תבניות עיצוב

Singleton
ה-Singleton מיושם ב-SystemManager, שמשמש כמרכז שליטה יחיד למערכת כולה. באמצעות השיטה getInstance(), מבטיחים שרק מופע אחד של המחלקה קיים, מה שמונע כפילויות ושומר על עקביות בנתונים כמו רשימת הנכסים. לדוגמה, כל פעולה שדורשת גישה למצב המערכת עוברת דרך אותו מופע, מה שמפחית סיכונים לבעיות סינכרון ומקל על ניהול משאבים משותפים. דפוס זה תורם לארגון יעיל של הקוד ומבטיח שכל חלקי המערכת פועלים בהרמוניה תחת מערכת אחת.

Factory (דפוס מפעל)
באמצעות ה-Factory, מחלקת UserFactory מטפלת ביצירת מופעי משתמשים בצורה מרוכזת ומסודרת. השיטה createUser(UserType type, int userId) מקבלת את סוג המשתמש ומחזירה את המופע המתאים, כמו Buyer או Broker, תוך הסתרת לוגיקת היצירה מהקוד הלקוח. גישה זו מפחיתה תלות ישירה במחלקות ספציפיות, מאפשרת הוספת סוגי משתמשים חדשים בלי לשנות קוד קיים ומשפרת את היכולת לתחזק את המערכת. לדוגמה, אם נרצה להוסיף סוג משתמש חדש, נשנה רק את ה-Factory, מה ששומר על ניקיון וגמישות.

Strategy
ה-Strategy מאפשר חיפוש נכסים גמיש ומותאם אישית דרך PropertySearchContext, שמשתמש באסטרטגיות כמו SearchByAveragePriceStrategy או SearchByStatusStrategy. השיטה setSearchStrategy() מאפשרת לשנות את האסטרטגיה בזמן ריצה, כך שניתן לעבור בין חיפוש לפי מחיר ממוצע לחיפוש לפי סטטוס בלי לשנות את הקוד הבסיסי. גישה זו הופכת את המערכת למודולרית, מקלה על הוספת קריטריונים חדשים בעתיד ומבטיחה שהלוגיקה המרכזית נשארת נקייה וברורה, תוך שמירה על יכולת התאמה גבוהה לחיפושים לפי פרמטרים משתנים.

Decorator
ה-Decorator מוסיף שכבות של שירותים לעסקאות בצורה חלקה וגמישה, דרך מחלקות כמו CleaningDecorator או MovingDecorator. אלה מרחיבים עסקה בסיסית (BasicDeal) על ידי שיפור השיטה executeDeal() עם יכולות נוספות, כמו ניקיון או הובלה, מבלי לגעת בקוד המקורי. לדוגמה, ניתן לשלב שירותים חדשים בלי לשנות את המבנה הבסיסי, מה שמקדם את עיקרון הפתיחה/סגירה ומאפשר התאמה אישית של עסקאות. הדפוס משפר את היכולת להרחיב את המערכת תוך שמירה על פשטות וניקיון.

Observer
ה-Observer מאפשר תקשורת אוטומטית בין חלקי המערכת, כפי שרואים ב-PropertyDeletionObserver. ממשק זה מאפשר לברוקרים לקבל התראות דרך onPropertyDeleted() כאשר מוכר מוחק נכס, מה שמבטיח שהם תמיד מעודכנים בשינויים,לפי הדרישה.

עקרונות SOLID

S - Single Responsibility Principle
כל מחלקה בפרויקט תוכננה כך שתתמקד בתפקיד אחד ברור ומוגדר, מה שמפחית מורכבות ומשפר את איכות הקוד. לדוגמה, Property אחראית אך ורק לניהול נתוני נכסים, בעוד SystemManager מתמקדת בתיאום פעולות ברמת המערכת. גישה זו מבטיחה ששינוי בתפקיד של מחלקה אחת לא ישפיע על אחרות, מה שמקל על בדיקות, תחזוקה וזיהוי בעיות. התוצאה היא מערכת נקייה יותר שקל יותר לפתח ולשדרג לאורך זמן, תוך שמירה על בהירות ופשטות.

O - Open/Closed Principle
המערכת בנויה כך שניתן להרחיב אותה מבלי לשנות קוד קיים, מה שמקדם גמישות ובטיחות. לדוגמה, ניתן להוסיף אסטרטגיות חיפוש חדשות או דקורטורים חדשים לעסקאות בלי לגעת בלוגיקה הבסיסית. עיקרון זה מאפשר למפתחים להוסיף פונקציונליות חדשה, כמו שירותים נוספים או סוגי משתמשים, מבלי להסתכן בשבירת הקוד הישן.

L - Liskov Substitution Principle
מחלקות כמו Buyer, Seller, ו-Broker ניתנות לשימוש במקום User מבלי לפגוע בהתנהגות התוכנית, מה שמבטיח עקביות ושימוש חוזר. לדוגמה, פונקציה שמצפה לקבל User יכולה לעבוד עם Broker בלי שום שינוי, שכן כל מחלקות עומדות בחוזה של המחלקה הבסיסית. עיקרון זה מפשט את השימוש בהיררכיית המחלקות, מקל על כתיבת קוד גנרי ומבטיח שהמערכת תמשיך לפעול כראוי גם כאשר משתמשים ברכיבים שונים.

I - Interface Segregation Principle
הפרויקט משתמש בממשקים קטנים וספציפיים כמו ViewPermission ו-EditPermission, כדי להבטיח שמחלקות לא ייאלצו ליישם שיטות מיותרות. לדוגמה, Buyer מממש רק ViewPermission, בעוד Broker מוסיף גם EditPermission, מה שמותאם בדיוק לצרכים של כל מחלקה. גישה זו מפחיתה תלות מיותרת, מקטינה את כמות הקוד הלא נחוץ ומשפרת את הבהירות, תוך שמירה על מבנה נקי שקל יותר לתחזק ולהבין.

D - Dependency Inversion Principle
מודולים ברמה גבוהה תלויים בהפשטות ולא במימושים קונקרטיים, מה שמגביר את הגמישות של המערכת. לדוגמה, SystemManager עובד עם ממשק Deal ולא עם דקורטורים ספציפיים, מה שמאפשר להחליף רכיבים בלי לשנות את הקוד שתלוי בהם. עיקרון זה מפחית תלות ישירה בין חלקי המערכת, מקל על בדיקות והחלפת מודולים ומשפר את היכולת להרחיב את הפרויקט, תוך שמירה על קוד נקי ובר תחזוקה.

עקרונות OOP

ירושה (Inheritance)
בפרויקט זה, הירושה היא כלי מרכזי לארגון הקוד ולשיפור היעילות שלו. מחלקת User, המוגדרת אבסטקטית, משמשת כבסיס משותף למחלקות Buyer, Seller, ו-Broker, ומאפשרת להן לרשת תכונות בסיסיות כמו userId (מזהה ייחודי) ו-userType (סוג המשתמש). מה שמבטיח שימוש חוזר בקוד תוך שמירה על מבנה ברור. הירושה מאפשרת גמישות רבה, שכן ניתן להוסיף סוגי משתמשים חדשים בעתיד מבלי לשנות את המחלקה הבסיסית, מה שמקל על תחזוקה והרחבה של המערכת תוך שמירה על עקביות בין כל סוגי המשתמשים.

פולימורפיזם (Polymorphism)
השימוש בפולימורפיזם בפרויקט מאפשר התנהגות מגוונת תחת ממשקים משותפים, מה שמקדם גמישות וקלות שימוש. ממשקים כמו ViewPermission, EditPermission, ו-DeletePermission מוגדרים כדי לתאר פעולות בסיסיות, ומחלקות כמו Seller ו-Broker מיישמות אותם באופן שונה בהתאם לתפקידיהן. לדוגמה, בזמן ש-Seller יכול למחוק נכס דרך deleteProperty, ה-Broker יכול לערוך פרטי נכס עם editProperty, והכל מתבצע תחת אותו ממשק. גישה זו מאפשרת לקוד לטפל בסוגי משתמשים שונים בצורה אחידה, מפשטת את הלוגיקה ומאפשרת החלפת רכיבים מבלי לפגוע בתפקוד המערכת, מה שתורם לניהול יעיל ולקוד נקי יותר.

כימוס (Encapsulation)
הכימוס בפרויקט נועדה להגן על נתונים ולשפר את אמינות הקוד, והיא מיושמת באמצעות תכונות פרטיות ושיטות גישה מוגדרות. במחלקת Property, לדוגמה, התכונות address, area, ו-pricePerSquareMeter מוגדרות כפרטיות, והגישה אליהן מותרת רק דרך שיטות כמו getAddress() או setArea(). בנוסף, השיטה getTotalPrice() מחשבת את המחיר הכולל של הנכס מבלי לחשוף את החישוב עצמו, מה שמסתיר את המימוש הפנימי ומגן על הנתונים מפני שינויים לא רצויים. גישה זו לא רק משפרת את הבטיחות, אלא גם מקלה על תחזוקה עתידית של הקוד, שכן שינויים פנימיים במחלקה לא ישפיעו על החלקים שמשתמשים בה.

הפשטה (Abstraction)
ההפשטה בפרויקט מתמקדת בהגדרת מה צריך לקרות מבלי להתעסק בפרטי האיך, מה שיוצר קוד נקי ומודולרי. מחלקת User האבסטרקטית מספקת תבנית כללית למשתמשים, בעוד ממשקים כמו Deal מגדירים פעולות כמו executeDeal מבלי לפרט את אופן הביצוע.השימוש בהפשטה מפחית את המורכבות, מאפשר שימוש חוזר בקוד ומקל על שילוב רכיבים חדשים, תוך שמירה על בהירות ופשטות במבנה המערכת.

גנריות (Genericity)
הפרויקט משתמש בגנריות כדי להגביר את הגמישות והשימוש החוזר בקוד. לדוגמה, `PropertySearchContext<T>` מאפשר לאסטרטגיות חיפוש להחזיר סוגים שונים של תוצאות, כמו `Double` עבור מחיר ממוצע או `List<Property>` עבור רשימת נכסים. זה מאפשר למערכת להתמודד עם סוגים שונים של חיפושים מבלי לשנות את הקוד הבסיסי, מה שתורם לקוד נקי וגמיש יותר.

מודולריות (Modularity)
הפרויקט בנוי בצורה מודולרית, עם הפרדה ברורה בין מחלקות ומודולים שונים. לדוגמה, `SystemManager` אחראי על ניהול המערכת כולה, בעוד `UserFactory` מטפל ביצירת משתמשים. גישה זו מאפשרת ניהול נפרד של חלקים שונים של המערכת, מקלה על תחזוקה והרחבה, ומפחיתה תלות בין מודולים, מה שתורם לקוד מאורגן וקל לתחזוקה.