תבניות לעיצוב תוכנה

1. רעיון ומטרה

- *כללי:*

design pattern מספק פתרון כללי המיועד לשימוש חוזר לבעיות נפוצות המתרחשות בעיצוב תוכנה. התבנית מציגה בדרך כלל יחסים ואינטראקציות בין מחלקות או אובייקטים. הרעיון הוא להאיץ את תהליך הפיתוח על ידי מתן פרדיגמות פיתוח / עיצוב בדוקות ומוכחות.

design patterns הינם אסטרטגיות נפרדות של שפת תכנות לפתרון בעיות נפוצות, זה אומר שתבנית העיצוב מייצגת רעיון, לא יישום מסוים.

על ידי שימוש ב-**design pattern** , אתה יכול להפוך את הקוד שלך לגמיש יותר, יעודי לשימוש חוזר וניתן לתחזוקה.

אין זה מן החובה ליישם תמיד **design pattern** בפרויקט שלך. **Design pattern** אינם מיועדים לפיתוח, **design pattern** נועד לפתרון בעיות נפוצות. וכאשר מתעורר צורך, אתה צריך ליישם תבנית מתאימה כדי למנוע בעיות כאלה בעתיד.

כדי לגלות באיזה תבנית להשתמש, כל מה שאתה צריך זה לנסות להבין את תבניות העיצוב ואת המטרות שלהם. רק על ידי כך, תוכל לבחור את המתאים.

- *מטרה:*

הבנת המטרה והשימוש של כל סוג מה-**design pattern** על מנת לבחור וליישם את התבנית הנכונה לפי הצורך.

- *דוגמא:*

במצבים רבים בעולם האמיתי, אנו רוצים ליצור רק מופע אחד של מחלקה. לדוגמה, יכול להיות רק נשיא פעיל אחד של מדינה בכל זמן נתון. תבנית זו נקראת תבנית Singleton.

דוגמאות תוכנה אחרות יכולות להיות כגון נרצה ליצר מחלקת DBconnection יחידה המשותפת למספר אובייקטים, שכן יצירת connection DB נפרד לכל אובייקט היא יקרה, ומכיוון שיש לנו הרבה מופעים של connection DB אנו מוכרחים להימנע מקריאה / כתיבה בו זמנית לDB מכמה מופעים וזה ע"י ה-Singleton.

באופן דומה נרצה מנהל מערכת בודד או מנהל שגיאות באפליקציה שמטפל בכל הבעיות וכל הבעיות מתנקזות אליו במקום ליצור מספר מנהלים.

נ.ב. תומצת מהמצגות של אלי חלסצ'י וממדריכים אינטרנטיים כ- javaTpoint, geeksForGeeks, netSolutions וכ"ו, נכתב כפי הבנת המפרסם, והבנת הקורא באחריותו בלבד.

2. סוגי ה-Design Pattern

ישנם בעיקר שלושה סוגים של דפוסי עיצוב:

Creational (יְצִירִי)	Structural (מִבנִי)	Behavioral (התנהגותי)
עוסק בתכנון יצירת האובייקטים	עוסק ביחסים בין מחלקות וישויות (הכלה/ הורשה..)	עוסק בהתקשרות בין עצמים בזמן ריצה
~ Factory Method ~ Abstract Factory ~ Builder ~ Singleton	~ Adapter ~ Decorator ~ Bridge ~ Facade	~ Command ~ Strategy ~ Observer

סוגי Design Patterns נפוצים

הערה

צימוד ו-**לכידות** הינם מושגים המתייחסים לרמת איכות הקוד כאשר קוד איכותי יותר הוא פחות תלותי פונקציונלית כלומר נרצה כמה שפחות קשר בין קטעים בקוד, צמידות מבטא תלותיות ולכידות מבטא את ההפך (האי-תלותיות), נרצה קוד פחות צמיד ויתר לכיד.

Creational

• Factory Method Pattern

Factory Method Pattern או Factory Method Pattern אומרת שפשוט מגדירים ממשק (interface) או מחלקה אבסטרקטית ליצירת אובייקט אבל נותנים לתת-מחלקות להחליט איזו מחלקה להפעיל. במילים אחרות, תת-מחלקות אחראיות ליצור את המופע של המחלקה.

Factory Method Pattern ידוע גם כ-Virtual Constructor.

<u>יתרונות של Factory Method Pattern</u>

~ Factory Method Pattern מאפשרת לתת-מחלקות לבחור את סוג האובייקטים ליצירה.

~ זה מקדם את הצימוד הרופף על ידי ביטול הצורך לאגד מחלקות ספציפיות ליישום לתוך הקוד. זה אומר שהקוד מקיים אינטראקציה אך ורק עם הממשק או המחלקה המופשטת שנוצרו, כך שהוא יעבוד עם כל מחלקות שמיישמות את הממשק הזה או שמרחיבות את המחלקה המופשטת.

<u>שימוש Factory Method Pattern</u>

~ כאשר מחלקה לא יודעת אילו מחלקות משנה יידרשו ליצור.

~ כאשר מחלקה רוצה שתתי המחלקות שלה יציינו את האובייקטים שייווצרו.

~ כאשר מחלקות האב בוחרות ביצירת אובייקטים לתת-מחלקות שלה.

• Abstract Factory Pattern

Abstract Factory Pattern אומר שמגדירים interface או מחלקה אבסטרקטית ליצירת אובייקטים קשורים, אך מבלי לציין את תת המחלקות הקונקרטיות שלהם. כלומר, Abstract Factory מאפשר למחלקה להחזיר מפעל של מחלקות. לכן זו הסיבה ש-Abstract Factory ברמה אחת מעל Factory Pattern.

ידוע גם כ- Kit.

יתרונות של Abstract Factory Pattern

~ Abstract Factory Pattern מבודד את קוד הלקוח מישומי המחלקות.
~ מקל על החלפת אובייקטים מאותה משפחה.
~ מקדם עקביות בין אובייקטים.

שימושים ל-Abstract Factory Pattern

~ כאשר המערכת צריכה להיות בלתי תלויה מיצירת האובייקט.

~ כאשר יש להשתמש באובייקטים הקשורים יחד, אז יש לאכוף את האילוץ הזה.

~ כאשר רוצים לספק ספריית אובייקטים שאינה מציגה את המימושים אלא רק חושפת ממשקים.

~ כאשר נרצה להגדיר מערכת עם אוביקט ממשפחה של אובייקטים.

- **Builder**

Builder Pattern אומר כך-"בנה אובייקט המורכב מאובייקטים פשוטים תוך שימוש בגישת שלב אחר שלב",
הוא משמש בעיקר כאשר לא ניתן ליצור אובייקט בשלב אחד כמו בדה-סריאליזציה (יצירת אובייקט מיבני מפרטים, כגון קובץ JSON) של אובייקט מורכב.

היתרונות העיקריים של Builder

~ מספק הפרדה ברורה בין הבנייה והייצוג של אובייקט.
~ מספק שליטה טובה יותר על תהליך הבנייה.
~ תומך בשינוי הייצוג הפנימי של אובייקטים.

- **Singleton**

Singleton Pattern אומר "תגדיר מחלקה שיש לה רק מופע אחד בזמן נתון ומספקת נקודת גישה גלובלית אליה".במילים אחרות, מחלקה חייבת להבטיח שיש ליצור רק מופע בודד ואובייקט בודד יכול לשמש את כל המחלקות האחרות.

ישנן שתי צורות של Singleton Pattern

1. מופע מוקדם: יצירת מופע בזמן טעינה.
2. מופע עצלן: יצירת מופע בעת הצורך.

יתרון של דפוס עיצוב Singleton

~ חוסך זיכרון מכיוון שלא נוצר אובייקט בכל בקשה, רק מופע בודד חוזר שוב ושוב.

שימוש בדפוס עיצוב Singleton

~ Singleton Pattern משמש בעיקר ביישומים מרובי הליכים ומסד נתונים.
הוא משמש ברישום, שמירה במטמון, thread pools, הגדרות תצורה וכו'.

Structural

- **Adapter**

Adapter אומר - "ממיר את הממשק של המחלקה לממשק אחר שלקוח רוצה".
במילים אחרות, לספק את הממשק לפי דרישת הלקוח תוך שימוש בשירותי מחלקה עם ממשק שונה.

Adapter ידוע גם בשם Wrapper.

היתרון של תבנית מתאם

~ מאפשר לשני או יותר אובייקטים שלא היו תואמים בעבר ליצור אינטראקציה.
~ מאפשר שימוש חוזר בפונקציונליות קיימת.

שימוש בדפוס המתאם

~ כאשר אובייקט צריך להשתמש במחלקה קיימת עם ממשק לא תואם.

~ כאשר אתה רוצה ליצור מחלקה לשימוש חוזר המשתפת פעולה עם מחלקות שאין להן ממשקים תואמים.

- **Decorator**

Decorator Pattern אומר "צרף עוד אחריות לאובייקט באופן דינמי".

במילים אחרות, The Decorator Pattern משתמש בקומפוזיציה (הכלה) במקום בירושה כדי להרחיב את הפונקציונליות של אובייקט בזמן ריצה.

Decorator ידוע גם בשם Wrapper.

<u>יתרון של ה-Decorator</u>

~ מספק גמישות רבה יותר מאשר הורשה סטטית.

~ משפר את יכולת ההרחבה של האובייקט, מכיוון ששינויים נעשים על ידי כתיבת מחלקות חדשות.

~ מפשט את הקידוד בכך שהוא מאפשר לך לפתח סדרה של פונקציונליות ממחלקות ממוקדות במקום לקודד את כל ההתנהגות לאובייקט.

<u>שימוש בתבנית ה-Decorator</u>

~ כאשר אתה רוצה להוסיף אחריות על אובייקטים באופן שקוף ודינמי מבלי להשפיע על אובייקטים אחרים.

~ כאשר אתה רוצה להוסיף אחריות לאובייקט שאולי תרצה לשנות בעתיד.

~ הרחבת הפונקציונליות על ידי סיווג משנה (הורשה / מימוש) כבר אינה מעשית.

- **Bridge Pattern**

Bridge Pattern אומר "נתק את הפונקציונליות המופשטת מהיישום כך שהשניים (הממשק והיישום) יכולים להשתנות באופן עצמאי".

#Bridge Pattern ידועה גם בשם handle או body.

<u>היתרון של Bridge Pattern</u>

~ מאפשר הפרדת היישום מהממשק.
~ משפר את יכולת ההרחבה.
~ מאפשר את הסתרת פרטי היישום מהלקוח.

שימוש ב-Bridge Pattern

~ כאשר אתה לא רוצה כריכה קבועה בין פונקציונליות מופשטת ליישומה.
~ כאשר יש צורך להרחיב גם את ההפשטה הפונקציונלית וגם את היישום שלה באמצעות מחלקות משנה.
~ הוא משמש בעיקר באותם מקומות שבהם שינוי שמתבצע ביישום אינו משפיע על הלקוחות.

- **Facade**

Facade Pattern אומר "ספק ממשק מאוחד ומפושט לקבוצת ממשקים בתת-מערכת, לכן זה מסתיר את המורכבות של תת-המערכת מהלקוח". במילים אחרות, Facade Pattern מתאר ממשק ברמה גבוהה יותר שמקל על השימוש בתת-מערכת.

למעשה, כל מפעל מופשט הוא סוג של חזית.

יתרונות ל Facade Pattern

~ מגן על הלקוחות מהמורכבות של רכיבי תת המערכת.
~ מקדם צימוד רופף בין תת-מערכות ללקוחותיה.

שימוש ב-Facade Pattern

~ כאשר אתה רוצה לספק ממשק פשוט לתת-מערכת מורכבת.
~כאשר קיימות מספר תלותיות בין לקוחות לבין מימוש המחלקות של ההפשטה.

Behavioral

- **Command**

Command pattern אומרת "הקפיצו בקשה מתוך אובייקט כפקודה והעבירו אותה לאובייקט המפעיל. אובייקט Invoker מחפש את האובייקט המתאים שיכול לטפל בפקודה הזו ולהעביר את הפקודה לאובייקט המתאים והאובייקט הזה מבצע את הפקודה".

ידוע גם כ-**Action** or **Transaction** .

יתרונות של Command pattern

~ מפריד בין האובייקט שמפעיל את הפעולה לבין האובייקט שמבצע את הפעולה בפועל.

~ מקל להוסיף פקודות חדשות, מכיוון שמחלקות קיימות נשארות ללא שינוי.

שימושי Command pattern

~ כאשר אתה צריך להגדיר פרמטרים של אובייקטים לפי סוג פעולה.

~ כאשר אתה צריך ליצור ולבצע בקשות בזמנים שונים.

~ כאשר אתה צריך לתמוך בפונקציונליות החזרה, רישום או עסקאות.

- **Strategy**

Strategy Pattern אומר "מגדיר משפחה של פונקציונליות, מקבץ כל אחד והופך אותם לניתנים להחלפה".

דפוס ה-strategy ידוע גם בשם מדיניות.

יתרונות Strategy Pattern

~ מספק תחליף לתתי-מחלקות.
~ מגדיר כל התנהגות בתוך המחלקה שלה, ומבטל את הצורך בהצהרות מותנות קודם.
~ מקל על הרחבה ושילוב של התנהגות חדשה מבלי לשנות את היישום.

שימושי Strategy Pattern

~ כאשר המחלקות המרובות נבדלות רק בהתנהגויות שלהן, למשל. Servlet API.
~ משמש כאשר אתה צריך וריאציות שונות של אלגוריתם.

- **Observer**

Observer Pattern אומר "פשוט הגדירו תלות של אחד לאחד כך שכאשר אובייקט אחד משנה מצב, כל התלויים בו מקבלים הודעה ומתעדכנים באופן אוטומטי".

Observer Pattern ידוע גם בתור Dependents או Publish-Subscribe .

יתרונות Observer Pattern

~ מתאר את הצימוד (תלותיות) בין העצמים ל- Observer .
~ מספק תמיכה לתקשורת מסוג שידור.

שימושי Observer Pattern

~ כאשר שינוי של מצב באובייקט אחד חייב לבוא לידי ביטוי באובייקט אחר מבלי לשמור את האובייקטים צמודים (להישמר מתלותיות של אחד בשני).
~ כאשר אנחנו כותבים את השלד (framework) ונצטרך לשפר אותה בעתיד עם observers' חדשים ועם מינימום שינויים.