בס"ד

הערכה חלופית תכנות מונחה עצמים
מתן מרקוביץ 322318080

פונקציית הmain בשאלה 1 נמצא במחלקה "FlightManager"
פונקציית הmain בשאלה 2 נמצאת במחלקה "CourseRegistrationSystem"

שאלה 1 חלק b:

1. **תבניות עיצוב:**

Strategy Pattern:

1. בכדי לבנות מנגנון לחיפוש טיסות הבחירה בstrategy עוזרת לקיבוץ כמה אלגוריתמים לסינון טיסות וקיבוצן תחת פעולה אחת. לכן כדי ליצור את מערכת הסינון שהתבקשתי, בניתי מחלקות למיון של טיסות לפי זמני המראה, מספר קונקשנים, מחיר הטיסה, זמן הטיסה וכלל הטיסות שלא בוטלו. על ידי הstrategy קיבצתי את כל המיונים לפעולה אחת עם switch case שעל פי קלט המשתמש תבחר את הסינון המתאים עבורו.
2. היתרון בשימוש בתבנית העיצוב במקרה הזה הוא שניתן לשפר את מנגנון החיפוש בצורה פשוטה ומהירה- על ידי הוספת מחלקה נוספת שמממשת את FlightSearchStrategy ולכן אם נרצה להרחיב את הקוד ולהוסיף מנגנוני מיון נוספים ניתן לעשות זאת מבלי לפגוע בשאר המחלקות בקוד ולהשאיר את המימוש כמו שהוא.
לולא השימוש בstrategy במקרה זה, הפתרון היה לבצע פעולות בנפרד על כל סינון וסינון, כלומר תהיה פונקציה למשל שמחזירה רק מיון לפי מחיר, וכדי לבצע מיון טיסות לפי זמני המראה נצטרך לבנות ולהשתמש בפונקציה נפרדת.

לדוגמה:

```
public class SearchByPrice implements FlightSearchStrategy {
  @Override
  public List<Flight> search(List<Flight> flights) {
    return flights.stream()
        .sorted(Comparator.comparingDouble(Flight::getCost))
        .collect(Collectors.toList());
  }
}
```

בדוגמת הקוד ניתן לראות את אלגוריתם חיפוש הטיסות במיון לפי מחיר, המחלקה מממשת את ה strategy שבו בעזרת switch case נפנה את המשתמש למימוש הנ"ל אם בחר בסינון לפי מחיר.

Composite Pattern:

1. על ידי מודל היררכי של composite ניתן להגדיר מתודות במחלקה שיחולו גם על כל המחלקות היורשות. בקוד שלי, השתמשתי ב composite לבניית חברות תעופה על ידי יצירת מחלקה אבסטרקטית "חברות תעופה" שמממשת מתודות שכל חברת תעופה מממשת לא משנה מה סוגה, למשל יצירת טיסה חדשה, ומחלקות הבן שלה "חברת תעופה עיקרית" ו- "תת חברת תעופה" מרחיבות את מימושי המחלקה האבסטרקטית לפי יעודה.
2. יתרונות התבנית הם שבעזרתה ניתן להוסיף לקוד בצורה נוחה סוג נוסף של חברת תעופה, במידה ונרצה להרחיב את הקוד למשל גם לחברות תעופה שמובילות משא (ולכן לא יהיו זקוקות למידע על נוסעים) או למשל לחברות מטוסים אזרחיים המיועדות לטיסות פנים לצורכי פנאי וכו'. בנוסף, התבנית תורמת לגמישות הקוד היות והוספת חברות תעופה מסוגים נוספים לא תגרום לעבודה כפולה משום שפונקציות של יצירת טיסה והצגת פרטי החברה כבר ממומשות.
למשל בדוגמת הקוד (מתוך main):

```
Flight f1 = ElAl.createFlight("LY001", "TLV", "AMS", 0, "12:00", "17:00", 500, "On time");
Flight f2 = Up.createFlight("LY002", "TLV", "JFK", 1, "14:00", "20:00",  1300, "On time");
```

ניתן לראות שלחברת התעופה העיקרית (ElAl) ולתת חברת התעופה (Up) אותו מימוש של יצירת טיסה חדשה היותר ופונקציה זו הוגדרה במחלקת האב "Airline" ולכן במקרה של הרחבת הקוד ויצירת סוג נוסף של חברת תעופה לא נצטרך לעשות עבודה כפולה כי פונקציה זו תהיה ממומשת עבורה.

Observer Pattern:

1. הבחירה בתבנית observer במימוש שלי מאפשרת שליחת עדכונים על ביטולי טיסות ועיכובים של טיסות. על ידי השימוש ב observer ניתן לשלוח לכל הנוסעים הרשומים למערכת ההתראות במקרה של שינוי בטיסה בזמן אמת במקום שכל נוסע יצטרך להתעדכן בעצמו על שינויי הטיסות ללא קשר לפעולות המתבצעות במחלקת הטיסות.
2. הobserver תורם לאפשרות הרחבת הקוד, כי בהוספת אובייקט או מחלקה נוספת לקוד, שגם תרצה להיות מעודכנת על מצב הטיסות (למשל בהוספת "מגדל פיקוח") כל מה שצריך לעשות על מנת שגם הוא יהיה מעודכן היא לממש את observer וכך הsubject יעדכן אותו גם. בנוסף, הobserver תורם לגמישות הקוד מכיוון שניתן לשנות במקום אחד את תוכן ההודעה ואת העדכונים הנשלחים במקום בכל מחלקה מקבלת בנפרד ועל ידי כך להימנע מעבודה כפולה.

   למשל בדוגמת הקוד הבאה ניתן לראות מימוש מתודת ביטול טיסה השולחת עדכון על ביטול הטיסה לכל הobservers באשר הם ללא תלות בtype או במחלקה.

```
    public void cancelFlight() {
        status = "cancelled";
        this.arraival_time = "xx:xx";
        this.departure_time = "xx:xx";
        System.out.println("\nFlight " + flightNumber + " is cancelled.");
        notifyObservers();
{
```

b.2 עקרונות תכנות מונחה עצמים:

1. הפרדת ממש ושימוש- השימוש בתבניות העיצוב מדגים את עקרון הפרדת הממשק מהמימוש על ידי כך שניתן לכתוב את הקוד בעזרת השימוש בתבניות שיוצקות לנו שלד שהוא גמיש ולא ספציפי ומאפשר מימושים שונים לקוד. למשל בstrategy אופן הפעלת מנגנון מיון הטיסות כלל אינו תלוי באופן המימוש האלגוריתם של כל מיון ומיון, בobserver למשל הוא מגדיר אופן יחסים בין שני אובייקטים ללא קשר למימוש הקוד ומה היחסים עצמם בין האובייקטים.

2. אינקפסולציה- תבניות העיצוב משפרות את האינקפסולציה במערכת על ידי כך שהן מרחיקות את המתודות עצמן מעיני המשתמש. למשל ב strategy אופן מימוש התבנית מותנה בכך שאלגוריתם הקצה האחראי למיון מוחבא בתוך מחלקות שמתחברות למימוש אחיד שהמשתמש כלל לא חשוף למה שקורה מאחוריו, ולמשל גם הcomposite שהמשתמש יכול לבצע פעולות על מחלקת האב שהתבנית מתרגמת לכל מחלקת בן את המימוש המתאים עבורה ובכך למשתמש אין כל צורך לדעת או להיות חשוף למימוש התוכנית.

3. פולימורפיזם- על ידי שימוש בפולימורפיזם ניתן להפוך את הקוד לגמיש יותר וכללי יותר היות והמשתמש לא מסתמך על אופן ביצוע הקוד אלא על הפעולות החשופות בפניו וכך למשל גם לאחר שהקוד הועבר למשתמש ניתן לשנות ולשפר את אופן השימוש אך להשאיר את אופן הפעולה זהה. למשל בstrategy של מנגנון חיפוש הטיסות יהיה ניתן לשפר את אחד האלגוריתמים אם ימצא אלגוריתם מיון מהיר יותר והמשתמש אפילו לא יהיה מודע לכך מכיוון שזה לא שינה את אופן פעולת הקוד או אופן ההפעלה שלו.

שאלה 2:

שימוש בdesign patterns:

- **Observer-** על ידי תבנית עיצוב זאת מימשתי עדכונים לסטודנטים בנוגע לקורס שרצו להרשם אליו אבל לא היו מקומות פנויים בקורס. השימוש דווקא בתבנית זאת תורם לאפשרות הרחבת מערכת ההתראות במידה ויתווספו גורמים נוספים למחלקת הקורסים ללא צורך בבניית מערכת התראות חדשה.

- **Decorator-** על ידי תבנית עיצוב זאת מימשתי אפשרות של הגדלת מספר הסטודנטים המקסימלי שיכולים להרשם לקורס מסויים. היתרון בתבנית זאת בקוד הוא האפשרות להרחבת הקוד למשל בהוספת גורמים נוספים לקורסים שלא נמצאים בבנאי שלהם.

- **Factory-** על ידי תבנית עיצוב זאת מימשתי פעולה אחת ליצירה של סוגים שונים של קורסים. היתרון בתבנית זאת הוא שהיא מאפשרת גמישות של הקוד במידה וארצה לשנות את אופן בניית הקורסים כמו למשל להוסיף שדה, אך המימוש של המשתנה בקצה לא ישתנה. בנוסף מאפשר גמישות להוספה של סוגי קורסים נוספים.

- **Singleton-** על ידי תבנית עיצוב זאת הגבלתי את userManager ל instance בודד ועל ידי כך יכולתי לבנות אותו מחדש בכל מחלקה בנפרד ולקבל את אותו הinstance שבעזרתו יכולתי לגשת לפונקציות של userManager עם הערכים ששמורים אצלו.

- **Composite-** על ידי תבנית עיצוב זאת, מימשתי מבנה היררכי של קורסים, וכך במקום לכל קורס מכל סוג לכתוב את אותו המימוש, ריכזתי הכל במחלקה אחת ממנה ירשו כל סוגי הקורסים. בנוסף מימוש זה תורם להרחבת הקורס במידה וארצה להוסיף סוג קורס נוסף.

שאלה 3:

1) סוג השגיאה- שגיאה לוגית.
הפונקציה אמורה לקחת את הרשימה בעלת האותיות הקטנות ובלולאת for להעתיק את השמות הכתובים באותיות קטנות ברשימה lowercaseLettersList לרשימה uppercaseLettersList ולהמיר אותה לאות גדולה. הפלט הרצוי הוא רשימה lowercaseLettersList שתישאר ללא שינוי, כלומר רשימה שמות אנשים באותיות קטנות, ורשימה uppercaseLettersList המכילה את אותם שמות האנשים אך באותיות גדולות.

בפועל, בתוך לולאת הfor הרשימה uppercaseLettersList מעתיקה בshallowCopy את השמות הנמצאים ברשימה lowercaseLettersList, כלומר מייצרת מצביע לאותו מקום בזיכרון ולכן כל שינוי שיתבצע ברשימה אחת על השמות שהועתקו יתבצע גם ברשימה השניה ולכן הפלט יהיה שתי רשימות עם שמות האנשים באותיות גדולות.

Output:

Upper case List:
ALICE
BOB
Lower case List:
ALICE
BOB

תיקון השגיאה- הוספת new בהעתקה, כלומר יצירת אובייקט חדש עם אותם ערכים (deepCopy) ומכיוון שמדובר בשני אובייקטים שונים, יש להם שני מצביעים שונים ולכן שינוי ערכי הראשון לא ישפיע על השני:

```
public static void main(String[] args) {
List<Person> uppercaseLettersList = new ArrayList<>();
List<Person> lowercaseLettersList = new ArrayList<>();
lowercaseLettersList.add(new Person("alice"));
lowercaseLettersList.add(new Person("bob"));

for (Person lowercasePerson : lowercaseLettersList) {        // המרה לפור-איץ' מטעמי נוחות
uppercaseLetterList.add(new Person(lowercasePerson.getName().toUpperCase()));      //deep copy
}

// Printing the first list
System.out.println("Upper case List:");
for (Person person : uppercaseLettersList) {
System.out.println(person.getName());
}
// Printing the second list
System.out.println("Lower case List:");
for (Person person : lowercaseLettersList) {
System.out.println(person.getName());
}
}
```

2) <u>סוג השגיאה</u>- שגיאת קומפילציה.
בתחילת הפונקציה, מצהירים על מערך דו מימדי ללא הצהרה על כל אחד מתת המערכים הפנימיים, ולכן בשורה (`positions[i][j] = new Position(i, j);`) נקבל שגיאת קומפילציה היות ומנסים לגשת לאיבר לא מאותחל.

<u>תיקון השגיאה</u>- מעבר בלולאה כפולה והצהרה על כל התת מערכים הפנימיים של Position[n][ ]

```
public static void main(String[] args) {
     int n = 10;
     Position[][] positions = new Position[n][];


     for ( int i =0; i < n; i++){
          for ( int j=0; i < n; j++){
               positions[ i ][ j ] = new Position(i,j);
          }
     }

     for (int i = 0; i < n; i++) {
          for (int j = 0; j < n; j++) {
               positions[i][j] = new Position(i, j);
          }
     }
     for (Position[] position_row: positions) {
          for (Position position: position_row) {
               System.out.print(position);
          }
          System.out.println();
     }
}
```

3) בכדי שמחלקת הבן תתפקד כמו מחלקת האב, כלומר במקרה שלנו שהפונקציה makeSound() תפלוט "animal sound" גם כאשר האובייקט עליו נפעיל את הפונקציה יהיה מסוג "Dog", כלומר נרצה לממש "shadowing"- שגם כאשר למחלקת הבן יש שדה באותו שם למחלקת האב הוא לא ידרוס אותה. את זאת נעשה באמצעות המילה השמורה "static" בהגדרת הפונקציה "makeSound()":

```
class Animal {
	final static String sound = "Animal sound";
	static void makeSound() {
		System.out.println(sound);
	}
}

class Dog extends Animal {
	final static String sound = "Bark";
	static void makeSound() {
		System.out.println(sound);
	}
}
.
.
.
```

מכיוון שפונקציות סטטיות משוייכות למחלקה ולא לinstance המימוש בפונקציית המain יהיה של מחלקת Animal ולא של הinstance Dog.