שאלה 1

.b

1. לתבניות העיצוב יש תפקיד מרכזי בפישוט התכנון והתחזוקה של מערכות שונות. במקרה שלנו, כל תבנית מספקת פתרון קל לבעיות שונות.

שם התבנית – **Strategy**

מטרת השימוש במערכת:

תבנית זו משמשת לפיתוח מנגנון חיפוש טיסות שיאפשר למשתמש לבחור את אסטרטגיית החיפוש הרצויה בזמן ריצה. לדוגמה, החיפוש יכול להתבצע לפי מחיר, זמן המראה, או משך הטיסה. על ידי השימוש בתבנית זו, המערכת מתאימה את האלגוריתמים השונים לחיפוש תוך ניתוח מהירות הרצפים וביצוע תהליכי חיפוש מתקדמים בזמן אמת.

יתרונות ומטרות:

השימוש בתבנית זו מאפשר למערכת לשנות את אופן חיפוש הטיסות בזמן ריצה. לדוגמה, חיפוש לפי מחיר/זמן המראה וכו היתרון בתבנית זו הו שניתן להוסיף אלגוריתמים נוספים לחיפוש טיסות בעתיד בקלות, כך שהמערכת תהיה מוכנה להתמודד עם דרישות חדשות של המשתמשים .בנוסף, התבנית מאפשרת עדכונים ושינויים בקוד באופן נפרד לכל אסטרטגיה. כך שנוכל לתחזק כל אלגוריתם (חיפוש במקרה שלנו) באופן עצמאי ללא תלות בשאר הקוד ובקלות.

דוגמאות מהקוד:

.1

```
public interface SearchStrategy {
   List<Flight> search(List<Flight> flights);
}
```

.2

```
public class MaxPriceSearch implements SearchStrategy {
    private final float maxPrice;

    public MaxPriceSearch(float maxPrice) {
        this.maxPrice = maxPrice;
    }

    @Override
    public List<Flight> search(List<Flight> flights) {
        List<Flight> filteredFlights = new ArrayList<>();
        for (Flight f : flights) {
            if (f.getPrice() <= maxPrice) {
                filteredFlights.add(f);
            }
        }
        return filteredFlights;
    }
}
```

שם התבנית – **Composite**

מטרת השימוש במערכת:

במערכת ניהול הטיסות, נדרשת היכולת לבנות מודל היררכי של חברות תעופה, כולל תתי-חברות וטיסות תחת כל חברה. תבנית המרכיבים מתאימה פה כי היא מאפשרת בניית מבנה רב-שכבתי, כך שניתן לנהל ולדווח על נתונים בצורה מרוכזת. כאשר חברות תעופה גדולות יכולות להכיל תתי-חברות רבות, התבנית הזו מאפשרת ניהול יעיל ומסודר של המידע

יתרונות ומטרות:

השימוש במבנה זה מאפשר למערכת להתמודד עם יעדים שונים ולהתאים את מבנה הנתונים בהתאם. לדוגמה, ניתן להוסיף חברות תעופה חדשות בקלות על ידי הוספת תתי-חברות וטיסות. בעזרת המבנה היררכי, ניתן להוסיף ענפים נוספים בקלות, כמו תתי-חברות חדשות או סוגי טיסות חדשים, לצורך תמיכה בעסקים משניים והרחבה של תוכניות נסיעה .השימוש במבנה זה מאפשר לנו לשנות ולתחזק חלקים ספציפיים במערכת מבלי להשפיע על שאר המבנה. לדוגמה, אם יש שינוי בטיסה של חברת תעופה מסוימת, ניתן לעדכן את הנתונים שלה בלבד מבלי להשפיע על נתוני חברות תעופה אחרות.

דוגמאות מהקוד:

.1

```
package FlightSystem.Data;

public interface Component {
   void display();
}
```

.2

```
public class Flight implements Component {
    private final int flightNumber;
    private long departureTime;
    private long arrivalTime;
    private int price;
    private final String from;
    private final String to;
```

שם התבנית - **Observer**

מטרת השימוש במערכת:
מערכת התראות לעדכונים על טיסות מסייעת לנוסעים ועובדי חברת התעופה לקבל עדכונים בזמן אמת על שינויים בטיסות, כמו שינויים בזמני המראה, ביטולים, או מבצעים חדשים. תבנית זו מאפשרת למשתמשים להירשם כצופים ולקבל עדכונים בזמן אמת, מה שמסייע להם לתכנן את הנסיעות שלהם באופן יעיל ולהיות מעודכנים על שינויים בטיסות.

יתרונות ומטרות:
מנגנון התראות בזמן אמת מאפשר למערכת לספק למשתמשים עדכונים מיידיים על מצבם של הטיסות. ובעצם נותן לנוסעים לתכנן נסיעותיהם בצורה מתקדמת יותר.
ניתן להוסיף אפשרויות התראה נוספות בעת צורך, כגון התראות על זמן קבלת המזוודות ועוד.
השימוש במנגנון ההתראות מאפשר עדכונים קלים ותחזוקה של הניהול של התראות ספציפיות בלבד, מבלי להשפיע על שאר הקוד במערכת.

דוגמאות מהקוד:

.1

```
package FlightSystem.Notifications;

public interface Subject<T> {
   void addObserver(Observer<T> observer);
   void removeObserver(Observer<T> observer);
   void notifyDataHasChanged(T data);
}
```

```java
package FlightSystem.Notifications;

import FlightSystem.Data.Flight;

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

public class FlightSubject implements Subject<Flight> {
    private final List<Observer<Flight>> observers = new ArrayList<>();
    private Flight flight;

    public FlightSubject(Flight flight) {
        this.flight = flight;
    }

    @Override
    public void addObserver(Observer<Flight> observer) {
        observers.add(observer);
    }

    @Override
    public void removeObserver(Observer<Flight> observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyDataHasChanged(Flight data) {
        for (Observer<Flight> observer : observers) {
            observer.update(data);
        }
    }

    public void updateFlightData(long departureTime, long arrivalTime, int price) {
        flight.setDepartureTime(departureTime);
        flight.setArrivalTime(arrivalTime);
        flight.setPrice(price);
        notifyDataHasChanged(flight);
    }
}
```

.3

```
package FlightSystem.Notifications;

public interface Observer<T> {
   void update(T data);
}
```

.4

```
public class ArrivalTimeObserver  implements Observer<Flight> {
   @Override
   public void update(Flight flight) {
      System.out.println("received update: Arrival Time has changed: " + flight.getArrivalTime());
   }
}
```

.2 נתייחס בשאלה לכל אחת מתבניות המימוש:

Observer

1. הפרדת ממשק ומימוש:

המטרה העיקרית של תבנית עיצוב זו היא להודיע/לעדכן אובייקטים אחרים על שינוי במצב כלשהו של אובייקט אחר מבלי שהם יתעסקו בו.
תבנית העיצוב מאפשרת הפרדה בין ממשק המגדיר את התפקיד של Observerובין המימוש שלו. בדוגמה שלנו, ישנם צופים שונים המעוניינים לקבל עדכונים על מאפיינים שונים של טיסה וזמן ההגעה, זמן היציאה, ומחיר. כל אחד מממש את הפעולה update בהתאם להתאמתו לסוג העדכון שהוא מעוניין לקבל.

2. אינקפסולציה:

באמצעות תבנית העיצוב ,Observer אנו מפרידים בין יצירת קשרים בין המצפים לבין הנושאים, ובין יצירת הקשרים אליהם. למשל, במערכת הטיסות שלנו, הפרטים של הצופה עצמו לא נחשפים, הם מקבלים עדכונים רק על שינויים בטיסה ו/או שינויים אחרים, כך שהם מקבלים רק את המידע הדרוש להם.

3. פולימורפיזם:

בתבנית העיצוב ישנם צופים **שונים** המעוניינים לקבל עידכונים על שינויים בטיסות. וכל אחד מהאובייקטים מממש את update.

Strategy

1. הפרדת ממשק ומימוש:

במקום לקשר את הפונקציות או האלגוריתמים ישירות לקוד הראשי, אנו משתמשים בממשק **SearchStrategy** כדי להגדיר את הפעולות המבוצעות. הממשק מגדיר את הפעולות או האלגוריתמים, והמחלקות הממשות את הממשק מממשות את הלוגיקה הספציפית לכל אלגוריתם. כאשר משתנה האלגוריתם או נדרש להוסיף חיפוש חדש, אין צורך לשנות את הקוד הראשי אלא רק ליצור מחלקה חדשה שמממשת את הממשק.

2. אינקפסולציה:

תבנית העיצוב מסייעת באינקפסולציה על ידי יצירת מבנה המאפשר לנהל היררכיה של חברות תעופה וטיסות. במקום לגלות את הפרטים הפנימיים של כל רמה בהיררכיה, המשתמש מתמקד רק בממשק הציבורי של המבנה. זה מאפשר לנו להוסיף חברות תעופה נוספות או טיסות בצורה גמישה, מבלי לשנות את הקוד הקיים ומבלי לחשוף את הפרטים הפנימיים למשתמש.

3. פולימורפיזם:

במערכת ניהול טיסות, השימוש בפולימורפיזם יאפשר לנו לטפל בסוגים שונים של אובייקטים באופן גמיש. לדוגמה, בשימוש בתבנית למנגנון החיפוש של הטיסות (ב **SearchStrategy**), כל אלגוריתם חיפוש מיושם כמחלקה נפרדת שמממשת את אותו ממשק. כשהמשתמש מבקש לחפש טיסה, המערכת יכולה להשתמש בפולימורפיזם כדי להחליט איזה אלגוריתם חיפוש להשתמש בו בזמן ריצה, בהתאם לבחירת המשתמש או לתנאים אחרים. כאשר מדובר בטיפול בסוגים שונים של אובייקטים, כמו מנגנוני חיפוש שונים או סוגים שונים של מנויים, ניתן להשתמש בפולימורפיזם על ידי הגדרת ממשק כללי עבור כל סוג, והממשק יכלול את הפעולות המשותפות לכל הסוגים. למשל, ממשק "חיפוש טיסה" עשוי להיות מורכב מפעולה כמו "בצע חיפוש", וכל מחלקת מנגן חיפוש תממש את הפעולה בצורה שונה עבור סוג החיפוש הספציפי שהיא מייצגת.

Composite

1. הפרדת ממשק ומימוש:

בתבנית העיצוב Composite, השימוש בממשק Component מאפשר להפריד את הממשק הציבורי מהמימוש הפנימי של הרכיבים במערכת. דוגמה לכך היא הממשק Component שמגדיר את הפעולות הבסיסיות של כל רכיב במערכת התעופה, ולמעשה מגדיר את האפשרויות שהמשתמש יכול להשתמש בהן. כך, הממשק מאפשר למשתמש להתמקד בשימוש בפונקציות ובתכונות של הרכיבים מבלי להיות צריך לדעת את הפרטים הפנימיים שלהם או איך הם מומשים.

2. אינקפסולציה:

בשימוש בתבנית המרכיבים, הממשק הציבורי של החברה תעופה מגדיר פעולות כמו "מצא טיסות" או "בצע הזמנה". כשמשתמש פונה לממשק כזה, הוא לא צריך לדעת איך החברה מאורגנת בתוך, כולל מי עושה מה ואיך הטיסות מתוזמנות. פשוט יש לו פונקציות שהוא יכול להשתמש בהן, והמערכת תעשה את השאר כדי לספק לו את התוצאות שהוא רוצה. כך הממשק מסתיר מאחוריו את הפרטים הפנימיים של המערכת ומאפשר למשתמש להתמקד רק בפעולות שהוא רוצה לבצע.

3. פולימורפיזם:

תבנית העיצוב Composite מאפשרת פולימורפיזם על ידי אפשרות לנהל רכיבים שונים במערכת כאילו הם חלק מאותה משפחה. במערכת התעופה, כל רכיב מממש את הממשק **Component**, ולכן ניתן לטפל בו באותה דרך כמו בכל רכיב אחר. זה מאפשר להוסיף רכיבים נוספים למערכת בקלות, כי הם יכולים להתנהל כאילו הם חלק מאותה משפחה כבר. כל רכיב יכול להפעיל פונקציות ותכונות באופן שונה, והמערכת יכולה לטפל בכל סוג של רכיב.