בס"ד

הפרויקט שלי מדגים שימוש בתבניות עיצוב קלאסיות ועקרונות תכנות מונחה עצמים:

1. תבניות עיצוב - Factory Method, Composite, Observer, Strategy, Decorator
2. עקרונות - OOP הכמסה, ירושה, פולימורפיזם, מופשטות
3. עקרונות - SOLID עקרון האחריות היחידה, עקרון פתוח-סגור, עקרון החלפת ליסקוב, עקרון הפרדת הממשק, עקרון היפוך התלות

השילוב של כל אלה יוצר מערכת מודולרית, גמישה, וקלה לתחזוקה שמדגימה היטב את התועלות של תכנות מונחה עצמים ותבניות עיצוב.

<u>Design Patterns:</u>

במערכת יישמתי חמש תבניות עיצוב, כל אחת מהן פותרת בעיה ספציפית בתכנון התוכנה:

1. Factory Method - מחלקת UserFactory המייצרת אובייקטים מסוג User על פי הפרמטר המתקבל.

- הפרדת אחריות - הקוד שיוצר את האובייקט מופרד מהקוד שמשתמש בו
- הסתרת מורכבות - מי שמשתמש בפקטורי לא צריך להכיר את כל מחלקות המשתמש
- גמישות בהרחבה - אפשר להוסיף סוגי משתמש חדשים בקלות
- ריכוז לוגיקת היצירה - כל לוגיקת היצירה של משתמשים מרוכזת במקום אחד

2. Composite - מחלקות Property SimpleProperty, ו- CompositeProperty המממשות היררכיית עצים של נכסים.

- מבנה היררכי טבעי - משקף באופן טבעי את המבנה ההיררכי של נכסים בנדל"ן (בניין ודירות בתוכו)
- פעולות רקורסיביות - מאפשר לבצע פעולות באופן רקורסיבי על העץ (לדוגמה, חישוב שטח כולל)
- הסתרת מורכבות - מסתיר מהלקוח את ההבדלים בין נכס פשוט לנכס מורכב .
- טיפול אחיד - מאפשר לטפל בנכסים פשוטים ומורכבים באותה צורה

3. Observer

- צימוד רופף - המוכר והמתווך לא צריכים להכיר זה את זה באופן ישיר
- תקשורת אוטומטית - מאפשר עדכון אוטומטי של המתווך כאשר נכס נמחק
- הרחבה קלה - אפשר להוסיף מתווכים נוספים שיצפו בשינויים בנכסים
- מידור מידע - כל אחד מקבל רק את המידע הרלוונטי לו

4. Strategy - ממשק PropertySearchStrategy עם מימושים שונים כגון ,ByStatus, PriceBySqm, ו.MeanPriceSqm-

- החלפת אלגוריתמים בזמן ריצה - מאפשר להחליף אסטרטגיית חיפוש בקלות
- הפרדת אחריות - כל אסטרטגיה אחראית רק על אלגוריתם החיפוש שלה
- הרחבה קלה - אפשר להוסיף אסטרטגיות חיפוש חדשות ללא שינוי בקוד הקיים
- ממשק אחיד - כל אסטרטגיות החיפוש מספקות ממשק אחיד לשימוש

5. Decorator - ממשק Deal, מחלקה בסיסית ,BasicDeal מחלקה מופשטת ,DealDecorator ומחלקות קונקרטיות כמו Cleaning, Haulage, Design ו- .GuaranteeService

- הוספת פונקציונליות בזמן ריצה - מאפשר להוסיף שירותים לעסקה בזמן ריצה
- שילוב גמיש - מאפשר לשלב שירותים שונים בצורות שונות
- עקרון פתוח-סגור - אפשר להוסיף שירותים חדשים ללא שינוי בקוד הקיים
- הימנעות מהתפוצצות מחלקות - מונע את הצורך ביצירת מחלקה נפרדת לכל שילוב אפשרי של שירותים

<u>עקרונות תכנות מונחה עצמים:</u>

בנוסף לתבניות העיצוב, הפרויקט מדגים שימוש במספר עקרונות תכנות מונחה עצמים:

1. הכמסה(Encapsulation)

יישום בקוד:

- שימוש במתודות גישה (getters/setters) במקום גישה ישירה לשדות
- הגדרת שדות כ private-כמו למשל password במחלקת User

איך תורם למערכת:

- הסתרת מידע - מגן על משתנים פנימיים מגישה לא מורשית
- בקרת גישה - מאפשר לוודא תקינות נתונים בכניסה וביציאה
- גמישות בשינויים - מאפשר לשנות את המימוש הפנימי ללא השפעה על הקוד שמשתמש במחלקה

2. ירושה (Inheritance)

יישום בקוד:

- היררכיית המחלקות User -> Buyer, Seller, Broker
- היררכיית המחלקות Property -> SimpleProperty, CompositeProperty

איך תורם למערכת:

- שיתוף קוד - מונע שכפול קוד בין סוגי המשתמשים השונים
- מבנה היררכי - מארגן את הקוד בצורה לוגית וברורה
- הרחבה קלה - מאפשר להוסיף סוגי משתמש חדשים בקלות

3. פולימורפיזם (Polymorphism)

יישום בקוד:

- שימוש בממשקים כמו Viewable, Editable, Deletable
- מתודות וירטואליות ב Property-כמו displayPropertyInfo(), getSqm(), getTotalPrice()

איך תורם למערכת:

- גמישות -מאפשר טיפול אחיד באובייקטים מסוגים שונים
- הפרדת ממשק ממימוש -מאפשר להחליף את המימוש ללא שינוי בקוד הקורא
- קוד מודולרי -מאפשר הוספת פונקציונליות חדשה ללא שינוי בקוד הקיים

4. מופשטות (Abstraction)

יישום בקוד:

- מחלקות מופשטות כמו Property ו- User
- ממשקים כמו Deal, PropertySearchStrategy, Observable

איך תורם למערכת:

- הסתרת מורכבות - מסתיר את המימוש המורכב מאחורי ממשק פשוט
- מיקוד בפונקציונליות - מאפשר למתכנת להתמקד במה האובייקט עושה, לא איך
- קוד יציב - מקטין את ההשפעה של שינויים פנימיים על שאר המערכת

<u>עקרונות SOLID :</u>

הפרויקט שלי מדגים גם יישום של עקרונותSOLID :

1. אחריות יחידה (Single Responsibility Principle)

יישום בקוד:

- כל מחלקה אחראית על תחום אחד, למשל PropertyFileReader אחראי רק על קריאה וכתיבה לקובץ

- הפרדה בין מחלקות המודל (Property) , מחלקות הלוגיקה (PropertySearcher) , ומחלקות התצוגה (תבנית MVC)

תרומה למערכת:

- קוד נקי ומובן - כל מחלקה עושה דבר אחד אבל עושה אותו טוב
- תחזוקה קלה - קל לשנות מחלקה כשיש לה אחריות ברורה
- בדיקות פשוטות - קל לבדוק מחלקה עם אחריות ממוקדת

2. פתוח-סגור (Open-Closed Principle)

יישום בקוד:

- אפשר להוסיף אסטרטגיות חיפוש חדשות מבלי לשנות את PropertySearcher
- אפשר להוסיף דקורטורים חדשים מבלי לשנות את BasicDeal

תרומה למערכת:

- הרחבה ללא שינוי - המערכת פתוחה להרחבה אך סגורה לשינוי
- יציבות - שינויים לא משפיעים על חלקים אחרים של המערכת
- מניעת באגים - הוספת פונקציונליות לא דורשת שינוי בקוד מוכח

3. החלפת ליסקוב (Liskov Substitution Principle)

יישום בקוד:

- אפשר להשתמש ב SimpleProperty-או ב- CompositeProperty במקום Property
- אפשר להשתמש בBuyer, Seller או- Broker במקום User

תרומה למערכת:

- שקיפות סוגים - אפשר להשתמש בתת-מחלקות במקום מחלקות הבסיס ללא שינוי בהתנהגות הצפויה
- פולימורפיזם נכון - מאפשר פולימורפיזם אמיתי עם שמירה על החוזה של מחלקת הבסיס
- הרחבה בטוחה - מבטיח שהרחבות לא שוברות את המערכת

4. הפרדת הממשק (Interface Segregation Principle)

יישום בקוד:

- ממשקים קטנים וממוקדים כמו Viewable, Editable, Deletableבמקום ממשק ענק אחד
- מחלקות מממשות רק את הממשקים הרלוונטיים להן

תרומה למערכת:

- ממשקים "רזים " - מחלקות לא נאלצות לממש מתודות שאינן רלוונטיות להן
- הפרדת תחומי אחריות - ממשק לכל תפקיד במקום ממשק כללי גדול
- גמישות - מאפשר שילוב מגוון של התנהגויות בלי ירושה מרובה

5. היפוך התלות (Dependency Inversion Principle)

יישום בקוד:

- תלות בהפשטות (PropertySearchStrategy) ולא במימושים קונקרטיים
- העברת תלויות כפרמטרים (כמו העברת Deal לדקורטור)

תרומה למערכת:

- צימוד רופף - מחלקות לא תלויות ישירות במחלקות אחרות
- בדיקות קלות - אפשר להחליף תלויות אמיתיות במוקים לצורך בדיקות
- גמישות - אפשר להחליף מימושים קונקרטיים ללא שינוי בקוד הצרכן
