תבניות עיצוב

Design Patterns

שיעור 1

תבניות עיצוב הן, באופן כללי ומופשט, מעין שפה משותפת כך שאפשר לדבר על מה עשינו באמצעות שמות.

<u>עכשיו קצת מפורט (מתוך הספר שבאתר):</u>

1. דו-שיח בין אובייקטים ממחלקות שונות.
2. אופן יצירת אובייקטים.
3. שדות מידע רלוונטיים בכל מחלקה.
4. אסטרטגיות ירושה/הכלה של מחלקות שונות.

ישנן תבניות יצירה, תבניות מבנה ותבניות התנהגות.

דוגמא לתבנית יצירה היא Factory Method שבה אני בעצם יוצר מופע בתוך מתודה ובכך אני מכמס (encapsulation) את כל המידע שעומד מאחורי יצירת הפרויקט.

דוגמא לתבנית עיצוב היא Adapter, נלמד עליה בהמשך, אבל דוגמא מהחיים זה מתאם, כמו מתאם שאנו קונים לחיבור מסוים למחשב. מבחינת תכנית זה תיאום שבא לעזור לנו כאשר אנחנו לא יכולים או לא מורשים לגעת במחלקות ולשנות שם דברים.

דוגמא להתנהגות, תבנית Visitor, היא עוסקת במצבים בהם יש לי גורם חיצוני (תהליך או מחלקה או משהו אחר) שיכול לבקר בתוכנית שלי ולעשות בה פעולות, כמו שרופא מטפל בילד שלי, הילד עדיין שלי גם אם הרופא הוא זה שמטפל.

נעבור על תבניות לפי סדר.

Singletone:

ישנם מקרים בהם יש מחלקה ואנו רוצים לוודא שיש מופע אחד בלבד.

נניח שיש קובץ כלשהו שמוגדר שרק מישהו אחד נוגע בו ואז כדי שאשתמש בו אני אצטרך שמי שנמצא בו עכשיו יפסיק את השימוש בו.

גם בתכנות קורה מצב שאנו כותבים בתוך ויש עוד תהליכים במחשב שכותבים לתוך אותו הקובץ ואם שני תהליכים ינסו לכתוב יחד אז בעצם תהיה התנגשות שקפיץ שגיאה.

מקרה נוסף נפוץ, הוא כאשר יש חיבור למסד נתונים כלשהו ויש חבילה שיוצרת את החיבור.

אם נפנה אל המחלקה מכמה תהליכונים, אני לא רוצה שכל מחלקה שנפנה דרכה תפתח מופע משלה (חיבור משלה) אני תופס המון מהרשת ואני לא בטוח שהשרת מוכן לכל כל הרבה פניות במקביל.

במקרים כאלו אנו משתמשים בתבנית singletone, תבנית שבאמצעותה אני מאפשר למחלקה שיווצר מופע אחד בלבד.

מעבר לשימושים כאלה, פחות משתמשים בזה.

<u>תיקוף של התבנית:</u>

1. חסימת אפשרות לגישה ציבורית למחלקה.
2. חסימת אפשרות ייצור של יותר ממופע אחד.
3. וידוא שהמופע שנוצר הוא המופע שנשאר תמיד ולא מתאפס ונוצר מופע חדש.
4. תמיכה בקוד למקרה של ריבוי תהליכים.

אז יש לנו בתוך המחלקה שאותה נרצה לעשות כ-singletone מופע סטטי שמחזיק את המופע ואובייקט בשביל הנעילה.

בנוסף יש בנאי שלא חשוף החוצה, ויש לנו מתודה סטטית שתוכל להנגיש החוצה את המופע.

```
class LockOptimaizedSingelton
{
    // PRIVATE CONSTRUCTOR!!!!

    private LockOptimaizedSingelton()
    {
    }

    // THE INSTANCE IS ALSO PRIVATE!

    private static LockOptimaizedSingelton instance;

    // WE ADD LOCK TO HANDLE MULTI THREADS

    private static readonly object lockObject = new object();

    // Creating Timestamp just to ensure single instance creation

    private long Timestamp = DateTimeOffset.UtcNow.ToUnixTimeMilliseconds();

    // You can get the single instance only from here:

    public static LockOptimaizedSingelton GetInstance()
    {
            if (instance == null)
            {
                lock (lockObject)
                {
                    instance = new LockOptimaizedSingelton();
                }
            }
        return instance;
    }

    public void SomeMethod()
    {
        Console.WriteLine($"timestamp is {Timestamp}");
    }
}
```

המספר שנקבל ב-Timestamp שזה חותמת זמן, הוא מהמחלקה DateTimeOffSet שיודעת להחזיר לי את השעה לפי גריניץ' שזה הזמן הבינלאומי ללא קשר למדינה מסוימת.

הסבר קטן:

בעולם המיחשוב הרבה פעמים צריך להתעסק בזמנים, מה שעושים לרוב זה להשתמש בזמנים בשניות או במילי-שניות, זה דבר מאוד נפוץ וניתן לראות בהרבה מקרים שהזמן שמור כמספר השניות החל מה-01/01/1970.

לדוגמא, כרגע הזמן הזה הוא 1641682437 וכך מתקדם לאט עם השניות כך שעכשיו כבר 1641682442.

נחזור לקוד:

השתמשנו ב-lock כדי ליישם את התיקוף האחרון ברשימה לעיל, כך שהפעולה הזו בעצם מניחה את התהליכונים שפונים אליו במקביל בתור, וזה שהגיע ראשון הוא זה שייכנס ראשון וכך לפי התור וימנע פנייה מרובה וקריסה של התכנית.

כדי לקרוא מה-Program למחלקה ולראות שאכן נוצר רק מופע אחד ניצור שני מופע בהפרש של שנייה באמצעות פקודת sleep וכך נראה בעצם שחותמת הזמן לא השתנתה מה שיוכיח את יצירת המופע הראשון והישארותו.

```
var ins3 = LockOptimaizedSingelton.GetInstance();

ins3.SomeMethod();

Thread.Sleep(1000);

var ins4 = LockOptimaizedSingelton.GetInstance();

ins4.SomeMethod();
```

Chain Of Responsibility:

התבנית הזו היא כמו שמה, העברת אחריות ממחלקה אחת לאחרת.

לדוגמא: כאשר אנו פונים לשירות לקוחות, לעיתים אנו מטפלים בכמה דברים ואז מגיעים למצב שאנו צריכים את טיפול אחראי המשמרת אז מעבירים אותנו אליו, לאחר מכן ישנו טיפול מסוים שנצטרך לעלות רמה למנהל המוקד ולאחר מכן אולי גם למנהל ראשי.

כמובן שלא תמיד נהיה חייבים לעבור את כל השרשרת הזו, כי יכול להיות שהבעיה תיפתר כבר אצל המוקדן הראשון.

התבנית עובדת כך:

1. לכל רמה (חוליה) תהיה מתודה בשם HandleRequest שתקבל את האובייקט שמייצג את הבקשה.
2. בתוך המתודה נפעל למילוי הבקשה או פתרון הבעיה – אם הצלחנו, מעולה, הבעיה נפתרה.
3. בדיקה האם הבעיה נפתרה.
4. לא נפתרה? – נבדוק אם יש חוליה נוספת ואם כן נעביר אליה את האפשרות לפתור את הבעיה וכך ממשיכים עד שהבעיה תיפתר (או שלא..)

אופן החיבור:

כל חוליה, או גורם מטפל, יחזיק מתודה בשם SetNext שתקבל את הגורם הבא. נשים לב שהגורם האחרון בשרשרת יקבל null.

האפשרות הזו תאפשר לנו להוסיף או להוריד חוליות כרצוננו או בהתאם לתכנית מבלי לשנות את המחלקות האחרות.

מבחינת הקוד:

יש לנו מחלקה אבסטרקטית בסיסית בשם BaseHandler שמחזיקה במתודות HandleRequest ו-SetNext. למחלקה הזו תהיינה מחלקות יורשות (כמובן שחייבות לממש את המתודות).

בתוכן אני מחזיק מופע מאותו הסוג (מסוג BaseHandler) אפילו שאלו לא אותם המופעים.

נראה ממש בקוד:

נניח שאנו רוצים לכתוב קוד שמוציא כסף מכספומט ואנו רוצים למשוך 770 שקלים.

אז יש לנו מחלקות שאחראיות לכל סוג של שטר, מחלקה של 200, מחלקה של 100, מחלקה של 50 ומחלקה של 20.

אז כל שלב יוסיף את החלק שלו לפתרון הבעיה (משיכת 700 השקלים).

מחלקת ה-200 תשלח 3 שטרות ותעביר למחלקת ה-100 שהיא תוציא שטר אחד ותעביר למחלקת ה-50, אם סיימנו לקבל את הכסף אז השרשרת תיעצר כאן, אחרת תמשיך. (במקרה שלנו היא אן תמשיך לקבל שטר נוסף מ-50 ועוד אחד מ-20).

כמובן שיכול להיות מקרה שבו לא פתרנו כלל את הבעיה, נניח אם הלקוח ינסה למשוך 770.5 ואז אין לנו איך לתת לו את חצי השקל הנוסף.

```
// Creating our basic abstract class for inheritance

public abstract class BillHandler
{
    protected BillHandler next;
    public  void SetNext(BillHandler next)
    {
        this.next = next;
    }

    public abstract void HandleRequest(int amount);
}
```

במחלקה הבסיסית הגדרנו משתנה next מסוג המחלקה עצמה כך שהמתודה SetNext מקבלת בתוכה ארגומנט מהסוג של המחלקה עצמה גם כן שיגדיר את המשתנה next של המחלקה לפי אותו הארגומנט – באמצעות המתודה הזו נוכל לקבוע מבחוץ מי תהיה החוליה הבאה בשרשרת.

והמתודה HandleRequest שהיא בעצם מקבלת את הבעיה\הבקשה.

<u>כעת ניצור מחלקה לשטר של 200:</u>

```
public class BillHandler200 : BillHandler
{

    public override void HandleRequest(int amount)
    {
        if (amount >= 200)
        {
            Console.WriteLine("Giving 200 X " + amount / 200);
        }

        if (amount % 200 > 0)
        {
            if (next != null)
            {
                next.HandleRequest(amount % 200);
            }
        }
    }
}
```

כך נבנה גם מחלקות של 100, 50 ו-20 שנראות אותו הדבר וההבדל היחיד הוא הסכום.

נשים לב שירשנו את המחלקה האבסטרקטית ולכן אנו מממשים את המתודות.

בטיפול הבעיה (HandleRequest), אנו בודקים אם הכמות שקיבלנו גדולה מ-200.

אם אכן גדולה המתודה תחזיר לנו את מספר השטרות הנכון לפי חלוקה ב-200 (כאשר המשתנה הוא int אז החלוקה תחזיר את השלם הקרוב מלמטה), אם לא גדולה, נבדוק אם השארית קיימת וגדולה מ-0.

אם גדולה, נעבור למחלקה המטפלת הבאה (במקרה שלנו כנראה שתהיה 100).

נזכור ש-next נמצא במחלקה שירשנו ממנה (כשיורשים – יורשים הכל), וכרגע אין בו משהו, אז נבדוק אם הוא null, כי אז אין בו הפעלה של המתודה. ואז אם הוא לא null נפעיל דרכו את הטיפול הבא.

<u>לצורך הדוגמא שלנו, אנו מגדירים את השרשרת בצורה הבאה ב-Program:</u>

```
BillHandler bill200 = new BillHandler200();
BillHandler bill100 = new BillHandler100();
BillHandler bill50 = new BillHandler50();
BillHandler bill20 = new BillHandler20();

bill200.SetNext(bill100);
bill100.SetNext(bill50);
bill50.SetNext(bill20);
bill20.SetNext(null);
```

יצרנו בעצם מופעים מסוג כל מחלקה באמצעות המחלקה הראשית שכולן יורשות ממנה.

לאחר מכן, הגדרנו את שרשרת הטיפול.

ברגע שנקרא לפונקציה HandleRequest עם סכום מבוקש השרשרת תתחיל את סדר הפעולות שלה.

```
bill200.HandleRequest(770);
```

כמובן שנוכל להגדיר שרשרת פיקוד שונה אם נגדיר ב-SetNext שירשור אחר.

Adapter:

קורים מקרים שבהם אנו מקבלים ממשק של מישהו אחר, נניח יש לנו ספק של אימיילים, והבוס אמר שיש ספק אחר שהמחיר שם טוב יותר ורוצה שנממש את זה לספק אחר.

הבעיה היא שהקוד שלנו כבר עובד עם הספק הנוכחי ואני צריך איזושהי התאמה כדי להתאים את הכל עכשיו לספק החדש.

התבנית הזו בעצם פותרת לנו את הבעיה.

בתבנית הזו אנו עוטפים את המחלקה בתצורה אחרת ונבין את זה יותר טוב לפי הדוגמא.

נניח שיש לי שני מחשבונים או יותר נכון שתי מחלקות שכל אחת היא מחשבון, אבל מחשבון אחד יודע לעבוד עם מספרים שלמים והשני יודע לעבוד עם מספרים עשרוניים.

נבין כמובן, שהדוגמא לא בדיוק מציאותית אבל זה לשם הרעיון.

הבעיה מתחילה כאשר דורשים מאיתנו להשתמש באחד ואני צריך תוצאה של השני.

מבחינת הקוד:

אנו יוצרים שני Interfaces, שהם ייצגו את המחשבונים, אחד רגיל ואחד מדעי (עשרוני).

```
interface ISimpleCalc
{
    int Add(int x, int y);
    int Sub(int x, int y);
    int Div(int x, int y);
    int Mul(int x, int y);
}


interface IScientificCalc
{
    double Add(double x, double y);
    double Sub(double x, double y);
    double Div(double x, double y);
    double Mul(double x, double y);
}
```

וכמובן את המחלקות של המחשבונים עצמם, כאשר כל אחד מממש את ה-Interface שמתאים לו.

```
public class SimpleCalculator : ISimpleCalc
{
    public int Add(int x, int y)
    {
        return x + y;
    }

    public int Div(int x, int y)
    {
        return x / y;
    }

    public int Mul(int x, int y)
    {
        return x * y;
    }

    public int Sub(int x, int y)
    {
        return x - y;
    }
}


public class ScientificCalculator : IScientificCalc
{
    public double Add(double x, double y)
    {
        return x + y;
    }

    public double Div(double x, double y)
    {
        return x / y;
    }
```

```
        public double Mul(double x, double y)
        {
            return x * y;
        }

        public double Sub(double x, double y)
        {
            return x - y;
        }
    }
```

כעת, נניח שיש לנו מתודה שמוגדרת באופן הבא:

```
    static void PrintMathResults(IScientificCalc calculator, double x, double y)
    {
        Console.WriteLine(calculator.Add(x, y));

        Console.WriteLine(calculator.Sub(x, y));

        Console.WriteLine(calculator.Div(x, y));

        Console.WriteLine(calculator.Mul(x, y));
    }
```

ה"בעיה" שלנו היא שהמתודה דורשת רק מחשבון מהסוג של ה-Interface המדעי.

במצב הנוכחי הקוד הבא יעבוד:

```
IScientificCalc scientificCalc = new ScientificCalculator();
double x = new Random().NextDouble() * 1000.0;
double y = new Random().NextDouble() * 1000.0;
PrintMathResults(scientificCalc, x, y);
```

כי אני מזין למתודה את מה שהיא צורכת.

אבל כאשר נכתוב את הקוד הבא:

```
// THIS WILL NOT COMPILE!!
ISimpleCalc simpleCalc = new SimpleCalculator();
int xInt = new Random().Next(1000);
int yInt = new Random().Next(1000);
PrintMathResults(simpleCalc, xInt, yInt);
```

המערכת לא תקבל את זה משום שאנו מכניסים מחשבון מהסוג הרגיל ולא המדעי.

וכאן נכנסת התבנית לתמונה.

אנו יוצרים מחלקה שמממשת את ה-Interface של המחשבון המדעי ומחזיקה בתוכה מופע של המחשבון הרגיל.

בכל מתודה ומתודה של חישוב בעצם יוחזר לנו חישוב לפי המופע הפנימי של מחשבון רגיל.

אז במחלקה הזו אנו בעצם מחזיקים את המימוש החדש (נניח את ספק האימיילים החדש מהדוגמא בהתחלה) ועטפנו את המחלקה, כלומר אנחנו כאילו משתמשים במחשבון המדעי אבל לא באמת משתמשים בו.

ברור שבמחשבון זה לא משהו יעיל אבל שוב, זה רק לשם ההמחשה.

```
public class AdapterCalc : IScientificCalc
{
    private ISimpleCalc _simpleCalc = new SimpleCalculator();

    public double Add(double x, double y)
    {
        return _simpleCalc.Add((int)x, (int)y);
    }

    public double Div(double x, double y)
    {
        return _simpleCalc.Div((int)x, (int)y);
    }

    public double Mul(double x, double y)
    {
        return _simpleCalc.Mul((int)x, (int)y);
    }

    public double Sub(double x, double y)
    {
        return _simpleCalc.Sub((int)x, (int)y);
    }
}
```

כעת הקוד הבא כבר כן יעבוד:

```
// BUT IF WE WILL USE THE ADAPTER - IT WILL WORK!

IScientificCalc adapterCalc = new AdapterCalc();
int xInt = new Random().Next(1000);
int yInt = new Random().Next(1000);
PrintMathResults(adapterCalc, xInt, yInt);
```

Factory:

עד היום רוב המחלקות שיצרנו, הן קטנות ולא בנייה רצינית וסיפור גדול.

אבל לא תמיד זה כך ומחלקות יכולות להיות מאוד גדולות ולבצע המון דברים, והיצירה של האובייקט הופכת להיות מסובכת יותר.

התבנית הזו אומרת לנו באופן כללי להוציא את העבודה הזו החוצה במקרה שהיא קשה מדי ומסובכת מדי בדרך הרגילה.

אז במצב הפשוט אנחנו יוצרים מופע עם המילה new וזהו. אבל כאשר אנו רוצים להחזיר שניים שונים שיהיו מאותו הסוג בדרך נחשוב על אבסטרקציה.

יש לנו מחלקה סטטית שהיא עושה פעולה אחת והיא יצירה של מופע חדש.

על פניו נראה כאילו לא עשינו כלום אבל יש לנו 2 יתרונות לדבר הזה:

1. נניח ובעתיד יוסיפו לי משהו בצורת הייצור ואצטרך להוסיף גם פרמטר, אז בגלל שלא עשיתי new בכל מקום שבו יצרתי מופע אלא השתמשתי בכולם עם המתודה של היצירה אז מספיק שאשנה שם.
2. אם נצטרך להחזיר אובייקט דומה אבל שונה. כלומר נניח יש לנו מחלקה אחת שהיא גיבוי במקרה שהשנייה לא עובדת אז נוכל לקרוא לה באמצעות היצירה עם תנאי (אם כן אז תיצור מופע x אחרת תיצור מופע של y).

מה לא לעשות:

- לא לעטוף אובייקטים של דוטנט – כמו רשימה, כי היא כבר משהו מוגדר.
- אם האובייקט פשוט לא צריך לעשות את כל המעטפת, רק כאשר הוא מורכב.

מה כן:

- אם יש לי ספרייה חיצונית, אז לא להשתמש בה ישירות אלא עם המעטפת.

<u>דבר נוסף קטן לפני שנמשיך:</u>

הרבה פעמים אנשים עושים Factory Method, כלומר מתודה שיוצרת את המופע אך ורק בגלל שזה נראה טוב יותר לעין ביצירה. כלומר, לא בהכרח משתמשים בזה רק לשם כל מה שדברנו עד עכשיו.

```
public interface IBuisnessThing
{
    void SomethingIntersting();
}


public class ConcreteBuisnessObject_3 : IBuisnessThing
{
    public ConcreteBuisnessObject_3(string name)
    {

    }

    public void SomethingIntersting()
    {

    }
}

public class SecondConcreteBuisnessObject : IBuisnessThing
{
    public SecondConcreteBuisnessObject(string name)
    {

    }

    public void SomethingIntersting()
    {

    }
}


public static class ObjectFactory_3
{
    public static IBuisnessThing Create()
    {
        return new ConcreteBuisnessObject_3("Moshe");

        var someSetting = true;
        if (someSetting)
        {
            return new ConcreteBuisnessObject_3("Moshe");
        }
        else
        {
            return new SecondConcreteBuisnessObject("bzzzzz");
        }
    }
}
```

שיעור 2

רקורסיה:

באופן כללי זה בעצם פונקציה שקוראת לעצמה.
נניח בדוגמא הבאה אנו רואים פונקציה שסופרת עד n בצורה רקורסיבית:

```
static void P(int n)
{
    if (n < 5)
    {
        Console.WriteLine("n = " + n);
        P(n + 1);
    }
}
```

דגש חשוב – תמיד ברקורסיה יהיה **תנאי עצירה**.

כאשר כתבנו את המתודה בצורה הבאה:

```
static void P(int n)
{
    if (n < 5)
    {
        Console.WriteLine("n = " + n);
        P(n + 1);
        Console.WriteLine("n (after) = " + n);
    }
}
```

מה שקרה זה שנכתבו כל ה-n-ים לפי ה-writeLine הראשון ורק אז נכתבו כל ה-n-ים לפי ה-writeLine השני.
זה קורה כך משום שהפונקציה קוראת לעצמה ולכן היא שומרת את ה-writeLine השני לסוף, ואחרי שהיא יוצאת לפי תנאי העצירה היא בעצם חוזרת להשלים ולבצע את כל מה שהיא שמרה.

עכשיו נראה דוגמא קצת יותר רצינית:

```
static int PrintNatural(int currentNumber, int targetNumber)
{
    if (targetNumber < 1)
    {
        return currentNumber;
    }
    targetNumber--;
    Console.WriteLine("{0}", currentNumber);
    return PrintNatural(currentNumber + 1, targetNumber);
}
```

החל מהמספר שהוזן ראשון (current) ייספרו ויודפסו המספרים בקפיצות של אחד כמספר הפעמים שהוזן בשני (target).

דוגמא נוספת ואחרונה:

```
static int CalculateSumRecursively(int n, int m)
{
    int sum = n;
    Console.WriteLine("Sum = " + sum);
    if (n < m)
    {
        n++;
        return sum += CalculateSumRecursively(n, m);
    }

    return sum;
}
```

כאשר בפונקציה הזו אני סוכם את המספרים מ-n ועד m.

Proxy:

נניח שאנחנו בונים את התוכנה שמפעילה רכב אוטונומי. בתוכנית הזו יש מחלקות וממשקים עם יכולות מגוונות.

התבנית הזו אומרת לי בעצם שכאשר יש לי תוצר עם יכולות מגוונות ורבות ואני רוצה להנגיש למישהו שמשתמש בקוד הזה רק חלק מצומצם מהיכולות אז אני משתמש ב-proxy שזה בעצם מעיין מתווך שמסנן בדרך לפי מה שרציתי.

נניח כאשר מגיע הטכנאי שבודק את הרכב, הוא לא יצטרך שכל הפעולות יהיו פתוחות בפניו, אלא רק חלק כמו סימון תקלה ומיקום שלה או עוד כמה דברים.

כמובן שאם הדוגמא הייתה על בנק אז הרבה יותר ברור לי שהפעולה של פתיחת כספת לא תהיה פתוחה ללקוח שרק מגיע להוציא כסף.

סך הכל מעט דומה ל-Adapter אבל לא לגמרי ונראה עכשיו.

אז ניקח את הדוגמא מהשיעור שמופיעה גם בספר הקורס:

לרכב יש כל מיני יכולות:

1. פתיחת דלת
2. סגירת דלת
3. התחלת נסיעה
4. סיום נסיעה
5. דיווח מיקום
6. דיווח מצב הדלק
7. התחלת תיקון
8. סיום תיקון

אז נניח שיש לנו שני Interfaces שאחד הוא פעולות הרכב עצמו והשני הוא פעולות תיקון של הרכב.

בנוסף תהיה לנו את מחלקת הרכב שתממש את שני ה-Interfaces ותהיה מחלקת proxy שהיא בעצם תהיה המחלקה של הטכנאי. אז איך ננגיש לטכנאי את מחלקת הרכב עצמה? נראה עכשיו.

אז יש לנו את שני ה-Interfaces:

```
public class GPS
{
    public string Latitude { get; set; }
    public string Longitude { get; set; }
}
```

זו מחלקה שנכתבה כדי שיהיה לי משתנה GPS שבו יש אורך ורוחב שמרכיבים לי קואורדינטה

```
public interface ICar
{
    void OpenDoor();
    void CloseDoor();
    void Drive();
    void Stop();
    void Reverse();
}

public interface IRescue
{
    GPS GetLocation();
    float GetGasReport();
    void StartRepair();
    void EndRepair();
}
```

אלו ה-Interfaces שלנו

```
public class Car : ICar,IRescue
{
    private GPS gps;

    private float gas;

    public void CloseDoor()
    {
        // ...
    }

    public void Drive()
    {
        // ...
    }

    public void EndRepair()
    {
        // ...
    }

    public float GetGasReport()
    {
        return gas;
    }

    public GPS GetLocation()
    {
        return gps;
    }
```

זו מחלקת רכב שמממשת את שני ה-Interfaces שלנו על כל מתודותיהם.

```
    public void OpenDoor()
    {
        // ...
    }

    public void Reverse()
    {
        // ...
    }

    public void StartRepair()
    {
        // ...
    }

    public void Stop()
    {
        // ...
    }
}

public class CarProxy : IRescue
{
    private Car real_car;

    public CarProxy(Car real_car)
    {
        this.real_car = real_car;
    }

    public void EndRepair()
    {
        real_car.EndRepair();
    }

    public float GetGasReport()
    {
        return real_car.GetGasReport();
    }

    public GPS GetLocation()
    {
        return real_car.GetLocation();
    }

    public void StartRepair()
    {
        real_car.StartRepair();
    }
}
```

זו מחלקת ה-proxy שאיתה אני בעצם מחליט מה לחשוף ומה לא

Strategy:

לפעמים יש מצב שבו יש לנו כמה אפשרויות לעשות משהו והבחירה באיזו דרך לבחור תלויה בכמה משתנים ואנחנו צריכים לשנות את ההתנהגות או יותר נכון, לנהל את הבחירה בלי שישימו לב בתוכנה.

נניח שיש לי רשימות מספרים ואני רוצה למיין אותן, ויש לי המון איברים.

נזכור שמבחינת המשתמש הרצון הוא להגיע למיון באופן מהיר ולא מעניין אותו מה קורה מאחורי הקלעים.

ניקח את הדוגמא של המיון ונראה אותה גם בקוד.

יש לנו בעצם שתי מחלקות שמממשות Interface של מיון אבל כל אחת דואגת למיין מצב אחר.

מחלקה אחת ממיינת עד מספר איברים מסוים ומחלקה שנייה כאשר עברנו את מספר האיברים הזה.

בנוסף תהיה לנו מחלקה שתנהל את הרשימה שבה נוסיף איברים וכאשר נעבור את מכסת האיברים שתוגדר המחלקה תדע להשתמש במיון הנכון.

```
public interface ISort
{
    void Sort(List<int> numbers);
}

public class SmallScaleSorter : ISort
{
    public void Sort(List<int> numbers)
    {
        numbers.Sort();
    }
}

public class LargeScaleSorter : ISort
{
    public void Sort(List<int> numbers)
    {
        numbers.Reverse();
    }
}

public class ListManager
{
    private ISort _sorter = new SmallScaleSorter(); // Start with the small sorter.
    private List<int> _numbers = new List<int>();
    private const int LIST_THREASHOLD = 3;

    public ListManager()
    {
    }

    public void AddNumber(int item)
    {
        _numbers.Add(item);
        if (_numbers.Count >= LIST_THREASHOLD)
        {
            _sorter = new LargeScaleSorter();
        }
    }
```

נשים לב שכאן ביקשנו ממנו לעשות reverse רק בשביל הדוגמא כדי שנראה את השינוי כשנריץ את הקוד.

במחלקת ניהול הרשימה אנו מחזיקים מופע של המיון הקטן וכמובן את הרשימה עצמה, כולם פרטיים.

בנוסף אנו מחזיקים משתנה const, שישמש אותי כדי להגדיר את מכסת האיברים למעבר בין סוגי מיונים.

ובעצם כאשר נעבור את המכסה הממיין שלי יהפוך להיות הממיין הגדול.

```
        public void RemoveNumber(int item)
        {
            _numbers.Remove(item);
            if (_numbers.Count < LIST_THREASHOLD)
            {
                _sorter = new SmallScaleSorter();
            }
        }

        public void Sort()
        {
            _sorter.Sort(_numbers);
        }

        public void PrintNumbers()
        {
            _numbers.ForEach(x => Console.Write($"{x},"));
            Console.WriteLine("------");
        }
    }
```

תזכורת קטנה: השוני בין const ל-readonly הוא ש-const שזה קבוע, לא ניתן יהיה לשנות אותו בין מופעים והוא מוגדר באופן אוטומטי כ-static, כלומר, לא תלוי במופע. לעומת זאת readonly יכול להשתנות ממופע למופע. בנוסף, const הופך לקבוע כבר בקומפילציה ו-readonly רק בשעת ריצת התוכנית.

State:

אז אנחנו מדברים על מצב שבו יש לנו איזשהי ישות שמתנהגת שונה לפי מצב מסוים.

לדוגמא, מעלית, שאם אני בזמן עלייה בתוך המעלית, הדלת לא נפתחת, אבל אם אהיה בחוץ והיא תגיע אליי הדלתות כן תפתחנה.

עוד דוגמא, אור, אם האור דלוק ונלחץ שוב על המתג להדלקה, לא יקרה כלום.

כעת נראה מימוש:

יש לנו מתג עם המצבים הבאים: דלוק, כבוי או שבור.

הפעולות שניתן לעשות הן להדליק או לכבות.

אם המתג דולק וננסה שוב להדליק, לא יקרה כלום אבל אם ננסה לכבות אז יקרה כיבוי. כעת אם ננסה לכבות לא יקרה כלום אבל להדליק שוב כן נוכל. אם אנסה לכבות אותו בכוח בזמן שהוא כבוי אז הוא יישבר ואז בזמן שהוא שבור גם אם ננסה להדליק וגם אם ננסה לכבות לא יקרה כלום, כי הוא שבור.

אז יש לנו Interface של פעולות המתג, שם יש שתי מתודות של הדלקה וכיבוי.

בנוסף יש לנו שלוש מחלקות מצבים: דלוק, כבוי ושבור שכל אחת מהן מממשת את ה-Interface.

בצד יש לנו מחלקה שנקראת Context שהיא מחזיקה את כל הלוגיקה ואת המצב של המתג.

נראה את הקוד בעמוד הבא.

```
public interface ILightState
{
    ILightState TurnOn();

    ILightState TurnOff();
}
```

Interface עם המתודות של הדלקה וכיבוי

```
public class IAmOn: ILightState
{
    public ILightState TurnOff()
    {
        Console.WriteLine("Turning off the light....");
        return new IAmOff();
    }

    public ILightState TurnOn()
    {
        Console.WriteLine("already on...");
        return this;
    }
}
```

מחלקת מצב דלוק.

אם נפעיל פעולת כיבוי, המצב יהפוך לכבוי, אחרת לא יקרה כלום.

```
public class IAmOff : ILightState
{
    public ILightState TurnOff()
    {
        Console.WriteLine("Light is already off... now it is broken....");
        return new IAmBroken();
    }

    public ILightState TurnOn()
    {
        Console.WriteLine("Turning on the light!");
        return new IAmOn();
    }
}
```

מחלקת מצב כבוי.

אם נפעיל פעולת כיבוי, המצב יהפוך לשבור, אחרת נדליק והמצב יהפוך לדלוק.

```
public class IAmBroken : ILightState
{
    public ILightState TurnOff()
    {
        Console.WriteLine("broken ... please call technician");
        return this;
    }

    public ILightState TurnOn()
    {
        Console.WriteLine("broken ... please call technician");
        return this;
    }
}
```

מחלקת מצב שבור.

אם נפעיל פעולת כיבוי, או פעולת הדלקה לא יקרה כלום

```
public class Context
{
    private ILightState _state;
    public Context(ILightState state)
    {
        _state = state;
    }
    public void TurnOnRequest()
    {
        _state = _state.TurnOn();
    }

    public void TurnOffRequest()
    {
        _state = _state.TurnOff();
    }
}
```

מחלקת Context.

מתחילה במשתנה פרטי שמחזיק את המצב הנוכחי, כאשר דרך הבנאי מקבלים את המצב הקיים.

כך כל מתודה של הדלקה\כיבוי תפעל לפי המתודה שבמחלקה של המצב הנתון.

כמובן שמבחינת ההרצה:

```
Context ctx = new Context(new IAmOff()); // starting as Off
ctx.TurnOnRequest(); // turn on
ctx.TurnOnRequest(); // do nothing. already on
ctx.TurnOffRequest(); // turn off
ctx.TurnOffRequest(); // break the switch ...
ctx.TurnOnRequest(); // do nothing (broken)
ctx.TurnOffRequest(); // do nothing (broken)
```

Composite:

אנחנו מדברים בעיקר על מבנה שמסתעף לפי היררכיה מסוימת, כמו עובדים בארגון או ארגון ספרים בחנות\ספרייה.

כמובן שהדוגמא הכי קרובה היא במחשב שלנו, שזה תיקיות וקבצים.

ההסתעפות היא בעצם עץ (מטאפורית) כאשר יש ענפים ועלים – ענף זו מחלקה בארגון ועלה זה העובד, או שענף זו תיקייה ועלה הוא קובץ.

אז במימוש יש לנו Interface שמתאים גם לענף וגם לעלים ובמימושים השונים אנחנו נחליט אם נממש גם את המתודות שעוסקות בעוד רמות או לא, נניח שיש מתודה שמבקשת להביא את הקבצים שבתיקייה ואת המתודה הזו לא נרצה לממש בקבצים (כלומר לא לכתוב שום קוד במתודה וכך היא לא תעשה כלום).

מבחינת הקוד:

לדוגמא שלנו נרצה לממש על ספריית קבצים שהמתודות שנשתמש בהן יהיו:

1. הצגה של ספריות וקבצים.
2. חישוב גודל של תת-ספריה.
3. מחיקה של תת-ספריה וכל התכולה שלה.

אז תהיה לנו מחלקה אבסטרקטית שתייצג רכיב – Component, שבה יהיה מאפיין שם ומתודות של הצגה, חישוב גודל, הוספה והסרה.

לאחר מכן תהיינה לנו שתי מחלקות שאחת תהיה מחלקת עלה והשנייה תהיה מחלקת Composite שמחזיקה בתוכה רשימה של Component – שזה בעצם ייצוג של תיקייה.

נראה את הקוד:

```
public abstract class Component
{
    public string Name { get; protected set; }

    public Component(string name)
    {
        Name = name;
    }

    public abstract void Add(Component c);

    public abstract void Remove(Component c);

    public abstract void Display(string space);

    public abstract long GetSize();
}
```

כאן אנו מחזיקים את המחלקה האבסטרקטית שמחזיקה בתוכה את המתודות הרצויות

מחלקת עלה (קובץ) שיורשת את Component ומממשת את המתודות, כאשר אין שימוש במתודות של הוספה ומחיקה

```
public class Leaf : Component
{
    public Leaf(string name) : base(name)
    {
    }

    public override void Add(Component c)
    {
        throw new NotSupportedException("Leaf element cannot add child!");
    }

    public override void Remove(Component c)
    {
        throw new NotSupportedException("Leaf element cannot remove child!");
    }

    public override void Display(string space)
    {
        Console.WriteLine($"{space} {this.Name} size: {GetSize().ToString("#,##0")}");
    }

    public override long GetSize()
    {
        FileInfo fileInfo = new FileInfo(this.Name);
        return fileInfo.Length; // this returns the size of the file
    }
}
```

כאן אנו רואים שימוש במחלקה FileInfo שמאפשרת לנו החזיר את גודל הקובץ באמצעות length

```
public class Composite : Component
{

    private List<Component> children = new List<Component>();

    public Composite(string name) : base(name)
    {
    }

    public override void Add(Component c)
    {
        children.Add(c);
    }

    public override void Remove(Component c)
    {
        children.Remove(c);
    }

    public override void Display(string space)
    {
        Console.WriteLine($"{space} {this.Name} [Folder size:
{GetSize().ToString("#,##0")}]");

        foreach (Component c in children)
        {
            c.Display(space + "    ");
        }
    }

    public override long GetSize()
    {
        long sum = 0;

        foreach (Component c in children)
        {
            // adding the size of each child to the sum
            sum = sum + c.GetSize();
        }

        return sum;
    }
}
```

מחלקת Composite שבה התחלנו עם רשימה של Component שיכולה להכיל קבצים וגם תיקיות.

מתודת Add שמוסיפה לרשימה.

מתודה Remove שמסירה מהרשימה.

מתודה Display שמקבלת "רווח" שיעזור ליצור הזחות בהדפסה של הפלט. נשים לב שבתוך המתודה ישנה לולאת Foreach שבה אנו קוראים שוב ל-Display כך שאם זה היה קובץ זה היה קופץ למחלקת עלה ומשתמש ב-Display שלה אחרת זה היה רקורסיה (קורא שוב ל-Display עצמה).

מתודה GetSize שגם בתוכה יש לולאת Foreach שרצה על הרשימה וסוכמת את הגדלים שהיא אוספת. גם כאן אם מדובר בקובץ זה יקפוץ ל-GetSize של עלה ואחרת זה יבצע רקורסיה.

כעת בנינו מתודה שבעצם יוצרת אובייקט מסוג Composite (וגם מקבלת אחד) כך שאחרי השימוש במתודה נוכל להשתמש ב-Display.

```
// This method gets all files and folders from the Composite parameter path
// and creates children recursively
static void PopulateFilesAndFolders(Composite c)
{

    // Get the folder data of the given composite
    DirectoryInfo directoryInfo = new DirectoryInfo(c.Name);

    // Get all the files from the folder and add them as leaves
    FileInfo[] listOfFiles = directoryInfo.GetFiles();
    foreach (FileInfo file in listOfFiles)
    {
        Leaf oneFile = new Leaf(file.FullName);
        c.Add(oneFile);
    }

    // Add all sub directories and add them as composite
    // Add all the sub directories files
    DirectoryInfo[] ListOfFolders = directoryInfo.GetDirectories();
    foreach (DirectoryInfo folder in ListOfFolders)
    {
        // add composite child
        Composite folderComposite = new Composite(folder.FullName);
        c.Add(folderComposite);

        PopulateFilesAndFolders(folderComposite); // go into the subfolder recursively
    }
}
```

<u>וכך המימוש ב-Main:</u>

```
Console.WriteLine("Composite Example.");
// Create the root for the base folder
Composite root = new Composite(@"C:\TEMP");

// Populate all children for the root
PopulateFilesAndFolders(root);

// Call display to show all folders, files and sizes
root.Display("");
```

שיעור 3

Prototype:

מטרתה היא האפשרות להעתקה ושכפול של אובייקטים.

נראה זאת מתוך דוגמא של מחלקה המייצגת נקודה ומחלקה שמייצגת קו.

על פניו נראה שאין כאן בעיה, הרי אפשר פשוט להשוות בין שאני אובייקטים ויצרנו אחד חדש, לא? אז לא..

אז בדוגמא שלנו קו מיוצר מכמה נקודות ועכשיו נראה את זה בפועל בתוך הקוד.

נגדיר ראשית נקודה במרחב שפועלת בעצם כמו מיקום ב-GPS לפי שיעור x ושיעור y:

```
public class Point
{
    public float X { get; set; }
    public float Y { get; set; }

    public Point(float x, float y)
    {
        X = x;
        Y = y;
    }
}
```

כעת אם היינו רוצים לשכפל בצורה הבאה:

```
Point p1 = new Point(4.0f, 5.1f);

Point p2 = new Point(p1.X, p1.Y);
```

אז באמת שכפלנו וזה נראה לא רע, אבל אם היינו צריכים לבנות משהו שמצריך הרבה משתנים ולא רק שניים זה כבר היה נראה מכוער ולא נוח. לכן אנו יוצרים מתודה המיועדת לתפקיד הזה בלבד.

קודם ניצור Interface שיחזיק מתודה בשם Clone, כלומר שכפול (או יותר נכון שיבוט).

וכמובן נממש את ה-Interface הזה במחלקת נקודה ונשדרג אותה מעט:

```
public interface Proto<T>
{
    T Clone();
}

class Point : Proto<Point>
{
    public float X { get; set; }
    public float Y { get; set; }

    public Point(float x, float y)
    {
        X = x;
        Y = y;
    }

    public Point Clone()
    {
        // create a copy of this object
        Point clone = new Point(this.X, this.Y);

        // return the clone
        return clone;
    }
}
```

ואז השכפול יתבצע בצורה הבאה:

```
Point_1 p2 = new Point_1(4.0f, 5.1f);

Point_1 p2clone = p2.Clone();
```

כעת נראה את המחלקה של השורה:

```
class Line : Proto<Line>
{
    public Point P1 { get; set; }
    public Point P2 { get; set; }

    public Line Clone()
    {
        // create a copy of this object
        Line clone = new Line(this.P1, this.P2);

        // return the clone
        return clone;
    }

    public Line(Point p1, Point p2)
    {
        P1 = p1;
        P2 = p2;
    }
}
```

כאן אנו רואים בעייתיות מסוימת, בגלל שאנו קוראים ל-P1 ו-P2 בצורה של reference, אז אם נשתמש בשתי שורות שונות אז תיווצר בעיה, נניח במימוש הבא:

```
Point p4 = new Point(4.0f, 5.1f);
Point p5 = new Point(9.18f, 3.04f);

Line l1 = new Line(p4, p5);
Line l2 = l1.Clone();

l1.P1.X = 4.5f;
```

בעצם בגלל שהנקודות הוגדרו כ-ReferenceType, כל שינוי שיתבצע בשורה אחת יתבצע אוטומטית גם בשורה השנייה!

הפתרון לבעיה הזו הוא פשוט.

אנו פשוט צריכים לבצע את השכפול על הנקודות עצמן בתוך המתודה של שכפול השורה.

כלומר:

```
public Line Clone()
{
    // create a copy of this object
    Line clone = new Line(this.P1.Clone(), this.P2.Clone());

    // return the clone
    return clone;
}
```

עדיין, זה פתרון יחסית מסורבל כי עדיין אצטרך לשנות את כל האובייקטים בדרך לשימוש.

יש העתקה של משתנים שנקראת העתקת צל (Shallow Copy) שזו העתקה כמו של משתנים פרימיטיביים, שזה בעצם העתקה רק ברובד העליון (של Reference Type).

אבל ההעתקה השנייה שנקראת העתקה עמוקה (Deep Copy) שזו העתקה בכל הרמות ולא רק ברמה השטחית.

אז לצורך ההדגמה, יש לנו שני מונחים שנקראים Serialization ו-Deserialization.

Serialization – להפוך משהו לייצוג שלו ב-string (כמו ב-JSON).

Deserialization – פעולה הפוכה.

עכשיו, למה הזכרנו את הנושא הזה?

כי בעצם כאשר אני מבצע סריאליזציה ואז דיסריאליזציה נוצר לי אובייקט חדש (כלומר אובייקט חדש מוחזר) וכך בעצם אני יכול לבצע שכפול מבלי ליצור מתודה Clone().

```
public static T DeepCopy<T>(T self)
{
    var json = System.Text.Json.JsonSerializer.Serialize(self);
    T obj = System.Text.Json.JsonSerializer.Deserialize<T>(json);
    return obj;
}
```

אז בנינו מתודה שמקבלת משתנה גנרי (וגם מחזירה אותו בחזרה).

מה שקורה בתוך המתודה זה פשוט סריאליזציה ודיסריאליזציה וכך מוחזר לנו אובייקט חדש משוכפל לגמרי בכל הרמות של הפרמטר שנתנו לפונקציה.

אזהרה!

יש חיסרון לשיטה הזו בשונה מהתבנית שלנו (Protoype).

השימוש בסריאליזציה ודיסריאליזציה דורש הרבה יותר משאבים מאשר ליצור Clone().

אם האובייקטים קטנים או שהתוכנה לא דורשת משהו קריטי מאוד אז באמת לא מפריע לי ועדיף ללכת על השיטה הפשוטה של סריאליזציה ודיסריאליזציה. אבל אם אני כותב משהו שחשוב שיעבוד במהירות ממש וכל רגע חשוב אז לא נשתמש בזה אלא בתבנית שלנו.

- עכשיו יש לנו אפשרות לעשות סריאליזציה ודיסריאליזציה לבינארי – שזה אומר שהאובייקט שלי יהפוך לרצף של אפסים ואחדים ואז זה יתבצע על המצב הזה וזה כן נחשב מהיר.
  אבל גם כאן יש חיסרון למתכנת, כדי להשתמש בזה ב-C# המחלקה חייבת להיות עם ה-Attribute שנקרא [Serializable].

אזהרה 2!

השפות מתפתחות עם הזמן ודברים שהשתמשו בהם בעבר, תכונות מסוימות, קיימים עדיין בשפה אבל לא בהכרח נרצה להשתמש בהם.

יש שני Interfaces קיימים שנקראים ICloneable ועוד אחד של MemberWizeClone שאם נשתמש במתודות שלהם אז נקבל בחזרה משתנה מסוג object שנצטרך להמיר אותו בחזרה וזה גם דורש המון משאבים.

האפשרויות האלו ישנות יותר ולכן לא היה בהם את האופציה לאפשרות גנרית.

:Builder

תבנית שימושית מאוד, והיא מיועדת גם כן לבניית אובייקט.

השימוש הוא ליצירת אובייקט שכל יצירה שלו הוא בצורה אחרת כמו שאני מכין פיצה אבל יכול להכין עם תוספות שונות, רוטב שונה וכו'..

נשתמש בדוגמא של בניית בית:

לבית יש רצפה, קירות, גג וכו'.. אבל גם החומרים משתנים, אני יכול לבנות מבטון, מעץ או מקרח.

נניח שאנו יוצרים מחלקה של בית שבה ארבעה משתנים שהם החלקים של הבית ולכל אחד מהם נבנה מתודה seter:

```
public class House
{
    private string basement;
    private string structure;
    private string roof;
    private string interior;

    public void SetBasement(string basement)
    {
        this.basement = basement;
    }

    public void SetStructure(string structure)
    {
        this.structure = structure;
    }

    public void SetRoof(string roof)
    {
        this.roof = roof;
    }

    public void SetInterior(string interior)
    {
        this.interior = interior;
    }
}
```

כעת ניצור מחלקה אבסטרקטית לבניית בית, היא אבסטרקטית מכיוון שאחר נרצה לירש אותה לצורך בניות שונות של בתים.

```
public abstract class HouseBuilder
{
    protected House house;

    public HouseBuilder()
    {
        house = new House();
    }

    public abstract void BuildBasement();

    public abstract void BuildStructure();

    public abstract void BuildInterior();
```

```
        public abstract void BuildRoof();

        public void ConstructHouse()
        {
            BuildBasement();
            BuildStructure();
            BuildInterior();
            BuildRoof();
        }

        public House GetHouse()
        {
            return house;
        }
    }
```

את המחלקה הזו נירש בשתי מחלקות `WoodHouseBuilder` ו-`IglooHouseBuilder` שמייצגות בעצם בניית בית מעץ ובניית בית מקרח (איגלו).

<u>וב-`Main` נראה את השימוש בצורה הבאה:</u>

```
            Console.WriteLine("Builder Example.");
            // Builder  - take the "object creation" logic - outside,
            // So it enable to build customized objects.
            HouseBuilder iglooBuilder = new IglooHouseBuilder();
            iglooBuilder.ConstructHouse();
            House house = iglooBuilder.GetHouse();
            Console.WriteLine("House is ready: " + house);
```

בעצם השתמשנו ב-`Builder` כדי להפעיל את המתודה ואז השתמשנו במופעים לפי ירושות.

יש תבנית דומה שנקראת **`Fluent Builder`**:

בתוכנית הזו ה-`Build` בונה מופע ישירות ולא דרך שרשור מתודות כמו שראינו קודם.

<u>אז לצורך הדוגמא בנינו מחלקה של המבורגר:</u>

```
    public class Burger
    {
        public int NumPatties { get; set; }
        public bool Cheese { get; set; }
        public bool Bacon { get; set; }
        public bool Pickles { get; set; }
        public bool Lettuce { get; set; }
        public bool Tomato { get; set; }
        public Burger(int numPatties = 1)
        {
            NumPatties = numPatties;
        }
        public Burger(bool cheese, bool bacon, bool pickles,
                      bool lettuce, bool tomato, int numPatties = 1)
        {
            Cheese = cheese;
            Bacon = bacon;
            Pickles = pickles;
            lettuce = lettuce;
            Tomato = tomato;
            NumPatties = numPatties;
        }
    }
```

ואז בנינו מחלקת Builder:

```
public class BurgerBuilder
{
    private Burger _burger = new Burger();
    public Burger Build() => _burger;
    public BurgerBuilder WithPatties(int num)
    {
        _burger.NumPatties = num;
        return this;
    }
    public BurgerBuilder WithCheese()
    {
        _burger.Cheese = true;
        return this;
    }
    public BurgerBuilder WithBacon()
    {
        _burger.Bacon = true;
        return this;
    }
    //(...) and more another methods
    // for choosing addings to my burger
}
```

כעת המימוש ב-Main:

```
        // Fluent builder.
        // this is possible but ... not so beautiful
        Burger uglyBurger = new Burger(true, true, false, false, false);

        // This is BETTER
        // This is BETTER
        // This is BETTER
        BurgerBuilder burgerBuilder = new BurgerBuilder();
        Burger awesomeburger = burgerBuilder.WithCheese()
                                            .WithBacon()
                                            .Build();
```

אכן בניית ההמבורגר יותר יפה בתצורה של התבנית.

Template:

הרעיון הוא שיש לנו מחלקה שיש לה כמה יורשים ויש תבנית מסוימת עם אפשרויות מימוש שונות.

נניח ששני עובדים רוצים להתקבל לחברת הייטק, אחד לפיתוח והשני לבדיקת תוכנה.

כאשר הם יגיעו לתהליך קבלה שניהם יעברו את אותו התהליך של הגשת קורות חיים, קבלת שיחת טלפון, מעבר מבחן והגעה לראיון, אבל, כל אחד יעבור מבחן שונה, יגיש קורות חיים שונים, ויעבור ריאיון אצל מישהו אחר.

- דגש לא קשור לנושא – קבועים ו-enums הם קריאים יותר ומבינים דרכם את המשמעות יותר בקלות.

אז ניצור enum של לרכזת משאבי אנוש (HR) עם אפשרויות של עבר או נכשל.

נזכיר ש-enum זה מעין סט קבוע של דברים שנוכל להשתמש בהם אחר כך בהתאם למצב מסוים.

האיברים שבתוך ה-enum נשמרים לפי הסדר לפי מספרים החל מ-0 אלא אם כן מספרנו בעצמנו.

כמובן שהשימוש ב-enum מסייע לנו עם טעויות פוטנציאליות שיכולות לקרות כאשר אני משתמש ב-string.

נמשיך – ניצור גם מחלקה אבסטרקטית של ריאיון ובתוכה נגדיר מתודה בוליאנית של הריאיון הטלפוני, מתודה int לציון ריאיון טכני, ומתודה נוספת int לציון מבחן – כל אלו אבסטרקטיות.

לאחר מכן יש את הריאיון של ה-HR שכולם מגיעים אליו שזו מתודה שמחזירה מה-enum שיצרנו אם עבר או לא.

בסוף מתודה נוספת שהיא ביצוע הריאיון עצמו שכולל מעבר על כלל השלבים.

```
public enum HRResult
{
    Passed,
    Failed
}

public abstract class Interview
{
    public abstract bool InitialInterview();
    public abstract int TechnicalQuiz();
    public abstract int Test();

    public HRResult HRInterview()
    {
        Console.WriteLine("Please fill you name, address, marital status...");
        return HRResult.Passed;
    }

    public bool PerformInterview()
    {
        if (InitialInterview() == false)
            return false;

        if (TechnicalQuiz() < 6)
            return false;

        if (Test() < 6)
            return false;

        if (HRInterview() == HRResult.Failed)
            return false;

        return true;
    }
}
```

כעת נרצה לממש את המחלקה הזו לשני תפקידים שונים כמו שאמרנו קודם: מפתח ובודק תוכנה.

כל אחד מהתפקידים יקבל מחלקה משלו שתירש את המחלקה האבסטרקטית שלנו וימש את המתודות בצורה המתאימה לו.

נזכיר שעצם זה שהוא יורש את המחלקה הוא יוכל לגשת למתודות שאינן אבסטרקטיות.

ואם נשתמש במילים שתואמות לנושא שלנו, אז המחלקות הללו יורשות מה-Template של ריאיון (השתמשנו במילה Template במקום מחלקה אבסטרקטית)

בעמוד הבא נראה את שתי המחלקות.

```
public class DevInterview : Interview
{
    public override bool InitialInterview()
    {
        string interviewer = "Dev Manager";
        Console.WriteLine("Hello, welcome to our company! My name is: " + interviewer);
        Console.WriteLine("Please tell us about yourself...");
        return true;
    }

    public override int TechnicalQuiz()
    {
        Console.WriteLine("What is binary search?");
        Console.WriteLine("Calculate the complexity of this algorithm...");
        Console.WriteLine("Implement a singleton");
        return new Random().Next(10);
    }

    public override int Test()
    {
        string test = "Software development text";
        Console.WriteLine("Complete the test. good luck: " + test);
        return new Random().Next(10);
    }
}

public class QAInterview : Interview
{
    public override bool InitialInterview()
    {
        string interviewer = "QA Manager";
        Console.WriteLine("Hello, welcome to our company! My name is: " + interviewer);
        Console.WriteLine("Please tell us about yourself...");
        return true;
    }

    public override int TechnicalQuiz()
    {
        Console.WriteLine("What is an automation?");
        Console.WriteLine("Write a test to check stability of a coin");
        Console.WriteLine("What issues can occur on a vending machine?");
        return new Random().Next(10);
    }

    public override int Test()
    {
        string test = "QA test";
        Console.WriteLine("Complete the test. good luck: " + test);
        return new Random().Next(10);
    }
}
```

נשים לב שהשתמשנו במימוש שהוא דוגמא בלבד ואינו מימוש אמיתי של המתודות הללו.

```
// it is SIMPLE to create TEMPLATES....
// nothing more than it - just template to some process...

Interview interview = new DevInterview();

if (interview.PerformInterview() == true)
    Console.WriteLine("The candidate was hired for the job!");
else
    Console.WriteLine("The candidate was rejected");
```

כאן אנו רואים דוגמא למימוש מתוך ה-Main שתעבוד בהתאם לנתונים שיתקבלו (לשם הדוגמא בלבד ולשם השינויים המתודות מחזירות משתנים רנדומליים).

Mediator (מתווך):

אם נניח יש לנו נמל תעופה, אז אף אחד לא יכול לנחות או להמריא ללא אישור ממגדל הפיקוח.
אמנם מגדל הפיקוח לא מבצע בעצמו את ההמראה או הנחיתה אבל עדיין זה לא יקרה בלי אישורם.
יש את הרצון לפעולה ויש את הפעולה עצמה, המעבר תלוי באישור המעבר.
בתבנית הזו משתמשים כאשר אנו רוצים ליצור הפרדה ברורה בין הבקשה לביצוע לבין הביצוע.
לדוגמא, משהו שקיים (לא לומדים את זה בקורס) זה ניהול תורים של פעולות שממתינו לביצוע, כמו עיבוד תמונות שהתמונות עוברות עיבוד אחת אחת ואז מקבלים את סיום הפעולה לאחר זמן מסוים.
במקרה שכזה אנחנו מכניסים תמונה, שולחים אותה לתור אבל העיבוד עוד לא התבצע, אחר כך העבודה נכנסת לעיבוד והפעולה מתבצעת.
דוגמא נוספת היא צ'אט, שם כל אחד יכול לשלוח לכל אחד הודעה והשרת המתווך יודע לגרום להודעה להישלח ולהגיע לכל אחד.
נדגיש שזו אחלה דוגמא כדי להמחיש קוד אבל בפועל דברים מהסוג הזה לא באמת עובדים כך.
אז ניצור Interface של user, שם תהיינה מתודות של שליחה וקבלה.
את ה-Interface נממש במחלקת user, בנוסף ניצור גם מופע של ה-Interface של Mediator.
כמו כן ה-Interface של IChatMediator, בו יש מתודת הרשמה למשתמש שמקבל user, ומתודת שליחת הודעה שמקבלת string ו-user ששולחת לכולם חוץ מלשולח. וכמובן את מחלקת ChatMediator.

```
public interface IUser
{
    void Send(string message);
    void Receive(string message);
}
public class User : IUser
{
    private IChatMediator mediator;
    private string name;

    public User(IChatMediator mediator, string name)
    {
        this.mediator = mediator;
        this.name = name;
    }

    public void Receive(string message)
    {
        Console.WriteLine(this.name + ": Received Message:" + message);
    }

    public void Send(string message)
    {
        Console.WriteLine(this.name + ": Sending Message: " + message + "\n");
        mediator.SendMessage(message, this);
    }
}

public interface IChatMediator
{
    void SendMessage(string msg, IUser user);
    void RegisterUser(IUser user);
}
```

יש לנו רשימה שמחזיקה את כל המשתמשים כך שהמתודה RegisterUser מוסיפה משתמש לרשימה והשליחה שולחת את ההודעה לכל איברי הרשימה (המשתמשים) למעט השולח.

```
public class ChatMediator : IChatMediator
{
    private List<IUser> usersList = new List<IUser>();

    public void RegisterUser(IUser user)
    {
        usersList.Add(user);
    }

    public void SendMessage(string message, IUser user)
    {
        foreach (IUser u in usersList)
        {
            // message should not be received by the user sending it.
            if (u != user)
            {
                u.Receive(message);
            }
        }
    }
}
```

המימוש ב-Main ייראה כך:

```
IChatMediator mediator = new ChatMediator();

IUser moshe = new User(mediator, "Moshe");
IUser haim = new User(mediator, "Haim");
IUser ronen = new User(mediator, "Ronen");

mediator.RegisterUser(moshe);
mediator.RegisterUser(haim);
mediator.RegisterUser(ronen);

moshe.Send("Hello everyone!");
haim.Send("What's up?");
```

נזכיר שהדוגמא כאן היא פחות מציאותית וזה רק מדמה, המילים של לשלוח ולקבל בתבנית הם לא בהכרח לשליחה וקבלת הודעות אלא ממש על הרשאות פעולה.

- ברדוגו טיפ

לרוב נעדיף ליצור מופע מהצורה האבסטרקטית שלו ולא מהמחלקה עצמה כדי שבעתיד (תיאורטית) נוכל להחליף את המחלקה למימוש אחר – בדוגמא שלנו יצרנו צ'אט על ה-Interface שלו ולא על ידי המחלקה עצמה.