exception
         */
```

```
@Before
public void setUp() throws Exception {
        CabinFileHandler mockHandler = mock(CabinFileHandler.class);
        when(mockHandler.readFromFile()).thenReturn(new LinkedList());
        cabMan = new CabinManager(mockHandler);
        mockin = mock(Cabin.class);
        mockin2 = mock(Cabin.class);
}

/**
 * @throws java.lang.Exception
 */
@After
public void tearDown() throws Exception {
}

/**
 * Test method for {@link uk.heriotwatt.sef.model.CabinManager#
     addCabin(uk.heriotwatt.sef.model.Cabin)}.
 */
@Test
public void testAddCabin() {
        Cabin cab = new Cabin(1, new int[] {2,3}, 55.0, Facilities.
            EN_SUITE, new Name("Test", "Test", "Test"), Condition.GOOD
            );
        cabMan.addCabin(cab);
        Assert.assertEquals(1, cabMan.getNumberOfCabins());
}

/**
 * Test method for {@link uk.heriotwatt.sef.model.CabinManager#
     getCabinAtIndex(int)}.
 */
@Test
public void testGetCabinAtIndex() {
        Cabin cab = new Cabin(1, new int[] {2,3}, 55.0, Facilities.
            EN_SUITE, new Name("Test", "Test", "Test"), Condition.GOOD
            );
        cabMan.addCabin(cab);
        Cabin cabIndex = cabMan.getCabinAtIndex(0);
        Assert.assertEquals(cabIndex, cab);
}

@Test(expected=IndexOutOfBoundsException.class)
public void testGetCabinAtIndexWrongIndex() {
        cabMan.getCabinAtIndex(1);
}
```

```
/**
 * Test method for {@link uk.heriotwatt.sef.model.CabinManager#
     findCabinByCabinNumber(int)}.
 */
@Test
public void testFindCabinByCabinNumber() {
        Cabin cab = new Cabin(1, new int[] {2,3}, 55.0, Facilities.
            EN_SUITE, new Name("Test", "Test", "Test"), Condition.GOOD
            );
        cabMan.addCabin(cab);
        Cabin cabFound = null;
        try {
                cabFound = cabMan.findCabinByCabinNumber(1);
        } catch (CabinNotFoundException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }
        Assert.assertEquals(cab, cabFound);
}

@Test(expected=CabinNotFoundException.class)
public void testFindCabinByCabinNumberWrongNumber() throws
    CabinNotFoundException {
        cabMan.findCabinByCabinNumber(1);
}

/**
 * Test method for {@link uk.heriotwatt.sef.model.CabinManager#
     getMaximumPossibleIncome()}.
 */
@Test
public void testGetMaximumPossibleIncome() {
        when(mockin.getCost()).thenReturn(50.0);
        when(mockin2.getCost()).thenReturn(40.0);
        cabMan.addCabin(mockin);
        cabMan.addCabin(mockin2);
        double getMaxIncome = cabMan.getMaximumPossibleIncome();
        Assert.assertEquals(90, getMaxIncome, 0);
}

/**
 * Test method for {@link uk.heriotwatt.sef.model.CabinManager#
     getCheapestCabinCost()}.
 */
@Test
public void testGetCheapestCabinCost() {
```

```
                when(mockin.getCost()).thenReturn(50.0);
                when(mockin2.getCost()).thenReturn(40.0);
                cabMan.addCabin(mockin);
                cabMan.addCabin(mockin2);
                double cheapest = 0;
                try {
                        cheapest = cabMan.getCheapestCabinCost();
                } catch (NoCabinsException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
                Assert.assertEquals(40, cheapest, 0);
        }

        @Test(expected=NoCabinsException.class)
        public void testGetCheapestCabinCostNoCabins() throws
            NoCabinsException {
                cabMan.getCheapestCabinCost();
        }

        /**
         * Test method for {@link uk.heriotwatt.sef.model.CabinManager#
             getExpensiveCabinCost()}.
         */
        @Test
        public void testGetExpensiveCabinCost() {
                when(mockin.getCost()).thenReturn(50.0);
                when(mockin2.getCost()).thenReturn(40.0);
                cabMan.addCabin(mockin);
                cabMan.addCabin(mockin2);
                double mostExpensive = 0;
                try {
                        mostExpensive = cabMan.getExpensiveCabinCost();
                } catch (NoCabinsException e) {
                        // TODO: handle exception
                }
                Assert.assertEquals(50, mostExpensive, 0);
        }

        @Test(expected=NoCabinsException.class)
        public void testGetExpensiveCabinCostNoCabins() throws
            NoCabinsException {
                cabMan.getExpensiveCabinCost();
        }
}
```

Listing 14: uk.heriotwatt.sef.model.tests.CabinTests.java

```
package uk.heriotwatt.sef.model.tests;


import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;

import uk.heriotwatt.sef.model.Cabin;
import uk.heriotwatt.sef.model.Condition;
import uk.heriotwatt.sef.model.Facilities;
import uk.heriotwatt.sef.model.Name;
import uk.heriotwatt.sef.model.PriceMapping;

public class CabinTests {

        private Cabin cabin;

        @Before
        public void setUp() throws Exception {
                cabin = new Cabin();
        }

        @Test
        public void testSetNumberOfBeds()
        {
                int[] numberToSet = new int[] {2};
                cabin.setNumberOfBeds(numberToSet);
                int[] numberSet = cabin.getNumberOfBeds();
                Assert.assertArrayEquals(numberToSet, numberSet);

                numberToSet = new int[] {2, 2, 4};
                cabin.setNumberOfBeds(numberToSet);
                numberSet = cabin.getNumberOfBeds();
                Assert.assertArrayEquals(numberToSet, numberToSet);
        }

        @Test(expected=IllegalArgumentException.class)
        public void testSetNumberOfBedsWithTooFewBeds()
        {
                int[] numberToSet = new int[] {1};
                cabin.setNumberOfBeds(numberToSet);
        }

        @Test(expected=IllegalArgumentException.class)
        public void testSetNumberOfBedsWithTooManyBeds()
        {
```

```
                int[] numberToSet = new int[] {9};
                cabin.setNumberOfBeds(numberToSet);
        }

        @Test
        public void testCostForFacilites()
        {
                PriceMapping pm = new PriceMapping();

                Facilities fac = Facilities.EN_SUITE;
                Condition con = Condition.FAIR;
                int[] beds = new int[] {2,2};
                double size = 49.99;

                double basePrice = 10;
                double conPrice = pm.getConditionPrice(con);
                double facPrice = pm.getFacilityPrice(fac);
                double sizePrice = pm.getSizeModifier(size);

                double expectedPrice = basePrice + conPrice + facPrice +
                    sizePrice + 5 * (2/4);

                Cabin cab = new Cabin(1, beds,size, fac, new Name("Ho", "ho",
                    "ho"), con);

                double price = cab.getCost();

                Assert.assertEquals(expectedPrice, price, 0);
        }

}
```

BIBLIOGRAPHY