UOW-logo

Informatics Institute of Technology

School of Computing

Software Development II Coursework Report

Module : 4COSC010C.2: Software Development II (2023)

Date of submission :

Student ID : 20220945 / 2052735

Student First Name : Rachala

Student Surname : Gunawardana

Tutorial group (day, time, and tutor/s):

"I confirm that I understand what plagiarism / collusion / contract cheating is and have read and understood the section on Assessment Offences in the Essential Information for Students. The work that I have submitted is entirely my own. Any work from other authors is duly referenced and acknowledged."

Name : Rachala Ovin Gunawardana

Student ID : 20220945 / 2052735

# Self-assessment form

|  |  |  |
| --- | --- | --- |
| Task | Self-assessment (select one) | Comments |
| 1 | Fully implemented  Partially implemented  Not attempted | 1. Instead of 0 and 1 for booked values, true and false is used. |
| 2 | Fully implemented  Partially implemented  Not attempted |  |
| Insert here a screenshot of your welcome message and menu: | | |
| 3 | Fully implemented  Partially implemented  Not attempted |  |
| 4 | Fully implemented  Partially implemented  Not attempted |  |
| 5 | Fully implemented  Partially implemented  Not attempted |  |
| 6 | Fully implemented  Partially implemented  Not attempted |  |
| Insert here a screenshot of the seating plan: | | |
| 7 | Fully implemented  Partially implemented  Not attempted | The column and row variables are stored as a Position in the Ticket class. |
| 8 | Fully implemented  Partially implemented  Not attempted |  |
| 9 | Fully implemented  Partially implemented  Not attempted |  |
| 10 | Fully implemented  Partially implemented  Not attempted |  |
| 11 | Fully implemented  Partially implemented  Not attempted |  |
| 12 | Fully implemented  Partially implemented  Not attempted |  |

# Test Plan

Complete the test plan describing which testing you have performed on your program. Add as many rows as you need.

## Part A Testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test case / scenario | Input | Expected Output | Output | Pass/Fail |
| Main menu only accepts numbers from 0-6, and maps to their correct function. | 0, 1, 2, 3, 4, 5, 6 | Respective menu shown at specific numbers. | Their respective menus correctly open. | Pass  Fail |
| Validation for inputs with specific types (string, char, int) | * 1. A,B,C,D for row letters.   2. Integer within range for inputs with min/max   3. Inputs without validation | For inputs with specific types/data marked to retry when the input validation fails | Validator prints an error message with accepted inputs and prompts the user again | Pass  Fail |
| Session continuation | After a command ends, the user inputs either Y/N to continue or quit respectively. | * 1. To continue the session with old seat state when “y” or “yes” is entered.   2. To exit the program after “n” or “no” is entered | * 1. Continues the session on both “y” and “yes”.   2. Quits on “n” and “no”. | Pass  Fail |
| Buy/cancelling commands | row: A, col: 12 | Buy and cancel commands should toggle the value at seats[row][col] with true and false respectively. | Buy -> A12 is true.  Cancel -> A12 is false. | Pass  Fail |
| Seating plan | Buy seat at A12.  Continue the session and select 4 in the menu. | Every seat but A12 to be O, and A12 as X. | A12 -> X  Everything else -> O | Pass  Fail |
| Seating limits | Fill up all the seats and try to buy another one. | When all the seats are bought, warn the user about it instead of mutating other state. | No state mutate. Seat availability is checked before any state gets mutated. | Pass  Fail |

## Part B Testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test case / scenario | Input | Expected Output | Output | Pass/Fail |
| Tickets being stored | Placeholder Person data with A10 ticket | Tickets array to regrow and append Ticket with position A10 and placeholder person. | Data of both placeholder person and position are stored on the array without overwriting existing data | Pass  Fail |
| Bought tickets being written to a file | First name: Rachala  Surname: Gunawardana  Row: A  Column: 10 | A file named A10.txt being stored on CWD/tickets directory. | Ticket gets written to CWD/tickets/A10.txt with the content: “Ticket A10 @ 150LKR of Rachala Gunawardana [ovin.20220945@iit.ac.lk](mailto:ovin.20220945@iit.ac.lk)” | Pass  Fail |
|  |  |  |  | Pass  Fail |
|  |  |  |  | Pass  Fail |
|  |  |  |  | Pass  Fail |
|  |  |  |  | Pass  Fail |
|  |  |  |  | Pass  Fail |

Are there any specific parts of the coursework which you would like to get feedback?

|  |
| --- |
| How we mutate fixed length arrays are inefficient. Like in the case of how tickets are pushed, we could’ve used a growable type like ArrayLists. |

You will need to demonstrate your understanding of the submitted code. Your tutor will arrange a coursework demonstration. During the coursework demonstration, your tutor will ask you to execute your program and questions on your code.

**Failure to attend the demonstration will result in 0 for the coursework.**

# Code

## PlaneManagement.java

import java.io.IOException;

public class Main {

static String menuMessage = """

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Menu Options \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. Buy a seat

2. Cancel a seat

3. Find first available seat

4. Show seating plan

5. Print tickets information and total sales

6. Search ticket

0. Quit

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

""";

public static void main(String[] args) throws UnsupportedOperationException {

System.out.println("Welcome to the Plane Management system");

System.out.println(menuMessage);

final Seats seats = new Seats();

boolean firstTime = true;

while (true) {

if (!firstTime) {

System.out.println();

if (!Prompter.promptConditional("Do you want to continue to the session?")) break;

System.out.println(menuMessage);

}

final int option = Prompter.promptRangeInteger(0, 6, "Please select an option: ", "You need to select a valid number!");

switch (option) {

case 1:

buySeat(seats);

break;

case 2:

cancelSeat(seats);

break;

case 3:

findFirstAvailable(seats);

break;

case 4:

showSeatingPlan(seats);

break;

case 5:

printTicketsInfo(seats);

break;

case 6:

searchTicket(seats);

break;

case 0:

return;

default:

throw new UnsupportedOperationException();

}

firstTime = false;

}

}

static void buySeat(Seats seats) {

Position position;

// We want the input process to continue until the user choose an available seat

while (true) {

position = Position.fromUserInput();

if (seats.isSeatAvailable(position)) break;

System.out.println("This seat is already taken. Please try another one!");

}

try {

final Person person = Person.fromUserInput();

final Ticket ticket = seats.buySeat(position, person);

ticket.save();

} catch (IOException e) {

System.out.printf("Failed to write ticket file with error: %s", e);

}

}

static void cancelSeat(Seats seats) {

Position position;

// We want the input process to continue until the user choose an occupied seat

while (true) {

position = Position.fromUserInput();

if (!seats.isSeatAvailable(position)) break;

System.out.println("This seat is already free. Please try another one!");

}

seats.cancelSeat(position);

}

static void findFirstAvailable(Seats seats) {

try {

final Position firstAvailable = seats.getFirstAvailable();

System.out.printf("Seat at %s is available! ", firstAvailable.toDisplayString());

if (!Prompter.promptConditional("Do you want to buy it?")) return;

final Person person = Person.fromUserInput();

seats.buySeat(firstAvailable, person);

} catch (IllegalStateException e) {

System.out.println("It seems that all the seats are taken :(");

}

}

static void showSeatingPlan(Seats seats) {

System.out.println(seats.toDisplayString());

}

static void printTicketsInfo(Seats seats) {

System.out.println("\n\*\*\*\* Ticket sales information \*\*\*\*");

Ticket[] tickets = seats.getTickets();

if (tickets.length == 0) {

System.out.println("None of the tickets were sold :(");

return;

}

int total = 0;

final StringBuilder builder = new StringBuilder();

for (int i = 0; i < tickets.length; i++) {

Ticket ticket = tickets[i];

total += ticket.getPrice();

builder.append(String.format("%s. %s\n", i + 1, ticket.toDisplayString()));

}

System.out.println(builder);

System.out.printf("With total sales resulting in %sLKR!\n", total);

}

static void searchTicket(Seats seats) {

final Position select = Position.fromUserInput();

if (seats.isSeatAvailable(select)) {

System.out.println("This seat is available!");

return;

}

final Ticket ticket = seats.getTicket(select);

System.out.printf("This seats is already taken by %s", ticket.toDisplayString());

}

}

## Position.java

import java.util.Set;

public class Position {

public static final int ROW\_LOWER\_BOUND = 0;

public static final int ROW\_HIGHER\_BOUND = 3;

public static final String[] ROW\_LETTERS = {"A", "B", "C", "D"};

public static final int COL\_LOWER\_BOUND = 0;

public static final int COL\_HIGHER\_BOUND = 13;

public static final int COL\_HIGHER\_BOUND\_ALT = 11;

public int col;

public int row;

/\*\*

\* Creates a position reference from {@link #col} and {@link #row}.

\* This class guarantees that the underlying coordinates are valid in the {@link Seats} class.

\*

\* @throws IndexOutOfBoundsException When either col or row are out-of-bound with the seating layout constraints.

\*/

public Position(int col, int row) throws IndexOutOfBoundsException {

validatePosition(col, row);

this.col = col;

this.row = row;

}

static void validateRow(int row) throws IndexOutOfBoundsException {

if (row < ROW\_LOWER\_BOUND || row > ROW\_HIGHER\_BOUND) throw new IndexOutOfBoundsException();

}

/\*\*

\* Returns the max column index relative to the row passed.

\*

\* @throws IndexOutOfBoundsException when the row is out-of-bound

\*/

public static int getColumnUpperBound(int row) throws IndexOutOfBoundsException {

validateRow(row);

return (row == 0 || row == 3) ? COL\_HIGHER\_BOUND : COL\_HIGHER\_BOUND\_ALT;

}

/\*\*

\* @throws IndexOutOfBoundsException when either the row or the column is out of bound.

\*/

static void validatePosition(int col, int row) throws IndexOutOfBoundsException {

validateRow(row);

if (col < COL\_LOWER\_BOUND || col > getColumnUpperBound(row)) throw new IndexOutOfBoundsException();

}

/\*\*

\* Returns the row letter of related row index.

\*

\* @throws IndexOutOfBoundsException When row is out-of-bounds from seat layout.

\* @see #getUnsafeRowLetter(int)

\*/

public static char getUnsafeRowLetter(int row) throws IndexOutOfBoundsException {

return switch (row) {

case 0 -> 'A';

case 1 -> 'B';

case 2 -> 'C';

case 3 -> 'D';

default -> throw new IndexOutOfBoundsException();

};

}

/\*\*

\* Returns the related row index to the row letter.

\* This is useful for internal references.

\*/

static int getRelatedRowIndex(char rowLetter) throws IndexOutOfBoundsException {

return switch (rowLetter) {

case 'a', 'A' -> 0;

case 'b', 'B' -> 1;

case 'c', 'C' -> 2;

case 'd', 'D' -> 3;

default -> throw new IndexOutOfBoundsException();

};

}

public static Position fromUserInput() {

final String rowLettersHuman = String.join(", ", ROW\_LETTERS);

final String rowLetter = Prompter.promptValidString(

Set.of(ROW\_LETTERS),

String.format("Insert a row letter (%s): ", rowLettersHuman),

String.format("Please insert a valid row letter out of %s!", rowLettersHuman)

);

final int localRow = Position.getRelatedRowIndex(rowLetter.charAt(0));

final int colHumanLowerBound = COL\_LOWER\_BOUND + 1;

final int colHumanUpperBound = getColumnUpperBound(localRow) + 1;

final int localCol = Prompter.promptRangeInteger(

colHumanLowerBound,

colHumanUpperBound,

String.format("Insert a column number for the seat (min %s, max: %s): ", colHumanLowerBound, colHumanUpperBound),

String.format("Please insert a valid number between %s and %s!", colHumanLowerBound, colHumanUpperBound)

) - 1;

return new Position(localCol, localRow);

}

public int getPrice() {

if (col <= 5) return 200;

if (col <= 9) return 150;

return 180;

}

/\*\*

\* Safer variant of {@link #getUnsafeRowLetter(int)}

\*/

public char getRowLetter() {

return getUnsafeRowLetter(row);

}

/\*\*

\* Returns the display string formatted as "[ROW\_LETTER][COL]"

\* <br />

\* E.g. A12, C2

\*/

public String toDisplayString() {

return String.format("%s%s", getRowLetter(), col + 1);

}

@Override

public String toString() {

return String.format("Position { row: %s, col: %s }", row, col);

}

@Override

public boolean equals(Object o) {

if (this == o) return true;

if (!(o instanceof Position position)) return false;

return col == position.col && row == position.row;

}

@Override

public int hashCode() {

return Objects.hash(col, row);

}

}

## Seats.java

import java.util.Arrays;

import java.util.NoSuchElementException;

/\*\*

\* All the methods under this class are protected by {@link Position},

\* so none of the methods that digest a position doesn't throw {@link IndexOutOfBoundsException}.

\* <br /> <br />

\* This safety comes from how {@link Position} validates col and row at initialization time.

\*/

public class Seats {

private final boolean[][] seats = new boolean[4][];

private Ticket[] tickets = {};

public Seats() {

seats[0] = new boolean[14];

seats[1] = new boolean[12];

seats[2] = new boolean[12];

seats[3] = new boolean[14];

}

public Ticket[] getTickets() {

return tickets;

}

public boolean isSeatAvailable(Position pos) {

return !seats[pos.row][pos.col];

}

/\*\*

\* Returns the ticket occupied by the given position.

\*

\* @throws NoSuchElementException when this position doesn't have an accompanying ticket.

\*/

public Ticket getTicket(Position pos) throws NoSuchElementException {

return Arrays.stream(tickets).filter(ticket -> ticket.position.equals(pos)).findFirst().orElseThrow();

}

/\*\*

\* Returns the first available position in the seat layout.

\*

\* @throws IllegalStateException When there are no seats available.

\*/

public Position getFirstAvailable() throws IllegalStateException {

for (int rowIdx = 0; rowIdx < Position.ROW\_HIGHER\_BOUND + 1; rowIdx++) {

final boolean[] row = seats[rowIdx];

for (int colIdx = 0; colIdx < Position.getColumnUpperBound(rowIdx) + 1; colIdx++) {

final boolean colTaken = row[colIdx];

if (!colTaken) return new Position(colIdx, rowIdx);

}

}

throw new IllegalStateException();

}

/\*\*

\* @throws IllegalStateException If the seat of row and col is already taken.

\* @see #isSeatAvailable(Position)

\*/

public Ticket buySeat(Position position, Person person) throws IllegalStateException {

if (seats[position.row][position.col]) throw new IllegalStateException("Seat is already occupied");

seats[position.row][position.col] = true;

final Ticket ticket = new Ticket(position, person);

// We are not allowed to use LinkedLists or ArrayLists :(

// What we do is essentially creating a new array with the same content, but with a size of + 1,

// so we can replace the last item in the array with our new ticket.

tickets = Arrays.copyOf(tickets, tickets.length + 1);

tickets[tickets.length - 1] = ticket;

return ticket;

}

/\*\*

\* @throws IllegalStateException If the seat of row and col is free.

\* @see #isSeatAvailable(Position)

\*/

public void cancelSeat(Position pos) throws IllegalStateException {

if (!seats[pos.row][pos.col]) throw new IllegalStateException("Seat is already free");

seats[pos.row][pos.col] = false;

tickets = Arrays.stream(tickets).filter(x -> !x.position.equals(pos)).toArray(Ticket[]::new);

}

public String toDisplayString() {

final StringBuilder builder = new StringBuilder();

builder.insert(0, " ");

for (int colIdx = 1; colIdx <= Position.COL\_HIGHER\_BOUND + 1; colIdx++) {

builder.append(String.format(" %s", colIdx));

builder.append(" ".repeat(colIdx > 9 ? 2 : 3));

}

builder.append("\n");

for (int rowIdx = 0; rowIdx < seats.length; rowIdx++) {

final boolean[] cols = seats[rowIdx];

final StringBuilder rowBuilder = new StringBuilder();

final char rowLetter = Position.getUnsafeRowLetter(rowIdx);

rowBuilder.append(String.format("%s |", rowLetter));

for (boolean isSlotTaken : cols) {

rowBuilder.append(String.format(" %s |", isSlotTaken ? 'X' : 'O'));

}

if (cols.length == 12) {

rowBuilder.append(" |".repeat(2));

}

builder.append(rowBuilder.append("\n"));

}

return builder.toString();

}

@Override

public String toString() {

final String aSeats = Arrays.toString(seats[0]);

final String bSeats = Arrays.toString(seats[1]);

final String cSeats = Arrays.toString(seats[2]);

final String dSeats = Arrays.toString(seats[3]);

return String.format("Seats {\n\ta: %s,\n\tb: %s,\n\tc: %s,\n\td: %s\n}", aSeats, bSeats, cSeats, dSeats);

}

}

## Prompter.java

import java.util.InputMismatchException;

import java.util.Scanner;

import java.util.Set;

public class Prompter {

/\*\*

\* Prompts the user to insert a valid integer within the start and end range.

\*/

public static int promptRangeInteger(int start, int end, String message, String invalidMessage) {

int value;

while (true) {

System.out.print(message);

try {

final Scanner input = new Scanner(System.in);

final int opt = input.nextInt();

if (opt < start || opt > end) throw new InputMismatchException();

value = opt;

break;

} catch (InputMismatchException e) {

System.out.println(invalidMessage);

}

}

return value;

}

/\*\*

\* Prompts the user for a string.

\*

\* @see #promptValidString(Set, String, String)

\*/

public static String promptString(String message) {

System.out.print(message);

final Scanner input = new Scanner(System.in);

return input.next();

}

/\*\*

\* Prompts the user for a string input defined in the valid set.

\*/

public static String promptValidString(Set<String> valid, String message, String invalidMessage) {

String value;

while (true) {

System.out.print(message);

final Scanner input = new Scanner(System.in);

final String str = input.next();

if (valid.contains(str)) {

value = str;

break;

}

System.out.println(invalidMessage);

}

return value;

}

/\*\*

\* Prompts the user to input a "yes" "no" condition.

\*/

public static boolean promptConditional(String message) {

boolean value;

while (true) {

System.out.print(message + " (Y/N): ");

try {

final Scanner input = new Scanner(System.in);

final String stringVal = input.next().toLowerCase();

value = switch (stringVal) {

case "y", "yes" -> true;

case "n", "no" -> false;

default -> throw new InputMismatchException();

};

break;

} catch (InputMismatchException e) {

System.out.println("Invalid input! Allowed values are either Y(es) or N(o)");

}

}

return value;

}

}

## Ticket.java

import java.io.File;

import java.io.FileWriter;

import java.io.IOException;

public class Ticket {

public Position position;

public Person person;

public Ticket(Position position, Person person) {

this.position = position;

this.person = person;

}

public int getPrice() {

return position.getPrice();

}

public void save() throws IOException {

final File folder = new File("./tickets");

final boolean folderCreated = folder.exists() || folder.mkdir();

if (!folderCreated) throw new IOException("Failed to create the tickets folder");

final String path = String.format("%s%s.txt", folder.getPath() + File.separator, position.toDisplayString());

final FileWriter writer = new FileWriter(path);

writer.write(toDisplayString());

writer.close();

}

public String toDisplayString() {

return String.format(

"Ticket %s @ %sLKR of %s",

position.toDisplayString(),

position.getPrice(),

person.toDisplayString()

);

}

}

## Person.java

public class Person {

public String name;

public String surname;

public String email;

public Person(String name, String surname, String email) {

this.name = name;

this.surname = surname;

this.email = email;

}

public static Person fromUserInput() {

final String name = Prompter.promptString("Insert your first name: ");

final String surname = Prompter.promptString("Insert your surname: ");

final String email = Prompter.promptString("Insert your email: ");

return new Person(name, surname, email);

}

public String toDisplayString() {

return String.format("%s %s <%s>", name, surname, email);

}

}