מודל השכבות

ארכיטקטורת שלושת השכבות

במסמך זה נראה כיצד לארגן את מבנה הפרויקטים שלנו כך שנוכל לבנות תשתית למערכת מבוזרת בפלטפורמת דוט-נט בהמשך כשנבנה פרויקט ונבצע בו שינויים נבין את החשיבות של החלוקה הנכונה עוד בשלב התשתית של פיתוח הפרויקט.

מחבר: אושרי כהן

תאריך יצירה: 09:36 27/11/2011

תאריך עדכון אחרון: 11:43 - 15/11/2012

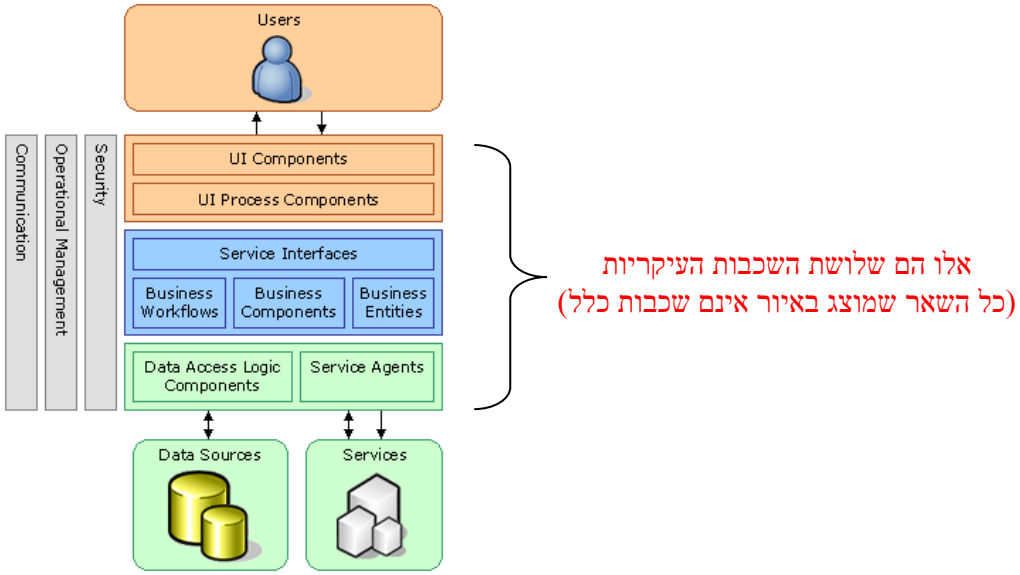
תוכן

מבוא

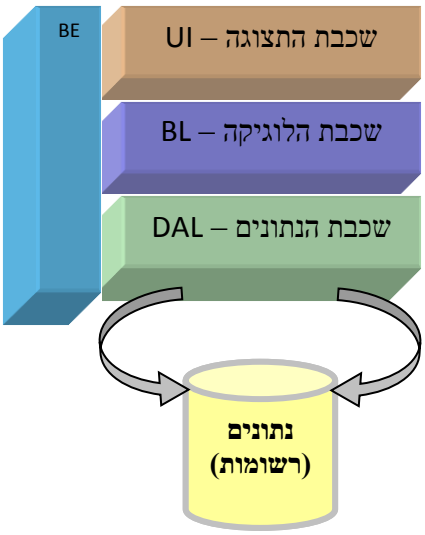
בתחילת דרכנו במדעי המחשב ,ולאחר שהתחלנו לרשום כמה פרוצדורות גילינו כי יעיל יותר לחלק קוד מסוים לפרוצדורות שונות ולמחלקות שונות ובכך הקוד יותר יעיל ,יותר מובן ויותר נתון לביקורת (ניתן לדעת באיזו פונקציה או באיזו מחלקה בדיוק יש בעיה).
כעת נתקדם יותר ונאמר כי ניתן לחלק את הפרויקט שלנו למספר פרויקטים קטנים שכולם יחד מרכיבים את הפרויקט כולו אבל כל חלק מהווה יחידה עצמאית של פרויקט בפני עצמו.

מודל השכבות:

באופן כללי המודל הרשמי (ע"פ מיקרוסופט) לחלוקה לשכבות מוצג באיור הבא:



מקובל לחלק פרויקטים עם מאגר נתונים (שיכול להיות קובץ,בסיס נתונים ,שירות) לשלוש שכבות:



שכבת התצוגה (Presentation Layer):
שכבה זו אחראית על התצוגה שהמשתמש רואה (יכול להיות דף אינטרנט ,תוכנה במחשב ,ישום בטלפון הנייד) ,שכבה זו היא היחידה שיודעת לאן מגיע הפלט\קלט בסופו של דבר מהפרויקט.
שכבה זו נקראת גם user interface ומכאן מקור השם UI (למרות שניתן גם להשתמש ב PL)

שכבת הלוגיקה (Business Layer):
שכבה זו אחראית על הלוגיקה שמאחורי הנתונים ,כלומר הופכת את הנתונים לברי משמעות ,אם לדוגמה הנתונים זה תוצאות של מדידות שונות אזי שכבה זו מפיקה תוצאה (שעדיין איננה ברת תצוגה) שיכולה להיות מה על המערכת לעשות.

שכבת הנתונים (Data Access Layer):
שכבה זו אחראית על לקיחת הנתונים מבסיס הנתונים ,שכבה זו היא השכבה היחידה בפרויקט שיודעת היכן נשמרים הנתונים (בקובץ טקסט או בבסיס נתונים או XML וכו...)

מה זה ה BE
BE זה המכיל אוסף של מחלקות המשמשות כמעין סבלים (נושאים את הנתונים)
הריי שכבת ה BL שולחת לשכבת ה UI איזשהו אובייקט (לדוגמה תחזית) כיצד מוגדר אותו אובייקט?
הטיפוס של אותו אובייקט מוגדר באוסף המחלקות של BE

איך זה עובד?

באופן כללי כל שכבה מהווה פרויקט בפני עצמו, כל שכבה מכירה רק את השכבה שמתחתיה, אף שכבה לא מכירה שכבה שמעליה.
במידת הצורך בטיפוסים מורכבים – כל שכבה תכיר את BE

דוגמה:
מערכת לחיזוי מזג האוויר:
שכבת UI משמשת כתצוגה שמציגה למשתמש את התחזית הצפויה ביום מהשבוע (ע"פ בחירת המשתמש)
שכבת BL מקבלת נתונים ומחשבת מה תהייה התחזית בעוד X ימים.
שכבת DAL יודעת לקרוא את הנתונים (כיוון הרוח, טמפרטורה, לחות, זווית כדור הארץ, עננות) ולהחזירם למי שמבקש.

המערכת פועלת כך:

- המשתמש בוחר יום נניח ע"י לחיצה על הכפתור, הפונקציה שפועלת עם לחיצת הכפתור קוראת לפונקציה שנמצאת ב BL שולחת לה את X ע"י זיהויי הכפתור.
- הפונקציה שנמצאת ב BL רוצה להפיק תשובה עבור הפונקציה שקראה לה ולכן משתמשת בפונקציה שנמצאת ב DAL על מנת לקרוא את הנתונים הספציפיים.
- שכבת ה DAL מקבלת הוראה משכבת ה BL (DAL עצמו לא מכיר את BL כמובן, הכול התחיל בגלל ש BL הפעיל פונקציה) ולכן קוראת את הנתונים מבסיס הנתונים ומעבירה אותם ל BL
- שכבת BL מבצעת את החישוב שהיה צריך לבצע ומחזירה את הנתונים לשכבת ה UI
- שכבת ה UI מקבלת את הנתונים ומציגה זאת למשתמש.

מבחינת הקוד:
רואים כי כל שכבה יוצרת קשר רק עם השכבה שתחתיה, כלומר UI מפעילה פונקציה שנמצאת ב BL ולא להיפך.
בספו של דבר BL מחזירה מידע ל UI אבל זאת מבלי ש BL תדע שהיא מחזירה את הנתונים דווקא ל UI (היא מחזירה פשוט את הנתונים למי שהפעיל את הפונקציה).

הערות:
המשתנה w שהגדרנו בקוד מסוג Weather – אנו מניחים כי קיימת מחלקה כזו והיא בעצם משמשת בתור ה BE שלנו במקרה הזה.

UI

```
public void ShowWeather(int day)
{
   Weather w =  bl.getWeather(DateTime.Now.Day - day);
   myShow(w);
}
```

BL

```
public Weather getWeather(int x)
{
    float x = dal.getTemperature();
    float y = dal.getWindDirection();
    // ....
    Weather w = someComputation(x,y,...);
    return w;
}
```

DAL

```
public float getWindDirection()
{
    // …
}
public float getTemperature()
{
    // …
}
```

נראה גם את הכיוון ההפוך בדוגמה:
נניח שעכשיו המשתמש רוצה להכניס נתון מסוים לבסיס נתונים (נניח כיוון הרוח)

המערכת פועלת כך:

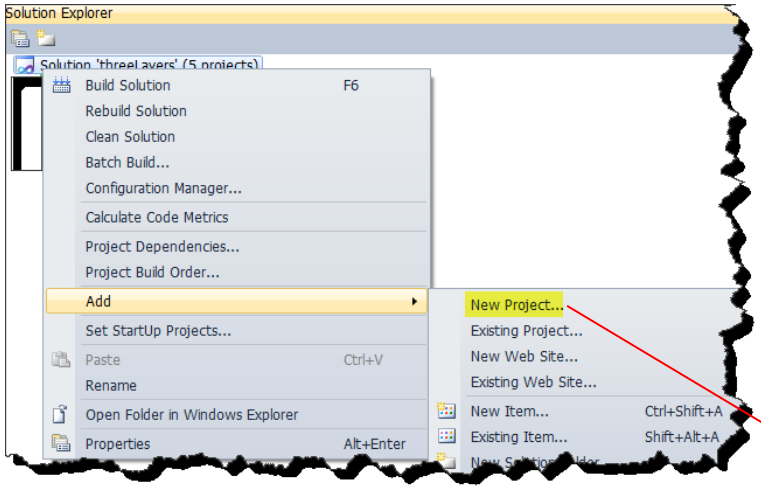
- המשתמש מזין נתונים נניח ע"י מילוי textBox ולחיצה על הכפתור, הפונקציה שפועלת עם לחיצת הכפתור קוראת לפונקציה שנמצאת ב BL שולחת לה את הנתון שנמצא ב textBox
- הפונקציה שנמצאת ב BL מפעילה פונקציה שנמצאת ב DAL על מנת להכניס את הנתון הספציפי
- שכבת ה DAL מקבלת הוראה משכבת ה BL (DAL עצמו לא מכיר את BL כמובן, הכול התחיל בגלל ש BL הפעיל פונקציה) ולכן מכניסה את הנתון לבסיס הנתונים.

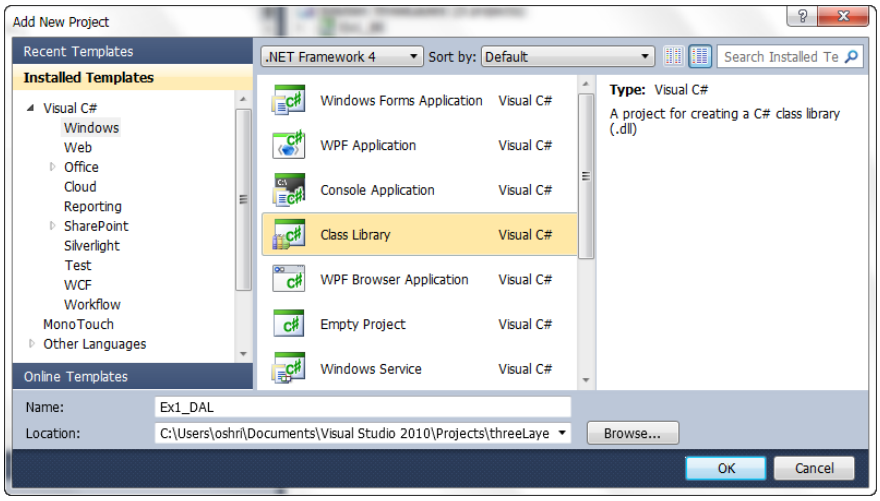
גם כאן כפי שהוסבר כל שכבה מכירה רק את השכבה שמתחתיה !

יצירת השכבות השונות ב visual studio 2010

בניית הפרויקטים שיהוו את שלושת השכבות

ב visual studio 2010 כול שכבה תמומש ע"י פרויקט עצמאי.
על מנת לעשות זאת יש ליצור ארבעה פרויקטים שונים ב solution
את שלושת הפרויקטים שמיצגים את BE, BL ו DAL נגדיר מסוג Class Library

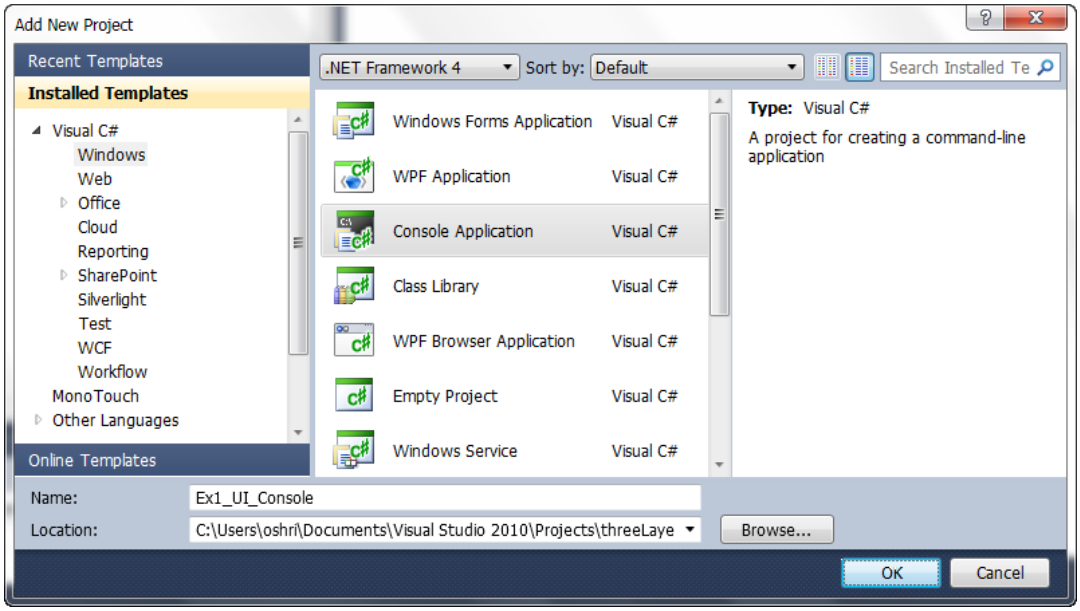




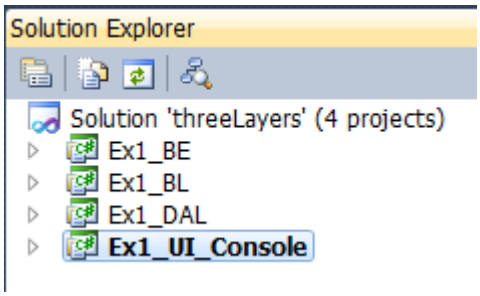
קצת על Class Library:

פרויקט מסוג Class Library זה פרויקט שאה שהוא מכיל זה רק אובייקטים ומחלקות שיכולים לשמש בפרויקטים אחרים.
אין לו מאשך למשתמש (לא ניתן להריץ אותו ישירות, אלא באמצעות פרויקט אחר שפונה אליו) ולכן זה מתאים לנו לשתי השכבות התחתונות ולשכבת ה BE.

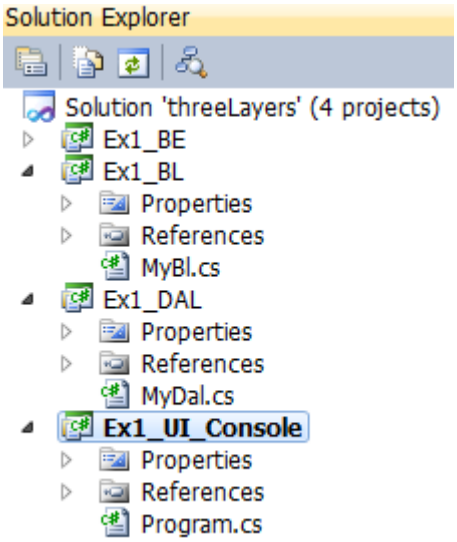
את שכבת ה UI נגדיר בהמשך כפרויקט מסוג Console application



כעת מצב הפרויקט הוא כזה:



בשכבות BL ו DAL נשנה את שם קובץ ה cs ונגדיר מחלקות בהתאם:



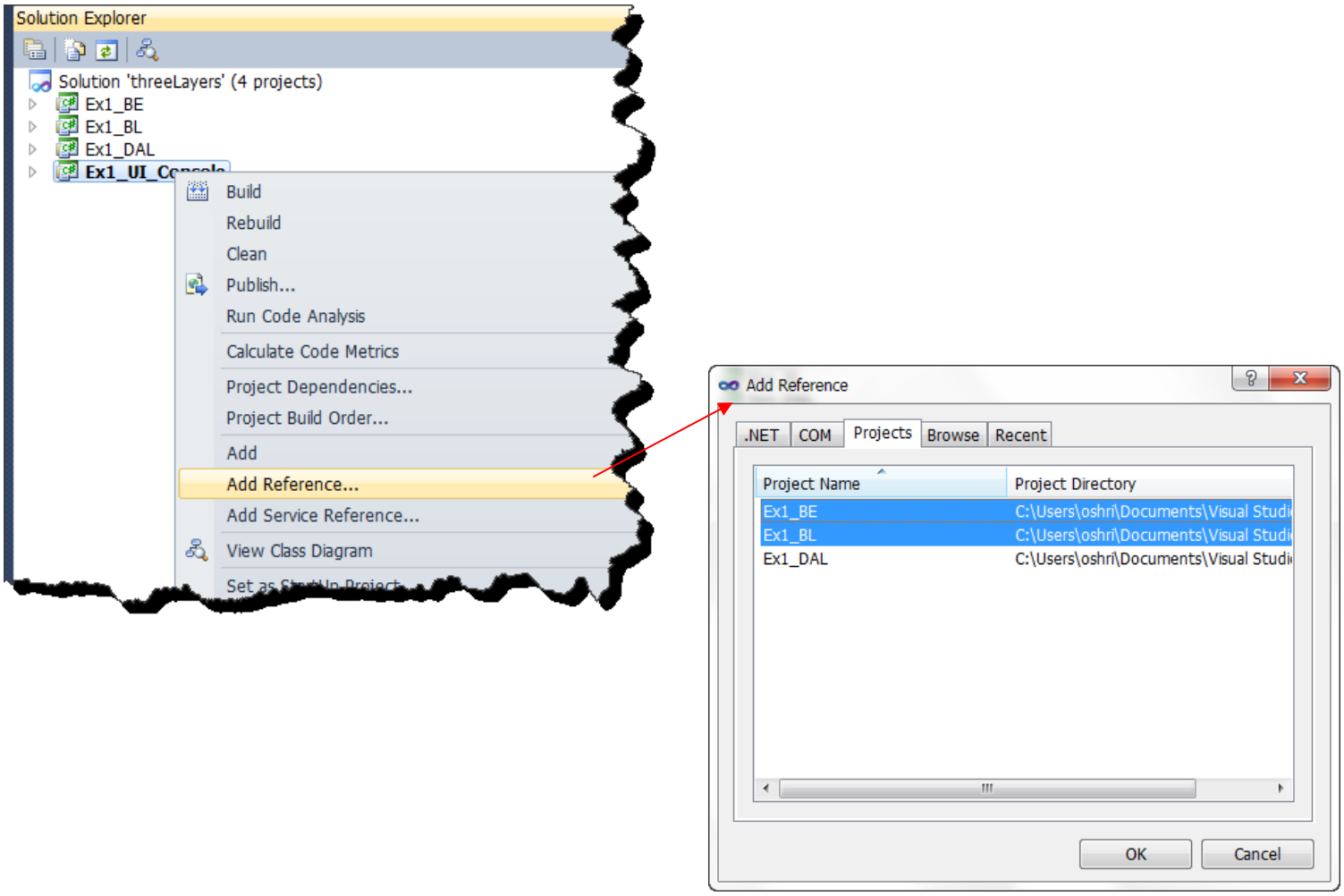
כלומר:
בפרויקט DAL במקום שם המחלקה והקובץ class1.cs שנוצרים כברירת מחדל, נגדיר קובץ ומחלקה בשם MyDal וכנ"ל גם עבור BL

יצירת קשרים References בין השכבות ב visual studio

כזכור שכבת UI צריכה להכיר את BL ושכבת BL צריכה להכיר את DAL.
וכולם מכירים את BE

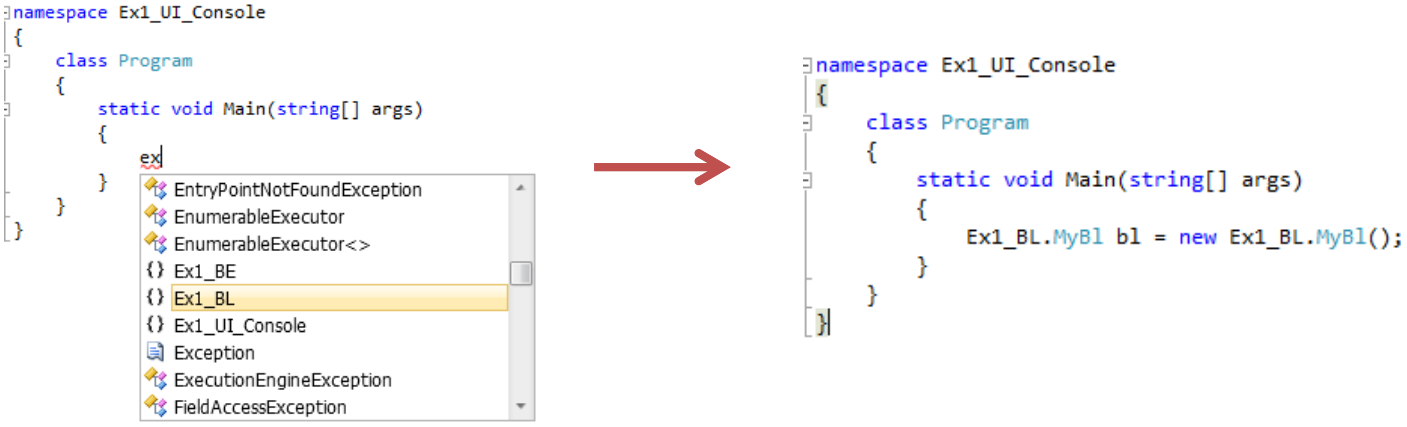
נתחיל עם UI:

לחיצה ימנית על הפרויקט UI ובחירה ב add reference ושם נבחר ב project ו BL

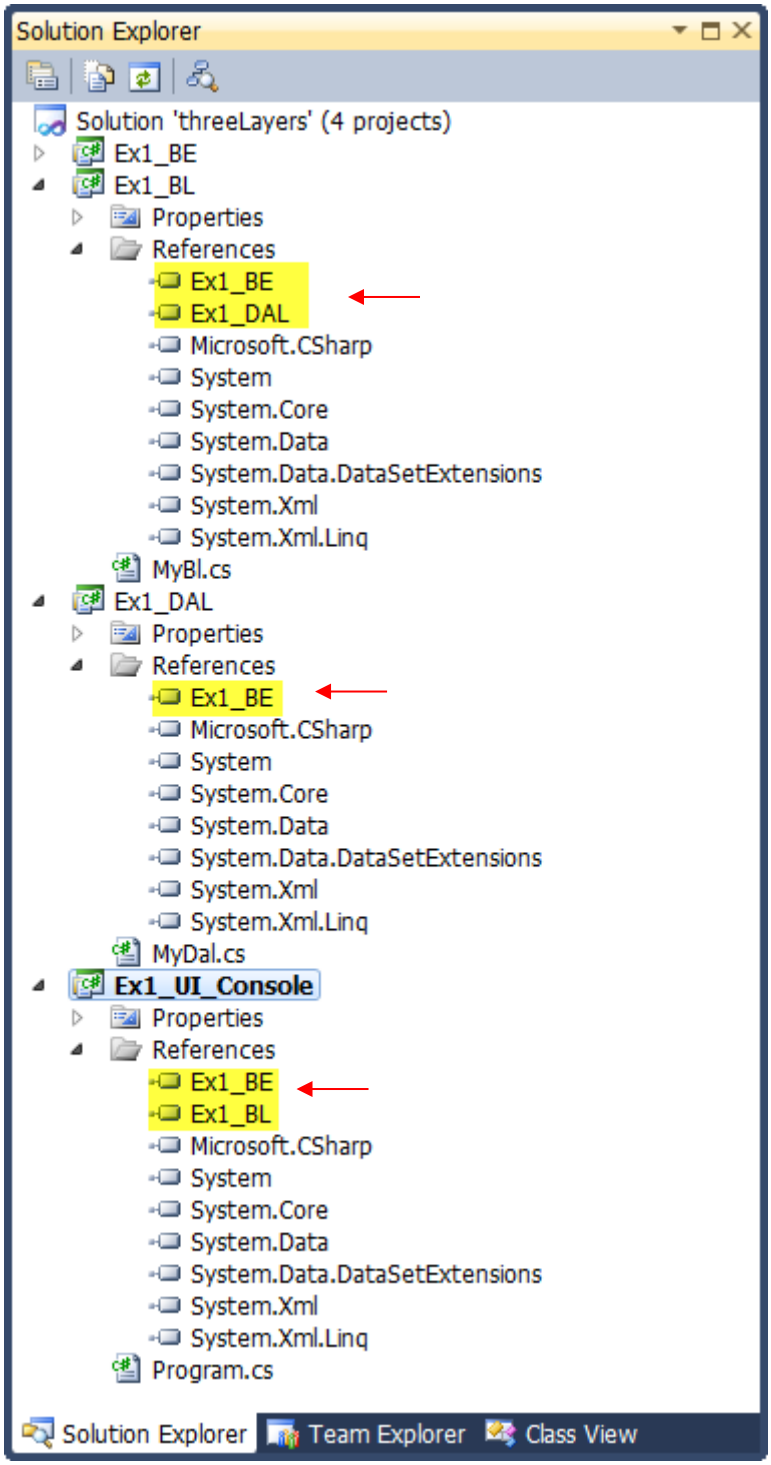


נבצע את אותו הדבר עבור BL שיקושר ל DAL ו BE
וגם ל DAL נקשר את BE

לאחר ביצוע שלב זה נוכל לגשת מפרויקט אחד לפרויקט אחר ע"י קידומת שם ה namespace שלו
לדוגמה עבור UI



שימו לב לקישורים תחת הקטגוריה reference שבכל פרויקט:



דוגמה:

דוגמה לפונקציה אחת בכל אחת מהשכבות BL ו DAL:

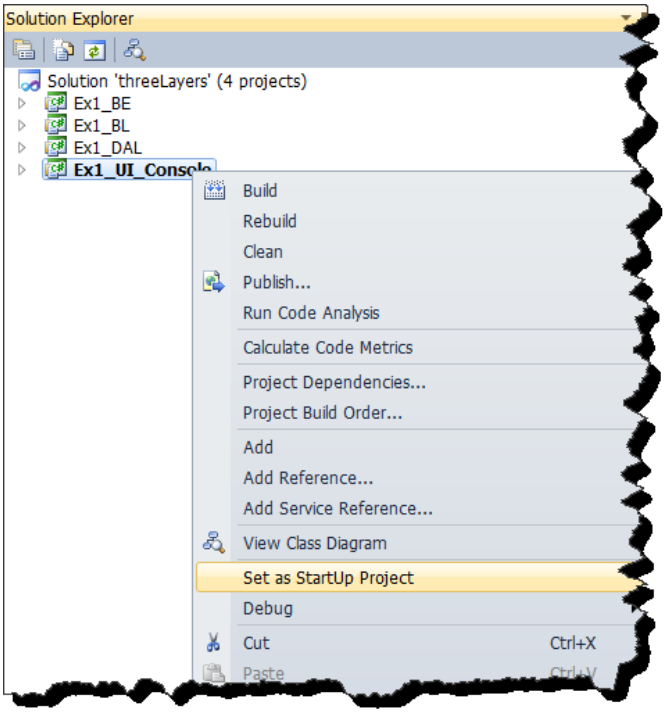
```
namespace Ex1_DAL
{
    public class MyDal
    {
        public string getValue()
        {
            return "dal send this value";
        }
    }
}
```

```
namespace Ex1_BL
{
    public class MyBl
    {
        Ex1_DAL.MyDal dal;

        public MyBl()
        {
            dal = new Ex1_DAL.MyDal();
        }

        public string getResult()
        {
            return "result is:" + dal.getValue();
        }
    }
}
```

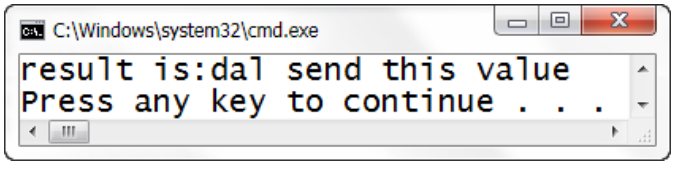
נגדיר ב program שב UI:

```
namespace Ex1_UI_Console
{
    class Program
    {
        static Ex1_BL.MyBl bl = new Ex1_BL.MyBl();

        static void Main(string[] args)
        {
            Console.WriteLine(bl.getResult());
        }
    }
}
```

נבחר את הפרויקט UI להיות הפרויקט שמתחיל לרוץ ראשון:

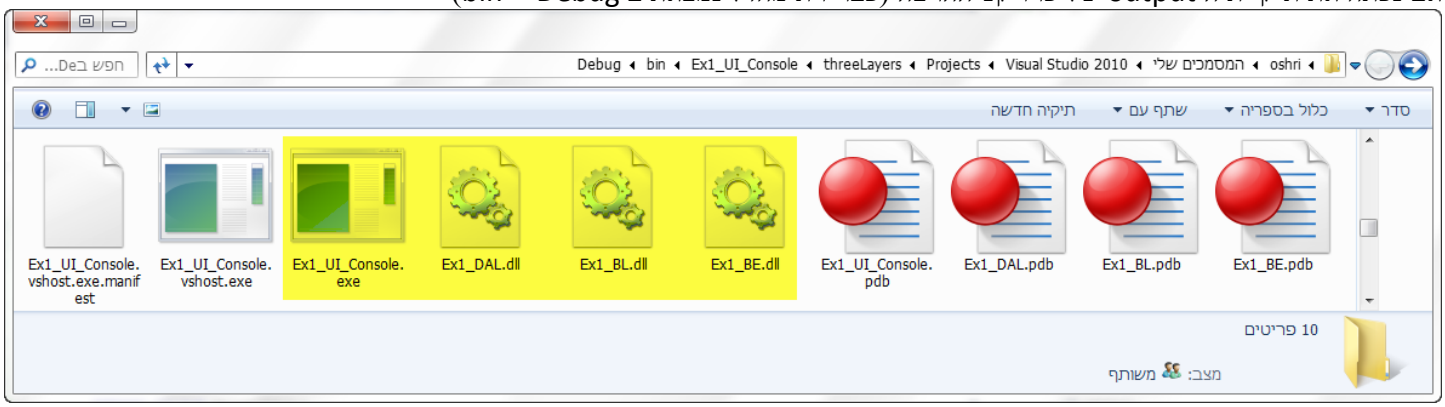


ונקבל את הפלט:



מה הרווחנו מהחלוקה הזו?

אם נפתח את תיקיית ה output של פרויקט ההרצה (כברירת מחדל נמצאת ב bin ->Debug)

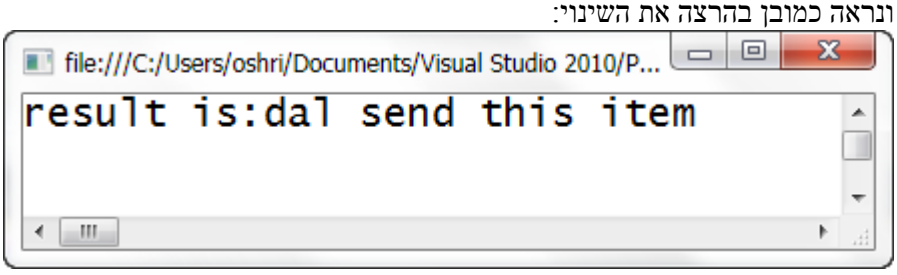


נראה שיש לנו 4 קבצים עיקריים שיחד מהווים את התוכנית שלנו.
אם נשלח ללקוח את שלושת הקבצים האלו הוא יוכל להפעיל את התוכנית באמצעותם.
אם אחד מהם חסר נקבל שגיאה רק בשימוש באותו קובץ בקוד (לדוגמה במקרה לעיל נוכל למחוק את ה DLL של BE)

כעת נוכל לגרור את ארבעת הקבצים לתיקייה אחרת נניח שזו התיקייה של הלקוח.
לשנות ב visual studio לדוגמה את DAL ל:

```
namespace Ex1_DAL
{
    public class MyDal
    {
        public string getValue()
        {
            return "dal send this item";
        }
    }
}
```

ונראה כמובן בהרצה את השינוי:



אבל הקבצים שאצל הלקוח כמובן לא השתנו.
כעת אנו צריכים לשנות אצל הלקוח רק את קובץ ה DAL והתוצאה שתתקבל אצלו תשקף את השינוי שנעשה.

היתרונות העיקריים:
בקבצי קוד גדולים זה חיסכון משמעותי בזמן קומפילציה.
יתכן שכל קובץ אסמבלי (BL DAL UI) נמצא כל אחד במיקום אחר (בעולם) ולכן כך נוכל לשדרג קובץ ספציפי באיזשהו מקום וזה יתעדכן אצל כולם.

מימוש נכון של השכבות

<u>רעיון ראשוני</u> (לא לביצוע) למימוש שמירה באמצעות מודל השכבות:

בשכבת ה DAL כמובן יש לנו מחלקה שמטפלת בעבודה עם הנתונים לדוגמה:

```
namespace Ex2_DAL
{
    public class Class_dal_save_at_list
    {
        public void add()
        {
        }
        public void remove()
        {
        }
        ///…
    }
}
```

בשכבת ה BL נצטרך להגדיר מופע של אותה מחלקה ואז להשתמש בנתונים:
בשכבת BL נניח יהיה את המחלקה הבאה:

```
namespace Ex2_BL
{
    public class Cass_BlAdapter
    {
       Ex2_DAL.Class_dal_save_at_list dal = new  Ex2_DAL.Class_dal_save_at_list();

       public void add()
       {
           dal.add();
       }
       public void remove()
       {
           dal.remove();
       }
        ///…
    }
}
```

<u>הבעיה:</u>
אם נרצה להגדיר ב DAL מחלקה אחרת לשמירה לדוגמה:

```
namespace Ex2_DAL
{
    class class_dal_save_at_DB
    {
        public void add()
        {
        }
        public void remove()
        {
        }
        ///…
    }
}
```

אזי נצטרך לשנות את המחלקה בשכבת ה BL כך:

```
namespace Ex2_BL
{
    public class Cass_BlAdapter
    {
       Ex2_DAL.Class_dal_save_at_DB dal = new  Ex2_DAL.Class_dal_save_at_DB();

       public void add()
       {
           dal.add();
       }
       public void remove()
       {
           dal.remove();
       }
        ///…
    }
}
```

במקרה הטוב כל הפונקציות בעלות אותו שם ולכן שינינו רק את השורה הראשונה של האיתחול:

```
Ex2_DAL.Class_dal_save_at_DB dal = new  Ex2_DAL.Class_dal_save_at_DB();
```

במקרה הגרוע גם שמות הפונקציות יהיו שונות
ולכן נדרשת תבנית תכנות שתאפשר לנו לבצע מינימום שינוים בעקבות שינוי במחלקת השמירה ב DAL

design by contract – תיכון על פי חוזה

<u>פתרון ראשון:</u>

נגדיר interface עם הפונקציות המתאימות ב DAL

ו BL יצור אובייקט רק של מי שיורש (מממש) את אותו interface ולכן אין בעיה ב BL עם הקריאות לפונקציות

ב DAL זה יראה כך:

```
namespace Ex3_DAL
{
    public interface IDAL
    {
        void add();
        void remove();
        ///…
    }
}
```

```
namespace Ex3_DAL
{
    class class_dal_save_at_DB : IDAL

    {
        public void add()
        {
        }
        public void remove()
        {
        }
        ///…
    }
```

```
namespace Ex3_DAL
{
    public class Class_dal_save_at_list : IDAL

    {
        public void add()
        {
        }
        public void remove()
        {
        }
        ///…
    }
```

וב BL זה יראה כך:

```
namespace Ex3_BL
{
    public class Cass_BlAdapter
    {
       Ex3_DAL.IDAL dal = new  Ex3_DAL.Class_dal_save_at_list();

       public void add()
       {
           dal.add();
       }
       public void remove()
       {
           dal.remove();
       }
        ///…
    }
}
```

כעת אם נרצה לשנות למימוש אחר ב DAL שמממש את האינטרפס IDAL
נוכל להיות בטוחים שנשנה רק את השורה:

```
Ex3_DAL.IDAL dal = new  Ex3_DAL.Class_dal_save_at_list();
```

factory

פתרון שני:

הפתרון הראשון בעצם עדיין מכריח את שכבת ה BL לרשום בפירוש מהו ה class שבאמצעותו משתמשים ב DAL
נרצה להשאיר את האחריות על כך רק לשכבת ה DAL ולכן נגדיר בשכבת ה DAL מחלקה שיש לה פונקציה אחת שסה"כ מחזירה מהו ה class שאתו משתמשים כרגע.

המחלקה ב DAL תיראה כך:

```
namespace Ex4_DAL
{
    public class FactoryDal
    {
        public IDAL getDal()
        {
            return new Class_dal_save_at_list();
        }
    }
}
```

ובמחלקה שבשכבת ה BL נשתמש כך:

```
namespace Ex4_BL
{
    public class Cass_BlAdapter
    {

        Ex4_DAL.IDAL dal;
        public Cass_BlAdapter()
        {
            Ex4_DAL.FactoryDal factory = new Ex4_DAL.FactoryDal();
            dal = factory.getDal();
        }

        public void add()
        {
            dal.add();

        }
        public void remove()
        {
            dal.remove();
        }
        ///…
    }
}
```

וכעת אם נרצה לשנות את המחלקה שבאמצעותה שומרים ב DAL נצטרך לבצע זאת רק בשכבת ה DAL כך:

```
namespace Ex4_DAL
{
    public class FactoryDal
    {
        public IDAL getDal()
        {
            return new Class_dal_save_at_DB();
        }
    }
}
```

והמחלקה ב BL נשארת כרגיל

תוספת אחרונה זה להגדיר את הפונקציה getDal ב Factory כ סטטית ואז לא חייב ליצור אובייקט Factory ב BL אלא כך:

ב DAL:

```
namespace Ex4_DAL
{
    public class FactoryDal
    {
        public static IDAL getDal()
        {
            return new Class_dal_save_at_list();
        }
    }
}
```

ב BL:

```
namespace Ex3_BL
{
    public class Cass_BlAdapter
    {
        Ex4_DAL.IDAL dal = Ex4_DAL.FactoryDal.getDal();

        public void add()
        {
            dal.add();

        }
        public void remove()
        {
            dal.remove();
        }
        ///…
    }
}
```

נספח א – חזרה על סוגי הרשאות ב #c:

כזכור ב #c קיימים סוגי ההרשאות המוכרות לנו גם מ ++c שהם public private protected וידוע לנו מה משמעותן.

סוג הרשאה internal

ב #c קיימת גם הרשאה שנקראת internal ומגדירה הרשאת גישה רק לקטעי קוד שנמצאים באותו האסמבלי דהינו באותו הפרויקט. ברירת המחדל של כל מחלקה זה להיות internal ולכן כל מחלקה בברירת מחדל מוכרת רק בתוך הפרויקט שלה, כלומר כל המחלקות שניצור ב DAL יהיו מוכרות רק ב DAL, אבל הריי נרצה להשתמש בהן מתוך BL, לכן הגדרנו אותן כ public.

ברירת מחדל של הרשאות ב #c
משתנים - private
פונקציות - private
מחלקות – internal

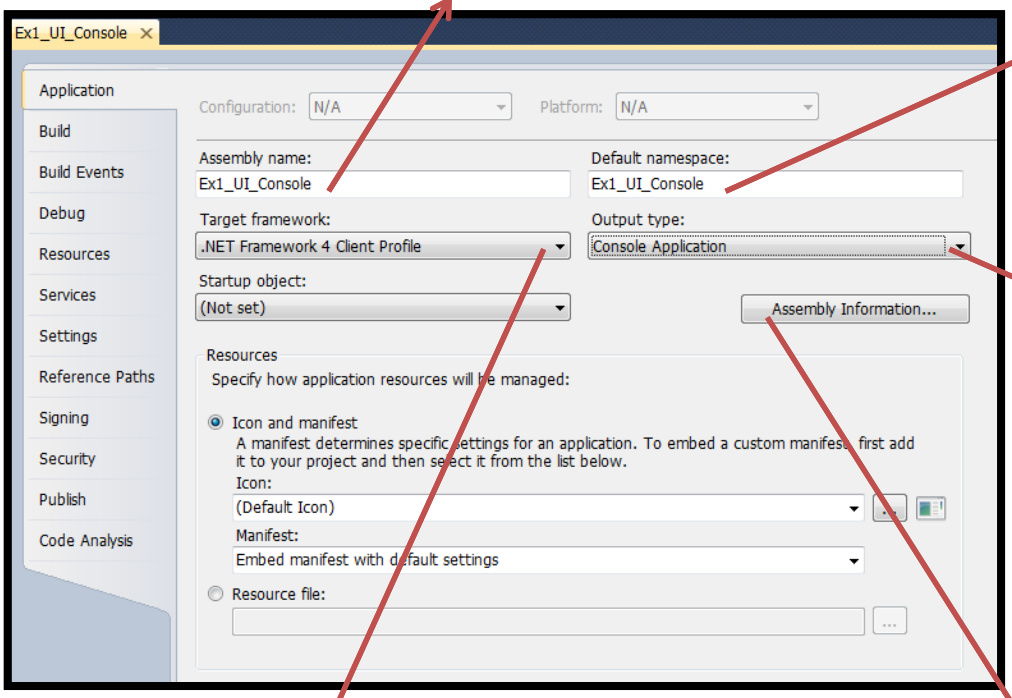
נספח ב – שינוי הגדרות של פרויקט ב visual studio 2010

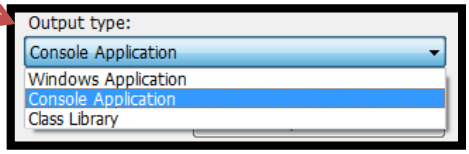
נוכל ללחוץ על כל פרויקט מקש ימני ולבחור ב properties (מאפייני הפרויקט)
בחלון שיפתח נוכל לראות את המאפיינים של הפרויקט:
לדוגמה עבור התווית AppLIcation נוכל לשנות את המאפיינים הבאים:

שם קובץ האסמבלי שיווצר

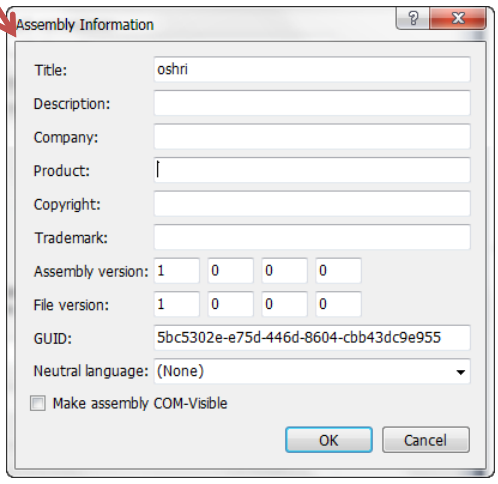
שם קובץ ה namespace שיווצר כברירת מחדל ביצירת item חדש.

סוג הפרויקט, פרויקט ריצה exe או פרויקט ספרייה dll

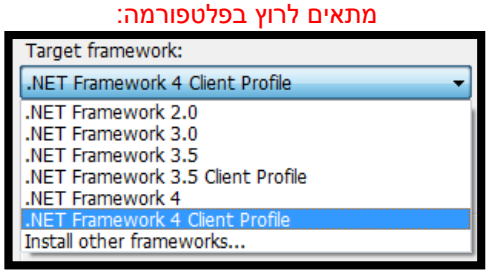




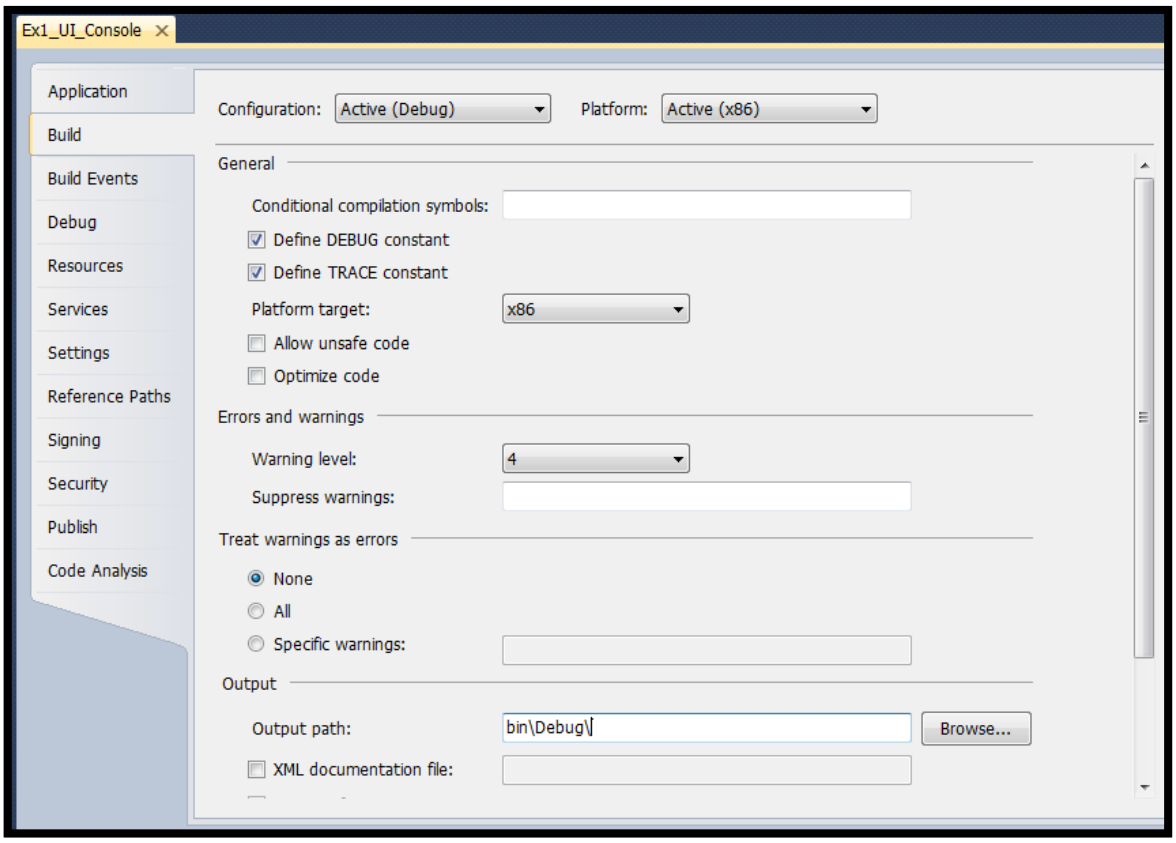
מידע נוסף על קובץ האסמבלי



מתאים לרוץ בפלטפורמה:



בתווית Build נוכל לשנות לדוגמה את תיקיית ה output של האסמבלי :



שיטות תיכנון תבניות עיצוב Design Patterns

Observer

הבעיה:
ברצוננו לבנות מחלקה חדשה שאחראית על איזשהו ערך שיכול להשתנות, ישנם מחלקות אחרות שאיננו יודעים על קיומם עדיין, אבל אנו יודעים שאותם המחלקות (לכשיוצרו, אם בכלל) היו רוצות לדעת שהערך השתנה, או יותר נכון: היו רוצות לדעת מה היה הערך הישן ומה כרגע הערך החדש.

המחלקה שלנו תהייה בערך כך (לפני השינוי\הפתרון):

```
public class MyValue
{
    private int myvalue = 0;

    public int Value
    {
        get { return myvalue; }
        set { myvalue = value; }
    }
}
```

הפתרון:
ע"י שימוש ב event:
המחלקה שלנו תגדיר event כך שכל מלחלקה אחרת תוכל להירשם לאותו event

נגדיר תחילה את המחלקה הבאה:

```
public class ValueChangedEventArgs : EventArgs
{
    public readonly int OldValue, NewValue;

    public ValueChangedEventArgs(int oldTemp, int newTemp)
    {
        OldValue = oldTemp;
        NewValue = newTemp;
    }
}
```

כעת נוכל להגדיר את ה delegate שיאפיין את ה event כך:

```
public delegate void ValueChangedEventHandler(Object sender, ValueChangedEventArgs args);
```

עכשיו נוסיף event תואם למחלקה שלנו להגדיר את המחלקה שלנו כך:

```
public class MyValue
{
    private int myvalue = 0;

    public event ValueChangedEventHandler ValueChanged;

    public int Value
    {
        get { return myvalue; }
        set { myvalue = value; }
    }

}
```

עכשיו כל מחלקה אחרת תוכל להירשם ל event שלנו ולהוסיף פונקציה משלה ע"י '+=' ל event ValueChanged שנמצא במחלקה שלנו.

אבל כרגע אנו לא עושים עם זה כלום.
אנו רוצים שברגע שהערך ישתנה נוכל להודיע זאת לכל מי שנרשם
אז ראשית ברור שעלינו להגדיר פונקציה חדשה שתפעיל את ה event שלנו (וע"י זה תפעיל את כל הפונקציות שנרשמו אליו).
נוסיף פונקציה `virtual protected void OnValueChanged(ValueChangedEventArgs args)`
(נתעלם ברגע זה למה דווקא וירטואל, זה יעבוד גם בלי)

```
public class MyValue
{
    private int myvalue = 0;

    public event ValueChangedEventHandler ValueChanged;

    virtual protected void OnValueChanged(ValueChangedEventArgs args)
    {
        if (ValueChanged != null)
        {
            ValueChanged(this, args);
        }
    }

public int Value
    {
        get { return myvalue; }
        set { myvalue = value; }
    }
}
```

שימו לב שיתכן ואף אחד לא נרשם לאירוע ו event שלנו שווה בעצם ל null ולכן יש צורך לבדוק זאת לפני ההפעלה של ה event
כך: `if (ValueChanged != null)`

עכשיו אמנם יש לנו את היכולת להפעיל את כל הפונקציות שנרשמו לאותו אירוע, אבל מתי בדיוק נעדכן אותם.

כמובן שהערך משתנה, ולכן נעדכן את ה set של הערך כך:
(וזה הפתרון הסופי של המחלקה שלנו)

```
public class MyValue
{
    private int myvalue = 0;

    public event ValueChangedEventHandler ValueChanged;

    virtual protected void OnValueChanged(ValueChangedEventArgs args)
    {
        if (ValueChanged != null)
        {
            ValueChanged(this, args);
        }
    }

    public int Value
    {
        get { return myvalue; }

        set
        {
            ValueChangedEventArgs args = new ValueChangedEventArgs(myvalue, value);
            myvalue = value;
            OnValueChanged(args);
        }
    }
}
```

כעת נראה כיצד מחלקה חדשה יכולה להשתמש באירוע של המחלקה שלנו.
נגדיר מחלקה שכל תפקידה יהיה להדפיס נתונים על ערך שהשתנה מהמחלקה MyValue .
כך:

```
public class ValueChangeObserver
{
    public ValueChangeObserver(MyValue t)
    {
        t.ValueChanged += this.ValueChange;
    }

    public void ValueChange(Object sender, ValueChangedEventArgs temp)
    {
        Console.WriteLine("ChangeObserver: Old={0}, New={1}, Change={2}",
            temp.OldValue, temp.NewValue,
            temp.NewValue - temp.OldValue);
    }
```

בפונקציה הבונה יש רישום לאירוע כך שכאשר יתרחש תופעל הפונקציה:

```
public void ValueChange(Object sender, ValueChangedEventArgs temp)
```

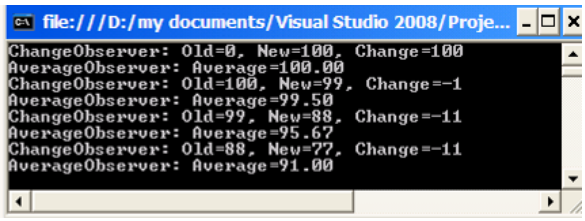
עם הפרמטרים שיתקבלו באותו אירוע.

נגדיר עוד מחלקה על מנת להראות רישום של שתי מחלקות שונות לאירוע.
תפקיד המחלקה הבאה היא להחזיר את ממוצע הערכים שקיבל ערך מהמחלקה MyValue .
המחלקה תוגדר כך:

```
public class ValueAverageObserver
{
    private int sum = 0, count = 0;

    public ValueAverageObserver(MyValue t)
    {
        t.ValueChanged += this.ValueChange;
    }

    public void ValueChange(Object sender, ValueChangedEventArgs temp)
    {
        count++;
        sum += temp.NewValue;

        Console.WriteLine("AverageObserver: Average={0:F}", (double)sum / (double)count);
    }
}
```

<u>דוגמה להרצה:</u>

```
public class MainClass
{
    public static void Main()
    {
        MyValue t = new MyValue();

        new ValueChangeObserver(t);
        new ValueAverageObserver(t);

        t.Value = 100;
        t.Value = 99;
        t.Value = 88;
        t.Value = 77;
    }
}
```



<u>על מנת לסבר את האוזן:</u>
ניתן לחשוב על המחלקה MyValue כעל מחלקה שמגדירה ערך של מנייה מסוימת בבורסה.
ויש מחלקה אחרת שמנהלת תיק לקוח שמשתמש באותה המנייה וצריך לדווח לו על הפסדים או רווחים ועל הממוצע הכללי מתחילת הרישום לאותה מנייה.
כמובן שהמחלקה MyValue אינה יודעת על קיומה של אותה מחלקה שמנהלת את תיק הלקוח ,אבל היא כן יודעת שבודאי יש מחלקה שתרצה לדעת על שינוי בערך המנייה.

Singleton

<u>הבעיה:</u>
נדרשת מחלקה שאנו רוצים לכל היותר רק מופע אחד שלה ושלא יהיה ניתן ליצור יותר ממופע אחד שלה.
הדרישות:
לכל היותר הכוונה שאם לא נעשה שימוש במחלקה זו אז לא נרצה ליצור אפילו אובייקט אחד ממנה.
הפתרון צריך להיות כפוי על כל קוד, כלומר לא יהיה ניתן ע"י שגיאה של מתכנת ליצור יותר ממופע אחד של אותה מחלקה.
יש לאפשר גישה לאותו אובייקט (אם נוצר כזה) מכל מקום בתוכנית.

<u>הפתרון:</u>
נגדיר מחלקה Singleton כך:

```
public class Singleton
    {
        private static Singleton instance;

        private Singleton()
        {

        }
        public static Singleton Instance
        {
            get
            {
                if (instance == null)
                    instance = new Singleton();
                return instance;
            }
        }
    }
```

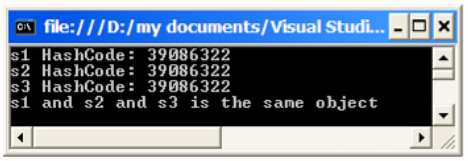
פונקציה בונה פרטית.

מאפשרים רק לקבל את המופע.
אם לא קיים מופע יוצרים אותו.

דוגמה לבדיקה שזה עובד:

```
static void Main()
{
 Singleton s1 =  Singleton.Instance;
 Singleton s2 =   Singleton.Instance;
 Singleton s3 =   Singleton.Instance;
 Console.WriteLine("s1 HashCode: {0} \ns2 HashCode: {1}\ns3 HashCode: {2} "
          , s1.GetHashCode(), s2.GetHashCode(), s3.GetHashCode());
   if (s1.Equals(s2) && s2.Equals(s3))
      Console.WriteLine("s1 and s2 and s3 is the same object");
 Console.ReadLine();

}
```

והפלט:



<u>הערה:</u>
הפתרון המוצע כאן איננו הפתרון המדויק לבעיית ה Singleton, כאן אנו לא לוקחים בחשבון מצב שבו יש עבודה במקביל (כמו שקיים כמעט בכל מערכת גדולה), במצב כזה יתכן שאחד התהליכונים יפנה ל get של ה singleton עוד לפני שנוצר אובייקט מסוג singeton ואז התנאי יהיה אמת כמובן, אבל מה יהיה אם לפני שאותו תהליכון יספיק ליצור מופע יגיע תורו של תהליכון אחר שגם עבורו כמובן התנאי יהיה אמת וכך לבסוף יהיו לנו שני תהליכונים שיצרו שני מופעים שונים של singleton.
במסגרת הקורס אנו לא מתייחסים למצב זה (כיוון שבין כה וכה אנו לא יוצרים אפליקציה לעבודה במקביל)