מטלה במונחה עצמים:

תז:213525512

חלק א: **תבניות עיצוב 1.1.**

הסבירו את הבחירה בתבניות עיצוב

הדגשה-דוגמאות הקוד ניתנות להעתקה והרצה זהו לא צילום מסך!

Strategy Pattern-

בהינתן שיש נוסע שמעוניין לבחור טיסה. יש מגוון קריטריונים שעשויים להשפיע על בחירת הטיסה שלו כמו מחיר הטיסה, זמן הטיסה ומשך זמן הטיסה. למשתמש יהיה קל יותר אם ניתן לו רשימת מיון של כל הטיסות בכל פעם על פי המיון שהוא רוצה. ולכן אנחנו רוצים לפתח תוכנית שיכיל אלגוריתמים שונים לחיפוש טיסות ויאפשר למשתמשים לבחור אסטרטגיית חיפוש בזמן ריצה. תבנית עיצוב זאת strategy pattern עונה על הדרישות האלו. הממשק FlightSearchStrategy מגדיר את החוזה המשותף לכל אסטרטגיות חיפוש הטיסות. הוא כולל פונקציה אחת , flightSearch(), המקבלת רשימת טיסות ומחזירה את הרשימה הממוינת לפי האסטרטגיה הספציפית. לכל שיטת מיון אנחנו כתבתי מחלקה משל עצמה שיהיה מיון אסטרטגי משלו. עשיתי לכל שיטות המיון interface משותף כך שאוכל במחלקה Searchלכתוב פונקציה אחת ולכלול את כל השיטות מבלי להשתמש ב instance of. ועדיין יהיה אפשר להחליף בקלות בין האסטרטגיות. ויהיה ניתן לבחור באסטרטגיית חיפוש ספציפית באמצעות שיטת setSearchStrategy()של מחלקת Search. שיטת orderedFlight()מבצעת את החיפוש באמצעות האסטרטגיה הנוכחית.

```
import java.util.List;

public interface FlightSearchStrategy {
   public List<Flight> flightSearch(List<Flight> f);
}




public class SFprice implements FlightSearchStrategy {
    @Override
    public List<Flight> flightSearch(List<Flight> flights){
       flights.sort(Comparator.comparing(Flight::getPrice));
        return flights;
    }
}


public class SFtimeOfFlight implements FlightSearchStrategy{

    public List<Flight> flightSearch(List<Flight> flights){

        flights.sort(Comparator.comparing(Flight::getDurationTime));
        return flights;
```

```
    }
}
import java.util.List;

public class Search {
    private FlightSearchStrategy searchSDtrategy;

    public void setSearchStrategy(FlightSearchStrategy searchStrategy)
{
        this.searchSDtrategy = searchStrategy;
    }
    public List<Flight> orderedFlight(List<Flight> flights) {
        return searchSDtrategy.flightSearch(flights);
    }

}
```

Pattern Composite:

אני אתחיל קודם מלהסביר כיצד הבנתי השאלה- להבנתי ישנו שדה תעופה ויש רשימה שנמצאת במחלקה שדה תעופה שמכילה רק "חברות גדולות", החברות האלה מכילות טיסות או חברות טיסה אחרות . נשים לב שכך, המחלקה airPort מכילה את כל הטיסות והחברות תעופה שקיימות בה באופן כללי. בשאלה חברות הטיסה יכולות להכיל טיסות וגם חברות תעופה אחרות ואז נוצר רשימה של חברות גדולות שכל אחת היא שורש של עץ שהעלים שלו הם טיסות והצמתים הם תתי חברות תעופה. לכן נשתמש בcomposite pattern שמאפשרת הרכבה של חברות תעופה וטיסות כך שנוצר מבנה דמויי עץ שמשקף את הקשרים ההירכיים בין חברות התעופה, תתי החברות והטיסות שלהן. וכך גם קל יותר לממש פתרון לבעיות שונות. למשל אני רציתי ששדה התעופה וחברות התעופה יוכלו להדפיס את הטיסות וחברות התעופה שלהן או לדעת אם טיסה או חברת תעופה מסויימת קיימת או שלא. קודם, יצרתי ה inteface שנקרא Airline שהוא יהיה כללי ויכיל פעולות שמשותפות גם לטיסות וגם לחברות תעופה. למשל, String Name(),containsItem(), Print() המחלקה airline composite שהיא מדמה "צומת" בעץ ממשת את הממשק Airline ומכילה רשימה של טיסות וגם חברות תעופה באמצעות (Airline) היא יכולה גם להסיר ולהוסיף לרשימה טיסות או חברות תעופה. במחלקה Flight שהוא ה"עלה" בעץ מממשת גם את הממשק Airline ובנוסף היא מכילה פעולות שמיוחדות לאובייקט טיסה.

```
public class AirPort {
public void printAirport() {
    System.out.println(airportName);
    for (Airline airline : allFlightsAndAirlines) {
        airline.print();
    }
}
}
```

כשאני רוצה להדפיס את כל שמות חברות הטיסה והטיסות שבבעלותם אני עוברת על הרשימה של החברות הגדולות שכל אחת כפי שאמרנו מחזיקה עץ משל עצמה

```
public interface Airline {

    boolean containsItem(String name);
    void print();
}

public class Flight implements Airline {
@Override
public void print() {
    System.out.println("Flight name  " + this.name + " price  " +
this.price + " time " + this.departureTime + " Duration Flight  " +
this.durationTime);
}
}

public class AirlineComposite implements Airline {
    private String name;
    private final HashSet<Airline> children;

    public AirlineComposite(String name) {
        this.name = name;
        this.children = new HashSet<>();
    }

    public void add(Airline item) {
        children.add(item);
    }

    public void removeItem(Airline item) {

        if (children.contains(item)) {
            this.children.remove(item);
        }
        // Remove from immediate children

        // Recursively remove from child composites if needed
        for (Airline child : children) {
            if (child instanceof AirlineComposite) {
                ((AirlineComposite) child).removeItem(item);
            }

        }


    }
    @Override
    public void print() {
    System.out.println("Airline: " + name);
    for (Airline item : children) {
        item.print();
    }
}
```

```
public class Main {
    public static void main(String[] args) {
benGurionAirport.printAirport();
}
```

באמצעות התבנית יכולתי לכתוב את הפונקציה ()print למשל שעוברת על ההיררכיה, מדפיסה את הפרטים של כל אובייקט (Airline) בין אם צומת עלים או צומת מורכב. היא גם הקלה עליי לכתוב את פונקציית ()ContainItem שמצריכה פעולה רקורסיבית בזכות ה"עץ".

Observer pattern-

בקוד אנחנו צריכים יש מערכת יחסים של אחד ורבים - שכל העובדים או הנוסעים שמתעניינים בשינוי של טיסה מסויימת, ברגע שיש בטיסה שינוי כלשהו הם יקבלו התראה על כך באופן אוטומטי.Observerהיא השיטה הטובה ביותר לממש את זה. כי היא מאפשרת ביטוי לכל מאורע ועדכונו למי שמעוניין באמצעות list שמחזיק בכל האנשים שמתעניינים בטיסה מסויימת. תחילה אסביר איך הקוד אצלי מתממש באמצעות התבנית של observer ואיך באמת המשתמשים, באמצעות השימוש, ה observers מקבלים עדכונים. מכיוון שבקוד יש שלוש מצבים בטיסה בחרתי גם ליצור את המחלקה

```
public enum ChangeType {
     FLIGHTDELAY, CANCELFLIGHT, SPECIALPRICE
}
```

במחלקה Flight יצרתי בתכונות רשימה של כל האנשים שמתעניינים בה . בעזרת הרשימה הקודמת יצרתי את הרשימה כך שכל מצב של טיסה מסויימת יחזיק אנשים שמתעניינים באותה טיסה ומתעניינים במצב מסויים למשל רק אם היא בוטלה. בצורה כזאת גם אם יהיה שינוי כלשהו בטיסה אך נוסע מסויים התעניין המצב אחר של אותה טיסה הוא לא יקבל עדכון על השינוי. ההרשמה וההסרה נמצאות גם כן בFlight

```
Map<ChangeType, List<FlightChanges>> observers = new HashMap<>();
public void subscribe(ChangeType[] changeTypes, FlightChanges
flightChanges) {
    for (ChangeType changeType : changeTypes) {
        List<FlightChanges> users = observers.get(changeType);
        users.add(flightChanges);
    }
}


public void notifySubscribers(ChangeType updateType, Flight flight) {
    List<FlightChanges> users = observers.get(updateType);
    for (FlightChanges listener : users) {
        listener.update(updateType, flight);
    }
}
```

יצרתי ממשק FlightChanges שמכיל את הפונק update. יצרתי גם ממשק People שהמחלקות של העובדים ושל הנוסעים יממשו אותה, בנוסף הם גם יממשו את FlightChanges. באמצעות הפונק ()registertoobserverspesicflight הם יכולים להרשם לקטגוריה שהם

מתעניינים בה בטיסה מסויימת. באמצעות מימוש הפונק update יכולים "להגיב" לשינוי שקורה לטיסה.

```
public interface FlightChanges {
    void update(ChangeType changeType, Flight flight);
}

public class Passenger implements People, FlightChanges {

@Override
public void registerToObserveSpecificFlight(ChangeType[] changes,
Flight flight) {
    flight.subscribe(changes, this);
}
@Override
public void update(ChangeType changeType, Flight flight) {

    switch (changeType) {
        case FLIGHTDELAY:
            System.out.println(this.name + " flight" + flight.getName()
+ " is delayed"+flight.getDurationTime());
            break;
        case CANCELFLIGHT:
            System.out.println(this.name + " flight" + flight.getName()
+ " cancelled");
            break;
        case SPECIALPRICE:
            System.out.println(this.name + " flight" + flight.getName()
+ "There is a special price for your flight"+flight.getPrice());
            break;
    }
```

במחלקה Flight ישנם כמה מצבים של טיסות שאנשים יכולים להתעניין בשינוי בהם כמו מחיר, עיכוב בטיסה או ביטולה. לדוגמא:

```
public class Flight implements Airline {
public void setCancelled() {
    this.isHapening = false;
    notifySubscribers(ChangeType.CANCELFLIGHT, this);

}
```

באמצעות הקוד, ברגע שהמנהל נאלץ לבטל טיסה למשל. אז כל האנשים או העובדים שרצו לקבל מידע על ביטול הטיסה הזאת יקבלו התראה אוטומטית שהטיסה בוטלה וכל זה באמצעות השימוש בObserver pattern.

2. יתרונות ומטרות בתבניות העיצוב

Strategy

ללא שימוש ב strategy pattern לכל מנגנון היינו צריכים להשתמש פעמים רבות בפעולות כמו switch case ו instance of כדי לבדוק באיזה אסטרטגית מיון אנחנו נמצאים. זהו פתרון, אך הוא לא טוב כי בכל פעם שנרצה להוסיף מיון מסוג חדש נצטרך לחזור לפעולת המיון ולהוסיף לה

אסטרגיה חדשה ולשנות שוב בכל המקומות הנחוצים בקוד את השינוי הדרוש וזה מסובך ולא יעיל.strategy pattern תורמת לגמישות שכן ניתן להוסיף אסטרטגיות חיפוש חדשות ללא צורך בשינוי מבנה הקוד הקיים אלא רק הוספה של מחלקה שמייצגת את האסטרטגיה החדשה.
גמישות זו נובעת מההפרדה הברורה בין לוגיקת החיפוש (המיושמת באסטרטגיות קונקרטיות) לבין ביצוע החיפוש במחלקה Search. בנוסף, בעזרת שימוש בstrategy pattern אנחנו יכולים באותה פונקציה להכיל אסטרטגיות מיון שונות ולעבור באופן דינמי בין אסטרטגיות חיפוש שונות בזמן ריצה על פי רצון המשתמש.
ניתנת להרחבה- ניתן להוסיף עוד שיטות מיון של קריטריונים אחרים. כאשר ניצור לשיטת המיון החדשה class משל עצמו אך לא בהכרח נצטרך לשנות פעולות שמשתמשות במיונים או מיונים אחרים כי הם אינם תלויים אחד בשני. למשל, לא נצטרך לשנות את FlightSearchStrategy.
תחזוקת הקוד-כל אסטרטגיה מיוצגת באמצעות מחלקה משלה שמכילה אלגוריתם חיפוש מיוחד לה, מה שמקל על זיהוי והבנת הקוד האחראי לכל התנהגות חיפוש. וכך קל לשנות ולתקן אותו במקרה הצורך. ואפשר להתמקד בשינוי אלגוריתם חיפוש אחד מבלי לחשוש מפגיעה בשאר אלגורתמי החיפוש בקוד.

Composite pattern

גמישות- אפשר היה למצוא פתרון נאיבי של רשימות שמחזיקות רשימות של טיסות ורשימות של חברות תעופה. אבל אז היה קוד לפונקציה היה מאוד ארוך ואם היינו רוצים להוסיף עדכון מסויים היינו צריכים להיכנס לכל הקוד הארוך ולשנות אותה שם- מורכב מאוד! בנוסף, אם אחנו רוצים להוסיף עוד שיטה כלשהיא למשל- דרך לנוע באוויר חוץ ממטוס למשל כדור פורח היינו צריכים לעבור על הפונקציה שכתבנו ולהוסיף בה את האפשרות לכדור פורח-מסובך מאוד!
Composite pattern פותרת את הבעיות האלה כי היא מאפשרת הרכבה דינמית של חברות תעופה ותתי-חברות בתוך ההיררכיה.בצורה כזאת, ניתן בקלות להוסיף, להסיר או לסדר מחדש טיסות או חברות תעופה מבלי לשנות את המבנה הכללי .
הרחבה- התבנית מאפשרת הוספה של חברות וטיסות חדשות בכל רמה של ההירכיה או להוסיף שירותי תעופה אחרים- כמו הרעיון שהצעתי לעיל (תעבורה בכדור פורח) בלי צורך לשנות את כל הקוד אלה לעשות מחלקה נוספת שגם היא תממש Airline. במקרה כזה אולי כדאי לעשות ל Flight וגם למחלקה של הכדור פורח מחלקה אבסטרקטית משותפת.
תחזוקה- בעזרת תבנית העיצוב הזאת המשתמשים יכולים לעבוד עם חברות תעופה וטיסות מבלי לדעת כיצד עובד כל המבנה ההיררכי ופרטי הקוד. בצורה כזאת נוכל לעשות שינויים במקרה הצורך בקוד מבלי להשפיע על הממש של המשתמש. בנוסף, הטיסות וחברות התעופה יורשות ממשק אחיד משותפAirline. אחידות זו מאפשרת פעולות עקביות כמו שליפת נתונים, הדפסות או דיווחים לאורך כל ההיררכיה בלי אבחנה אם זה חברת תעופה או טיסה. היא מפשטת את הקוד ומפחיתה את הצורך בטיפול מיוחד בסוגים שונים של צמתים. בצורה כזאת גם יש פחות "כפיליות" בקוד והרבה יותר קל לשנות את הקוד במקרה הצורך.

Observer pattern

גמישות- התבנית צופים יכולים להרשם או לבטל את הרישום בקלות באופן דינאמי בהתאם לצרכים הספציפיים שלהם . תבנית Observer מחלקת את הקוד למודולים נפרדים: מצבי טיסה וצופים בצורה כזאת שינוי באחד לא ישפיע על הצד השני.

הרחבה- ניתן להוסיף סוגים חדשים של אנשים כמו העובדים או הנוסעים מבלי לשנות את הקוד הקיים. ניתן להוסיף סוגיי שינויים חדשים כמו מקומות שהתפנו מבלי להשפיע על הקוד הקיים. מחלקת Flightיכולה להודיע לצופים על סוגי שינויים אלה באמצעות השיטה notifySubscribers
תחזוקה- באמצעות התבניות הזאת נוצר מצב שכל הלוגיקה של ההודעות מרוכזת במחלקת Flight , כך שבמקרה הצורך ניתן לבצע את השינויים במקום אחד, וזה לא מופזר בכל הקוד.
בנוסף, התבנית עוזרת להפריד בין הטיסות לצופים, מה שמאפשר הבנה יותר טובה וזה מקל על ביצוע שינויים מבלי להשפיע על חלקים לא קשורים במערכת

2. **עקרונות תכנות מונחה עצמים:**

הפרדת ממשק ומימוש:
כאשר השתמשנו בתבניות העיצוב בקוד הרבה פעמים השתמשנו בממשק. הממשק הוא כללי- הוא מתייחס לחתימות והצהרות, ואילו המימוש מתייחס לקוד שייכתב בפועל שזה נעשה באמצעות כל מיני מחלקות שממשות את הממשק ובדרכים שונות שזה בדיוק עקרון הפרדת הממשק מהמימוש. אפשר לראות את זה Pattern Strategy שכל אחד מאלגורתמי המיון יורשים את הממשק FlightSearchStrategy ולכן כולם ממשים את הפונקציה flightSearch() , אבל כל אחד אותה בדרך יחודית לו וכך אפשר למיין את הטיסות בדרכים שונות באמצעות אותה פעולה.
ב :Pattern Composite. העקרון מתבטא ברעיון שגם הטיסות וגם חברות התעופה יממשו אותו ממשק Airline וכך יהיה ניתן ליצור רשימה שמכילה שני אובייקטים שונים-כי הם יורשים מאותו ממשק. ואז לעשות כל מני פעולות על הרשימה הזאת כמו להוסיף טיסות או חברות תעופה וכו'
ב -Observer pattern היינו צריכים לממש את הממשק FlightChanges ובצורה הזאת חייבנו שכל ה people - הנוסעים והעובדים חייבים לממש את הפעולה update שבאמצעותה הם מקבלים התראות שונות אם לעדכון על איחור בטיסה ואם על מחיר מיוחד.

אינקפסולציה:
מטרת האינקפסולציה היא להסתיר את הפרטים הפנימיים של אובייקט ולהציג ממשק מוגדר המאפשר אינטראקציה איתו בלי לגלות עליו יותר מידי פרטים שיכולים לחשוף את הקוד ואף לבלבל את המשתמש. ב Pattern Strategy המשתמש אינו יודע כיצד בדיוק המיון עובד . הוא מכניס את שיטת המיון שהוא רוצה ובאמצעות הפעולה orderedFlight הוא יקבל רשימה ממויינת בנושא שיבחר שנמצאת ב Search.
Pattern Composite אותו רעיון. המשתמש לא מודע לכל המבנה של ההיררכי של חברות התעופה והטיסות –וכיצד הוא מחזיק חברות וגם טיסות וכו' הוא מקבל משק AirPortin.
ובאמצעותו הוא יודע שהוא יכול להדפיס את כל הטיסות למשל-ואין לא מושג איך זה מתממש.
-Observer pattern. המשתמש שוב לא מודע לכיצד התוכנית עובדת ואיך בדיוק הביטול של הטיסה גורם לכך שכל מי שנרשם לטיסה בנושא השזה יקבל התראה. הוא נחשף רק לממשק People ובאמצעותו הוא יודע שאפשר להשתמש בפעולה registerToObserveSpecificFligh כדי להרשם להתעניינות בטיסה מסויימת.

פולימורפיזם:
בקוד פולימורפי, ניתן להשתמש באותה פונקציה או משתנה עם טיפוסים שונים, והקוד יתנהג בצורה שונה בהתאם לטיפוס הספציפי בשימוש. אפשר לראות את זה מתממש ב Pattern Strategy. כאשר כל המיונים ממשים אותו interface והפעולות שמשתמשות באופרטור של interface הזה יכולות לשמש לכל סוגי המיון כמו flightSearch.
Pattern Composite יוצרים רשימה שמכילה גם טיסה וגם חברת תעופה. בצורה כזאת, ניתן להפעיל כל מיני פעולות על הרשימה הזאת למשל הדפסה. והאלגוריתם עצמו תוך כדי ריצה ידע שאם האובייקט הזה הוא מסוג טיסה הוא צריך להפעיל את פונק ההדפסה הספציפי של טיסה. ואם מסוג אחר, הוא צריך להפעיל את פונק ההדפסה הספציפי של חברת תעופה. ב Observer pattern יש מגוון של מקרים שיכולים לקרות: איחור טיסה, ביטולה או שינוי מחיר. באמצעות השימוש ב enum ChangeType. כל המקרים נכנסים לאותו קטגוריה כביכול ואז ניתן לעשות עליהם פעולה משותפת כמו ההתארה למתעניינים. אבל בפונקציית העידכון עצמה ההיחסות לכל מקרה היא נפרדת ולכל מקרה מודפס משהו אחר. בצורה כזאת המערכת הצליחה לטפל במקרים השונים באופן גמיש כי במקרה של הוספת מקרה קל לשנות אותה ולהוסיף לפונק העדכון אפשרות להדפסה נוספת שרצויה ולכן היא גמישה. גם לגבי הobservers עצמם. הנוסעים והעובדים נכנסים לאותו list של צופים ע"י כך ששניהם ממשים את הממשק People.

שאלה 2 – עיצוב חופשי :

השתמשתי ב5 תבניות עיצוב:

Factory- השתמשתי בתבניות העיצוב הזאת בשביל יצירת סוגי הקורסים השונים. יצרתי את שלושת המחלקות ElectiveCourse, requiredCou,SeminarCourse. בכל אחת מהן כתבתי יצירת אובייקט יחודית של עצמה. לאחר מכן, יצרתי את המחלקה: FctoryCourse שהיא מחלקת "אב" משותפת שהמטרה שלה היא לייצר קורסים מסוגים שונים תוך כדי זמן הריצה של הקוד באמצעות אותה הפקודה. התבנית מאפשרת לי להוסיף בקלות סוגים חדשים של קורסים מבלי לשנות את הקוד מכיוון שאנחנו משתמשים באותו שם של פונק בשביל ליצור כל סוג קורס. היא גם מאפשרת לנו להפריד בין יצירת האובייקט והשימוש בו, וכך להפחית את התלות בין הקוד לבין המחלקה של האובייקט . דרך יצירת האובייקט לא ידועה למי שמשתמש באובייקט, זה הופך את הקוד ליותר מודולרי וקל לתחזוקה.

Flywight- בשביל למנוע יצור של אותו קורס מספר פעמים שלא לצורך. בצורה כזאת אם גם המתרגל וגם המרצה מנסים ליצור את אותו קורס, מי שישלח את הפקודה ראשון באמת ייצור את הקורס ויקבל אותו אבל מי שישלח שני יקבל את אותו קורס שהראשון יצר. עשיתי זאת, באמצעות יצירת רשימה של כל הקורסים בפונק addCours בדקתי אם כבר מספר הקורס נמצא ברשימה. אם לא, הוספתי אותו לרשימה. אם כן, אני את הקורס שמצאתי בלי ליצור אובייקט חדש. באמצעות השימוש בתבנית הזאת מנעתי אפשרות לייצר את אותם קורסים פעם אחר פעם וכך מנעתי בעייה אפשרית של צריכת זכרון גבוה ובילבול בקרב הסטודנטים שנרשמים, כך שכול מי שירצה להרשם לקורס מסויים ירשם לאותו קורס שמשותף לכולם וישמר הסדר- שלכל קורס מותר להרשם עד 50 סטודנטים למשל .

Observer- בחרתי להשתמש בתבנית הזאת כדי לוודא שאם לסטודנט לא היה מקום בקורס שניסה להירשם אליו, אם ישתנה המצב ומשהו שנירשם יחליט כי הוא לא רוצה כבר את הקורס. מי שרצה להירשם לקורס איך לא היה לו מקום יקבל באופן אוטומטי התראה התפנה מקום והוא יכול להירשם. בצורה כזאת גרמתי לכך שלא יהיה תלות בין הobserver לבין הsubject. כך כשבהמשך כשהיי צריכה להוסיף פונק ועשיתי שינויים במחלקת קורס לא חששתי מפגיעה כלשהי בצופים של הקורס.וזה הוסיף לי סדר לקוד.

Singelton - בחרתי להשתמש בתבנית העיצוב הזאת כי רציתי תהיה מערכת אחת שתנהל את כל ההרשמה, ההתנתקויות והחיבורים למערכת הרשמה- כך תהיה גישה מבוקרת לאובייקט אחד בלבד, שיהיה ניתן לגשת אליו מכל מקום במערכת וזה גם יחסוך זכרון ויפשט את ניהול המשאבים וימנע התנגשויות ובעיות שיכולות להיווצר מיצירת מספר אובייקטים של מערכת ההרשמה.

Façade- בחרתי בתבנית עיצוב זו כדי לאגד בצורה פשוטה את כל הפעולות שהמשתמש יכול לרצות לעשות במערכת. בעצם, הסתירתי את המימוש של כל המחלקות השונות ותתי המחלקות והקשרים בניהן מעיני המשתמש במטרה להקל עליו להשתמש במערכת בלי צורך להכיר את המימוש המורכב של המערכת. המחלקה שיצרתי כאן נקרא RegistrationSystem. ובאמצעות הפונקציות שתאפשר להרשם, להתנתק, להתחבר וכו'

אינקפסולציה- למשל כשכתבתי פעולות שונות במחלקה RegisterationSystem . באמצעותם הסתרתי את המימוש של הקוד שלי מהמשתמש. למשל, בניית פונקציה שתוסיף קורס והיא תהיה אחראית ליצור אותו, להוסיף אותו לרשימת הקורסים הקיימים וכו ועדיין, המשתמש רק יצטרך להכניס את פרטי הקורס שהוא רוצה. בצורה כזאת יכולתי להציג ממשק קל יותר ומובן לעבודה.

פולימורפיזם- למשל כל ה"אנשים" בתוכנית שלי: המרצה, המתרגל והסטודנט ירשו מהממשק User. בצורה כזאת יכולתי ליצור רשימת נרשמים, פעילים במערכת שהן משותפות לכל האנשים שהטיפוס שלהן היה User. לכתוב פונקציות שונות בעיקר במחלקה RegisterationSystem שטיפוס שהן קיבלו היה מסוג User לכן הם התאימו לכל סוגי המשתמשים ולא הייתי צריכה לכתוב פונקציות שעושות את אותו הרעיון לסוגים שונים בתוכנית. לדוגמא: User.grtID(). מה שהיה גורם לשכפול של קוד מיותר.

ממשקים וירושה- בקוד שלי למשל המרצה והמתרגל ירשו ממשק שנקרא UniWorker שהוא הרחיב את הממשק User שמומש גם ע"י המחלקה של הסטודנט. כך שכולם בסוף ירשו מUser. יצירת הממשק מבטיח שמחלקות מיישמות ייצמדו לכל שיטות וההתנהגויות המוגדרות בו. וזה עזר לשמור על עקביות בקוד, לפחות טעיות ולפחות שכפול בקוד (ב UniWorker לא היה צריך לכתוב שוב פעולות בUser) . בנוסף, זה אפשר לי להשתמש בעקרון הפולימורפיזם עליו הרחבתי לעיל.

חלק ב' – שאלה תכנותית המבוססת על התרגולים

1. יש כאן שגיאה לוגית אבל נשים לב שעדיין הקוד ירוץ ויתקמפל.

סיבת השגיאה-בלולאת ה for אנחנו עוברים על כל המצבעים של האובייקטים ברשימת lowercaseLettersList ומוסיפים את כל המצבעים של האובייקטים מרשימה זו לuppercaseLettersList כלומר עכשיו כל האובייקטים שהרשימה lowercaseLettersList מצביעה אליהם אז גם uppercaseLettersList מצביע לאותם אובייקטים עצמם. ולכן בשורה לאחת מכן, כשמשנים כל שם באובייקט בuppercaseLettersList להיות עם אותיות גדולות. בגלל ששני הרשימות מצביעות לאותם אובייקטים אז גם השמות ב lowercaseLettersList יהיה עם אותיות גדולות

פלט:

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

הצעה לתיקון:

הרשימה uppercaseLettersList תכיל אובייקטים חדשים שהם יהיו העתק של האובייקטים lowercaseLettersList. נשאיר את הקוד כפי שהוא ונשנה רק את לולאת ה for כך:

```
for (int i = 0; i < lowercaseLettersList.size(); i++) {
    uppercaseLettersList.add(new
Person(lowercaseLettersList.get(i).getName().toUpperCase()));
}
```

כלומר, אנחנו משנים רק את השם באופן מדוייק ויוצרים באמצאות המילה new אובייקט חדש שאין לא שום תלות והפנייה לאובייקטים שנמצאים lowercaseLettersList. כך, שינוי באחת הרשימות לא תשפיע על הרשימה השניה.

2. יש כאן שגיאה בזמן ריצה. התוכנית לא תרוץ היא תחזיר לנו"NullPointerException".

סיבת השגיאה- בשורת האיתחול של המערך אנחנו יוצרים מערך דו מיימדי בעל n שורות. כל שורה במערך היא מערך פנימי בפני עצמה וצריך להגיד אותו. מכיוון ש Position[][] positions [][]= new Position[n] בסוגריים המודגשים לא ניתן ערך כלשהו המערך הפנימי הוא null כי הוא לא מאותחל ולא נוכל לשים אובייקט Position במקום כלשהו במערך הדו מיימדי. ולכן תחזור שגיאה

הצעה לתיקון:

```
Position[][] positions = new Position[n][n];
```

בצורה כזאת המערך הדו מיימדי וכל תתי המערכים בו מאותחלים כראוי. הנחתי כי צריך להוסיף דווקא n כיוון ששתי הלולאות רצות עד n.

3. אנחנו רוצים שהאובייקט animal ידפיס Animal sound כמו בחלקת האב , וגם נוכל להדפיס "Bark" אם נרצה כמו במחלקת הבן . ולכן במקום השורה void makeSound() בשתי המחלקות נכתוב static void makeSound(). מכיוון שעכשיו שתי הפונקציות הן סטטיות הוא ילך לפי סוג האובייקט. ובגלל שהתוכנית יוצרת את האובייקט animal שיהיה מסוג Animal ורק משייכת לו מופע חדש של Dog אז הוא ידפיס Animal sound. אם נרצה להדפיס "Bark" נוכל לשנות את סוג האובייקט לDog. אפשרות אחרת, היא למחוק את המשתנה Sound במחלקה Dog. ואז כשהתוכנית תרוץ היא תחפש איפה מוגדר Sound והיא תמצא אותו רק במחלקה של Animal ששם הוא מוגדר להיות Animal sound ולכן זה מה שיודפס. בשביל להדפיס "Bark" נוכל לכתוב Bark במקום sound. אבל האופציה הזאת פחות עדיפה.