הנדסת תוכנה – פיתוח מונחה עצמים

Singleton pattern – תבנית סינגלטון

מוטיבציה

לפעמים בפיתוח מערכת או תוכנית מסוימת יש חשיבות להגביל כמות מופעים של מחלקה מסוימת למופע יחיד. דוגמאות מערכתיות יכולות להיות מנהל חלונות יחיד, גישה לקובץ או מנהל קבצים בכלל, מנהל תור למדפסת. בדרך כלל התבנית הזאת חשובה כשמדובר בניהול מרוכז של משאבים (חיצוניים או פנימיים) או נקודת גישה גלובלית למשאב כלשהו.

לפעמים מרחיבים את המושג הזה לניהול כמות מוגבלת ומוגדרת של משאבים מסוג כלשהו – אנחנו לא נתעמק בכיוון הזה.

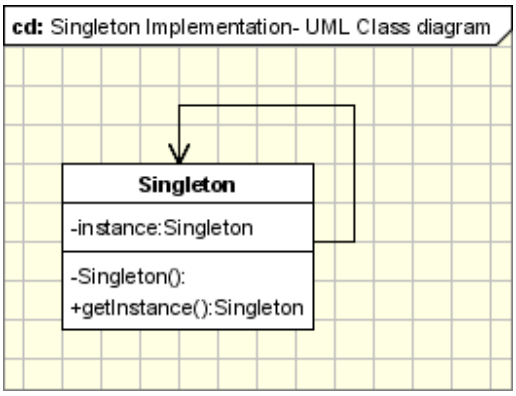
התבנית הזאת נהית שימושית מאד כאשר משלבים אותה עם תפנית פקטורי (Factory) שנלמד בהמשך.

במקרה שלנו של מודל שלושת השכבות – נרצה מופע יחיד לשכבה – קודם כל כדי להגביל שכבת DL שתהיה נקודת גישה יחידה לשמירת הנתונים.

דרישות יישום

כל יישום חייב למנוע כל אפשרות ליצירת מופע נוסף (למופע היחיד). חייבים לתת אפשרות השגת הפניה למופע היחיד בצורה פשוטה וברורה.

תרשים המחלקה של היישום הפשוט עבור סינגלטון:



נכיר קצת את ספת התרשימים של UML עבור תרשים מחלקות:

- החץ בצורה שבתרשים מראה יחס של שימוש של מחלקה אחת במחלקה אחרת. במקרה הזה המחלקה משתמשת בעצמה. זאת אומרת – במחלקה Singleton יש שדה מסוג Singleton או פונקציה שמקבלת פרמטר או מחזירה ערך מסוג Singleton.
- המלבן של המחלקה מחולק לשלושה קטעים:

- הקטע העליון מכיל שם המחלקה (ויכול גם להכיל תיאור של תכונות מיוחדות שלה). שם המחלקה יתחיל מאות גדולה.
- הקטע האמצעי מכיל את השדות והמאפיינים (Properties)
- הקטע התחתון מכיל את הפונקציות (methods) של המחלקה

- צורת הרישום של שדה ושל פונקציה מכיל שלושה חלקים:
  - סימן שמסמן הרשאת גישה לשדה public (+) או private (-) או protected (#)
  - השם (ע"פ הכללים שאתם כבר מכירים)
  - אם שדה\פונקציה סטטי(ת) – יופיע קו תחת השם
  - במקרה של פונקציה – רשימת פרמטרים בסוגריים
  - סימן ':' ולאחריו הסוג של השדה או הערך המוחזר מפונקציה
  - בשביל שדה – סימן '=' וערך אתחול (אופציונלי)

יישום ב-#C

בחזרה לתבנית הסינגלטון. החלק הבסיסי של התבנית הינו שדה סטטי – בדרך כלל בשם ***instance*** – (בתרשים חסר קו תחתון) שיכיל את המופע היחיד (השדה חייב להיות עם הרשאה מוגבלת [private] למניעת גישה אליו מבחוץ), הבנאי של המחלקה – גם חייב להיות עם הרשאה מוגבלת [private] כדי למנוע יצירת מופעים מבחוץ. כמו כן פונקציה סטטית – בדרך כלל בשם ***getInstance*** – עם הרשאת גישה ציבורית (public) כדי לקבל הפנייה למופע של סינגלטון. ב-#C נוכל להשתמש בגטר (getter) של property עבור השדה (***Instance***) במקום פונקציה מיוחדת.

נ.ב. ברוב המקרים נגדיר את המחלקה של סינגלטון בצורה שלא ניתן לרשת ממנה (ב-#C משתמשים במילת מפתח ***sealed*** בהגדרת המחלקה).

דוגמה ליישום רגיל של מחלקה כסינגלטון (הדוגמה הינה בטוחה בסביבה מרובת תהליכונים אך ללא נעילות):

```
// Thread-safe singleton example without using locks
public sealed class MySingleton
{
    private static readonly MySingleton instance = new MySingleton();

    // Explicit static constructor to tell C# compiler
    // not to mark type as beforefieldinit
    static MySingleton() {}

    private MySingleton() {}

    // The public Instance property to use
    public static MySingleton Instance { get { return instance; } }

    // Implementation specific data members and methods...
    ......
}
```

על בסיס זה דוגמה של מחלקת DL של שלוש שכבות:

```
public sealed class DL_List : IDL
{
    private static readonly MySingleton instance = new DL_List();
```

```
    // Explicit static constructor to tell C# compiler
    // not to mark type as beforefieldinit
    static DL_List() {}

    // Explicit private constructor to avoid direct usage of new DL_List outside the class
    DL_List() {}

    // The public Instance property to use
    public static DL_List Instance { get { return instance; } }

    private List<Student> studentList = new List<Student>(); // better do it by var initializer
    private List<Course> courseList = new List<Course>();    // better do it by var initializer

    public bool AddStudent(Student student)
    {
       ......
}
```

אפשר להשתמש ב-Template שבניתי לצורך יצירת הסינגלטון (הדוגמה קצת מורכבת ומשתמשת ב-Reflection ונעילות עבור ריצה בטוחה בסביבה מרובת תהליכונים) כדלקמן.

Reflection – הינו שיטה מתוחכמת ומתקדמת של תכנות בכל השפות מונחות עצמים. קצת נגענו בזיטה הזאת בדוגמה של הדפסה אוניברסלית של תוכן השדות. בגדול, מושג ה-Reflection הינו יכולת תוכנה לבחון אובייקטים ומחלקות ו\או לשנות את התנהגותם בזמן ריצה – וכל זה ברמה של "הסתכלות פנימה" על הסוגים של השפה – זאת אומרת על המבנה הפנימי של המחלקות מבחינת מערכת NET.. השיטה הזאת מתוחכמת ונותנת כלים לגשת למידע פנימי על המחלקות, שדות שלהן, פונקציות שלהן וכו' ואפילו לשנות אותם. השימוש בשיטה הזאת יכול להיות מועיל מאד מחד גיסא אך מסוכן מאד ומוביל לתוצאות הרסניות מאידך גיסא.

הדוגמה הזאת של סינגלטון איננה חובה ומיועדת לסטודנטים מתקדמים המעוניינים ללמוד את הנושא יותר לעומק. נ.ב. הדוגמה נבנתה ונבדקה ע"י דן זילברשטיין – היא מתבססת על הניסיון העולמי בתחום אך איננה העתקה מספריה או יישום קיים.

```
[Serializable]
public class SingletonException : Exception
{
    public SingletonException(string text) : base(text) { }
    public SingletonException(string text, Exception e) : base(text, e) { }
    protected SingletonException(SerializationInfo info, StreamingContext context)
                                                                        : base(info, context) { }
}
```

```
public abstract class Singleton<T> where T : Singleton<T>
{
   // Singleton pattern Step#2 - make the static and protected constructor
   static Singleton() {}
   protected Singleton () {}

   class Nested
   {
      // Singleton Step#3 - _instance is initialized to null
      internal static volatile T _instance = null;
      internal static readonly object _lock = new object(); // for multithreading
      static Nested() {}
   }

   // Singleton Step#4 - separate property ensures lazy class instatiation, no setter, getter only.
   public static T Instance
   {
      get
      {
         if (Nested._instance == null) // if it is already not null – no need for lock()
         {
            lock (Nested._lock) // for multithreading
            {
               if (Nested._instance == null) // double check
               {
                  // Get the type of the generic class and check whether it is sealed (Reflection)
                  Type t = typeof(T);
                  if (t == null || !t.IsSealed)
                     throw new SingletonException(string.Format("'{0}' must be a sealed class", t.Name));

                  // Get the instance constructor of the deriving class
                  // Ensure it to be non-public and parameterless
                  // (still using Reflection)
                  ConstructorInfo constr = null;
                  try
                  {
                     constr = t.GetConstructor(BindingFlags.Instance | BindingFlags.NonPublic,
                                               null, Type.EmptyTypes, null);
                  }
                  catch (Exception ex)
                  {
                     throw new SingletonException(string.Format("A private/protected constructor " +
                                                                "is missing for '{0}'.", t.Name), ex);
                  }
                   // Also exclude internal or default constructors (still Reflection)
                  if (constr == null || constr.IsAssembly)
                     throw new SingletonException(string.Format("A private/protected constructor " +
                                                                "is missing for '{0}'.", t.Name));

                  // Create new instance and invoke its constructor by Reflection
                  Nested._instance = (T)constr.Invoke(null);
               }
            }
         }
         return Nested.__instance;
      }
   }
}
```

והשימוש בשכבת ה-DL עבור הדוגמה של סטודנטים וקורסים כדלקמן:

```
// C# Singleton pattern Step#1 recommends using "sealed" class to avoid derivated violation
sealed class DL_List : SingletonThreaded<DL_List>, IDL
{
    // Singleton pattern Step#2 - make one and only one paramterless constructor to be private
    DL_List() { }

    List<Student> studentList = new List<Student>(); // better do it by var initializer
    List<Course> courseList = new List<Course>();    // better do it by var initializer

    public bool AddStudent(Student student)
    {
       ......
}
```

Factory Method pattern – תבנית פקטורי מתוד (מפעל)

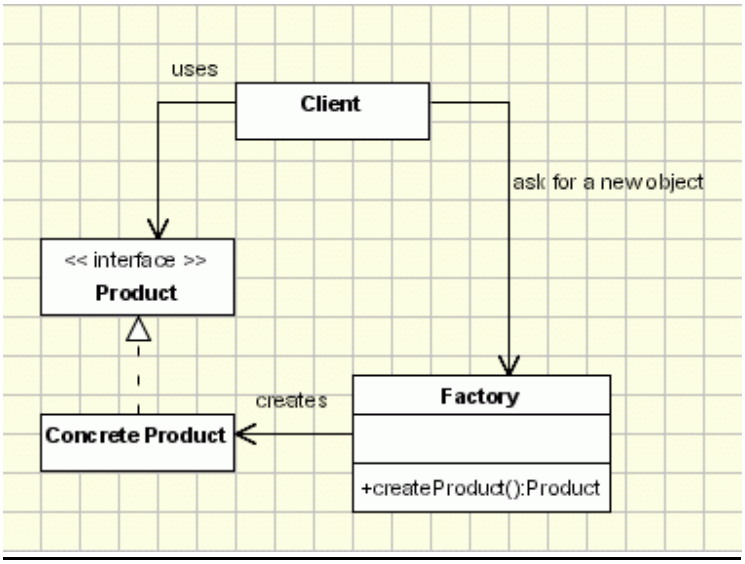
מוטיבציה

תבנית המפעל הינה אחת התבניות הנפוצות והשימושיות בפיתוח מונחה עצמים. תפקידו (כפי שראינו כשלמדנו את המודל של שלוש שכבות) לנתק את היישום של שרות מהלקוחות שלו (**clients**). הצורות המלאות והנפוצות שלו תמצאו תחת שמות **Factory Method Pattern** כמו כן **Abstract Factory Pattern**. אנחנו נתמקד צורה פשוטה ובסיסית יותר שדרושה לנו לצורך הפרוייקט ושהודגמה כבר במודל שלושת השכבות ונרחיב קצת על התבנית של **Simple Factory** שהיא גרסה מצומצמת של Fatory Method.

דרישות יישום

- חוסמת את היישום ואת בניית המופע של השרות (**Product** בתרשים) מהלקוחות.
- ההתייחסות לשרות דרך האיטרפייס שלו.

תרשים המחלקה של היישום הפשוט עבור מפעל:



אלמנטים חדשים בתרשים: חץ מקוקו עם סוף "משולש" מסמל ירושה מאינטרפייס. בחצים של שימוש ניתן לרשום במפורש צורת השימוש. במודל של שלושת השכבות השתמשנו:

- **IBL** – האינטרפייס "***Product***"
- **MyBL** – היישום שלנו של BL – "***Concrete Product***" – היורש מ-**IBL**
- **UI** או **PL** – הלקוח שלנו – "***Client***". נ.ב. אמנם בדוגמה שלנו יש רק לקוח אחד אך בפועל יכולים להיות מספר לקוחות לשרות
- **FactoryBL** – המפעל שלנו – "Factory"
- **getBL()** – "***createProduct()***"

יישום ב-C#

שימו לב שבדוגמת שלושת השכבות שלנו הפונקציה **getBL()** הינה פונקציה סטטית שחוסכת מאתנו יצירת מופע של המפעל. הדבר פשוט ומתאים לנו בסביבה של שרות בודד ולקוח בודד לשרות, אך הצורה הזאת איננה נכונה ועלולה לגרום לבעיות ביישומים. לצערנו נמצא ברשת הרבה דוגמאות של יישום ה-C# לתבנית הזאת ע"י מחלקת מפעל סטטית. הצורה הנכונה כדלקמן (נשתנש הפעם בדוגמה עבור **DL**).

האינטרפייס של השרות שלנו **IDL**. יש לנו מספר יישומים של השרות ב-**DL**. כפי שנראה במפעל. הלקוח שלנו – הינו **MyBL**. מחלקת המפעל של **DL** כדלקמן:

```
public class FactoryDL
{
    public IDL getDL(string typeDL)
    {
        switch (typeDL)
        {
            case "List":
                // Singleton Pattern - use Instance
                return DL_List.Instance;
            case "XML":
                // Singleton Pattern - use Instance
                return DL_XML.Instance;
            default:
                return null;
        }
    }
}
```

אנחנו גם בדרך כלל נזרוק חריגה מתאימה ב-**default** במקום החזרת סתם **null**.

השורה של קבלת גישה לשרות **DL** במחלקת ה-**MyBL** כדלקמן:

```
private DAL.IDL dl = new FactoryDL().getDL("List");
```

צפוי שחלקכם תזדעקו עכשיו – יש תלות בשרות ספציפי בקוד של הלקוח! זה נכון – אז זה המבנה הפורמלי של התבנית. כמו כן התלות איננה ישרה – אנחנו לא ניגשים בקוד של הלקוח ביישום של השרות ישירות אלא עדיין רק דרך האינטרפייס.

אפשר להישאר במסגרת התבנית ולבטל את התלות לגמרי ע"י כך שסוג השרות יישמר בקובץ קונפיגורציה של התוכנית שלנו ופונקציית **getDL()** תחזור להיות ללא פרמטרים – היא תיגש לקובץ הקונפיגורציה בתיקח משם את הערך של המחרוזת המבטאת את השרות הנדרש.

ברוב המקרים נרצה שהמפעל שלנו יהיה סינגלטון – זה אחד השימושים הנפוצים של סינגלטון. לכן ניצור את המחלקה כדלקמן:

```
public sealed class FactoryDL : IDL
{
    private static readonly FactoryDL instance = new FactoryDL();
```

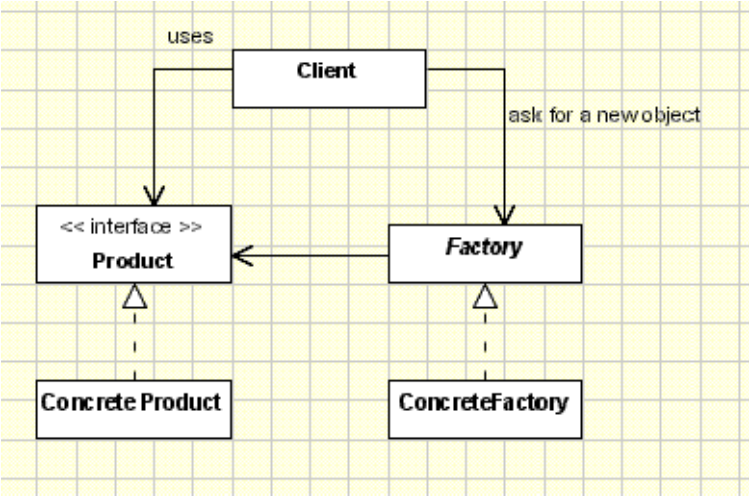
```
    static FactoryDL() {}
    private FactoryDL() {}
    public static FactoryDL Instance { get { return instance; } }

    public IDL getDL(string typeDL)
    {
        switch (typeDL)
        {
            case "List":
                // Singleton Pattern - use Instance
                return DL_List.Instance;
            case "XML":
                // Singleton Pattern - use Instance
                return DL_XML.Instance;
            default:
                return null;
        }
    }
}
```

ואז נכתוב במחלקת ה-**MyBL** כדלקמן:

```
private DAL.IDL dl =  FactoryDL.Instance.getDL("List");
```

למעשה – הצורה המלאה של התבנית הינה תבנית של Factory Method. התבנית הזאת מאפשר לנתק את הקשר בין המפעל לבין המוצרים הספציפיים. התרשים של התבנית כדלכמן:



הסביבה של NET. ושפה #C לא נותנת אפשרות להריץ קוד סטטי או בנאי סטטי בזמן של טעינת המחלקה. התבנית המלאה הנ"ל דורשת מהשירותים לבצע רישום מפורש של המפעלים שלהם לפני שכל לקוח ישתמש במפעל הכללי. המימוש המלא ב-#C דורש ידע מעמיק בסביבת NET. ושימוש מתוחכם (הרבה יותר מסינגלטון שהדגמתי לעיל) ב-Reflection. לא ניכנס ליותר עומק בנושא.

אמנם סקרנו את האפשרויות – אך מבחינת הפרויקט לא נדרש מכם תחכום יתר. מי שישתמש בצורות הנכונות יותר והמתקדמות יותר יזכה בנקודות בונוס, אך אפשר להשתמש בסינגלטון הפשוט והמפעל הסטטי ללא השפעה שלילית על ציון הפרויקט.