# תבניות עיצוב

## הגדרה

תבנית עיצוב (Design pattern) היא פתרון כללי לבעיה שכיחה בעולם עיצוב התוכנה תוך שמירה על עקרונות עיצוב קוד נכונים. תבנית עיצוב אינה עיצוב סופי שניתן להעבירו הישר לקוד, אלא תיאור או תבנית לדרך לפתרון בעיה, שעשויה להיות שימושית במצבים רבים. מתכנתים מתבססים על הפתרון המופשט וממרחיבים אותו כדי שיתאים לתוכנית שלהם. תבניות עיצוב מונחות עצמים מציגות לרוב יחסים וקשרי גומלין בין מחלקות או אובייקטים, בלי לפרט את המחלקות או אובייקטי היישום הסופיים המעורבים. אלגוריתמים אינם נחשבים כתבנית עיצוב, כיוון שהם פותרים בעיות חישוביות ולא בעיות עיצוב.[

### יתרונות

* Understandable - הגדרת התבניות תחת רעיון אחד, ובעיקר הגדרת שם התבנית מקילות על ההבנה בין מפתחים ממקומות שונים בהתייחסות לבעיה ולדרך פתרונה, כיוון שישנה שפה משותפת.
* Reusability - בנוסף כיוון שהתבניות עצמן מוגדרות באופן מופשט אך בצורה פורמלית, הן אינן תלויות בשפה או בסביבת פיתוח יחידה, דבר המאפשר צבירה והמשכיות של הידע הנלמד.
* Efficiency - שימוש חוזר בתבנית בדוקה וידועה עוזר לנו לכתוב קוד מהר ולחסוך בשגיאות.

## תיעוד

הגדרת תבנית עיצוב תכיל את הפרמטרים הבאים:

* שם תבנית העיצוב וסיווגה.
* תיאור מופשט של הבעיה אותה תבנית העיצוב נועדה לפתור.
* הצגת הפתרון בצורה מורחבת הכוללת תרשים OMT‏ או UML‏.
* תוצאות והשלכות של יישום הפתרון המוצע.

דוגמה:

שם - תבנית Singleton, תבנית יצירה.

בעיה - יש צורך במופע יחיד של מחלקה.

פתרון - הגדרת בנית המחלקה רק בתוכה והחצנת פונקציה שרק בקריאה הראשונה אליה תיצור מופע של המחלקה (ראו פרוט בערך עצמו).

השלכות - שימוש לא נכון בתבנית העיצוב עלול להוביל לקוד בו המחלקות בצמידות או תלות (Coupling) גבוהה.

## סוגי תבניות

באופן כללי, ניתן לחלק את התבניות לשלושה סוגים:

### תבניות יצירה (Creational Patterns)

מטרת תבניות אלו היא ליצור הפשטה לתהליך היצירה של אובייקטים, על ידי ביצוע הפרדה בין המערכת לבין יצירה או תצוגה של האובייקטים הנוצרים. כלומר, לא נתייחס לסוג המחלקה ממנה ניצור אובייקט ותהליך היצירה עצמו. בנוסף, לאחר היצירה הגישה לאובייקט הנוצר תעשה באמצעות ממשק.

### תבניות מבנה (Structural Patterns)

מטרתן של תבניות אילו היא להקל על עיצוב המערכת על ידי זיהוי דרך פשוטה לממש קשרים בין אובייקטים.

### תבניות התנהגות (Behavioral Patterns)

כאשר מפתחים מערכת גדולה ישנו קושי בניתוח אינטראקציה בין אובייקטים, לכן עולה הצורך להגדיר סמכויות ולאפיין את אופי התקשורת בין האובייקטים שבה. תבניות התנהגות, עוסקות בבעיות אלו, בדומה לארגון, על ידי הגדרת הסמכויות הנדרשת והתקשורת בין האובייקטים. ירושה והרכבה עוזרים לתבניות התנהגותיות בהגדרת היחסים בין המחלקות.

# תבניות יצירה

## Singleton

תבנית סינגלטון היא תבנית אשר נועדה למקרים בהם מעוניינים להגביל את יצירת האובייקטים של מחלקה מסוימת למופע אחד ויחיד. התבנית צריכה לעמוד בשני עקרונות: ניתן ליצור אובייקט יחיד של המחלקה, והגישה למחלקה היא גלובלית, כלומר כולם יכולים לגשת את המחלקה וליצור את האובייקט הראשון.

### מימוש

למימוש סינגלטון ניצור משתנה מחלקה סטטי פרטי שהוא מסוג המחלקה עצמה, כך שכבר בתחילת התוכנית קיים רק מופע אחד של המשתנה הזה. אמנם עדיין לא מאותחל לו שום ערך. לכן ניצור בנאי private שהוא יאתחל את הערך של משתנה המחלקה הסטטי, אך ניתן לגשת אליו רק מתוך המחלקה עצמה.

כעת ניצור פונקציה סטטית שבודקת אם המשתנה הסטטי כבר אותחל, אם כן מחזירה אותו, ואם לא מפעילה את הבנאי. כך אנו בעצם מבטיחים שהבנאי יופעל רק פעם אחת וייווצר רק אובייקט אחד מסוג המחלקה.

אמנם עדיין יש חשש שאם שני תהליכונים ניגשים לפונקציית הבנייה באותו הזמן, ייתכן מצב בו שניהם ייצרו אובייקט, מצב אותו אנו מנסים למנוע. לכן נגדיר רק את הקטע קוד הקריטי כמסונכרן ולא את כל הפונקציה, כדי לא ליצור פקק במצב שיש תהליכונים רבים.

public class Singleton {

private static Singleton instance;

private Singleton(){…}

public static Singleton getSingletongetInstance(){

if(instance == null){

synchronized(Singleton.class){

if(instance == null)

instance = new Singleton();

}

}

return instance;

}

}

# תבניות התנהגות

## Observer

תבנית עיצוב הנועדה למקרים בהם יש צורך שאובייקטים הנקראים Observers / Listeners יוכלו לקבל עדכון על שינוי כלשהו המתרחש באובייקט אחר הנקרא Subject. ה-Subject "מודיע" ל- Observersעל שינויים המתרחשים בו, בדרך כלל על ידי קריאה לאחת מהשיטות שלהם. תבנית זו שימושית ביותר בעולם האמיתי ויש לה דוגמאות רבות: נרשמים לערוץ יוטיוב שמצפים לקבל עדכון כאשר הערוץ מעלה סרטון, סוכנות ידיעות שמעדכנת ערוצי חדשות, וכו'.

### מימוש

כדי לממש את התבנית נצטרך לממש שני ממשקים: Subject ו-Observer. ל-Subject יש שלוש שיטות המאפשרות להירשם או לבטל רישום בתור צופה, ושיטה נוספת לעדכן את כל מי שנרשם כצופה ולשלוח אליו את העדכון. ואילו ל-Observer יש שיטה אחת שאומרת מה לעשות כאשר ה-Subject שולח עדכון.

public interface Subject {

public void register(Observer o);

public void unregister(Observer o);

public void notifyObserver();

}

public interface Observer{

public void update(); //update something

}

כל אובייקט שנרצה שישמש ב-Subject יממש את הממשק שלו, וכל אובייקט שנרצה שישמש ב-Observer יממש את הממשק שלו ונרשום אותו ל-Subject. שימוש בתבנית Observer שומרת על עקרון הכימוס. ה-Subject אינו צריך לדעת דבר על האובייקטים התלויים בו חוץ מזה שהם מממשים את Observer וצופים בו, ומבחינתו הוא נפרד מהם. וכן ה-Observers אינם יודעים דבר על מי שהם מאזינים לו, חוץ מזה שהוא מממש את Subject.

## Strategy

נרצה להגדיר מחלקה שמשתמשת במשפחה של אלגוריתמים עם אותה מטרה, אך המימוש משתנה בהתאם לצורך. לדוגמא: מחלקה שמפעילה אלגוריתמי מיון על מערך, ובהתאם לסוג המערך מפעילה בכל פעם אלגוריתם שונה שהוא הכי יעיל. בנוסף, נרצה שמחלקה זו תהיה מודולארית, כלומר שאם נוסיף אלגוריתם מיון חדש או שנשנה אלגוריתם קיים - לא נצטרך לשנות את המחלקה.

### מימוש

נגדיר ממשק בשם Strategy ולו שיטה אחת execute(). לכל אלגוריתם נכתוב מחלקה נפרדת שתממש את Strategy. בכל מחלקה השיטה execute תריץ את האלגוריתם המייצג את המחלקה.

כעת, במחלקה הראשית המשתמשת באלגוריתמים נגדיר בנאי שמקבל כפרמטר טיפוס מסוג Strategy. עובר כל אובייקט שנבנה ממחלקה, זו נכניס את המחלקה שמממשת את Strategy שאנו רוצים להשתמש באלגוריתם שלה.

public interface Strategy {

execute(int a, int b);

}

public class Add implements Strategy {

public int execute(int a, int b)

return a + b;

}

public class Subtract implements Strategy {

public int execute(int a, int b)

return a - b;

}

public class Context {

private Strategy strategy;

public Context(Strategy strategy) {

this.strategy = strategy;

public int executeStrategy(int a, int b) {

return strategy.execute(a, b);

}

## Visitor

Visitor זוהי תבנית עיצוב המאפשרת להגדיר פעולה על אובייקטים מסוגים שונים שאין להם מכנה משותף, מבלי לגעת בהיררכיית המחלקות שלהם. נשתמש בתבנית עיצוב זו כאשר יש לנו איזושהי היררכיית מחלקות שאנו לא רוצים לגעת בה יותר מדי כדי שלא להשפיע על קוד קיים, אך אנו צריכים להרחיב מספר מחלקות שלא קשורות ביניהם בהיררכיה זו שיבצעו פעולה מסוימת.

### מימוש

נגדיר ממשק Visitor ולו שיטה visit עבור כל מחלקה שאנו רוצים להוסיף לה שיטה. visit תקבל פרמטר מסוג המחלקה אותה היא רוצה להרחיב. נגדיר מחלקה כלשהי שתממש ממשק זה, ועבור כל שיטה תממש מה היא רוצה שהמחלקה שהתקבלה כפרמטר תבצע.

כעת נגדיר ממשק Element ולה שיטה אחת accept שמקבלת אובייקט מסוג ממשק Visitor כפרמטר. כל אחת מהמחלקות שאנו רוצים להרחיב תממש את Element, ובשיטה accept תפעיל את השיטה visit ותשלח את עצמה כפרמטר. נוכל ליצור מספר מחלקות שממשות את Visitor וכך להוסיף כמה שיטות שנרצה. כל מחלקה שמממשת את Visitor מוסיפה שיטה אחת למחלקות. כך בעצם אנו מאפשרים לכל מחלקה להוסיף כמה שיטות שנרצה עם שינוי מינימלי של המחלקות.

pubic interface Visitor{

public void visit(ClassA a);

public void visit(ClassB b);

}

public class VisitorOne{

public void visit(ClassA a){…}

public void visit(ClassB b){…}

}

public class VisitorTwo{

public void visit(ClassA a){…}

public void visit(ClassB b){…}

}

public interface Element{

public void accept(Visitor v);

}

public class ClassA{

public void accept(Visitor v){ //v can be VisitorOne or VisitorTwo

visit(this)

}

public class ClassB{

public void accept(Visitor v){

visit(this)

}