



סדנה בתכנות מונחה עצמים

מדריך למידה

دني קלפון

20586

מהדורה פנימית מעודכנת
לא להפצה ולא למכירה
מק"ט 20586-5132

צוות הקורס

כותב: דני קלפון

אחראי אקדמי: פרופ' ראובן אביב

אחראי אקדמי ללמידה המודפסת: פרופ' דוד לורנץ

עורכת: יהודית גוגנהיימר

הדפסה דיגיטלית – יולי 2014

© תשע"ד – 2014. כל הזכויות שמורות לאוניברסיטה הפתוחה.
בית החזאה לאור של האוניברסיטה הפתוחה, הקריה ע"ש דורותי דה ROTSHILD, דרך האוניברסיטה 1, ת"ד 808, רעננה 4353701.
The Open University of Israel, The Dorothy de Rothschild Campus, 1 University Road, P.O.Box 808, Raanana 4353701.
Printed in Israel.

אין לשכפל, להעתיק, לצלם, להקליט, לתרגם, לאחסן במאגר מידע, לשדר או לקלוט בכל דרך או בכל אמצעי אלקטרוני, אופטי,
מכני או אחר כל חלק שהוא מהחומר שבספר זה. שימוש מסחרי בחומר הכלול בספר זה אסור בהחלט, אלא ברשות מפורשת
ובכתב ממדור זכויות יוצרים של האוניברסיטה הפתוחה.

תוכן העניינים

7	פתח דבר
8	מתוכנת הסדנה
9	יחידה א – שפת C# 5.0
11	פרק 1 – יסודות שפת C#
11	1.1 הקדמה
11	1.2 הכוונה להכנות הרצאה
12	1.3 מהי פלטפורמת .NET
14	1.4 תוכניות לדוגמה
23	פרק 2 – עקרונות OOP בסיסיים בשפת C#
23	2.1 הקדמה
23	2.2 הכוונה להכנות הרצאה
24	2.3 תוכניות לדוגמה
29	פרק 3 – Interfaces and Collections
29	3.1 הקדמה
29	3.2 הכוונה להכנות הרצאה
30	3.3 תוכניות לדוגמה
35	פרק 4 – Callback Interfaces, Delegates and Events, Lambda Expressions
35	4.1 הקדמה
35	4.2 הכוונה להכנות הרצאה
37	פרק 5 – LINQ
37	5.1 הקדמה
37	5.2 דוגמאות קוד
41	יחידה ב – נושאים בעיצוב תוכנה מונחית עצמים
43	פרק 6 – UML: Unified Modeling Language
43	6.1 הקדמה
43	6.2 סיווג דיאגרמות UML
44	6.3 הכוונה להכנות הרצאה

פרק 7 – star UML	47
7.1	47
7.2	47
7.3	48
7.4	50
7.5	51
7.6	52
7.7	53
פרק 8 – דפוסי עיצוב (Design Patterns)	54
8.1	54
8.2	55
8.3	56
8.4	57
8.5	59
8.6	67
חידה ג – טכנולוגיות .NET	69
פרק 9 – XML	71
9.1	71
9.2	71
9.2.1	74
9.2.2	76
פרק 10 – ADO.NET	76
10.1	76
10.2	76
10.3	77
10.4	78
10.5	79
פרק 11 – ORM: Object Relational Mapping	80
11.1	80
11.2	81
11.3	81
11.4	81

83	הדגמה צעד אחר צעד של Entity Framework Model First	11.5
122	פרק 12 – WPF: Windows Presentation Foundation – 12	
122	הקדמה	12.1
122	הכוונה להכנת הרצאה	12.2
125.....	חידה ד – case study – 12	
127.....	פרק 13 – n-Tier Software Architecture – 13	
127	הקדמה	13.1
127	ארQUITקטורת 3 השכבות	13.2
128	יתרונות הארכיטקטורה	13.3
129	הכוונה להכנת הרצאה	13.4
131	חידה ה – פרויקט הגמר – 13	
133.....	פרק 14 – פרויקט הגמר – 14	
133	שלבי הגשת הפרויקט	14.1
134	הנחיות כלויות	14.2
135	נקודה למחשבה	14.3
136.....	פרק 15 – מסמך האפיון והניתוח – 15	
137.....	פרק 16 – מסמך עיצוב ותיכון – 16	
138.....	פרק 17 – מימוש והגשת הפרויקט – 17	
139.....	פרק 18 – פרויקט לדוגמה – 18	
293	נספח א – התקנת סביבת העבודה – 18	
294.....	נספח ב – קובץ מאמרים להעשרה – 18	



File #0003243 belongs to Roei Daniel- do not distribute

פתח דבר

מטרתה של סדנה זו היא התרנסות עם הפרדיגמה של תכנות מונחה עצמים באמצעות שימוש בשפת **C#**.

הסדנה תכלול לימוד של נושאים מתקדמים וחדשים של שפת **C# 5.0** וארQUITטורת **.NET 4.5**. חלקה העיקרי של הסדנה יוקדש לנושאים הקשורים בעיצוב מונחה עצמים.

במסגרת הקורס תתנסו באיסוף, עיבוד והציג מידע באופן ברור וממצה ותתמודדו עם עיצובו ותוכנותו של פרויקט בהיקף גדול.

ספרי הלימוד בקורס הם :

- Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides,
"Design Patterns: Elements of reusable Object-Oriented Software". Addison-Wesley Professional, 1995.
- Andrew Troelsen, *"Pro C# 5.0 and the .NET 4.5 Framework"*, 6/Ed. Apress 2012.

הספר הראשון מתמקד בנושא של מבניות עיצוב, נושא עיקרי בסדנה. הספר השני נועד לספק מקור ידע בנושא שפת **C# 5.0** ופלטפורמת **.NET 4.5**.

מדריך הלמידה מכיל הכוונה והדרכה ללימוד הנושאים שיוצגו בסדנה בצורה הרצאות סטודנטים. לכל נושא, ניתן הסבר על מהותו, הכוונה להכנות ההרצאה תוך מיקוד נושאי ההרצאה, ורשימת מקורותביבליוגרפיים מומלצים, ומספר דוגמאות.

במסגרת הסדנה יוגש פרויקט מסכם בהיקף גדול. הפרויקט מיישם את כל הנושאים הנלמדים בסדנה, הולכה למעשה. בסוף מדריך הלמידה תוכלו למצוא הסבר על אופן הגשת פרויקט הגמר.

הסדנה מיועדת לסטודנטים בעלי ידע קודם בנושא **Object Oriented Programming**. ולכן, חלק מהפרקים העוסקים בנושא זה ייסקר באופן מזויר וחלקו יילמד באופן עצמאי, על מנת "ליישר קו", ולעומוד על ההבדלים בין **C#** לשפות מונחות עצמים אחרות.

מאחר ששפרי הלימוד מקיפים מאוד, מומלץ כי תקרו את הספרים על פי ההנחיות המפורטות במדריך הלמידה.

אנו מוחלים לכם הצלחה בלימוד הסדנה ומקווים כי תפיקו ממנה תועלת.



מתקנות הסדנה

הסדנה מחולקת לשני חלקים עיקריים: למידת נושאי הלימוד, והגשת פרויקט גמר המגישים נושאים אלה.

נושאי הלימוד עוסקים הן בהקניית ידע יישומי והן בהקניית ידע תאורטי. כל הנושאים מכונים בעבר פרויקט הגמר. לצורך מהשאה לאופן העבודה הנדרש להגשת הפרויקט הסופי, יפותח פרויקט בהיקף קטן שבו יבואו לידי ביטוי חלק מהנושאים הנלמדים בסדנה.

נושאי הלימוד בסדנה יועברו כהרצאות על ידי הסטודנטים עצם ווינטן עליהם ציון. בפגש הראשון בסדנה יינתן הסבר קצר על הנושאים השונים. במהלך המפגש יחולקו נושאי ההרצאות בין הסטודנטים. הנוכחות במפגש זה היא חובה. יתר מפגשי הסדנה מוקדשים ללמידה הנושאים עצם.

כל סטודנט נדרש ללמידה הייטב את הנושא שקיבל בפגש הראשון, ולהכין עליו הרצאה באורך שעה. ביחידה הבאה של מדריך הלמידה תמצאו הכוונה והדרכה לכל נושא, וכן מיקוד לתוכנים שיש לכלול בהרצאה שתועבר בכיתה.

לאחר למידה נושא הסדנה, יהיה בידיכם הידע הנדרש להגשת פרויקט הגמר. את פרויקט הגמר יש להגיש בכמה שלבים. 1) ניתוח ואפיון. 2) עיצוב. 3) מימוש. לאחר הגשת הפרויקט, יתואם מועד להזגמה והגנה עליו. ביחידה ה' של מדריך הלמידה תמצאו הסבר מפורט על דרישות ההגשה של כל אחד משלבים אלה.

באטר הקורס תוכלו למצוא רשות נושאים אפשריים לפרוייקט הגמר,لوح מועדי ההרצאות ושיבוצי הסטודנטים, וכן את הממצאות של הנושאים שיועברו על ידיכם בסדנה. באתר ישנה גם קבוצת דיוון שבמסגרתה ניתן להתייעץ בנושאי הרצאות ופרויקט הגמר.

הרכיב ציון הסדנה

- **מטלה 11 – 30%**

הציון על מטלה זו כולל את ציון הרצאה שהועברה בכיתה.

- **מטלה 12 – 35%**

הציון על מטלה זו יורכב משני חלקים: הציון על מסמך הניתוח והאפיון (40%) והציון על הגשת מסמך העיצוב (60%).

- **מטלה 13 – 35%**

הציון על מטלה זו מtabסס על הערכת מימוש הפרויקט והגנה עליו.

C# 5.0 – שפת א – ייחידה



פרק 1 – יסודות שפת C#

1.1 הקדמה

בهرצתה זו עוסוק בהיכרות ראשונית עם שפת C# 5.0 ועם פלטפורמת NET 4.5. מטרת ההרצאה היא לתת סקירה על מגוון נושאים בסיסיים.

ההרצאה תכלול דוגמאות המביאות לידי ביטוי ידע מפרקים שונים בספר. המטרה אינה לרודת לעומק הפרטים בכל פרק אלא להציג את מירב הנושאים על מנת להציג סקירה ראשונית של השפה.

2.1 הכוונה להבנת ההרצאה

2.1.1 נושאים שיש לסקור בהרצאה

- סקירה על פלטפורמת NET.
- היכרות עם סביבת העבודה.
- כתיבת תוכנית והרצתה.
- הצגת מגוון תוכניות להמחשה ולהיכרות עם השפה בנושאים שונים. בין היתר יודגמו הנושאים הבאים:
 - קלט ופלט
 - עבודה עם קבצים
 - טיפול בחיריגות Exception Handling
 - טיפוס נתונים שונים, בייחוד עבודה עם מחרוזות System.Object
 - הגדרת מחלקה ויצירת מופעים עם בנאים שונים Properties
 - עבודה פשוטה עם Collections
 - הדגמת יכולות פשוטות של LINQ
 - שימוש פשוט ב- Delegate



1.2.2 ביבליוגרפיה

- פרקים 4-1 בספר הלימוד – סקירה והיכרות בסיסית עם סביבת העבודה והשפה.
- פרקים 5-6 בספר הלימוד – עקרונות בסיסיים של תכנות מונחה עצמים ב- C#.
- פרק 7 בספר הלימוד – טיפול בחיריגות.
- פרק 12 בספר הלימוד – LINQ על קצה המזלג.
- פרק 20 בספר הלימוד – עבודה עם קבצים וסיריאלייזציה של אובייקטים.
- אתר www.codeproject.com : CodeProject
- מאמר מומלץ : <http://www.codeproject.com/KB/cs/quickcsharp.aspx>
- האתר : www.csharp-station.com
- http://brainbell.com/tutors/C_Sharp/index.html

1.3 מהי פלטפורמת .NET.

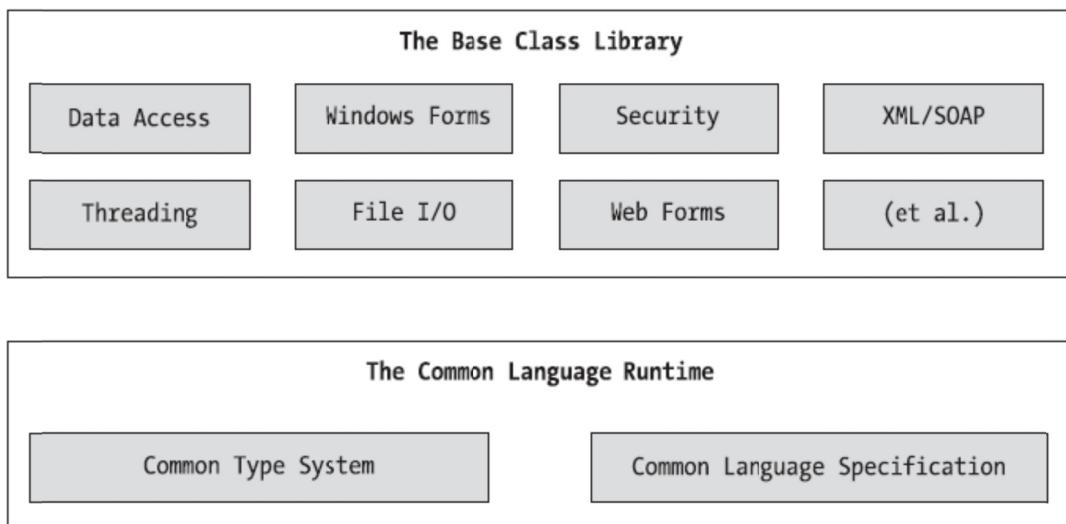
Managed Framework. היא פלטפורמה המאפשרת פיתוח אפליקציות תוכנה המכונת: Application. פלטפורמה זו מספקת מהדרים לשפות שונות, כלים לבנייה ולניפוי שגיאות, והרצה של Managed Application) משומש Application. אנו אומרים כי **האפליקציה מנוהלת** (Managed Application) שההרצה מנוהלת על ידי ה-.NET Framework.

בשונה מתוכנית בשפת C' למשל, תוכניות ב- C# אין מהודרות ישירות לשפת המכונה של המחשב. המהדר ממיר את קוד התוכנית לקובצי Microsoft Intermediate Language בשפת בינאים בשם (MSIL) , שנקרו Assemblies . כל שפות התכונות הנתמכות על ידי פלטפורמת ה- .NET. ממיר את התוכנית לשפת בינאים זו.

את תוכנית ה- MSIL מרים החלק שנקרא CLR – Common Language Runtime – CLR, שהוא לב לבה של הפלטפורמה.CLR מכיל מרכיבים שונים כגון : Garbage Collector ווד. JIT Compiler (FCL) בتوز ה- .NET Framework. ישן ספריות בהן אנו עושים שימוש בתוכניותינו. ספריות אלה נקראות Just In (Time) שבו קוד עובר המרה לשפת המכונה בזמן המתאים בו הוא נדרש, ה- CLR מתרגם באמצעות JITer את חלקו התוכנית המורכבים בדיקת בזמן שנדרש השימוש בהם, זה כולל גם את המחלקות.NET Framework בספריות של .NET

לקובץ ה- MSIL יש אמנים סיומת EXE אך הם אינם יכולים להיות מורצים במחשב שלא מותקנת בו פלטפורמת .NET.

להלן תרשימים מהספר המתאר את מבנה ה- .NET Framework :



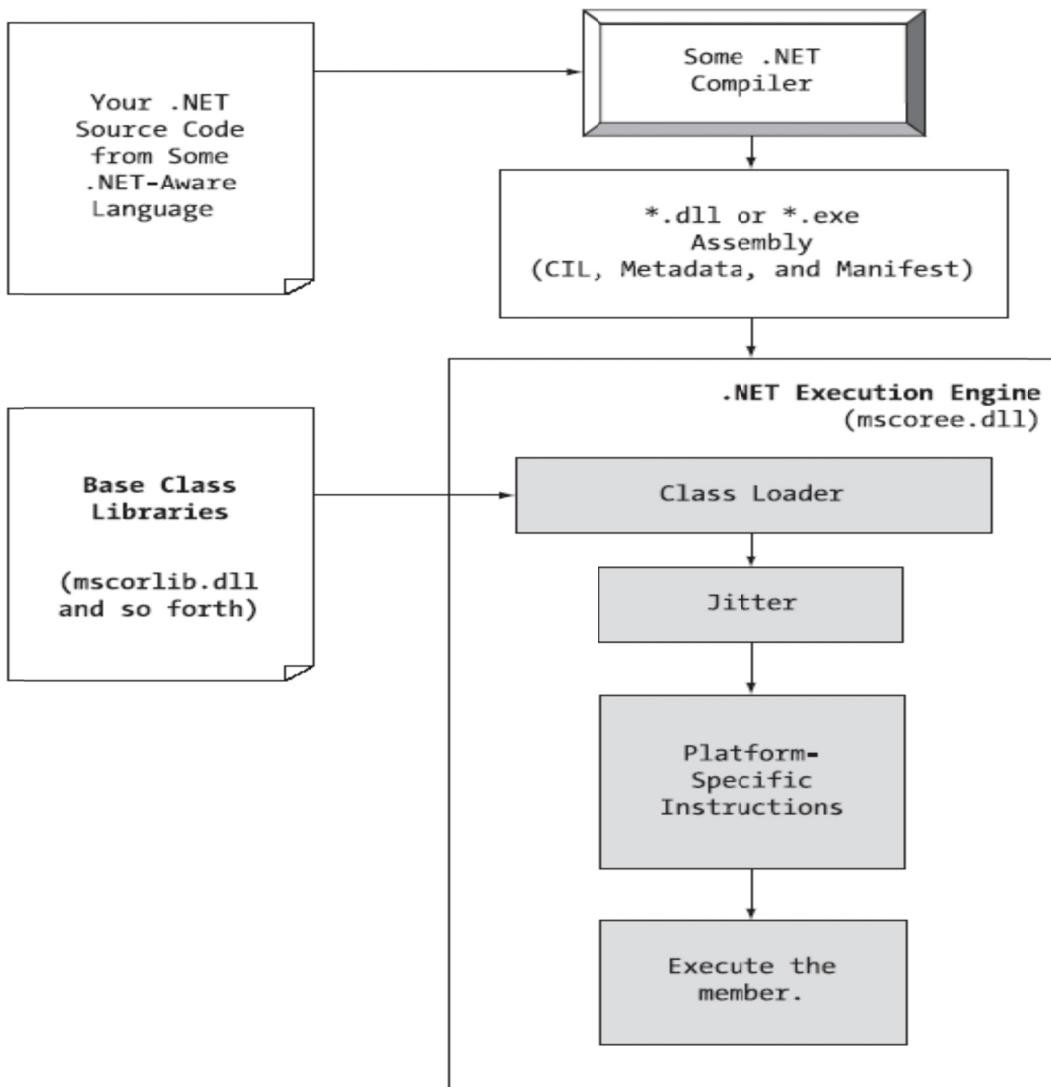
ה-CTS – Common Type System מתאר את כל הסוגים של טיפוסי הנתונים שנתמכו בזמן ריצה, ואת הקשרים ביניהם וכן מידע על אופן ייצוגם כ-METADATA. (כל טיפוס נתוני שהוא נגדי או השתמש מומר ומוצג על ידי הקומpileר בפורמט אחיד שנקרא METADATA).

ה-CLS – Common Language Specification מכיל קבוצת טיפוסים ומבנים מוסכמים המשותפת לכל השפות בהם תומך ה- Framework, כך שאם אנו בונים רכיב ספריה חדש המשתמש אך ורך באלמנטים הנתמכים על ידי CLS, אזי ניתן להבטיח שכל שפות התכונות שביהם תומך ה-Framework יחו עם כך בשлом. וכן, אם אנו עושים שימוש בטיפוס או במבנה שהם מחוץ להגדרות של CLS, אזי אין הבטחה שכל שפת תכונות תוכל לתקשר עם הרכיב שבינו.

להעמקת הקריאה במבנה ה-.NET Framework, תוכלו לעיין בפרק 1 בספר הלימוד.



להלן תרשيم המתאר את תהליך ההיידור של תוכנית ב-C#.



1.4 תוכניות לדוגמה

להלן מספר תוכניות להדגמה:

1.4.1 לולאת foreach

בנוסף לולאות המוכרות: while, for, do while מספקת השפה לולאה פחota מוכרת אך שימושית מאוד בתוכניות שנפגש בהמשך, והיא לולאת foreach. לולאה זו יודעת לסרוק אוסף כלשהו של איברים, לבצע פעולה על כל אחד מאיברי האוסף, ולהתקדם (באופן אוטומטי) לאייר הבא באוסף.

```
using System;
class loop_example
{
    public static void Main()
    {
        int[] grades = {100, 40, 90, 50, 70};

        foreach (int g in grades)
        {
            Console.WriteLine("{0}", g);
        }
    }
}
```

בתוכנית זו מוגדר מערך ומאותחל במספר ערכיים. לאחר מכן, מודפס תוכנו על ידי לולאת `foreach`. ההסבר הטוב ביותר להבנת אופן הפעולה של לולאה זו הוא פשוט תרגומה הישיר לעברית באופן הבא: "לכל מספר שלם `g` במערך `grades` בצע את גוף הלולאה". הלולאה דואגת עצמה לסריקת האוסף איבר אחר איבר. לולאה זו נוחה מאד לשימוש, ונרבבה להשתמש בה בתוכניותינו. נציין כי ניתן להפעילה על כל סוגי הנתונים, כולל אובייקטים; במקרה זה היא הופעלה על `int`-ים.

lolatforeach היא יישום של **דפוס העיצוב** Iterator, שהוא אחד מאוסף דפוסי העיצוב הנפוצים. על נושא דפוסי עיצוב נלמד בפרק 8. להבנת דפוס העיצוב Iterator עליו מושתתת lolatforeach תוכלו לקרוא בספר הדן בדפוסי עיצוב.

1.4.2 עבודה עם מחרוזות

מחרוזות הן טיפוס שימושי מאוד בשפה, ולא מעט מתודות מקובלות כפרמטרים או מחזירות כתשובה אובייקט מטיפוס מחרוזת. שפת C# מספקת את המחלקה `string` המייצגת מחרוזות ומאפשרת ביצוע פעולות רבות על מחרוזות.

להלן תוכנית המדגימה עבודה עם מחרוזות וביצוע מספר פעולות עליהם. התוכנית מקבלת משורה הפקודה `args` אוסף מחרוזות שעליהן תפעל. מחרוזות אלה מועברות למethode `Main` באמצעות הפרמטר .args



```

using System;
using System.Collections.Generic;
using System.Text;

namespace Example
{
    class String_Ex
    {
        public static void Main(string[] args)
        {
            foreach (string s in args)
            {
                Console.WriteLine("The string is : {0}", s);
                Console.WriteLine("Does the string contain
                    'B'?{0}", s.Contains("B"));
                Console.WriteLine(s.Replace("*", "#"));
                Console.WriteLine(s.ToUpper());
            }
        }
    }
}

```

להלן הפלט המתקיים:

```

D:\>string_ex aBc 5*6
The string is : aBc
Does the string contain 'B'?True
aBc
ABC
The string is : 5*6
Does the string contain 'B'?False
5#6
5*6

```

הסביר:

תוכנית זו מקבלת משורה הפוקודה `aosf machrozo`. לכל מהרווז באוסף זה היא מבצעת את הפעולות הבאות: הדפסת המחווזות, הדפסת `true/false` לשאלת האם המחווזות מכילה את האות `B`, החלפת של TWO * במחווזות בתו #, והמרת המחווזות לאותיות גדולות.

Delegates 1.4.3

הטיפוס `delegate` הוא אובייקט המשמש כ"מצבייע לפונקציה". אובייקט זה יכול אף להציבו במספר מethodot bio-zemnit, וניתן לוזמן דרך את המתוודות שאליוון הוא מצביע. המונח שמקובל לומר הוא: "האובייקט יזמן המתוודות שביצעו רישום () ל- registration זה."

קראו כעת את התוכנית הבאה המדגימה את אופן העבודה עם delegate, ואת ההסבר שלאחריה :

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Example
{
    public class delegate_Program
    {
        delegate void Operation(int val1, int val2);

        public static void Add(int val1, int val2)
        {
            Console.WriteLine(" Result of Add = {0}",
            val1+val2);
        }

        public static void Subtract (int val1, int val2)
        {
            Console.WriteLine(" Result of Sub = {0}",
            val1-val2);
        }

        public static void Main()
        {
            Operation Oper;
            Console.WriteLine("Enter + or - ");
            string optor = Console.ReadLine();
            Console.WriteLine("Enter 2 operands");
            string opnd1 = Console.ReadLine();
            string opnd2 = Console.ReadLine();
            int val1 = Convert.ToInt32 (opnd1);
            int val2 = Convert.ToInt32 (opnd2);
            if (optor == "+")
                Oper = Add;
            else
                if (optor == "-")
                    Oper= Subtract;
                Else{
                    Oper = Add;
                    Oper += Subtract;
                }
            Oper(val1, val2);
        }
    }
}
```

פלט התוכנית:

```
Enter + or -  
&  
Enter 2 operands  
12  
9  
Result of Add = 21  
Result of Sub = 3
```

הסבר התוכנית:

- בtower המחלקה הוגדר טיפוס Operation. טיפוס זה הוא delegate ומספק אפשרות להצבעה על מתודות שחתימתן כוללת 2 ארגומנטים מטיפוס int ולא ערך מוחזר. שימוש לב Ci Operation הוא רק טיפוס.
- במתודה Main, הוגדר משתנה Oper מטיפוס Operation האמור. משתנה זה מסוגל "להציג" על מתודות בעלות חתימה התואמת את הגדרת Operation, כגון המתודות Add ו-.Subtract
- לפי הקלט מהמשתמש שבוחר בפועלות חיבור או חיסור, מבצעים רישום של המתודה המתאימה. במקרה של פעולה חיבור נרשמת המתודה Add ב-Oper. ובמקרה של פעולה חיבור נרשמת המתודה Subtract ב-Oper. בכל מקרה אחר נרשמות הן המתודה Add והן המתודה Subtract ב-Oper.
- בהמשך נעשית הפעלה של המתודות הרשומות אצל Oper תוך שליחת הפרמטרים שנקלטו מהמשתמש.

1.4.4 טיפול בחיריגות

שפט # C מספקת לנו לנו לתפיסה ולטיפול בחיריגות הנדרקות עקב תקלת או שגיאה בזמן ריצה. כאשר נדרקת חריגה והיא אינה מטופלת, חריגה זו בסופו של דבר תיתפס על ידי ה-CLR שידאג להזפיס את פרטי החריגה שנוצרה, וכן לסיים את פעולה התוכנית. כמובן שהייה מעוניינים כי תוכנית תהיה ידידותית למשתמש במידה רבה ביותר, ולא תסימן את עובודתה עקב חריגה שנוצרה. לדוגמה, כאשר תוכנית מסוימת לשרת מרוחק ואנייה מצלילה, תיזרק לאחר פרק זמן מסוים חריגה עקב אי יכולת להתחבר לשרת. במקרה כזה לא נרצה שהחריגה תגיע ל-CLR שמצויד יסיים את פעולה התוכנית. במקומות זאת נעדיף לטפל בחיריגה בעצמנו וננסה שוב להתחבר לשרת המרוחק. כך ממשיך כמה פעמים, ורק אם כל הניסיונות נכשלים, רק אז נdfsiss בעצמנו הודעה על אי-יכולת להתחבר לשרת, ונסיים ביזמתנו את התוכנית.

נראה CUT כיצד קטע קוד שועלול לייצר חריגה בזמן ביצועו, ובמקרה שתהיה חריגה נוכל לTrap אותה ולטפל בה כרצוננו.

על מנת שנוכל לTrap ולטפל בחיריגה בעצמנו, שפט # C מעמידה לרשותנו את המילים השמרות עצם ו-`catch`. נראה תחילה את אופן השימוש בהם :

```
try{  
    ...  
}  
catch()  
{  
    ...  
}  
catch()  
{  
    ...  
}  
finally  
{  
    ...  
}
```

בבלוק ה-`catch` נמצא קטע קוד העולול לייצר חריגה. לאחריו נמצאים קטעי `catch` שאמורים לTrap מגוון חריגות אפשריות להיזרק במהלך ביצוע קטע הקוד שב-`catch`. ניתן לכתוב מספר קטעי `catch`, אחד לכל סוג חריגה. קטע ה-`finally` הוא קטע שיבוצע בכל מקרה, גם אם תהיה חריגה וגם אם לאו (קטע זה הוא אופציונלי).

נראה CUT דוגמה מסכמת לטיפול בחיריגות המדגימה אף את האפשרות להגדרת חריגה חדשה על ידינו.

```

using System;
using System.Collections.Generic;
using System.Text;

namespace Example
{
    class NoEvenException : Exception
    {
        public NoEvenException()
        {
            HelpLink = "www.Microsoft.com";
        }

        public NoEvenException(string msg)
        {
            HelpLink = "www.Microsoft.com";
            Source = msg;
        }
    }
    static public class DoSomeMath
    {
        static public int Add(int a, int b)
        {
            if (((a % 2) == 0) || ((b % 2) == 0))
            {
                NoEvenException ex = new NoEvenException();
                //throw the exception up to the sender
                throw (ex);
            }
            return a + b;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            int r;
            try
            {
                r = DoSomeMath.Add(1, 1);
                Console.WriteLine("Result of Add(1,1) is {0}", r);
                r = DoSomeMath.Add(7, 2);
                Console.WriteLine("Result of Add(7,2) is {0}", r);
            }

            catch (NoEvenException e)
            {

```

```

Console.WriteLine("Exception !!! - Even number!!!!");

Console.WriteLine("Exception: {0}", e);
Console.WriteLine(e.HelpLink);
Console.WriteLine("Press Enter to throw new
NoEvenException to the OS")

Console.ReadLine();
throw new NoEvenException("throw the exception up to
OS");
}

catch (Exception e)
{
    Console.WriteLine("Exception : {0}", e);
}
finally
{
    Console.WriteLine("finally part....do nothing");
}
Console.WriteLine("End of program")
Console.ReadLine();
}
}
}

```



הסבר התוכנית:

- התוכנית מגדרה תחילת חריגה חדשה בשם `NoEvenException`. ולכן מחלקה זו יורשת מהמחלקה הבסיסית `Exception`.
- לאחר מכן ישנה המחלקה `DoSomeMath` המגדירה בתוכה מתודה סטטית בשם `Add`. מתודה זו מקבלת שני מספרים כפרמטרים ומחזירה את סכומם. מתודה זו לא מתירה שליחת פרמטרים זוגיים: במקרה שאחד מהפרמטרים הוא זוגי, המתודה נורקת חריגת `NoEvenException` שהוגדרה על ידינו.
- במחלקה `Program` נמצאת מתודת `Main` שבה מtbody זימון למתודת `Add` פעם אחת עם פרמטרים אי-זוגיים. בזימון זה לא נורקת כל חריגה, ולכן מוחזרת ומודפסת תוצאת הסכום. בזימון השני נשלים פרמטרים שחילקם זוגיים, ולכן תיזורק חריגת שתיתפס על ידי בלוק ה-`catch` הראשון במתודת `Main`. בלוק זה מדפיס את פרטי החריגה, וכן ממשיך לנורק אותה לעבר מערכת הפעלה, שאף תדפיס עצמה את החריגת.
- בלוק ה- `finally` מtbody אף הוא.
- שימוש לב Ci ההודעה: `End of program` לא מודפסת במקרה זה, עקב העברת החריגת לרמת מערכת הפעלה.

פרק 2 – עקרונות OOP בסיסיים בשפת C#

2.1 הקדמה

בחרצאה זו יוצגו מספר נושאים ואופן יישום בשפת C#. נושאים אלה ידועים לכם מקורסים קודמים, והמטרה היא לא ללמדם שוב אלא רק להציג את אופן יישום בשפת C#, ול ישיר קו מבחינת הידע כחנה לҚරאת הנושאים המרכזיים של הסדרה.

הרצאה תכלול דוגמאות עבור הנושאים הבאים : **כימוס, ירושה, פולימורפיזם, העמסת אופרטורים ו-Generics**.

2.2 הכוונה להכנות הרצאה

הרצאה צריכה להיות מבוססת על גישה פרקטית, ובה יוצגו מגוון תוכניות תוך מתן הסבר לחלקים מרכזיים בהם. חלק מהתוכניות יהיו תוכניות מסכמתות המשלבות שימוש בכלל הנושאים.

2.2.1 נושאים שיש לסקור בהרצאה

- אופן הגדרת מחלקה והתמקדות בנושא **כימוס**. הדוגמאות שיוצגו יכללו שימוש במרקבים הבאים : **bananums, Properties, Partial Class, Write only fields, Read only fields**.
- **ירושה ופולימורפיזם**. הצגת דוגמאות להיררכיות ירושה של מחלקות, תוך הדגמת פולימורפיזם. מילוט המפתח בנושאים אלה הם : **private, public, protected, virtual, override, new, base, sealed, is, as, System.Object** מפתח אלה והדגמת אופן פועלתו.
- **Methods Extension**
- **העמסת אופרטורים**. הדגמת אופן העמסה של אופרטור אונרי ושל אופרטור ביןרי.
- **Generics**. דוגמאות למחלקה גנריית, מתודה גנריית והיררכיות מחלקות גנריות.
- **דוגמה מסכמת גדולה המשלבת את כל הנושאים האמורים.**



2.2.2 ביבליוגרפיה

- **כימוס – פרק 5** בספר הלימוד.
- **ירושה ופולימורפיזם – פרק 6** בספר הלימוד.
- **העמסת אופרטורים – פרק 11** בספר הלימוד.
- **Generics – פרק 9** בספר הלימוד.
- **אתר www.codeproject.com**
- **http://brainbell.com/tutors/C_Sharp/index.html**
- **<http://www.csharp-station.com/Tutorials/Lesson07.aspx>**
- **<http://www.csharp-station.com/Tutorials/Lesson08.aspx>**
- **<http://www.csharp-station.com/Tutorials/Lesson09.aspx>**

2.3 תוכניות לדוגמה

2.3.1 דוגמה להעמסת אופרטורים

התוכנית הבאה מדגימה שימוש בהעמסת אופרטורים לחיבור מספרים מרוכבים:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Example
{
    public class Complex
    {
        private int real;
        private int imaginary;

        public Complex(int real, int imaginary)
        {
            this.real = real;
            this.imaginary = imaginary;
        }

        // Declare which operator to overload (+),
        // the types that can be added (two Complex objects),
        // and the return type (Complex):
        public static Complex operator +(Complex c1,
        Complex c2)
        {
            return new Complex(c1.real + c2.real,
                c1.imaginary + c2.imaginary);
        }

        // Override the ToString() method
    }
}
```

```

public override string ToString()
{
    return System.String.Format("{0} + {1}i", real,
    imaginary);
}
}

class TestComplex
{
    static void Main()
    {
        Complex num1 = new Complex(2,3);
        Complex num2 = new Complex(3,4);

        // Add two Complex objects through the
        //overloaded plus operator:
        Complex sum = num1 + num2;

        // Print the numbers and the sum
        //using the overriden ToString method:
        System.Console.WriteLine("First complex number:
{0}", num1);
        System.Console.WriteLine("Second complex number:
{0}", num2);
        System.Console.WriteLine("The sum of the two
numbers: {0}", sum);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
}

```

פלט התוכנית המתקבל:

First complex number: 2 + 3i
 Second complex number: 3 + 4i
 The sum of the two numbers: 5 + 7i

2.3.2 דוגמה לירושה:

בדוגמה זו ישנה הגדלה של מחלקה בסיס אבסטרקטית לייצוג צורה, וממנה נגזרות מחלקות לייצוג עיגול, מלבן, וריבוע. לכל צורה יש צבע ושטח.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Example
{
    public abstract class Shape
    {
        protected string color;

        public Shape(string color)
        {
            this.color = color;
        }

        public string getColor()
        {
            return color;
        }

        public abstract double Area();
    }

    public class Circle : Shape
    {
        private double radius;

        public Circle(string color, double radius)
            : base(color)
        {
            this.radius = radius;
        }
    }
}
```

```

        public override double Area()
    {
        return System.Math.PI * radius * radius;
    }
}

public class Rectangle : Shape
{
    private double W, L;

    public Rectangle(string color, double W, double L)
        : base(color)
    {
        this.W = W;
        this.L = L;
    }

    public override double Area()
    {
        return W * L;
    }
}

public class Square : Shape
{
    private double Len;

    public Square(string color, double Len)
        : base(color)
    {
        this.Len = Len;
    }

    public override double Area()
    {
        return Len * Len;
    }
}

```



```
public class ShapesProg
{
    static void Main()
    {
        Circle myCircle = new Circle("orange", 3);
        Rectangle myRectangle = new Rectangle("red", 8, 4);
        Square mySquare = new Square("green", 4);

        Console.WriteLine("Circle Color is " +
myCircle.getColor());
        Console.WriteLine(" Circle area is : " +
myCircle.Area());
        Console.WriteLine(" Rectangle Color is " +
myRectangle.getColor());
        Console.WriteLine(" Rectangle area is " +
myRectangle.Area());
        Console.WriteLine(" Square color  is " +
mySquare.getColor());
        Console.WriteLine(" Square area is " +
mySquare.Area());
        Console.ReadLine();

    }
}
```

פלט התוכנית:

```
Circle color is orange
Circle area is :  28.2743338823081
Rectangle color is red
Rectangle area is 32
Square color is green
Square area is 16
```

פרק 3 – Interfaces and Collections

3.1 הקדמה

הרצאה זו עוסקת בנושא מנשקים וחויבותם. כן יוצגו מנשקים המספקים על ידי פלטפורמת .NET. ומגוון אוסףים (Collections) הממשים מנשקים אלה.

3.2 הכוונה להבנת ההרצאה

ההרצאה צריכה להיות מבוססת על גישה פרקטית, ובה יוצגו מגוון תוכניות תוך מתן הסבר לחלקים מרכזיים בהם. חלק מהתוכניות יהיו תוכניות מסכמות המשלבות שימוש בכלל הנושאים.

3.2.1 נושאים שיש לסקור בהרצאה

- מהו מנשך. השוואתו לעומת מחלקה אבסטרקטית.
- דוגמה לאופן הגדרת מנשך ושימוש בו.
- הורשה מרובה של מנשקים + הדגמת שימוש.
- היררכיה של מנשקים (ירושה בין מנשקים) + הדגמת שימוש.
- העברת מנשך כפרמטר למתודה.
- מנשך כערך מוחזר ממתחודה.
- סקירת מנשקים המוגדרים ב-.NET (... Ienumerable, Icomparable,)
- אוסףים (Collections) – סיווג האוסףים השונים, אילו מנשקים הם ממשיים, והציגת מספר תוכניות המדגימות שימוש במגוון אוסףים.

3.2.2ביבליוגרפיה

- מנשקים – פרק 8 בספר הלימוד
- אוסףים – פרק 9 בספר הלימוד
- MSDN
- www.underwar.co.il/library.asp?Page=Programming#cat105
- http://brainbell.com/tutors/C_Sharp/index.html



3.3 תוכניות לדוגמה

להלן מספר תוכניות המדגימות שימוש במנשכים ואוספים:

3.3.1 דוגמה לשימוש ב- `IEnumerable`

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Text;

namespace Example
{
    public class Animal
    {
        public Animal(string AName, string AColor)
        {
            this.Name = AName;
            this.Color = AColor;
        }

        public string Name;
        public string Color;
    }

    public class Zoo : IEnumerable
    {
        private Animal[] A;
        public Zoo(Animal[] Z)
        {
            A = new Animal[Z.Length];
            for (int i = 0; i < Z.Length; i++)
            {
                A[i] = Z[i];
            }
        }
    }
}
```

```

        public IEnumarator GetEnumerator()
    {
        return new ZooEnum(A) ;
    }
}

public class ZooEnum : IEnumarator
{
    public Animal[] A;

    int position = -1;

    public ZooEnum(Animal[] Anm)
    {
        A = Anm;
    }

    public bool MoveNext()
    {
        position++;
        return (position < A.Length);
    }

    public void Reset()
    {
        position = -1;
    }

    public object Current
    {
        get
        {
            try
            {
                return A[position];
            }
        }
    }
}

```



```
        catch (IndexOutOfRangeException)
    {
        throw new InvalidOperationException();
    }
}

class Ex
{
    static void Main()
    {
        Animal[] ZooZoo = new Animal[3] {
            new Animal("Cow", "Black"),
            new Animal("Horse", "White"),
            new Animal("Zvuv", "Metali"),
        };

        Zoo z2 = new Zoo(ZooZoo);
        foreach (Animal anm in z2)
            Console.WriteLine(anm.Name + " " + anm.Color);
        Console.ReadLine();
    }
}
```

פלט התוכנית :

Cow Black
Horse White
Zvuv Metali

3.3.2 דוגמה לשימוש עם HashTable

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Text;

namespace Example
{
    public class CL
    {
        static void Main()
        {
            // create and initialize a new hashtable
            Hashtable persons = new Hashtable();
            persons.Add("00444444", "David Ben Gurion");
            persons.Add("00111111", "Shai Agnon");
            persons.Add("00222222", "Naomi Shemer");
            persons.Add("00333333", "Yizhak Rabin");

            // get the keys from the hashtable
            ICollection k = persons.Keys;
            // get the values
            ICollection v = persons.Values;

            // iterate over the key ICollection
            foreach (string key in k)
            {
                Console.WriteLine("{0} ", key);
            }

            // iterate over the values collection
            foreach (string value in v)
                Console.WriteLine("{0} ", value);
            Console.ReadLine();

        }
    }
}
```



פלט התוכנית:

0022222

0033333

0044444

0011111

Naomi Shemer

Yizhak Rabin

David Ben Gurion

Shai Agnon

פרק 4 Callback Interfaces, Delegates – 4 and Events, Lambda Expressions

4.1 הקדמה

הרצאה זו עוסקת במבנה מנגןוני Callback. ניתן לבנות Callback באמצעות שימוש במנשקים. במסגרת שיעור זה נלמד על המחלקות delegate ו- event המספקות על ידי פלטפורמת.NET. ונראה כיצד לבנות בעזרם מנגןון Callback בצורה נוחה ומהירה יותר מאשר עם מנסקים.

4.2 הכוונה להכנות ההרצאה

ההרצאה צריכה להיות מבוססת על גישה פרקטית, ובה יוצגו מגוון תוכניות תוך מתן הסבר לחלקים מרכזיים בהם. רצוי להסביר את התוכניות בליווי תרשימים הממחישים את פעולת התוכניות.

4.2.1 נושאים שיש לסקור בהרצאה

- מהו ?Callback
- הצעת דוגמה מפורטת למימוש מנגןון Callback על ידי שימוש במנשקים בלבד.
- מהו ?Delegate
- דוגמה פשוטה לאופן הגדרה ושימוש.
- הצעת דוגמה למימוש מנגןון Callback באמצעות Delegates
- Multicast Delegate
- Anonymous methods
- Lambda Operator and Lambda Expressions
- המחלקה Event. יש להסביר בדומה ל-.Delegate
- EventArgs ו- EventHandler עבודה עם



4.2.2 ביבליוגרפיה

- פרק 10 בספר הלימוד
- MSDN
- www.c-sharpcorner.com
- www.csharpfriends.com
- <http://www.billbblog.com/Programming/4/C%-Callbacks>
- http://brainbell.com/tutors/C_Sharp/index.html

פרק 5 – LINQ

5.1 הקדמה

כאשר אנו עובדים מול מקור נתונים כלשהו (אוסף, קובץ XML, בסיס נתונים וכדומה) אנו בדרך כלל מבצעים פעולות כגון: הוספה, עדכון, מחיקה ובחירה של נתונים הרצויים לנו.

LINQ (Language Integrated Query) היא תוספת לSYSTEMFORTRAN הפיתוח המבוססת על שפת SQL. SQL היא שפת שאלות דומה ל-SQL על כל מקור נתונים שברצוננו לעבוד מולו. היתרונו בכך הוא פשוטה בכתיבה הביטויים (השאלות) וכן בזרת העבודה איחוד מול כל מקור נתונים.

5.2 דוגמאות קוד

דוגמה 1:

בדוגמה הבאה מוצגת תוכנית שבה ישנו אוסף מסווג List של אובייקטים מסווג customers. באמצעות שאלתinq התוכנית מוצאת ומדפיסה את שמות כל הלקוחות שגילם קטן מ-25.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Linq;

namespace Linq
{
    public class Customer
    {
        public string Name { get; set; }
        public int Age { get; set; }
    }

    class Program
    {

        static void Main(string[] args)
        {
            List<Customer> customers = new List<Customer>();

            customers.Add(new Customer() { Name = "Joe", Age = 20 });
            customers.Add(new Customer() { Name = "Steve", Age = 29 });

            var customerUnder25 =
                from customer in customers
                where customer.Age < 25
                select customer;

            foreach (Customer customer in customerUnder25)
            {
                Console.WriteLine("Customer name: {0}", customer.Name);
            }
            Console.ReadLine();
        }
    }
}
```



דוגמה 2:

התוכנית הבאה מדגימה שוב הרצת שאילתת Linq על אוסף לkusות ומציאת הלkusות שגילם קטן מ-30. הפעם עברו לkusות אלה מוחזר אובייקט מסווג אחר.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Linq;

namespace Linq
{
    public class Customer
    {
        public string Name { get; set; }
        public int Age { get; set; }
    }

    public class Gift
    {
        public int GiftCode { get; set; }
        public Customer Customer { get; set; }
    }

    class Program
    {

        static void Main(string[] args)
        {
            List<Customer> customers = new List<Customer>();

            customers.Add(new Customer() { Name = "Joe", Age = 20 });
            customers.Add(new Customer() { Name = "Yossi", Age = 18 });
            customers.Add(new Customer() { Name = "Avi", Age = 18 });
            customers.Add(new Customer() { Name = "Steve", Age = 31 });

            var myQuery =
                from customer in customers
                where customer.Age < 30
                select new Gift() { GiftCode = 1, Customer = customer };

            foreach (Gift gift in myQuery)
            {
                Console.WriteLine("Customer name: {0}", gift.Customer.Name);
            }
            Console.ReadLine();
        }
    }
}
```

דוגמה 3:

בדוגמה הבאה מוגדרות 2 שיטות LINQ המדגישות שימוש ב- `order by` המוכר משפט SQL

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Linq;

namespace Linq
{
    public class Customer
    {
        public string Name { get; set; }
        public int Age { get; set; }
    }

    class Program
    {

        static void Main(string[] args)
        {
            List<Customer> customers = new List<Customer>();

            customers.Add(new Customer() { Name = "Joe", Age = 20 });
            customers.Add(new Customer() { Name = "Yossi", Age = 18 });
            customers.Add(new Customer() { Name = "Avi", Age = 18 });
            customers.Add(new Customer() { Name = "Steve", Age = 29 });

            var myQuery1 =
                from customer in customers
                orderby customer.Age, customer.Name
                select customer;

            var myQuery2 =
                from customer in customers
                orderby customer.Age descending, customer.Name descending
                select customer;

            foreach (Customer customer in myQuery1)
            {
                Console.WriteLine("Customer name: {0}", customer.Name);
            }

            Console.WriteLine();

            foreach (Customer customer in myQuery2)
            {
                Console.WriteLine("Customer name: {0}", customer.Name);
            }

            Console.ReadLine();
        }
    }
}
```

דוגמה 4:

בדוגמה הבאה מודגמת שאילתת LINQ המדגימה שימוש ב- group by המוכר משפט .SQL

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Linq;

namespace Linq
{
    public class Customer
    {
        public string Name { get; set; }
        public int Age { get; set; }
    }

    class Program
    {

        static void Main(string[] args)
        {
            List<Customer> customers = new List<Customer>();

            customers.Add(new Customer() { Name = "Joe", Age = 20 });
            customers.Add(new Customer() { Name = "Yossi", Age = 18 });
            customers.Add(new Customer() { Name = "Avi", Age = 18 });
            customers.Add(new Customer() { Name = "Steve", Age = 31 });

            var myQuery =
                from customer in customers
                group customer by customer.Age into ageGroup
                select new { Age = ageGroup.Key, Count = ageGroup.Count() };

            foreach (var item in myQuery)
            {
                Console.WriteLine("{0} were in group of age {1}", item.Count,
                                  item.Age);
            }
            Console.ReadLine();
        }
    }
}
```

יחידה ב – נושאים בעיצוב תוכנה МОНОХИЯ УЦМІМ

File #0003243 belongs to Roei Daniel- do not distribute

פרק 6 – Unified Modeling Language

6.1 הקדמה

UML (Unified Modeling Language) היא אוסף של דיאגרמות שבאמצעותן ניתן לתאר בצורה ויזואלית את הדרישות, הארכיטקטורה, הפריסה והמצבים של מערכת.

UML מאחדת את עבודותיהם של Rumbaugh (1991), Booch (1994), Jacobson (1992) ועובדות נוספות.

באמצעות שפת UML ניתן לתאר בצורה מקיפה את כל תהליך הניתוח והעיצוב של מערכת מונחית עצמאים.

6.2 סיווג דיאגרמות UML

שפת UML מורכבת משלוש עשרה דיאגרמות שונות. דיאגרמות אלה מיועדות לתאר היבטים שונים על המערכת והן נחלקות ל-3 קטגוריות:

6.2.1 דיאגרמות מבנה

דיאגרמות מבנה מתארות את המבנה הסטטי של המערכת, תוך התמקדות באלמנטים המרכיבים את המערכת ללא תלות בזמן. מדיאגרמות אלה ניתן ללמוד על המחלקות וה모פעלים הקיימים במערכת. כמו כן, ניתן ללמוד על הקשרים והתלויות בין המחלקות במערכת. דיאגרמות המבנה כוללות את הדיאגרמות הבאות:

- Class Diagram
- Object Diagram
- Component Diagram
- Composite Structure Diagram
- Package Diagram
- Deployment Diagram



6.2.2 דיאגרמות הניווט

דיאגרמות הניווט הם אוסף דיאגרמות שמחישות כיצד מתבצע תהליך מסוים במערכת. התיאור כולל את האירועים והפעולות המתרחשים במערכת, וכן את תגובת המערכת ותיאור התוצאות. דיאגרמות הניווט כוללות את הדיאגרמות הבאות:

- Use Case Diagram
- Activity Diagram
- State Machine Diagram

6.2.3 דיאגרמות אינטראקטיביות

דיאגרמות אינטראקטיביות הן אוסף דיאגרמות שבאמצעותן ניתן לתאר ברמת פירוט גבוהה יותר את האינטראקטיביות המתקיים בין אובייקטים. הדיאגרמות מתראות את הניווט האובייקטיבים ואת האינטראקטיבית של קבוצת אובייקטיבים להשלמת משימה מסוימת. דיאגרמות האינטראקטיביות כוללות את הדיאגרמות הבאות:

- Sequence Diagram
- Timing Diagram
- Interaction Overview Diagram
- Communication Diagram

6.3 הכוונה להכנת הרצאה

במסגרת הרצאה בנושא זה ייסקרו מגוון הדיאגרמות השונות. לכל דיאגרמה יוצג הסבר מפורט של כל הסימנים הגרפיים המרכיבים אותה, מטרתה של הדיאגרמה ובאיילו סיטואציות היא נחוצה, כל זאת תוך מתן דוגמה קונקרטית.

הרצאה תכלול דוגמה לתיאור מילולי של מערכת (לא מסובכת מדי) שברצוננו לבנות. באמצעות הדיאגרמות השונות של UML יינתן תיאור של דרישות המערכת, עיצוב מבנה המערכת ותהליכיים שונים המתרחשים בה.

6.3.1 נושאים שיש לסקור בהרצאה

- שפת UML – רקע כללי, מטרות.
- דיאגרמות מבנה – תאור כל הדיאגרמות בקטgorיה זו : סימוניים של הרכיבים השונים ומבנה הדיאגרמה.
- דיאגרמות התנהגות – תאור כל הדיאגרמות בקטgorיה זו : סימוניים של הרכיבים השונים ומבנה הדיאגרמה.
- דיאגרמות אינטראקציה – תאור כל הדיאגרמות בקטgorיה זו : סימוניים של הרכיבים השונים ומבנה הדיאגרמה.
- דוגמה מסכמת – הצגת תיאור מילולי של מערכת מסוימת, והדגמת ניתוח ועיצוב המערכת באמצעות דיאגרמות UML. בפרט יש להציג את הדיאגרמות הבאות :

Use Case Diagram, Class Diagram, Activity Diagram, Sequence Diagram.

6.3.2ביבליוגרפיה

- Shoemaker, M. (2004) “*UML Applied: A .Net Perspective*”. Apress 2004, <http://www.devshed.com/c/a/Practices/Introducing-UMLObjectOriented-Analysis-and-Design>
- Gorman, J. (2005) “*UML for managers*”.
- Hoffer, J.A. and George, J.F. and Valacich, J.S. (2002). *Modern systems analysis & design*. Prentice Hall.
- Booch, G. (1987). *Software engineering with ADA*. BENJAMIN/CUMMINGS
- Bell, D. (2003). “UML basics: An introduction to the Unified Modeling Language”. IBM, <http://www.ibm.com/developerworks/rational/library/769.html>

- Pilone, D. and Pitman, N. (2005). *UML 2.0 in a Nutshell*. O'Reilly Media,
<http://books.google.com/books?id=Blbdm9se5mIC&printsec=toc&dq=+basics+OR+uml+%22composite+structure+diagram%22&output=html>
- Braun, D. and Sivils, J. and Shapiro, A. and Versteegh, J. (2001). *UML TUTORIAL*.
Kennesaw State University
http://atlas.kennesaw.edu/~dbraun/csis4650/A&D/UML_tutorial/index.htm
- *UML Tutorial*. Sparx Systems
http://www.sparxsystems.com.au/UML_Tutorial.htm
- Ambler, S.W. (2007). *Introduction to the Diagrams of UML 2.0*.
<http://www.agilemodeling.com/essays/umlDiagrams.htm>

פרק 7 – Star UML

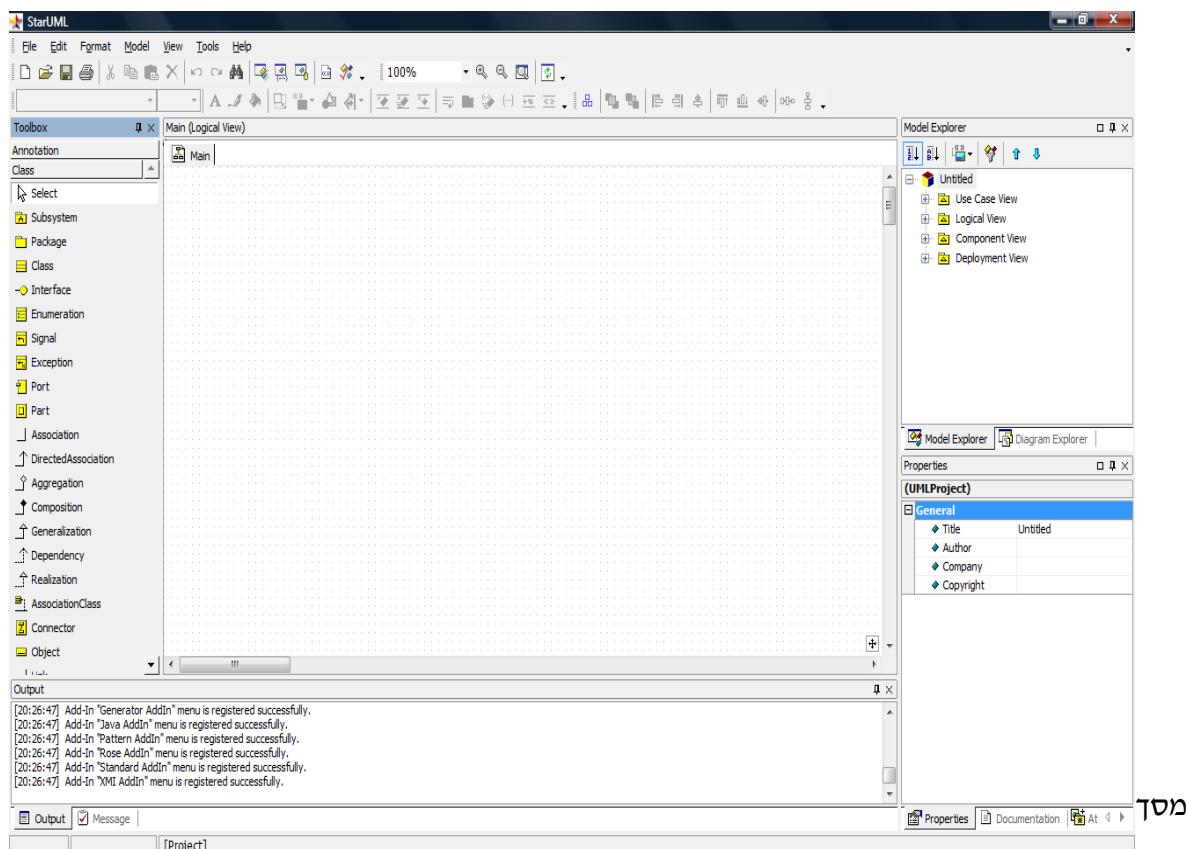
7.1 הקדמה

הו פרויקט קוד פתוח המספק תוכנה לעריכת דיאגרמות UML. התוכנה היא חינמית (ניתנת בחינם), גמישה, מהירה, וניתנת להרחבה. מטרתו של פרויקט זה היא החלפת כלים גדולים ומסחריים כדוגמת Rational Rose Forward Engineering Star UML. Rational Rose הוא ב- Reverse Engineering, הפיכת קוד לדיאגרמות UML. בנוסף ניתן גם לדיאגרמות UML לקוד, והו ב- מודולים שונים. בנוסף ניתן גם להמיר חלק מהדיאגרמות לסטנדרטים שונים באופן אוטומטי, דבר החושך המון כתיבה.

תוכנת Star UML נמצאת בתקליטור המצורף לחבילת הקורס. תהליך ההתקנה הוא פשוט ומהיר. באתר הפרויקט (<http://staruml.sourceforge.net/en/>) ניתן למצוא מידע וסבירים מקיפים על אופן השימוש בתוכנה.

7.2 הסבר קצר על סביבת העבודה

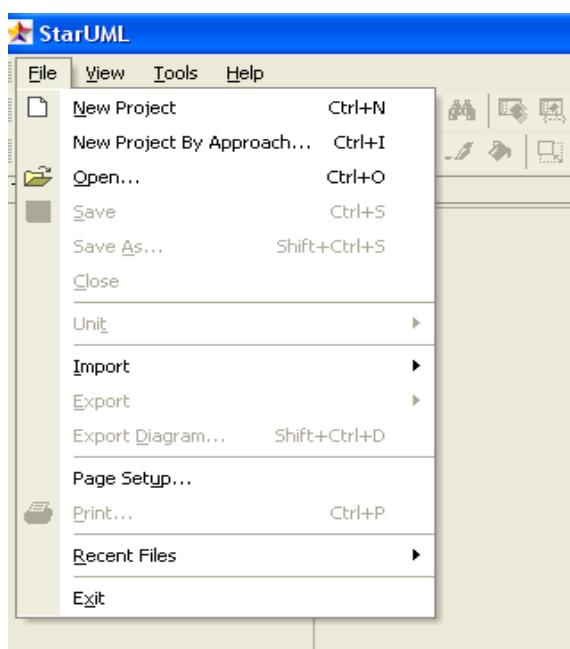
לאחר התקנת התוכנית והפעלה יתקבל המסך הבא:



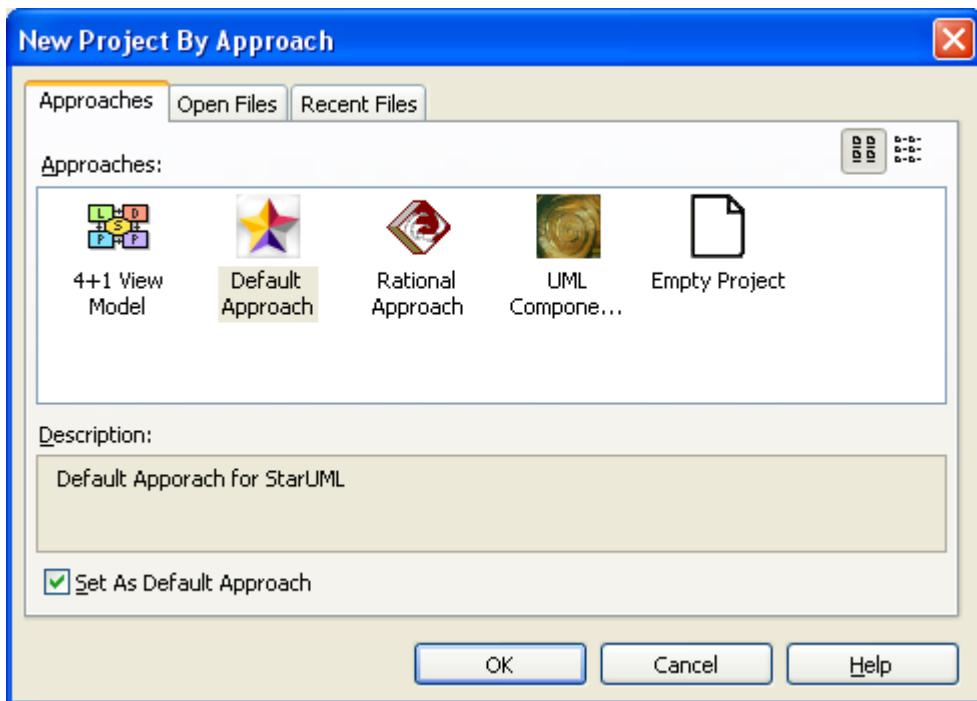
- **אלמנטים** – נמצאים בצד הימני של סביבת העבודה. מגירית אלמנטים אלה לשטח השירות נרכיב את הדיאגרמה הרצוייה. האלמנטים כוללים את כל המרכיבים הדרושים לשרטוט דיאגרמות UML.
- **שטח השירות** – חלקה המרכזי של סביבת העבודה.
- **מודלים** – נמצאים בצד הימני העליון של סביבת העבודה. המודלים מבטאים היבט ספציפי של המערכת הפיזית. ההיבט יכול להיות ניתוחי, עיצובי, או היבט מנוקדת מבטו של המשמש במערכת. כל מודל כולל בתוכו אוסף דיאגרמות רלוונטיות מכל הדיאגרמות הקיימות ב-UML. הנקודות ורמת הפירוט של המודלים נתונה לשיקולנו וניתן למחוק או להוסיף מודלים כרצונו.
- **הגדרות** – נמצאות בצד הימני התחתון של סביבת העבודה. כאשר נעמוד על דיאגרמה כלשהי או על רכיב מסוים בתחום דיאגרמה בשיטה השירות, נוכל לראות את כל ההגדרות הרלוונטיות עבורו בתיבת ההגדרות.
- **מסך הפלט** – חלקה התחתון של סביבת העבודה. כולל את פלט התוכנית.

7.3 יצרת פרויקט חדש

על מנת ליצור פרויקט חדש יש לבחור בתפריט File באופציה New Project by Approach



בעקבות הבחירה יתקבל המסלך הבא :



מסלול זה מאפשר לנו ליצור פרויקט ריק או פרויקט המכיל בסיס מסוים. אם נבחר בפרויקט ריק נדרש להגדיר בעצמו מודולים ובתוכם דיאגרמות שאנו מעוניינים לעצב. ואם נבחר ב-Rational Approach קיבל הגדרות של מודלים מוכנים עם דיאגרמות שעליינו לבנות. מטעמי נוחות, נעבוד עם Rational Approach. כموבן שניית להוסיף או לגרוע מודולים ודיאגרמות לפי צרכינו.

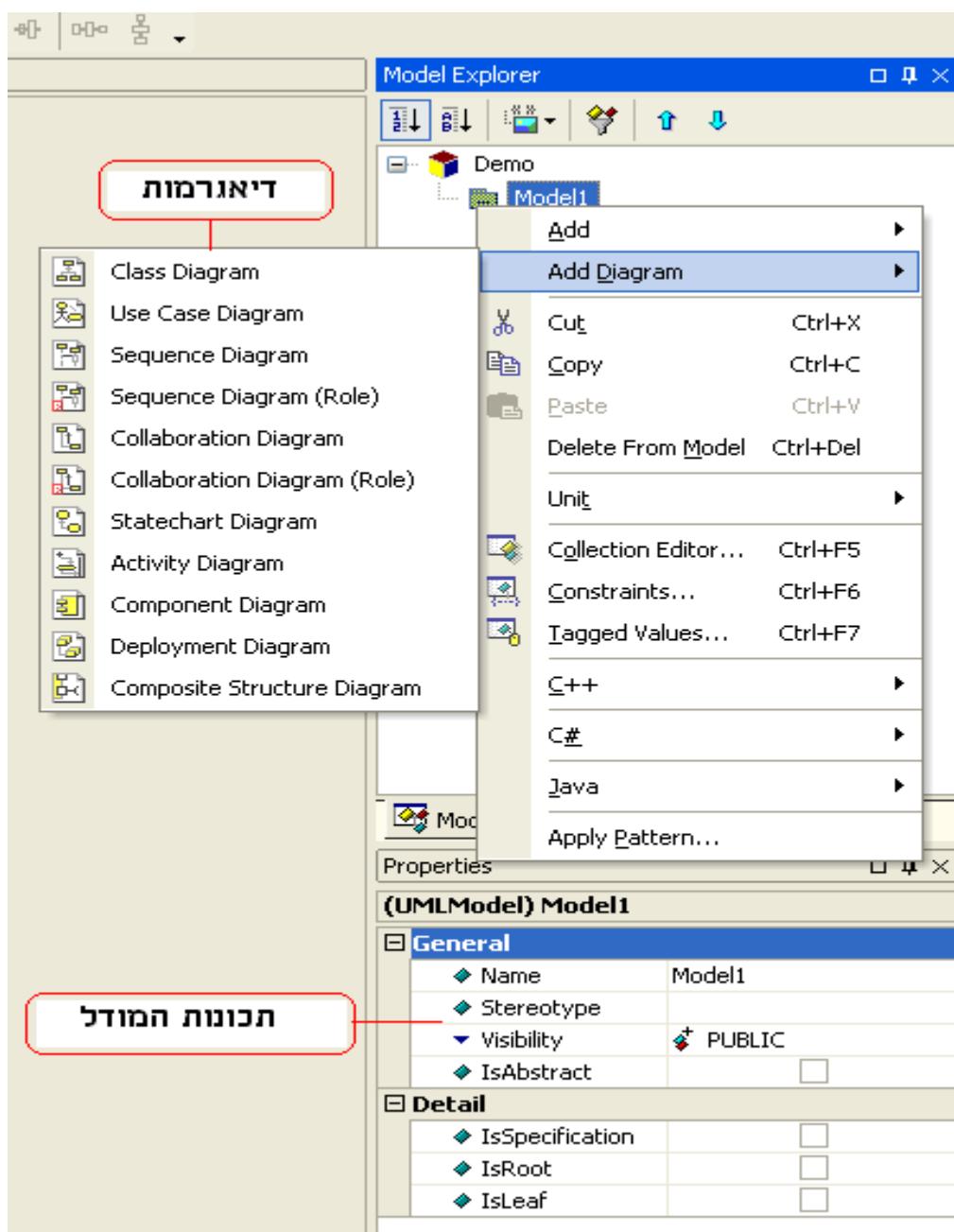
בבנייה דוגמה, אין צורך להביא לידי ביתוי את כל הדיאגרמות האפשריות. אלו נסתפק בשלושה מודולים עיקריים (שנוצרים במסגרת בחירת פרויקט מסוג Rational Approach). מודולים אלה כוללים :

- **היבט על השימוש במערכת** – כולל בתוכו דיאגרמות Use Case.
- **היבט עיצובי של המערכת** – כולל בתוכו Class Diagram ואפשר גם Object Diagram להמיר את ה-Class Diagram לקוד בשפת C# בלחיצת כפתור, וכך לחסוך זמן בקידוד ולהתמקד בעיצוב המערכת בצורה גרפית ו邏輯ית יותר.
- **היבט של תהליכי המערכת** – כולל בתוכו Activity Diagram, Sequence Diagram.

כל שנותר כעת הוא להוסיף לכל מודל את הדיאגרמות הרלוונטיות לו, ולעצבם בהתאם לעזרת תיבת האלמנטים.

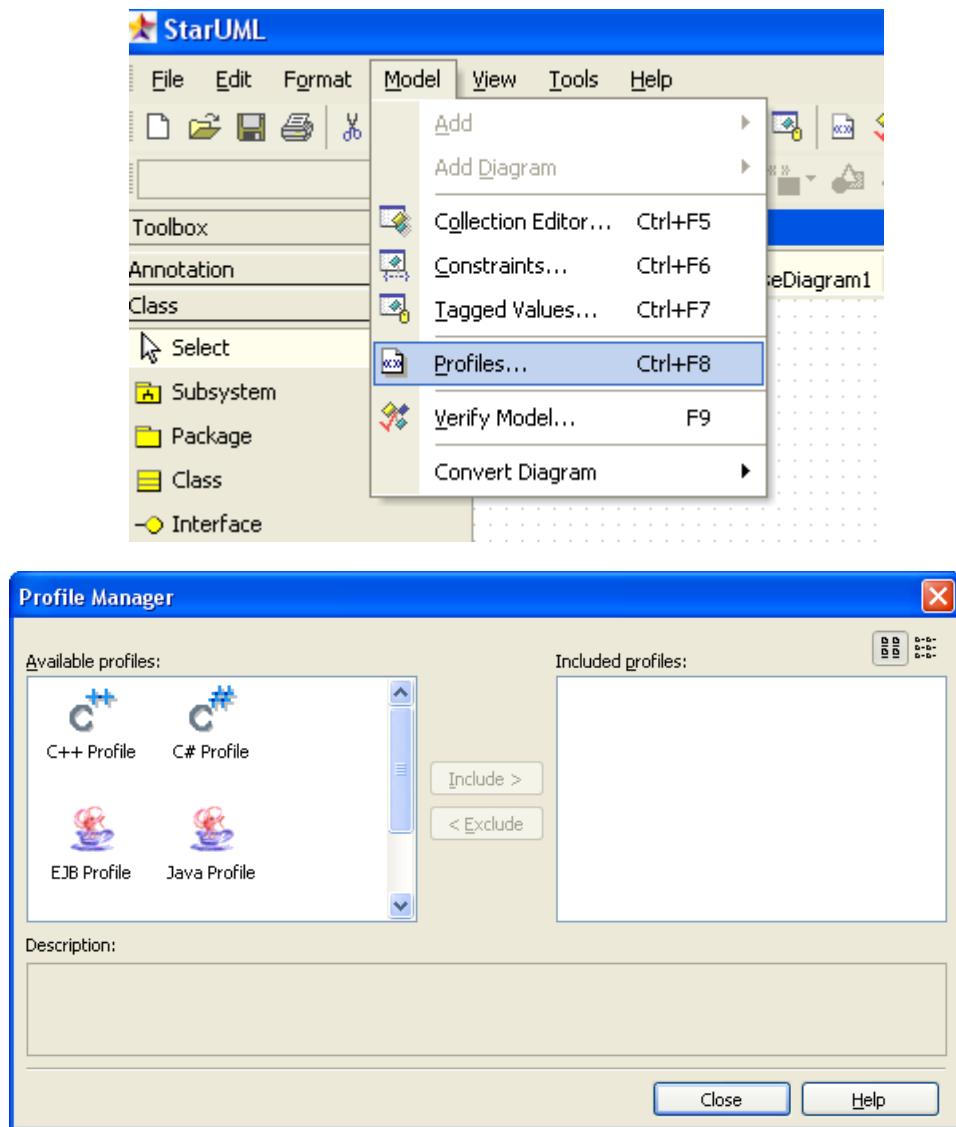
7.4 הוספת דיאגרמה למודל

בלחיצה על הכפתור הימני בעכבר בזמן שעומדים על המודל המתאים, ייפתח תפריט המאפשר הוספת דיאגרמה חדשה. בתפריט זה יש לבחור את סוג הדיאגרמה הרלוונטי למודל שלנו מבחינת היבט שלו על המערכת. כך נראה מסך הוספת הדיאגרמה למודול מסוים :

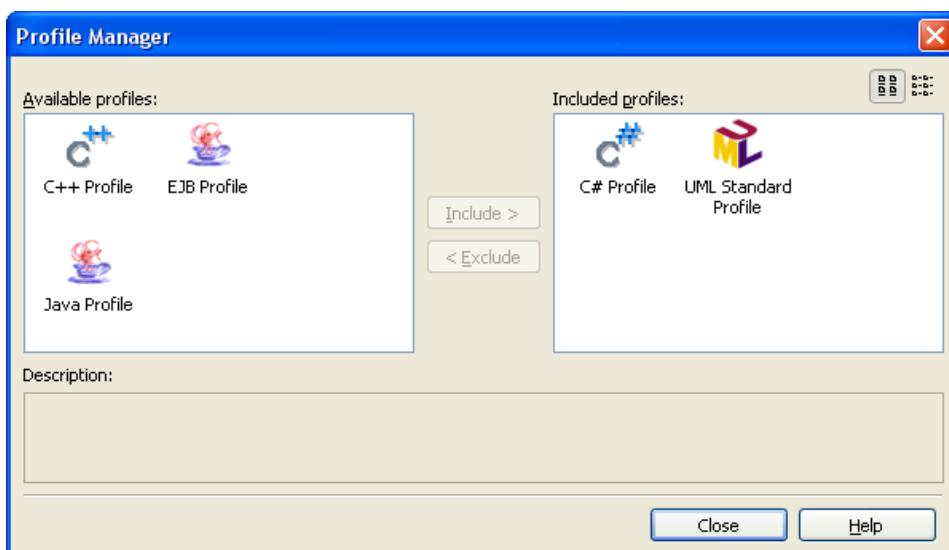


7.5 הוספת C# Profile

לפני עיצוב הדיאגרמות רצוי להוסיף תחילת לפרויקט את C# Profile, על מנת שזמן בינוי הדיאגרמה יוכל לבחור בשמות המוגדרים בשפת C#. למשל, כאשר נעצב מחלקה חדשה ונרצה להויסיף לה member חדש, נctrוך לקבוע ל-member זה את טיפוסו (int, byte וכו'). על מנת לקבל את השמות המקובלים לטיפוסים כפי שקיים בשפת C#, נחוצה ההוספה של ה-C# Profile. תהליך ההוספה כולל את המרכיבים הבאים:



ב- C# Profile Manager, כפי שמתואר
וכן ב-UML Standard Profile רצוי לבחור הן ב- .
במסך הבא :



מעתה הדיאגרמות שנעיצב יכללו תמייהה בשמות הנוהגים בשפת C#, דבר הכרחי על מנת ליצא את הדיאגרמות לקוד בשפת C#.

7.6 כיצד להמשיך מכאן ?

כעת, לאחר הסבר בסיסי זה על השימוש בתוכנה, עלייכם ללמוד את יתר הפרטיהם הקטנים : כיצד לעצב מחלקה, עיצוב ממשק, בניית Use Cases Diagrams וכו'. לשם כך, עלייכם ללמוד באמצעות האתר של StarUML באינטראנט (שם ישנים הסבירים מפורטים מאוד) כיצד לבנות את הדיאגרמות השונות, כיצד להמיר לקוד בשפות שונות (במקרה שלו שפה C#), וכייד להפיק מסמכים תיעוד שונים באופן אוטומטי.


קראו ותרגולו את הסבירים הנמצאים באתר האינטראנט של פרויקט
[Star UML](http://staruml.sourceforge.net/en/)

7.7 הכוונה להבנת הרצאה

7.7.1 נושאים שיש לסקור בהרצאה

הסבר כללי על אופן השימוש בתוכנה (כפי שהודגש כאן).

תהליך בנייה של דיאגרמות, בפרט :

Use Case Diagram .1

Class Diagram (כולל מחלקות, מנשכים, ירשות, קשרי has-a ו-is-a) .2

Activity Diagram .3

Sequence Diagram .4

המרת Class Diagram # לשפת C# לקוד במשפט •

יצירת קובצי מסמכים מ- Use Case Diagram (דבר שיכול להיות לעזר בזמן כתיבת פרויקט הגמר).

הדגמת תהליך – יצירת Class Diagram – Reverse Engineering מקוד נתון.

רצוי שכל הדיאגרמות שיודגמו יהיו על דוגמה למערכת מסוימת שברצוננו להקים.

הדגמת דיאגרמות נוספות (State Chart Diagram ועוד) במסגרת מגבלות הזמן שיישאר.

פרק 8 – דפוסי עיצוב (Design Patterns)

1. הקדמה

עיצוב מערכת מונחית עצמים היא מלאכה סבוכה. לא די להגיע למימוש שעובד, צריך להגיע למימוש שקל לתחזקה ולהרחבה. עיצוב שתומך בעקרונות של תכנון מונחה עצמים, כגון: הסתרת מידע, שימוש בחזרה וכדומה, מסייע להגיע למימוש כזה. אולם קשה לחזות מראש מה כדאי להסתיר, ובאיזה עצבן את הקשרים בין המחלקות על מנת להגיע למימוש כזה.

לא מפתיע לגלוות שמנתחי מערכות ומתכננים חסרי ניסיון מגאים לפתרונות פחות מוצלחים ממנתחי מערכות ומתכננים בעלי ניסיון. כאשר מנתח מערכות או מתכנן ותיק ניתן לעצב מערכת חדשה, הפתרון שהוא מציג מורכב בחלוקת מפתרונות מוצלחים שהוא יצר או הכיר בעבר. תארו לעצמכם שהיינו יכולים להעניק לכל מנתח מערכות ותוכנת מתחילה את הידע הנכבר זהה בהתחלה דרכו. זה הרעיון המרכזי שעומד מאחורי דפוסי עיצוב.

כאשר סוקרים פתרונות תכנון מוצלחים, במערכות שונות, ניתן למצוא דפוסי עיצוב שחוזרים על עצמם. דפוסים מוצלחים אלה יכולים להוות בסיס לעיצוב מערכות מונחיות מוצלחות חדשות. מה שנדרש לנו הוא צורה מסוימת לרשום "התורה שבעל פה" המגיעה אלינו מפי המנוסים.

אבל לפני הכל, קצת היסטוריה!

נושא דפוסי העיצוב החל לרקום עור ונגידים בסוף שנות השבעים של המאה ה-20 על ידי קריסטופר אלכסנדר (Christopher Alexander), שלמרבה ההפתעה לא היה איש מחשבים אלא ארכיטקט. הוא הציג את המושג בהקשר של תכנון ארכיטקטוני, ואף פרסם כמה ספרים מעניינים בנושא, וביניהם:

- A Pattern language, Publisher: Oxford University Press, ISBN: 0195019199, (1976)
- The Timeless Way of Building Oxford University Press; ISBN: 0195024028; (1979)

נוסף לדפוסי העיצוב הרבים המופיעים בספרים אלה, המחבר מציג בפני הקורא את הרציונל מאחורי רעיון איסוף והיעוד דפוסי העיצוב.

במדיי המחשב, מושג זה קיבל ד晖פה בכנס '87 OOPSLA בעקבות מאמר מאת Kent Beck ו-Ward Cunningham שהוצע במסגרת הכנס. הנושא החל לעורר עניין רב אצל אנשים נוספים כדוגמת Erich Gamma, שחקר את הנושא בין השנים 1988-1991, Erich Gamma, John Vlissides, Ralph Johnson, Richard Helm, Erich Gamma Design פרשו את הספר:

חבורה זו נודעה בכינוי Gang of Four או בקיצור GoF. מאז ועד היום פורסמו מאמרים רבים בנושא.

אחד מספרי הלימוד בסדנה הוא הספר הקלסי של GoF, שכל מהנדס תוכנה ראוי שיהיה לו "על המדף". הספר סוקר 23 דפוסים שונים. במסגרת הרצאות בסדנה יילמדו מרבית הדפוסים, עם דגש על השכיחים שבhem.

בහמץ תינתן הכוונה לקריאת הספר ודוגמה מלאה לאחד הדפוסים כולל דוגמת קוד. דוגמה זו תנחה אתכם כיצד למדוד את יתר הדפוסים, ובעיקר כיצד לבנות את הרצאה המיועדת לנושא זה.

8.2 הכוונה לקריאת הספר של GoF

דפוסי העיצוב נחלקים לשלווש קטגוריות:

- **דפוסי יצירה** – דפוסים העוסקים בתהליך ייצור אובייקט.
- **דפוסי מבנה** – דפוסים העוסקים בהרכבה של מחלקות ואובייקטים.
- **דפוסי התנהגות** – דפוסים העוסקים באופן שבו אובייקטים או מחלקות מתקשנות וחולקות אחריות ביניהם.

הספר מורכב מ- 5 פרקים:

8.2.1 חלק א – מבוא

- **פרק 1** – פרק מבוא, שكريאוו חשוב. הפרק מגדר מה הם דפוסי עיצוב, וסוקר מושגים ונוסאים שיבאו לידי ביטוי במרבית הדפוסים בהמץ.
- **פרק 2** – פרק זה מציג ניתוח של שימוש מעבד לתמלילים. הפרק מציג עיצוב ראשון למעבד לתמלילים, ובהמשך מוסיף שינויים ותוספות (כגון תמייה במספר מסמכים, שינוי פונטיים ועוד) וمعدכן בהתאם את העיצוב. חשוב לקרוא פרק זה, על מנת לקבל מושג על אופן החשיבה והגישה לפתרון הבעיה.

8.2.2 חלק ב – סקירה מלאה של כל דפוסי העיצוב

בחלק זה נסקרים כל דפוסי העיצוב לפי הקטגוריה אליה הם שייכים. כל דפוס מסוים בצורה מפורשת. ההסביר כולל: מוטיבציה לשימוש בדף, תרשימים UML המתאר את הדפוס, בעיות נפוצות בהם הדפוס נדרש, דוגמת קוד למימוש הדפוס, המלצות לשימוש נכון בדף.

- **פרק 3 – סקירת דפוסי יצירה.**
- **פרק 4 – סקירת דפוסי מבנה.**
- **פרק 5 – סקירת דפוסי התנהגות.**

לפני קראת הספר, חשוב לרענן את הזכורון בנושאים הבאים: העקרונות הבסיסיים של OOD כפי שנלמדו בהרצאה בנושא OOA/D, בפרט העקרונות הבאים: OCP, ISP, DIP, תרבותה לעומת ירושה, תכנות למנשכים ולא למימושים, צימוד נמוך (decouple) בין אובייקטים, הדיקות (cohesion). וכן בתרשימי UML, בפרט ב- : Class Diagram, Association, Delegation, Composition. כל אלה יבואו לידי ביטוי, הלאה למעשה, בדפוסי העיצוב השונים.

8.3 סיווג דפוסי העיצוב

את דפוסי העיצוב ניתן לסוג לפי :

- **מטרת הדפוס** – האם הדפוס יצרתני, מבני, או התנהגוני.
- **מרחב הדפוס** – האם הדפוס הוא למרחב המחלקה או למרחב האובייקט.
- **מרחב המחלקה** – דפוסים המתמקדים בקשרים הקיימים בין מחלקות ומחלקות היורשות מהם. הקשרים נובעים בעיקר בעקבות ירושה, ולכון הקשרים הם סטטיים.
- **מרחב האובייקט** – דפוסים העוסקים בקשרים בין אובייקטים, ולכון הקשרים יותר דינמיים (נקבעים בזמן ריצה). מרבית הדפוסים משתמשים למרחב זה.

את סיווג דפוסי העיצוב על פי מטרתם ועל פי המרחב לו הם שייכים ניתן לסכם בצורה טבלה כך :

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

8.4 הכוונה להבנת ההרצאה

8.4.1 נושאים שיש לסקור בהרצאה

במסגרת ההרצאה יילמדו מגוון דפוסי עיצוב, בפרט השכיחים שבהם. ההרצאה צריכה לכלול הדגמה של כ-3 דפוסים מכל קטגוריה (יצירה, מבנה, ותנהגות). צורת הצגת הדפוסים צריכה להיות ברוח הדוגמה שותוצג בהמשך על אחד מהדפוסים (הדפוס [Decorator](#)).(Decorator

בפרט ייסקרו דפוסי העיצוב השכיחים הבאים :

Abstract Factory, Adapter, Composite, Factory Method, Observer, Strategy, Template Method.

8.4.2 ביבליוגרפיה

- Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides, "Design Patterns, Elemenets of Reusable Object Oriented Software", 1995.
- "Head First Design Pattern", Freeman & Freeman, O'REILLY,2004.
- <http://www.csc.calpoly.edu/~dbutler/tutorials/winter96/patterns/>
"Welcome to the wonderful world of Design Patterns". A tutorial by Brian T. Kurotsuchi.
- <http://g.oswego.edu/dl/pd-FAQ/pd-FAQ.html>
Patterns FAQ maintained by Doug Lea.
- <http://www.tml.hut.fi/~pnr/Tik-76.278/gof/html/>
UML models for all the patterns that appear in the gang of four catalogues.
- <http://www.industriallogic.com/papers/learning.html>
A learning guide to design patterns.
- <http://www.hillside.net/patterns/books/#Gamma>
A list of design patterns books.
- http://www.ciol.com/content/technology/sw_desg_patt/
Software design patterns.
- <http://www.ddj.com/articles/1998/9806/>
Automating Design-Pattern Identification.
- <http://davidvancamp.com/oodworkshop.html>
NASA Vision2000 Object-Oriented Design Workshop

8.5 הסבר מפורט ודוגמה למימוש של דפוס עיצוב

על מנת להמחיש כיצד ללמד דפוס עיצוב מסוים, וכייד רצוי להכין את ההרצאה בנושא, ניתן כאן הסבר מكيف בתוספת דוגמה לדפוס העיצוב Decorator (נקרא גם Wrapper). דפוס זה שייך לקטגוריה **דפוסי מבנה**, הואיל והפתרון שהוא מציע זו בדרך שבה מחלקות צרכות להיות מורכבות.

ההסבר יכלול דוגמה של בעיה. בתחילת נציג פתרון ל쿄 ונדון בחסרונותיו. לאחר מכן נציג את דפוס העיצוב Decorator ונסביר כיצד לישמו על הבעיה על מנת להשיג פתרון מוצלח יותר.

נזכיר שהדוגמה לדוגמה מהספר (ספר נוסף ומומלץ מאוד) :

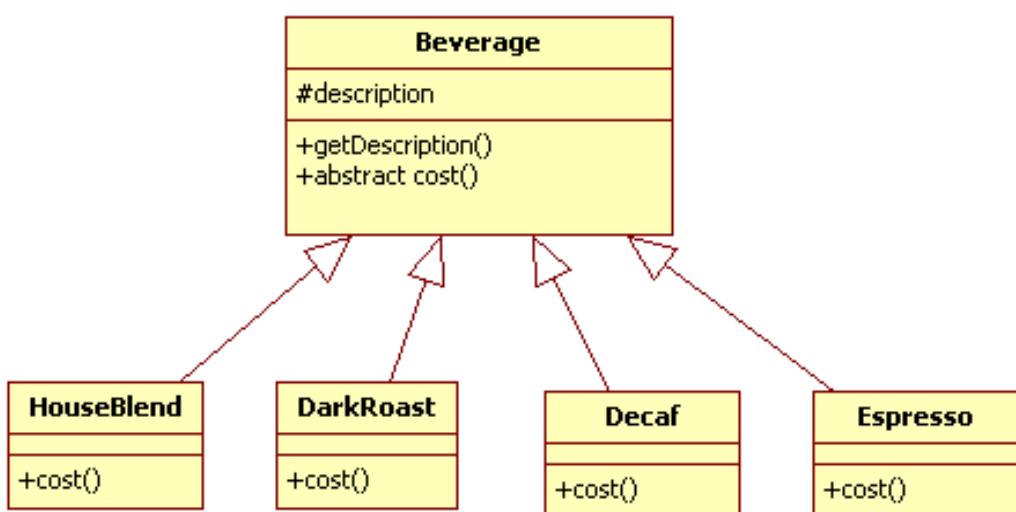
"Head First – Design Patterns", Freeman & Freeman, O'REILLY. 2004.

8.5.1 דוגמה לדפוס העיצוב Decorator

לרשות בתיה הקפה סטארבאו יש סניפים רבים. לאחר שהרשת גָּדלה במתירות, ומספר הסניפים שלה הולך וגדל, סטארבאו זוקה לשדרוג מערכת הזמן הממוחשבת שלא כדי לעמוד במנועו הגודל של משקאות אותם היא מציעה.

המצב הנוכחי של מערכת הזמן בנוי ממחלקה אב אבסטרקטית של משקה, ממנה יורשים מחלקות המייצגות את סוגי המשקאות השונים. בכל משקה ישנה מתודה המחזיר את עלותו, וכן הוא ירוש תוכנה ממחלקת האב המייצגת את תיאור המשקה.

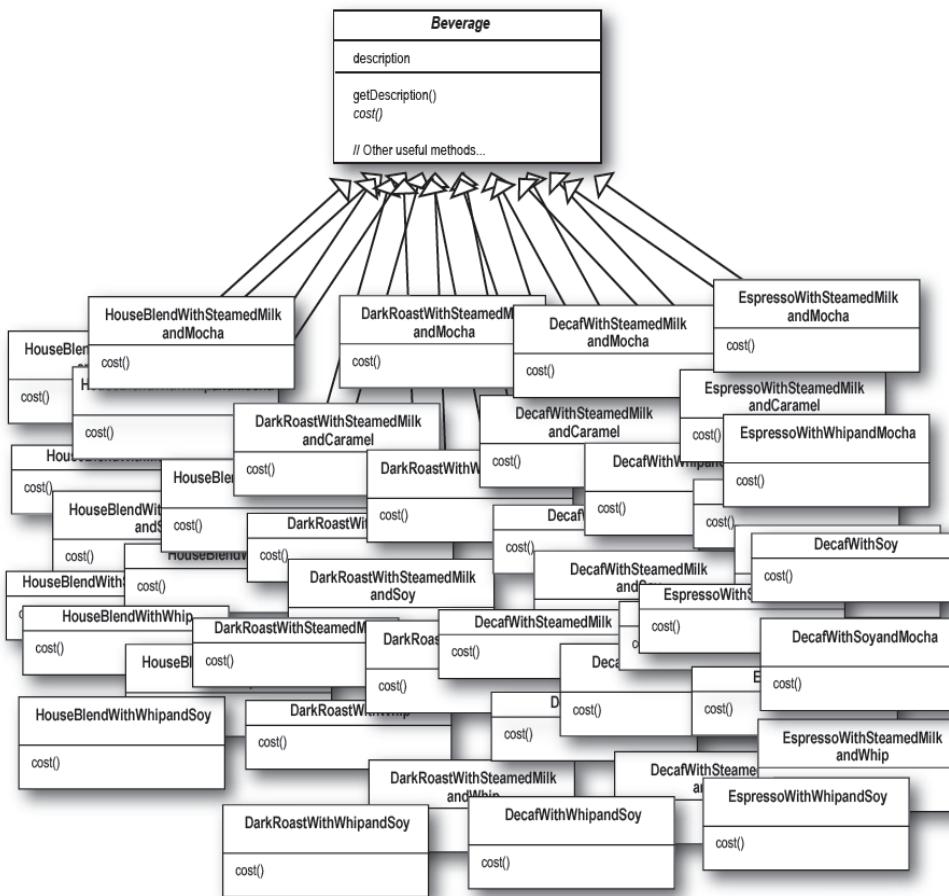
כך נראה תרשימים המערכת הנוכחיות :



כל משקה ניתן להוסיף תוספות שונות, כגון: חלב מורתה, חלב סוויה, מוקה, ציפוי של חלב מוקצף וצדומה. לכל תוספת כזו יש עלות משלה המתווספת לעלות המשקה עצמו.

כעת נדרש להכניס המערכת הזרנוקה הממוחשבת את התמיכה בתוספות השונות. הפתרון שהוצע נראה

כذ:



מה קיבלנו? "התפוצצות של מחלקות".

הפתרון דלעיל יוצר מחלקות חדשות כמספר הצירופים האפשריים של משקאות ותוספות. ברור שפתרון כזה אינו מתקין על הדעת, מאחר שייהיה קשה לתחזק פתרון זה במקרה של שינויים שונים. למשל: אם מחיר של תוספת ישנה, נצטרך לעדכן את המחיר בכל המחלקות הכוללות תוספת זו. או אם יתוסף משקה מסווג חדש (למשל iced tea), נצטרך להוסיף מחלקות חדשות לקוד עבור המשקה החדש עם כל התוספות האפשריות. כמו כן, אם ללקוח ירצה לשאקה את אותה תוספת מספר פעמים, למשל פעמיים מוקה, בפתרון הנוכחי אין אפשרות באפשרות זו.

איך יוצאים מהתסבוכת הזו?

כאן נכנס לתמונה דפוס העיצוב: **Decorator Pattern**. היכן?

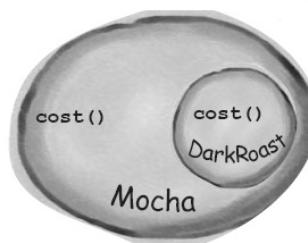
ניקח אובייקט של משקה מסוים ונעטוּף (נקשת decorate) אותו בתוספות שונות בזמן ריצה!!! (להזיכרכם, זה ניתן למימוש על ידי שימוש בהרכבה, המבוטא על ידי Association בתרשיimi UML). כשנפעיל את Method cost במחלקה העוטפת (מחלקת התוספה) לחישוב מחירו של המשקה (עם

התוספה), מותודה זו תפעיל גם את מותודה cost של המשקה אותו היא עוטפת (יבוצע באמצעות Delegation). נמchioז זאת בצורה גרפית:

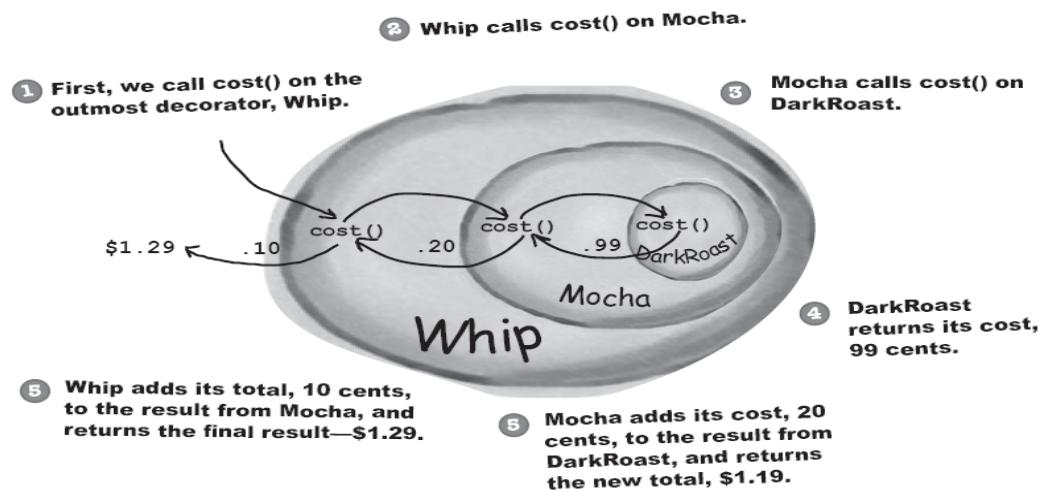
ניח אובייקט של משקה מסויים, לדוגמה המשקה DarkRoast.



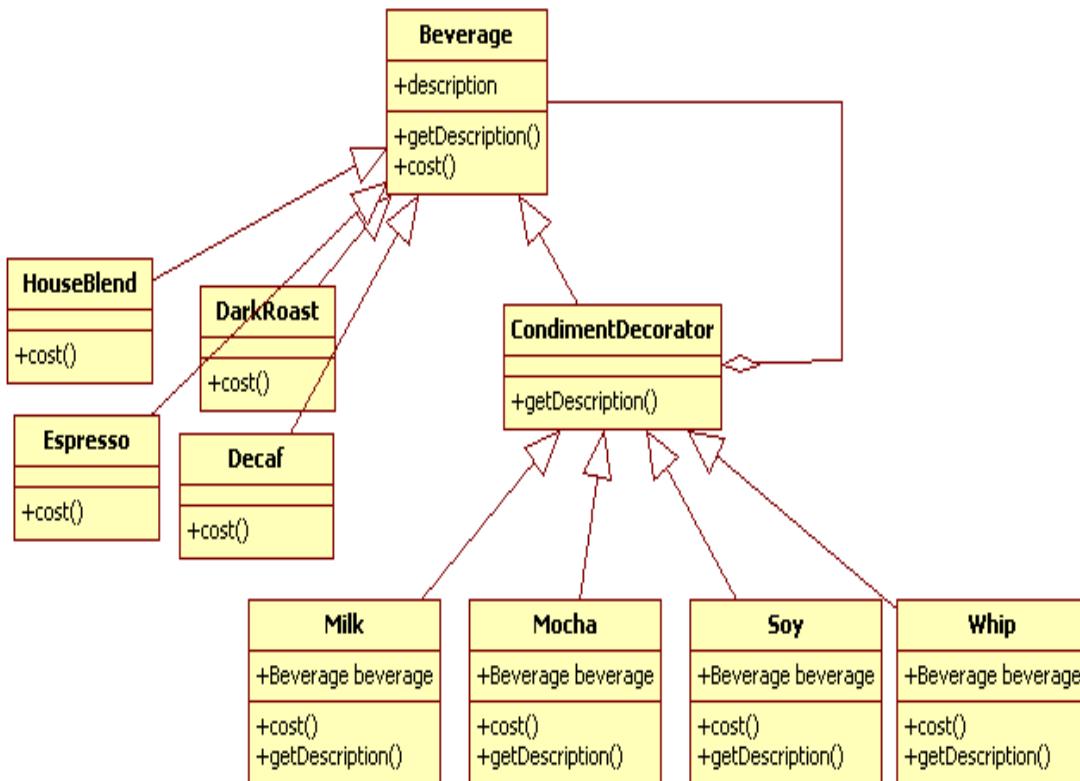
נ nich כי הלקוח מעוניין להוסיף למשקה תוספה של מוקה. לכן, ניצור אובייקט מוקה שייעטוּף בתוכו את אובייקט המשקה.



עטוף את המשקה (משקה+מוקה) גם בקצת (whip) ונחשב את העלות הכוללת: נפעיל את מותודה cost() באובייקט הקישוט החיצוני ביותר, והוא – באמצעות העברת הקריאה לмотודה cost() של האובייקטים אותם הוא עוטף – יחשב את הועלויות של כל שאר התוספות ויווסף אותן לעולות שלו:



איך מיישמים זאת?
נביט תחילה בתרשים הבא המתאר את הפתרון המוצע:



שים לב לפתרים הבאים הנובעים מהתרשים :

- קיימת מחלוקת אב אבסטרקטית של משקה, ממנו יורשים הנו סוגי המשקאות (המשקאות הקונקרטיים), והן מחלוקת אבסטרקטית של תוספת.
- מהחלוקת האבסטרקטית של תוספת יורשים התוספות השונות (התוספות הקונקרטיות).
- **שימוש לב** –חלוקת האבסטרקטית של תוספת לא רק יורשת מחלוקת משקה, אלא בנוסף גם עוטפת בתוכה (הרכבה) אובייקט מסווג משקה.

שאלה: מדוע נחוצה הנו ירושה והן הרכבה?

תשובה: ההרכבה נחוצה לצורך היכולת לעטוף. אך מדובר חשוב שאובייקט תוספת יירש משקה? הסיבה לכך היא שתוספת העוטפת בתוכה משקה, **חייבת** להיות עצמה סוג של משקה על מנת שנוכל לעטוף גם אותה בעוד תוספות.

כמו כן, ראוי לשים לב כיצד באים לידי ביטוי העקרונות החשובים של תכניות עיצוב כפי שקרהתם עלייהם בפרק הראשון של הספר :

- **עיקרונו OCP** –חלוקת סגורה לשינויים אך פתוחה להרחבות.
- העדפת הרכבה על פני ירושה.

8.5.2 מימוש הפתרון בעזרת Decorator Pattern

נתאר את שלבי כתיבת התוכנית על פי התרשים לעיל:

- כתיבת המחלקה האבstractית Beverage לייצוג משקה :

```
public abstract class Beverage
{
    protected String _Description;
    public string Description
    {
        get
        {
            return _Description;
        }
        set
        {
            _Description = value;
        }
    }
    public abstract double cost();
}
```

- נגידר את המחלקה Espresso המייצגת משקה קונקרטי :

```
public class Espresso : Beverage
{
    public Espresso()
    {
        Description = "Espresso";
    }

    public override double cost()
    {
        return 1.99;
    }
}
```

המחלקה מimplements את המתודת האבstractית cost וכן קובעת את תיאורו של המשקה.



- נגידר את המחלקה האבסטרקטית CondimentDecorator המייצגת תוסף למשקה:

```
public abstract class CondimentDecorator : Beverage
{
    protected Beverage _beverage;

    public Beverage beverage
    {
        get
        {
            return _beverage;
        }
        set
        {
            _beverage = value;
        }
    }
}
```

שימושו לב לחלק החשוב ביותר בתבנית עיצוב זו: המחלקה גם יורשת מהמחלקה משקה, וגם עוטפת בתוכה אובייקט מסווג משקה. האובייקט הנעטף ייקבע בזמן ריצה!!!

- נגידר כעת את המחלקה Mocha לייצוג תוסף קונקרטי מוקה למשקה:

```
public class Mocha : CondimentDecorator
{
    public Mocha(Beverage beverage)
    {
        this.beverage = beverage;
        Description = this.beverage.Description + ",  
Mocha";
    }

    public override double cost()
    {
        return beverage.cost() + 0.5;
    }
}
```

מחלקה זו בניי המקבל כפרמטר אובייקט מסווג משקה שיש לעטוף. אובייקט זה מוכנס (על ידי שימוש ב-*Property*-*beverage*) למשתנה *beverage* שהוכן מראש במחלקה האבסטרקטית. כמו כן, תיאورو של האובייקט *Mocha* גם אובייקט מסווג משקה בעקבות הירושה של תוסף משקה) נקבע להיות תיאورو של האובייקט הנעטף בתוספת התיאור "Mocha".

- באופן דומה, נגידר את מחלקה **Whip** לתיאור תוסף קונקרטי של הקצפה למשקה.

```
public class Whip : CondimentDecorator
{
    public Whip(Beverage beverage)
    {
        this.beverage = beverage;
        Description = this.beverage.Description + ",  
Whip";
    }

    public override double cost()
    {
        return beverage.cost() + 0.2;
    }
}
```

- נכתוב כעת את התוכנית הראשית המדגימה את השימוש בתבנית העיצוב.

התוכנית מגדרה אובייקט משקה מסווג Espresso ומדפיסה את פרטיו. לאחר מכן מגדרה אובייקט נוסף של משקה Espresso ומוסיפה (מקשחת) אותו בתוספת כפולה של מוקה, ובהקצפה.

```
class Program
{
    static void Main(string[] args)
    {
        Beverage beverage = new Espresso();
        Console.WriteLine("{0} ${1}",
        beverage.Description,beverage.cost());

        Beverage beverage2 = new Espresso();
        // Mocha Decorator
        beverage2 = new Mocha(beverage2);
        //more Mocha Decorator
        beverage2 = new Mocha(beverage2);
        // Whip Decorator
        beverage2 = new Whip(beverage2);
        Console.WriteLine("{0} ${1}",
        beverage2.Description,beverage2.cost());
    }
}
```

שימוש לב לאופן בו מוסיפים לשקה את התוספות. יכולת זאת מושגת הודות לירושא ולעטיפה בה משתמשת המחלקה לייצוג תוסף. ללא הירושא בנוסף, דבר זה לא היה מותאפשר.

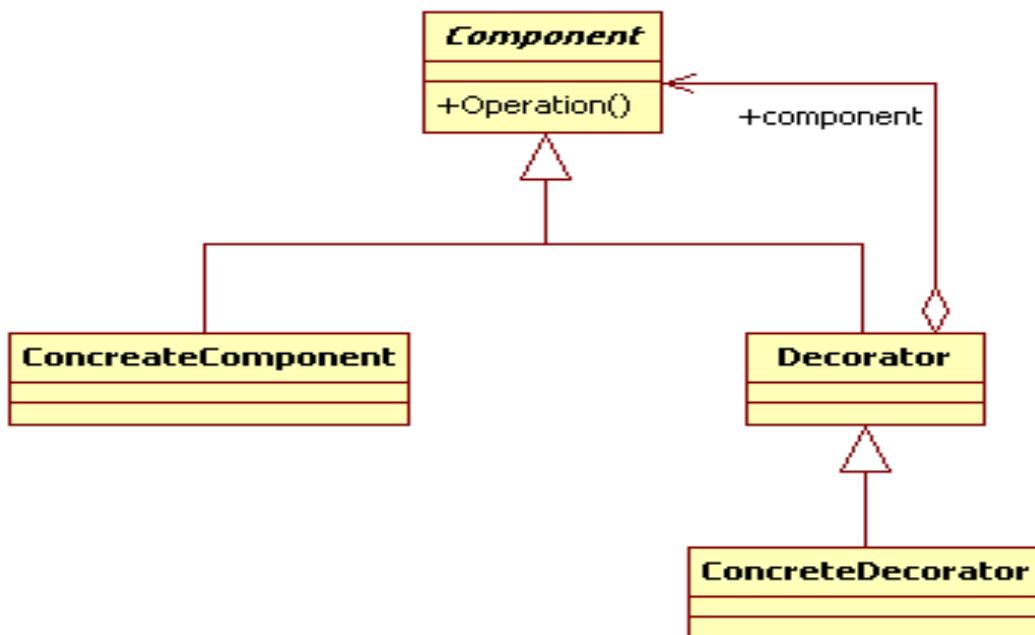
פלט התוכנית המתקבל :

Espresso \$1.99

Espresso ,Mocha ,Mocha ,whip \$3.19

8.5.3 סיכום דפוס העיצוב Decorator

להלן תרשים ה-UML הרשמי של דפוס העיצוב Decorator :



תרשים זה תואם במדויק לבניית הפתרון שיוושמה עבור דוגמת המשקאות של רשת סטארבז באופן הבא :

- ה-Component מייצג את המשקה.
- ה-Decorator מייצג את התוספת למשקה.
- ה-ConcreteComponent מייצג את המשקאות השונים.
- ה-ConcreteDecorator מייצג את התוספות השונות.

המודיבציה לשימוש ב-Decorator Pattern: כאשר מעוניינים לצרף למחלקה פונקציונליות נוספת באופן דינמי.

8.6 תרגול נוסף

עכבו את דיאגרמת ה-UML של הפתרון עבור מערכת ההזמנות של רשות סטארבז באמצעות תוכנת StarUML שנלמדה במסגרת הרצאה בנושא UML. לאחר השלמת שרטוט הדיאגרמה, בצעו יצוא שלה לקוד ב-#C. הוסיפו תוכן מתאים לכל מトודה. כמו כן, הריצו את התוכנית.

תרגלו זאת גם על דפוסי עיצוב נוספים.

שימוש לב: התוכנה StarUML כוללת בתוכה את כל דיאגרמות ה-UML של כל דפוסי העיצוב, כך שבבחירה שכבר ימniaת ניתן לקבל שרטוט מוקן של דפוס עיצוב.



File #0003243 belongs to Roei Daniel- do not distribute

יחידה 6 – טכנולוגיות .NET



File #0003243 belongs to Roei Daniel- do not distribute

פרק 9 – XML

9.1 הקדמה

הרצאה זו עוסקת בנושא XML, שהוא פורמט לשימירת נתונים. פורמט זה יתרוניות רבים, והוא שימושי ורלוונטי לנושא של עבודה מול בסיסי נתונים, שליחת מידע בראש, קבצי קונפיגורציה, ואף בפיתוח ממשק משתמש גרפי באמצעות WPF (שם נעשה שימוש בקובץ בפורטט XAML). פורטט זה שימושי מאד גם באתר אינטרנט המיצאים מידע בפורטט XML (שירותי RSS). שפת C# מספקת מחלקות לעובדה עם קבצי XML, המאפשרות ייצור, קריאה ועיבוד של קבצי XML. בנוסף ניתן לעבוד מול קבצי XML באמצעות LINQ to XML (LINQ to XML) ולבצע על קובץ ה-XML פעולות של חוספה, עדכון ומחיקה של מרכיבים בו, ואף הרצת שאילתות בחירה. באמצעות סריאלייזציה ניתן לשמור גרען של אובייקטים לתוכן קובץ XML.

9.2 הכוונה להכנות ההרצאה

9.2.1 נושאים שיש לסקור בהרצאה

- סקירת הנושא.
- מבנה לוגי ותחביר השפה + דוגמאות.
- אלמנטים ומאפיינים של אלמנטים (Elements, Attributes).
- Validity – בדיקת תקיפות של מסמך XML. לדוגמה : XML Schema, DTD .
- מחלקות בשפת C# לטיפול במסמכים XML. הדוגמת ניתוח (parsing) למסמך XML, הדוגמה לייצרת מסמך XML, והדוגמה להוספה מידע למסמך XML קיימים.
- XML Path Language (XPath) – כלי סטנדרטי לאיתור אינפורמציה המוחשנת במסמך XML או במבנהו.
- Query Language by W3C – XQuery
- Extensible Stylesheet Language Transformations (XSLT) – שפה לתיאור כיצד תוכן של מסמך XML אמרוי להיות מוצג (לדוגמה בדף אינטרנט).
- LINQ to XML
- סריאלייזציה של אובייקטים לקובץ XML



9.2.2 ביבליוגרפיה

- World Wide Web Consortium (W3C)
<http://www.w3.org/>
- Online community for XML-related standards
<http://www.xml.org/> (join forums at Oasis)
- Microsoft XML Developer Center
<http://msdn2.microsoft.com/en-us/xml/default.aspx>
- <http://www.topxml.com/xml/learnxml.asp>
- Melton, Jim. "Querying XML: XQuery, XPath, and SQL/XML in Context." San Francisco, Calif. Morgan Kaufmann, 2006. ISBN 1-55860-711-0
- Walmsley, Priscilla, "Definitive XML Schema". Upper Saddle River, NJ: Prentice Hall PTR, 2002. ISBN 0-13-065567-8
- Holman, G. "Definitive XSLT and XPath". Upper Saddle River, NJ: Prentice Hall PTR, 2002.
- Walmsley, Priscilla. "XQuery", Farnham: O'Reilly, 2006. ISBN 0-596-00634-9
- פרקים 20, 24 בספר הלימוד

3.9 דוגמה לעבודה עם קובץ XML

להלן דוגמה המדגימה ייצירת קובץ XML פשוט. הקובץ שומר מידע על מכוניות.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Xml.Linq;

namespace XMLDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            XElement document = new XElement("Cars",
                new XElement("Car", new XAttribute("Manufacture", "Ford"),
                    new XElement("Model", "Mustang"),
                    new XElement("AvailableColors",
                        new XElement("Color", "Red"),
                        new XElement("Color", "Black") ) ) );
            document.Save("Cars.xml");
        }
    }
}
```

הסביר:

התוכנית יוצרת אלמנט שורש (Cars root element), תחתיו נוצר אלמנט בשם Car עם מאפיין (attribute) המתאר את יצרן הרכב. תחת האלמנט Car נוצר תת-אלמנט של Model ותת-אלמנט נוסף של AvailableColors הכלל תתי-אלמנט של Color. לבסוף התוכנית שומרת את ה- XML שיוצרה לקובץ בשם Cars.xml שנראה כך:

```
<?xml version="1.0" encoding="utf-8"?>
<Cars>
    <Car Manufacture="Ford">
        <Model>Mustang</Model>
        <AvailableColors>
            <Color>Red</Color>
            <Color>Black</Color>
        </AvailableColors>
    </Car>
</Cars>
```

דוגמה נוספת:

התוכנית הבאה מדגימה טעינה של קובץ XML שנוצר מהדוגמה הקודמת, והוספה מידע על שתי מכוניות נוספות לקובץ. לבסוף נשמרים הפרטים לקובץ חדש בשם CarsModified.xml;

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Xml.Linq;

namespace XMLDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            XDocument document = XDocument.Load("Cars.xml");
            XElement elementToAdd =
                new XElement("Car", new XAttribute("Manufacture", "BMW"),
                            new XElement("Model", "X"),
                            new XElement("AvailableColors",
                                new XElement("Color", "White"),
                                new XElement("Color", "Grey"),
                                new XElement("Color", "Blue")
                            )
                );
            document.Descendants("Cars").First().Add(elementToAdd);
            XElement elementToAdd2 =
                new XElement("Car", new XAttribute("Manufacture", "Renault"),
                            new XElement("Model", "Clio"),
                            new XElement("AvailableColors",
                                new XElement("Color", "Red"),
                                new XElement("Color", "Black"),
                                new XElement("Color", "Green")
                            )
                );
            //document.Descendants("Cars").First().Add(elementToAdd2);
            document.Descendants("Cars").First().AddFirst(elementToAdd2);
            document.Save("CarsModified.xml");
        }
    }
}
```

4.9 דוגמה לעבודה עם LINQ to XML

התוכנית הבאה טוענת את הקובץ CarsModified.xml שנוצר מהדוגמה הקודמת (זה שמכיל פרטיים על 3 מכוניות). התוכנית מוחפשת בתוך הקובץ באמצעות LINQ to XML את כל הzbיעים שערכם אדום וمسירה אותם מהקובץ. לאחר מכן מוחפשת את כל המכוניות שמאפיין היצורן שלהם הוא רנו ומוחקת רכבים אלה מהקובץ (המחיקה כוללת את כל תת-האלמנטים של רכבים אלה).

לבסוף התוצאה נשמרת לקובץ בשם CarsModifiedDeleted.xml

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Xml.Linq;

namespace XMLDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            XDocument document = XDocument.Load("CarsModified.xml");

            var colorsToRemove =
                from availableColors in document.Descendants("AvailableColors")
                where (string)availableColors.Element("Color") == "Red"
                select availableColors.Element("Color");

            colorsToRemove.ToList().ForEach(element => element.Remove());
            // colorsToRemove.ToList().Remove(); //another option

            // removing Renault

            var carToRemove =
                (from car in document.Descendants("Car")
                where (string)car.Attribute("Manufacture").Value == "Renault"
                select car).ToList();

            carToRemove.Remove();
            document.Save("CarsModifiedDeleted.xml");
        }
    }
}
```

פרק 10 – ADO.NET

10.1 הקדמה

ADO.NET (Active Data Objects) הם אוסף של ספריות ומחלקות המאפשרים לעבוד מול מקורות מידע שונים: קבצי טקסט, קובצי XML, ובבסיסי נתונים שונים (Access, SQL, Oracle). ניתן להשתמש ב-ADO.NET מכל שפת תכנות הנתמכת על ידי פלטפורמת .NET. במסגרת הרצאה זו, יודגש השימוש ב-ADO.NET לעבודה מול בסיס הנתונים SQL-SERVER.

10.2 אופני עבודה מול בסיס נתונים

: ADO.NET קיימים שני אופני עבודה אפשריים מול בסיס הנתונים בהם תומך ה-

Connected 10.2.1

באופן עבודה זה, ההתקשרות עם הנתונים היא ישירות למקום בו הם מאוחסנים. באמצעות שימוש ב-connection object נוצרת ההתחברות לבסיס הנתונים, ובאמצעות command objects נעשות הפעולות על הנתונים. באופן עבודה כזה, עבור כל פעולה נעשית ההתחברות לבסיס הנתונים, ביצוע הפעולות עליון, ולבסוף התנטקנות ממנה. בקורס עבודה זו, אין חיסכון בזמן העבודה, כי הפעולות נעשות ישירות מול מקור הנתונים עצמו, דבר שאף מייקר את עלות התקשרות עצמה.

Disconnected 10.2.2

באופן עבודה זה, אנו מחזיקים בזיכרון טבלאות של בסיס הנתונים DataTable, שם חלק ממה שנקרה על מנת להשיג טבלאות אלה, אנו קוראים אותם ממקור הנתונים על ידי DataAdapter, ואז ההתקשרות ממקור הנתונים מסתיימת. מעתה, כשbidינו אוסף הטבלאות, נוכל לפעול עליהם: לחסיף, למחוק ולעדכן. בגמר העבודה, נוכלשוב לבצע התקשרות לבסיס הנתונים באמצעות ה-DataAdapter, וכל הפעולות שביצענו יבוצעו הלאה למעשה על מקור הנתונים, ואז תסתיים ההתקשרות עם מקור הנתונים. באופן עבודה כזה, אנו משיגים הן מהירות עבודה, מאשר שהנתונים מוחזקים בזיכרון, והן חיסכון בעליות של התקשרות.

במסגרת הרצאה יסבירו שני אופני העבודה מול בסיס הנתונים תוך מתן דוגמאות. כמו כן, כדוגמה מסכמת תוכן שכבת (Data Access Layer) DAL

. קראו את פרקים 21 ו-22 בספר הלימוד.



10.3 הכוונה להבנת הרצאה

10.3.1 נושאים שיש לסקור בהרצאה

- מבט כללי על מבנה ה- ADO.NET
- Data Providers
- Data Provider Factory Model
- עבודה עם בסיס נתונים במצב מחובר
 - Data Readers .1
 - SqlCommand .2
 - Database Transactions .3

• עבודה עם בסיס נתונים במצב מנותק

- DataSet .1
- DataColumns .2
- DataRow .3
- DataTables .4
- Data Adapters .5
- Filling DataSet .6
- .7 המרת XML ל-XML

- דוגמה מסכמת – הציגת שכבה ה- DAL (Data Access Layer) של פרויקט קטן בנושא כלשהו.

10.3.2ביבליוגרפיה

- פרקים 21, 22 בספר הלימוד
- www.csharp-station.com
- www.functionx.com/csharp
- <http://csharpcomputing.com/Tutorials/TOC.htm>



10.4 תוכניות לדוגמה

10.4.1 תוכנית להציג נתונים ממאגר נתונים

להלן תוכנית המדגימה פעולה במאגר נתונים. הדוגמה פועלת על בסיס הנתונים NorthWind המספק על ידי מיקרוסופט דוגמה יחד עם ה-SQL SERVER. בסיס נתונים זה מכיל נתונים מכירות של חברת דמיונית שמייבאת מוצרים מיוחדים מכל העולם.

```

using System;
using System.Collections;
using System.Collections.Generic;
using System.Text;
using System.Data;
using System.Data.SqlClient;

namespace Example
{
    class EXAMPLE1
    {
        static void Main()
        {
            // connection
            SqlConnection cn = new SqlConnection(
                "Data Source=(local);"
                "Initial Catalog=Northwind;"
                "Integrated Security=SSPI");

            SqlDataReader dr = null;
            try
            {
                // Open the connection
                cn.Open();

                // use the connection with command object
                SqlCommand cmd = new SqlCommand("select *"
                    "from Customers", cn);

                // get SQL query results
                dr = cmd.ExecuteReader();

                // print the CustomerID of each record
                while (dr.Read())
                {
                    Console.WriteLine(dr[0]);
                }
            }
        }
    }
}

```

```
        finally
        {
            // close the reader
            if (dr != null)
            {
                dr.Close();
            }
            // Close the connection
            if (cn != null)
            {
                cn.Close();
            }
        }
    }
}
```

הסבר התוכנית:

- שים לב שבחרנו תחילה ב- connection מסוג sql. לאחר מכן הוגדר cn שהוא ה-
.SqlConnection object וטיפוסו הוא • - לאחר קביעת ההתקשרות עם בסיס הנתונים, נעשית פתיחת תקשורת. - כדי להשתמש בתקשורת זו אנו בונים את cmd Object QSqlCommand מסוג Command Object שהוא הולך בלחוקות בסיס הנתונים, ואת התקשרות המתקבלת כפרמטרים את אפשרות הבחירה של כל הלוחות בסיס הנתונים, ואת התקשרות שנוצרה.
 - בשלב הבא אנו מבצעים את הפקודה ומתקבלים את הרשומות הדרושים לתוכן dr. לאחר מכן ישנה סריקה של כל הרשומות ב-dr על ידי שימוש ב- read והדפסת מספר הזיהוי של כל לקוח בסיס הנתונים דלעיל. בטבלת לקוחות, העמודה הראשונה היא מספר הזיהוי של לקוח ולכון ישנה פניה ל-[0].dr לבסוף ישנה סגירה של ההתקשרות.

10.5 תרגיל מסכם

המשק צריך לתמוך באפשרות של הוספה, עדכון ומחיקה של שירותים מטבלאות אלה. את כל ההזמנות שבוצעו עברו מוצר מסוים, וכן קיבל את רשיית המוצרים שספק ספק מסוים. בנה משיק חלוניי המאפשר להציג תוכן טבלאות אלה, וכן אפשר למצוא מוצרים במלאי, והזמנות. בסיס הנתונים אמרור לכלול טבלאות המייצגות שירותי ניtron לבצע הזמנות של פרטיים מספקים. ביחס הנתונים אמרור לכלול טבלאות המספקים כור בסיס נתונים לייצוג מחסן. ביחס מאוחסנים פרטיים, לכל פריט ישנו ספק המספק אותו. כמו כן

פרק 11 – ORM: Object Relational Mapping

11.1 הקדמה

Object Relational Mapping (ORM) היא "טכניקת תכונות שנועדה להמיר מידע מפורט של בסיס נתונים רלציוני, לפורמט של אובייקט בסביבה מונחית עצמים. כלי ORM נועדו לפשט את תהליך הפיתוח ולשפר ביצועים" (מתוך ויקיפדיה).

המידע במערכת מונחית עצמים הוא אוסף של **אובייקטים** מורכבים שעלייהם נעשות מניפולציות שונות. לעיתים, אובייקטים אלה מהווים ישות המורכבת מפרטים רבים ומישמת עקרונות כגון ירושה, פולימורפיזם, תבניות עיצוב וכדומה. כאשר מערכת מונחית עצמים זקופה למידע על מנת לבצע פעולה, היא צריכה את המידע זמין בצורת אובייקט, שהוא הפורמט הטבעי והנוח לעובודה בסוג סביבה זאת. אובייקטים הם אבני הבניין בסביבה מונחית עצמים.

הואיל והמידע על אודות האובייקט שמור בסיס נתונים רלציוני, שם הוא נשמר כאוסף של נתונים סקלריים פשוטים (לא אובייקטיבים) על פני אוסף של טבלאות שונות, מתוරת בעיות מייפוי של מידע בין שתי מערכות המייצגות את המידע באופן שונה. כאן נכנסים לתמונה כלי המיפוי האוטומטיים המגשרים על פער זה. חשוב לציין ש מבחינת מערכת מונחית עצמים, תהליך השמירה של אובייקט ושהזורו אמורים להיות שקופים למערכת (Transparent Persistence). אפשר לומר שכלי המיפוי מספקים סוג של בסיס נתונים מונחית עצמים (ולא רלציוני) עבור מערכת מונחית עצמים.

צפו בסרטון הבא הממחיש את תפקידו ואת יכולותיו של כלי המיפוי:

http://www.service-architecture.com/object-relational-mapping/articles/transparent_persistence.html

Active Record Pettern 11.2

11.3 כלי מיפוי

במסגרת הרצאה נלמד לעבוד עם שני כלי מיפוי. אחד מהם הוא פרויקט של קוד פתוח

Castle Project 11.3.1

כלי מיפוי זה מקנה לאובייקטים במערכת מונחית עצמים כלשהי את יכולת לשמר ולשחזר את עצם בסיס נתוניים רלוונטיים. השימוש בפרויקט זה דורש מאיתון כמתכנתים לכתוב תחילת את האובייקטים שבהם אנו עושים שימוש במערכת מונחית עצמים. באמצעות כתיבת מספר הגדרות פשוטות נוספת לאובייקטים אלה, מתקבל באופן אוטומטי המיפוי אל בסיס נתונים רלוונטי בו ישמרו האובייקטים וכן יכולות שמירה, שחזור, ביצוע שאילתות ועוד על בסיס נתונים זה. בפרויקט זה, Castle Project, כיוון המיפוי הוא מהאובייקטים לבסיס הנתוניים, ככלمر אנו נדרשים לכתוב את האובייקטים ובתמורה מקבלים את אופן התנהלות שמירתם ושמורות מול בסיס הנתוניים.

The Entity Framework 11.3.2

זהו כלי נוסף חלק מפלטפורמת .NET, אשר מספק יכולות דומות לאלה הקיימים בכלים הקודמים. אופן העבודה עם הכלים מתואר בספר הלימוד בפרק 23. ואנו נתאר כאן בהמשך צעדי פעולה כדי לעבוד עם הכלים.

11.4 הכוונה להכנות הרצאה

11.4.1 נושאים שיש לסקור בהרצאה

- סקירת הבעה וביאור מונחים הקשורים בנושא ORM, בפרט יש להסביר את המונחים הבאים:
 - ORM .1
 - Persistence .2
 - Active Record Pattern .3
 - NHibernate .4
- סקירה כללית של פרויקט Castle Project והדגמת העבודה אליו.



- סקירת Entity Framework ואופני העבודה איתו. נדרש להציג כיצד עובדים עם EF בגישה מודול תחילת, בגישה בסיס נתונים תחילת, וכן במצב שכבר יש לנו מחלקות משלנו וברצוננו לשדרוג ביכולות לעובדה מול בסיס נתונים.
- הסבר מפורט על אופן השימוש בכלים הניל. ההסבר צריך לכלול:
 1. אופן התקנה.
 2. תהליך השימוש – דרישות והגדרות שונות.
- דוגמת סיכום – בניית שכבת DAL באופן אוטומטי באמצעות שימוש באחד מכלי המיפוי עבור מערכת מידע קטנה מסוימת (כגון : מוצריים, ספקים והזמנות).

11.4.2ביבליוגרפיה

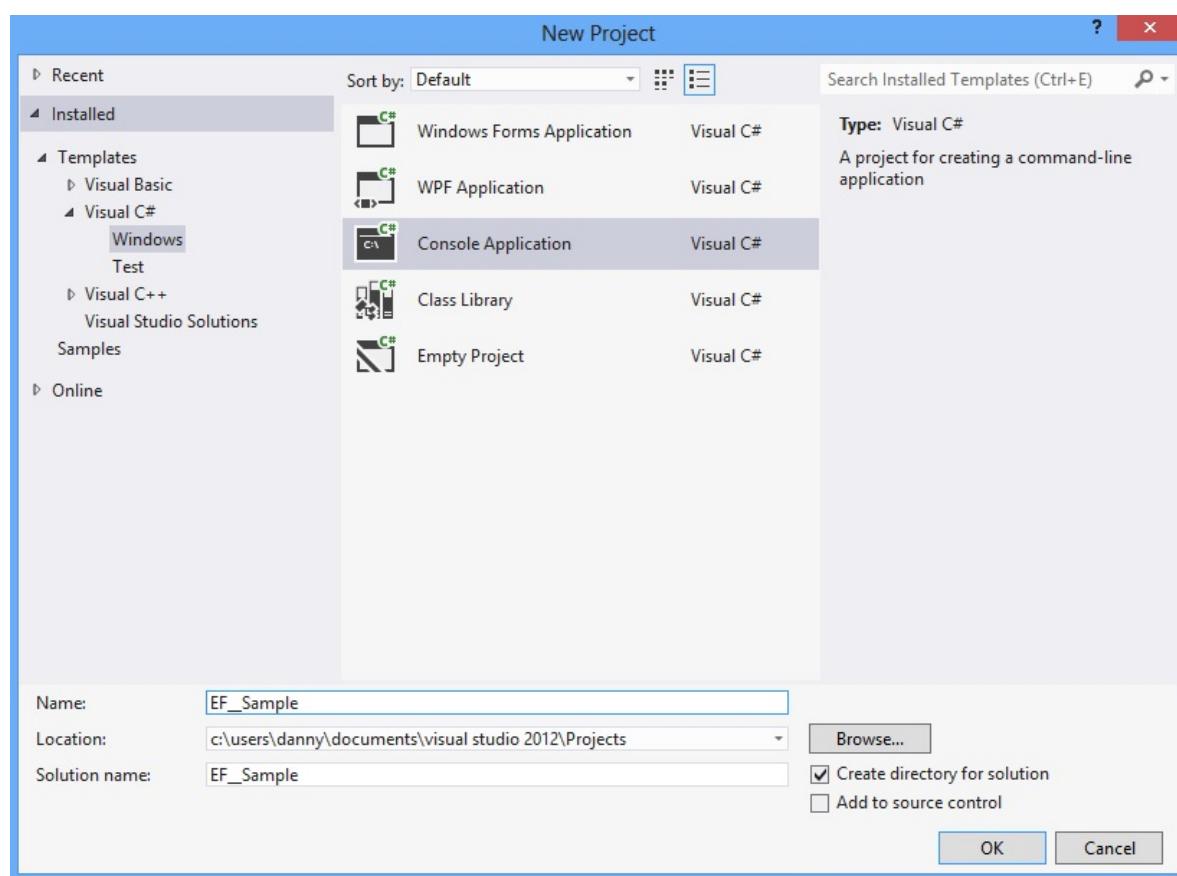
- מאמר מקיף בנושא מיפוי אובייקטים.
<http://www.agiledata.org/essays/mappingObjects.html>
- הסברים מקיפים כולל סרטוני וידאו על EF באתר MSDN של מיקרוסופט.
<http://msdn.microsoft.com/en-us/data/ef.aspx>
- אתר פרויקט ה- Castle Project :
<http://www.castleproject.org/activerecord>
- פרק 23 בספר הלימוד.

11.5 הדוגמה צעד אחר צעד של Entity Framework Model First

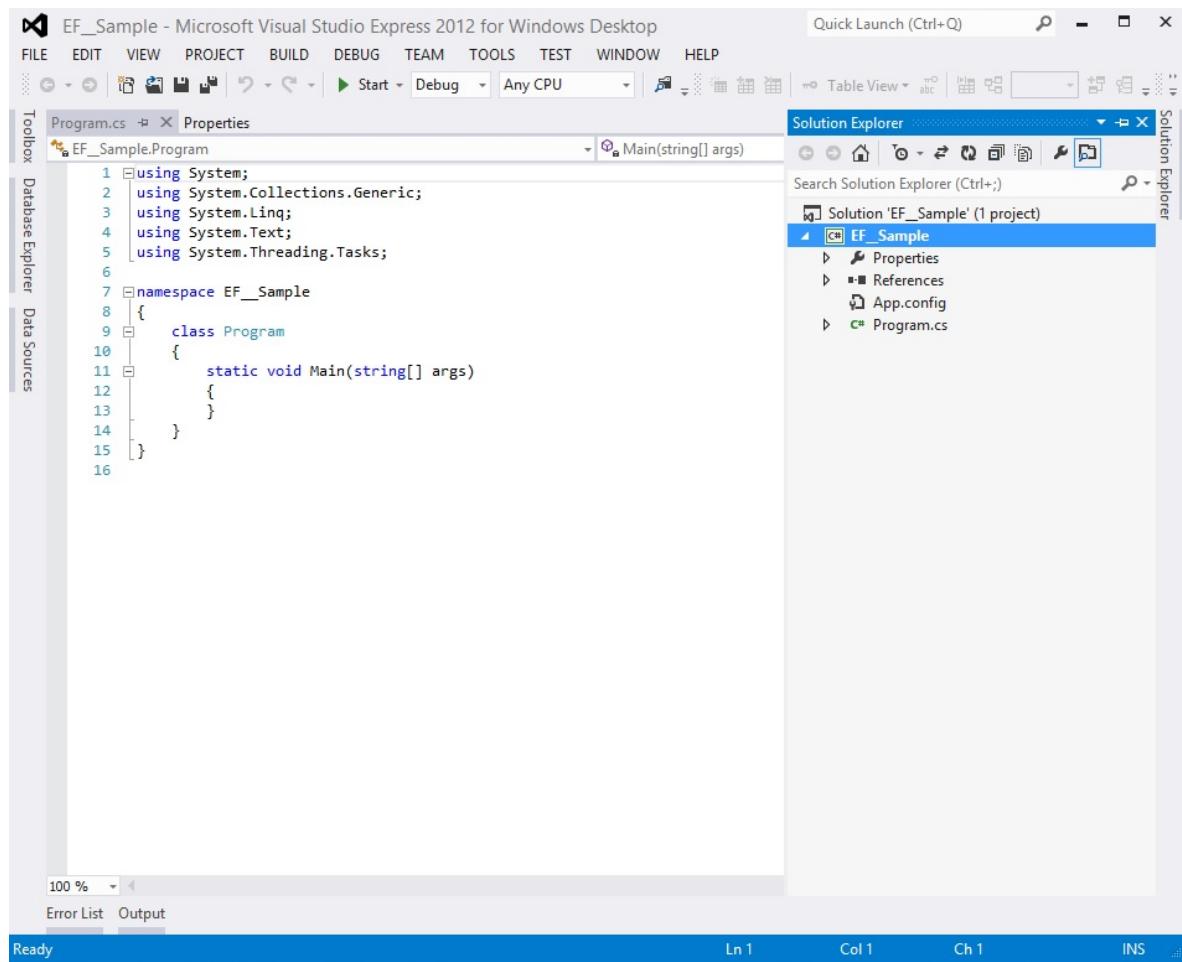
בדוגמה זו נדגים בנית אפליקציה העוסקת בניהול מידע על סטודנטים, קורסים והרשומות של סטודנטים לקורסים. את שכבות הנתונים נבנה באמצעות שימוש ב- Entity Framework וNSTAMES בגישה של בניית מודל תחיליה. כלומר, במצב זה אין בידינו בסיס נתונים קיים שמןנו ליצור מודל, וכן אין בידינו קוד קיים עם מחלקות שלפיהן ניתן לבנות מודל ובסיס נתונים. בМИלים אחרות, אנו מתחילה לבנות את המודל עבור אפליקציה ריקה ממחלקות ולא בסיס נתונים קיים.

נתאר כעט שלב אחר שלב בנית אפליקציה עם מודל נתונים, לאחר מכן דוגמה לתוכנית המדגימה פעולה עם המודל שייצרנו. ההנחה היא שכבר התקנתם במחשב שלכם את [SQL Server](#).

בשלב ראשון ניצור פרויקט חדש מסוג Console Application שנדרא לו לצורך העניין
כמפורט באירור למטה

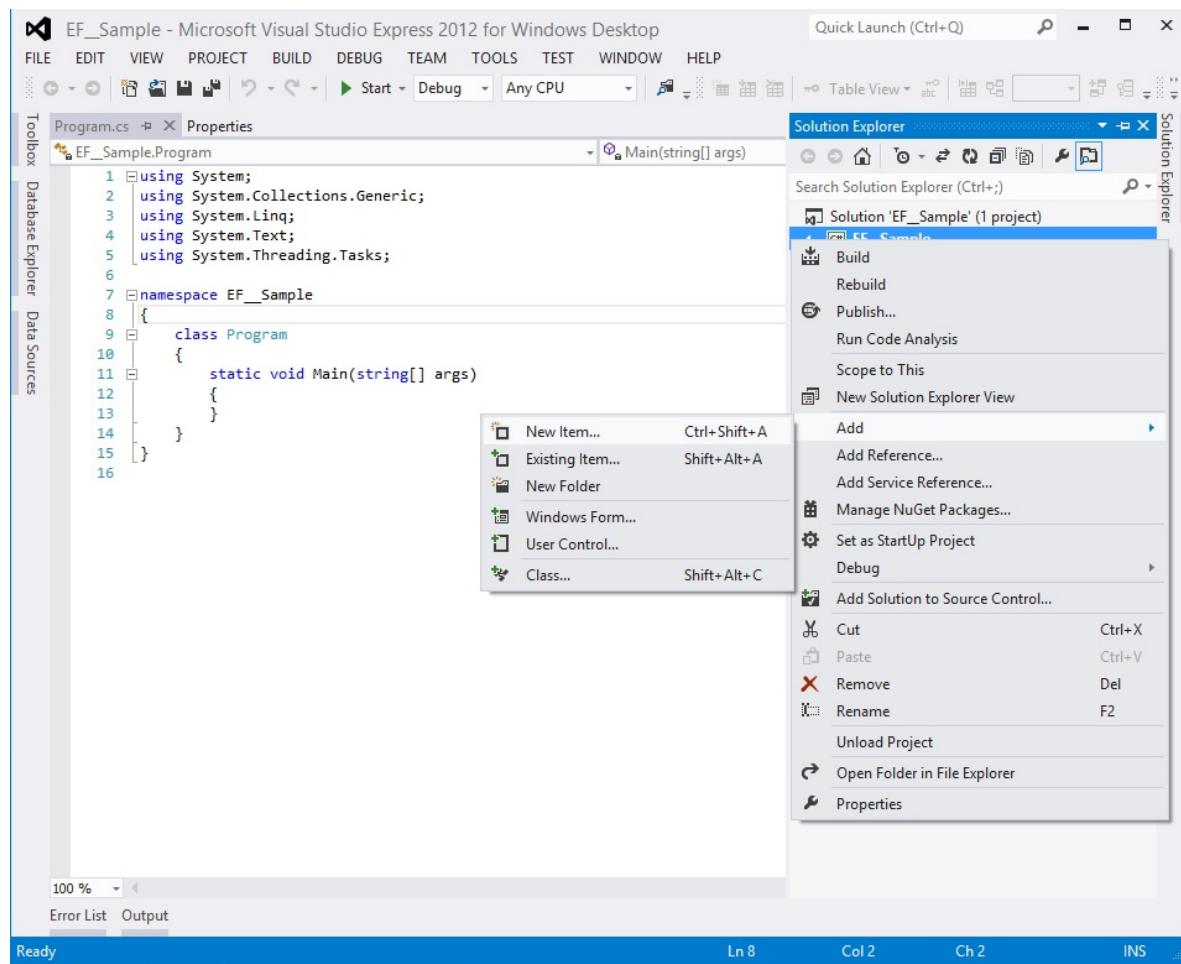


לאחר ייצרת הפרויקט קיבלנו את המסך הבא, וכפי שאתם יכולים לראות זו אפליקציה ריקה לחלוטין.

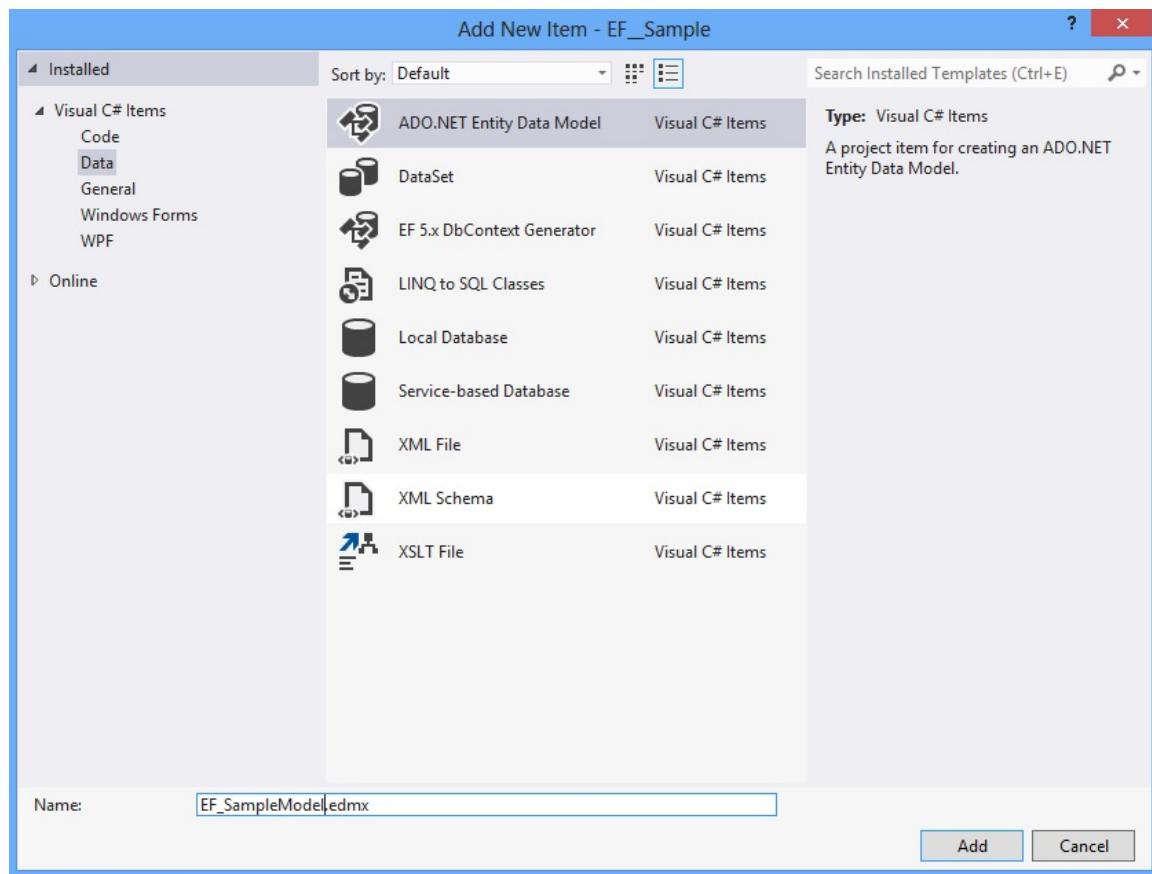


נתחיל כעת להוסיף לאפליקציה אפשרות לבניית מודל קונספטוAli עבור שכבת הנתונים.

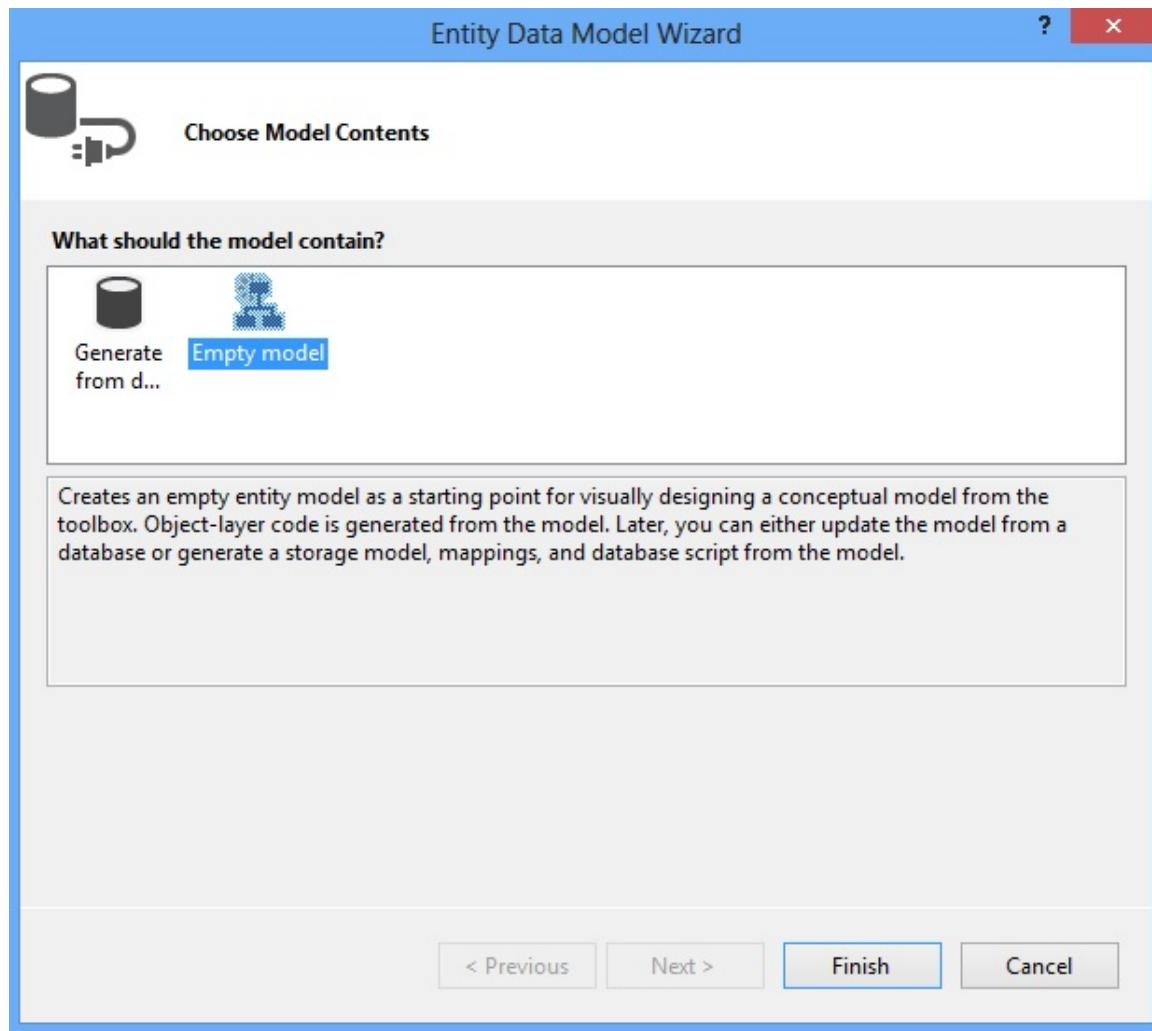
לשם כך, ניגש ל- Solution Explorer ונקлик על הכפתור הימני בעבר כאשר אנו עומדים על שם הפרויקט. נקבל את התפריט המתוואר במאז הבא. בתפריט נבחר ב- Add ולאחר מכן ב- .New Item.



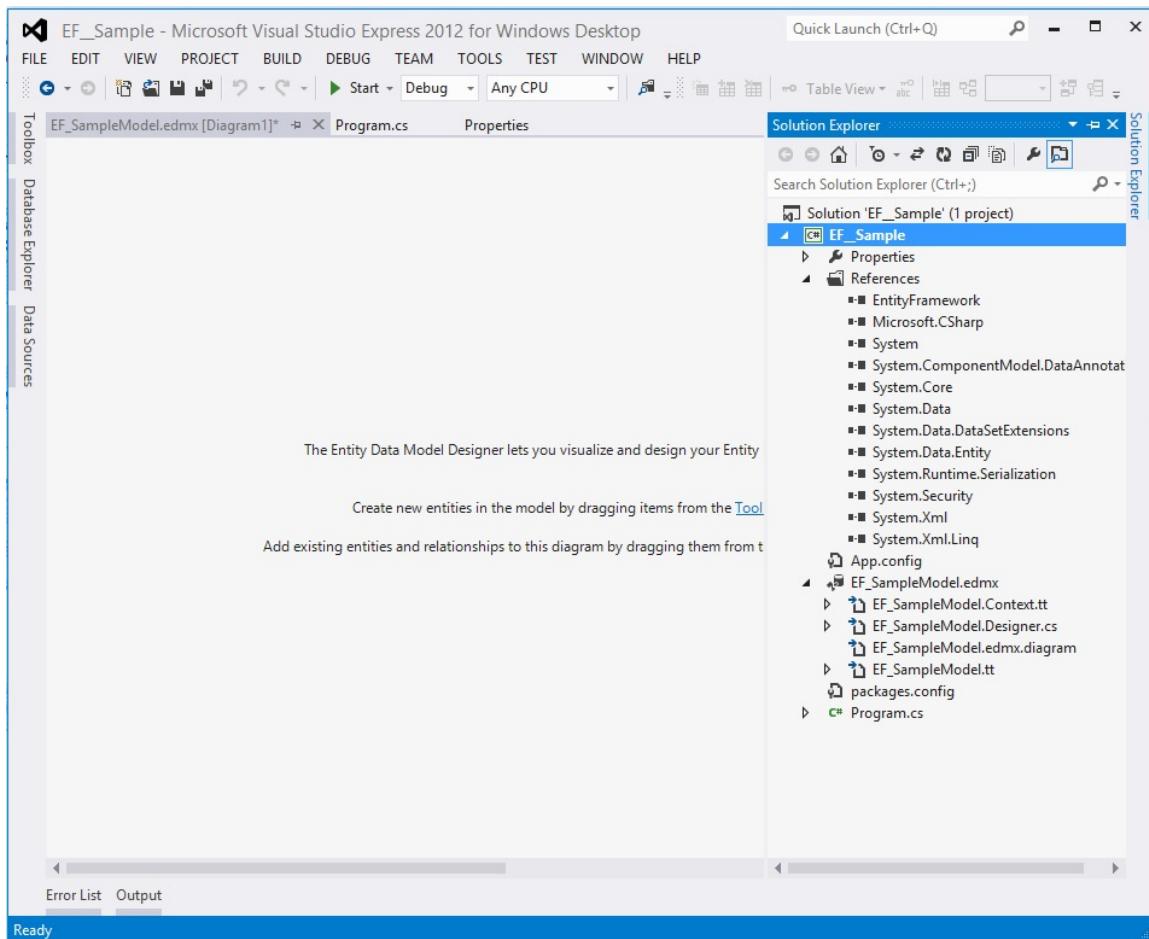
לאחר מכן נקבל את המ███ הבא, ובו ניגש ל- Data ושם נבחר ב- ADO.NET Entity Data Model
לצורך הדוגמה נשנה את שמו של המודול ונקרא לו בשם EF_SampleModel



לאחר מכן קיבל מסך שבו נישאל האם נרצה לבנות מודל מבוסיס נתונים קיים או שמא נרצה להתחיל מבנית מודל ריק? כפי שציינו בתחילת, לאפליקציה שלנו אין בסיס נתונים קיים, ולכן נבחר באופציה של Empty model ולחץ על כפתור Finish כמתואר במסך הבא:

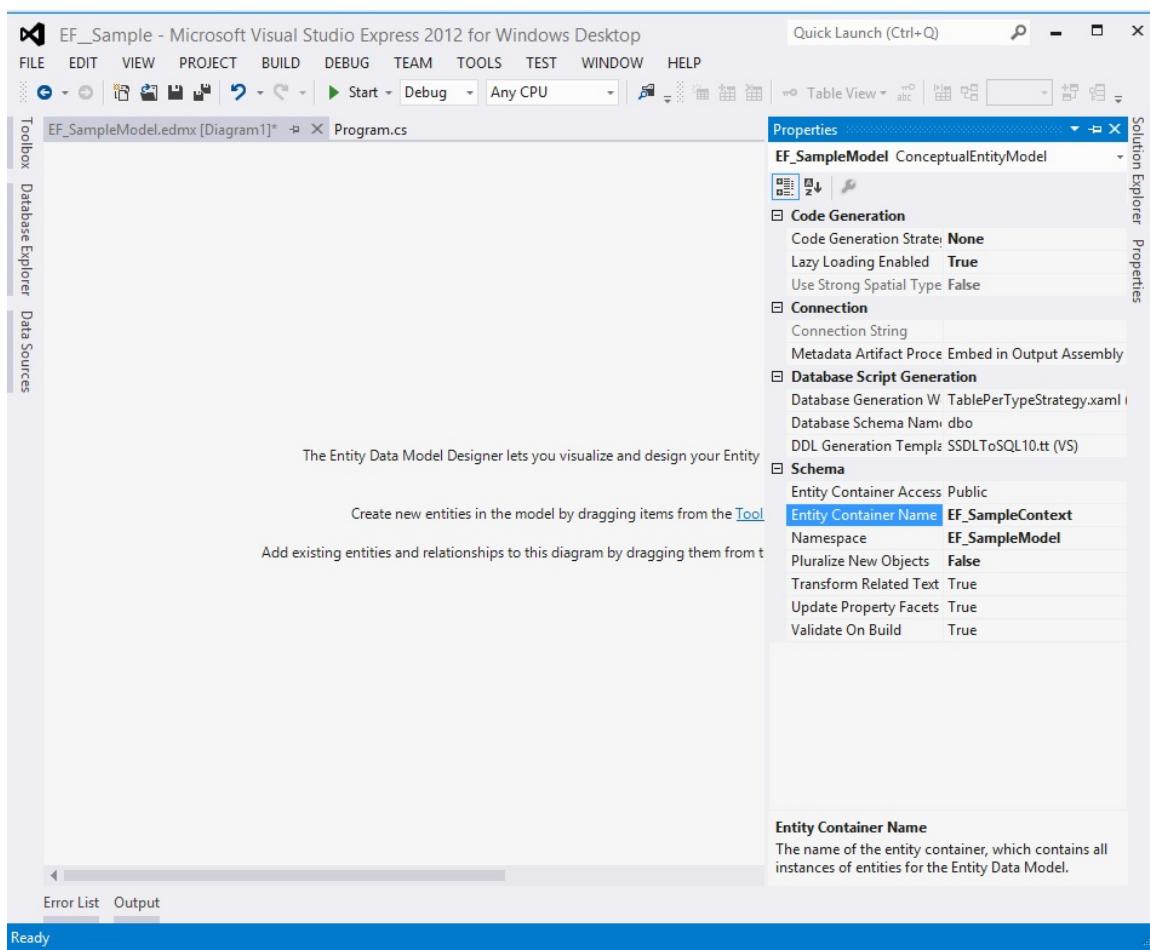


כפי שניתן לראות במסמך הבא, לאפליקציה שלנו נוספו מרכיבים חדשים. אחד מהם הוא הקובץ EF_SampleModel.edmx המאפשר לנו, באמצעות שימוש ב-Designer, לעצב לעצמו את המודל הكونספטואלי של שכבות הנתונים.

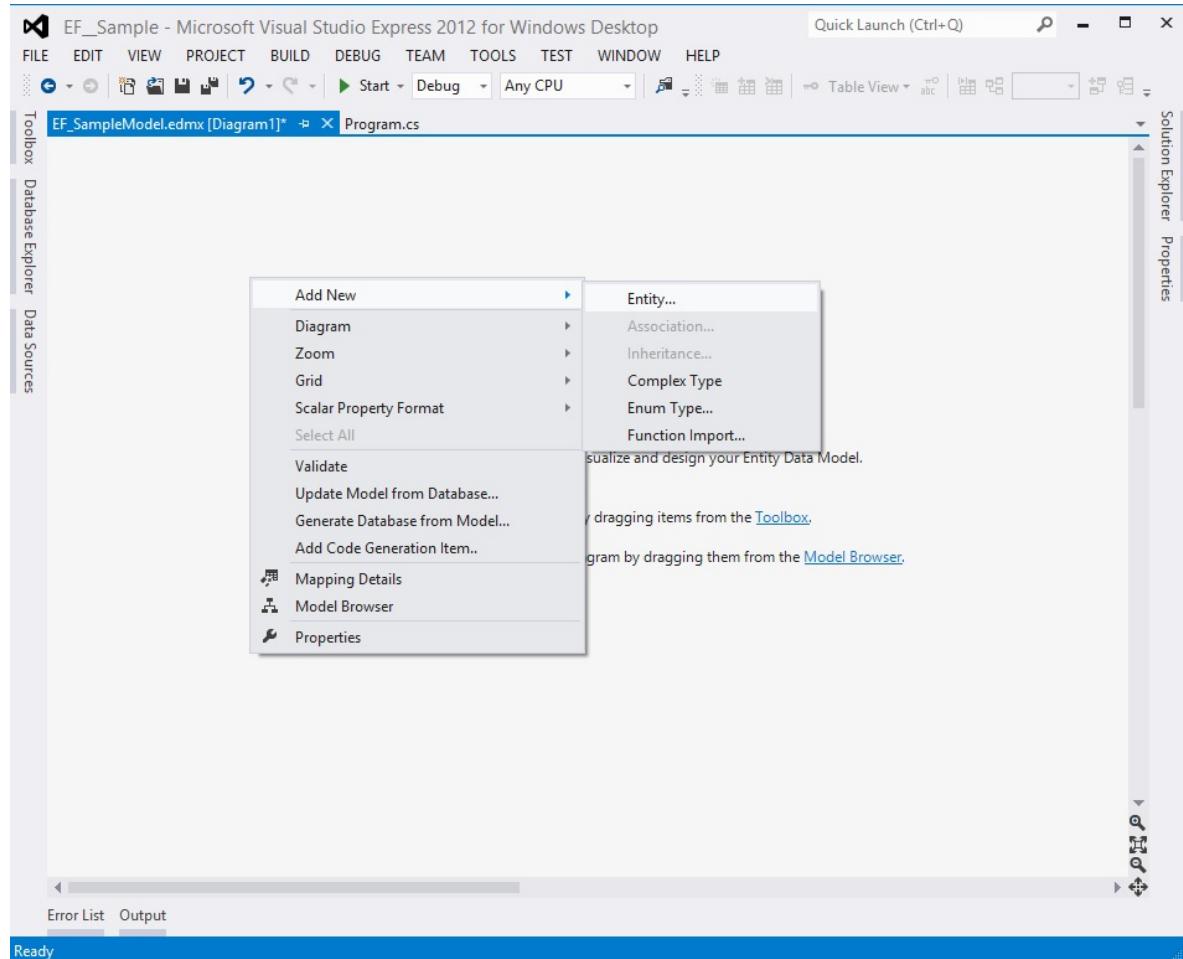


נקליק עם הלחצן הימני בעבר על שטח השירות (החלק המרכזי בmse) ובחר ב- Properties לשינוי של הגדרות שונות עבור המודול שנבנה בהמשך. בmse Properties נשנה את המאפיין ששמו EF_SampleContext Entity Container Name כמתואר בmse הבא.

מצין שלא חיבים לעשות זאת, והסיבה ששינו את שם המאפיין היא בגלל העבודה שבשימוש התהילה נוצר עבורנו מחלקה שיורשת ממחלקה בשם DBContext, ולכן מתאים יותר לקרוא למחלקה שתיווצר בשם החדש שניתן למאפיין.



נחוור שוב לשטח הشرطוט, ונתחיל במלאתו עיצוב המודל הקונספטוואלי של מודל הנתונים. המודל מורכב מישויות ומקשרים שביניהם. ניצור תחילה ישות המתארת פרטיים על קורס. לשם כך נלחץ בשטח הشرطוט על הכפתור הימני בעבר ונבחר ב- Add New ולאחר מכן ב- Entity כמפורט בסעיף הבא:



נקבל את המסלך הבא ובו נקליד פרטים על היחסות שברצוננו ליצור, כמפורט בסלך הבא.

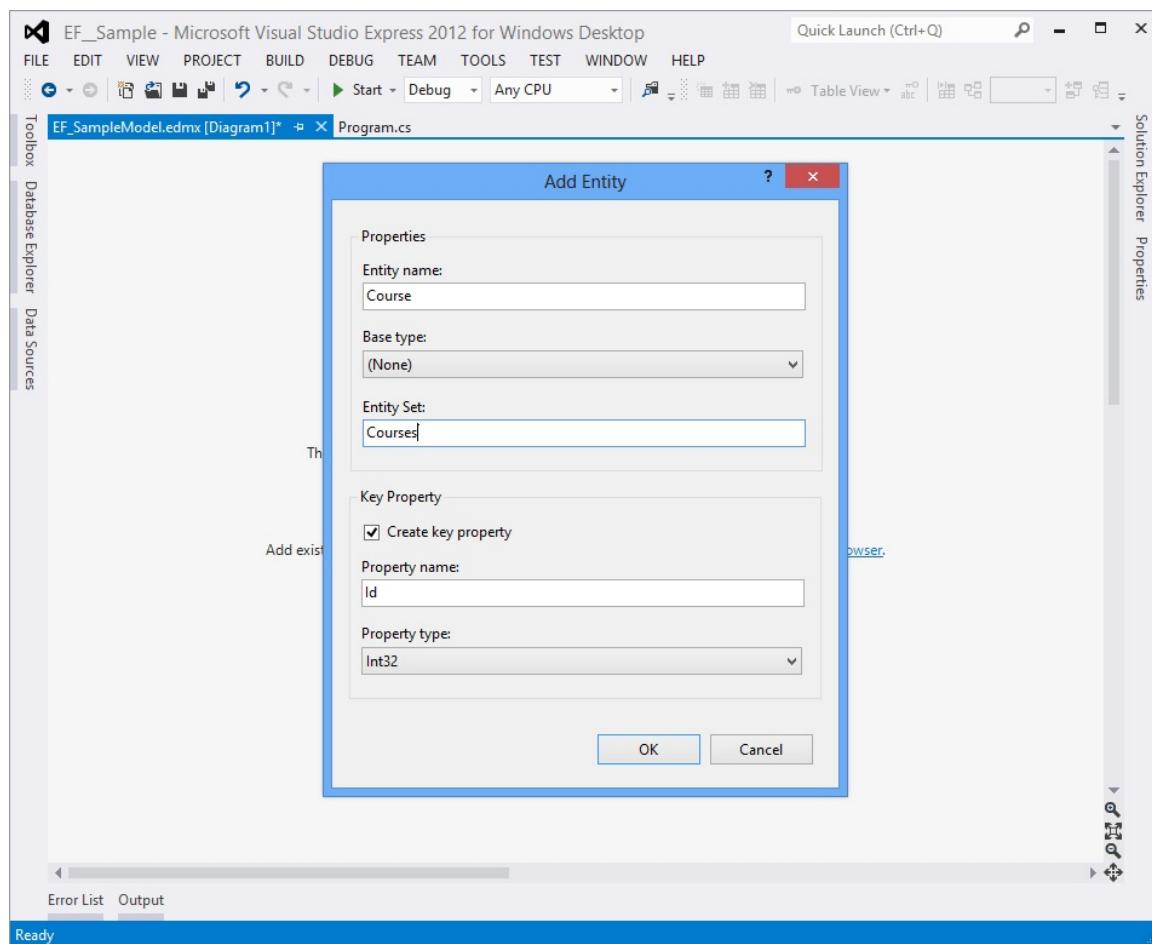
נקlid בשדה Entity Name אם שמה של היחסות, שנקרא לה Course.

בשדה Entity Set נקליד Courses (זה יהיה השם של הטלחה שתיווצר בסיס הנתונים לשימרת מידע על יישוות של קורסים ולכך טבעי לקרוא לה Courses).

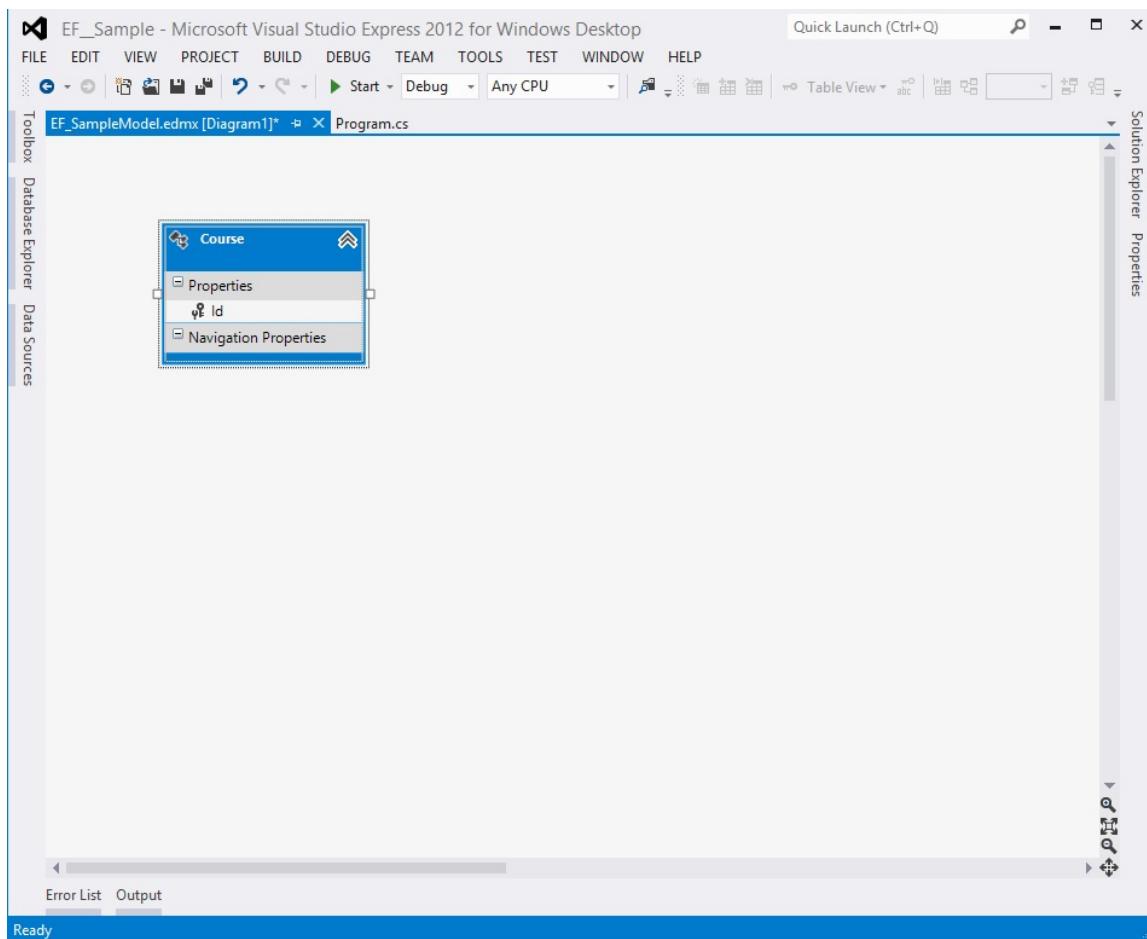
נסמן את האופציה של Create Key Property על מנת שלטבלה זו יהיה מפתח ייחודי.

שםו של המפתח יהיה Id וטיפוסו הוא Int32.

לחץ על כפתור OK.

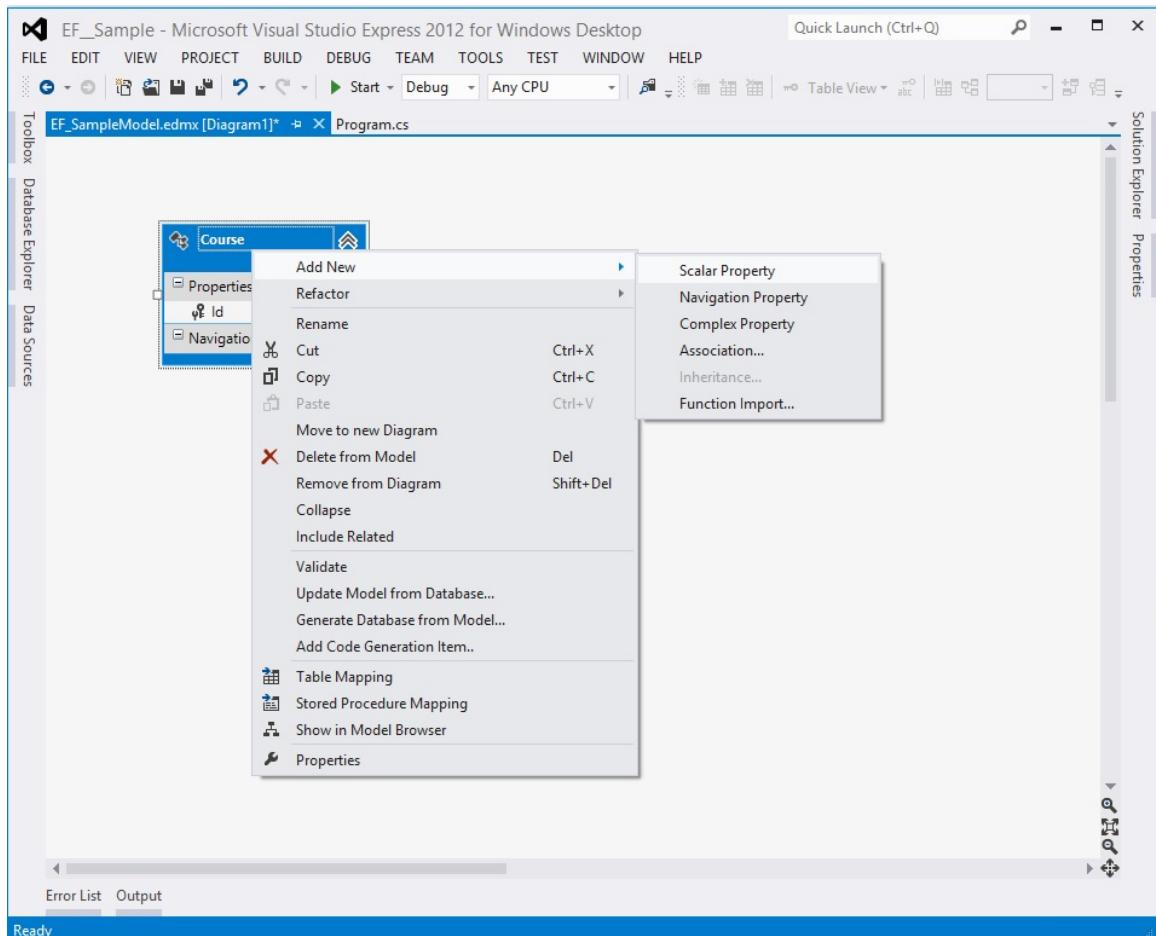


כפי שניתן לראות, נוצרה לנו יישות בשם Course בسطح השירות. כרגע המאפיין היחיד של היישות הוא המפתח Id של הקורס.

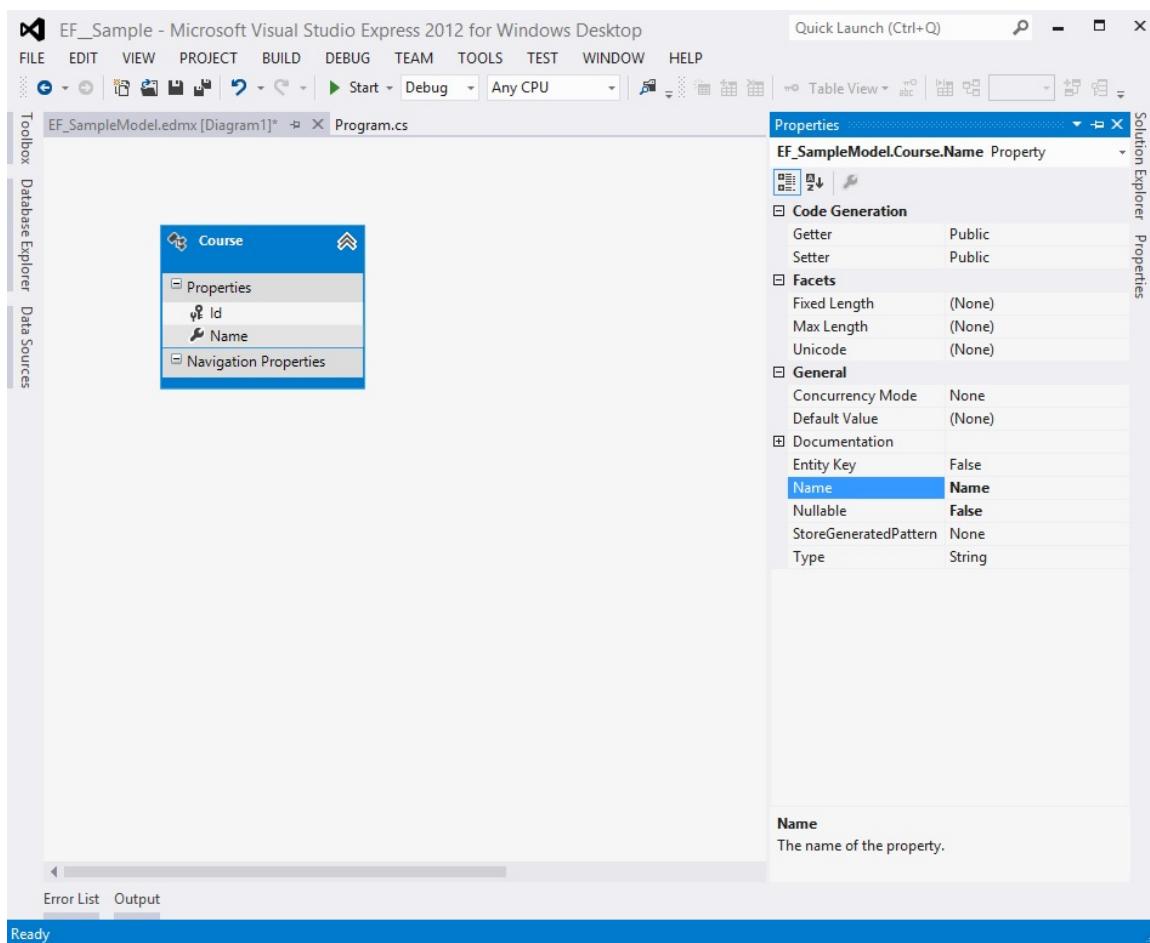


נוסף לישות Course מאפיין נוסף המתאר את שמו של הקורס ונקרא לו Name. שם כך נלחץ עם הכפרטור הימני בעכבר על הישות Course ונקח ב- Add New Scalar Property, כמתואר בסיס

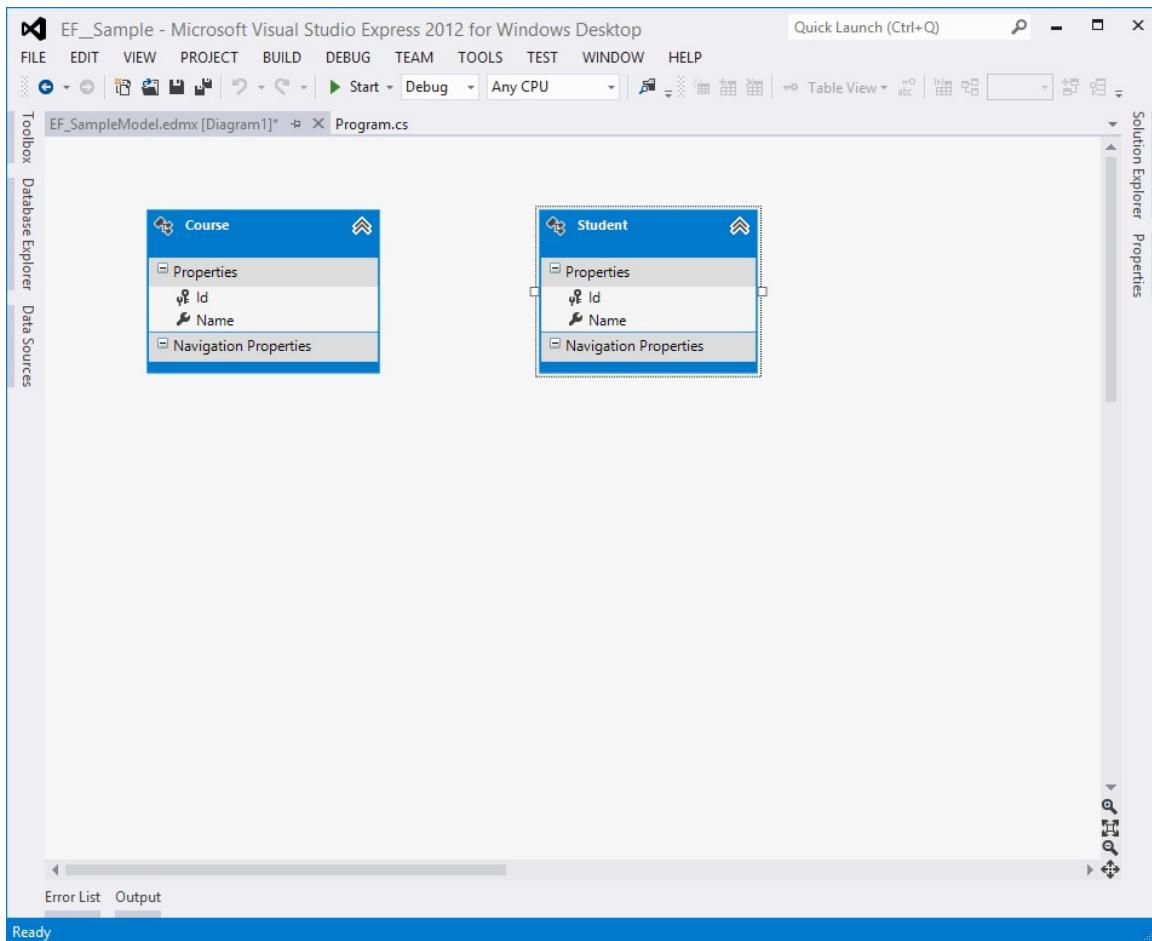
הבא :



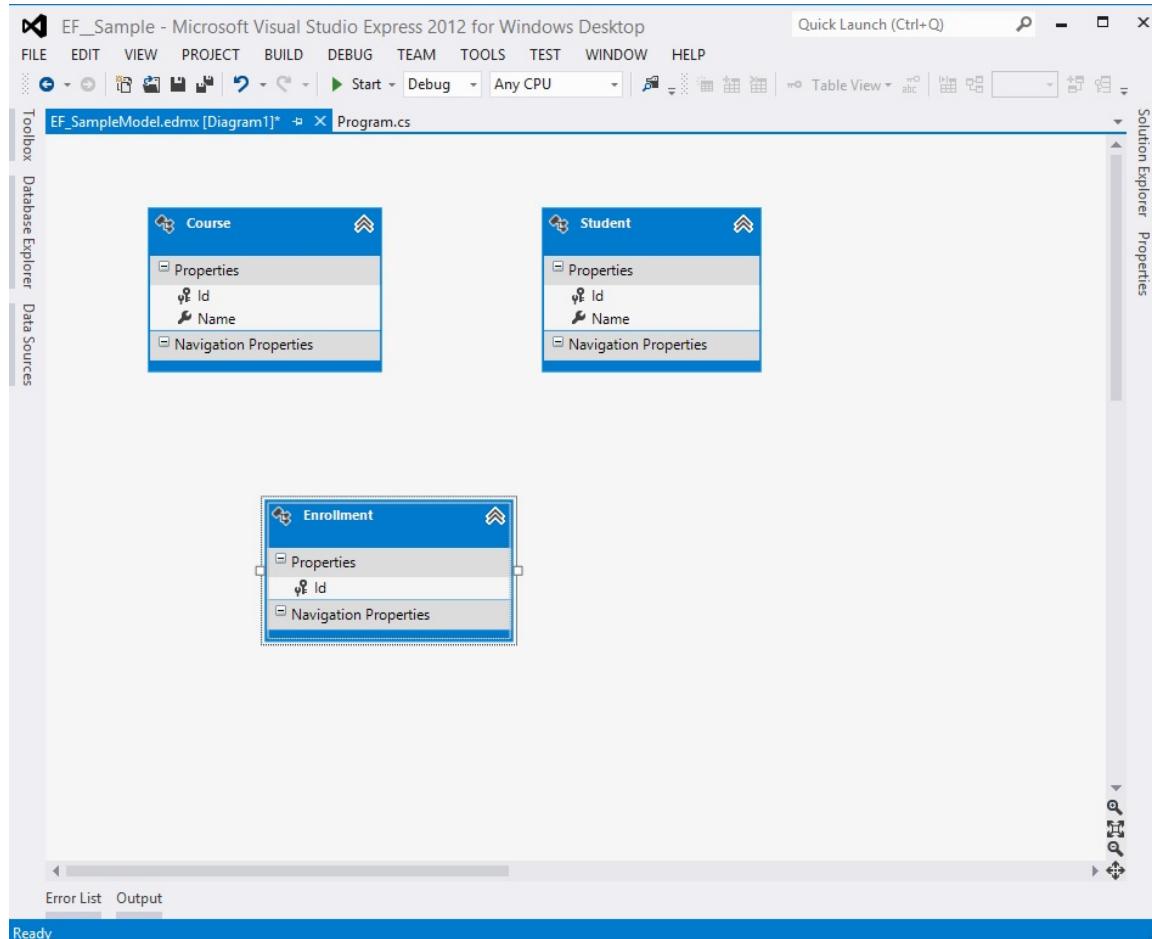
כפי שניתן לראות במסך הבא, התווסף ליישות מאפיין חדש, נקרא לו בשם Course וnlchz עלי עם הלחצן הימני בעבר ונבחר ב- Properties, שם נוכל לקבוע בשדה Type את סוג (במקרה זה מחרוזת). כן נוכל לקבוע בשדה Nullable האם נתיר לו להיות עם ערך ריק בטבלת הקורסים בבסיס הנתונים, נבחר ב- Nullable שדה Nullable מאחר שלכל קורס אנו מצפים שהוא שם, ולכן לא נרצה להיתר לשדה זה להיות ריק.



באופן דומה לתהיליך שבו יצרנו את היחוס Course, ניצור יישות נוספת לתיאור פרטיים על סטודנט ונקרא לה Student. גם לסטודנט יהיה מפתח בשם Id ושם Name מסווג מחוوظת המותאר את שמו של הסטודנט. לאחר ייצירת היחסות Student נראית בשטח השירות את שתי היחסויות שייצרנו כמפורט בסען הבא:



כעת נוסף יישות שלישית למודל שלנו כמתואר בסעך הבא. יישות זו מתארת מידע על הרשומות של סטודנטים לקורסים. הוספת היישות נעשית באופן דומה לצירוף היישויות של סטודנט וקורס. נקרא לישות החדשה Enrollment, ובשלב זה לא נוסף לה מאפיינים נוספים מעבר למפתח Id שיש לה.

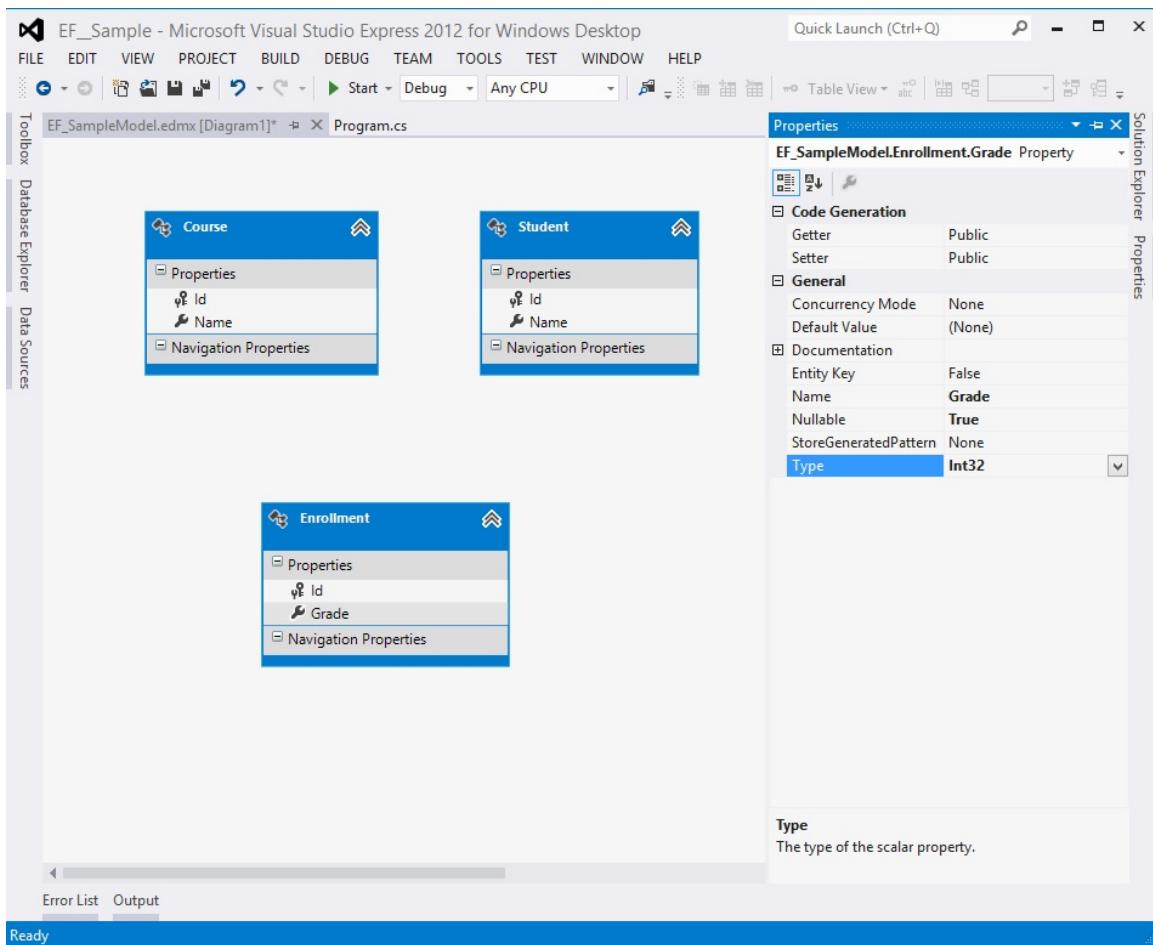


לישות Enrollment, המיצגת פרטיים על הרשמה, נרצה שהיו המאפיינים הבאים:

ה- Id של הסטודנט שמעורב בהרשמה וה- Id של הקורס המעורב בהרשמה, וכן שדה בשם Grade המתאר את הציון של הסטודנט בקורס.

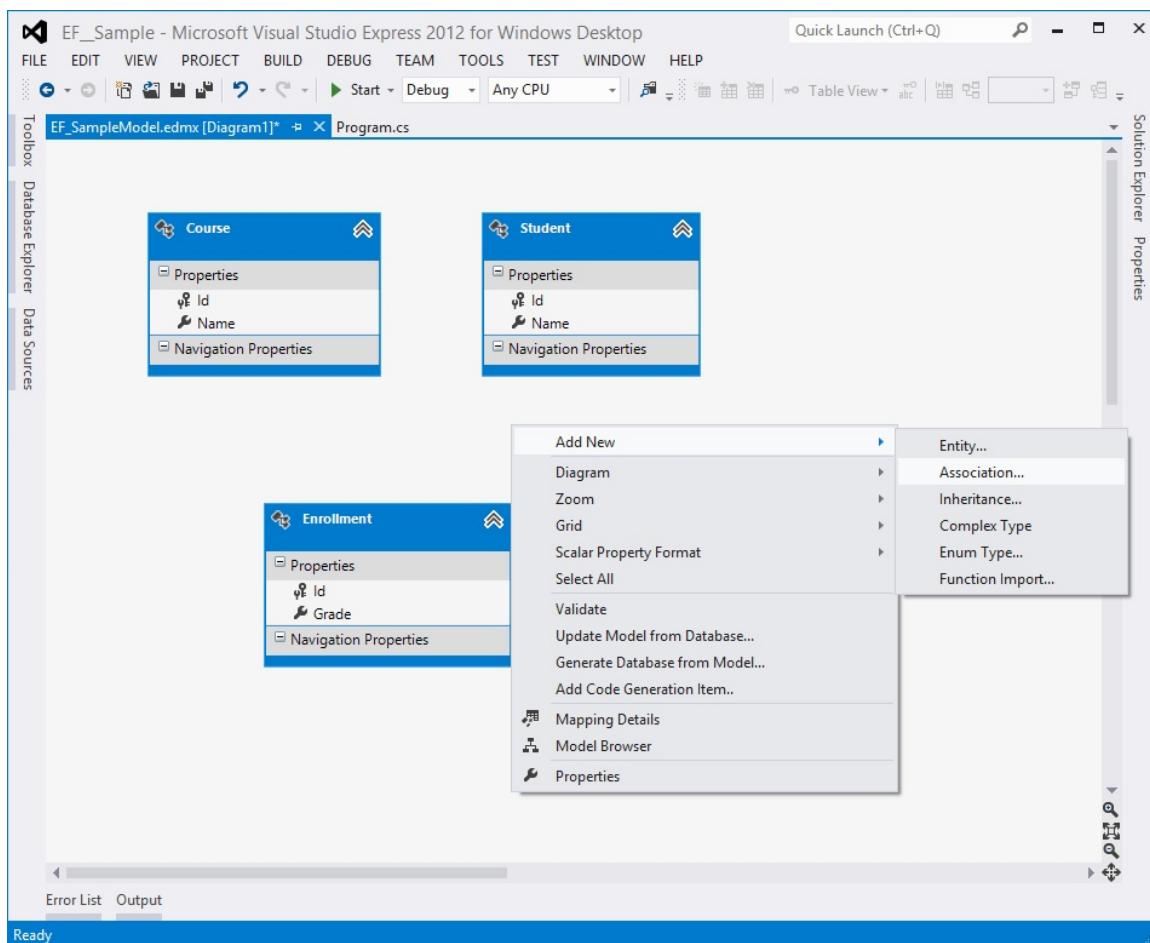
אנו נוסיף רק את המאפיין Grade ונשנה את ה- Properties שלו כך שיוכל להיות שדה ריק בסיס הנתונים (עד שהציגו לא ניתן נשאר ריק, וכך צרי לאפשר ערכי Null עבורו) וכן נשנה את טיפוסו ל- Int32, כמתואר בסעיף הבא.

לישות הרשמה יש קשר עם יישיות קורס וסטודנטים המעורבות בה. בהמשך נוסיף קישורים בין היישויות שיגרמו לכך שלישות Enrollment יהיה מאפייניהם נוספים של ה- Id של הקורס וה- Id של הסטודנט שקשורים להרשמה. וכן אין טעם שנוסיפים בעצמנו.



כעת, לאחר שייצרנו את כל היחסיות הנדרשות, נותר לקבוע רק את הקשרים ביניהם. כפי שציינו קודם, ישנו שני קשרים, האחד בין יישות קורס לישות הרשמה, והשני בין יישות סטודנט לישות הרשמה.

נלחץ כעת בשטח ריק בלוח השרטוט על הלחצן הימני בעבר, נבחר ב- Add New וואז ב- Association יוצרת יחס בין יישויות, כמפורט בסעיף הבא:

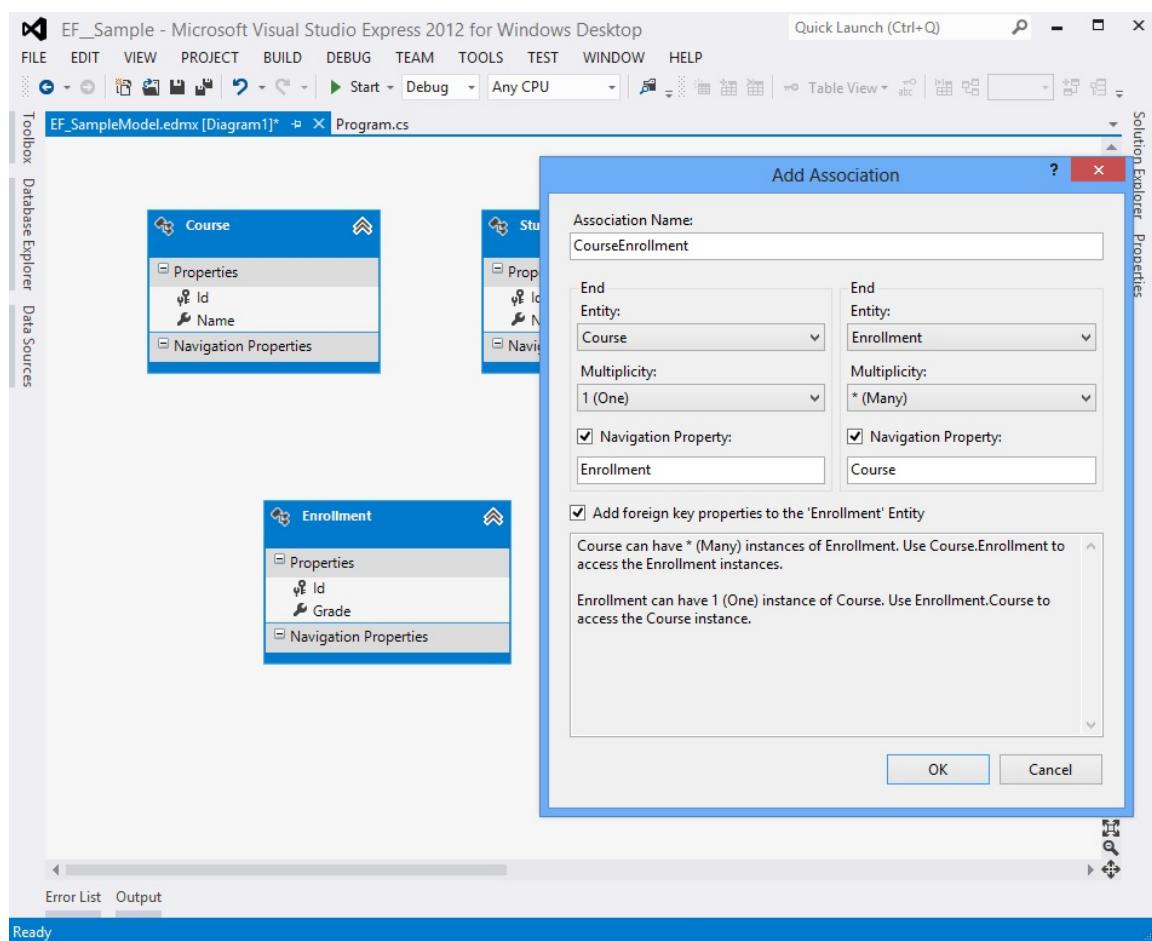


לאחר מכן נקבל מסך שבו נקליד את פרטי היחס בין היחסות הרצויות. נתחל בתיאור היחס שבין יישות קורס ליחס הרשמה. בשדה End Entity השמאלי נבחר ביחס Association Name הימני נבחר ביחס Course, ובשדה End Entity השמאלי נבחר ביחס Enrollment. שמו לב, שדה Course משנתה אוטומטית בכל פעם שאנו בוחרים ערכים בשדות ה-End Entity.

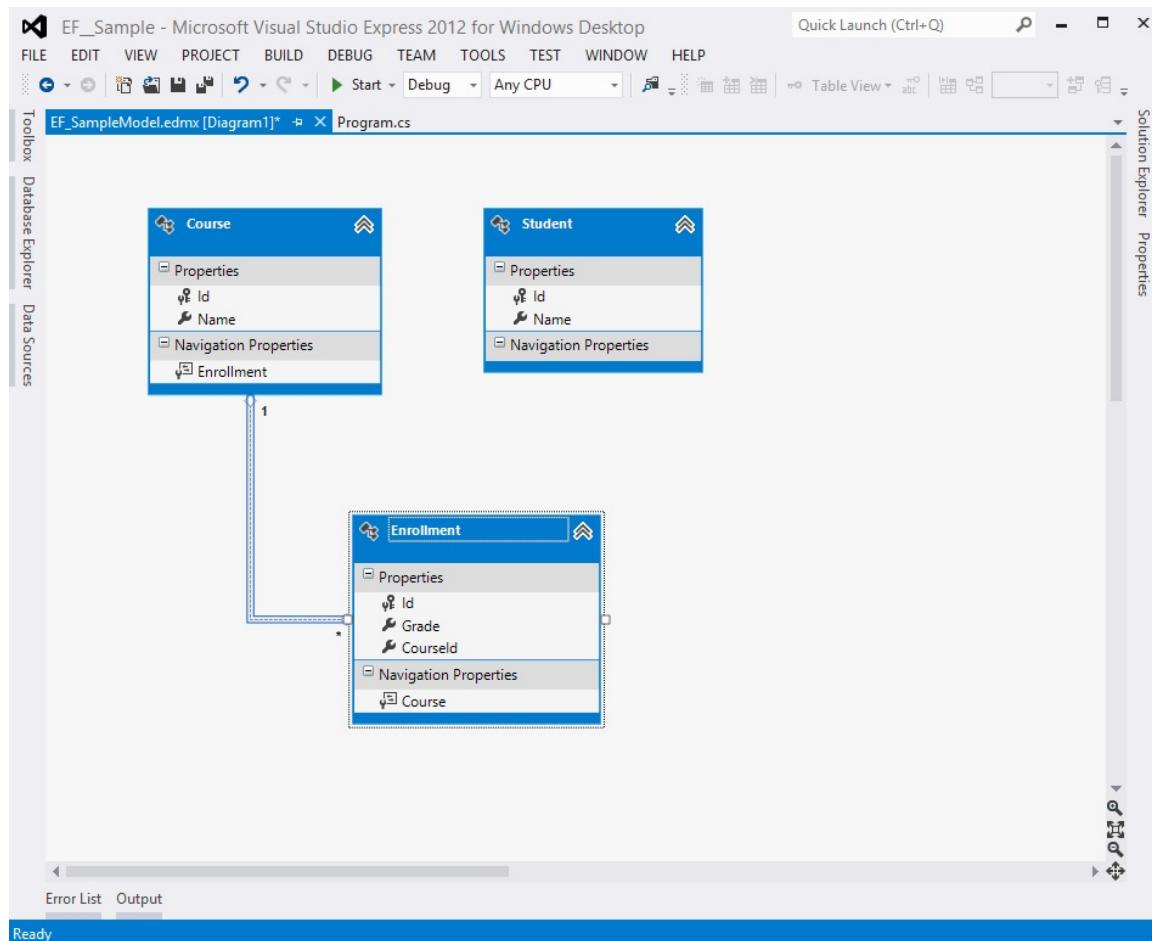
כעת נקבע את מידת הריבוי (multiplicity) של הקשר. בכך שמאלי נבחר ב- 1 עבור Course ובצד ימין נבחר ב- Many עבור יישות הרשמה. המשמעות של בחירה זו היא שלכל קורס יכול להיות הרבה הרשםות, וכל הרשמה מתאימה רק לקורס מסוים.

נסמן את השדה Add foreign key property to the Enrollment Entity, וזה מה שאמור להוסיף ליחסות של הרשמה מאפיין נוסף שהוא Id של הקורס שמעורב בהרשמה.

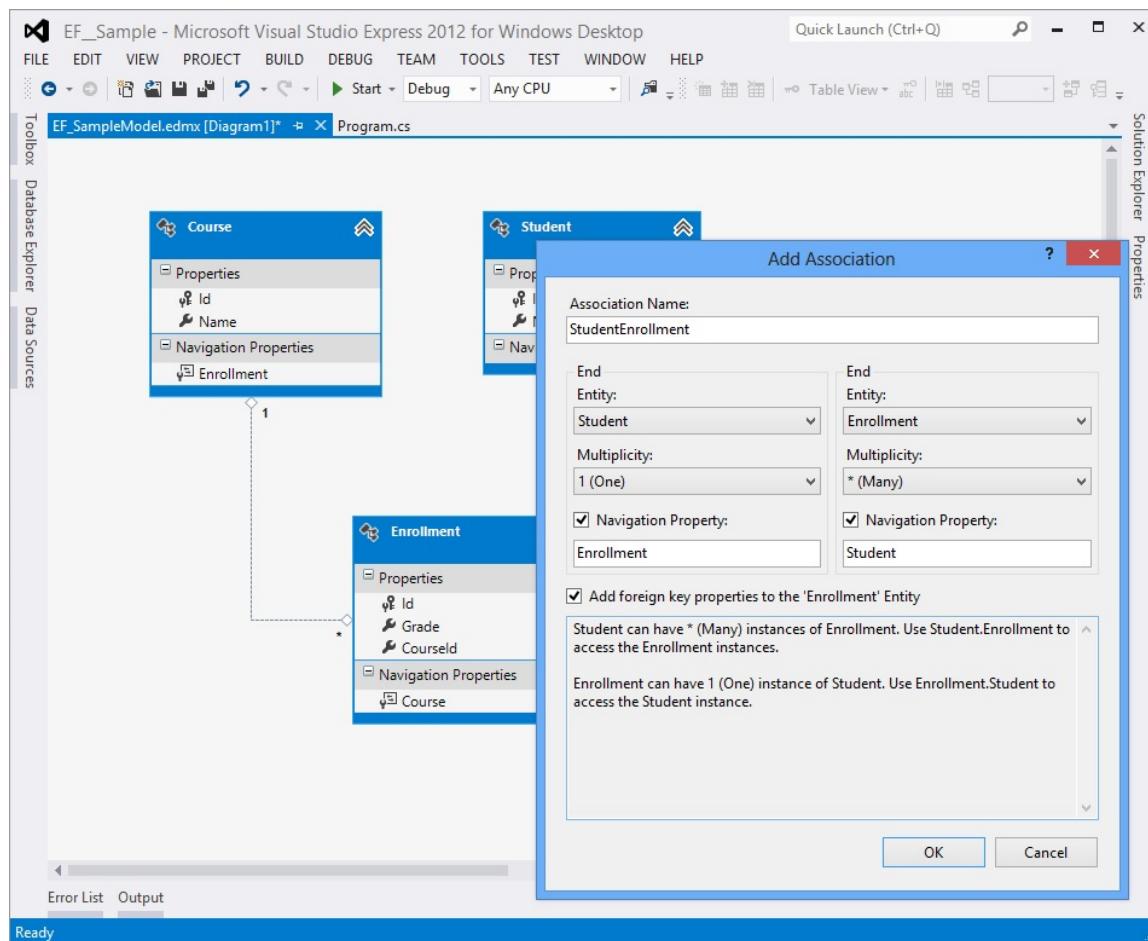
לאחר מילוי הפרטים כפי שמתוארים בסיסically הבא, נלחץ על כפתור OK.



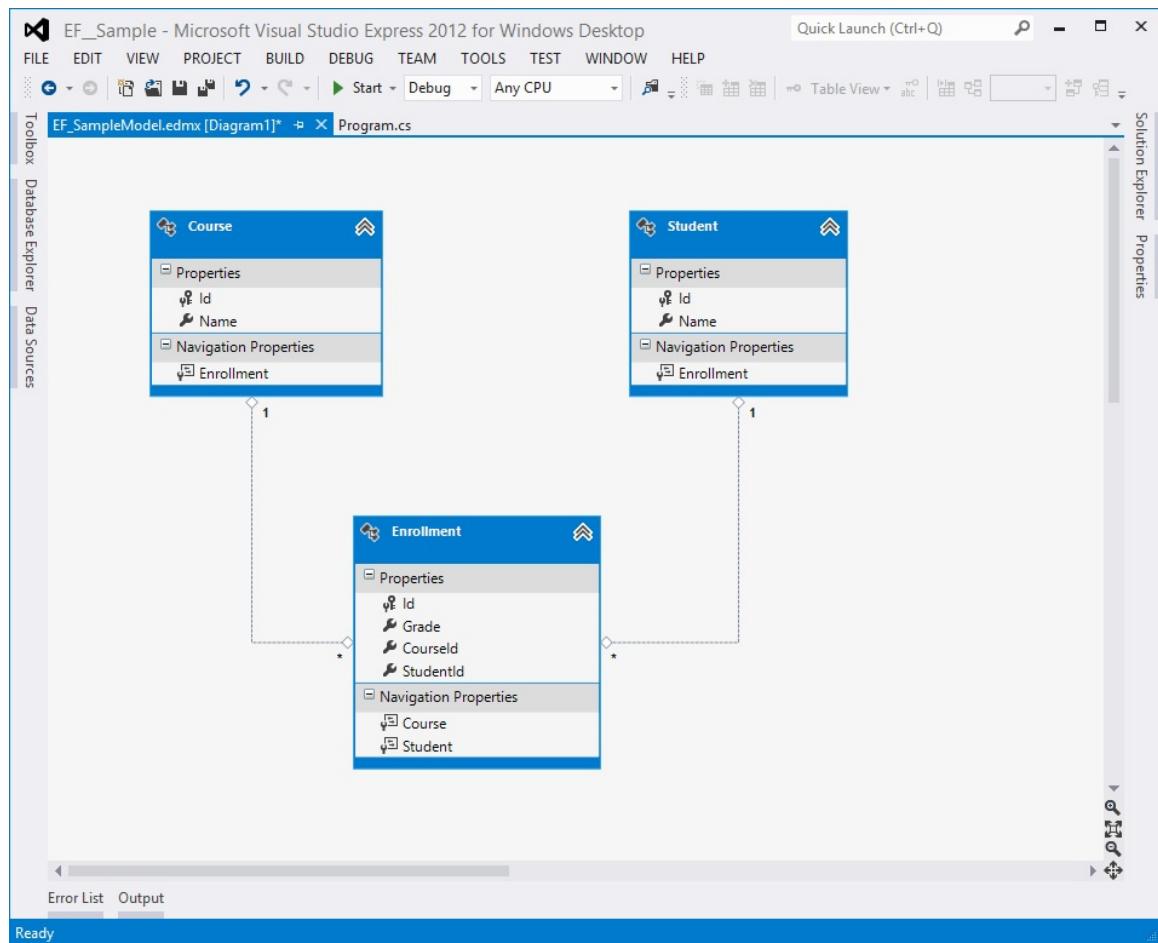
נסתכל-cut על לוח השרוטט כמתואר במאז הבא, ונראה שההווסף קשור של אחד לרבים (*..1) בין יישות קורס לישות הרשמה. נשים לב, שלילישות ההרשמה התווסף מאפיין נוסף בשם courseId בשם CourseId.



באופן דומה נוסיף עטת קשר של אחד לרבים בין יישות סטודנט לישות הרשמה. סוג הקשר הוא אחד מהסיבה של סטודנט מסוים ייתכנו במספר הרשומות, אך כל הרשמה היא של סטודנט מסוים. נמלא את השדות כפי שמצוין בסעך הבא ונלחץ על OK.

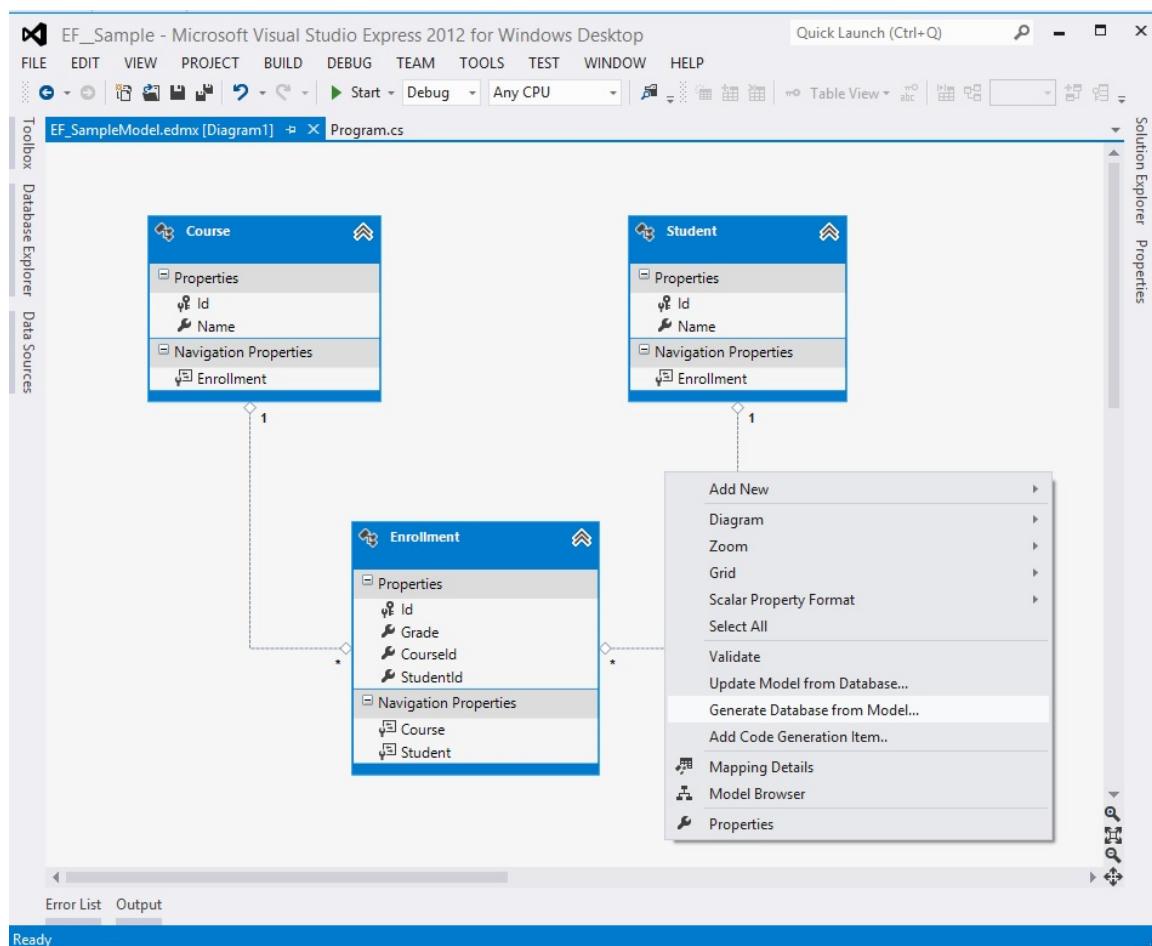


קיבלו אפוא בלוח השרות קשר בין סטודנט להרשמה כמתואר בסע' הבא:

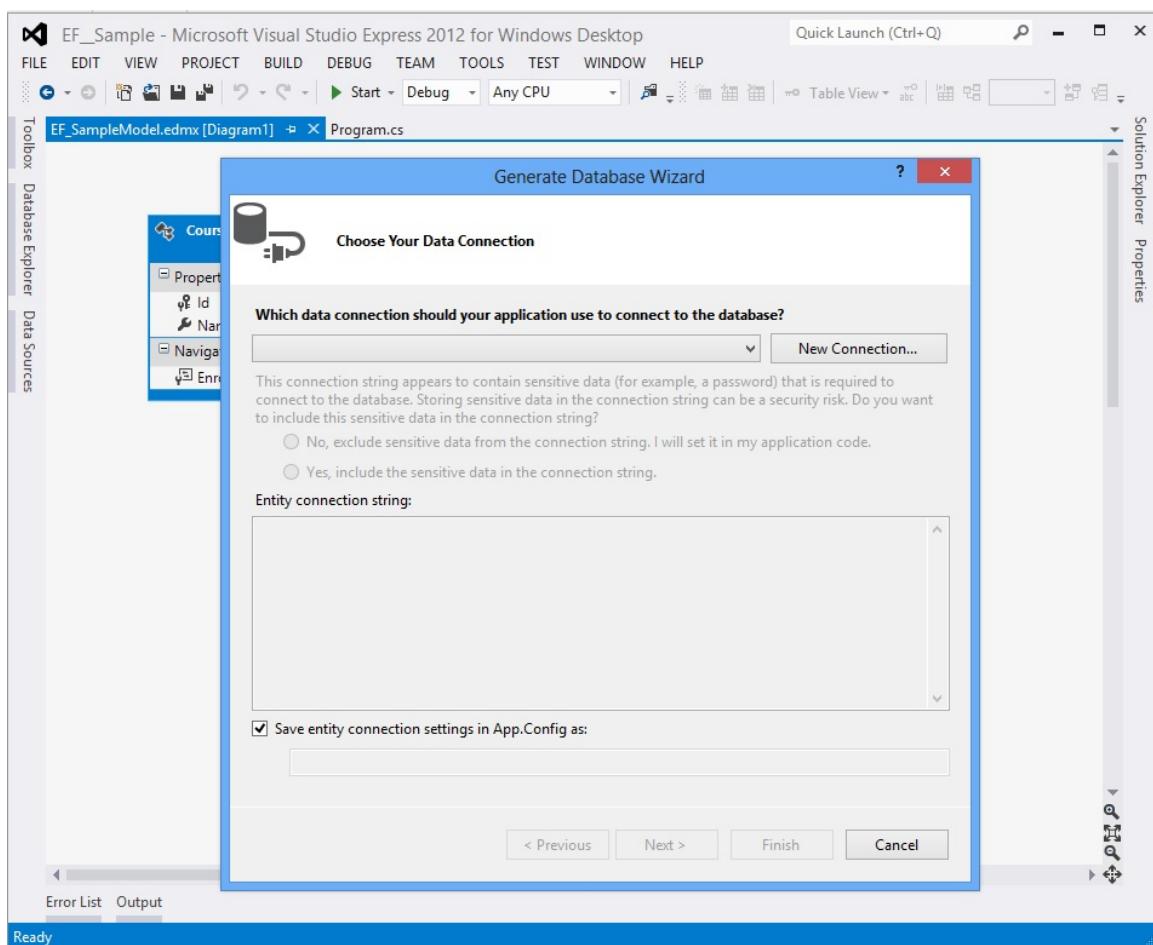


כעת, משהלמנו את ייצרת המודל הקונספטואלי, נרצה לייצר את המודל הפיסי. כלומר, לייצר את בסיס הנתונים והטבלאות שבו יוכל לשמר נתונים של קורסים, סטודנטים והרשות.

לשם כך, נלחץ בשטח ריק בלוח השירות על הלחץ הימני בעבר ונבחר באופציה שנקראת **Generate Database from Model** כמתואר במסך הבא :

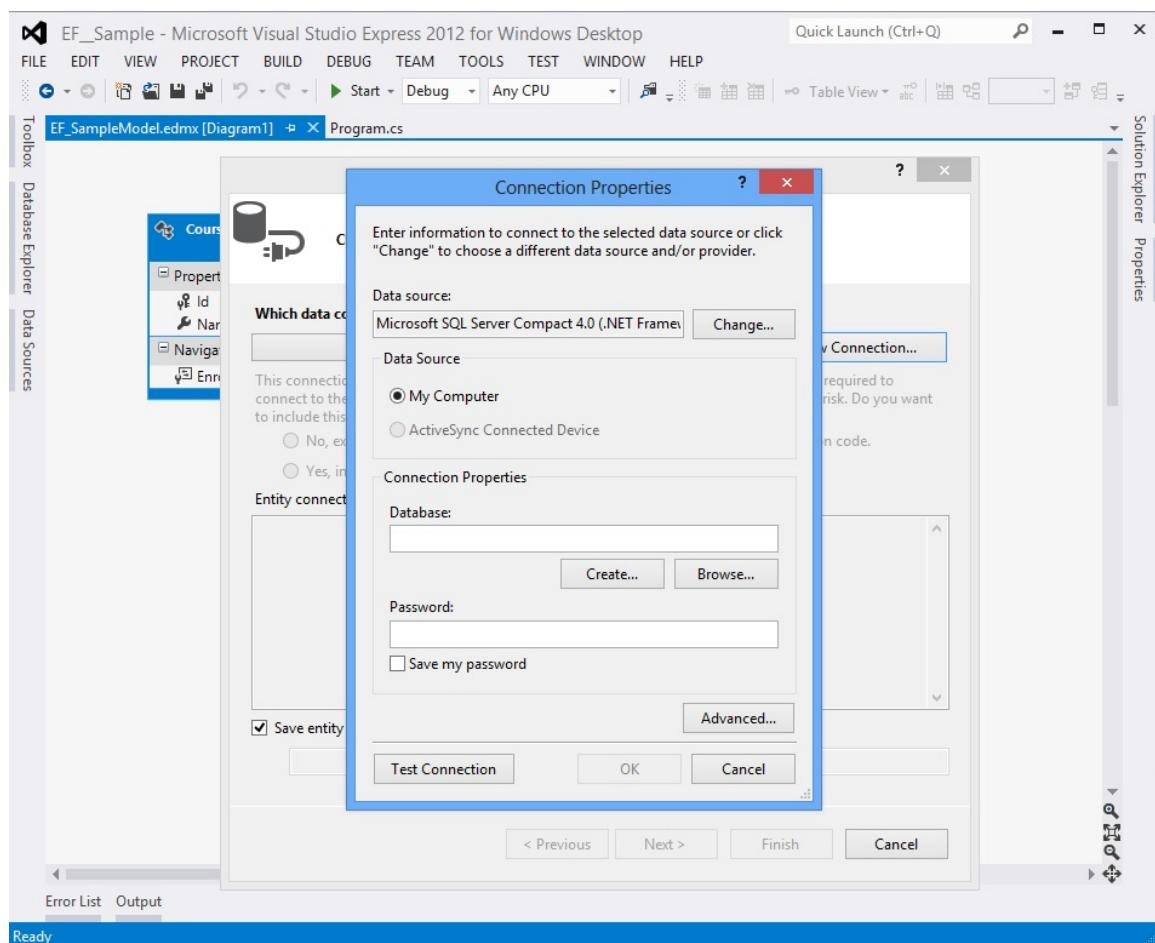


נקבל את המסך הבא, שבו נצטרך לבחור באיזה סוג בסיס נתונים לעבוד וכן ליצור את החיבור אליו.
לחץ על הcptor .New Connection

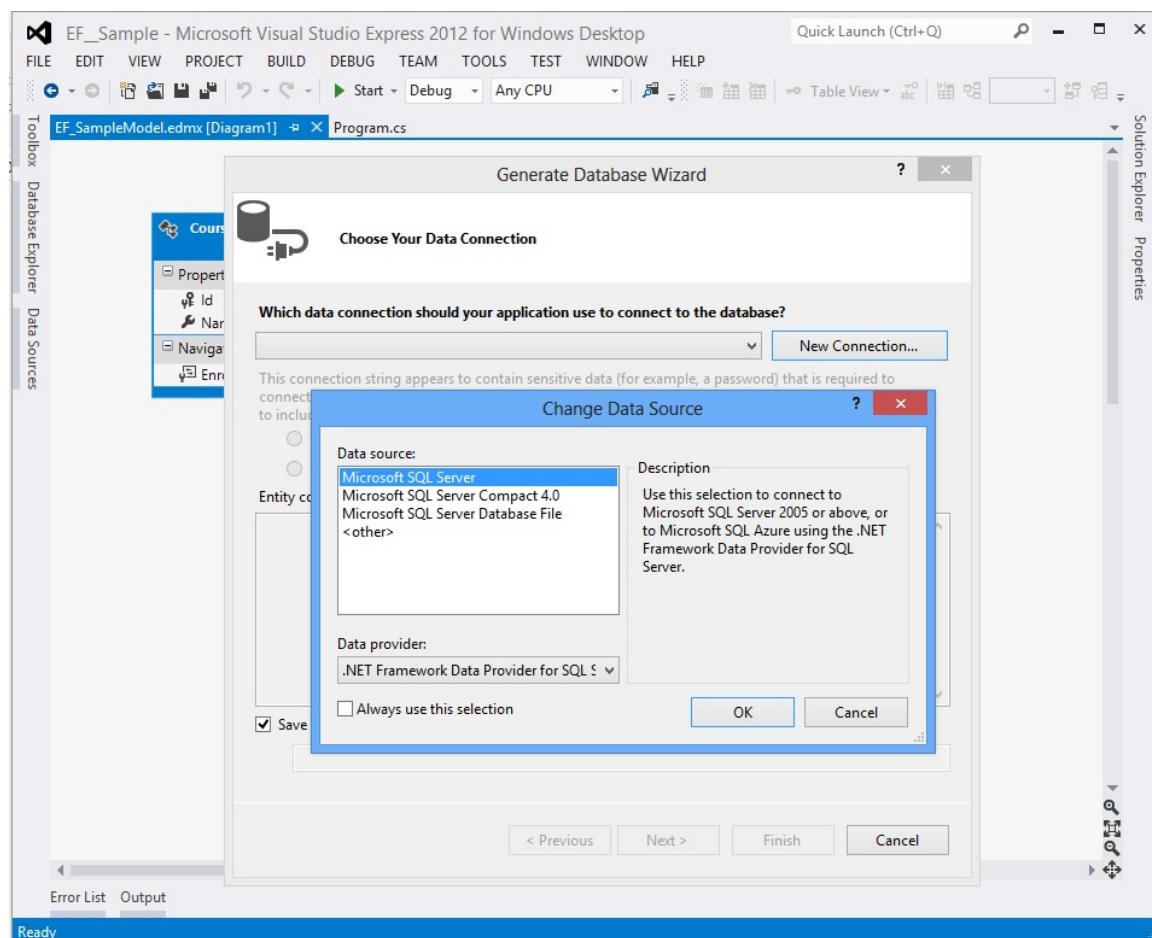


במסך שvíיפתך לנו נלחץ על כפתור Change על מנת לשנות את בסיס הנתונים שאיתו נרצה לעבוד.

בדוגמה זו אנו מעוניינים לעבוד מול בסיס הנתונים Sql Server.

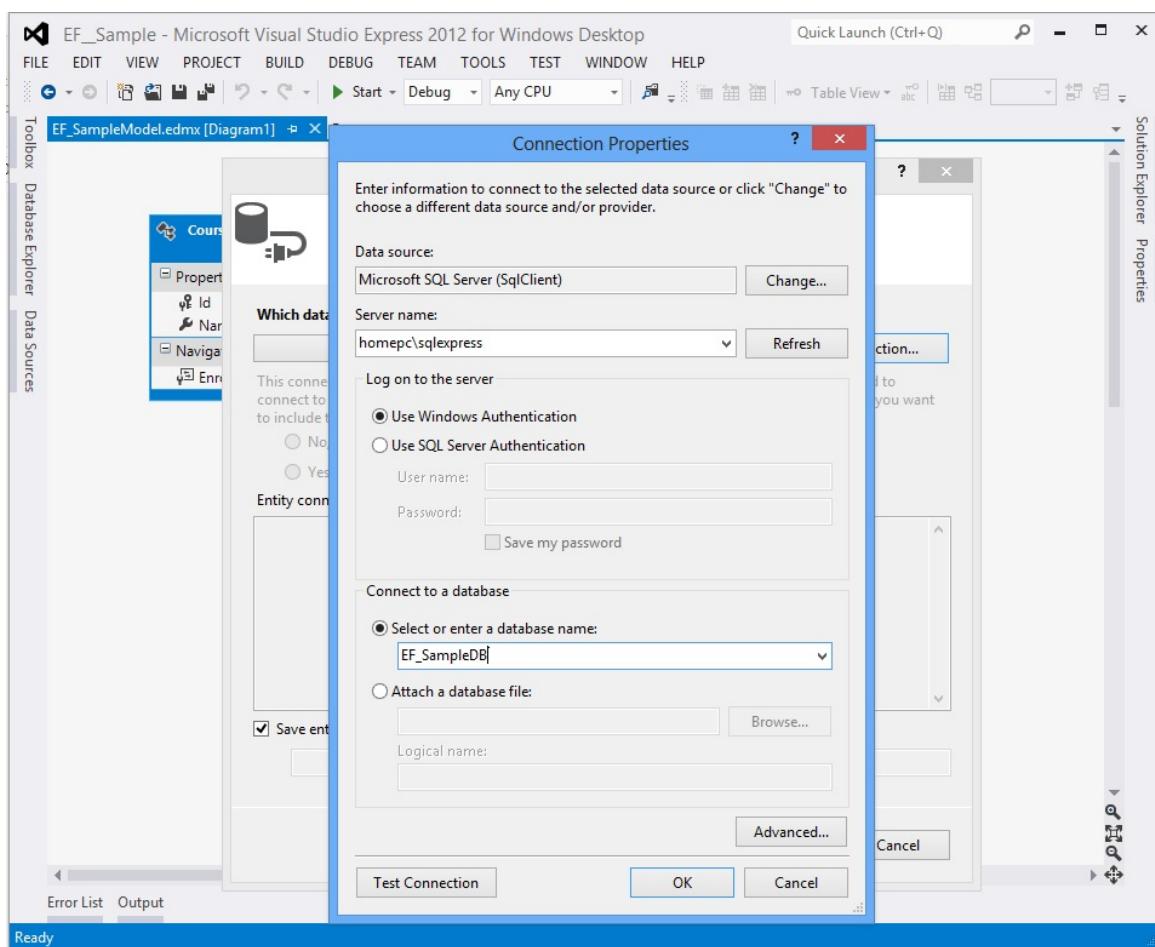


קיבלנו את המספר הבא, ובו נבחר ב- Microsoft SQL Server
ונלחץ על כפתור OK.

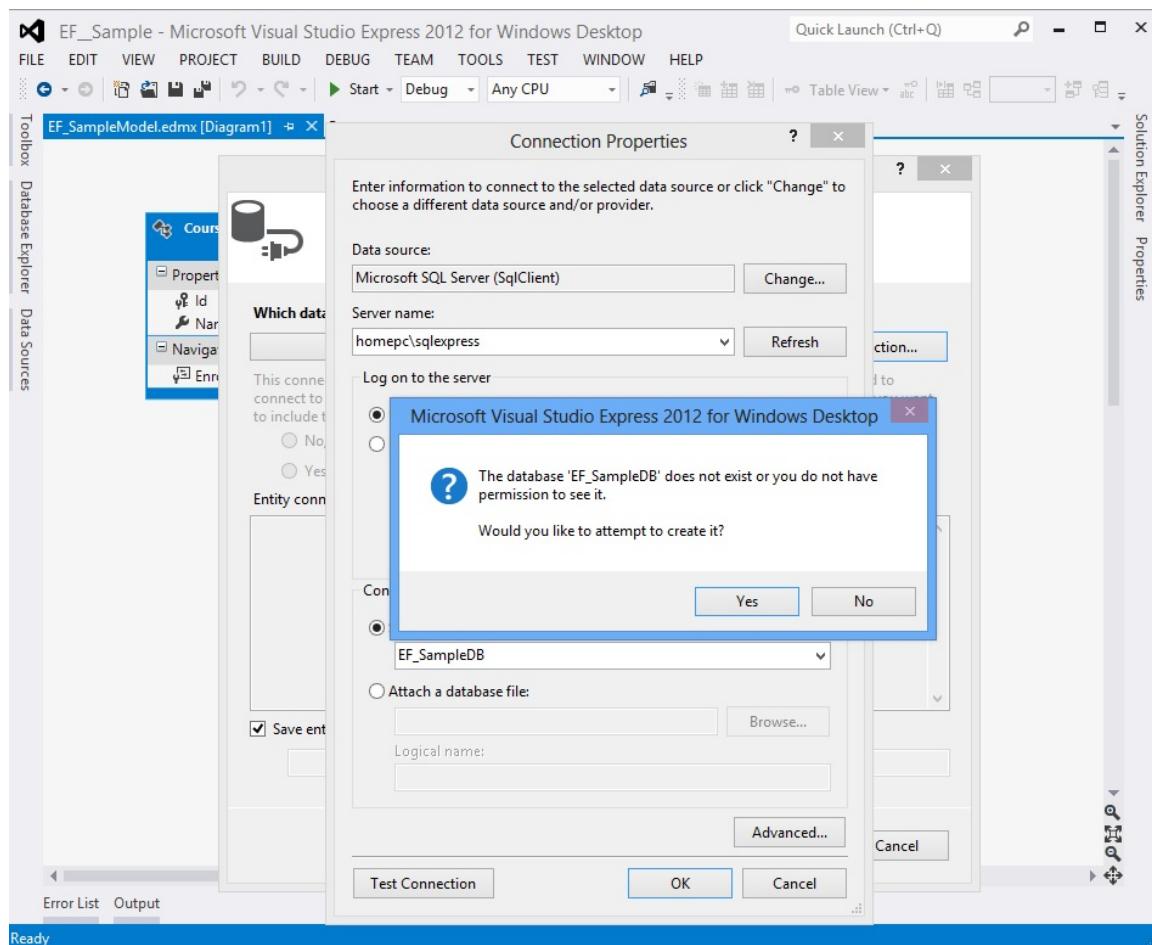


חזרנו למסך שבו סוג בסיס הנתונים נקבע להיות SQL Server, וכעת צריך לספק את שמו של שרת בסיס הנתונים. שמו של השירות מורכב ממשמו של המחשב שלכם (במקרה שלי homepc) לאחריו backslash ולאחר מכן sqlexpress. (ייתכן שבמחשב שלכם צריך להקליד רק את שם המחשב, זה תלוי בהגדרות שביצעתם בזמן התקנת SQL Server עצמה)

בחلك של נקליד בשדה Connect to database את שמו של database name שנרצה שייווצר עבורנו. נקרא לו לצורך הדוגמה .EF_SampleDB נלחץ כעת על כפתור OK.

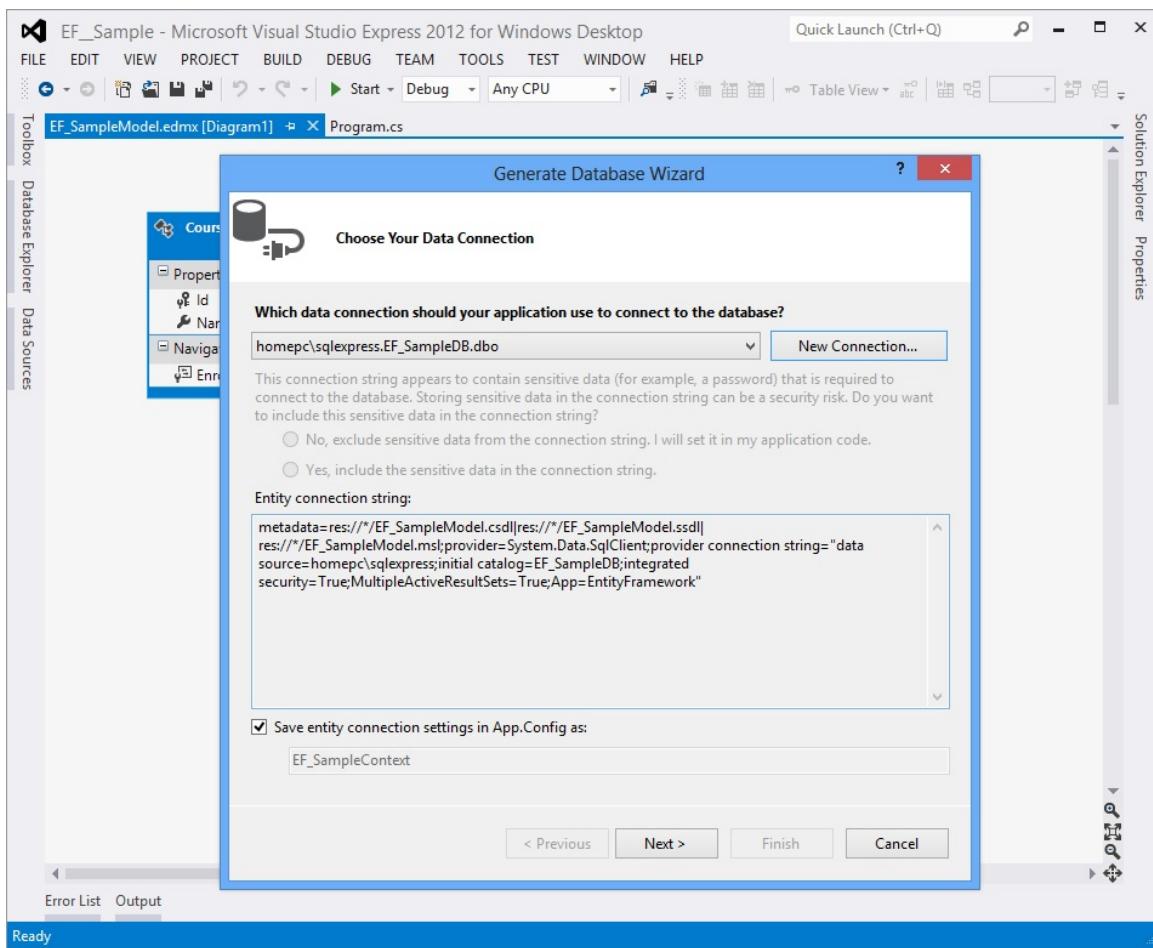


נקבל חלון שמתՐיע שבסיס נתונים בשם EF_SampleDB אינו קיים ונישאל האם ברצונו שהוא ייוצר.
נבחר וナルץ על כפתור Yes.



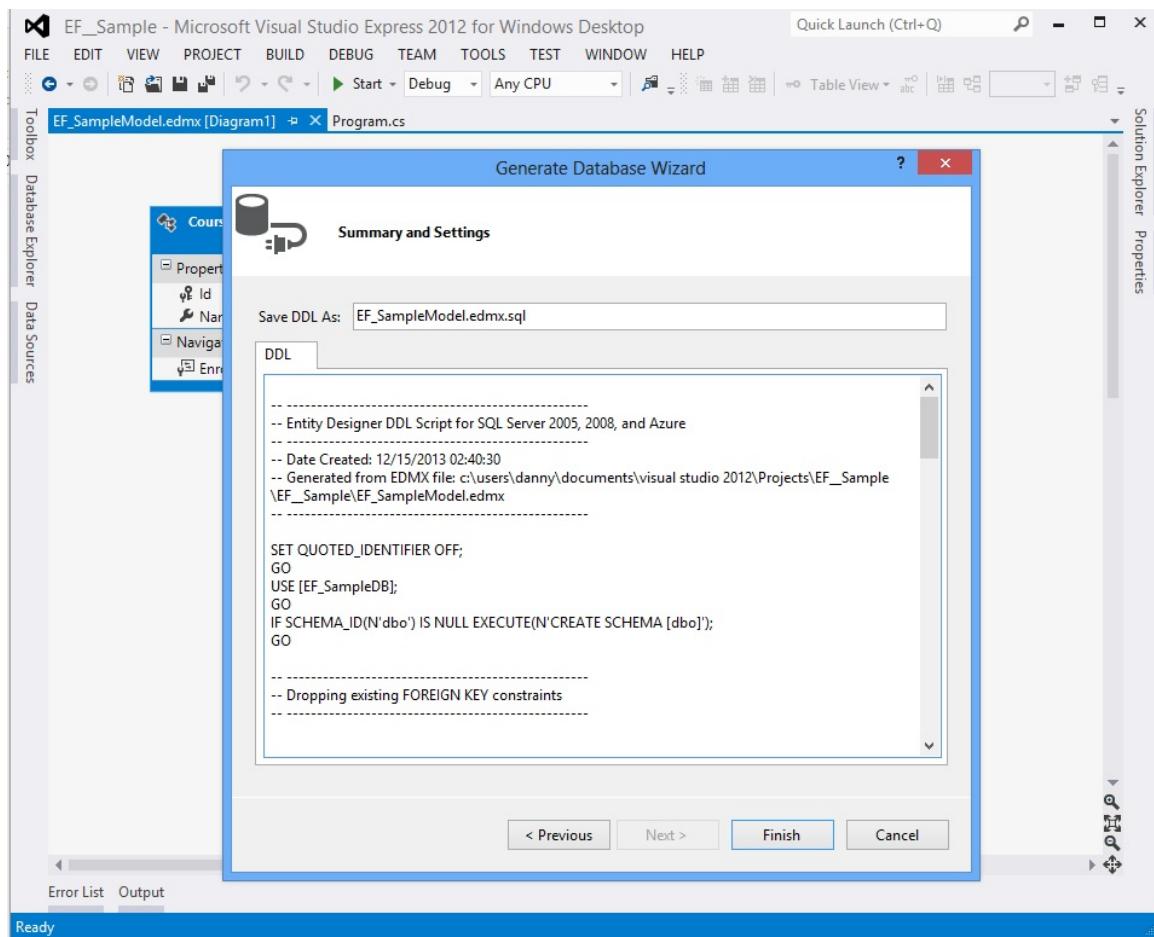
זה מחייב אותנו ללחון הראשון שבו נדרשו לבחור מיהו בסיס הנתונים, וכעת לאחר שmileano את כל הפרטים, קיבלו את ה- Connection String להתחברות לבסיס הנתונים.

נלחץ על כפתור Next להמשך התהליך.

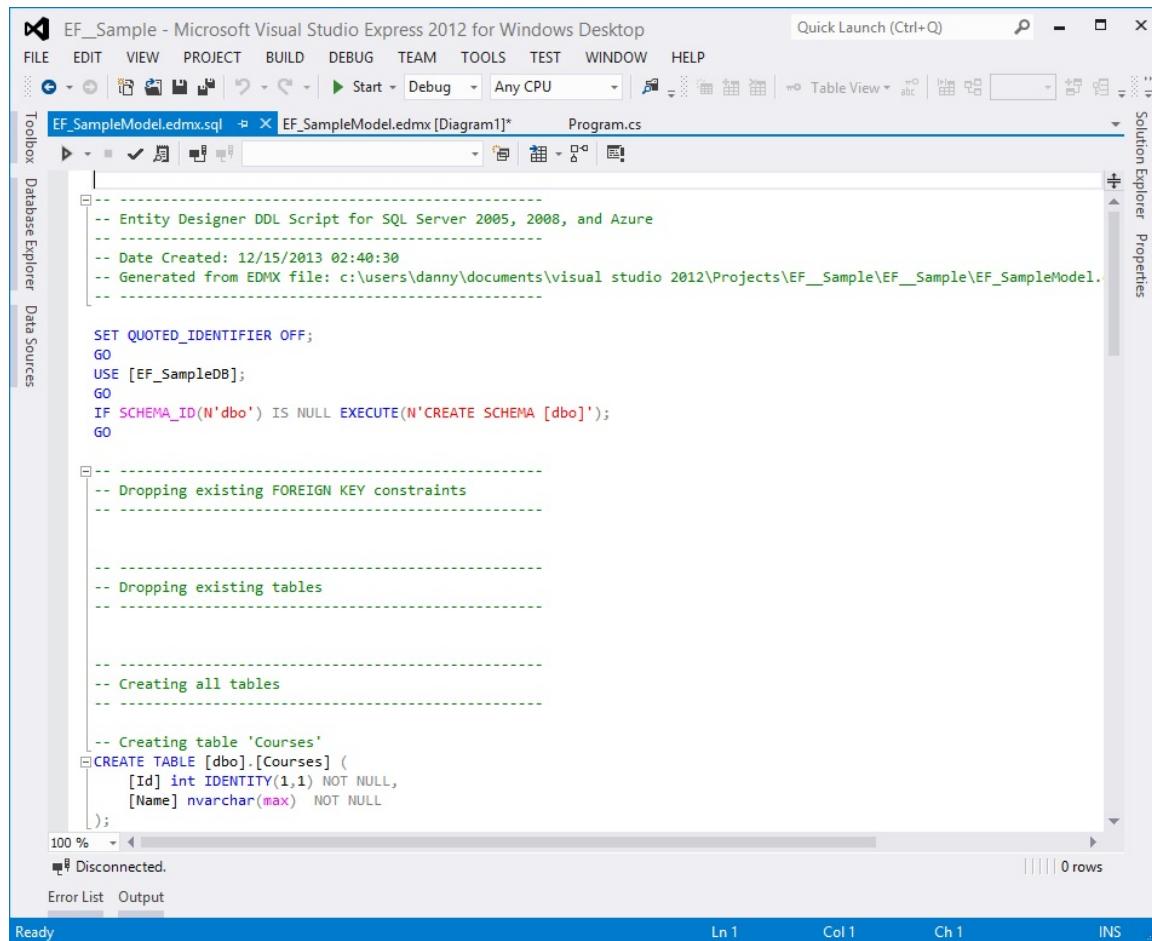


כעת נקבל חלון ש�示ריע על כך שעומד להיווצר עבורנו קובץ DDL שהוא script המכיל פקודות ליצירת בסיס נתונים. את הקובץ שייווצר נctrץ להריץ על מנת שבבסיס הנתונים ייווצר עבורנו.

נלחץ כעת על כפתור .Finish



כפי שהנכם רואים בסעך הבא, לאפליקציה שלנו הותוסף קובץ חדש המכיל את ה-script ליצירת בסיס נתונים. קובץ זה נקרא בשם EF_SampleModel.edmx.sql



The screenshot shows the Microsoft Visual Studio Express 2012 interface. The title bar reads "EF_Sample - Microsoft Visual Studio Express 2012 for Windows Desktop". The menu bar includes FILE, EDIT, VIEW, PROJECT, BUILD, DEBUG, TEAM, TOOLS, TEST, WINDOW, and HELP. The toolbar has various icons for file operations. The main window displays the "Entity Designer DDL Script for SQL Server 2005, 2008, and Azure". The script starts with setting QUOTED_IDENTIFIER OFF, using the [EF_SampleDB] schema, and creating the dbo schema if it doesn't exist. It then drops existing FOREIGN KEY constraints, tables, and finally creates the 'Courses' table with columns Id (int, IDENTITY(1,1), NOT NULL) and Name (nvarchar(max), NOT NULL). The status bar at the bottom shows "Ready", "Ln 1", "Col 1", "Ch 1", and "INS".

```
-- Entity Designer DDL Script for SQL Server 2005, 2008, and Azure
-- Date Created: 12/15/2013 02:40:30
-- Generated from EDMX file: c:\users\danny\documents\visual studio 2012\Projects\EF_Sample\EF_Sample\EF_SampleModel.edmx

SET QUOTED_IDENTIFIER OFF;
GO
USE [EF_SampleDB];
GO
IF SCHEMA_ID(N'dbo') IS NULL EXECUTE(N'CREATE SCHEMA [dbo]');
GO

-- Dropping existing FOREIGN KEY constraints
-- Dropping existing tables
-- Creating all tables
-- Creating table 'Courses'
CREATE TABLE [dbo].[Courses] (
    [Id] int IDENTITY(1,1) NOT NULL,
    [Name] nvarchar(max) NOT NULL
);
```



לחץ בשטח של הקובץ הניל על הלחצן הימני בעבר וນבחר ב- Execute. אופציה זו אמורה להריץ את ה- script וליצור בתוך ה- SQL Server בסיס נתונים חדש.

```

-- Entity Designer DDL Script for SQL Server 2005, 2008, and Azure
-- Date Created: 12/15/2013 02:40:30
-- Generated from EDMX file: c:\users\danny\documents\visual studio 2012\Projects\EF_Sample\EF_Sample\EF_SampleModel.edmx

SET QUOTED_IDENTIFIER OFF;
GO
USE [EF_SampleDB];
GO
IF SCHEMA_ID(N'dbo') IS NULL EXECUTE(N'CREATE SCHEMA [dbo]');
GO

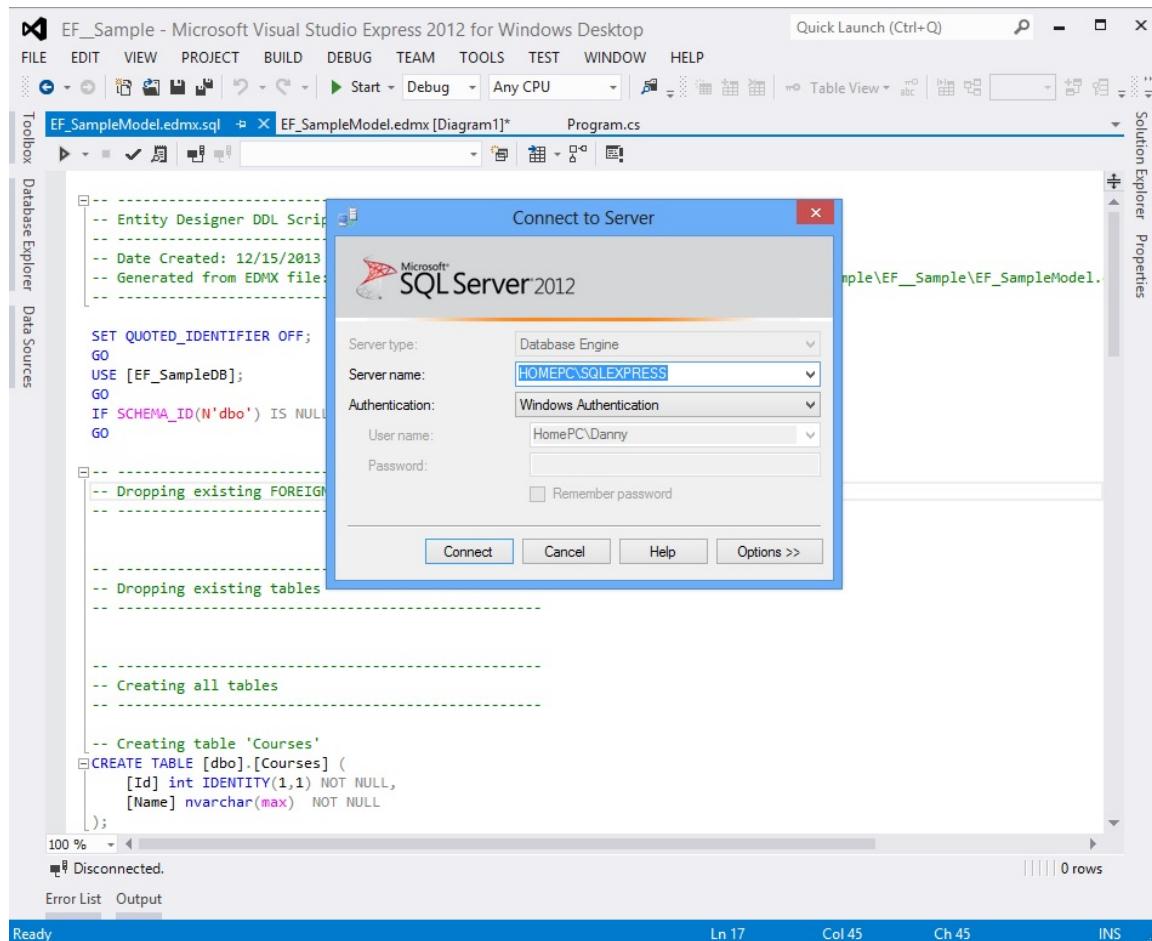
-- Dropping existing FOREIGN KEY constraints
-- Dropping existing tables
-- Creating all tables

-- Creating table 'Courses'
CREATE TABLE [dbo].[Courses] (
    [Id] int IDENTITY(1,1) NOT NULL,
    [Name] nvarchar(max) NOT NULL
);

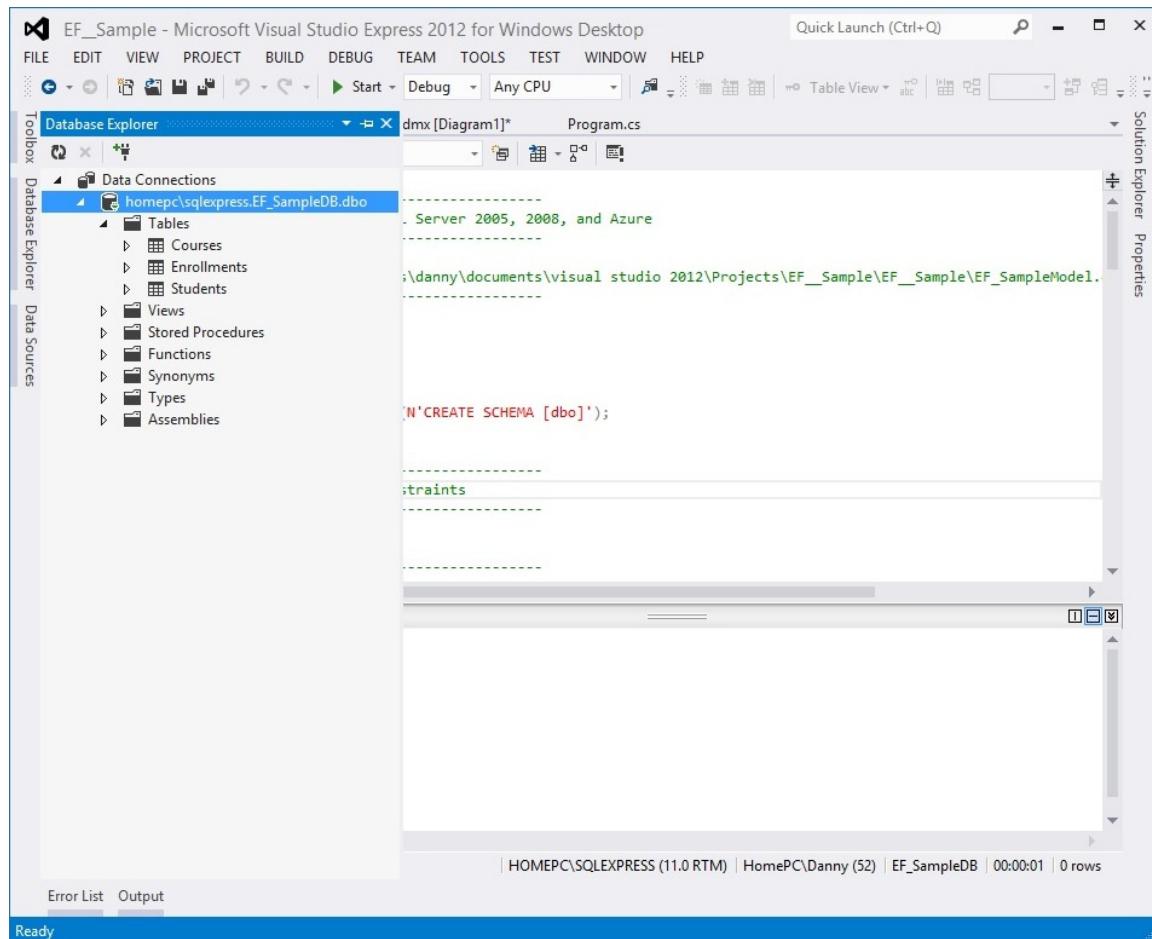
```

נקבל את המסך הבא שבו אנו מאשרים להתחבר לבסיס הנתונים SQL Server על מנת להריץ בו את .script

ליחז על כפתור Connect.



לאחר סיום הריצת ה-script נוכל לראות תחת Database Explorer שנוצר בסיס נתונים חדש בשם Courses, Students, EF_SampleDB ותחת Tables שנוצרו שלוש טבלאות בשם : Courses, Students ו-Enrollments, כמפורט בסעיף הבא.



כמו כן, נשים לב שנוצרו עבורנו מספר מחלקות תחתה - Solution Explorer בשמות:
 Course.cs, Student.cs, Enrollment.cs
 אנו נעבוד מול מחלקות אלה בכל פעם שנרצה ליצור אובייקט מסווג סטודנט, קורס או הרשמה.

The screenshot shows the Microsoft Visual Studio Express 2012 interface. The Solution Explorer on the right lists the project structure:

- Solution 'EF_Sample' (1 project)
 - EF_Sample
 - Properties
 - References
 - EntityFramework
 - Microsoft.CSharp
 - System
 - System.ComponentModel.DataAnnotations
 - System.Core
 - System.Data
 - System.Data.DataSetExtensions
 - System.Data.Entity
 - System.Runtime.Serialization
 - System.Security
 - System.Xml
 - System.Xml.Linq
 - App.config
 - EF_SampleModel.edmx
 - EF_SampleModel.Context.tt
 - EF_SampleModel.Designer.cs
 - EF_SampleModel.edmx.diagram
 - EF_SampleModel.tt
 - Course.cs
 - EF_SampleModel.cs
 - Enrollment.cs
 - Student.cs
 - EF_SampleModel.sql
 - packages.config
 - Program.cs

The SQL Server Object Explorer on the left shows the EF_SampleDB database with tables:

- Courses
- Students
- Enrollments

The SQL Server Object Explorer also shows the following objects:

- EF_SampleModel.edmx
- EF_SampleModel.tt
- EF_SampleModel.Context.tt
- EF_SampleModel.Designer.cs
- EF_SampleModel.edmx.diagram
- EF_SampleModel.sql
- packages.config
- Program.cs

The SQL Server Object Explorer also shows the following objects:

- EF_SampleModel.edmx
- EF_SampleModel.tt
- EF_SampleModel.Context.tt
- EF_SampleModel.Designer.cs
- EF_SampleModel.edmx.diagram
- EF_SampleModel.sql
- packages.config
- Program.cs

אם נסתכל לדוגמה בקובץ Course.cs שנוצרה עבורנו והיא מכילה בין היתר את המאפיינים Id ו- Name שקבעו בזמן יצירת המודל הקונספטוואלי, כמתואר בסעיפים:

הבא:

```

1 ///////////////////////////////////////////////////////////////////////////////
2 // <auto-generated>
3 //   This code was generated from a template.
4 //
5 //   Manual changes to this file may cause unexpected behavior in your app.
6 //   Manual changes to this file will be overwritten if the code is regenerated.
7 // </auto-generated>
8 //
9
10 namespace EF_Sample
11 {
12     using System;
13     using System.Collections.Generic;
14
15     public partial class Course
16     {
17         public Course()
18         {
19             this.Enrollment = new HashSet<Enrollment>();
20         }
21
22         public int Id { get; set; }
23         public string Name { get; set; }
24
25         public virtual ICollection<Enrollment> Enrollment { get; set; }
26     }
27 }
28

```

The Solution Explorer pane shows the following project structure:

- EF_Sample (selected)
 - Properties
 - References
 - EntityFramework
 - Microsoft.CSharp
 - System
 - System.ComponentModel.DataAnnotations
 - System.Core
 - System.Data
 - System.Data.DataSetExtensions
 - System.Data.Entity
 - System.Runtime.Serialization
 - System.Security
 - System.Xml
 - System.Xml.Linq
 - App.config
 - EF_SampleModel.edmx
 - EF_SampleModel.Context.tt
 - EF_SampleModel.Context.cs
 - EF_SampleModel.Designer.cs
 - EF_SampleModel.edmx.diagram
 - EF_SampleModel.tt
 - Course.cs
 - EF_SampleModel.cs
 - Enrollment.cs
 - Student.cs
 - EF_SampleModel.edmx.sql
 - packages.config
 - Program.cs

כמו כן, נסתכל על המחלקה העיקרית שמולאנו לעבוד בכל פעם שנרצה לפנות לבסיס הנתונים, כפי שיתואר בהמשך בדוגמא קוד.

cut נבצע save לכל הקבצים על מנת שהמודול שבנו יישמר.

בשלב זה השלכנו את בניית מודל הנתונים שלנו והוספנו לאפליקציה הrickה שמננה התחלו.

כל שנותרcut זה להדגים כיצד עובדים באפליקציה מול המודול שבנו.

```

1 //<auto-generated>
2 // This code was generated from a template.
3 //
4 //
5 // Manual changes to this file may cause unexpected behavior in your app
6 // Manual changes to this file will be overwritten if the code is regenerated
7 // </auto-generated>
8 //

10 namespace EF_Sample
11 {
12     using System;
13     using System.Data.Entity;
14     using System.Data.Entity.Infrastructure;
15
16     public partial class EF_SampleContext : DbContext
17     {
18         public EF_SampleContext()
19             : base("name=EF_SampleContext")
20         {
21         }
22
23         protected override void OnModelCreating(DbModelBuilder modelBuilder)
24         {
25             throw new UnintentionalCodeFirstException();
26         }
27
28         public DbSet<Course> Courses { get; set; }
29         public DbSet<Student> Students { get; set; }
30         public DbSet<Enrollment> Enrollments { get; set; }
31     }
32 }
33

```

נראה כתת תוכנית המדגימה שימוש במודל הקונספטואלי שבנו. התוכנית מדגימה הוספת רשומות חדשות לטבלאות בסיס הנתונים, פעולה מחיקה, פעולה עדכון, וכן ביצוע שאלתא.
להלן התוכנית ולאחריה תמצאו הסבר לחלקים שונים בה.

```

1  using System;
2  using System.Collections.Generic;
3  using System.Data;
4  using System.Linq;
5  using System.Text;
6  using System.Threading.Tasks;
7
8  namespace EF__Sample
9  {
10    class Program
11    {
12      static void Main(string[] args)
13      {
14        #region adding some rows to DB
15        using (var db = new EF_SampleContext())
16        {
17          #region adding student example
18          ///adding student example
19          Console.WriteLine("enter name of student:");
20          var sname = Console.ReadLine();
21          Student s = new Student { Name = sname };
22          db.Students.Add(s);
23          db.SaveChanges();
24        #endregion
25
26        #region adding course example
27        ///adding course example
28        Console.WriteLine("enter name of course:");
29        var cname = Console.ReadLine();
30        Course c = new Course { Name = cname };
31        db.Courses.Add(c);
32        db.SaveChanges();
33      #endregion
34
35        #region adding enrollment example
36        /// adding enrollment example
37        Enrollment e = new Enrollment();
38        e.CourseId = c.Id;
39        e.StudentId = s.Id;
40        db.Enrollments.Add(e);
41        db.SaveChanges();
42      #endregion adding enrollment example
43    }
44  #endregion
45

```

```

46 #region delete example
47 using (var db = new EF_SampleContext())
48 {
49
50     #region adding another student example
51     //adding another student example
52     Console.WriteLine("enter name of student:");
53     var sname2 = Console.ReadLine();
54
55     Student s2 = new Student { Name = sname2 };
56     db.Students.Add(s2);
57     db.SaveChanges();
58     #endregion
59
60     // delete example
61     Console.WriteLine("enter id of student you want to delete: ");
62     Int32 iddel = Convert.ToInt32(Console.ReadLine());
63     var stud = db.Students.Find(iddel);
64
65     if (stud != null)
66     {
67         db.Students.Remove(stud);
68         db.SaveChanges();
69         Console.WriteLine("student {0} removed", iddel);
70     }
71     else Console.WriteLine("cannot find student with id {0}", iddel);
72
73 }
74 #endregion
75
76 #region update example
77 using (var db = new EF_SampleContext())
78 {
79     // update example
80     Console.WriteLine("enter id of student you want to update: ");
81     Int32 idupd = Convert.ToInt32(Console.ReadLine());
82     var stud2upd = db.Students.Find(idupd);
83     if (stud2upd != null)
84     {
85         Console.WriteLine("enter new name for student " + stud2upd.Name);
86         stud2upd.Name = Console.ReadLine();
87         db.Students.Attach(stud2upd);
88         db.Entry(stud2upd).State = EntityState.Modified;
89         db.SaveChanges();
90         Console.WriteLine("student {0} updated", idupd);
91     }
92     else Console.WriteLine("cannot find student with id {0}", idupd);
93 }
94 #endregion
95

```

```
96     #region query example
97     using (var db = new EF_SampleContext())
98     {
99         // query example
100        var query = from enr in db.Enrollments
101            orderby enr.Id
102            select enr;
103
104        foreach (var item in query)
105        {
106            Console.WriteLine("student {0} is enrolled to {1} course",
107                item.Student.Name,
108                item.Course.Name);
109        }
110    }
111    #endregion
112
113    Console.WriteLine("Press any key to exit...");
114    Console.ReadKey();
115
116    }
117}
118}
```

הסבר התוכנית:

- **שורה 3** – נדרש להוסיף את מרחב השמות System.Data לתוכנית.
- **שורה 15** – מדגימה ייצרת מופיע של המחלקה EF_SampleContext שהיא בעצם המחלקה שדרוכה לבצע פעולה על בסיס הנתונים.
- **שורות 23-29** – שורות אלה מדגימות קליטת מחוזות המייצגת שם של סטודנט שברצוננו להוסיף לטבלת הסטודנטים בסיס הנתונים כסטודנט חדש. לאחר קליטת המחוזות יוצרם אובייקט מטיפוס המחלקה Student (המחלקה שנוצרה אוטומטית עבורנו בזמן בניית המודל) ומאתחלים אותו כך שם הסטודנט יהיה השם של המחוזות שנקלטה. שמו לב, אין צורך לקבוע ערך ל- Id של הסטודנט לאחר שהוא מפתח של הטבלה והוא קיבל כבר ערך אוטומטי (מספר רצ). בשורה 22 ישנה הוספה האובייקט מסוג Student שייצורו לבסיס הנתונים, ולבסוף ביצוע SaveChanges .
- **שורות 33-39** – שורות אלה מדגימות קליטת מחוזות המייצגת שם של קורס חדש שברצוננו להוסיף לטבלת הקורסים בסיס הנתונים. לאחר מכן, ייצרת אובייקט מסוג Course והוספתו לבסיס הנתונים. ולבסוף ביצוע SaveChanges .
- **שורות 35-42** – שורות אלה מדגימות הוספה הרשמה של סטודנט לקורס. שמו לב, לשדה Grade לא ניתן ערך. שדה זה הוגדר בשלבי בניית המודל כשדה שיכול להיות עם ערך null .
- **שורות 50-59** – שורות אלה מדגימות הוספה של סטודנט נוסף לטבלת הסטודנטים בסיס הנתונים בדומה لما שבוצע בשורות 19-23 .
- **שורות 62-72** – שורות אלה מדגימות קליטת Id של סטודנט שברצוננו למחוק מטבלת הסטודנטים בסיס הנתונים. בשורה 64 מתבצעת תחילה בדיקה מול בסיס הנתונים האם קיימים שם סטודנט עם Id כפי שקבענו. אם כן, מוחזר אובייקט מסוג Student עם פרטיו הסטודנט שעומד להימחק. אם לא, מוחזר null . אם הסטודנט נמצא, מבוצעת פעולה מחיקה (shoreה 68) ולאחריה SaveChanges .
- **שורות 77-95** – שורות אלה מדגימות עדכון של רשומה קיימת בסיס הנתונים. השורות מדגימות קליטת Id של סטודנט שברצוננו לעדכן את שמו בסיס הנתונים.
- **שורות 111-117** – שורות אלה מדגימות ביצוע שאלתה באמצעות שיטות LINQ על בסיס הנתונים. במקרה זה השאלה מחזירה את כל הרשומות בסיס הנתונים ממויינות בסדר עולה על פי מספר הרשמה. לאחר מכן, לכל הרשמה מדפיסים את שם הסטודנט ואת שם הקורס שלו נרשם .

הሪיצו תוכנית זו ב-Debug Mode וצפו בטבלאות בסיס הנתונים כיצד הן מתעדכנות לאורך ריצת התוכנית. (לא לשוכח לבצע רענון טרם הצפיה בטבלאות על מנת לצפות בתוכן העדכני).



פרק 12: Windows Presentation – 12 Foundation

12.1 הקדמה

במפגש זה נלמד כיצד לבנות ממשק משתמש גרפי באמצעות שימוש ב-WPF. ממשק גרפי הבניי באמצעות WPF מתיואר על-ידי קובץ XAML (מכונה "zammel") שהם ראשית התיבות של eXtensible Application Markup Language (אך עם הרחבות רבות). זוהי קובץ שודמה במבנה שלו לקובץ XML (אך קובייצי XAML (כגון blend שהוא חלק מסביבת הפיתוח VS2013)).

12.2 הכוונה להכנות ההרצאה

ההרצאה תכלול הסבר והדוגמה "חייה" על אופן בניית ממשק משתמש גרפי פשוט. כמו כן, יינתן הסבר על הפקדים השונים הקיימים וכן תיאור הוספותם למשק פשוט.

12.2.1 נושאים שיש לסקור בהרצאה

- המוטיבציה מאחוריו WPF לעומת WinForms
- קובץ XAML – הסבר על המבנה והתחביר של הקובץ.
- הדוגמת בניית קובץ XAML.
- בניית אפליקציית WPF פשוטה עם שימוש ב-XAML.
- סקירת מגוון פקדים (Controls) והוספותם לאפליקציה.
- תרגיל מסכם – הצגת דוגמה מסכמת שכללת שימוש בפקדים שונים, אנימציות, ויכולות שונות של WPF.

12.2.2ביבליוגרפיה

- פרקים 27-31 בספר הלימוד.
- MSDN
- Code Project
- אתר של MSDN עם המון דוגמאות של ממשקים גרפיים המציגים שימוש בפקדים שונים ויכולות שונות של WPF. לכל דוגמה יש הסבר ודוגמה קוד נלווה.
<http://archive.msdn.microsoft.com/wpfexamples>



File #0003243 belongs to Roei Daniel- do not distribute

يحدة ٤ Case Study –



File #0003243 belongs to Roei Daniel- do not distribute

n-Tier Software – 13 Architecture

13.1 הקדמה

ארQUITטורת n-Tier היא ארכיטקטורת תוכנה שבה האפליקציה מורכבת ממספר שכבות לוגיות מופרדות. כל שכבה מתחברת רק עם שכבה הנמצאת מתחתייה. לכל שכבה תפקיד ברור ומוגדר. שכבות האפליקציה אינן חיבות להיות פיסית על אותו מחשב. כך ניתן להפריד תהליכיים שונים במערכת ולהריץ שכבה מסוימת הדורשת משאים רבים על שרת היכול לספק זאת, בעוד חלקי המערכת האחרים רצים בשרתים נפרדים.

13.2 ארכיטקטורת 3 השכבות

נוהג לפתח אפליקציה ב-3 שכבות עיקריות (3-Tier) לפחות. שכבות אלה מחלקות את האפליקציה לשכבות הבאות:

13.2.1 שכבת תצוגה (PL-Presentation Layer)

שכבה זו אחראית על הצגת הנתונים בלבד. היא מורצת בדרך כלל במחשב הלקוח. לאחר שינוי הפרדה ברורה ומוחלטת בין השכבות, ניתן לשדרג בעתיד שכבה זו כך שתתמוך בצורות הצגת נתונים שונות (ממשק web-י או Win Forms) מבלי לבצע שינויים ביתר השכבות.

13.2.2 שכבת האפליקציה (BL- Business Logic Layer)

שכבה זו אחראית על עיבוד מידע המתקבל משכנת התצוגה (חישובים, בדיקות...) שכבה זו מתחברת עם שכבת הגישה נתונים על מנת לשומר או לקבל מידע לו היא זוקה. ניתן יהיה בעתיד להכניס שינוייםelogיקת המערכת ולשפר תהליכיים שונים, וכל זאת מבלי לבצע שינויים ביתר השכבות.

13.2.3 שכבת גישה לנוטונים (DAL- Data Access Layer)

שכבה זו אחראית על שימרת הנתונים ואחזורים. שכבה זו מספקת את הנתונים לשכנת האפליקציה בצורת אובייקטים, אם כי הנתונים עצם נשמרים בדרך כלל בפורמט שאינו מייצג אובייקט (כפי שראינו בפרק 11, קיימים כלי מיפוי ORM המגשרים על חוסר התאימות הקיימים בין פורמט של אובייקט לבין הפורמט שבו הוא נשמר). ניתן יהיה בעתיד להחליף את הצורה בה נשמרים הנתונים, או את סוג בסיס הנתונים, וכל זאת בצורה שקופה מול יתר השכבות. כל עוד שכבת האפליקציה ממשיכה לקבל את מבקשה משכנת הגישה לנוטונים, אין זה משנה כיצד נשמרים הנתונים.

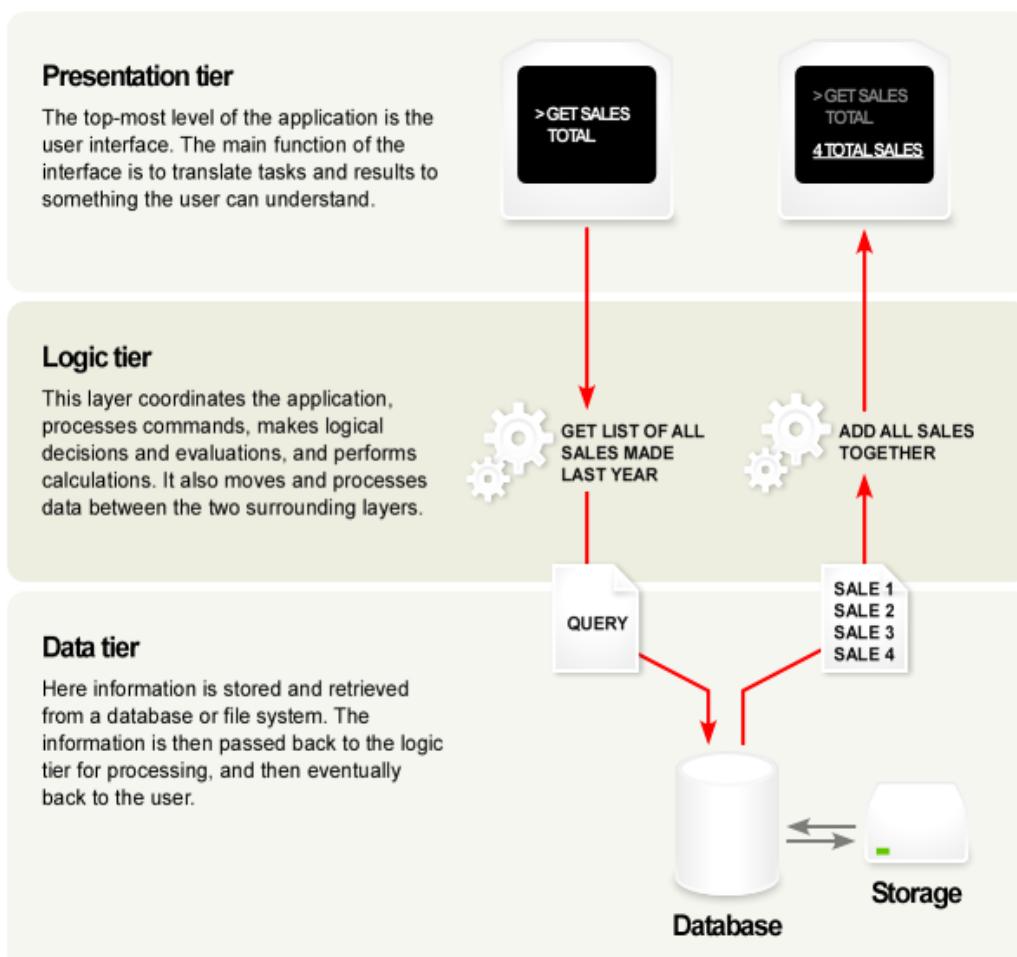


13.3 יתרונות הארכיטקטורה

להפרזה האפליקציה למספר שכבות עצמאיות ישנו יתרונות רבים :

- גמישות לשינויים – כל שכבה יכולה להשתנות בנפרד.
- כוותים שונים יכולים לעבוד על שכבות שונות.
- לכל שכבה היכולת לגודל בנפרד.
- המערכת מסודרת ומודולרית.
- קלה לתחזוקה.
- המערכת אמינה יותר.
- עומדת ביעדים של הנדסת תוכנה.

להלן תרשימים המתאר את המבנה של מערכת הבניה בצורה 3-Tier. התרשימים ממחיש את התפקיד של כל שכבה ואת אופן האינטראקציה שלה מול שכבה סמתחתיה (ראה גם קישור 3 ברשימה הביבליוגרפיה) :



13.4 הכוונה להכנות הרצאה

במסגרת הרצאה יוצג פיתוח של פרויקט קטן בהיקפו, הבנוי על פי ארכיטקטורת 3-Tier. הסטודנט המרצה על נושא זה יפתח שכבת BL המביאה לידי ביטוי נושאים קודמים שנלמדו, כגון: תבניות עיצוב, ירושה, פולימורפיזם, שימוש במנשקים. יתר השכבות ימומשו באופן בסיסי ביותר. דהיינו, שכבת התצוגה תציג את המידע ל-Console, וscriçãoת הגישה לנוטונים תשמר את הנוטונים באמצעות סריאלייזציה ל-XML. כך שההתמונות בהרצאה זו צריכה להיות בשכבה ה-
.BL.

לאחר מכן, תודגם החלפת שכבת התצוגה מהתצוגת Console לתצוגה גרפית נאה ונוחה יותר לשימוש באמצעות WPF. כמו כן, תוחלף שכבת הגישה לנוטונים בשכבה חדשה שבה הנוטונים נשמרים בבסיס הנתונים SQL Server. את שכבת הנתונים ניתן לבנות באמצעות ADO.NET או באמצעות שימוש באחד הכלים ORM שנלמדו. כך נקבל מהשנה לפיתוח מערכת בשכבות, וסדרוג שכבות בעתיד ללא תלות בשכבות האחרות.

13.4.1 נושאים שיש לסקור בהרצאה

- הצגת מודל השכבות והשוואה למודלים אחרים (יתרונות וחסרונות).
- הסבר על תפקיד שכבת התצוגה.
- הסבר מפורט על תפקיד שכבת האפליקציה.
- הסבר מפורט על תפקיד שכבת הגישה לנוטונים.
- הצגה והסביר מפורט של דוגמת הפרויקט הבנוי בארכיטקטורת 3-Tier.



13.4.2 ביבליוגרפיה

- <http://www.developerfusion.com/article/3058/boosting-your-net-application-performance/2/>
- <http://www.15seconds.com/issue/011023.htm>
- <http://encyclopedia.thefreedictionary.com/n-tier+architecture>
- <https://secure.codeproject.com/KB/cs/NTier.aspx>
- http://kula.student.usp.ac.fj/class-shares/CS313/Notes/3-tier_architecture/three_tier_architecture.asp.htm

יחידה ה – פרויקט הגמר



File #0003243 belongs to Roei Daniel- do not distribute

פרק 14 – פרויקט הגמר

במסגרת הסדנה יינתן לכם נושא לפרויקט (בצורת דרישות לכוח). יהיה عليיכם לאפיין ולנתח את הנושא, וכן לעצב ולממש פתרון על סמך הכלים שנלמדו במהלך הסדנה.

על כל אחד משלבים אלה יינתן ציון, ויחד הם יהוו את הציון עבור הפרויקט (המהווה 70% מהציון הכללי בסדנה).

באטר הקורס יופיעו נושאים אפשריים לפרויקט לבחירתכם. כמו כן, ישנה אפשרות למשתמש בפרויקט עצמאי המוצע על ידיכם, בתנאי שיעמוד בהיקף ובדרישות הסדנה. הגשת הפרויקט כוללת את הצגתו ואת ההגנה עליו.

את הפרויקט יש להגיש במספר שלבים. בכל שלב תידרשו להגיש מסמך מותאים. על מסמכים אלה לקבלו ציון שיהווה חלק מהציון הסופי בסדנה.

1.4. שלבי הגשת הפרויקט

הגשת פרויקט הגמר תבוצע בשלושה שלבים:

- בחירת נושא לפרויקט והגשת מסמך רקע. (מהווה 2% מהציון על ממן 12)
- הגשת מסמך ניתוח אפיון. (מהווה 38% מהציון על ממן 12)
- הגשת מסמך עיצוב ותיכון. (מהווה 60% מהציון על ממן 12)
- מימוש והגשת מסמך סיכום (ממן 13). מסמך הסיכום יכולול:
 - מסמך האפיון המתוקן.
 - מסמך העיצוב המתוקן.
 - מימוש המערכת.
 - מדריך למשתמש.
- הצגת הפרויקט והגנה עליו.



14.2 הנחיות כלליות

14.2.1 הנחיות כלליות לבחירת נושא

נושא הפרויקט שתבחרו למשרץ צריך לכלול שימוש בנושאים הבאים :

- מנשך משתמש גרפי.
- הפרויקט יעבד מול בסיס נתונים וישתמש בכלים שספקת שפת C# לעובדה עם בסיסי נתונים. (ניתן לעשות שימוש בכלים ORM חיצוניים כפי שנלמדו בסדנה).
- בעיצוב הפרויקט ומימושו נדרש להביא לידי ביטוי את הנושאים שנלמדו בסדנה. בייחוד נדרש הקפדה על עקרונות של תכונות מונחה עצמים, שימוש בדפוסי עיצוב רלוונטיים, ופיתוח לפי שכבות.

14.2.2 הנחיות כלליות לאופן העבודה על הפרויקט

- את הפרויקט ניתן להגיש בזוגות.
- נדרש לתת את הדעת על שינויים עתידיים אפשריים.
- מומלץ ורצוי לעבור טוב על הניתוח ועל התכנון לפני מתחilibים בביוץ. מומלץ להתייעץ עם אנשים רבים ולעיין בספרים רבים.... אין להתחיל בפיתוח לפני הושלם התכנון.
- שאלות לגבי הפרויקט ניתן וארכוי להפנות לקבוצת הדיון של הקורס. בכלל, מומלץ להיכנס מדי פעם לקבוצת דיון זו, ולהציג על השאלות. השאלות (והתשובות עליהם) עשויות לחסוך לכם לא מעט זמן בשלב התכנון, ואולי אף בשלב התכנון של הפרויקט.

14.2.3 הנחיות כלליות לאופן הגשת הפרויקט

- הפרויקט יהיה מתווד היטב, ומלווה בתרשיimi UML רלוונטיים לתיאור מבנה המערכת ותפקודה.
- הפרויקט יוגש בשלושה חלקים :
 - הגשת מסמך אפיון וניתוח (OOA) – פירוט בהמשך.
 - הגשת מסמך עיצוב (OOD) – פירוט בהמשך.
 - מימוש הפרויקט וכנתיבת מדריך למשתמש.

- הגשתו הסופית של הפרויקט תכלול את ההגשות הקודמות כאשר הם מאוגדות יחד בתיק פרויקט מסודר.
- על מדריך המשמש לכלול:
 - פירוט כל הצעדים הנדרשים על מנת להפעיל את המערכת.
 - כל מידע נוסף הנדרש להפעלת המערכת, כגון: קודים לכינסה וכדומה.
 - תיאור של דוגמת מינימום, על מנת לאפשר לבדוק לדעת איזה מידע הוא יכול לשלוּף מותוך דוגמה זו.

14.3 נקודת למחשבה

בספרם *Case studies in object oriented analysis and design* מספרים יורדן ורגיליה את הסיפור האמייתי הבא:

במדינה אירופאית, המוסד לביטוח לאומי החליט להתקין מערכת מידע חדשה, במקומות ישנה ומסורבלת קיימות. הובאו מנתחי מערכות מומחמים, שהחליטו שיש להציג את המערכת החדשנית בגישה מונחת עצמים. במסגרת הניתוח הוגדרו מחלקות של "אזור", "פנסיה", "موظב" ועוד מחלקות רבות נוספות. אולם, זמן מה לאחר הפעלת המערכת החדשנית התגללה הכישלון בניתוח המערכת: רוב פעולות התחזוקה במערכת נבעו משינויים בחוק הביטוח הלאומי. הבעייה הייתה שבעת הגדרת המערכת לא הוגדרה מחלוקת חוקים. השפעות החוקים השונים על המערכת נטמו בכל חלק המערכת, וכך נדרש שינוי היקפיים בכל חלק המערכת עבור כל שינוי של חוק.

סיפור זה בא להבהיר שתי נקודות עיקריות.

הראשונה היא שיטות פיתוח מונחות עצמים מכילות אמנים בחובן תקווה לשיפור ממשמעותי בתהיליך פיתוח תוכנה, ביכולת לשימוש חזר, ובקללה על תהליך ביצוע הרחבות בעתיד, אך זאת בתנאי שבחירה המחלקות נעשתה באופן נכון!!!

הנקודה השנייה היא שבעת תכננו מערכת מונחת עצמים, אין זה מספיק לייצג את כל פרטי המערכת הקיימת באותו רגע באופן יעיל, אלא יש צורך להזכיר זמן ומחשבה לשינויים עתידיים אפשריים, שאת אופיין לפחות ניתן אולי לחזות באופן חלקי בעת תכנון המערכת.

לאור עובדה זו, במסגרת פרויקט זה עליכם להציג שני שינויים היגיוניים אפשריים. בתיק התכנון עליכם להציג שינויים עתידיים אפשריים אלה, ולפרט כיצד המערכת שאותה תכננתם ומימושם תומכת ביצוע פשוט של שינויים עתידיים אלה (או במילים אחרות, לתכנן את המערכת באופן כזה ששינויים עתידיים אפשריים אלה יהיו ניתנים להוספה בנסיבות יחסית).



פרק 15 – מסמך האפיון והניתוח

- מסמך האפיון והניתוח יהיה 40% מהציוון של תכנון הפרויקט (ממ"נ 12).
- המסמך יכלול הסבר מלא ומפורט של המערכת כלפי הלוקו.
- עליים ללימוד את הנושא של הפרויקט, ולנתח את הדרישות השונות, כולל אלה שלא נכתבו במפורש. (כלומר, עליים לייצג הן את צד הלוקו המציב דרישות ותיאור של המערכת, והן את הצד המנחה שלומד את דרישות הלוקו ומוסיף דרישות נוספות שמתבקשות בלי שהлокו ציין במפורש).
- המסמך לא אמור להכיל שום פרטי עיצוב ומימוש (מחלקות, ירושות...) אלא מסמך עבור הלוקו (שאינו מבין דבר בתכנות), כך שאם הלוקו יקרא מסמך זה, הוא יבין את המערכת שתאחס עתידיים לבנות עבورو וייתן את הסכמתו לכך.
- מסמך הניתוח יכלול:
 - תיאור הבעיה.
 - תיאור של מרכיבי המערכת השונים.
 - תיאור מפורט של סוגים המשמשים השונים במערכת, והפעולות שכל אחד מהם יכול לבצע.
 - דיאגרמות use case diagram ו- activity diagram לתהליכיים העיקריים במערכת.
 - הדוגמת מסכי המערכת לлокו – לשם המבנתה המערכת שתיבנה לлокו, יתוארו בΖורת ויזואלית עבור כל סוג משתמש המסתכנים השונים שיהיו במערכת (כולל נתוניים). כך יקבל הלוקו המכחשה למערכת שתיבנה עבورو. (עם זאת, צורת המסתכנים ותוכנם יכולים להשתנות בשלבי הפיתוח).
 - מדריך למשתמש.
- במסגרת הבדיקה הסופית של הפרויקט, תיבדק מידת התאמתו למסמך האפיון והניתוח עליו התחייבתם.

פרק 16 – מסמך עיצוב ותיקון

- מסמך זה (תיק תכנון הפתרון) מהוות 60% מהציון עבור ממ"ן 12.
- מסמך העיצוב יכול תחילת את מסמך האפיון והנitionה המתוקן.
- על המסמך להכיל תיאור מפורט של מבנה התוכנה, מבחינות: מנשכים, מחלקות וקשרים ביןיהם, פיתוח בשכבות, **מבנה עיצוב** (Design Patterns) ועוד.
- נדרש לעשות שימוש בדיאגרמות UML מתאימות (Use Case Diagram, Class Diagram, Activity Diagram, Sequence Diagram).
- המסמך יכול דיאגרמות הממחישות הן את מבנה המערכת והן תהליכי חשובים המתרחשים בה.
- המסמך יכול תיאור מפורט של המחלקות, המנסכים ושאר רכיבי המערכת.
- לכל מחלקה יתוארו: היררכיות הירושה שלה, תוכנה (משתנים ומתחוזות) וקשריה עם מחלקות אחרות.
- תיאור של מוסכמות שמורות במערכת.
- תיאור הנחות עבודה.
- מסמך העיצוב יכול תיאור של שינויים עתידיים אפשריים במערכת וסביר על אופן הטענתן במערכת.
- העיצוב צריך להיבנות באופן גמיש כך שלא יהיה תלוי בIMPLEMENTATION ספציפי של רכיב תוכנה כזה או אחר. הכוונה היא שיש לעצב את הפתרון ללא תלות, למשל, בסוג בסיס הנתונים שבו ייעשה שימוש, או בצורה הצגת הנתונים (מנשך WPF או מנשך Web).
- במסגרת הבדיקה הסופית של הפרויקט ייבדקו כלל המימושים ומידת התאמתם למסמך העיצוב.



פרק 17 – מימוש והגשה הפרויקט

- הגשה זו היא ההגשה הסופית של הפרויקט ומהווה 35% מהתקציב הכללי, ונחשבת כמטלה 13.
- ההגשה תכלול את הפריטים הבאים:
 - מסמך האפיון והניתוח המתוקן.
 - מסמך עיצוב הפתרון המתוקן.
 - הוראות התקינה ומדריך למשתמש.
 - קוד התוכנית.
- לאחר הגשת הפרויקט תזומנו לבדיקתו. הבדיקה תכלול הרצה והדגמה של המערכת, וכן הסבר על חלקים נבחרים במסמך העיצוב ויישומים בקוד.

פרק 18 – פרויקט לדוגמה

כפי שהוסבר קודם, הגשת הפרויקט נעשית במספר שלבים: הגשת הצעת פרויקט וקבלת אישור לבניה, הגשת מסמך ניתוח ואפיון של המערכת, הגשת מסמך עיצוב של המערכת, ולבסוף מימוש המערכת והגשת תיק פרויקט המכול את כל השלבים.

להלן דוגמה למסמך אפיון וניתוח לדוגמה כפי שהוגש באחד הסטודנטים הקודמים. מטרת דוגמה זו היא לספק כיוון וגישה לבניית מסמך אפיון וניתוח, אם כי זהה זהה מחייבת וניתן לעצב את המסמך באופן שונה שימוש בכל תיעוד היוצרים מסמכים באופן אוטומטי.

מסמך האפיון כולל גם מדריך למשתמש בליויי מסכימים המדגימים כיצד תיראה המערכת הסופית.

בעמודים הבאים תמצאו:

- מסמך אפיון לדוגמה עבור מערכת ניהול עסקים קטנים.
- מדריך למשתמש של מערכת זו.
- מסמך עיצוב (תיקון) מפורט עבור מערכת זו.



File #0003243 belongs to Roei Daniel- do not distribute

SmartBiz

מסמך אפיון וביתוח

גרסה 2.0

شمונות הסטודנטים:

רותם אלישדה

עומר בר-און



File #0003243 belongs to Roei Daniel- do not distribute

תוכן עניינים

145	כללי
146	יעוד המערכת
146	דרישות המערכת
150	מרכיבי המערכת
150	משתמשי המערכת
153	דיאגרמות Use-Case
153	ניהול משתמשים במערכת
154	ניהול לקוחות
155	ניהול מלאי וזמן ציוד
156	ניהול תעריפים ותמחור
158	רכישת שירות או מוצר
159	חלוקת עבודה
160	דיאגרמות Activity
160	רכישת שירות / מוצר
161	חלוקת עבודה
162	ניהול מלאי וזמן
163	ניהול לקוחות



File #0003243 belongs to Roei Daniel- do not distribute

כללי

מערכת זו מיועדת לניהול עסקים קטנים.

מדובר במערכת גנרייה המתאימה לרוב סוגי העסקים הקטנים מכיוון שהיא מספקת פונקציונאליות כללית הדורשת במרבית העסקים שמהירים מוצרים או נותנים שירות.

המערכת נותנת מענה לעסקים קטנים שכיום לא משתלים לבנות עבורים מערכת מידע ייעודית שתאפשר להם ניהולiesel של העסק, אך אף על פי כן, הם יכולים להיעזר במערכת הנ"ל בצד לשפר את ביצועיהם העסקיים.

כיום, עסקים קטנים אשר לא עושים שימוש במערכת מידע ייעודית, לא יכולים לבצע מעקבiesel אחר ציבור ללקוחותיהם, המלאי הזמן, הכנסות, ההוצאות ועוד.

כתוצאה לכך, מתרחשת פגיעה בתפקיד העסקים, למשל, בעיות בשימור לקוחות, חסרים המלאי זמן, הוצאות מנויפות לא בקרה, והכנסות שאינן מספיקות לפועלות השוטפת של העסק.

המערכת הגנרייה תומכת בסוגי משתמשים שונים בשלושה דרגים: **הנהלה** – מנהל העסק. **تיאום ובקרה** – ראש צוות, מנהל מכירות. **תפועל** – עובד צוות.

מלבד פיתוח המערכת הגנרייה, אנו נדגים שימוש במערכת עבור "מעבדת מחשבים פרטית". נთאר בקווים כלליים את פרטי העסק.

מעבדת המחשבים עוסקת במכירת ציוד-קצה ובתיקונים.

בראש המעבדה עומד **מנהל המעבדה** אשר אחראי על תפעול המוצרים, קביעת מוצריהם, בקרה על המלאי וניהולה השוטף. תחתיו עומד **ראש צוות טכנאים** אשר אחראי על חלוקת העבודה בין הטכנאים ומעקב אחר ביצועה. תחתיו עומדים **טכנאי מעבדה** אשר מבצעים את התיקונים ויכולים לעקוב/לעדכן עבודות אשר שיוכו להם. כמו כן, במעבדה עובד **מנהל מכירות שתפקידו** להניל את לקוחות, למכוור ציוד-קצה או שירותים (כגון תיקונים, או התקנות), לניל את המלאי והזמןות הצמוד, והוא כפוף לשירות הנהלה.



יעוד המערכת

המערכת תשמש עסקים קטנים לניהול פעילותם. היא תחליף התנהלות ידנית שכוללת לרוב מסמכים, כרטוסות לקוח ורשימת מלאי. המערכת מאפשרת ריכוזיות המידע, גישה נוחה ומהירה אליו, מידור הרשותות בהתאם לרמת הרשותות, גיבוי וייצוא של הנתונים, בדגש על פשטות הפעול, יציבות המערכת וזמן נתיחה הגבוהה.

דרישות המערכת

- המערכת תספק יכולות לניהול לקוחות העסק:

- הוספה/הסרה/עדכון של לקוחות.
- חיפוש אחר לקוחות.
- צפיה בפרטיו לקוחות.
- צפיה בהיסטורית לקוחות.

- המערכת תספק יכולות לניהול תעריפים ותמחור:

- תמחור המוצרים/שירותים.
- עדכון תעריפים גורף.
- קביעת ערך המע"מ.

- המערכת תספק יכולות לניהול מלאי והזמנות ציוד:

- הוספה/הסרת מוצר מלאי.
- ביצוע הזמנת ציוד.
- עדכון סטאטוס הזמנה.
- עדכון כמות המלאי.

- המערכת תספק יכולות לניהול מכירות:

- ביצוע רכישה.
- פתיחת קריית שירות.
- חיבור לקוחות בגין מוצר/שירות.

- הפקת חשבונית ללקוח.
- המערכת תספק יכולות להזנת מבצעים:
 - עדכון מבצעים:
 - הזרמת מבצע עבור מוצר בודד.
 - הזרמת מבצע עבור קטגורית מוצרים.
 - הזרמת מבצע הנחה כוללת.
 - הזרמת מבצע מוצר שני חינם (1+1).
 - צפיה במבצעים.
- המערכת תספק יכולות לחלוקת העבודה ומעקב אחר ביצועה:
 - הקצאת קריית שירות לעובד צוות.
 - מעקב אחר קריאות שירות.
 - עדכון סטאטוס קריית שירות.
 - ניהול היסטורית קריית שירות לצרכי תיעוד.
- המערכת תספק יכולות להפקת דוחות:
 - הפקת דוח הכנסות/הוצאות.
 - הפקת דוחות לקוחות.
 - הפקת דוח הזמנות ציוד.
 - הפקת דוח מכירות.
 - הפקת דוח קריאות שירות.
 - הפקת דוחות פעילות.
- גישה מבוקרת אל המערכת תבצע באמצעות שם משתמש וסיסמה, ותאפשר לכל משתמש לקבל את רמת המידור המתאימה, כך שייחשף אך ורק אל תוכנים הרלוונטיים עבורו.



- אפשרות להוספה/הסרת משתמשים במערכת וקייעת רמת ההרשאות שלהם.
- אפשרות לקביעת הגדרות כלליות עבור המערכת.
- המערכת תספק אפשרות לטעינת קובץ מודול כדי לטעון למערכת רשימות של סוגי שירותים, סוגי מוצרים וחברות.
- המערכת תספק ממשק משתמש נוח ופשוט.
- המערכת תעמוד בדרישות הביצועים הבאות:
 - **זמן ויציבות –** עדモת הקצה יהיה זמינות לפחות 92% משעות העבודה, בעוד שהשתתת המאחסן את בסיס הנתונים יהיה זמין לפחות 96% משעות העבודה.
 - **גיבויים:**
 - גיבוי יומי עבור השינויים שבוצעו באותו יום (incremental).
 - גיבוי שבועי מלא (full).
 - **זמן תגובה –** פעולות במערכת (למעט הפקת דוחות) יארכו לא יותר מ-2 שניות לכל פעולה. פעולות להפקת דוחות יארכו לא יותר מ-10 שניות.
 - **קיבולות –** על המערכת לתרום בקבולות הבאות:
 - עד 10,000 לקוחות.
 - עד 1,000 סוגי מוצרים שונים במלאי.
 - עד 1,000 סוגי שירותים שונים.
 - עד 50 משתמשים במערכת.
 - עד עלות של 1,000,000 ₪ למוצר / שירות בודד.
 - המערכת תהיה גמישה לשינויים עתידיים ורחבה פונקציונליות.
 - המערכת לא תספק את היכולות הבאות בשלב הנוכחי:
 - דוחות נוכחות והתחשבנויות שכר לעובדים.

- שמירת מידע היסטורי אודזות מבצעים.
- שמירת מידע היסטורי לגבי שיעור המע"מ והתמחור.
- המערכת לא תיזום התקשרות מול ספקים להזמנת ציוד.
- תמיכה בהחזרת מוצרים או ביטול שירותים מצד לקוחות.
- ביצוע חיוב בפועל של כרטיסי אשראי.
- אפשרות לעבודה במקביל על ידי יותר משתמש אחד.



מרכיבי המערכת

בסיס נתונים

בסיס הנתונים ממוקם בשרת הייעודי, אשר יהיה נגיש מכל עמדות הקצה בעסק. בסיס הנתונים יחבר לשרת גיבוי הייעודי, שיבצע גיבוי יומי לשינויים שהתווספו, וגיבוי שבועי מלא.

עמדות קצה

המערכת תותקן על גבי כל אחת מעמדות הקצה, ותתקשר מול בסיס הנתונים הממוקם בשרת הייעודי. לכל אחד משתמשי המערכת תריה עמדת קצה משלו.

משתמשי המערכת

מנהל המערכת (Administrator)

מנהל המערכת הוא משתמש ייחודי המוגדר מראש, אשר לא ניתן למחוק אותו. הוא היחיד שמסוגל לבצע את הפעולות הבאות:

- הוספת/הסרת משתמשים במערכת וקבעת רמת הרשות.
- קביעת הגדרות כלליות במערכת.
- טעינת קובץ מודול.

מנהל העסק

מנהל העסק נמצא בראש היררכיית העסק ומשתיך לדרג הנהלה. הוא מפקח באופן ישיר על דרג התיאום והבקרה, ואחראי על שימור הליקות. בנוסף הוא קובע את אסטרטגיות התמחור והמבצעים. למנהל העסק יכולת לביצוע הפעולות הבאות:

- צפיה בפרטיו הליקוחות ובהיסטוריה הליקוחות.
- הוספת/הסרת מוצר במלאי, קביעת מחיר וכמות קיימת.

- קביעת ערך המע"מ.
- עדכון תעריפים גורף.
- שינוי תמחור המוצרים/שירותים.
- קביעת מבצעים.
- הפקת דוחות ניהול.

ראש צוות

ראש צוות משתיר לדרג התיאום והבקרה. הוא כפוף לדרג הנהלה, ואחראי על התחנולות השוטפת של דרג התפעול. ראש הצוות אחראי על חלוקת העבודה לדרג התפעול, כמו כן, תפקידו כולל פיקוח על ביצוע העבודה.

לראש הצוות יכולת לביצוע הפעולות הבאות:

- הקצאת קריאות שירות לעובד צוות.
- מעקב אחר קריאות שירות.
- הפקת דוחות תפעוליים.

מנהל מכירות

מנהל מכירות משתיר לדרג התיאום והבקרה. הוא כפוף לדרג הנהלה. הוא אחראי על אינטראקציה מול הלוקחות, ניהול מלאי, וכן, תיאום קריאות שירות מול ראשי הצוותים. כמו כן, הוא גם אחראי על הזמינות הצדוק וקליטתו. למנהל המכירות יכולת לביצוע הפעולות הבאות:

- ביצוע רכישות של לקוחות.
- הוספה/הסרה/עדכון/חיפוש של לקוחות.
- פתיחת קריאות שירות.
- הזמנת ציוד וקליטתו.
- עדכון סטטוס הזמנת ציוד.



- חיוב לקוחות בגין מוצר/שירות.
- הפיקת חשבון לקוחות ללקוחות.
- הפיקת דוחות מכירות, הזמןנות ולקוחות.
- מתן הצעת מחיר ללקוחות.

עובד צוות

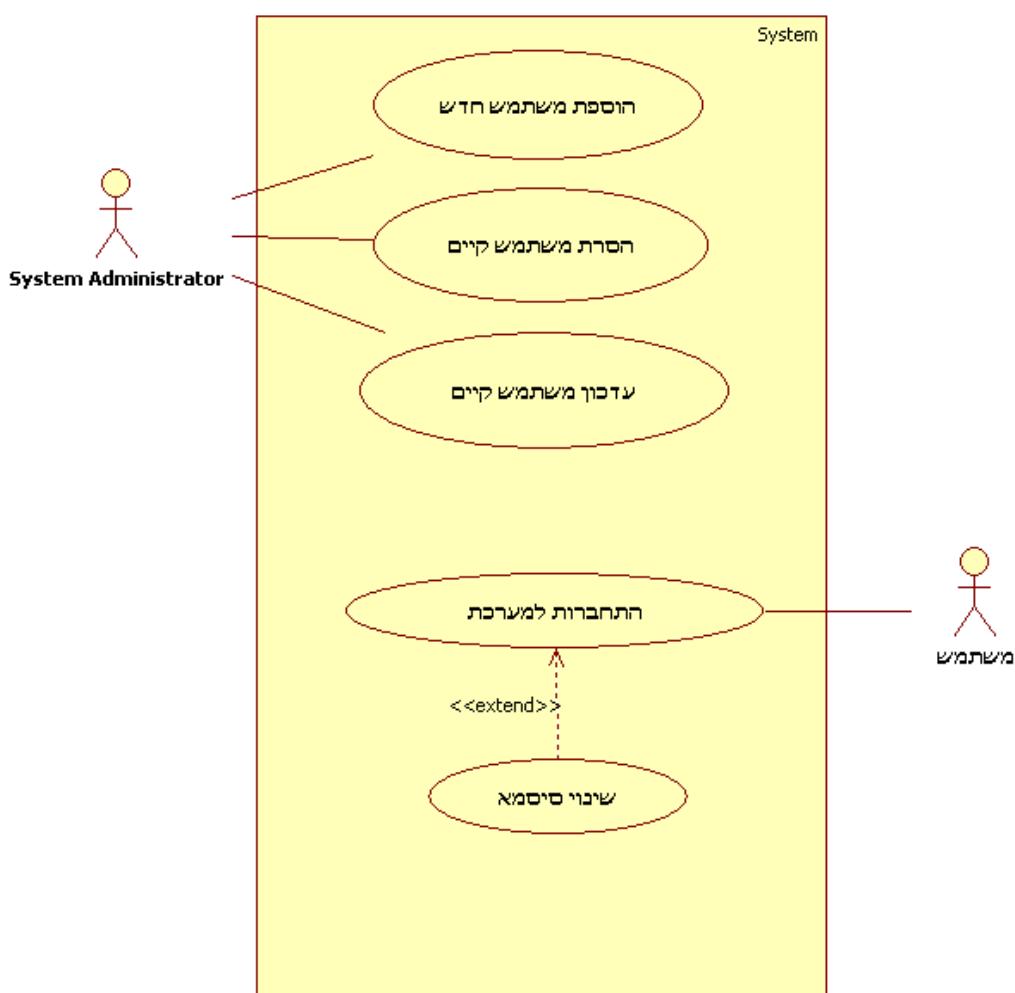
עובד צוות משתיר לדרג התפעול, והוא כפוף לדרג התקioms והבקרה. הוא אחראי לביצוע פיזי של המטלות שהוטלו עליו, וכן עדכון סטאטוס הביצוע במערכת, לרבות עדכון שעות העבודה והחומרים שבהם היה צריך.

עובד צוות יכול לביצוע הפעולות הבאות:

- עדכון סטאטוס של קריית השירות.
- מעקב אחר עבודות המשויות אליו.

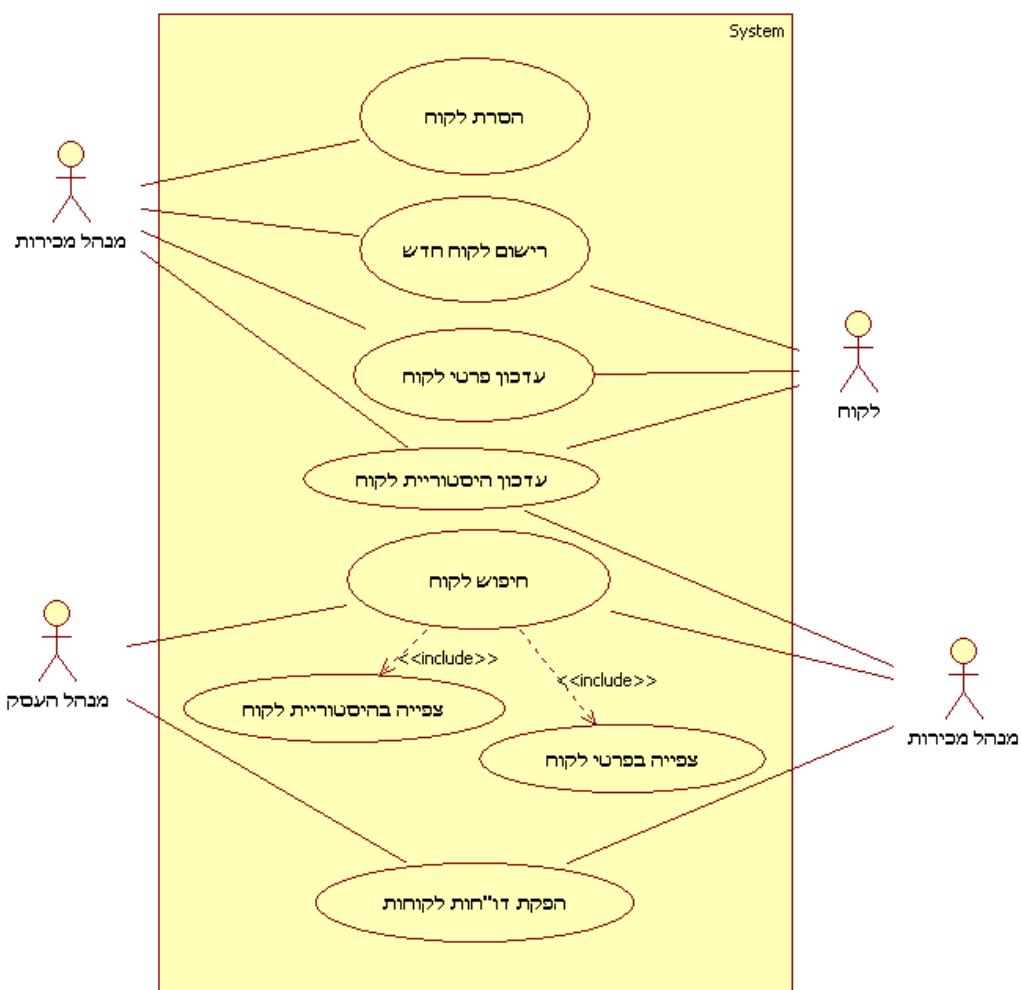
דיאגרמות Use-Case

ניהול משתמשים במערכת



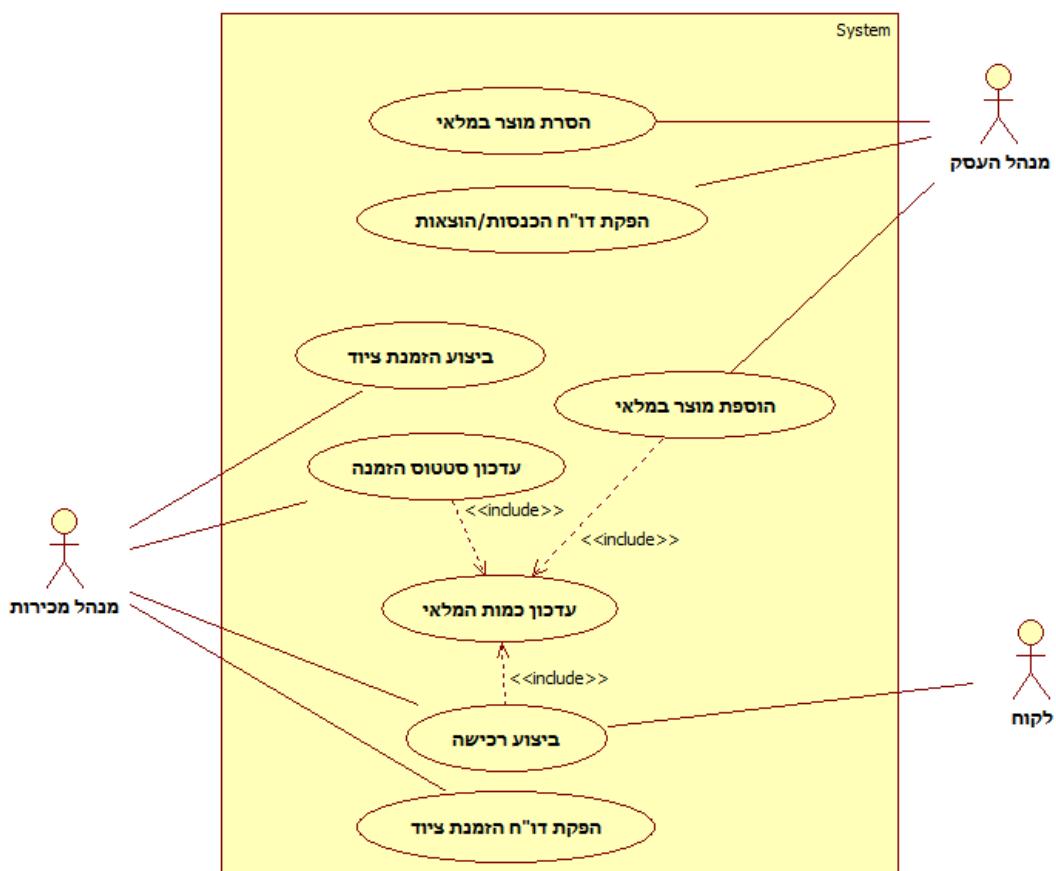
1. מנהל המערכת מוסיף משתמש חדש עבור העובד בעסק.
2. מנהל המערכת מסיר משתמש קיים העובד בעסק.
3. מנהל המערכת מעדכן את פרטיו של העובד קיים בעסק.
4. מנהל המערכת קובע הרשות עבור משתמשים במערכת.
5. משתמש מתחבר למערכת.
6. משתמש משנה סיסמא.

ניהול ללקוחות

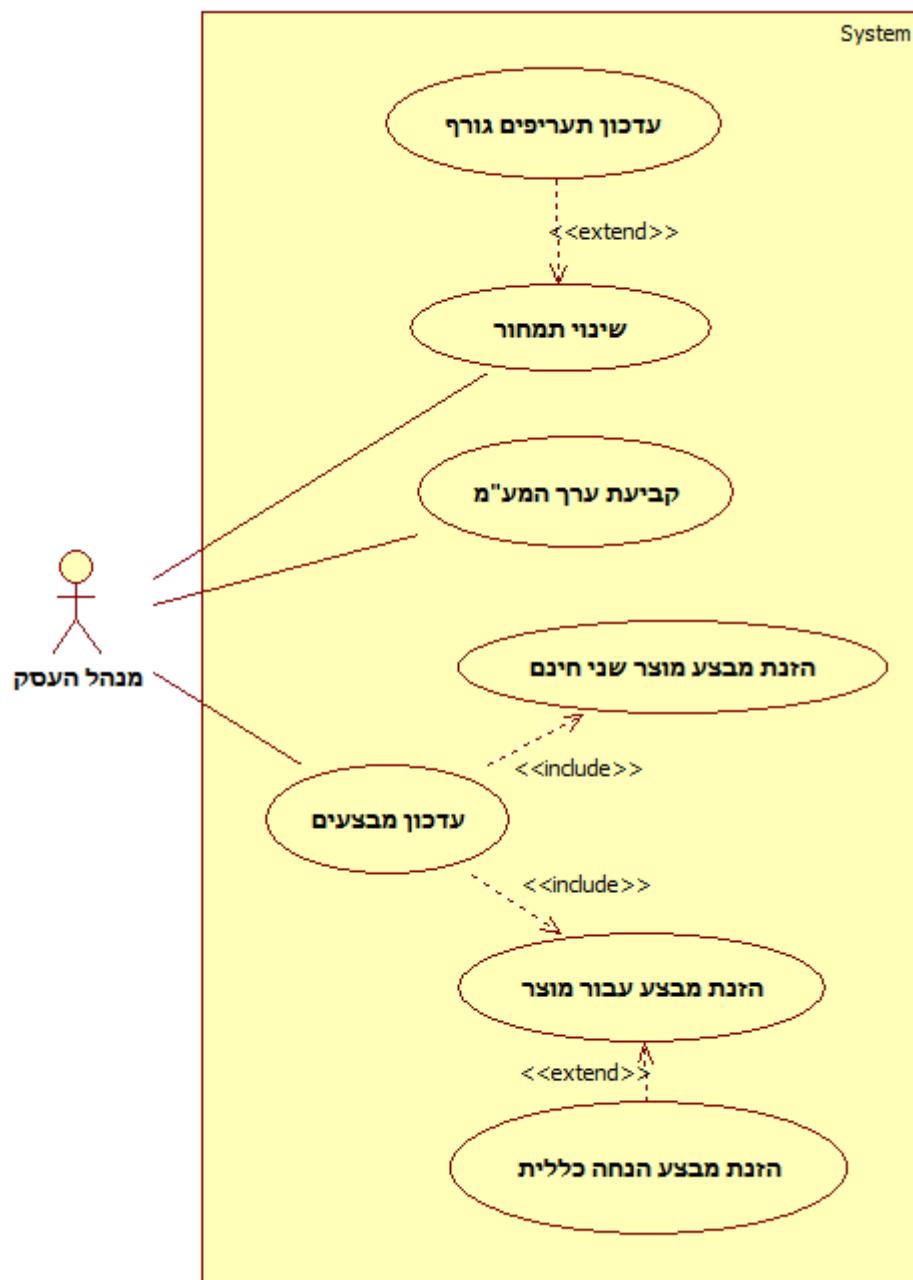


1. מנהל העסק מhapus אחר פרטי לקוח/היסטוריה לקוח במערכת.
2. מנהל העסק ומנהל המכירות מפיקים דוחות לקוחות.
3. לקוח יוצר קשר עם מנהל מכירות ונרשם במערכת כללקוח חדש.
4. מנהל מכירות מסיר לקוח קיים מהמערכת.
5. מנהל מכירות מעדכן את פרטיו של לקוח קיים / ההיסטוריה לקוח.
6. מנהל מכירות מhapus אחר פרטי לקוח/היסטוריה לקוח במערכת.
7. מנהל מכירות מעדכן ההיסטוריה לקוח במערכת.

ניהול מלאי והזמנת ציוד



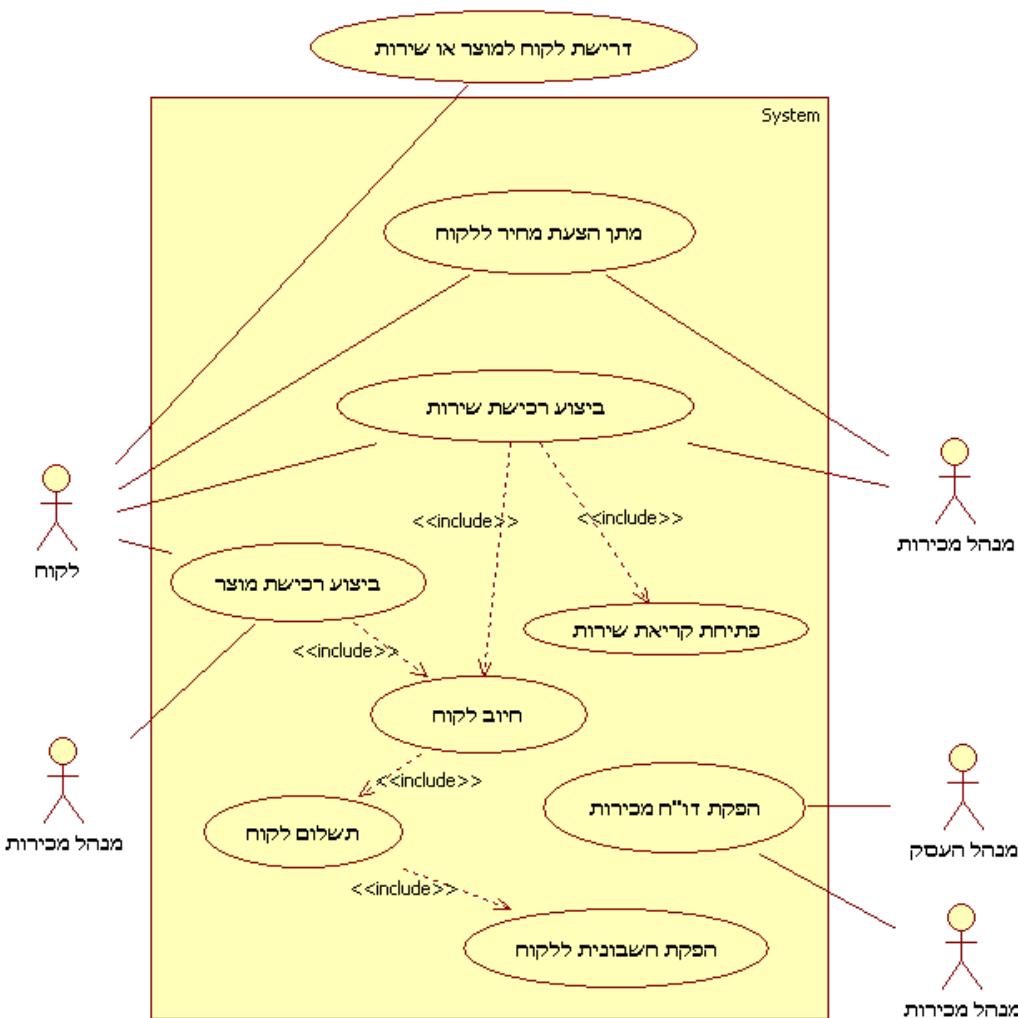
1. מנהל העסק מוסיף מוצר חדש אל המלאי.
2. מנהל העסק מסיר מוצר קיים מהמלאי.
3. מנהל מכירות מבצע הזמנת ציוד.
4. מנהל מכירות מעדכן סטטוס הזמן וכתוצאה מכך המלאי מתעדכן.
5. לקוח מבצע רכישה וכתוצאה מכך המלאי מתעדכן.
6. מנהל מכירות מפיק דוח הזמן ציוד.
7. מנהל העסק מפיק דוח הכנסות/הוצאות.

ניהול תעריפים ותמחור

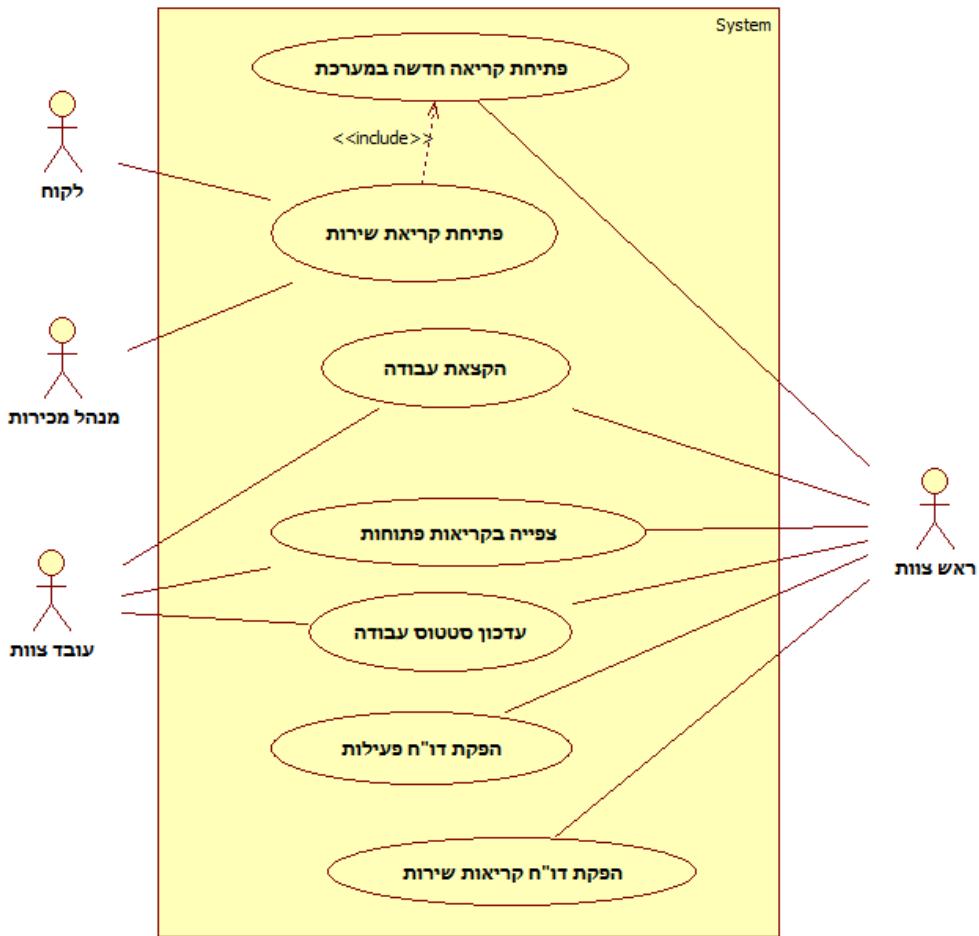
1. מנהל העסק מתמחר מוצר / שירות.
2. מנהל העסק מבצע עדכון תעריפים גורף.
3. מנהל העסק קובע את ערך המע"מ.

4. מנהל העסק מזין מבצע מוצר שני חינם (1+1) או מזין מבצע עבור מוצר, כאשר יש אפשרות לבצע הэнט מה被执行 ל מוצר בודד / קטgorיה / המוצרים כולם.



רכישת שירות או מוצר

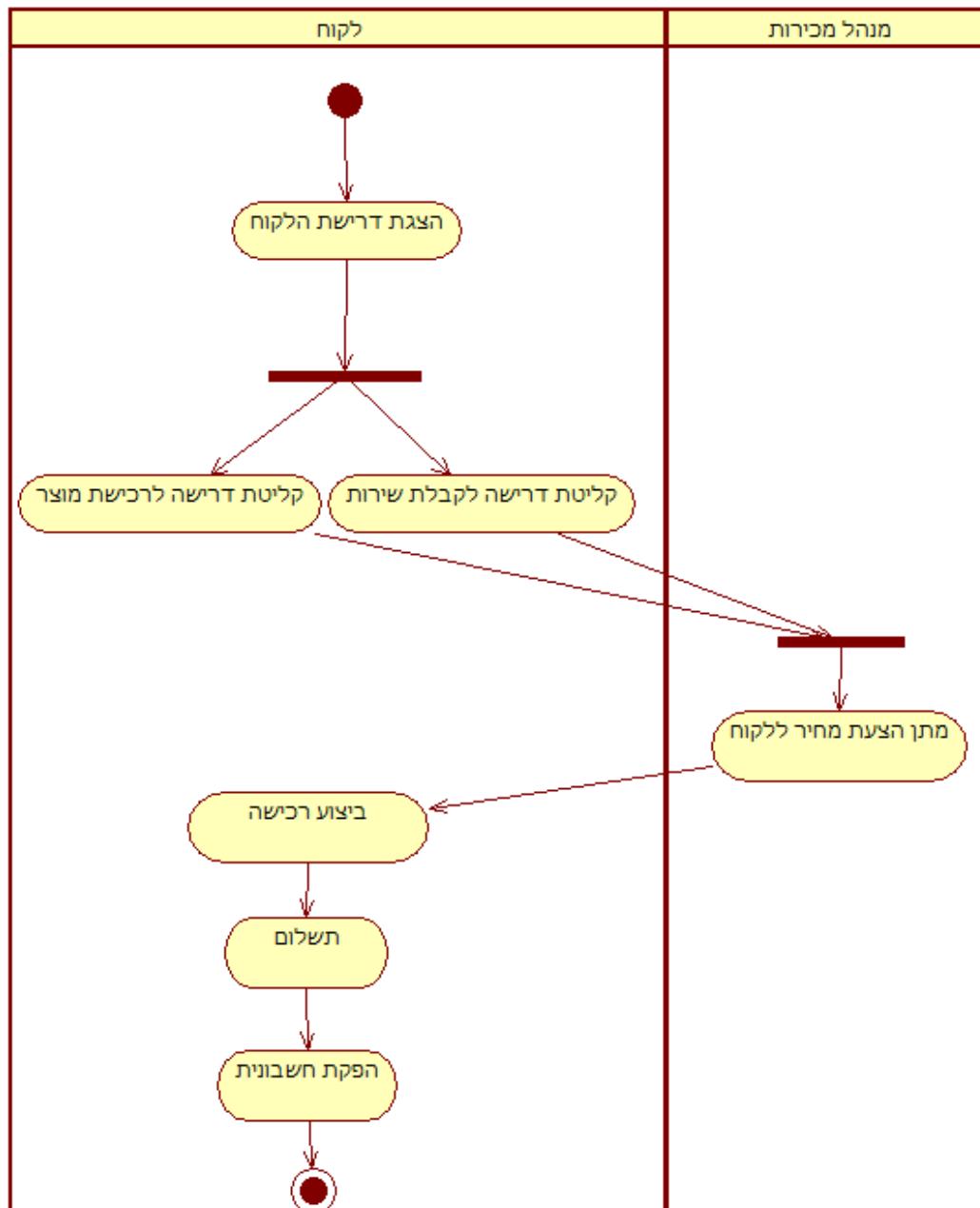
1. הלוקח מעוניין לרכוש מוצר או לקבל שירות.
2. מנהל המכירות נותן העטת מחיר ללקוח.
3. לקוח רוכש מוצר / שירות אצל מנהל המכירות.
4. מנהל המכירות מחייב לקוח, כתוצאה לכך הלקוח משלם, ובהתאם לכך מנהל המכירות מפיק חשבונית עבור התקבולים מהלקוח.
5. לקוח רוכש שירות וכתוצאה לכך נפתחת קריית שירות במערכת.
6. מנהל העסק מפיק דוח מכירות.

חלוקת עבודה

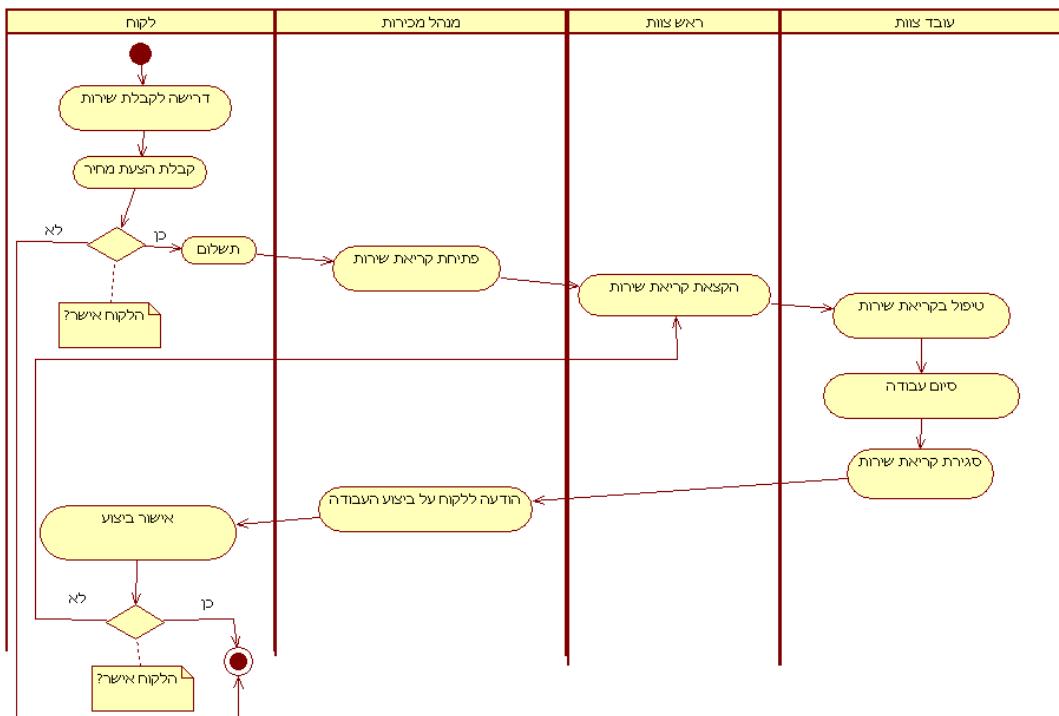
1. ל Koh פונה למנהל המכירות בבקשת לפתוח קרייאת שירות, כתוצאה מכך קרייה חדשה נפתחת במערכת.
2. ראש צוות מקבל את הקריאה שנפתחה ומבצע הקצאת עבודה לאחד מעובדי הצוות.
3. עובד הצוות מבצע את העבודה וمعدכן סטטוס ביצוע העבודה.
4. עובד הצוות יכול לצפות בקרייאות הפתוחות הרלוונטיות אליו.
5. ראש הצוות יכול לצפות בכל הקרייאות הפתוחות.
6. ראש הצוות יכול לעדכן בעצמו סטטוס עבודה.
7. ראש הצוות יכול להפיק דוח פעילות / קרייאות שירות.

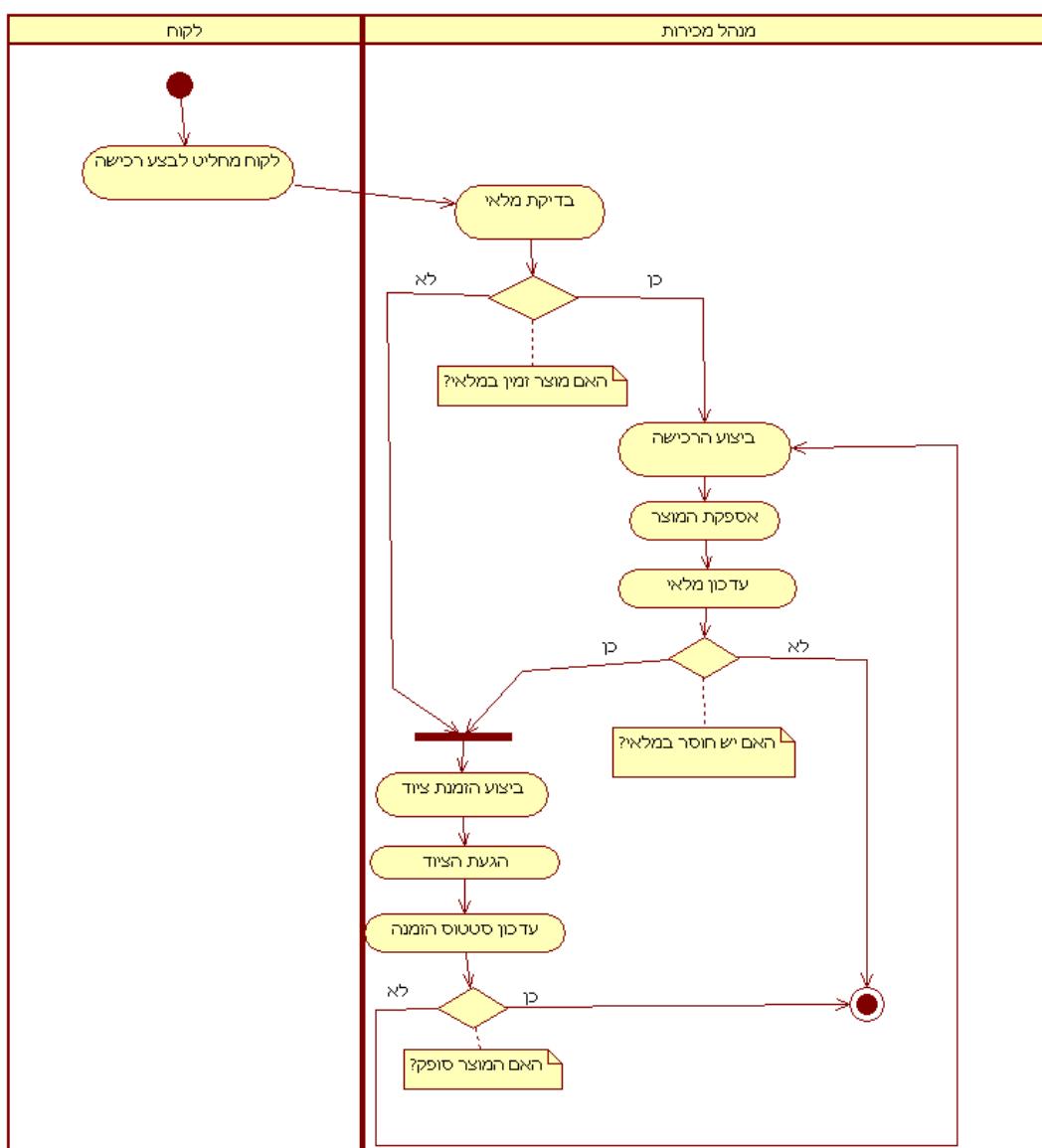
דייגראמות Activity

רכישת שירות / מוצר

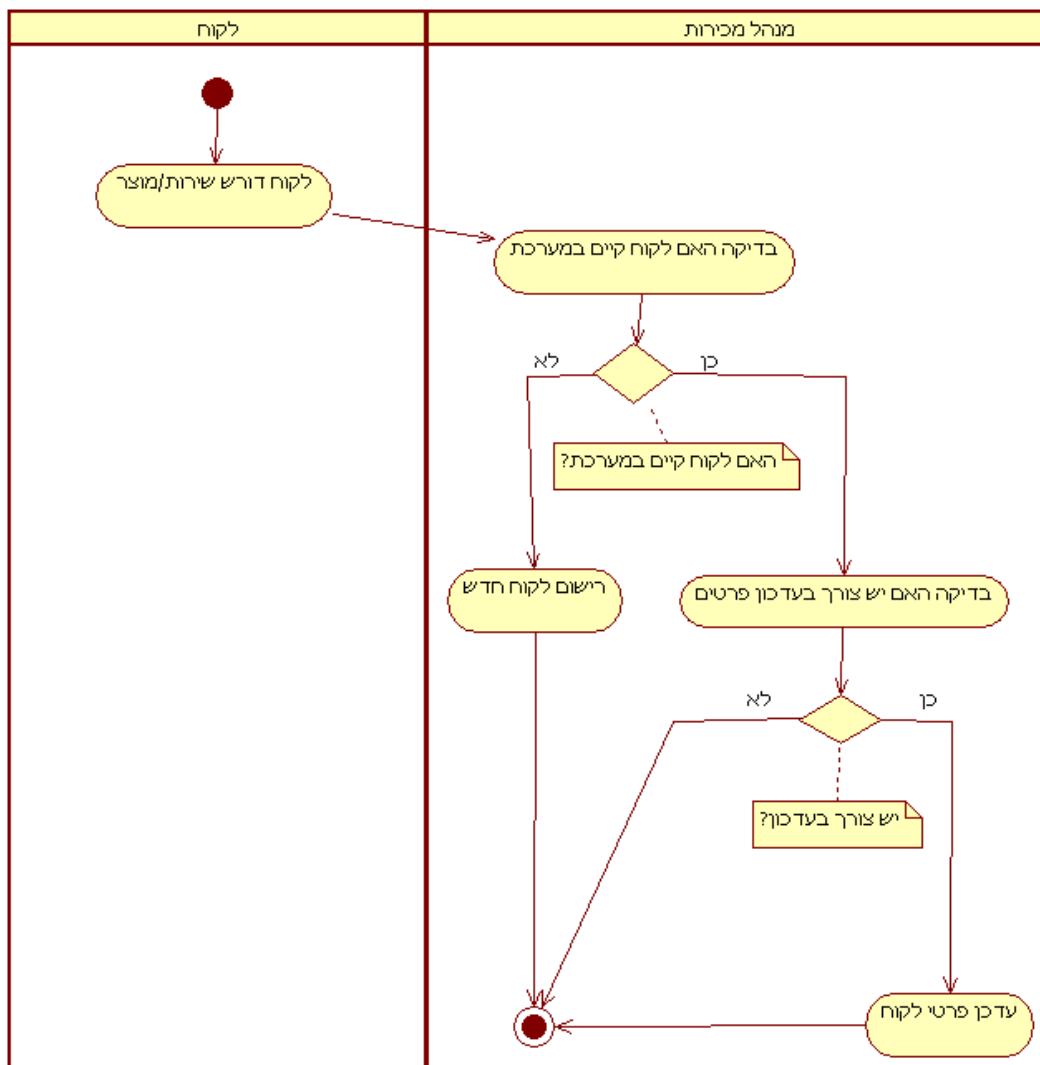


חלוקת עבודה



ניהול מלאי וחווננות

ניהול לקוחות



File #0003243 belongs to Roei Daniel- do not distribute

SmartBiz

מדריך למשתמש

גרסה 2.0

שמות הסטודנטים:

רותם אלישדה

עומר בר-און



File #0003243 belongs to Roei Daniel- do not distribute

תוכן עניינים

169.....	תיאור המערכת
170.....	dagshim ונהנחות לגבי פועלות המערכת
172.....	סרגלי כלים
174.....	התחברות למערכת
175.....	שינוי סיסמה
176.....	מסמך ראשי השיר למשתמש Admin
177.....	ניהול משתמשים (User Management)
179.....	הוספת משתמש חדש או עדכון משתמש קיימ
181.....	הגדרות כלליות (Global Settings)
183.....	מסמך ראשי (מנהל העסק)
184.....	מסמך ראשי (מנהל מכירות)
185.....	מסמך ראשי (ראש צוות)
186.....	מסמך ראשי (עובד צוות)
187.....	ניהול לקוחות (מנהל העסק)
188.....	ניהול לקוחות (מנהל המכירות)
189.....	טופס פרטי לקוח (מנהל העסק)
190.....	טופס פרטי לקוח (מנהל המכירות)
192.....	ההיסטוריה של לקוח (מנהל העסק)
193.....	ההיסטוריה של לקוח (מנהל המכירות)
194.....	ניהול מכירות
196.....	קבלת התקבולים
197.....	הפקת חשבוניות
198.....	תמחרור מוצרים
199.....	ניהול מלאי (מנהל העסק)
200.....	ניהול מלאי (מנהל מכירות)
202.....	הוספת פריט חדש למלאי או עדכון פריט קיימ

SmartBiz

204.....	ניהול הזמינות ציוד
206.....	הוספת הזמנה חדשה / עדכן או צפיה בהזמנה קיימת
208.....	ניהול מוצרים
209.....	חלוקת עבודה (ראש הוצאות)
210.....	חלוקת עבודה (עובד צוות)
212.....	קריאה שירות (ראש הוצאות)
213.....	קריאה שירות (עובד צוות)
214.....	עריכת אירוע היסטוריה
215.....	מסמך דוחות (מנהל העסוק)
216.....	מסמך דוחות (מנהל המכירות)
217.....	מסמך דוחות (ראש הוצאות)
218.....	דו"ח לקוחות
219.....	דו"ח מכירות
220.....	דו"ח הכנסות-הוצאות
221.....	דו"ח הזמינות
222.....	דו"ח קריאות שירות
224.....	דו"ח פעילות

תיאור המערכת

מערכת זו מיועדת לניהול עסקים קטנים. מדובר במערכת גנריית המתאימה לרוב סוגים של עסקים קטנים מכיוון שהוא מספקת פונקציונליות כללית הדרישה במרבית העסקים שמדובר בהם מוצריהם או שירותים. המערכת נותנת מענה לעסקים קטנים שכנים לא משתמשם לבנות עבורים מערכת מידע ייעודית שתאפשר להם ניהול יעיל של העסק.

המערכת מאפשרת טעינת מודולים הייחודיים לכל סוג עסק, המתאים אותה לצורכיهم. מלבד פיתוח המערכת, אנו נדגים את תפעולה השוטף באמצעות דוגמא של עסק קטן: "מעבדת מחשבים פרטית".

מסמכי האפין, הניתוח והעיצוב יתארו את המערכת כפי שהיא פועלת בצורה גנריית.

1. המערכת תשמש לניהול עסק פרטי

2. משתמשי המערכת

- **מנהל העסק** [לדוגמה: מנהל המעבדה]
- **ראש צוות** [לדוגמה: ראש צוות טכנאים]
- **עובד צוות** [לדוגמה: טכניי המעבדה]
- **מנהל מכירות**

3. יכולות המערכת

- **ניהול לקוחות העסק** (לשימוש מנהל העסק ומנהל המכירות)
- **ניהול תעריפים ותמחרור** (לשימוש מנהל העסק)
- **ניהול מוצרים** (לשימוש מנהל העסק)
- **ניהול מלאי** (لשימוש מנהל העסק ומנהל המכירות)
- **ניהול זמינות ציוד** (لשימוש מנהל המכירות)
- **ניהול מכירות וחיבור לקוחות** (לשימוש מנהל המכירות)
- **חלוקת עבודה ומעקב אחריה** (לשימוש ראש הצוות ועובד הצוות)
- **הפיקת דוחות**

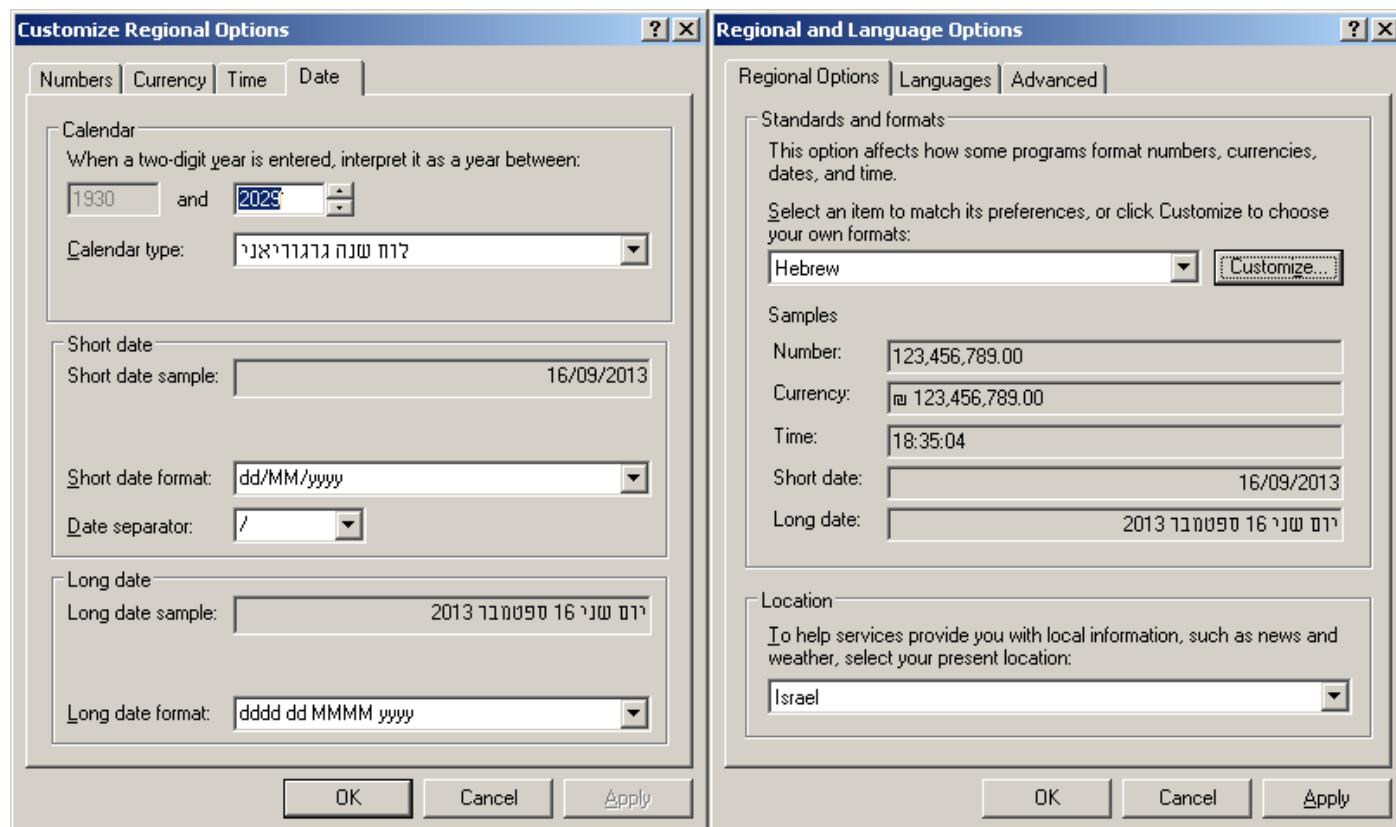


יתר דרישות המערכת ו貌וּנָה המלא יפורטו במסמך ניתוח ו貌וּן המערכת.

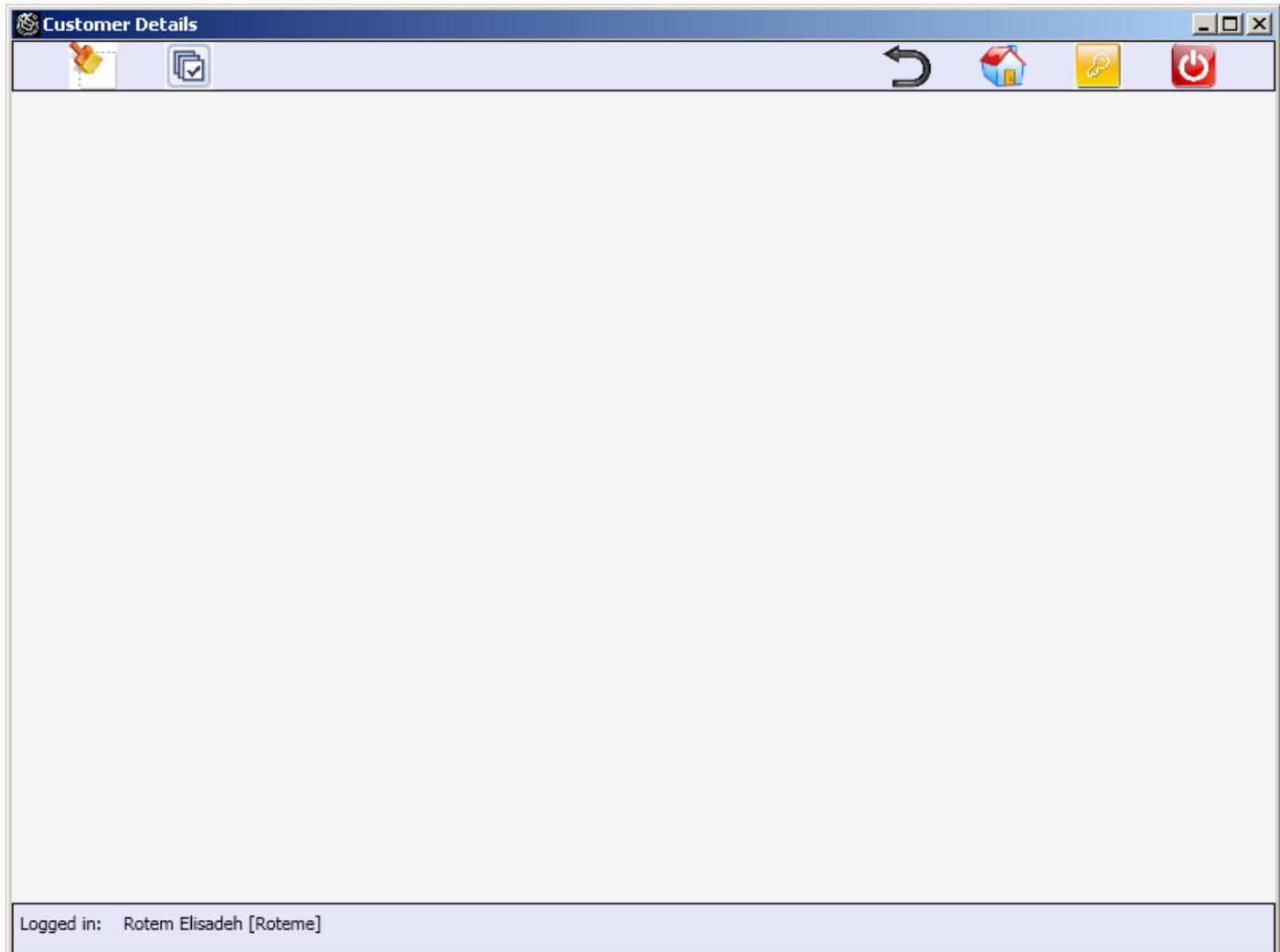
דגשים והנחות לגבי פועלות המערכת

- המערכת תומכת במסמך סוג הרשות, כאשר יתכן ומשתמש מסוים ישוויכו מספר הרשות.
- במקרה הנ"ל, המערכת תציג את הפונקציונליות שמתאפשרת מכל הרשותות ייחדי (לדוגמא: במסך הראשי יתכן שימוש אשיר שוייכו אליו כל הרשותות תהיה יכולה לכל הceptors הזרים לככל המשתמשים במסך זה).
- כמו כן, חשוב להעיר, כי במידה ובמספר מסוים קיימת לבעל הרשותה אחת פונקציונליות חלקייה, ואילו לבעל הרשותה אחרת פונקציונליות מורחבת, אזי אם משתמש נתון יהיה בעל שתי הרשותות שתוארו לעיל, הוא יוכל להשיג לפונקציונליות המורחבת.
- בכל הדוחות הקיימים במערכת ובכל טבלאות הנתונים (Data Grid) של הטפסים השונים, ברירת המחדל היא להציג את כל הנתונים הקיימים במערכת. כמו כן, שדה חיפוש ריק מצין שאין הגבלת חיפוש עבור שדה זה.
- דוחות המערכת יאפשרו להציג הכנסות, הוצאות, ומומוּנִות שנמכרו, יחד עם זאת, היא לא תציג את הרוחות שהעסק הרווח עבור אותן מכירות. הסיבה לכך, היא שנוסח הנהלת החשבונות הוא מחוץ לתחום המערכת.
- המערכת אינה תומכת בהחזרת מוצרים מצד הלוקוח. בנוסף המערכת אינה תומכת בזיכוי הלוקוח עבור מוצרים או שירותים כלשהם. ההנחה היא שככל המוצרים והשירותים שניתנו ללוקוח הינם תקנים ואין החזרות מצד הלוקוח.
- במערכת זו, לכל מוצר ושירות קיים מחיר בטווח 0 עד 1,000,000. מחיר זה לא כולל את המעוּם ואת ההנחות התקופתיות. חישוב ההנחה מתבצע לאחר הוספת המעוּם.
- כל שדות הטקסט שאינםאפשרים ריבוי שורות מוגבלים ל-50 תווים.
- המערכת מאפשרת ללוקוח לשלם בכרטיס אשראי, אך החיבור בפועל אמר להתבצע באופן ידני במקביל לעבודה עם המערכת, ולא באמצעותה. כמו כן, איןנה שומרת מידע כלשהו על כרטיסי אשראי.
- המערכת מאפשרת ייצור של עד 1,000 סוגי מוצרים שונים במלאי ועוד 1,000 סוגי שירותים שונים. המספר הסידורי של סוג המוצר נע בתווך 1,000 – 1,999. המספר הסידורי של סוג השירות נע בתווך 2,000 – 2,999.
- סימנת האדמיניסטרציה במערכת היא SmartBiz (case sensitive), והיא איננה ניתנת לשינוי.

- מערכת זו איננה תומכת בהתקשרות מול ספקים חיצוניים.
- כל המודולים במערכת שעובדים עם תאריכים מצפים לערכים בפורמט: yyyy/MM/dd. لكن, לפני שמתחלים לעבוד על המערכת יש להיכנס ללוח הבקעה ← Regional and Language Options ולוודא שסוג לוח השנה הינו: "לוח שנה גregorיאני" ובנוסף ש- date format הינו: dd/MM/yyyy. ניתן להיעזר בצילומי מסך הבאים:



הגדרות לתאריכים במערכת הפעלה "חלונות"

סרגלי כלים

- לחיצה על כפתור ה-Exit מאפשרת יציאה מסודרת מהמערכת.



- לחיצה על כפתור ה-Logoff מנטקמת את המשתמש הנוכחי מהמערכת.



- לחיצה על כפתור ה-Home מוחזירה את המשתמש אל המסר הראשי.



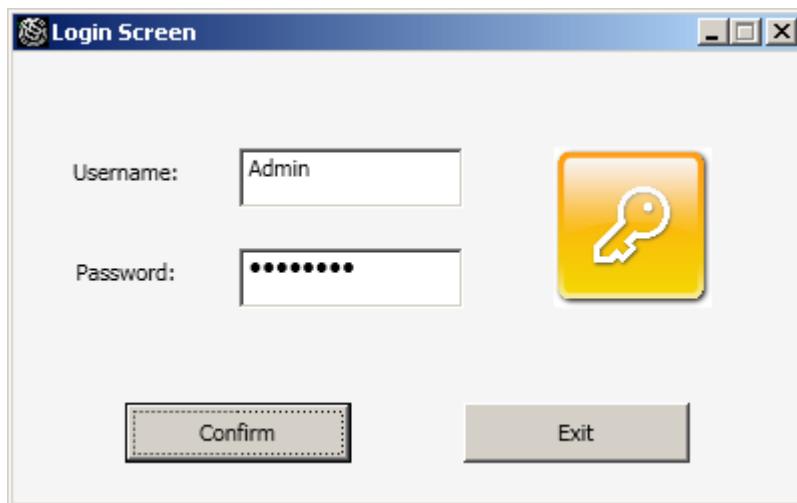
- לחיצה על כפתור ה-Turn-U מוחזירה את המשתמש אל המסר הקודם.



- לחיצה על כפתור ה-Select All מסמנת את כל השורות ב-Data grid.



- לחיצה על כפתור ה-Clear מנקה את כל שדות הטקסט הפעילים.
- בסרגל הכלים התחתון יופיע תמיד שם המלא (שם פרטי + שם משפחה) של המשתמש המחבר כעת ומיד אחריו בסוגרים מרובעים יופיע שם המשתמש עצמו (Username).

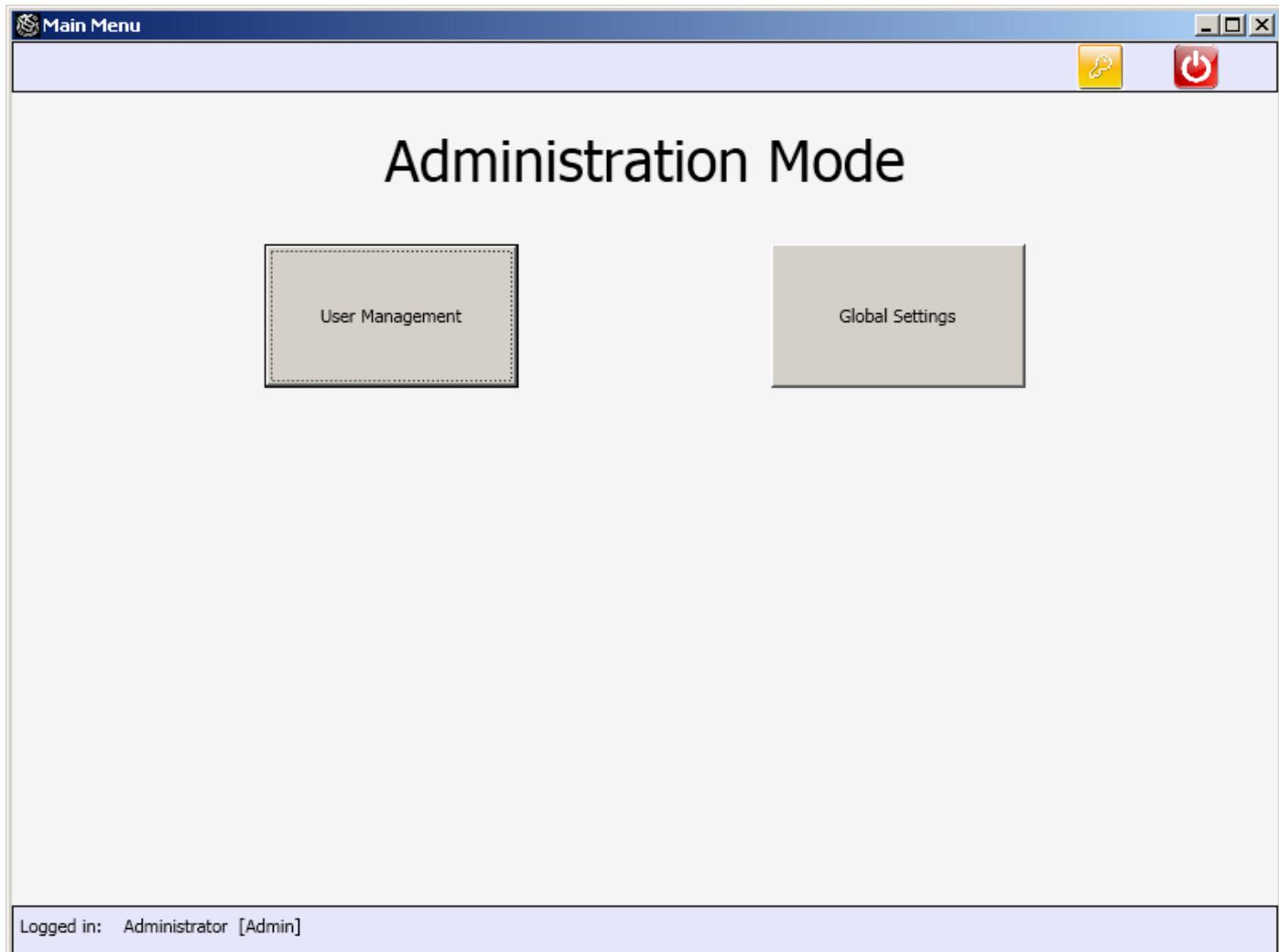
התחברות למערכת

- מסך זה מאפשר התחברות למערכת באמצעות שם משתמש וסיסמה.
- לאחר תחילת ההזדהות, המערכת תציג למשתמש את המסך הראשי עם הפונקציונליות המתאימה בהתאם להרשאות המשתמש אשר מוגדרות במערכת.
- למערכת קיים משתמש ברירת מחדל בשם `Admin`, אשר באמצעותו ניתן להגדיר את המשתמשים השונים במערכת ואת הרשאותיהם, כמו כן, להגדיר הגדרות כלליות.
- הסיסמה של המשתמש `Admin` תינוק ללקוח בעת מסירת המערכת.

שינוי סיסמא

- מסר זה מופיע עבור כל משתמש בכניסה הבאה שלו אל המערכת לאחר שמנהל המערכת בחר באפשרות: *so on user must change password at next login.*
- מסר זה מבקש מהמשתמש להזין את הסיסמה הישנה שלו, ולהזין סיסמא חדשה במקומה.
- מסר זה מבקש מהמשתמש לרשום את הסיסמא החדשה פעמיים מטעמי זהירות.
- יש לבחור את הסיסמא החדשה תחת האילוצים הבאים: אורך, מורכבות ואיסור על מתן סיסמא זהה לשנה. אילוצי האורך והמורכבות יקבעו בהתאם לగרסא ולפי אילוצי העסק.
- לחיצה על כפתור ה-*Confirm* יאשר את שינוי הסיסמא.
- לחיצה על כפתור ה-*Exit* יסגור את החלון.

מסך ראשי השויך למשתמש Admin



- לאחר ההתחברות עם המשתמש Admin למערכת, מקבלים גישה למצב מיוחד בשם .Administration Mode.
- מצב זה מאפשר להיכנס לאחד משני תת-תפריטים:
 - ניהול משתמשים
 - הגדרות כלליות.

ניהול משתמשים (User Management)

The screenshot shows the 'User Management' window. At the top, there's a toolbar with icons for adding a user (blue square with a checkmark), deleting a user (red square with a minus sign), and saving changes (yellow square with a checkmark). Below the toolbar is a table displaying user information:

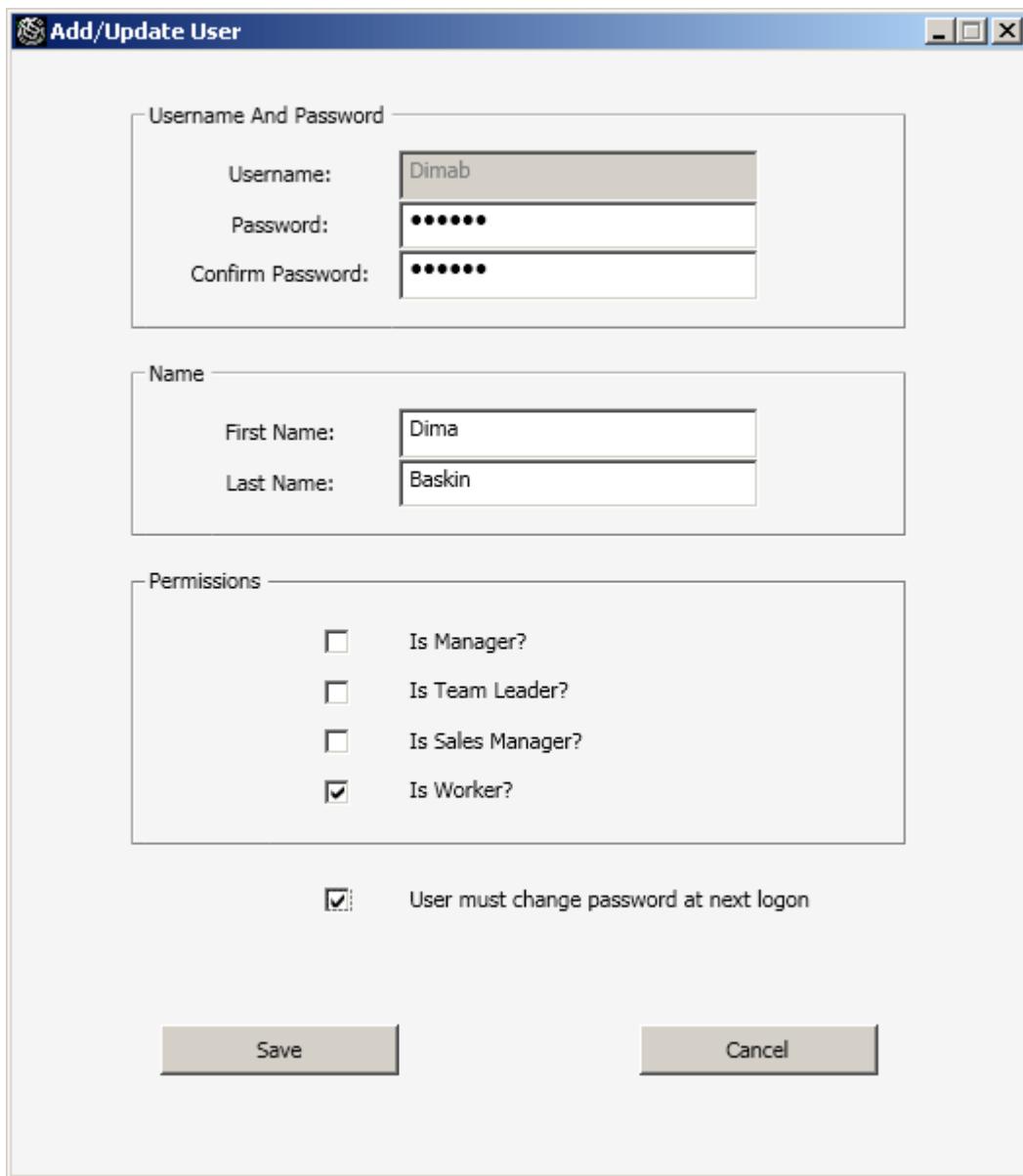
Username	First Name	Last Name	Is Manager?	Is Team Leader?	Is Sales Manager?	Is Worker?
Dannyk	Danny	Kaplan	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Davidc	David	Cohen	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Dimab	Dima	Baskin	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Roteme	Rotem	Elisadeh	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>

At the bottom of the window, there are two buttons: 'Add User' and 'Remove User'. A status bar at the very bottom indicates 'Logged in: Administrator [Admin]'.

- המסר מציג את רשימת המשתמשים המוגדרים כרגע במערכת (למעט המשתמש Admin שהוא בעל רשות super-user ומשמש עבור כניסה למצב Administration).
- לכל משתמש מוצגות הרשאות שהוגדרו עבורי, כאשר ישן ארבעה סוגי הרשאות:
 - הרשות ניהול העסק
 - הרשות ראש צוות
 - הרשות ניהול מכירות
 - הרשות עובד

- כל משתמש יכול להיות בעל מספר הרשות במקביל, דבר אשר אפשר לו פונקציונליות מוגדלת זואת בהתאם להרשות שקייבל.
- ניתן להוסיף משתמשים באמצעות כפתור ה-User Add.
- במקרה להסרה משתמש/ים יש לסמן אותם וללחוץ על כפתור ה-User Remove.

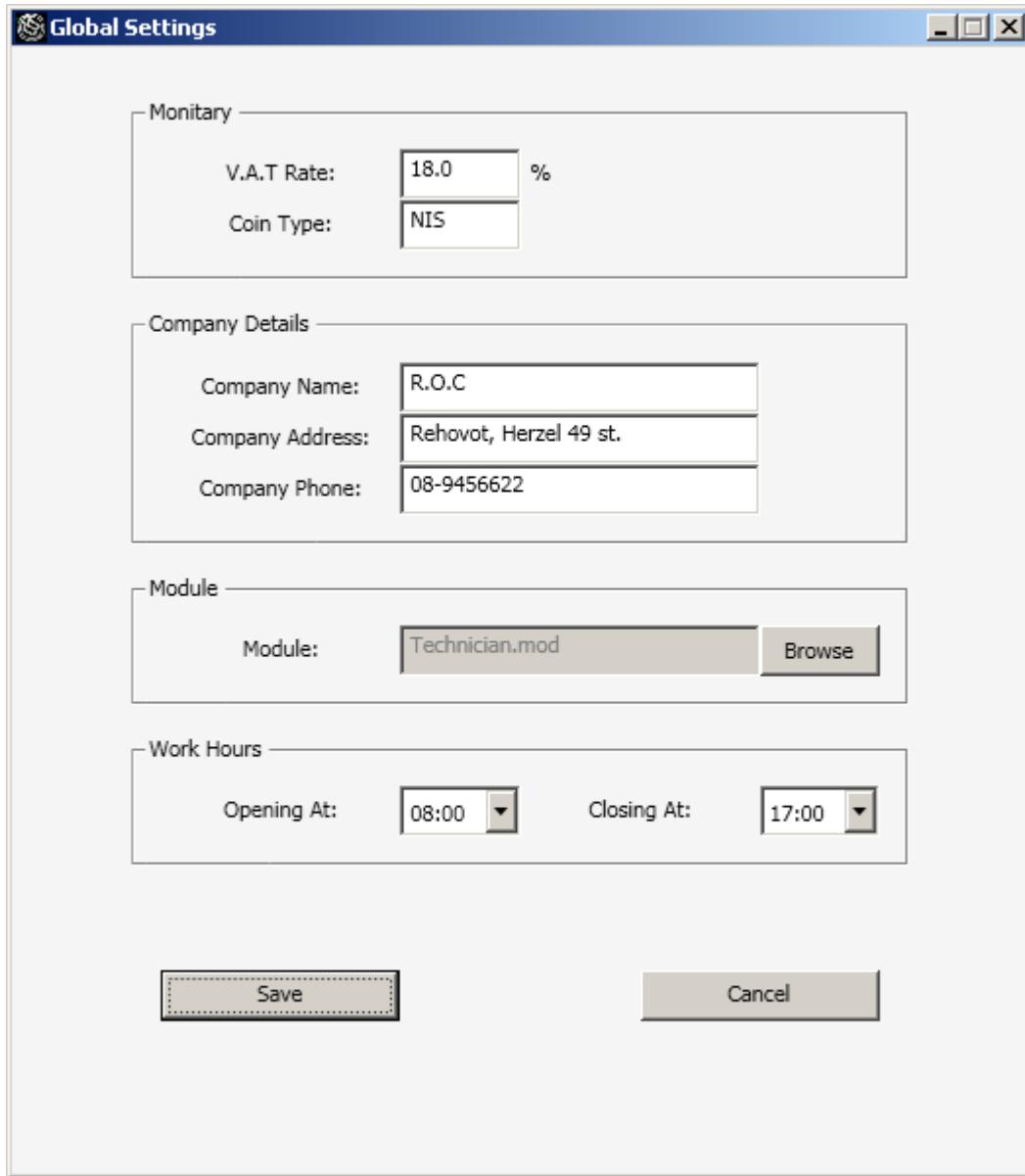
הוספה משתמש חדש או עדכון משתמש קיים



- מסך זה מאפשר להוסיף משתמש חדש אל המערכת או לוחילופין לערוך משתמש קיים.
- במידה והמשתמש הוא משתמש חדש, הטופס יופיע עם פרטים ריקים, ויש למלאו על פי ההנחיות המופיעות בו.
- במידה והמשתמש כבר קיים, הטופס יופיע עם הפרטים הקיימים, וניתן יהיה לשנותם (מלבד שם המשתמש).

- סימון התיבה: **user must change password at next logon** ולחיצה על הכפתור **Save** תיאלץ את המשתמש להחליפ סיסמה בהתחברות הבאה למערכת. אותו משתמש לא יוכל להתחבר אל המערכת לפני שינוי את סיסמתו.
- לאישור ושמירה יש ללחוץ על כפתור **Save**. לביטול יש ללחוץ על כפתור **Cancel**.

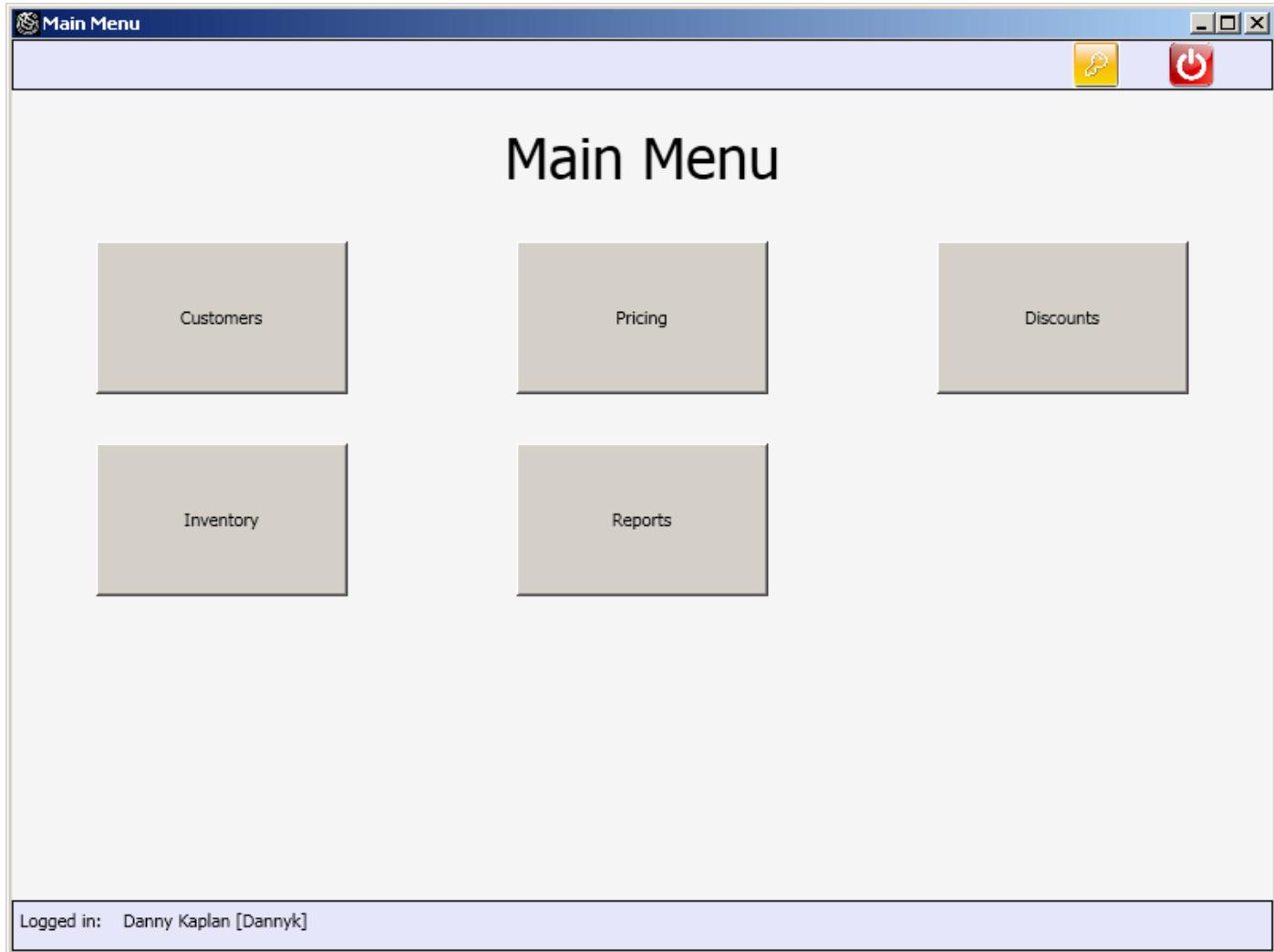
הגדרות כלליות (Global Settings)



- מסך זה מאפשר לקבוע מספר הגדרות כלליות הדרשיות לשימוש התוכנית:
 - הגדרות סופיות (מע"מ, ומطبع בשימוש)
 - פרטי העסק (שם העסק, כתובת, וטלפון)
 - מודול פעיל (ניתן לבחור מודול לשימוש כגון Technician או להשאר ריק)

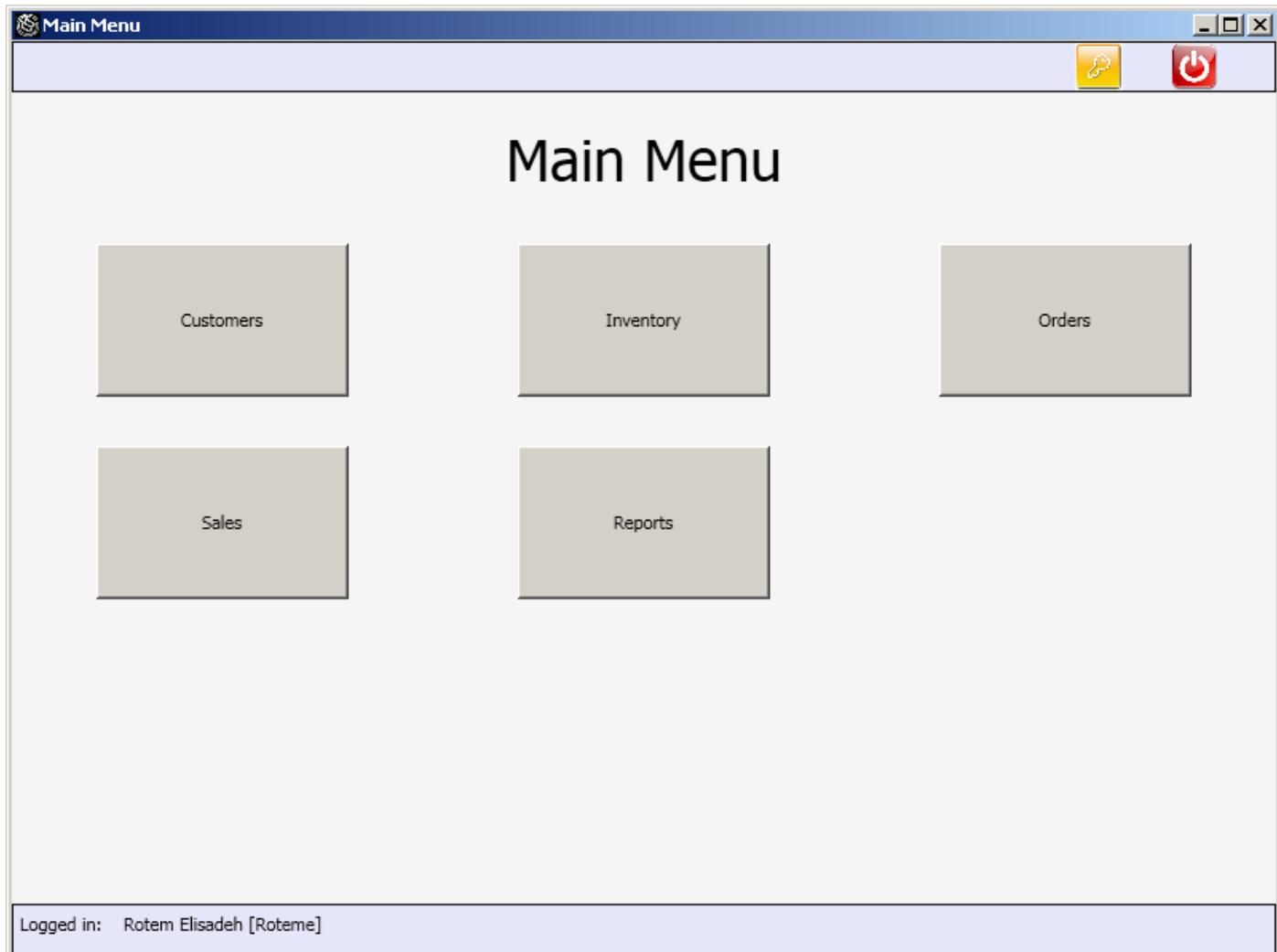
- שעות הפעילות של העסק (משמש עבור הדוחות)
- לאישור ושמירה יש ללחוץ על כפתור ה-Save. לביטול יש ללחוץ על כפתור ה-Cancel.
- אם השדה של קובץ המודול שונה, אז לחיצה על כפתור ה-Save תטعن תחילת המודול ורק לאחר מכן, אם הטעינה הצלילה אזי תבוצע שמירה של הנתונים.

מסך ראשי (מנהל העסק)



- לאחר התחברות של משתמש בעל הרשאה 'מנהל העסק' יופיעו הcptורים הבאים:

- ניהול לקוחות (Customers)
- ניהול מחירים (Pricing)
- ניהול מבצעים (Discounts)
- ניהול מלאי (Inventory)
- הפקת דוחות (Reports)

מסך ראשי (מנהל מכירות)

- לאחר התחברות של משתמש בעל הרשאה 'מנהל מכירות' יופיעו הcptורים הבאים:

- ניהול לקוחות (Customers)

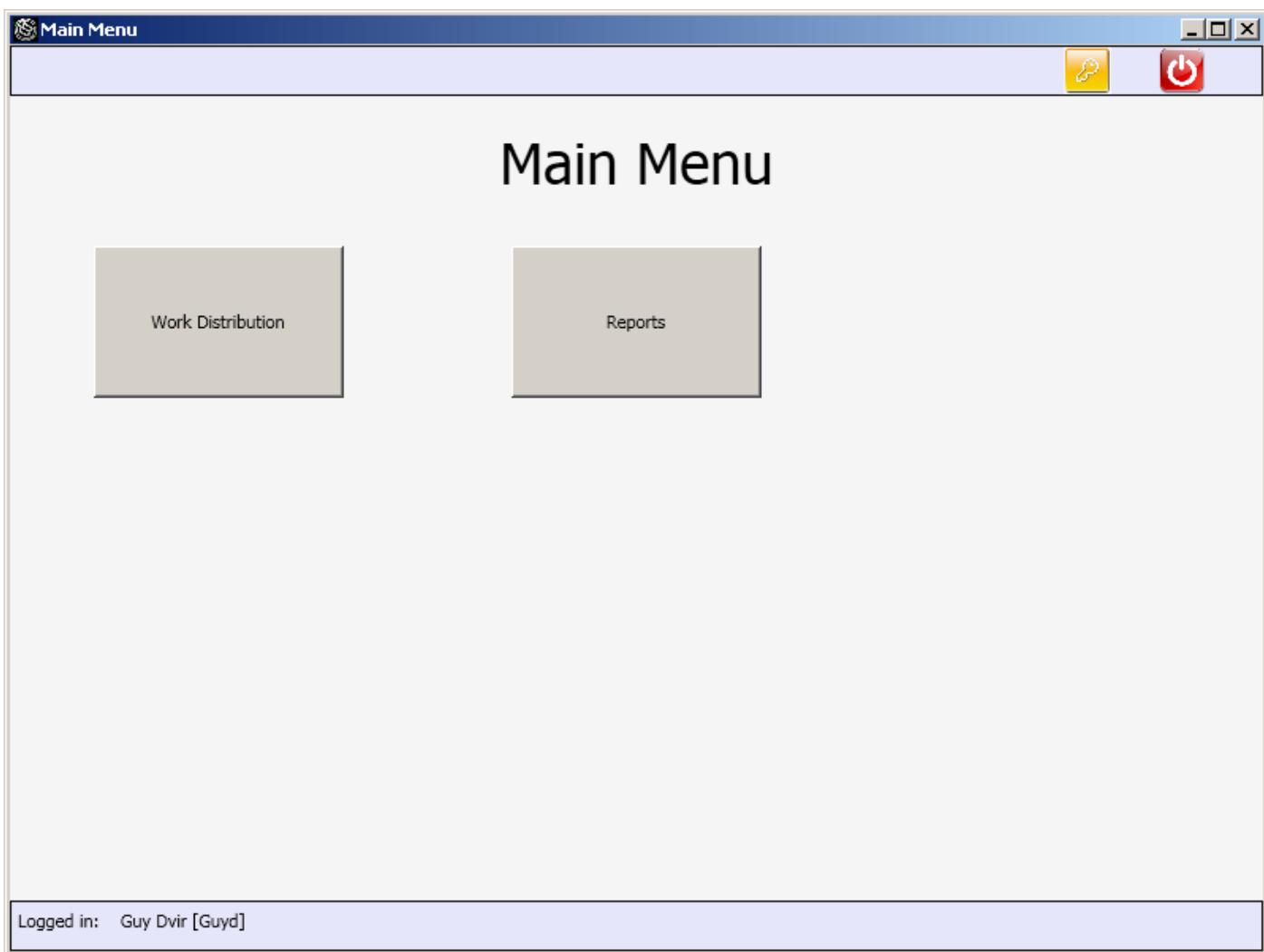
- ניהול מלאי (Inventory)

- ניהול הזמנות (Orders)

- ניהול מכירות (Sales)

- הפקת דוחות (Reports)

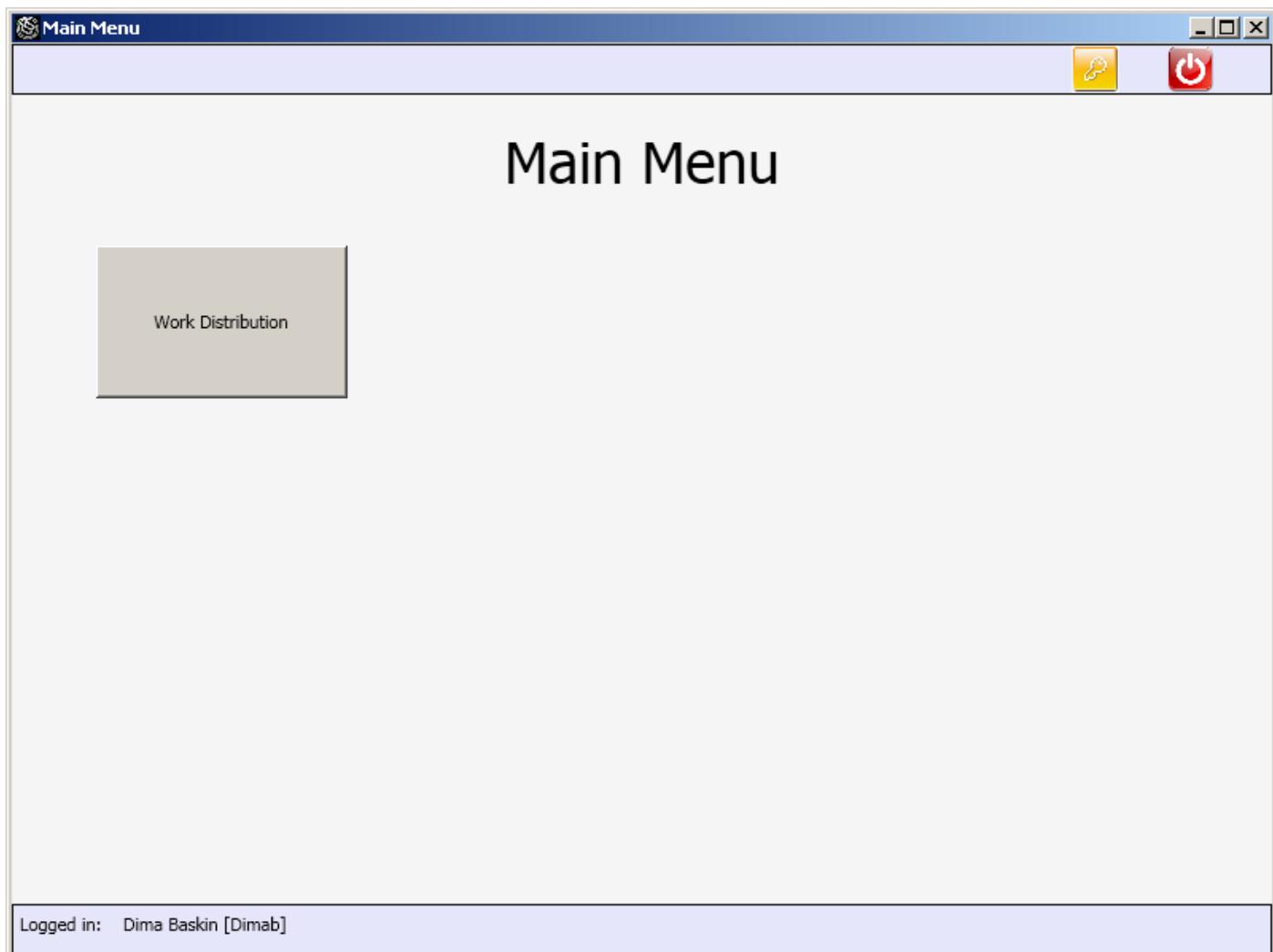
מסך ראשי (ראש צוות)



- לאחר התחברות של משתמש בעל הרשאה 'ראש צוות' יופיעו הcptורים הבאים:
 - חלוקת עבודה (Work Distribution)
 - הפקת דוחות (Reports)

SmartBiz

מסך ראשי (עובד צוות) (Main Screen (Team Worker))



- לאחר התחברות של משתמש בעל הרשאה 'עובד צוות' יופיע הכפטור הבא:

- חלוקת עבודה (Work Distribution)

הערה: יתכנו ו阿里ציות נוספות של מסך ראשי כאשר משתמש בעל שתי הרשאות שונות או יותר מתחבר אל המערכת.

ניהול לכוחות (מנהל העסק)

Customers

The screenshot shows a Windows application window titled "Customers". The top bar includes standard window controls (minimize, maximize, close) and icons for file operations (New, Open, Save, Print, Exit). A toolbar below the menu bar contains icons for search (magnifying glass), filter (checkmark), and other functions.

A "Filter By" section is located at the top left, featuring a dropdown menu with options: "By Id" (selected), "By Name", and "By City". To the right of the dropdown is a search input field and a "Search Customer" button.

The main content area displays a table of customer data:

ID	First Name	Last Name	City	Street	Zip Code	Phone Home	Phone Mobile	Phone Fax	Birth Date
011531746	Rotem	Elisadeh	Asdod	Irisim 1 st.	60860	08-8691410	053-2255515		06/12/198
012345678	Gilad	Mor	Tel Aviv	Zion 45 st.	60890	03-8691111	052-2588899	03-8691112	20/08/198
032554422	Omer	Bar-On	Rehovot	Hertzel 45 st.	76804	08-9455333	054-2631111	08-9455443	03/07/198
123456789	Boris	Levin	Beer Sheva	Herzel 2 st.	80800	08-8694040	050-1235245	08-8694041	20/08/201

At the bottom left, a message states "Total Records Found: 4".

- מסך ניהול הליקוחות מאפשר למנהל העסק גישה לצפיה בלבד בפרטי הליקוחות, כאשר הוא יכול להפעיל חיפוש על פי אחד מהשדות הנתונים ולהציג את אותם הליקוחות אשר מתאימים כתוצאה מהחיתוך.
 - לחיצה כפולה על ליקוח שהתקבל כתוצאה מהחיתוך, תפתח מסך מפורט אודותיו.

ניהול לקוחות (מנהל המכירות)

ID	First Name	Last Name	City	Street	Zip Code	Phone Home	Phone Mobile	Phone Fax	Birth Date
011531746	Rotem	Elisadeh	Asdod	Irisim 1 st.	60860	08-8691410	053-2255515		06/12/198
012345678	Gilad	Mor	Tel Aviv	Zion 45 st.	60890	03-8691111	052-2588899	03-8691112	20/08/198
032554422	Omer	Bar-On	Rehovot	Hertzel 45 st.	76804	08-9455333	054-2631111	08-9455443	03/07/198
123456789	Boris	Levin	Beer Sheva	Herzel 2 st.	80800	08-8694040	050-1235245	08-8694041	20/08/201

Total Records Found: 4

Add Customer Remove Customer

Logged in: Rotem Elisadeh [Roteme]

- מסך ניהול לקוחות מאפשר למנהל המכירות גישה לצפיה ו שינוי בפרטי לקוחות.
- מנהל המכירות יכול להפעיל חיפוש על פי אחד מהשדות הנתונים ולהציג את אותם לקוחות אשר מתקבלים כתוצאה מהחיתוך.
- לחיצה כפולה על לקוח שהתקבל כתוצאה מהחיתוך, תפתח מסך מפורט אודותיו.
- מנהל המכירות יכול להוסיף לקוח חדש על ידי לחיצה על כפתור Add Customer.
- מנהל המכירות יכול לסמן לקוח אחד או יותר וללחוץ על כפתור Remove Customer בצדיו. למחוק את לקוחות מממערכת המידע.

טופס פרטי לקוחות (מנהל העסק)

Customer Details

The screenshot shows a Windows application window titled "Customer Details". The interface includes:

- Name:** Fields for First Name ("Boris") and Last Name ("Levin").
- Address:** Fields for Street ("Herzel 2 st."), City ("Beer Sheva"), and Zip Code ("80800").
- Phone:** Fields for Home ("08-8694040"), Mobile ("050-1235245"), and Fax ("08-8694041").
- History:** A table listing customer interactions:

Hid	Type	Date	Time	Description
1000	Price Offer	16/09/2013	15:28	Customer received price off
1001	Purchase	16/09/2013	15:28	Customer purchased at 16/09/2013
1002	Payment	16/09/2013	15:28	Customer paid at 16/09/2013
1003	Service Call	16/09/2013	15:28	A new service call [Call Id: 1003]
- Log-in:** Status bar at the bottom left showing "Logged in: Danny Kaplan [Dannyk]".

- מסך זה מתאים לתוצאה מלחיצה כפולה על רשומה של לקוח במסך ניהול לקוחות.
- מנהל העסק לא יכול לעורר את שדות הטקסט המופיעים בטופס מכיוון שיש לו הרשות צפיה בלבד.
- מנהל העסק יכול לצפות ברישומי ההיסטוריה של הלוקו על ידי לחיצה כפולה על אחת מרשומות ההיסטורית. (חלקים נוצרים במערכת באופן אוטומטי וחלקים באופן ידני ע"י מנהל המכירות).

SmartBiz

טופס פרטי לקוחות (מנהל המכירות)

Customer Details

Name

First Name: Boris
Last Name: Levin

Address

Street: Herzl 2 st.
City: Beer Sheva
Zip Code: 80800

Phone

Home: 08-8694040
Mobile: 050-1235245
Fax: 08-8694041

History

Hid	Type	Date	Time	Description
1000	Price Offer	16/09/2013	15:28	Customer received price off
1001	Purchase	16/09/2013	15:28	Customer purchased at 16/09/2013
1002	Payment	16/09/2013	15:28	Customer paid at 16/09/2013
1003	Service Call	16/09/2013	15:28	A new service call [Call Id: 1003]

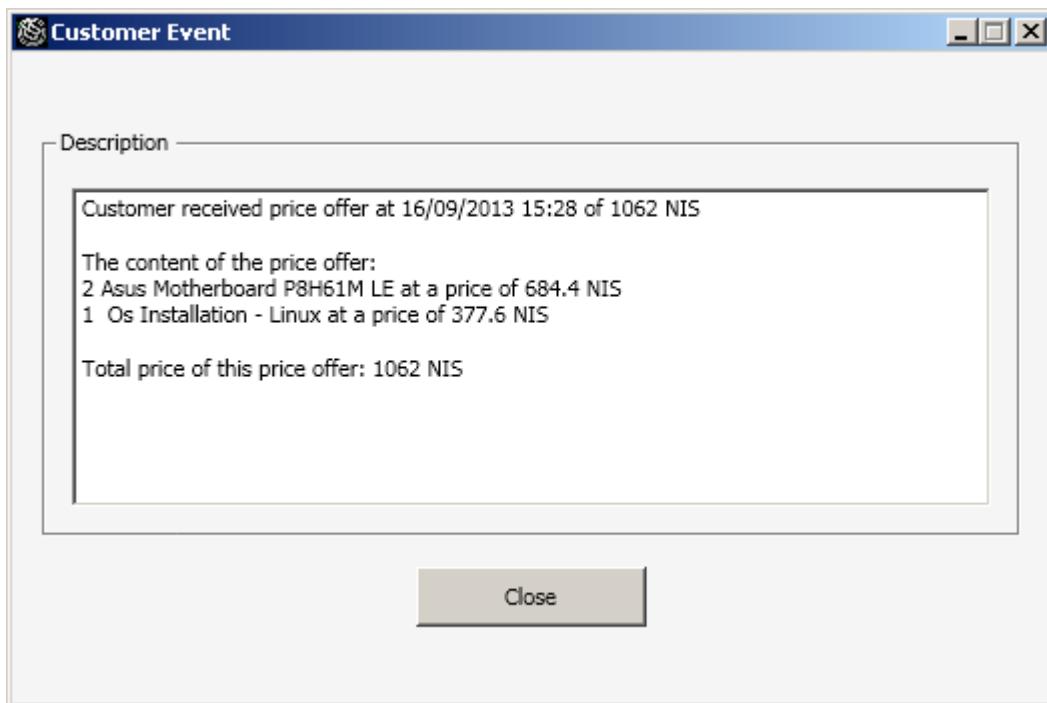
Add Remove

Save Cancel

Logged in: Rotem Elisadeh [Roteme]

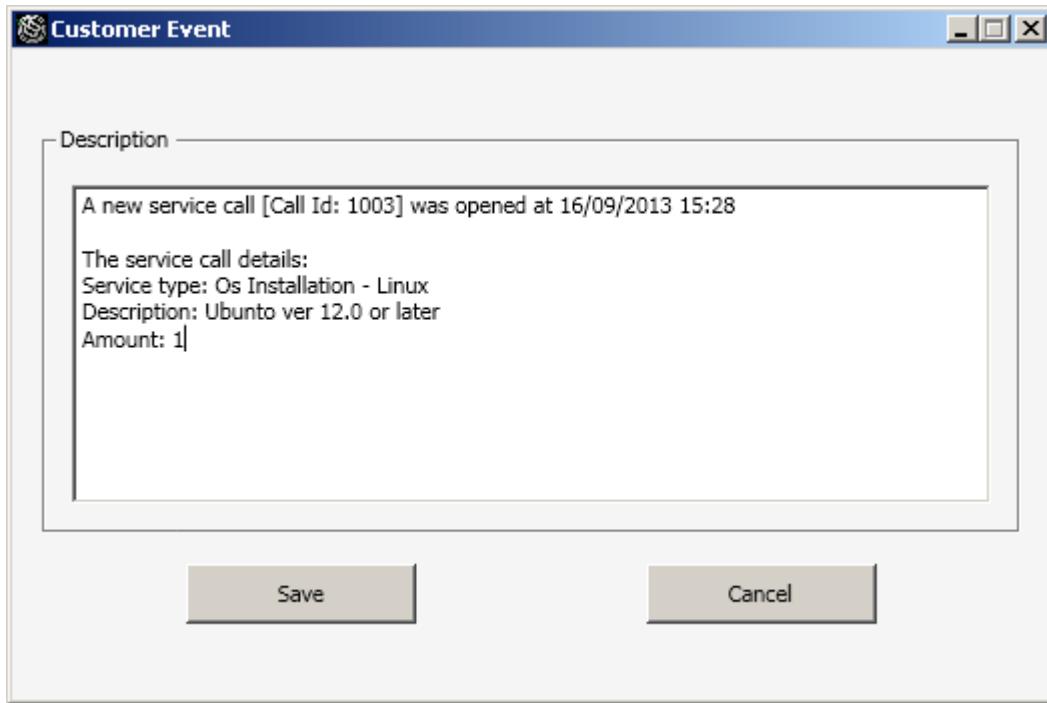
- מסך זה מתאים לתצואה מלחיצה כפולה על רשומה של לקוח במסמך ניהול לקוחות.
- מנהל המכירות יכול לעורר את פרטיו של הלקוח באמצעות שדות הטקסט אשר מופיעים בטופס, מלבד תעודת הזהות של הלקוח.
- מנהל המכירות יכול לצפות/לערוך רישומי היסטוריה של לקוח על ידי לחיצה כפולה על אחת מרשומות ההיסטוריה.
- בנוסף, מנהל המכירות יכול להוסיף/להסר רישומי היסטוריה של לקוח על ידי שימוש בכפתורים Add ו-Remove.

- ב כדי לשמר את השינויים בפרטי הלוקוח יש ללחוץ על כפתור ה-Save.
- ב כדי לבטל את השינויים שבוצעו בפרטי הלוקוח יש ללחוץ על כפתור ה-Cancel.

ההיסטוריה ללקוח (מנהל העסק)

- מסר זה מאפשר לצפות באירועו ההיסטורי של לקוח.
- ההיסטוריה נוצרת באופן אוטומטי ע"י פעולות שונות במערכת או באופן ידני ע"י מנהל המכירות. כר' למשל, בדוגמה ש�示ת להלן מתואר רישום שנוצר אוטומטית עבור לקוח לאחר קיביל הצעת מחיר ממנהל המכירות. בדוגמה הנוכחית ניתן לראות את תוכן ההצעה, הערות הכלולות ואת זמן מתן ההצעה.

ההיסטוריה ל��וח (מנהל המכירות)



- מסר זה מאפשר לצפות/לערוך אירוע היסטוריה ל��וח.
- ההיסטוריה נוצרת באופן אוטומטי ע"י פעולות שונות במערכת או באופן ידני ע"י מנהל המכירות. כר' למשל, בדוגמה שמצוגת להלן מתואר רישום שנוצר אוטומטית עבור לקוח אשר נפתחה עבורו קריית שירות במערכות בעקבות שירות אשר רכש. בדוגמה הנוכחית ניתן לראות את תוכן הפניה, מספר הקראיה שנוצרה באופן אוטומטי ואת זמן פתיחת הקראיה.
- בשונה מהמסמך הקודם, במסר זה ניתן לערוך את הטקסט המופיע בתיאור האירוע וללחוץ על כפתור ה-Save לשמרות השינויים.
- לחיצה על כפתור ה-Cancel תבטל את השינויים שבוצעו.

SmartBiz

ניהול מכירות

The screenshot shows the 'Sales' module of the SmartBiz application. The window title is 'Create Price Offer'. At the top right are standard window controls (minimize, maximize, close). Below them are three icons: a house, a magnifying glass, and a power button.

The main area contains several input fields and a table:

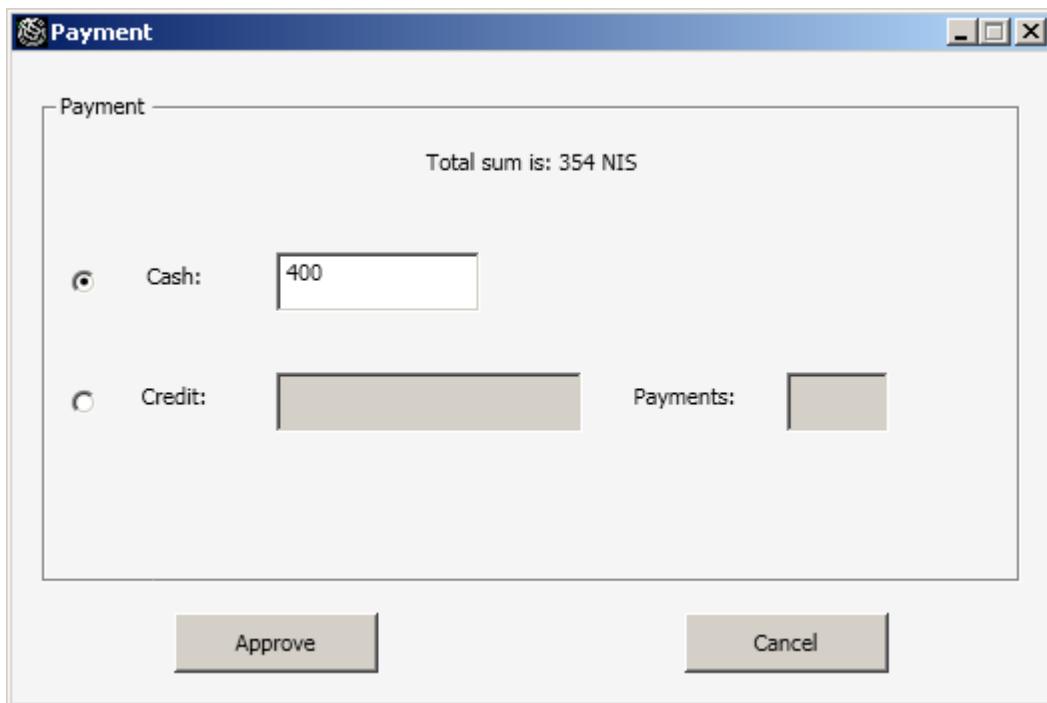
- A 'Link To Customer' button and a status message: 'Status: No Customer Is Linked'.
- Manufacturer, Type, and Name dropdowns.
- Service Type: 'Os Installation - Windows' selected, with a Description field.
- Amount: '1' entered in a text box.
- A table showing product details:

Pid	Manufacturer	Type	Name	Price Per Unit Inc. V.A.T	Amount	1+1	Dis	Total Price
1011	Kingston	Memory	KVR1333D3N9/2G	118	3	<input type="checkbox"/>	0	354
- Total Cost: 354 NIS.
- Action buttons at the bottom: Approve, Add, Remove (disabled), and Record Price-Offer.

At the bottom left, it says 'Logged in: Rotem Elisadeh [Roteme]'.

- מסך זה זמין למנהל המכירות בלבד, ומשמש אותו בכדי להציג הצעות מחיר ולבצע מכירה ללקוחות.
- עם הכניסה למסך זה יש לקשר ללקוח.
- בכדי לקשר ללקוח זה יש ללחוץ על כפתור Link To Customer. (כפתור זה יפתח את מסך ניהול הלקוחות ויאפשר לבחור לקשר קיימים / ליצור חדש ולשייר אותו לטופס זה באמצעות כפתור 'יעודי שעלי' כתוב Sale (Link To Sale)).

- ברגע שלקוח יקשר כהלה, הסטאטוס שכרגע מופיע במצב אדום, ישנה לצבוע ירוק, וכן הטקסט המופיע בו יוחלף בטקסט "Customer Is Linked" (כאשר במקום המילה יופיע השם המלא של הלוקוח).
- לאחר קישור הלוקוח, מנהל המכירות יוסיף מוצרים אותם מעוניין הלוקוח לרכוש או לחילופין יבחר בסוג שירות בו הלוקוח מעוניין (זאת על ידי בחירה בcpfutor הרדיו העליון, או בcpfutor הרדיו התיכון, בהתאם) יבחר נמות, ולאחר מכן, לחץ על cpfutor Add.
- המסך מאפשר לצפות בהנחות שיש כרגע על מוצרים שנבחרו. אם השדה 1+ מסומן משמעו שה מוצר משתתף בהנחה שני מוצרים במחיר של אחד. אם מופיע ערך שונה מ-0 תחת השדה (%) משמעו שה מוצר תחת אותו אחוז הנחה שמספרת בשדה.
- במידה ומנהל המכירות רוצה להסיר מוצר/שירות שהוסיף קודם לכן להצעת המחיר, הוא יסמן אותם ולחץ על cpfutor Remove.
- רק לאחר שלקוח מקשור לטופס זה, וההצעה מחיר כוללת לפחות מוצר או שירות אחד, אז cpfutor Approve יופיע כזמן להחיצה, וניתן יהיה ללחוץ עליו כדי לעבור למסך קבלת תקבולות. בחירה בcpfutor Approve משמעותה שלוקוח קיבל לידי הצעת המחיר.
- לצורך הבהרה: על אף שלא ניתן להתקדם לשלב קבלת התקבולות לפני שקשר לлокוח, ניתן להציג הצעת מחיר לлокוח מזמן שלא רוצה להזדהות, ורק רוצה לקבל הצעת מחיר.
- לחיצה על cpfutor Record Price-Offer תיצור רישום היסטורייה באופן אוטומטי שיישמש כתיעוד להצעת המחיר שניתנה לлокוח. במידה ומנהל המכירות לא לחץ על cpfutor זה, ההצעה מחיר שנותן לлокוח לא תתווד בכרטיס הלוקוח. רק לאחר שלקוח מקשור לטופס זה, וההצעה מחיר כוללת לפחות מוצר או שירות אחד, אז cpfutor Record Price-Offer יופיע כזמן להחיצה.

קבלת התקבולים

- מסך זה מציג את סכום הרכישה, ומציג אפשרות תשלום באמצעות כסף מזומנים או כרטיס אשראי.
- במידה ונבחר כסף מזומנים הסכום מזון בשדה הטקסט.
- במידה ונבחר כרטיס אשראי, מספר הלקוח ומספר התשלומים מזונים בשדות המתאיםים.
- לאחר שההזנה הסת衣ימה יש ללחוץ על כפתור ה-Approve בצד לאישור את הרכישה.
- במידה והליך התחרט ניתן ללחוץ על כפתור ה-Cancel.
- עם לחיצה על כפתור ה-Approve תיווצר רשומת היסטוריה באופן אוטומטי שתתעד את פרטי הרכישה והעלות ותשאיר ללקוח שביצע את הרכישה. בנוסף הנקודות של המוצרים שנרכשו תגሩ מהמלאי ובעור השירותים הנרכשים יפתחו קרייאות שירות חדשות.

הפקת חשבונית

- מסך זה מציג חשבונית שתפקידו ללקוח.
- החשבונית כוללת את פרטי העסק, תאריך וזמן הרכישה, שם המוכרן ומספר החשבונית.
- כמו כן, מופיע הסכום שהתקבל מהלקוח, עלות הרכישה, והעודף אשר יש להחזיר ללקוח.
- לחיצה על כפתור **the Close** תסגור את החלון.

SmartBiz

תמחור מוצרים

The screenshot shows the 'Pricing' window of the SmartBiz application. At the top, there are two input fields: 'V.A.T Rate:' set to 18% and 'Raise/Lower Prices By:' set to 10%. Below these are two buttons: 'View Price Changes' and 'Save'. The main area contains a table of products with columns for Pid, Manufacturer, Type, Name, Current Price, Current Price Inc. Current V.A.T, New Price, and New Price Inc. New V.A.T. The table includes items like Motherboards, Processors, Memory, Graphics Cards, and OS installations. At the bottom of the window, there is a 'Cancel' button.

Pid	Manufacturer	Type	Name	Current Price	Current Price Inc. Current V.A.T	New Price	New Price Inc. New V.A.T
1004	Gigabyte	Motherboard	GA-G41M-COMBO	322	379.96	354.2	417.96
1005	Intel	Motherboard	DG43RK	240	283.2	264	311.52
1006	Asus	Motherboard	P8H61M LE	290	342.2	319	376.42
1007	Gigabyte	Motherboard	GA277DS3H	517	610.06	568.7	671.07
1008	Intel	Processor	Core i5 3470 Tray	799	942.82	878.9	1037.1
1009	Intel	Processor	Core i7 3770 Tray	1290	1522.2	1419	1674.42
1010	Intel	Processor	Pentium G2030	261	307.98	287.1	338.78
1011	Kingston	Memory	KVR1333D3N9/2G	100	118	110	129.8
1012	Gigabyte	Graphics Card	GeForce GT 630	350	413	385	454.3
2002	N/A	Os Installation - Windows	N/A	300	354	330	389.4
2003	N/A	Os Installation - Linux	N/A	320	377.6	352	415.36
2004	N/A	Anti Virus Installation	N/A	20	23.6	22	25.96

- מסך זה מאפשר לעשות בו שימוש לעדכון תעריפים ומחירים לכל המוצרים / שירותים
(שינוי מחיר למוצר בודד יבוצע בחלון אחר).
- מסך זה מאפשר לקבוע תעריף מע"מ שונה מזה שנקבע לאחרונה (התעריף הראשון נקבע במסך הגדרות כלליות).
- מסך זה מאפשר להוריד/להעלות את התעריפים של המוצרים כולם באחוז כלשהו.
- לחיצה על הכפתור View Price Changes תאפשר לראות את רמת המחיר החדשם לאחר ביצוע שינויים בשדות הטקסט.
- טבלת הנתונים תציג בכל עת את כל המוצרים והשירותים הזמינים (בכדי להראות כיצד שינוי המחיר ישפיעו עליהם בצורה רוחבית).
- לחיצה על כפתור Save תשמר את השינויים שביצעתו.
- לחיצה על כפתור Cancel תבטל את השינויים שביצעתו.

ניהול מלאי (מנהל העסק)

Inventory

The screenshot shows a Windows-style application window titled "Inventory". At the top, there's a toolbar with icons for home, search, and power. Below it is a "Filter Products" section with three radio button options: "Pid:" (selected), "Manufacturer:" (Intel), and "Service Type:". There are also dropdown menus for "Type:" and "Name:", and a "With Amount From:" input field with a "To:" field next to it. A "Search Product" button is located to the right of these fields. Below the filter section is a table of product data:

Pid	Amount	Manufacturer	Type	Name	Price	V.A.T	Price Inc. V.A.T	Is Service?	Is Product?
1005	20	Intel	Motherboard	DG43RK	240	43.2	283.2	<input type="checkbox"/>	<input checked="" type="checkbox"/>
1008	12	Intel	Processor	Core i5 3470 Tray	799	143.82	942.82	<input type="checkbox"/>	<input checked="" type="checkbox"/>
1009	2	Intel	Processor	Core i7 3770 Tray	1290	232.2	1522.2	<input type="checkbox"/>	<input checked="" type="checkbox"/>
1010	5	Intel	Processor	Pentium G2030	261	46.98	307.98	<input type="checkbox"/>	<input checked="" type="checkbox"/>

Total Records Found: 4

Add **Remove**

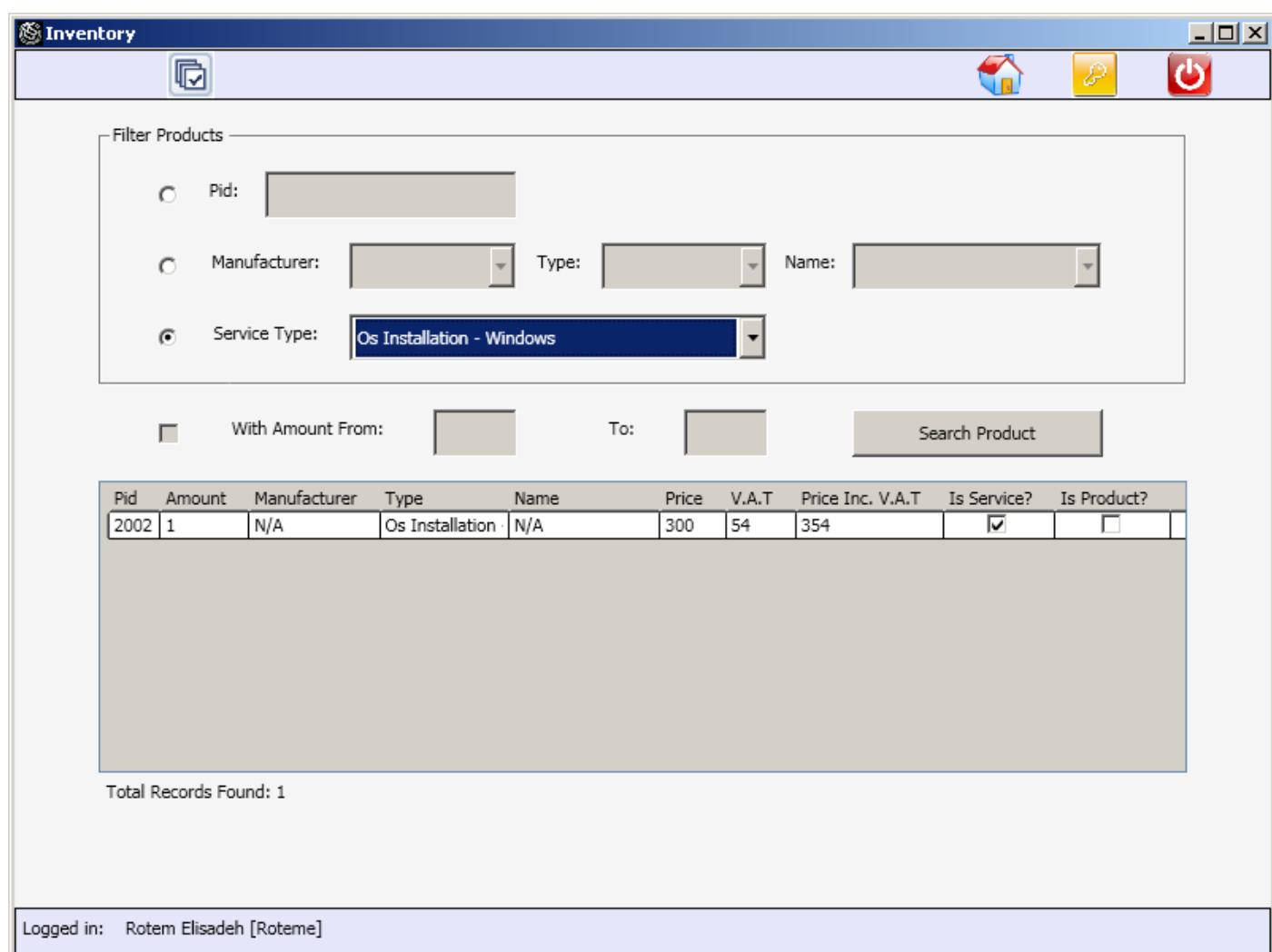
Logged in: Danny Kaplan [Dannyk]

- מסך זה משמש בכדי לצפות במוצרים / שירותים הנמצאים במלאי העסק, כאשר יש אפשרות לבצע סינון על פי קטגוריות שונות.
- בחירה בcptor הרדיו העליון תאפשר סינון על פי שדה Pid (Id = מק"ט).
- בחירה בcptor הרדיו האמצעי תאפשר סינון על פי יצרן, סוג, ושם המוצר.
- בחירה בcptor הרדיו התחתון תאפשר סינון על פי סוג השירות.
- בחירה בתיבת הסימון תאפשר סינון על פי טווח כמות מוצר במלאי, וניתן לשלהבה יחד עם האפשרות לסינון המוצרים בכדי להגיע לסינון אידיאלי.

SmartBiz

- לחיצה כפולה על אחת הרשומות מאפשר עדכון הרשומה, הן מבחינת הכמות הנמצאת כרגע במלאי (למשל עברו גריית מלאי) והן מבחינת העלות ליחידה.
- לחיצה על כפתור ה-Add מאפשר להוסיף מוצר חדש למלאי שלא היה קיים לפני כן.
- סימון מוצר אחד או יותר ולחיצה על Remove תסיר את סוג המוצר מהמלאי באופן קבוע.

ניהול מלאי (מנהל מכירות)



The screenshot shows the 'Inventory' module of the SmartBiz system. At the top, there's a toolbar with icons for Home, Search, and Logout. Below it is a 'Filter Products' section with three radio button options: 'Pid:' (selected), 'Manufacturer:', and 'Service Type:'. The 'Service Type:' dropdown is set to 'Os Installation - Windows'. Below the filter section are buttons for 'With Amount From:' and 'To:', and a 'Search Product' button. A table below displays one record:

Pid	Amount	Manufacturer	Type	Name	Price	V.A.T	Price Inc. V.A.T	Is Service?	Is Product?
2002	1	N/A	Os Installation	N/A	300	54	354	<input checked="" type="checkbox"/>	<input type="checkbox"/>

Total Records Found: 1

At the bottom left, a status bar shows 'Logged in: Rotem Elisadeh [Roteme]'

- בעת גישה למסך ניהול המלאי, מנהל המכירות מקבל גרסה מצומצמת מבוחינה פונקציונלית של המסר הקודם שהוצג. וזאת מכיוון שרק מנהל העסק היה רשאי להוסיף מוצרים חדשים אל המלאי או להסיר אותם, וכן, לעדכן אותם.

SmartBiz

הוספה פריט חדש למלאי או עדכון פריט קיים

Add/Update Inventory Item

Product Details

Manufacturer:	Intel	Cost Per Unit (NIS):	1250
Type:	Processor	Existing Amount:	5
Name:	Core i7 3770 Tray		

Service Details

Type	
Service Cost:	NIS

Buttons: Save | Cancel

- מסך זה מאפשר הוספה מוצר / שירות חדש אל המלאי או לעריכת מוצר / שירות קיים.
- ניתן להיעזר ביצרנים, קטגוריות, ושמות קיימים או לחילופין להוסיף חדשים.
- יש לבחור האם מעוניינים להוסיף מוצר או שירות באמצעות כפתורי הרדיו.
- בעת הוספה מוצר חדש, יש לציין מהו מחירו (לפניהם), וכן מהי הכמות הנוכחיות הקיימות במלאי.
- בעת הוספה שירות חדש, יש לציין מהו מחירו.
- בעת עדכון פריט קיים, רק השדות Cost ו-Amount יהיו זמינים.

- לחיצה על כפתור ה-Save שומרת את פרטי הפריט החדש, ומיצרת באופן אוטומטי Product Id (Pid) עבורו.
- לחיצה על כפתור ה-Cancel סגורת את החלון מלכתחילה ליצור פריט חדש במלאי.
- **לצרכי הבהרה:** טיפול בהזמנות ציוד שותפות יעשה במסך "יעוד". יש לשנות כמות ציוד אל המלאי באמצעות מסך זה כאשר רוצים לשנות את הכמות בעקבות מקרה חריג.

SmartBiz

ניהול הזמנות ציוד

The screenshot shows the 'Orders' module of the SmartBiz application. At the top, there are search and filter options: 'Search By Oid:' with a text input field, and 'Or Choose' with dropdowns for 'Status' (empty), 'Dealer' (set to 'Bob Dealer'), and date ranges ('From Date: 05/02/2013 15' and 'To Date: 16/09/2013 15'). Below the filters is a 'Search Orders' button. The main area displays a table of order records:

Oid	Dealer	Date Ordered	Time Ordered	Status	Total Cost
1000	Bob Dealer	04/09/2013	10:49	Pending	3800
1002	Bob Dealer	04/09/2013	10:50	Completed	1200

Below the table, it says 'Total Records Found: 2'. At the bottom are four buttons: 'Add Order', 'Approve Order', 'Cancel Order', and 'Remove Order'. A status bar at the bottom left indicates 'Logged in: Rotem Elisadeh [Roteme]'.

- מסך זה מאפשר לניהל את הזמנות הציוד עבור מלאי העסוק, בכך שהעסק יכול להציג מחדש בפריטי מלאי קיימים.
- המסך מאפשר לחפש אחר הזמנות קיימות באמצעות פילטרים שונים.
- לחיצה על כפתור ה-Order Add פותח מסך ליצירת הזמנה חדשה.
- סימון הזמנה/ות ולחיצה על כפתור ה-Order Cancel מעביר את אותה הזמנה לסטטוס Canceled Pending בלבד.

- סימון הזמנה/ות ולחיצה על כפתור ה-Approve Order מעבירה את אותן הזמנות לסטאטוס Completed, וכן, גורמת להוספה אוטומטית של אותן מוצריים אל המלאי בכמות שפורטו בהזמנות. ניתן לאשר הזמנות שהן בסטאטוס Pending בלבד.
- סימון הזמנה/ות ולחיצה על כפתור ה-Remove Order מוחק את אותן הזמנות מהמערכת. ניתן למחוק הזמנות שהן בסטאטוס Canceled בלבד.
- לחיצה כפולה על אחת ההזמנות תפתח את מסך העדכון הזמנה. במסך זה ניתן לעורר את פרטי ההזמנה.

SmartBiz

הוספה הזמנה חדשה / עדכון או צפיה בהזמנה קיימת

Add/Update Order

Form State: Edit Order [Oid = 1000, Order status: Pending]

Dealer: Bob Dealer

Filter Products

Manufacturer:	Kingston	Type:	Memory	Name:	KVR1333D3N9/2G
Amount:	1	Cost Per Unit: 58 NIS			

Add Remove

Pid	Manufacturer	Type	Name	Cost Per Unit	Amount	Total Cost
1009	Intel	Processor	Core i7 3770 Tray	1320	2	2640
1011	Kingston	Memory	KVR1333D3N9/2G	58	20	1160

Total Cost Of Order: 3800 NIS

Save Cancel

Logged in: Rotem Elisadeh [Roteme]

- מסר זה מאפשר ליצור הזמנה חדשה. לצרכי הבהרה, המשמעות היא תיעוד של הזמנות SMB צבעים בעסק, ולא התממשקות ישירה מול אתר הספק.
- בתוית העליונה ניתן לראות את מצב הטופס. קיימים 3 מצבים שונים: 1. יצירת הזמנה חדשה. 2. עריכת הזמנה בסטאטוס Pending. 3. צפיה בהזמנה שנמצאת בסטאטוס Canceled או Completed.
- מסר זה מאפשר להוסיף כמה פריטים להזמנה ייחידה.
- ניתן לבחור ארכור ותיק מוצריים שכבר קיימים במערכת (תזכורת: מנהל העסק מוסיף מוצרים חדשים אל המלאי) וזאת על ידי הפילטרים.

- מסך זה מאפשר לבחור את הheiten ליחידה (מסופקת על ידי גורם חיצוני).
- מסך זה מאפשר לבחור את הכמות שאותה רוצים להזמין.
- לחיצה על כפתור ה-Add מוסיף להזמנה את הפריט שאות פרטיו מילאנו.
- בחירה של פריט/ים כלשהו ולחיצה על כפתור Remove מסיר את אותו הפריט מההזמנה.
- מסך זה מציג את הheiten הכוללת של ההזמנה החדשה שיצרנו.
- לחיצה על כפתור ה-Save עברו הזמנה חדשה, שומרת את הזמנה בסטאטוס Pending ומיצרת באופן אוטומטי Id (Order Id).
- לחיצה על כפתור ה-Cancel סוגרת את החלון הנוכחי לשמר את השינויים האחרונים.



SmartBiz

ניהול מוצרים

The screenshot shows the 'Discounts' window in SmartBiz. At the top, there are sections for 'Choose Discounts' (Mass Discount: 10%, 1+1 Discount: Yes) and 'Filter Products' (Pid: [empty], Manufacturer: Intel, Type: [empty], Name: [empty], Service Type: [empty]). Below these are search and save buttons. A large table lists products with columns: Pid, Manufacturer, Type, Name, Discount (%), 1+1, Price, V.A.T, Price Inc. V.A.T, and Price Inc. V.A.T & Discount. The table contains the following data:

Pid	Manufacturer	Type	Name	Discount (%)	1+1	Price	V.A.T	Price Inc. V.A.T	Price Inc. V.A.T & Discount
1005	Intel	Motherboard	DG43RK	10	<input type="checkbox"/>	240	43.2	283.2	254.88
1008	Intel	Processor	Core i5 3470 Tray	10	<input checked="" type="checkbox"/>	799	143.82	942.82	848.54
1009	Intel	Processor	Core i7 3770 Tray	10	<input checked="" type="checkbox"/>	1290	232.2	1522.2	1369.98
1010	Intel	Processor	Pentium G2030	10	<input checked="" type="checkbox"/>	261	46.98	307.98	277.18

Logged in: Danny Kaplan [Dannyk]

- מסך זה מאפשר לנו ניהול המוצרים של בית העסק.
- ניתן למצוא סוגי מוצרים / שירותים על ידי שימוש בפילטרים הקיימים במסך.
- במידה ורוצים להציג מוצרים על מוצרים / שירותים מסוימים יש לסמן את אותם מוצרים, או לחרופין, ללחוץ על כפתור ה-All Select בצד לסמן את כל המוצרים / שירותים שהתקבלו כתוצאה מהחיתוך.
- יש אפשרות לבחור שני סוגי שונים של מוצרים:
 - מוצר Mass Discount – קביעת הנחה (באחוזים) לכל הרשומות שסומנו.
 - מוצר 1+1 – קביעת מוצר 1+1 לכל הרשומות שסומנו.

- כדי להחיל את המבצע 1+1 יש לסמן את התיבה המתאימה ולבחר ב-Yes לפני הלחיצה על כפתור ה-Save. כדי לבטל את המבצע יש לסמן את התיבה המתאימה ולבחר ב-No לפני הלחיצה על כפתור ה-Save.
- לחיצה על כפתור ה-Save שומר את המבצעים שנבחרו על הרשומות שסומנו.

חלוקת עבודה (ראש הצוות)

The screenshot shows the 'Work Distribution' application window. At the top, there are search options: 'Search By Call Id:' with a text input field, and 'Or Choose' with dropdowns for 'From Date' (01/08/2013), 'To Date' (16/09/2013), 'Service Type', 'Status', and 'Assigned To'. Below these are buttons for 'Search' and 'Print'. The main area displays a table of work items:

Call Id	Service	Amount	Date	Status	Assigned to	Description
1000	Os Installation - Windows	3	06/09/2013	Unassigned	Unassigned	95
1001	Os Installation - Linux	3	06/09/2013	In progress	Shalom Gil	Unix
1002	Anti Virus Installation	2	06/09/2013	Unassigned	Unassigned	Avg or Norton
1003	Os Installation - Linux	1	16/09/2013	Unassigned	Unassigned	Ubuntu ver 12.0 or later

Total Records Found: 4

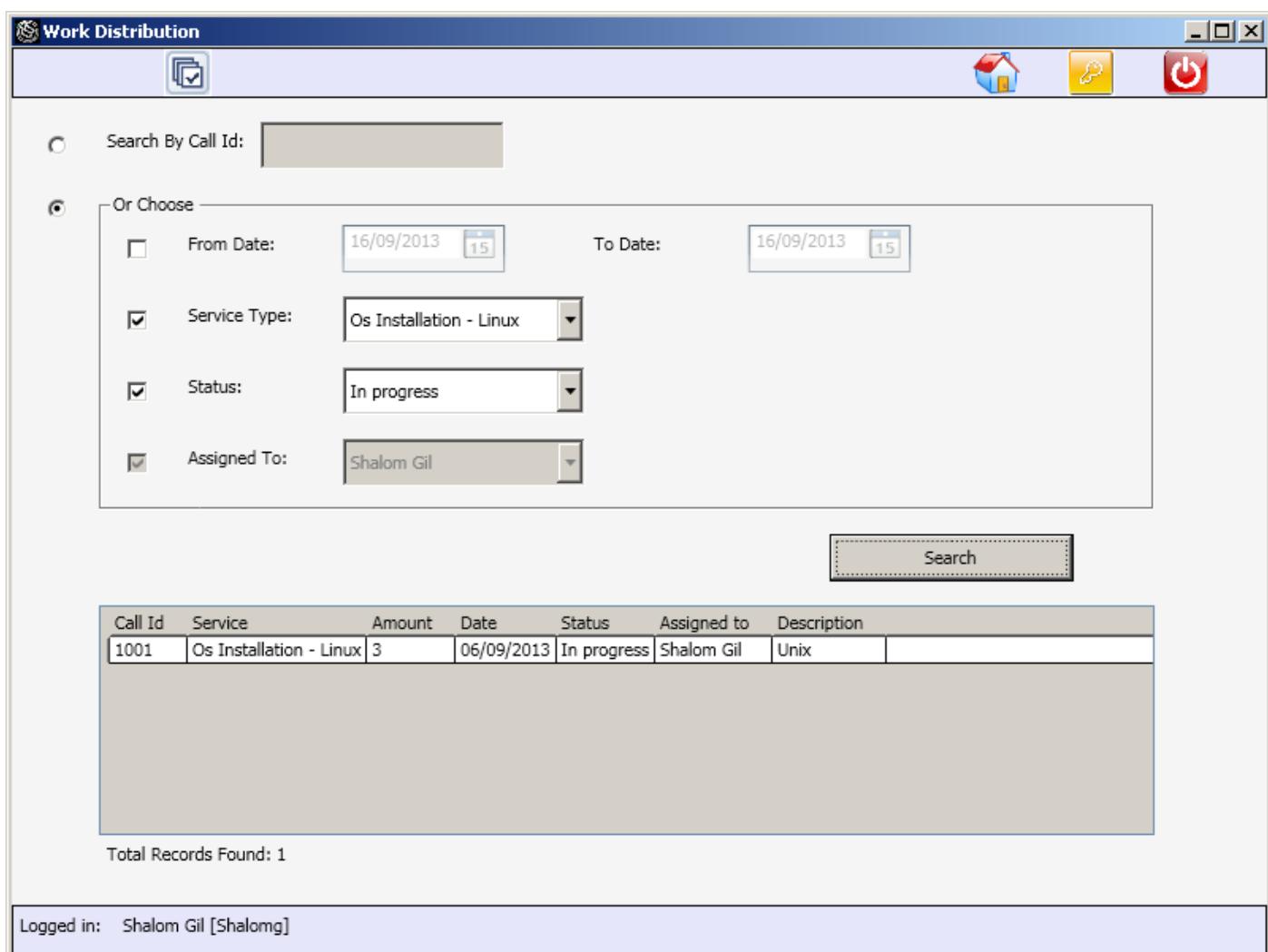
Logged in: Guy Dvir [Guyd]

- מסך זה מאפשר לראש הצוות לצפות בכל קריאות השירות שנפתחו במערכת.

SmartBiz

- כל קריית שירות נפתחת לראשונה בסטטוס Unassigned לאחר שמנהל המכירות מכר ללקוח שירות כלשהו.
- ראש הצוות יכול לעשות שימוש בפילטרים השונים כדי להגיע לחיתוך המבוקש של הקריאות השונות. (סימון על ידי מספר קריאה [Id Call] או מאפיינים שונים).
- לחיצה כפולה על אחת הרשומות פותחת מסך שמאפשר עדכון פרטי הקריאה.

חלוקת עבודה (עובד צוות)



The screenshot shows the 'Work Distribution' module interface. At the top, there are search and filter options:

- Search By Call Id:** A text input field.
- Or Choose:** A group of checkboxes for filtering:
 - From Date:** Set to 16/09/2013.
 - To Date:** Set to 16/09/2013.
 - Service Type:** Set to "Os Installation - Linux".
 - Status:** Set to "In progress".
 - Assigned To:** Set to "Shalom Gil".
- A **Search** button at the bottom of the search area.

The main area displays a table of work records:

Call Id	Service	Amount	Date	Status	Assigned to	Description
1001	Os Installation - Linux	3	06/09/2013	In progress	Shalom Gil	Unix

Total Records Found: 1

At the bottom, a status bar indicates: Logged in: Shalom Gil [Shalomg]

- בניגוד לראש הצוות, עובד הצוות יכול לצפות אף ורק בפרטי קריאות השירות ששוויכו אליו. כלומר, אלו שדה ה-Assigned-Shade שווה לשם של העובד.

- בדומה לראש הוצאות, עובד הוצאות יכול לעשות חיפוש על פי מספר קריאה (Cid) או על ידי שימוש בפילטרים השונים.
- לחיצה כפולה על אחת הרשומות פותחת מסך שמאפשר עדכון פרטי הקריאה.

SmartBiz

קריאת שירות (ראש הוצאות)

Customer Service Call

Details

- Call Id: 1002
- Date and Time: 06/09/2013 15:42
- Service type: Anti Virus Installation
- Description: Avg or Norton
- Amount: 2

Add History

The customer demands a full installation!

Add

Actions

Assign to a team member: Igor Baruch

Change status: In progress

View History

Hid	Type	Date	Time	Comments
1002	Assigned	16/09/2013	17:07	Igor Baruch is assign
1003	Started	16/09/2013	17:07	The service call chan
1004	Manual	16/09/2013	17:09	The customer demar

Remove

Logged in: Guy Dvir [Guyd]

- מסך זה מאפשר לראש הוצאות להקצתו את קריאת השירות לאחד מעובדי הוצאות, וכן לנהל את היסטוריית קריאת השירות.
- ראש הוצאות יכול להקצתו את קריאת השירות לעובד צוות על ידי בחירה בשמו בתיבת Change States. Assign to a team member.
- בצדדי לאשר את ההקצאה ו/או סטאטוס הטיפול יש ללחוץ על כפתור ה-Update.
- ראש הוצאות יכול להוסיף פריט ההיסטוריה חדש על ידי כתיבתו בתיבת הטקסט Add History, ולאחר מכן, ללחוץ על הכפתור Add.

- ראש הצוות יכול להסיר פריט/ים היסטוריה קיימים על ידי סימון אותו פריט/ים ולחיצה על כפתור Remove.
- מלבד האפשרות להוספה ידנית של היסטוריה, המערכת מייצרת רישום אוטומטי כאשר מקצים קריאת שירות לאחד מעובדי הצוות, וכן, כאשר משנים>Status.
- לחיצה כפולה על אחת מרשומות ההיסטוריה תפתח חלון עם פרטי הרשומה.

קריאת שירות (עובד צוות)

The screenshot shows the 'Customer Service Call' application window. The top menu bar includes 'File', 'Edit', 'View', 'Tools', 'Help', and icons for 'New', 'Open', 'Save', 'Print', 'Home', 'Logout', and 'Power'. The main interface is divided into four sections:

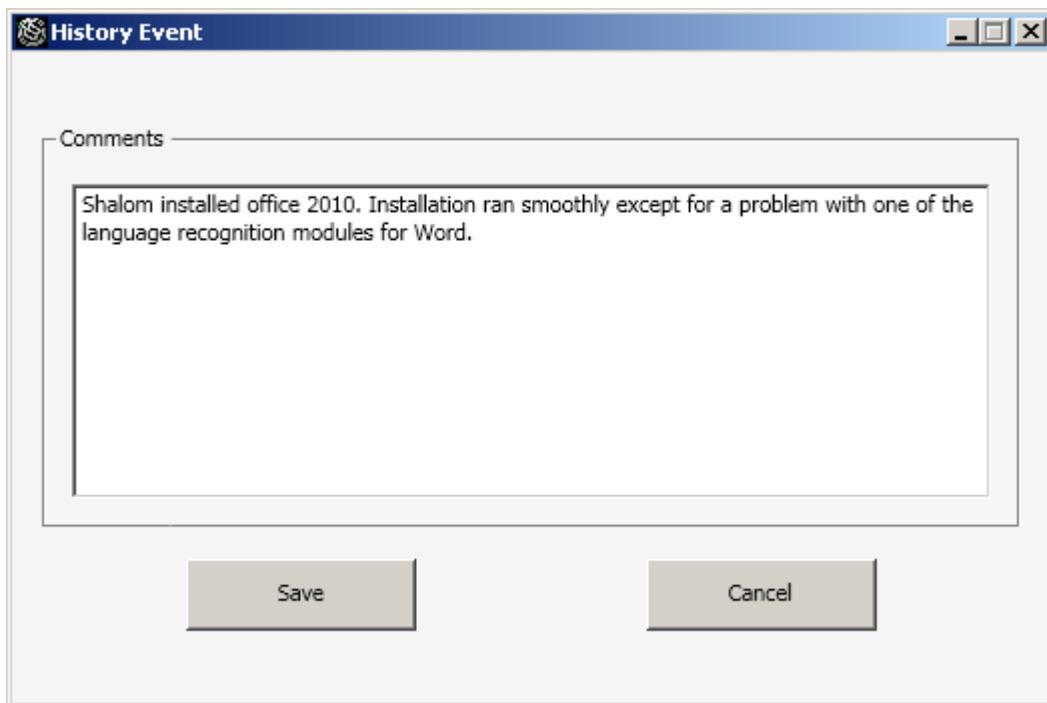
- Details:** Contains fields for Call Id (1001), Date and Time (06/09/2013 15:41), Service type (Os Installation - Linux), Description (Unix), and Amount (3).
- Add History:** A text input field containing 'Time installation: 55 Min.' with an 'Add' button below it.
- Actions:** Includes dropdown menus for 'Assign to a team member' (Shalom Gil) and 'Change status' (Completed), with an 'Update' button at the bottom.
- View History:** A table listing service history entries:

Hid	Type	Date	Time	Comments
1000	Assigned	16/09/2013	17:02	Shalom Gil is assigned
1001	Started	16/09/2013	17:02	The service call chan...
1006	Manual	16/09/2013	17:13	Time installation: 55
1008	Finished	16/09/2013	17:14	The service call chan...

 With navigation arrows and a 'Remove' button at the bottom.

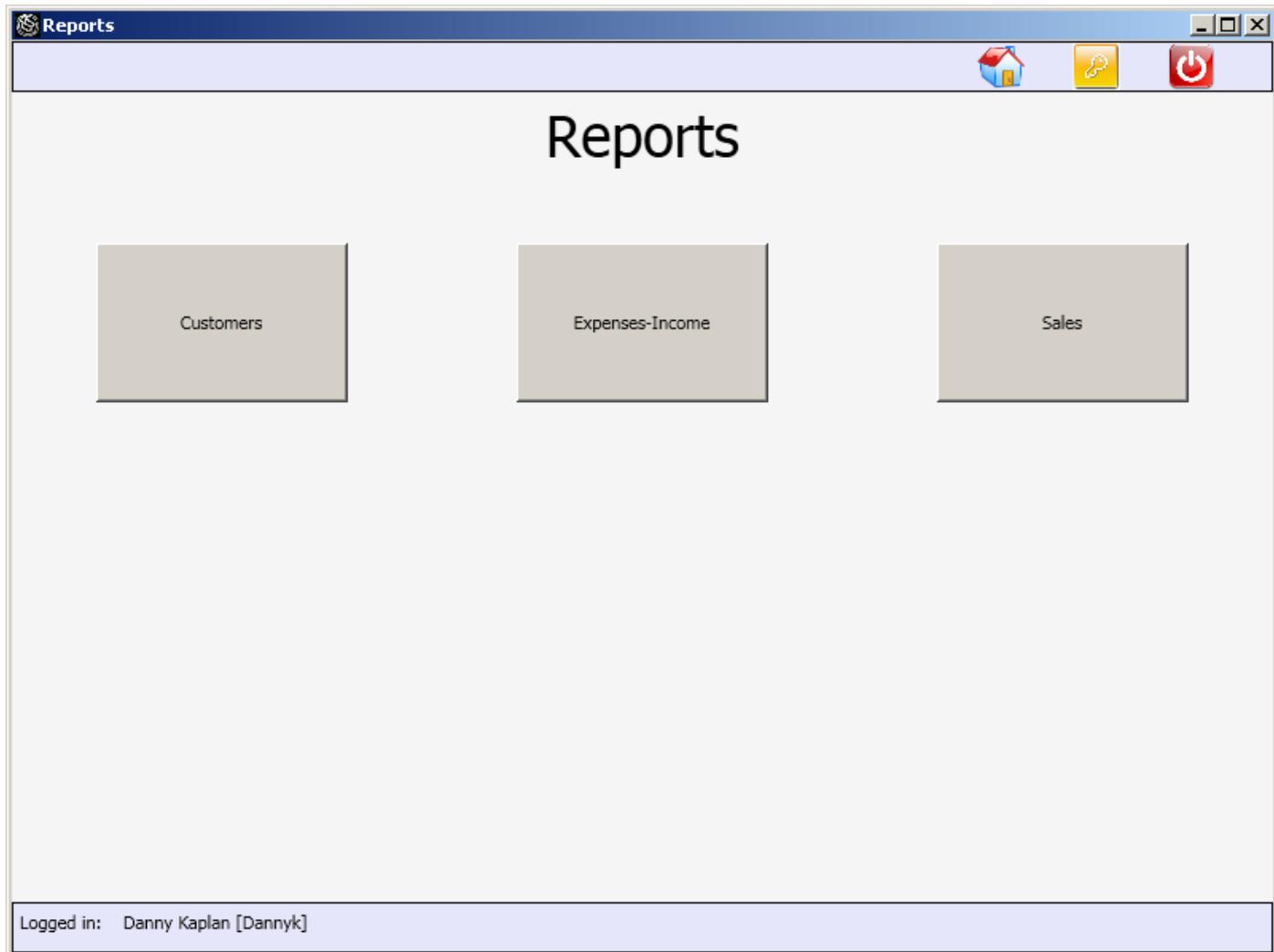
At the bottom left, a status bar displays 'Logged in: Shalom Gil [Shalomg]'

- לעובד הצוות מסר דומה לזה של ראש הצוות למעט האפשרות להקצות קריאת שירות לעובד צוות.

עריכת אירוע היסטוריה

- מסך זה מאפשר לצפות / לעורר את ההיסטוריה קריית השירות כפי שזו נכתבת במסך קריית השירות.
- במסך זה ניתן לעורר את הטקסט המופיע בתיאור האירוע וללחוץ על כפתור ה-Save לשימירת השינויים.
- לחיצה על כפתור ה-Cancel תבטל את השינויים שבוצעו.

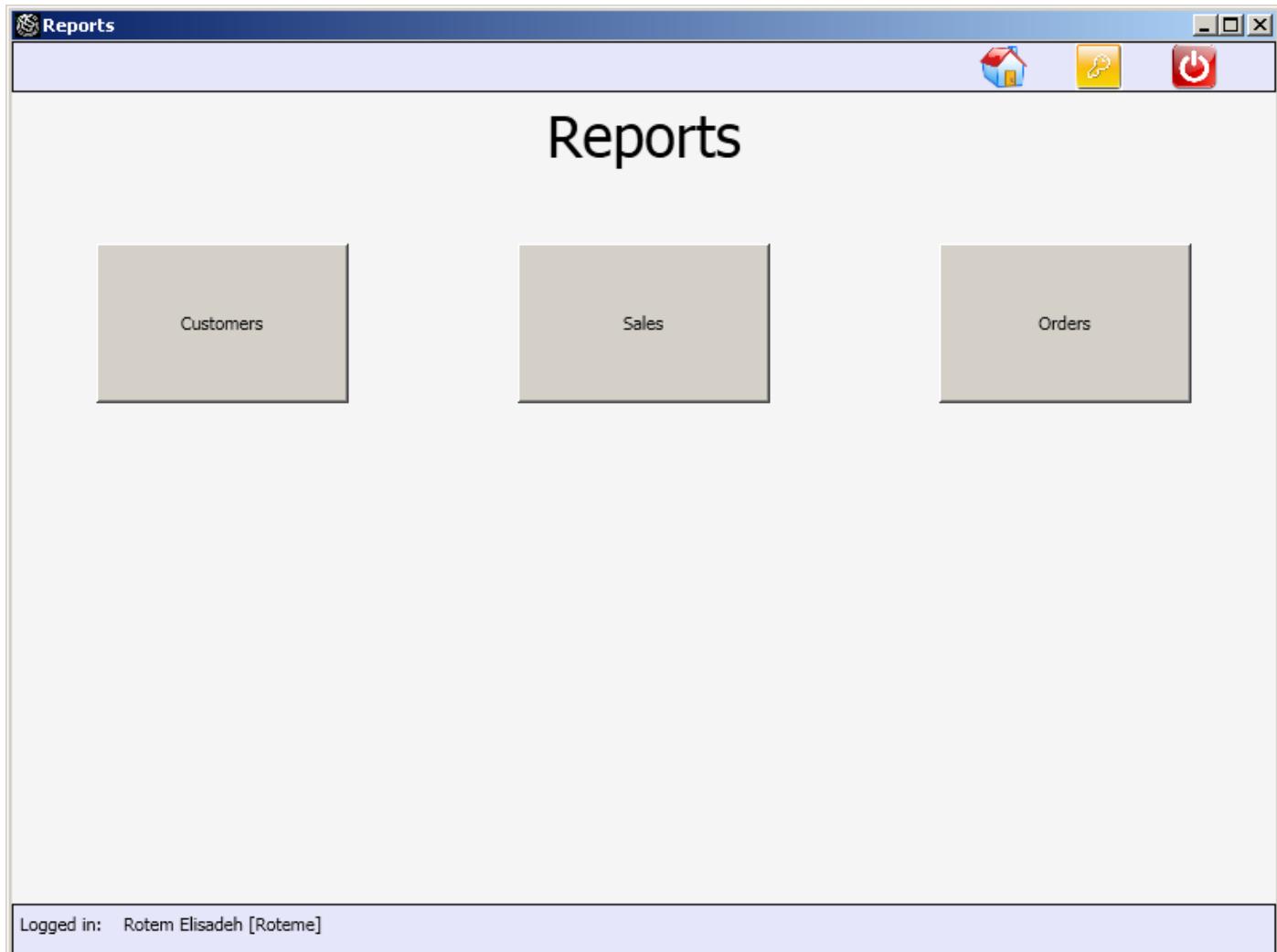
מסמך דוחות (מנהל העסק)



- מסר זה מאפשר למנהל העסק לבחור בהפקת הדוחות הבאים:
 - דוח לקוחות (Customer)
 - דוח הכנסות-הוצאות (Expenses-Income)
 - דוח מכירות (Sales)

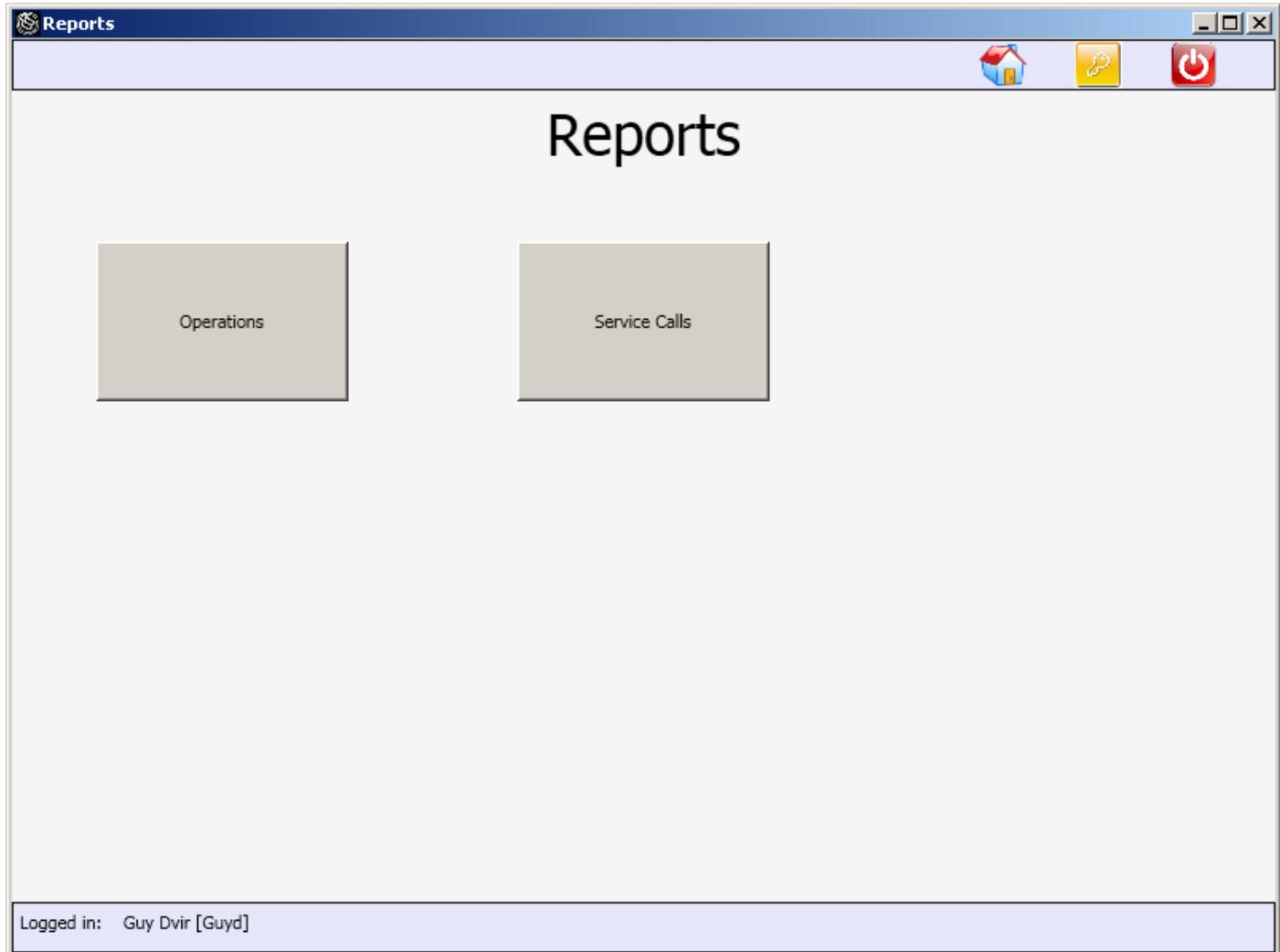
SmartBiz

מסך דוחות (מנהל המכירות)



- מסך זה מאפשר למנהל המכירות לבחור בהפקת הדוחות הבאים:
 - דוח לקוחות (Customer)
 - דוח מכירות (Sales)
 - דוח הזמנות ציוד (Orders)

מסך דו"חות (ראש הצוות)



- מסך זה מאפשר לראש הצוות לבחור בהפקת הדוחות הבאים:

- דוח פעילות (Operations)
- דוח קריאות שירות (Service Calls)

הערה: יתכנו ואריאציות נוספות של מסך דוחות כאשר משתמש בעל שתי רשאות שונות או יותר מתחבר אל המערכת.

SmartBiz

דו"ח לקוחות

Customers Report

The screenshot shows a Windows application window titled "Customers Report". At the top, there is a toolbar with icons for back, forward, home, search, and power. Below the toolbar is a "Filter By" section with two rows of date and time selection boxes. The first row has "From Date:" set to "30/07/2013 15" and "To Date:" set to "16/09/2013 15". The second row has "From Hour:" set to "15:00" and "To Hour:" set to "17:00". A "Produce" button is located below these filters. The main area is titled "Customers" and contains a grid table with columns: Id, First Name, Last Name, Address, Phone Home, Phone Mobile, and Phone Fa. Two records are visible:

Id	First Name	Last Name	Address	Phone Home	Phone Mobile	Phone Fa
011531746	Rotem	Elisadeh	Asdod, Irisim 1 st.	08-8691410	053-2255515	
032554422	Omer	Bar-On	Rehovot, Hertzel 45 st.	08-9455333	054-2631111	08-9455

At the bottom of the grid, it says "Total Records Found: 2".

Logged in: Rotem Elisadeh [Roteme]

- מסך זה מאפשר לצפות בכל לקוחות שהזנו למערכת לפי פילטרים שונים.
- מסך זה מאפשר להציג לקוחות שהזנו למערכת בטוויח תאריכים נתון ו/או בטוויח שעות נתון.
- דו"ח זה מספק את המידע כמה ואילו לקוחות הזנו למערכת בתוויר של תאריכים ו/או שעות.

The screenshot shows the 'Sales Report' window with the following details:

- Filter By:**
 - From Date: 16/08/2013 15
 - To Date: 16/09/2013 15
 - From Hour: 08:00
 - To Hour: 17:00
- Sales:**

Pid	Manufacturer	Type	Name	Amount	Is Service?	Is Product?
1006	Asus	Motherboard	P8H61M LE	2	<input type="checkbox"/>	<input checked="" type="checkbox"/>
1011	Kingston	Memory	KVR1333D3N9/2G	3	<input type="checkbox"/>	<input checked="" type="checkbox"/>
2003	N/A	Os Installation - Linux	N/A	1	<input checked="" type="checkbox"/>	<input type="checkbox"/>
- Total Records Found: 3

Logged in: Rotem Elisadeh [Roteme]

- מסך זה מאפשר לצפות בכל המכירות (ציוד / שירות) שבוצעו לפי פילטרים שונים.
- מסך זה מאפשר להציג מכירות שבוצעו בטווח תאריכים נתון ו/או בטווח שעות נתון.
- מסך זה מאנגד את סך הכמות שנמכרו מכל אחד מסוגי המוצרים / שירותים (כלומר, אם מוצר נתון מופיע בכמות 10, משמע הדבר שנמכרו 10 מוצרים מאותו סוג באותה תקופה זמן אך יתכן שהם שייכים למכירות שונות).

SmartBiz

דו"ח הכנסות-הוצאות

Expenses-Income Report

Filter By

From Date: 16/08/2013 15 To Date: 16/09/2013 15

From Hour: 08:00 To Hour: 17:00

Produce

Summary

Expenses:

Equipment: 5850 NIS

Income:

Equipment: 1038.4 NIS

Services: 377.6 NIS

Logged in: Danny Kaplan [Dannyk]

- מסך זה מאפשר הצגה של הכנסות והוצאות על פי פילטרים שונים.
- המסר מאפשר להציג את הכנסות והוצאות בטוויח תאריכים נתון /או בטוויח שעות נתון.
- ההוצאות הן אלו שהוצאו על הזמן ציוד באופן כללי באותה תקופה זמן (לאו דווקא בעבר פריטים שנרכשו באותה תקופה) ואין קשר ישיר ביניהן לבין הכנסות, שכן, אליהם התקבלו עבור רכישה של לקוחות ושירותים שנייתנו ללקוחות באותה תקופה (לדוגמא, אם לקוח רכש פריט ציוד ב-100 ש"ח, לא תציג כנגד זה הוצאה בסך הוצאות אותן פריט עלה לבית העסק).
- דו"ח זה יכול לעזור למנהל העסק לאמוד האם העסק רווחי לפי תקופות זמן שונות.

דו"ח הזמנות

Orders Report

The screenshot shows the 'Orders Report' window with the following interface elements:

- Filter By:**
 - Status: Pending
 - Dealer: Bob Dealer
 - From Date: 01/09/2013 To Date: 16/09/2013
- Buttons:** Back, Home, Help, Exit.
- Orders Table:**

Oid	Dealer	Date Ordered	Status	Total Cost
1000	Bob Dealer	04/09/2013	Pending	3800
1001	R.O.C	05/09/2013	Pending	850
- Statistics:** Total Records Found: 2, Total Cost: 4650 NIS.
- Log-in Information:** Logged in: Rotem Elisadeh [Roteme].

- מסך זה מאפשר לצפות בכל ההזמנות שבוצעו לפי פילטרים שונים.
- מסך זה מאפשר להציג ההזמנות שבוצעו בטוויח תאריכים נתון ו/או על פי ספק ו/או על פי סטטוסו.
- דו"ח זה מאפשר לראות את סך העלות הכוללת של כל ההזמנות שהתקבלו כתוצאה מהחיתוך.

SmartBiz

דו"ח קריאות שירות

The screenshot shows the 'Service Calls Report' window. At the top, there are filter options for 'From Date' (16/08/2013) and 'To Date' (16/09/2013), and for 'From Hour' (08:00) and 'To Hour' (17:00). A 'Produce' button is located below the filters. Below the filters, a table titled 'Service Calls' displays data for three workers: Dima Baskin, Rotem Elisadeh, and Tomer Lev. The columns show 'Worker Name', 'In Progress' (count), and 'Completed' (count). The table data is as follows:

Worker Name	In Progress	Completed
Dima Baskin	1	0
Rotem Elisadeh	1	1
Tomer Lev	0	0

Below the table, two statistics are shown: 'Total Service Calls In Progress: 2' and 'Total Service Calls Completed: 1'. At the bottom left, it says 'Logged in: Guy Dvir [Guyd]'

- מסך זה מאפשר לקבל מידע אודוט כמות קרייאות השירות בהם טיפול או מטפל כל אחד מעובדי הוצאות בטווח תאריכים נתון /או בטווח שעות נתון.
- מסך זה מציג עבור כל אחד מעובדי הוצאות מהו מספר קרייאות השירות שנפתחו באותו פרק זמן נבחר ושוויכו לכל עובד צוות.
- מסך זה מציג עבור כל אחד מעובדי הוצאות מהו מספר קרייאות השירות שנפתחו באותו תקופה זמן, והביאו אותו לסגירה (Completed).
- מסך זה מציג עבור כל אחד מעובדי הוצאות מהו מספר קרייאות השירות שנפתחו באותו תקופה זמן, והן עדין בטיפול (In Progress).

- מסך זה מציג נתוני אודוט קריאות שירות עבור סך העובדים:
 - סך קריאות השירות שהיו בסטאטוס Progress חס וונפתחו באותה תקופה זמן נבחרת.
 - סך קריאות השירות שהיו בסטאטוס Completed וונפתחו באותה תקופה זמן נבחרת.

SmartBiz

דו"ח פעילות

The screenshot shows the 'Operations Report' window with the following details:

- Filter By:**
 - From Date: 16/08/2013 15
 - To Date: 16/09/2013 15
 - From Hour: 08:00
 - To Hour: 17:00
 - Assigned To: [dropdown menu]
- Service Calls:**

Service Type	Unassigned	In Progress	Completed
Os Installation - Windows	0	0	3
Os Installation - Linux	0	3	0
Anti Virus Installation	0	2	0

Total Records Found: 3 Total Operations Found: 8
- Logged in:** Guy Dvir [Guyd]

- דו"ח זה מאפשר לצפות בפרטים אודוט סוג השירותים השיכים לקריאות שירות שנפתחו בתקופת זמן מסויימת.
- ניתן לבצע סינון ספציפי יותר ע"י שימוש בשדות הפילטרים.
- דו"ח זה מציג את כמות סוג השירותים שעבורם נפתחו קריאות שירות באותה תקופה זמן ולא שייכו לאף עובד (Unassigned).
- דו"ח זה מציג את כמות סוג השירותים שעבורם נפתחו קריאות שירות באותה תקופה זמן ונמצאות בסטטוס 'בティול' (In Progress).
-

- דו"ח זה מציג את כמות סוגי השירותים שעבורם נפתחו קריאות שירות באותה תקופה זמן ונמצאות בסטטוס 'טופל' (Completed).
- בדו"ח זה ניתן לצפות בכמות הכללת של הפעולות השונות.

File #0003243 belongs to Roei Daniel- do not distribute

SmartBiz

מסמך תכנון ועיצוב

גרסה 1.0

שמות הסטודנטים:

רותם אלישדה

עומר בר-און



File #0003243 belongs to Roei Daniel- do not distribute

תוכן עניינים

233.....	תיאור כללי של המערכת
233.....	הנחות עבודה בכתיבת הפרויקט
234.....	מוסכמת רישום
235.....	שכבות בסיס הנתונים (SmartBiz.Dal namespace)
237.....	טבלת משתמשים (Users)
237.....	טבלת הגדרות כלליות (Global Settings)
238.....	טבלת לקוחות (Customers)
238.....	טבלת היסטורית ללקוחות (Customer History)
239.....	טבלת מוצרים (Products)
239.....	טבלת מלאי (Inventory)
239.....	טבלת הנחות (Discounts)
240.....	טבלת חשבונית (Bills)
240.....	טבלת פירוט רכישה (Purchase Details)
241.....	טבלת שירותים (Services)
241.....	טבלת הזמנות ציוד (Orders)
241.....	טבלת פירוט הזמןה (Order Details)
242.....	טבלת חלוקת עבודה (Work Distribution)
242.....	טבלת היסטורית קריאות שירות (Service Calls History)
243.....	טבלת יצרנים עבור קובץ מודול (Module Product Manufacturers)
243.....	טבלת סוגים מוצרים עבור קובץ מודול (Module Product Types)
243.....	טבלת סוגים שירותים עבור קובץ מודול (Module Service Types)
244.....	שכבות הלוגיקה (SmartBiz.BI namespace)
245.....	מחלקות כלליות
245.....	Constants.cs
245.....	Convertor.cs
245.....	CustomerValidator.cs
245.....	DBController.cs
246.....	GlobalSettings.cs
247.....	GlobalValidator.cs
247.....	Module.cs
248.....	StringGenerator.cs

SmartBiz

248.....	UserAuthenticator.cs
248.....	UserValidator.cs
249.....	ממשקים ומחלקות אבסטרקטיות
249.....	Aggregate.cs
249.....	History.cs
249.....	IDatabaseObject.cs
250.....	Iterator.cs
250.....	Person.cs
251.....	Purchasable.cs
252.....	Strategy.cs
252.....	UserDefinition.cs
253.....	מחלקות עבר אובייקטים במערכת
253.....	Bill.cs
253.....	ComplexityCheckerA.cs
253.....	ComplexityCheckerB.cs
254.....	ComplexityCheckerC.cs
254.....	Concretemerator.cs
254.....	Context.cs
254.....	CurrentUser.cs
255.....	Customer.cs
255.....	CustomerHistory.cs
256.....	Order.cs
256.....	OrderItem.cs
256.....	Product.cs
257.....	Purchase.cs
257.....	Service.cs
257.....	ServiceCall.cs
258.....	ServiceCallHistory.cs
258.....	.ServiceRecord.cs
258.....	Setting.cs
259.....	ShoppingCart.cs
259.....	ShoppingCartItem.cs
259.....	User.cs
260.....	UserRecord.cs

SmartBiz

261.....	שכבות הציגה (SmartBiz.PI namespace)
261.....	UserControls
261.....	UpperToolbar.cs
262.....	LowerToolbar.cs
262.....	משקיים ומחלקות אבסטרקטיות
262.....	BasicForm.cs
262.....	NavigationalForm.cs
262.....	PopupForm.cs
263.....	MenuForm.cs
263.....	Observer.cs
263.....	Subject.cs
264.....	מחלקות כלליות
264.....	FilterOrders.cs
264.....	FilterProducts.cs
265.....	PageCache.cs
265.....	טפסים קונקרטיים במערכת
265.....	AddUpdateInventoryItemForm.xaml.cs
265.....	AddUpdateOrderForm.xaml.cs
265.....	AddUpdateUserForm.xaml.cs
266.....	AdminMenuForm.xaml.cs
266.....	BillForm.xaml.cs
266.....	ChangePasswordForm.xaml.cs
266.....	CustomerDetailsForm.xaml.cs
266.....	CustomerHistoryEventForm.xaml.cs
266.....	CustomersForm.xaml.cs
267.....	CustomersReportForm.xaml.cs
267.....	DiscountsForm.xaml.cs
267.....	ExpensesIncomeReportForm.xaml.cs
267.....	GlobalSettingsForm.xaml.cs
267.....	InventoryForm.xaml.cs
268.....	LoginForm.xaml.cs
268.....	MainMenuForm.xaml.cs
268.....	OperationsReportForm.xaml.cs
268.....	OrdersForm.xaml.cs

SmartBiz

269.....	OrdersReportForm.xaml.cs
269.....	PaymentForm.xaml.cs
269.....	PricingForm.xaml.cs
269.....	ReportsMainForm.xaml.cs
269.....	SalesForm.xaml.cs
270.....	SalesReportForm.xaml.cs
270.....	ServiceCallForm.xaml.cs
270.....	ServiceCallHistoryEventForm.xaml.cs
270.....	ServiceCallReportForm.xaml.cs
270.....	UserManagementForm.xaml.cs
271.....	WorkDistributionForm.xaml.cs
272.....	דף עי'זוב (Design Patterns)
272.....	Singleton
273.....	Iterator
274.....	Strategy
275.....	Observer
276.....	שינויים עתידיים אפשריים
279.....	נספח: דיאגרמות
279.....	דיאגרמת מסד הנתונים (Data Base Diagram)
280.....	דיאגרמות מחלקות (Class Diagrams)
280.....	דיאגרמת מחלקות עבור המחלקות הכלליות בשכבה ה-Bl
281.....	דיאגרמת מחלקות עבור האובייקטים בשכבה ה-Bl
282.....	דיאגרמת מחלקות עבור המחלקות הכלליות וה-UserControls בשכבה ה-1M
284.....	דיאגרמת מחלקות עבור הטפסים בשכבה ה-1M
286.....	דיאגרמות רצף (Sequence Diagrams)
286.....	דיאגרמת רצף עבור הפתק הצעת מחיר ללקוח
288.....	דיאגרמת רצף עבור ביצוע רכישה ע"י לקוח
289.....	דיאגרמת רצף עבור החלפת סיסמה של משתמש
290.....	דיאגרמת רצף עבור יצירת הזמנה חדשה במערכת
291.....	דיאגרמת רצף עבור הקצאה וטיפול בקריאה שירות.

תיאור כללי של המערכת

המערכת בנויה על סמך מודל 3-Tier, שבו המערכת מופרדת לשלוש שכבות:

- שכבה בסיס הנתונים (Database Layer)
- שכבה הלוגיקה (Business Logic Layer)
- שכבה הציגות (Presentation Layer)

כל שכבה מתחברת עם השכבה מעלה באמצעות ממשק קבוע מראש. ההפרדה לשכבות מאפשרת לנו להחליף בעtid את המשק בין שתי שכבות מבלי להחליף את המשק בין שתי השכבות האחרות, וכן, להחליף את המימוש של אחת השכבות מבלי להחליף את המימוש של השכבות האחרות, ובכך לחסוך בשכתוב קוד קיימ, ואפשר גמישות לשינויים.

הנחות עבודה בכתיבת הפרויקט

1. בכל רגע מחובר רק משתמש אחד במערכת.
2. לא ניתן למחוק מהמערכת משתמש/לקוחות בכך למנוע מצבים של חוסר-עקבות נתונים, שכן כדי למשוך "הסירה" שלהם, השדה הבוליאני `is_active` ישנה ערכו לערך האמת `false`.
3. שם פרטי ושם משפחה מורכבים ממילה אחת בלבד ללא רווחים.
4. שם מלא מורכב משם פרטי ושם משפחה עם רווח ביניהם.
5. שם משתמש מכיל אותיות בלבד.
6. במערכת זו, לכל מוצר ושרות קיימ מחיר בתווך 0 עד 1,000,000. מחיר זה לא כולל את המע"מ ואת הנחות התקיפות. חישוב ההנחה מתבצע לאחר הוספת המע"מ.
7. המערכת מאפשרת יצירה של עד 1,000 סוגי מוצרים שונים במלאי ועוד 1,000 סוגי שירותים שונים. המספר הסידורי של סוג המוצר נע בתווך 1,000 – 1,999. המספר הסידורי של סוג השירות נע בתווך 2,000 – 2,999.
8. כל המודולים במערכת שעובדים עם תאריכים מצפים לערכים בפורמט: yyyy/MM/dd. שכן נדרש להתאים את הגדרות התאריכים במערכת הפעלה בהתאם (מידע נוסף על כך נמצא במדריך למשתמש).
9. מערכת זו איננה תומכת בהתקשרות מול ספקים חיצוניים.

מוסכמות רישום

1. כל שם מחלוקת/שיטה/שדה שכוללת יותר מיליה אחת, נכתבת עם אות גדולה בתחילת כל מיליה נוספת.
 2. כל מחלוקת מחלוקת באות גדולה. למשקיקים נוספים אותן לפני שם המחלוקת.
 3. מחלוקת וממשקיקים שהינם מהווים משתתפים בדףו עיצוב כלשהו, אך אינם מייצגים אובייקט קונקרטי, יקבלו את שם המשתתף בדףו העיצוב.
 4. כל שדה (Member) פרטיה בחלוקת מחלוקת ב-'_'.
5. קבועים ו-enumים מכילים אותיות גדולות בלבד.
6. פרמטרים לשיטה או משתנים בתוך שיטה מחלוקת באות קטנה.
7. שיטות שהן ציבוריות מחלוקת באות גדולה, ושיטות פרטיות (או מוגנות) מחלוקת באות קטנה.
8. כל פקד בטופס מחלוקת בשם (עם אותה קטנה), ואח"כ בסוג הפקד כאשר כל שתי מילים מופרדות ע"י '_'

שכבה בסיס הנתונים (SmartBiz.Dal namespace)

שכבה זו אחראית על התקשרות מול מסד הנתונים, וביצוע הפעולות השוטפות מולו, כגון שטיפה, עדכון, הסרה, ושליפת נתונים. המערכת עשויה שימוש במסד הנתונים Microsoft SQL Server 2008 ומעלה.

שכבה זו מכילה את בסיס הנתונים smartbiz.mdf.

שכבה זו מכילה מחלקה בשם DBManager אשר מכילה רק שיטות סטטיות. מחלקה זו אחראית ליצור את ההתקשרות מול מסד הנתונים באמצעות אובייקט מסוג SqlConnection כאשר ה-connectionString מוגדר בקובץ App.config. בנוסף, מחלקה זו אחראית לשולף נתונים באמצעות Delete, Insert ו-Update.

מחלקה זו מכילה את השיטות הבאות:

InitSqlConnection – אתחול האובייקט SqlConnection ופתיחה ההתקשרות מול מסד הנתונים. CloseSqlConnection – ניתוק ההתקשרות מול מסד הנתונים.

QueryCmd – שיטה מקבלת כפרמטר מחרוזת המייצגת השאילתת מסוג Select. השיטה שולחת את השאילתת למסד הנתונים ומחזירה אובייקט מסוג DataTable המכיל את תוצאות השאילתת. NonQueryCmd – שיטה מקבלת מחרוזת המייצגת השאילתת מסוג Insert, Update או Delete.

השיטה שולחת את השאילתת למסד הנתונים ומחזירה ערך בוליאני המציין האם הפעולה בוצעה בהצלחה.

ScalarCmd – שיטה מקבלת מחרוזת המייצגת השאילתת מסוג Select הכללת שימוש באחת מהפונקציות הבאות: Max, Min, Sum, Count, Avg. השיטה מוחזירה מחרוזת המייצגת את הערך המוחזר מהשאילתת.

בדי לעשות שימוש בשכבה בסיס הנתונים, ראשית יש לאותחל את ההתקשרות מול בסיס הנתונים. דבר זה מתבצע בבניאי של הטופס הראשון המופיע במערכת (טופס ה-Login). הבניאי של הטופס מאותחל את ההתקשרות מול בסיס הנתונים באמצעות פניה למחלקה DBController שנמצאת בשכבה ה-Bl. מחלקה זו יוצרת את ההתקשרות ע"י קריאה לשיטה הסטטית InitSqlConnection במחלקה DBManager.

השכבה ממומשת באופן מודולארי כך שניתן יהיה בעtid להחליף את בסיס הנתונים לאחר, על ידי ביצוע שינויים מינימליים בלבד, וכל זאת מבלי לשנות את פרטי שימוש השכבות האחרות.

עקרונות מנחים לגבי שכבה בסיס הנתונים:

- הטבלאות מעוצבות בצורה BCNF (Boyce-Codd normal form).
- הטבלאות עוצבו כך שתיווצר כפילות מינימאלית ככל האפשר של נתונים הנשמרים בהן.
- בחרנו למש את התקשורת למסד הנתונים בשיטה המקושרת (Full Connected). קלומר החיבור בין היבום למסד הנתונים הינו קבוע. בחרנו למש את התקשורת למסד הנתונים בשיטה זו מכיוון שעדכניות המידע חשובה למשתמשי המערכת.
- החישון הוא שיטה זו יוצרת עומס על השרת שבו נמצא מסד הנתונים.
- היתרון של שיטה זו הינה:
 - תמיד עובדים עם המידע העדכני ביותר.
 - כמעט ואין צורך לטפל בקונפליקטים המתרחשים בין לקוחות שונים המתפלים באותו המידע.
 - הפעולות מול מקור המידע מתבצעות במהירות.

טבלה משתמשים (Users)

טבלה זו מכילה מידע אודות המשתמשים הקיימים במערכת. הטבלה מכילה את השדות הבאים:

הסבר	טיפוס	שדה
שם משתמש. מפתח ראשי	מחרוזת	username
שם פרטי	מחרוזת	first_name
שם משפחה	מחרוזת	last_name
הרשאות מנהל?	בוליאני	is_manager
הרשאות ראש צוות?	בוליאני	is_teamleader
הרשאות מנהל מכירות?	בוליאני	is_salesmanager
הרשאות עובד?	בוליאני	is_worker
סיסמה	מחרוזת	password
האם פעיל במערכת?	בוליאני	is_active
האם המשתמש חייב לבצע שינוי סיסמה בהתחברות הבאה למערכת?	בוליאני	is_must_change_password

טבלת הגדרות כלליות (Global Settings)

טבלה זו מכילה מידע אודות הגדרות כלליות הקיימות במערכת. כל שורה בטבלה היא הגדרה המורכבת ממספר וערך של המפתח. הטבלה מכילה את השדות הבאים:

הסבר	טיפוס	שדה
הגדרה במערכת. מפתח ראשי	מחרוזת	[key]
ערך עבור ההגדרה	מחרוזת	value

בטבלת ההגדרות הכלליות ישן 8 הגדרות קבועות, עבורן יש לקבוע ערכים. להלן ההגדרות:

הסבר	שדה
אחוז המע"מ	vat
סוג המטבע	coin_type
שם החברה	company_name
כתובת החברה	company_address
טלפון החברה	company_phone
שם הקובץ של המודול שנטען	module
שעת הפתיחה של העסק	open_at
שעת הסגירה של העסק	close_at

טבלת לקוחות (Customers)

טבלה זו מכילה מידע אודוט ללקוחות בית העסק. הטבלה מכילה את השדות הבאים:

הסבר	טיפוס	שדה
מזהה לקוח. מפתח ראשי.	מחרוזת	<u>cust_id</u>
תאריך לידה	תאריך	<u>birth_date</u>
שם פרטי	מחרוזת	<u>first_name</u>
שם משפחה	מחרוזת	<u>last_name</u>
עיר	מחרוזת	<u>city</u>
רחוב	מחרוזת	<u>street</u>
קוד מיקוד	מספר שלם	<u>zip_code</u>
טלפון בית	מחרוזת	<u>phone_home</u>
טלפון נייד	מחרוזת	<u>phone_mobile</u>
fax	מחרוזת	<u>phone_fax</u>
תאריך יצירה	תאריך	<u>date_created</u>
זמן יצירה	זמן (מומר לטיפוס bigint)	<u>time_created</u>
האם הלקוח פעיל במערכת?	בוליאני	<u>is_active</u>

טבלת היסטוריה ללקוחות (Customer History)

טבלה זו מכילה מידע אודוט היסטורית הלקוחות. הטבלה מכילה את השדות הבאים:

הסבר	טיפוס	שדה
מזהה רשומה. מפתח ראשי. מספר אוטומטי החל מ-1000.	מספר שלם	<u>hid</u>
מזהה לקוח. מפתח זר.	מחרוזת	<u>cust_id</u>
סוג הרשמה	מחרוזת	<u>type</u>
תאריך יצירה	תאריך	<u>date_created</u>
זמן יצירה	זמן (מומר לטיפוס bigint)	<u>time_created</u>
תיאור הרשמה	טקסט	<u>description</u>

טבלת מוצרים (Products)

טבלה זו מכילה מידע אודות סוגי המוצרים השונים הקיימים במערכת. הטבלה מכילה את השדות הבאים:

הסבר	טיפוס	שדה
מזהה מוצר. מפתח ראשי.	מספר שלם	<u>pid</u>
מספר אוטומטי החל מ-1000.		
סוג המוצר	מחרוזת	type
יצרן	מחרוזת	manufacturer
שם המוצר	מחרוזת	name
מחיר המוצר. שדה זה יכול להכיל את הערך null.	מספר ממשי	price

טבלת מלאי (Inventory)

טבלה זו מכילה מידע אודות כמותי המוצרים השונים הקיימים במלאי העסק. הטבלה מכילה את השדות הבאים:

הסבר	טיפוס	שדה
מזהה מוצר. מפתח ראשי.	מספר שלם	<u>pid</u>
מפתח זר.		
כמות המוצר	מספר שלם	amount

טבלת הנחות (Discounts)

טבלה זו מכילה מידע>About הנחות השונות בבית העסק. הטבלה מכילה את השדות הבאים:

הסבר	טיפוס	שדה
מזהה מוצר. מפתח ראשי.	מספר שלם	<u>pid</u>
מפתח זר.		
מבצע 1+1?	בוליאני	one_plus_one
אחוז הנחה על המוצר	מספר ממשי	discount

טבלה חשבונית (Bills)

טבלה זו מכילה מידע אודות כל הרכישות שבוצעו בבית העסק. הטבלה מכילה את השדות הבאים:

הסבר	טיפוס	שדה
מספר חשבונית. מפתח ראשי.	מספר שלם	<u>receipt</u>
מספר אוטומטי החל מ-1000.		
מזהה לקוח. מפתח זר.	מחרוזת	<u>cust_id</u>
תאריך יצירה	תאריך	<u>date_created</u>
זמן יצירה	זמן (מומר לטיפוס bigint)	<u>time_created</u>
סה"כ עלות הציוד	מספר ממשי	<u>t_price_equipment</u>
סה"כ עלות השירותים	מספר ממשי	<u>t_price_services</u>

הערה: השדה id cust אפשר בגרסת עתידית ל קישור בין חשבונית ללקוח.

טבלה פירוט רכישה (Purchase Details)

טבלה זו מכילה מידע כמותית שנרכשו מכל מוצר בכל עסקה שבוצעה.

טבלה זו מקשרת בין הטבלאות "חשבונית" ל"מורים" בקשר מסווג רבים-רבים.

טבלה מכילה את השדות הבאים:

הסבר	טיפוס	שדה
מספר חשבונית. מפתח ראשי.	מספר שלם	<u>receipt</u>
מספר זר. מפתח ראשי.	מספר שלם	<u>pid</u>
מזהה מוצר. מפתח זר.	מספר שלם	
כמות המוצרים	מספר שלם	<u>amount</u>

טבלת שירותים (Services)

טבלה זו מכילה מידע אודות השירותים השונים שמוצעו בית העסק.
הטבלה מכילה את השדות הבאים:

הסבר	טיפוס	שדה
מזהה שירות. מפתח ראשי. מספר אוטומטי החל מ-2000.	מספר שלם	<u>pid</u>
סוג השירות	מחרוזת	type
מחיר השירות. שדה זה יכול להכיל את הערך null.	מספר ממשי	price

טבלת הזמנות ציוד (Orders)

טבלה זו מכילה מידע אודות הזמנות הציוד של בית העסק. הטבלה מכילה את השדות הבאים:

הסבר	טיפוס	שדה
מזהה הזמנה. מפתח ראשי.	מספר שלם	<u>oid</u>
שם הספק	מחרוזת	dealer
סטאטוס ביצוע	מחרוזת	status
העלות הכוללת של ההזמנה	מספר ממשי	cost
תאריך יצירת הזמנה	תאריך	date_created
שעת יצירת הזמנה	זמן (מומר לטיפוס bigint)	time_created

טבלת פירוט הזמנה (Order Details)

טבלה זו מכילה מידע אודות פרטי כל אחת מההזמנות הציוד.

טבלה זו מקשרת בין הטעיות "הזמנות-ציוד" ל"מוצרים" בקשר מסווג רבים-רבים.

הטבלה מכילה את השדות הבאים:

הסבר	טיפוס	שדה
מזהה הזמנה. מפתח ראשי. מספר זר.	מספר שלם	<u>oid</u>
מזהה מוצר. מפתח ראשי. מספר זר.	מספר שלם	<u>pid</u>
הכמות של המוצר בהזמנה	מספר שלם	amount
המחיר הכוללי של המוצר בהזמנה	מספר שלם	cost

טבלת חלוקת עבודה (Work Distribution)

טבלה זו מכילה מידע אודוט קריאות השירות הקיימות במערכת.
הטבלה מכילה את השדות הבאים:

הסבר	טיפוס	שדה
מזהה קריאה. מפתח ראשי. מספר אוטומטי החל מ-1000.	מספר שלם	<u>call_id</u>
מזהה העובד המבצע (username). מפתח זר.	מחרוזת	assigned
תיאור השירות	מחרוזת	service
כמות השירות	מספר שלם	amount
תאריך פתיחת הקריאה	תאריך	date_created
זמן פתיחת הקריאה	זמן (מספר לטיפוס bigint)	time_created
סטאטוס הביצוע	מחרוזת	status
תיאור קריאת השירות	מחרוזת	description
מזהה לקוח שעבורו נפתחה הקריאה. מפתח זר.	מחרוזת	cust_id

טבלת היסטוריה קריאות שירות (Service Calls History)

טבלה זו מכילה מידע אודוט היסטורית קריאות השירות. הטבלה מכילה את השדות הבאים:

הסבר	טיפוס	שדה
מזהה רשומה. מפתח ראשי. מספר אוטומטי החל מ-1000.	מספר שלם	<u>hid</u>
מזהה קריאה. מפתח זר.	מספר שלם	<u>call_id</u>
הערות	טקסט	comments
תאריך פתיחת הרשימה	תאריך	date_created
זמן פתיחת הרשימה	זמן (מספר לטיפוס bigint)	time_created
סוג הרשימה	מחרוזת	type

טבלת יצרנים עבור קובץ מודול (Module Product Manufacturers)

טבלה זו מכילה את רשימת היצרנים שהתווסףו למערכת לאחר טעינת קובץ המודול.

הסבר	טיפוס	שדה
שם היצרן.	מחוזת	<u>manufacturer</u>

טבלת סוגי מוצרים עבור קובץ מודול (Module Product Types)

טבלה זו מכילה את רשימת סוגי המוצרים שהתווסףו למערכת לאחר טעינת קובץ המודול.

הסבר	טיפוס	שדה
סוג המוצר.	מחוזת	<u>type</u>

טבלת סוגי שירותים עבור קובץ מודול (Module Service Types)

טבלה זו מכילה את רשימת סוגי השירותים שהשירותים שהתווסףו למערכת לאחר טעינת קובץ המודול.

הסבר	טיפוס	שדה
סוג השירות.	מחוזת	<u>type</u>

שכבה הлогיקה (SmartBiz.BI namespace)

שכבה זו מקשרת בין שכבת בסיס הנתונים לשכבת הציגה, והיא מהוות הפשטה של האובייקטים הקיימים במערכת. כל אובייקט מיוצג באמצעות מחלוקת משלה. מחלוקת אלו יתוארו בהמשך.

עקרונות מנהים בשכבה הולוגיקה:

- בעת פניה לשכבת בסיס הנתונים (Dal), שכבת הולוגיקה תספק את בדיקות אימות הנתונים (Validation) וכן, בדיקת חריגות (Exceptions) עבור הפלטים שמתקבלים ממנה.
- עקרון הרכimos (Encapsulation) בא לידי ביטוי בשכבה הולוגיקה, בכך שאנו מסתירים את המידע הנשמר במחלקות וחושפים אותו כלפי חוץ באמצעות תכונות (Properties) בלבד. מעבר לתרונות הידועים של עקרון הרכimos, הקוד הופך להיות יותר גמיש, מהבחןיה שבמידה וירצוי להוסיף הגבלות מסוימות על ערכי המידע או בדיקות נוספות יהיה להוסיף זאת לכל מידע שנשמר וזאת מבלתי צורך לשכתב קוד קיימ במקומות אחרים.
- עקרון ה-SRP (Single Responsibility Principle) יבוא לידי ביטוי בשכבה הולוגיקה בכך שכל מחלוקת תקבע אחריות אחת בלבד.
- עקרון ה-LSP (Liskov Substitution Principle) יבוא לידי ביטוי בשכבה הולוגיקה בכך שכאשר השימוש יאפשר זאת, אז פונקציות העוסקות שימוש במצביים של Base Class יהיו מסוגלות לעשות שימוש ב-Derived Classes מבלתי לדעת על כך. (למשל בביצוע fill של רשות מוצרים או שירותים ל-Data Grid, הפונקציה תקבע רשות אובייקטים מטיפוס Purchasable מבלתי לדעת בפועל מהו ה-Derived Class).
- עקרון ה-Open-Closed Principle יבוא לידי ביטוי בכך שהקוד יבנה בצורה מודולרית וgemäßישה תור תמייה בהרחבות עתידיות, כך שהמודולים יהיו סגורים לשינוי, אך ניתנים להרחבה.
- שימוש בדפוסי עיצוב (Design Patterns) רלוונטיים כדי שהעיצוב של שכבה זו יהיה גמיש לשינויים ויעמוד ביעדי הנדסת תוכנה נוספים.

עבור עתה על המחלוקות השונות.

מחלקות כלליות

Constants.cs

מחלקה זו מכילה אוסף של קבועים אשר רלוונטיים לכל המערכת. במחלקה זו נגידר קבועים אשר השימוש בהם יעשה מכמה מחלקות שונות במערכת. דוגמה לקבועים אשר צריך לעשות בהם שימוש בכלל המערכת הינה: שם המשתמש והסיסמה של מנהל המערכת.

Convertor.cs

מחלקה זו מכילה אוסף של שיטות סטטיות הממירות בין ערכים וטיפוסים. כל שיטה במחלקה זו מקבלת פרמטר אחד ומחזירה את הערך לאחר ביצוע הממרה. דוגמה לשיטה השויכת למחלקה זו היא שיטה אשר מקבלת תאריך מטיפוס DateTime ומחזירה מחרוזת המייצגת תאריך בפורמט yyyy/MM/dd.

מחלקה זו תכיל את השיטה הבאה:

```
public static string ConvertToDBDate(DateTime date)
```

CustomerValidator.cs

מחלקה זו מכילה אוסף של שיטות סטטיות המבצעות אימומות עבור תוכנות השיויכת לישות 'לקוח'. כל שיטה במחלקה זו מקבלת מחרוזת המכילה את ערך התוכנה ומחזירה ערך בוליאני הקובע האם המחרוזת מכילה ערך חוקי עבור תוכנה זו.

מחלקה זו תכיל את השיטות הבאות:

```
public static bool ValidateCustomerID(string cid)
```

```
public static bool ValidateCustomerZipCode(string zipCode)
```

כמו כן, למחלקה זו ניתן להוסיף שיטות נוספות לבדוק את התכונות השונות של הישות 'לקוח'.

DBController.cs

מחלקה זו מכילה שתי שיטות סטטיות המבצעות חיבור וניתוק התקשרות למסד הנתונים. שיטות אלה פונות אל ה-DBManager בשכבה ה-Dal כדי לATCH ו לנתק את ההתקשרות בהתאם.

מחלקה זו תכיל את השיטות הבאות:

```
public static void InitDB()
```

```
public static void CloseDB()
```



GlobalSettings.cs

מחלקה זו מכילה שיטות סטטיות המחזירות ערכים של הגדרות כלליות במערכת. כל שיטה במחלקה זו מחזירה ערך של הגדרה כללית כלשהי. לכל אחת מהשיטות מוגדר קבוע המכיל את ערך ברירת המחדל במקרה שהגדרה זו לא נמצאה בבסיס הנתונים.

עבור כל הגדרה כללית השמורה במערכת, קיימת שיטה המחזירה את הערך השמור בבסיס הנתונים של אותה הגדרה מתאימה.

מחלקה זו תכיל את השיטות הבאות:

```
public static string GetCoinType()  
  
public static string GetCompanyName()  
  
public static string GetCompanyAddress()  
  
public static string GetCompanyPhone()  
  
public static double getVatRate()  
  
public static string GetOpeningHour()  
  
public static string GetClosingHour()
```

GlobalValidator.cs

מחלקה זו מכילה אוסף של שיטות סטטיות המבצעות אימונות עבור תכונות כלשהן במערכת. כל שיטה במחלקה זו מקבלת מחרוזת המכילה את ערך התכונה ומחזירה ערך בוליאני הקובע האם המחרוזת מכילה ערך חוקי עבור תכונה זו. מחלקה זו תכילה שיטות לאימונות תכונות שונות אשר ערכיהן נקבעים ע"י קלט מהמשתמש. דוגמאות לתכונות אשר יש לאמת לפני ששמרים את ערכיהן בסיסי הנטונים הם: כמות (מספר שלם אי-שלילי), מחיר (מספר שלם בתווך 0-1,000,000), מספר תשלום (מספר שלם בתווך 1-24) ועוד ...

מחלקה זו תכילה את השיטות הבאות:

```
public static bool ValidateVatRate(string vat)
public static bool ValidateCredit(string credit)
public static bool ValidatePayments(string payments)
public static bool ValidateDiscount(string discount)
public static bool ValidateCompanyHours(string openAt, string closeAt)
public static bool ValidateAmountAtInsert(string amount)
public static bool ValidateAmountAtCreate(string amount)
public static bool ValidateAmountAtSearch(string amount)
public static bool ValidateCost(string cost)
```

Module.cs

מחלקה זו מכילה אוסף של שיטות סטטיות המבצעות ניתוח (Parsing) של קובץ המודול. מחלקה זו מכילה את השיטה הראשית:

```
public static bool ParseModule(string filePath)
```

המקבלת נתיב מלא הכלל את שם הקובץ, מבצעת עליו את פעולה הניתוח, מוסיף את הנטונים המתאימים לטבלאות המודול בסיסי הנטונים ומחזירה לבסוף ערך בוליאני הקובע האם הפעולה הצליחה.

מחלקה זו כוללת את כל השיטות הפרטיות ואת הקבועים הדרושים לצורך פעולה הניתוח (Parsing).



StringGenerator.cs

מחלקה זו מכילה אוסף של שיטות סטטיות המקבלות פרמטר אחד או יותר, כל שיטה מייצרת מחרוזת טקסט המכילה בתוכה נתונים הנגזרים מפרמטרים אלו. מחלקה זו נועדה לתמוך במנגנון ייצור האירועים האוטומטיים (עבור ליקוחות וקריאות שירות), כל שיטה מחזירה את תוכן האירוע שנוצר באופן אוטומטי כאשר התוכן נבנה בהתאם לפרמטרים שהשיטה מקבלת.

מחלקה זו תכיל את השיטות הבאות:

```
public static string GenerateString_ChangeStatus(string status)

public static string GenerateString_CustomerPurchased(ShoppingCart shoppingCart)

public static string GenerateString_PriceOffer(ShoppingCart shoppingCart)

public static string GenerateString_CustomerPayment(double sum, Bill bill)

public static string GenerateString_NewServiceCall(ServiceCall newCall)

public static string GenerateString_CloseServiceCall(ServiceCall newCall)
```

UserAuthenticator.cs

מחלקה זו מכילה שיטות סטטיות המאפשרות לאמת את פרטי ההתחברות של משתמש אל המערכת, באמצעות ייצור קשר עם מסד הנתונים והחזרת תשובה. היא מכילה את השיטה הבאה:

```
public static bool AuthenticateUser(string username, string password)
```

UserValidator.cs

מחלקה זו מכילה שיטות סטטיות המאפשרות לוודא פרטי הנוגעים למשתמשים במערכת. היא מאפשר לוודא שהסיסמה עומדת בתנאי חזק מסויימים (מספר מינימלי של תווים ומורכבות מסוימת). היא מאפשר לוודא שם המשתמש שורצים להגדיר עומד בkonvensioot שם מסוימת (אותיות בלבד!).

היא מכילה את השיטות הבאות:

```
public static bool ValidatePassword(string password)

public static bool ValidateUsername(string username)
```

ממשקים ומחלקות אבסטרקטיות

Aggregate.cs

מחלקה אבסטרקטית המכילה שיטה אבסטרקטית ייחידה:

```
public abstract Iterator CreateIterator();
```

מחלקה זו הינה חלק מדפו העיצוב Iterator אשר עליו נರחיב בהמשך.

History.cs

מחלקה אבסטרקטית המייצגת אירוע היסטוריה במערכת. היא בעלת התכונות הבאות:

```
public int Hid { get; set; }
```

```
public DateTime DateCreated { get; set; }
```

```
public Int64 TimeCreated { get; set; }
```

```
public string Data { get; set; }
```

```
public string Type { get; set; }
```

IDatabaseObject.cs

מחלקה זו מייצגת משק אחיד לכל אובייקט במערכת אשר משמש גם כאובייקט שנשמר בבסיס הנתונים. משק זה מאפשר את הכנסת האובייקט לבסיס הנתונים, את עדכונו או את מחיקתו. בנוסף על כן, משק זה מגדיר שיטה נוספת המחזירה ערך בוליאני הקובע האם אובייקט זה קיים בסיס הנתונים. יתר השיטותמחזירות ערך אמת 'true' אם הפעולה הצליחה, ו-'false' אם הפעולה נכשלה.

משק זה מגדיר את השיטות הבאות:

```
bool InsertToDB();
```

```
bool RemoveFromDB();
```

```
bool UpdateDB();
```

```
bool IsExistInDB();
```

SmartBiz

Iterator.cs

מחלקה אבסטרקטית המכילה ארבע שיטות אבסטרקטיות:

```
public abstract void First();  
  
public abstract ShoppingCartItem Next();  
  
public abstract bool IsDone();  
  
public abstract ShoppingCartItem CurrentItem();
```

מחלקה זו הינה חלק מדפו העיצוב Iterator אשר עליו נרחב בהמשך.

Person.cs

מחלקה אבסטרקטית המייצגת אדם. מחלקה זו מגדרה את התכונות והשיטות המשותפות למשתמשים ולקוחות. היא בעלת התכונות הבאות:

```
public string FirstName { get; set; }  
  
public string LastName { get; set; }  
  
public bool IsActive { get; set; }
```

מחלקה אבסטרקטית המייצגת פריט אשר ניתן לרכוש. מחלקה זו מגדירה את התכונות והשיטות המשותפות למוצרים ו שירותים. היא בעלת התכונות הבאות:

```
public int Pid { get; set; }

public string Type { get; set; }

public double Price { get; set; }

public bool OnePlusOne { get; set; }

public double Discount { get; set; }
```

מחלקה זו מכילה את השיטות הבאות:

```
public double GetVatValue()

public double GetPriceIncVat()

public double GetPriceIncVat(double vat)

public static double GetPriceIncVat(double price, double vat)

public double GetPriceIncVatAndDiscount()

public double GetPrice(double percent)

public double GetTotalPrice(int amount)
```



Strategy.cs

מחלקה אבסטרקטית המכילה שיטה אחת אבסטרקטית:

```
public abstract bool checkComplexity(string password);
```

שיטה זו מקבלת מחרוזת המייצגת סיסמה, השיטה מחזירה ערך בוליאני 'true' אם הסיסמא מורכבת מספיק על פי האלגוריתם המממש את השיטה האבסטרקטית. השיטה מחזירה ערך בוליאני 'false' אם הסיסמא אינה עומדת בתנאי המורכבות על פי אותו האלגוריתם.

מחלקה זו הינה חלק מדפו העיצוב Strategy אשר עליו נרחב בהמשך.

UserDefinition.cs

מחלקה אבסטרקטית המייצגת הגדרה של משתמש. מחלקה זו נועדה כדי להגדיר את כל התכונות והשיטות אשר משותפות לאובייקט 'משתמש' ולאובייקט 'המשתמש הנוכחי'. היא בעלת התכונות הבאות:

```
public string UserName { get; set; }

public string Password { get; set; }

public bool IsManager { get; set; }

public bool IsTeamleader { get; set; }

public bool IsSalesmanager { get; set; }

public bool IsWorker { get; set; }

public bool IsMustChangePassword { get; set; }
```

בנוסף, מחלקה זו מimplements את כל השיטות של הממשק `IDatabaseObject` וירושת מהמחלקה `Person`.

מחלקה עבורה אובייקטים במערכת**Bill.cs**

מחלקה המממשת את הממשק `IDatabaseObject`, מייצגת חשבונית ללקוח במערכת. היא בעלת התכונות הבאות:

```
public int Receipt { get; set; }

public string CustId { get; set; }

public DateTime DateCreated { get; set; }

public Int64 TimeCreated { get; set; }

public double TotalPriceEquipment { get; set; }

public double TotalPriceServices { get; set; }
```

ComplexityCheckerA.cs

מחלקה היורשת מ-`Strategy`, וממשת את השיטה האבסטרקטית:

```
checkComplexity(string password)
```

שיטה זו מימושת את אלגוריתם בדיקת המורכבות באופן הבא: השיטה בודקת האם המחרוזת מכילה לפחות 2 (מוגדר קבוע - `MINIMUM_AMOUNT`) תוויים אשר כל אחד מהם שייר לקובוצה אחרת, כאשר הקבוצות הן: אותיות קטנות, אותיות גדולות, ספרות וסימנים אחרים.

מחלקה זו הינה חלק מדף העיצוב `Strategy` אשר עליו נרחב בהמשך.

ComplexityCheckerB.cs

מחלקה היורשת מ-`Strategy`, וממשת את השיטה האבסטרקטית:

```
checkComplexity(string password)
```

שיטה זו מימושת את אלגוריתם בדיקת המורכבות באופן הבא: השיטה בודקת האם המחרוזת לא מכילה שני תוויים זהים עוקבים. כמובן, על פי אלגוריתם זה, סיסמה מורכבת הינה סיסמה אשר לא מכילה שני תוויים זהים עוקבים.

מחלקה זו הינה חלק מדף העיצוב `Strategy` אשר עליו נרחב בהמשך.

ComplexityChecker.cs

מחלקה הירושת מ-Strategy, ומימוש את השיטה האבסטרקטית:

```
checkComplexity(string password)
```

שיטה זו מימוש את אלגוריתם בדיקת המורכבות באופן הבא: השיטה בודקת האם המחרוזת לא מכילה שני תווים עוקבים שהicityים לאותה קבוצה (אותיות קטנות, אותיות גדולות, ספרות וסימנים אחרים). כמו כן, על פי אלגוריתם זה, סיסמה מורכבת הינה סיסמה אשר לא מכילה שני תווים עוקבים שהicityים לאותה קבוצה.

מחלקה זו הינה חלק מדף העיצוב Strategy אשר עליו נרחב בהמשך.

ConcreteIterator.cs

מחלקה הירושת מהמחלקה האבסטרקטית Iterator, מייצגת Iterator קונקרטי המתאים לאובייקט ShoppingCart.

מחלקה זו הינה חלק מדף העיצוב Iterator אשר עליו נרחב בהמשך.

Context.cs

מחלקה המחזיקה הפניה (Reference) לאובייקט מסווג Strategy.

מחלקה זו מכילה את השיטה:

```
public bool ContextInterface(string password)
```

מחלקה זו הינה חלק מדף העיצוב Strategy אשר עליו נרחב בהמשך.

CurrentUser.cs

מחלקה הירושת מהמחלקה האבסטרקטית UserDefinition, מייצגת את המשתמש הנוכחי שמחוברCurrently.

מחלקה זו הינה חלק מדף העיצוב Singleton אשר עליו נרחב בהמשך.

Customer.cs

מחלקה הירושת מהמחלקה האבסטרקטית Person ומימוש את הממשק .IDatabaseObject. מחלקה זו מייצגת לקוחות רשום במערכת. היא בעלת התכונות הבאות:

```
public string Cid { get; set; }

public string City { get; set; }

public string Street { get; set; }

public int ZipCode { get; set; }

public string PhoneHome { get; set; }

public string PhoneMobile { get; set; }

public string PhoneFax { get; set; }

public DateTime DateCreated { get; set; }

public Int64 TimeCreated { get; set; }

public DateTime BirthDate { get; set; }
```

CustomerHistory.cs

מחלקה הירושת מהמחלקה האבסטרקטית History ומימוש את הממשק .IDatabaseObject. מחלקה זו מייצגת רשומות היסטוריה עבור לקוח במערכת. בנוסף על התכונות שהמחלקה ירושת מהמחלקה History, היא בעלת התכונה הבאה:

```
public string CustomerId { get; set; }
```

Order.cs

מחלקה זו מimplements את הממשק `IDatabaseObject`. מחלקה זו מייצגת הזמנה ציוד מספק חיצוני. היא בעלת התכונות הבאות:

```
public int Oid { get; set; }

public string Dealer { get; set; }

public string Status { get; set; }

public double Cost { get; set; }

public DateTime DateCreated { get; set; }

public Int64 TimeCreated { get; set; }
```

OrderItem.cs

מחלקה זו מimplements את הממשק `IDatabaseObject`. מחלקה זו מייצגת מוצר כלשהו בהזמנה ציוד מספק חיצוני. היא בעלת התכונות הבאות:

```
public int Oid { get; set; }

public int Pid { get; set; }

public int Amount { get; set; }

public double Cost { get; set; }
```

Product.cs

מחלקה הירושת מהמחלקה האבסטרקטית `Purchasable` ומimplements את הממשק `Purchasable` מחלקה זו מייצגת מוצר במערכת. בנוסף על התכונות שהמחלקה ירושת מהמחלקה `Purchasable` היא בעלת התכונות הבאות:

```
public string Manufacturer { get; set; }

public string Name { get; set; }

public int Amount { get; set; }
```

Purchase.cs

מחלקה המimplements את הממשק `IDatabaseObject`. מחלקה זו מייצגת רכישה של מוצר או שירות מסוים. מחלקה זו מכילה תכונות המקשרות בין המוצר או השירות לבין החשבונית השיכת אותה רכישה. אובייקט זה מכיל תכונה נוספת המפרטת את הכמות של המוצר או השירות הנרכש. היא בעלת התכונות הבאות:

```
public int Receipt { get; set; }

public int Pid { get; set; }

public int Amount { get; set; }
```

Service.cs

מחלקה הירושת מהמחלקה האבסטרקטית `Purchasable` ומimplements את הממשק `IDatabaseObject`. מחלקה זו מייצגת שירות במערכת. למחלקה זו אין תכונות נוספות בנוסף על התכונות שהמחלקה ירושת מהמחלקה `Purchasable`.

ServiceCall.cs

מחלקה המimplements את הממשק `IDatabaseObject`. מחלקה זו מייצגת קריית שירות במערכת. קריית שירות במערכת נפתחת באופן אוטומטי לאחר שלקוח מסוים ביצע רכישה של שירות כלשהו. כל קריית שירות מקושרת ללקוח אחד בלבד ובכל רגע נתון הקרייה משוויכת לעובד אחד לכל היותר. היא בעלת התכונות הבאות:

```
public int Cid { get; set; }

public string Assigned { get; set; }

public string ServiceType { get; set; }

public int Amount { get; set; }

public DateTime DateCreated { get; set; }

public Int64 TimeCreated { get; set; }

public string Status { get; set; }

public string Description { get; set; }

public string CustomerId { get; set; }
```

ServiceCallHistory.cs

מחלקה היורשת מהמחלקה האבסטרקטית History ומימושת את הממשק IDatabaseObject. מחלקה זו מייצגת רשומת היסטוריה של קריית שירות במערכת. בנוסף על התכונות שהמחלקה יורשת מהמחלקה History, היא בעלת התכונה הבאה:

```
public int CallId { get; set; }
```

ServiceRecord.cs

מחלקה זו מייצגת רשומה של שירות מסוים. רשומה זו מכילה הפניה (Reference) לשירות (אובייקט מסוג Service) ושלושה מונימים המייצגים כמה קריאות שירות עבור סוג השירות הנוכחי נמצאות בסטאטוס: לא משוייך ("Unassigned"), בתהליך ("In Progress") והסתיים ("Completed"). מחלקה זו נועדה כדי ליצור אובייקטים אשר יתאימו לרשותות בדוח הפעולות. היא בעלת התכונות הבאות:

```
public Service TheService { get; set; }

public int UnassignedCounter { get; set; }

public int InProgressCounter { get; set; }

public int CompletedCounter { get; set; }
```

Setting.cs

מחלקה המimplements את הממשק IDatabaseObject. מחלקה זו מייצגת הגדרה כללית במערכת. כל הגדרה מורכבת ממפתח (string) וערך (string). מחלקה זו נועדה לייצר אובייקטים שיהוו הגדרות כלליות במערכת כגון: שם העסק, כתובת העסק, שעת פתיחה, שעת סגירה ועוד. היא בעלת התכונות הבאות:

```
public string Key { get; set; }

public string Value { get; set; }
```

ShoppingCart.cs

מחלקה הירושת מהמחלקה האבסטרקטית Aggregate. מחלקה זו מייצגת עגלת קניות (או סל קניות) במערכת. מחלקה זו הינה מחלוקת קונקרטית של המחלוקת Aggregate והינה חלק מדף העיצוב Iterator אשר עליו נרחב בהמשך. עגלת הקניות משמשת כדי לאוסף מוצרים או שירותים בעת ביצוע מכירה ללקוח או כדי לאוסף מוצרים לצורך ביצוע הזמנה מספק חיצוני.

מבנה הנתונים שבעזרתו נמשך את עגלת הקניות יהיה רשותה של אובייקטים מסווג ShoppingCartItem. לכן היא בעלת השדה הבא בלבד:

```
private List<ShoppingCartItem> _itemsList;
```

ShoppingCartItem.cs

מחלקה זו מייצגת פריט השיר לעגלת הקניות (ShoppingCart). מחלקה זו מכילה הפניה (Reference) לאובייקט מסווג Purchasable, כלומר כל פריט הקשור למוצר או שירות מסוים. כל מופע של מחלוקת זו מכיל הפניה לפריט עצמו (המוצר או השירות), הכמות של הפריט, עלות ההזמנה, ותיאור. היא בעלת התכונות הבאות:

```
public Purchasable Item { set; get; }

public int Amount { set; get; }

public double OrderCost { set; get; }

public string Description { set; get; }
```

User.cs

מחלקה הירושת מהמחלקה האבסטרקטית UserDefinition. מחלוקת זו מייצגת משתמש קיימ במערכת. להבדיל מהמחלוקת CurrentUser, מחלוקת זו נועד לייצר מופעים רבים מכיוון שהשימוש באובייקטים של מחלוקת זו נועד כדי לנוהל קבוצה של משתמשים במערכת. מחלוקת זו אינה מוסיפה תכונות נוספות למעבר לתכונות אשר עוברות בירושה מהמחלוקת UserDefinition.

UserRecord.cs

מחלקה זו מייצגת רשומה של משתמש מסוים. רשומה זו מכילה הפניה (Reference) למשתמש (אובייקט מסוג User) ושני מונחים המייצגים כמה קרייאות שירות נמצאות בסטאטוס זה אשר משוויכות אותה המשמש. הסטאטוסים של הקרייאות המשוויכות למשתמש הינם: בתהילר ("In Progress") והסתיים ("Completed"). מחלוקת זו נועדה כדי ליצור אובייקטים אשר יתאים לרשומות בדו"ח קרייאות השירות. היא בעלת התכונות הבאות:

```
public User TheUser { get; set; }

public int InProgressCounter { get; set; }

public int CompletedCounter { get; set; }
```

שכבה הציגה (SmartBiz.PI namespace)

שכבה זו מציגת ממשך משתמש (ՈՒ) שיאפשר לו ליצור אינטראקציה עם המערכת. כמו כן, שכבה זו מתממשקת עם שכבת הלוגיקה, בכך לקבל ממנה שירותים (שהיא עצמה מספקת שירות או בעקיפין באמצעות שכבת בסיס-הנתונים).

בטפסים שאינם מהווים חלונות ציצים (PopUp) יעמוד לרשות המשתמש סרגל כלים עליון לביצוע פעולות כלויות (כגון חזרה לתפריט הראשי, שינוי משתמש וכיו'ו') וסרגל כלים תחתון לקבלת חיוי על המשתמש המחבר כעת.

עקרונות מנהים בשכבה הציגה:

- ממשך המשתמש יהיה נוח ופשטן לשימוש.
- כל המחלקות של הטפסים במערכת יירשו ממחלקות אבסטרקטיות אשר יגדירו הגדרות אחידות לכל אותם הטפסים היורשים. כך יהיה ניתן לחלק את הטפסים לקבוצות ובהמשך לקבוע הגדרות מסוימות לקבוצה ספציפית של טפסים. כמו כן, בראש הריררכיה תוגדר המחלקה BasicForm כך שכל הטפסים במערכת יירשו ממנה. עיצוב זה יאפשר לנו גמישות לשינויים עתידיים.
- שימוש ב-UserControls ומחלקות כליות עבור חלקים המשותפים לטפסים רבים כדי למנוע שכפול קוד ועל מנת לבצע שימוש חוזר.
- שימוש ב-Grid בקבצי ה-XML כדי לאפשר הגדרה והתאמאה של החלונות לכל רצולzie אפשרית.
- שימוש בדפוסי עיצוב (Design Patterns) רלוונטיים כדי שהעיצוב של שכבה זו יהיה גמיש לשינויים ויעמוד בידי הנדרשת תוכנה נוספים.

להלן רשימת המחלקות בשכבה זו:

UserControls

UpperToolbar.cs

UserControl אשר משמש כسرיג כלים עליי המופיע במרבית הטפסים. הוא מאפשר לבצע פעולות כלויות כגון: יציאה מהמערכת, התנטוקות של המשתמש הנוכחי, חזרה לאחרור, חזרה לדף הבית,

ኒקוי הטופס ובחרה מרובה. יש לציין כי לכל טופס זמינים אך ורק הרכיבים הרלוונטיים עבורו, והשאר מבוטלים.

LowerToolbar.cs

UserControl אשר משמש כסריג כלים תחתון. הוא מציג חיווי על המשתמש המחבר כרגע למערכת.

ממשקים ומחלקות אבסטרקטיות

BasicForm.cs

מחלקה אבסטרקטית היורשת מהמחלקה Window. מחלקה אבסטרקטית זו משמשת כאב-טיפוס אחד לכל סוגי הטפסים במערכת. כדי ליצור הגדרות אחידות לכל הטפסים במערכת, אנו נגדיר הגדרות אלה תחת הבניי של מחלקה זו. במחלקה זו נגדיר שתכונת the-visibilityTKabel את הערך Visible כדי שברירת המחדל של כל אובייקט מסווג טופס במערכת יהיה גלוי למשתמש מיד לאחר יצירת האובייקט. בנוסף, נגדיר את מיקום החלון למרכז המסך כדי שכל הטפסים במערכת יופיעו במרכז.

מחלקה זו מאפשרת תמייהה בביטויים שינויים עתידיים שתפקידם לכל הטפסים במערכת, כגון: צבע הרקע של הטפסים או כל פעולה או תוכנה אחרת אשר משותפת לכל הטפסים במערכת.

NavigationalForm.cs

מחלקה אבסטרקטית היורשת מהמחלקה BasicForm. מחלקה זו משמשת כאב-טיפוס לכל הטפסים במערכת אשראפשרים ניווט בין מסכים. במחלקה זו נגדיר תכונות ושיטות אשר יהיו משותפות לכל טפסי הנútמות במערכת. כמו כן, שוני סרגלי הכללים אשר יהיו זמינים בין היתר למטרות ניווט.

מחלקה זו תשמש גם לביצוע שינויים עתידיים שתאפשרו לכל טפסי הניוט במערכת.

PopupForm.cs

מחלקה אבסטרקטית היורשת מהמחלקה BasicForm וממסמת את הממשק Observer. מחלקה זו משמשת כאב-טיפוס לכל הטפסים במערכת אשר מהווים מסכי Popup. במחלקה זו נגדיר תכונות ושיטות אשר יהיו משותפות לכל טפסי ה-*Popup* במערכת.

עבור חלק מטפסי ה-kernel ניתן ליצור מופעים ללא הגבלה כדי שהמשתמש יוכל לפתח כמה טפסים במקביל, אך כסוגרים את טופס הנינוichi צריך לסגור גם את כל מסכי ה-kernel הפותחים כתה. לשם כך, מחלוקת זו מימושת את הממשק **Observer** והינה חלק מדף העיצוב אשר עליו נרחב בהמשך.

MenuForm.cs

מחלקה אבסטרקטית היורשת מהמחלקה **NavigationalForm**. מחלוקת אבסטרקטית זו משמשת כאב-טיפוס לכל הטפסים במערכת אשר מהווים תפריט. טפסים אלו מכילים קבוצה של כפתורים המנוטים אל טפסים שונים במערכת. מחלוקת זו מכילה את המימוש של אלגוריתם סידור הcptors. השיטה אשר מימושת אלגוריתם זה מקבלת קבוצה של כפתורים, ובהתאם למספר קביעים אשר מוגדרים בחלוקת זו, השיטה מסדרת את הcptors לפי שורות ועמודות. מחלוקת זו תשמש גם לביצוע שינויים עתידיים שתקפים לכל טפסי התפריטים במערכת.

Observer.cs

ממשק המגדיר שיטה אחת בשם **Update**. ממשק זה הינו חלק מדף העיצוב אשר עליו נרחב בהמשך.

Subject.cs

מחלקה אבסטרקטית אשר מכילה רשימה של אובייקטים מסווג **Observer** ומימושת את הפעולות Notify , Detach ,Attach . מחלוקת זו הינה חלק מדף העיצוב **Observer** אשר עליו נרחב בהמשך.

מחלקה כללית**FilterOrders.cs**

מחלקה זו מכילה אוסף של שיטות סטטיות המבצעות פעולות הקשורות לשינון ההזמנות במערכת. מכיוון שמנגנון השינון (Filter) עברו הזמן מומッシュ בכמה טפסים שונים במערכת, השיטות במחלקה זו נועדו כדי לבצע שימוש חוזר. לעומת מנגנון השינון של הרזמנות במערכת משתמשים בשיטות של מחלקה זו כדי למנוע כפיליות בקוד.

מחלקה זו מכילה את השיטות הבאות:

```
public static void FillDealerCmb(ComboBox dealerCmb)
```

```
public static void FillStatusCmb(ComboBox statusCmb)
```

```
public static string SearchByOrderDetails(ComboBox statusCmb, ComboBox dealerCmb,
```

```
    DatePicker fromDate, DatePicker toDate, CheckBox statusChb, CheckBox dealerChb,
    CheckBox datesChb)
```

FilterProducts.cs

מחלקה זו מכילה אוסף של שיטות סטטיות המבצעות פעולות הקשורות לשינון מוצרים במערכת. מכיוון שמנגנון השינון (Filter) עברו מוצרים מומッシュ בכמה טפסים שונים במערכת, השיטות במחלקה זו נועדו כדי לבצע שימוש חוזר. לעומת מנגנון השינון של המוצרים במערכת משתמשים בשיטות של מחלקה זו כדי למנוע כפיליות בקוד.

מחלקה זו מכילה את השיטות הבאות:

```
public static void FillTypesIntoComboBox<T>(ComboBox comboBox, List<T> purchasables)
where T : Purchasable
```

```
public static void FillManufacturersIntoComboBox(ComboBox comboBox, List<Product>
products)
```

```
public static void FillNamesIntoComboBox(ComboBox comboBox, List<Product> products)
```

```
public static void ManufacturerCmbSelectionChanged(ComboBox manufacturerCmb,
    ComboBox typeCmb, ComboBox nameCmb)
```

```
public static void TypeCmbSelectionChanged(ComboBox manufacturerCmb, ComboBox
typeCmb, ComboBox nameCmb)
```

PageCache.cs

מחלקה הירושת מהמחלקה האבסטרקטית Subject. מחלקה זו מייצגת את זיכרון המטמון של הדפים במערכת. מחלקה זו מחזיקה הפניות (References) לטופס הנוכחי (Current Page), לטופס הקודם (Last Page) ולכל טפסי ה-popup אשר פתוחים כרגע במערכת (נמצאים בראשימה של האובייקטים מסווג Observer אשר מוגדרת במחלקה Subject). מחלקה זו נועדה לתמוך בטפסי הניווט ובסרגלי הכלים כאשר יש לבצע פעולות כגון: סגירת חלון או חזרה אחרת.

מכיוון שיש צורך במוועע ייחד של זיכרון המטמון במערכת, מחלקה זו הינה חלק מדפו העיצוב singleton אשר עליו נרחב בהמשך.

בנוסף, מחלקה זו הינה חלק מדפו העיצוב observer אשר גם עליו נרחב בהמשך.

מחלקה זו מכילה את השיטות הבאות:

```
public static PageCache GetInstance()
public void CloseAllPopupForms()
```

טפסים קונקרטיים במערכת

AddUpdateInventoryItemForm.xaml.cs

מחלקה הירושת מהמחלקה האבסטרקטית Popup. מחלקה זו מייצגת את הטופס: "הוספת פריט חדש למלאי או עדכון פריט קיימ". באמצעות טופס זה ניתן להוסיף מוצר או שירות חדש למלאי, בנוסף ניתן לעורוך מוצר או שירות קיימ.

AddUpdateOrderForm.xaml.cs

מחלקה הירושת מהמחלקה האבסטרקטית NavigationalForm. מחלקה זו מייצגת את הטופס: "הוספת הזמנה חדשה / עדכון או צפיה בהזמנה קיימת". באמצעות טופס זה ניתן להוסיף הזמנה חדשה מול ספק חיצוני, בנוסף ניתן לעורוך או לצפות בהזמנות קיימות. סטאטוס ההזמנה יקבע את הרשאות הצפיה או העריכה.

AddUpdateUserForm.xaml.cs

מחלקה הירושת מהמחלקה האבסטרקטית PopupForm. מחלקה זו מייצגת את הטופס: "הוספת משתמש חדש או עדכון משתמש קיימ". באמצעות טופס זה ניתן להוסיף/ערוך/לצפות משתמשים. בטופס זה ניתן להגדיר את פרטי המשתמש, הרשאותיו ואת סיסמתו.

AdminMenuForm.xaml.cs

מחלקה הירושת מהמחלקה האבסטרקטית NavigationalForm. מחלקה זו מייצגת את הטופס: "טופס ראשי השיר למשתמש Admin". באמצעות טופס זה ניתן לנוט לטפסים אשר מנהל המערכת רשאי לפתח. באמצעות טופס זה ניתן לפתח טופס לניהול משתמשים או קביעת הגדרות כלליות.

BillForm.xaml.cs

מחלקה הירושת מהמחלקה האבסטרקטית PopUpForm. מחלקה זו מייצגת את הטופס: "הפקת חשבונית". טופס זה מהווה חשבונית ללקוח ומציג את פירוט הקנייה שבוצעה ופרטים רלוונטיים נוספים.

ChangePasswordForm.xaml.cs

מחלקה הירושת מהמחלקה האבסטרקטית BasicForm. מחלקה זו מייצגת את הטופס: "שינוי סיסמה". טופס זה נפתח לאחר שימוש מסויים הגדה במסך ה-Login ונקבע עבורי ההגדרה שמחיבת לשנות את הסיסמה ולהמשיך לאחר מכן את הפעולות השוטפת במערכת, או לבצע יצאה ולשנות את הסיסמה מאוחר יותר. ככל מקורה לא תאפשר כניסה אל המערכת עבור אותו המשתמש עד אשר יבצע שינוי סיסמה.

CustomerDetailsForm.xaml.cs

מחלקה הירושת מהמחלקה האבסטרקטית NavigationalForm. מחלקה זו מייצגת את הטופס: "פרטי לקוח". באמצעות טופס זה ניתן להוסיף לקוח חדש או לעורר או לצפות בלקוח קיימ. בטופס זה מצוים כל הפרטים האישיים של הלקוח כולל היסטורית לקוח. הרשות בטופס זה נקבעות על סמך הרשותיו של המשתמש הנוכחי.

CustomerHistoryEventForm.xaml.cs

מחלקה הירושת מהמחלקה האבסטרקטית PopUpForm. מחלקה זו מייצגת את הטופס: "ההיסטוריה לקוח". באמצעות טופס זה ניתן לעורר או לצפות באירוע היסטורית לקוח. הרשות בטופס זה נקבעות על סמך הרשותיו של המשתמש הנוכחי.

CustomersForm.xaml.cs

מחלקה הירושת מהמחלקה האבסטרקטית NavigationalForm. מחלקה זו מייצגת את הטופס: "ניהול לקוחות". באמצעות טופס זה ניתן לחפש לקוח על פי מספר תכונות. טופס זה מאפשר לצפות

בקבוצה שלLKוקחות ובנוסף להסיר או להוסיף לKוקחות מהמערכת. הרשות בטופס זה נקבעות על סמך הרשאותיו של המסתמש הנוכחי.

CustomersReportForm.xaml.cs

מחלקה הירושת מהמחלקה האבסטרקטית `NavigationalForm`. מחלקה זו מייצגת את הטופס: "דו"ח ל��וחות". באמצעות טופס זה ניתן להפיק דו"ח המציג את כל הליקוחות שהתווסףו למערכת בטוויח של תאריכים ושעות.

DiscountsForm.xaml.cs

מחלקה הירושת מהמחלקה האבסטרקטית **NavigationalForm**. מחלקה זו מייצגת את הטופס: "ניהול מבצעים". באמצעות טופס זה ניתן לקבוע מבצעים על המוצרים או השירותים הקיימים בעסק. ניתן לקבוע שני סוגי מבצעים: 1. הנחה לפי אוחזים. 2. מבצע 1+1. הטופס מאפשר גמישות רבה בכך שניתן להשתמש במנגנון הסינון ולבחור קטגוריה של מוצר מסוים על פי יצרן או על פי יצרן מסווג, ולקבוע את הרנהה רק על קטgorיה ספציפית.

ExpensesIncomeReportForm.xaml.cs

מחלקה הירושת מהמחלקה האבסטרקטית NavigationalForm. מחלקה זו מייצגת את הטופס "דו"ח הכנסות-הוצאות". באמצעות טופס זה ניתן להפיק דו"ח המציג את הכנסות העסק מול הוצאות העסק. חשוב להעיר כי לא ניתן להפיק באמצעות דו"ח זה מידע על הרווחים של העסק. ניתן להפיק את הדו"ח על פי טווח של תאריכים ושעות.

GlobalSettingsForm.xaml.cs

מחלקה הירושת מהמחלקה האבסטרקטית PopupForm. מחלקה זו מייצגת את הטופס: "הגדרות כלויות". באמצעות טופס זה ניתן לעורר את הגדרות הכלליות של המערכת. טופס זה נגיש אך ורק למננהל המערכת (Admin) והוא רשאי לקבוע הגדרות שונות אשר ישפיעו על כלל המערכת. באמצעות טופס זה ניתן לטעון קובץ מודול על מנת להתאים את המערכת לסוג עסק ספציפי.

InventoryForm.xaml.cs

מחלקה הירושת מהמחלקה האבstarטקטית NavigationalForm. מחלקה זו מייצגת את הטופס: "ניהול מלאי". באמצעות טופס זה ניתן לחפש מוצרים או שירותים הקיימים במלאי העסוק. במרכז,

טופס זה מאפשר להוסיף מוצריים/שירותים חדשים או להסיר קיימים. ההרשאות בטופס זה נקבעות על סמך הרשותיו של המשתמש הנוכחי.

[LoginForm.xaml.cs](#)

מחלקה הירושת מהמחלקה האבסטרקטית BasicForm. מחלקה זו מייצגת את הטופס: "התחברות למערכת". באמצעות טופס זה ניתן להתחבר אל המערכת באמצעות שם משתמש וסיסמה. למעשה זהו הטופס הראשי שנפתח עם עליית המערכת. טופס זה מגביל את הגישה למערכתvr שרק משתמשים קיימים ופעילים יכולים להתחבר אל המערכת.

[MainMenuForm.xaml.cs](#)

מחלקה הירושת מהמחלקה האבסטרקטית MenuForm. מחלקה זו מייצגת את הטופס: "מסך ראשי". באמצעות טופס זה ניתן לנוט בין טפסי המערכת שאיליהם קיימת גישה. ההרשאות בטופס זה נקבעות על סמך הרשותיו של המשתמש הנוכחי, ובהתאם להרשאות אלו, יופיעו הcptors המתאימים שדרכם ניתן לנוט לטפסים הרלוונטיים.

טופס זה מאפשר גמישות רבה, כאשר משתמש פועל בעל הרשות מתחבר אל המערכת, טופס זה מציג בפנוי את כל הcptors הרלוונטיים גם אם משתמש קיימות כמו הרשות בו-זמןית.

[OperationsReportForm.xaml.cs](#)

מחלקה הירושת מהמחלקה האבסטרקטית NavigationalForm. מחלקה זו מייצגת את הטופס: "דו"ח פעילות". באמצעות טופס זה ניתן להפיק דו"ח אודות סוג השירותים השונים הקיימים בעסק. דו"ח זה מציג כמה קריאות שירות קיימות עבור כל סוג שירות בכל אחד מהסטאטוסים הבאים: "Completed", "In Progress", "Unassigned" ו-"-". טופס זה מאפשר להפיק את הדו"ח על פי טווח של תאריכים ושעות ובנוסף על פי שירט לעובד מסוים.

[OrdersForm.xaml.cs](#)

מחלקה הירושת מהמחלקה האבסטרקטית NavigationalForm. מחלקה זו מייצגת את הטופס: "ניהול הזמנות ציוד". באמצעות טופס זה ניתן ליצור הזמנות ציוד מול ספק חיצוני. טופס זה מאפשר גם לצפות או לערוך הזמנות ציוד קיימות. הרשות העERICA נקבעת על פי סטאטוס ההזמנה. באמצעות טופס זה ניתן לשנות את סטאטוס ההזמנה ובכך להגדיל כמויות של מוצריים קיימים במלאי העסק.

OrdersReportForm.xaml.cs

מחלקה היורשת מהמחלקה האבסטרקטית NavigationalForm. מחלקה זו מייצגת את הטופס: "דו"ח הזמנות". באמצעות טופס זה ניתן להפיק דו"ח הזמנות. טופס זה מאפשר להפיק את הדו"ח על פי טווח של תאריכים או/ו על פי ספק או/ו על פי סטאטוס הזמנה. בדו"ח זה ניתן לראות רשימה של כל ההזמנות המתאימות ובנוסף את העלות הכלולת שלהן.

PaymentForm.xaml.cs

מחלקה היורשת מהמחלקה האבסטרקטית PopupForm. מחלקה זו מייצגת את הטופס: "קבלת התקבולים". באמצעות טופס זה ניתן לקלוט את תשלום הלקוח בעקבות רכישה שביבצע. בטופס זה ניתן לבחור בין שתי אפשרויות תשלום: מזומנים או אשראי. קליטת אמצעי התשלום ואישורו יבצעו מספר פעולות כגון: הפתק חשבונית, פתיחת קריאות שירות עבור השירותים הנרכשים, גריעה המלאי ויצירת תיעוד בהיסטוריית לקוחות.

PricingForm.xaml.cs

מחלקה היורשת מהמחלקה האבסטרקטית NavigationalForm. מחלקה זו מייצגת את הטופס: "תמחרר מוצרים". באמצעות טופס זה ניתן לשנות את ערך המע"מ ובנוסף להוריד/להעלות את המאפיינים של המוצרים כולם באחוז כלשהו. טופס זה מאפשר להציג את השינויים לפני שמירת הנתונים.

ReportsMainForm.xaml.cs

מחלקה היורשת מהמחלקה האבסטרקטית MenuForm. מחלקה זו מייצגת את הטופס: "מסך דו"חות". באמצעות טופס זה ניתן לנoot בין טפסי הדו"חות השונים קיימת גישה. ההרשאות בטופס זה נקבעות על סמך הרשאותיו של משתמש הנוכחי, ובהתאם להרשאות אלו, יופיעו הcptוראים המתאימים שדרכם ניתן לנoot לטפסי הדו"חות הרלוונטיים.

SalesForm.xaml.cs

מחלקה היורשת מהמחלקה האבסטרקטית NavigationalForm. מחלקה זו מייצגת את הטופס: "ניהול מכירות". באמצעות טופס זה ניתן לבצע מכירה ללקוח או להציג הצעת מחיר. כדי לבצע מכירה ללקוח יש לחבר ללקוח ספציפי ולאחר מכן לחבר את המוצרים או/ו השירותים שאوتם הלקוח מעוניין לרכוש. ניתן לראות בכל שלב את העלות הכלולת של ההצעה. עלות זאת כוללת את המע"מ ואת ההנחהות השונות במידה וקיימות.

SalesReportForm.xaml.cs

מחלקה הירושת מהמחלקה האבסטרקטית NavigationalForm. מחלקה זו מייצגת את הטופס: "דו"ח מכירות". באמצעות טופס זה ניתן להפיק דו"ח המאגד את סך המכירות שנמכרו מכל אחד מסוגי המוצרים / שירותים. דו"ח זה מאפשר להציג מכירות שבוצעו בטוווח תאריכים נתון / או בטוווח שבועות נתון.

ServiceCallForm.xaml.cs

מחלקה הירושת מהמחלקה האבסטרקטית NavigationalForm. מחלקה זו מייצגת את הטופס: "קריאת שירות". טופס זה מאפשר מעקב אחר קריאת שירות. באמצעות טופס זה ניתן להקצות את קריאת השירות לאחד מעובדי הוצאות, לעדכן סטאטוס ולנהל את היסטורית קריאת השירות. הרשאות הקצאה לאחד מעובדי הוצאות בטופס זה נקבעת על סמך הרשותינו של המשתמש הנוכחי.

ServiceCallHistoryEventForm.xaml.cs

מחלקה הירושת מהמחלקה האבסטרקטית PopupForm. מחלקה זו מייצגת את הטופס: "אירוע היסטורית קריאת שירות". טופס זה מאפשר לצפות / לעורר את ההערכות של רשומות ההיסטוריה הנוכחית.

ServiceCallReportForm.xaml.cs

מחלקה הירושת מהמחלקה האבסטרקטית NavigationalForm. מחלקה זו מייצגת את הטופס: "דו"ח קריאת שירות". באמצעות טופס זה ניתן להפיק דו"ח אודוות כמות קריאות השירות בהם טיפול או מטפל כל אחד מעובדי הוצאות בטוווח תאריכים נתון / או בטוווח שבועות נתון. טופס זה מציג עבור כל אחד מעובדי הוצאות מהו מספר קריאות השירות שנפתחו באותה תקופה זמן, והביאו אותן לסגירה (Completed). בנוסף, טופס זה מציג עבור כל אחד מעובדי הוצאות מהו מספר קריאות השירות שנפתחו באותה תקופה זמן, והן עדין בטיפול (In Progress).

UserManagementForm.xaml.cs

מחלקה הירושת מהמחלקה האבסטרקטית NavigationalForm. מחלקה זו מייצגת את הטופס: "ניהול משתמשים". באמצעות טופס זה ניתן להוסיף או להסיר משתמשים מהמערכת. טופס זה מציג את כל המשתמשים הקיימים במערכת אשר נמצאים במצב פעיל. לחיצה כפולה על אחת הרשומות תפתח את הטופס שמאפשר לצפות / לעורר את הפרטים המלאים של המשתמש.

WorkDistributionForm.xaml.cs

מחלקה הירושת מהמחלקה האבסטרקטית NavigationalForm. מחלקה זו מייצגת את הטופס: "חלוקת עבודה". באמצעות טופס זה ניתן לחפש אחר קריאות שירות הקיימות במערכת. ניתן לבצע סינון בעת החיפוש על פי תאריכים, סוג השירותים, סטאטוס או/ו שיוך. לחיצה כפולה על אחת הרשומות תפתח את טופס קריית השירות עם הפרטים המלאים של אותה הקריאה.

ראש הצוות רשאי לראות את כל קריאות השירות הקיימות במערכת. עובד רשאי לראות רק את קריאות השירות המשתייכות אליו. הרשות זו נקבעת על סמך הרשותי של המשתמש הנוכחי.

דפוסי עיצוב (Design Patterns)

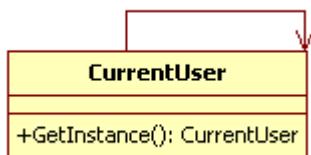
דףוֹיִן העיצוב בהם יעשה שימוש בפרויקט המ:

Singleton

תבנית שתפקידה לוודא שקיים מוליך מסווג אחד בדיק, בנוסף התבנית מספקת מצביע גלובלי שיאפשר לגשת למופיע זה. שימוש בתבנית זו ימנע מהיווצרותן של שגיאות לוגיות הנגרמות כתוצאה מיצירה של כמה מופעים עבור מחלקות שנדרש עבורם מופיע אחד לכל היותר.

בתבנית זו נעשה שימוש במחלקות הבאות:

1. CurrentUser – מחלקה זו מייצגת את המשתמש הנוכחי שמחובר למערכת. מכיוון שבכל רגע נתון יכול אף ורק משתמש אחד להיות מחובר אל המערכת, נעשה שימוש בתבנית Singleton במחלקה זו. שימוש בתבנית העיצוב Singleton יעזר לנו לוודא שלא קיים יותר מופיע אחד של המחלקה CurrentUser. למופיע זה אנו ניגש מחלקים שונים בתכנית כדי לבדוק את הרשותיו של המשתמש הנוכחי. בנוסף יעשה שימוש בתוכנות נוספות של האובייקט, לדוגמה: הצגת שמו המלא של המשתמש הנוכחי בסרגל הכלים התחתון.



2. PageCache – מחלקה זו מייצגת את זיכרון המטען של הדפים במערכת. מכיוון שבכל רגע נתון יכול להיות רק זיכרון מטען אחד במערכת, נעשה שימוש בתבנית Singleton במחלקה זו. שימוש בתבנית העיצוב Singleton יעזר לנו לוודא שלא קיים יותר מופיע אחד של המחלקה PageCache. למופיע זה אנו ניגש מחלקים שונים בתכנית כדי לגשת לדף הנוכחי, לדף הקודם ולכל חלונות ה-Popup אשר פתוחים כרגע במערכת.

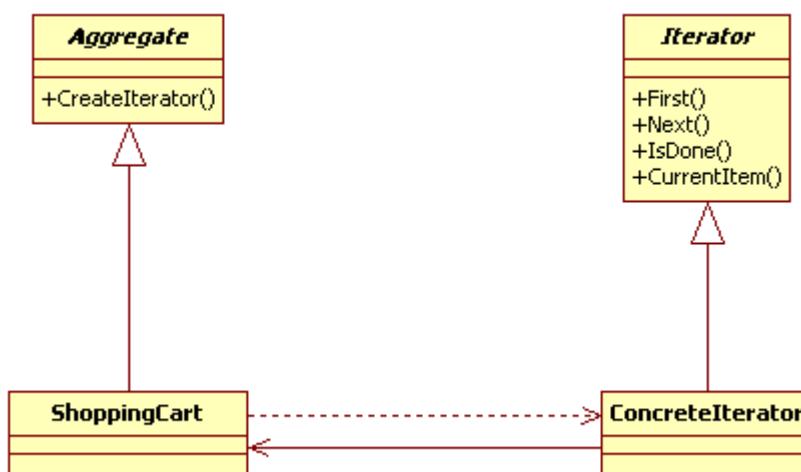


Iterator

תבנית המספקת דרך לגשת באופן סדרתי לאיברים של אובייקט מורכב מבעלי לחושף את הייצוג הפנימי שלו. הตำบลית Iterator מפרידה בין הממשק של קריית הנתונים למבנה הנתונים עצמו, ובכך מאפשרת לשנות את מבנה הנתונים ללא שינוי הקוד של המשתמש.

בתבנית זו נעשה שימוש במחלקות הבאות:

1. ShoppingCart – מחלקה זו מייצגת את עגלת הקניות (סל קניות) של המוצרים והשירותים במערכת. המשמשים של מחלקה זו צריכים לבצע פעולות כגון: סריקה של כל הפריטים בעגלה לצורך חישוב עלות כוללת, סריקה של כל המוצרים בעגלה לצורך גירעה מהמלאי ברגע שהליך שלם על הרכישה וכו'. כל המשמשים בחלוקת זו אינם צריכים לדעת מהו מבנה הנתונים שבו מאוחסנים הפריטים. משתמשים אלו יכולים להסתפק ב-Iterator שבאמצעותו יכולים לעבור על כל הפריטים בעגלה ללא חשיפה של הייצוג הפנימי שלה. המחלקה ShoppingCart תשתתף בתבנית Iterator בתפקיד – Iterator אשר יורש מהמחלקה האבסטרקטית Aggregate.



Strategy

tabniti magdirah meshpacha shel algoritmim, mcmast (Encapsulate) cel achd mham, vma'asheret la'hali'f at shi'mosh bhem. Kolomr, tabniti ma'asheret l'shnot at algoritmim ba'ofen uzmai la'a telot b'mashamim shel mchaka.

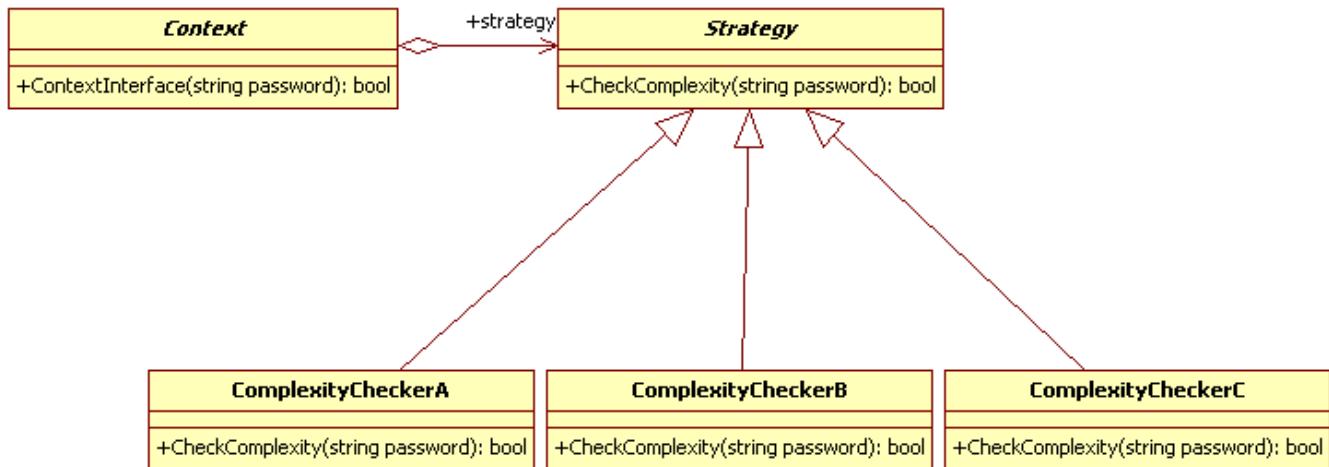
tabniti zo nusa she'imosh b'mchakot ha'avot:

1. A ComplexityChecker – mchaka zo mi'zgat at algoritm b'dikat morcavot ha'rason. Algoritm zeh makbl kklut mchrozot v'vadk ha'm mchrozot mchila le'fchot 2 (mogdr ckbo - MINIMUM_AMOUNT) towim asher cel achd mham shir lkavza another, caser kavzot ha': otiot ktnot, otiot gdolot, sforot v'simanim achrim.

2. B ComplexityCheckerB – mchaka zo mi'zgat at algoritm b'dikat morcavot ha'seni. Algoritm zeh makbl kklut mchrozot v'vadk ha'm mchrozot la'mchila shni towim zehim ukbim. Kolomr, ul pi algoritm zeh, sisma morcavat hina sisma asher la'mchila shni towim zehim ukbim.

3. C ComplexityCheckerC – mchaka zo mi'zgat at algoritm b'dikat morcavot ha'seli'shi. Algoritm zeh makbl kklut mchrozot v'vadk ha'm mchrozot la'mchila shni towim ukbim shiyicim la'otah kavza (otiot ktnot, otiot gdolot, sforot v'simanim achrim). Kolomr, ul pi algoritm zeh, sisma morcavat hina sisma asher la'mchila shni towim ukbim shiyicim la'otah kavza.

cl achd mshloshet algoritmim hino mesh'taf batpekid ConcreteStrategy shel tabniti Strategy. Cimous algoritmim la'obiyektim shoniim ya'asr lnou la'hali'f at algoritmim ba'ofen uzmai la'a telot b'mashamim shel mchaka. Kr nocel lehosi'f b'dikot morcavot sisma nosofot, shelb camha b'dikot shi'tbazu bratzf au la'hali'f binian la'a telot b'mashamim. Shimosh tabniti ya'asr lnou gamishot le'shiniyim utidiim b'mida v'niraza lehosi'f lemashl algoritm morcavot nosof.



Observer

תבנית המגדירה תלות אחד-ל רבים בין האובייקטים, כאשר מצב של אובייקט אחד משתנה, כל שאר האובייקטים שתלוים בו מקבלים התראה על השינוי ומתעדכנים באופן אוטומטי.

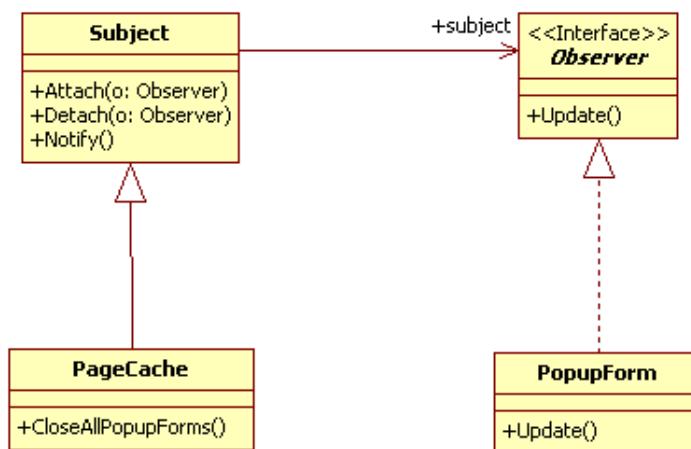
בתבנית זו נעשה שימוש במחלקות הבאות:

1. PageCache – מכוון שמחלקה זו מייצגת את זיכרון המטען של הדפים במערכת, מחלוקת זו צריכה להכיל רשימה של הפניות (References) לכל חלונות-hkupop שפותחים כרגע במערכת. כאשר אנו מבצעים סגירה של חלון ניוט אשר דרכו פתחנו מספר חלונות מסווג hkupop, יש לבצע סגירה מסודרת של כל חלונות-hkupop שפותחים כרגע, מכיוון שאינם רלוונטיים עוד. לפיכך, מחלוקת זו תשתוף בתפקיד ConcreteSubject של התבנית Observer. מחלוקת זו תירוש מהמחלקה האבstarktית Subject וכך תקבל ממנה בירושה את הפעולות הבאות: Attach – להוספה אובייקטים מסווג Observer, Detach – להסרת אובייקטים מסווג Observer ו-Notify – לקריאה לכל האובייקטים הרשומים להתקען לפי המצב הנוכחי. כאשר נסגור את החלון הנוכחי, ה-hkupop יישנה את מצבו מ- REGULAR_STATE ל-EXIT_STATE, ולאחר מכן ייקרא ל-Notify כדי לעדכן את כל חלונות-hkupop שנרשמו לה-hkupop כדי שיבצעו סגירה.

2. PopupForm – מחלוקת זו מייצגת את כל טפסי-hkupop במערכת. מחלוקת זו תמשש את הממשק Observer המכיל שיטה אחת בשם Update. שיטה זו תעדכן את מצב הטופס במצבו הנוכחי של ה-hkupop. כאשר האובייקט מסווג PopupForm מעדכן את מצבו, הוא בודק האם מצב זה הינו EXIT_STATE, אם כן, אז החלון יבצע סגירה, אחרת ימשיך לפעול כרגיל. מחלוקת זו מימושת את הממשק Observer ולכן היא משתתפת בתפקיד ConcreteObserver של התבנית Observer.

SmartBiz

באמצעות שימוש בתבנית זו אנו מונעים צמידות חזקה בין המחלקות PopupForm ו-PageCache. כך בעצם נוכל בעתיד להגדיר סוג טפסים נוספים אשר ימשו את המשק Observer ויירשם באמצעות Attach ל觀察者 PageCache כדי להתעדכן מולו לגבי מצבו הנוכחי ולפעול בהתאם. ככלומר, תבנית זו מאפשרת לנו גמישות לשינויים עתידיים.



שינויים עתידיים אפשריים

מערכת זו תוכננה ועוצבה כך ששינויים עתידיים יהיו קלים ליישום. בסכム כתה כיצד הגמישות לשינויים באה לידי ביטוי ואילו שינויים ניתן יהיה לבצע בעתיד.

1. מערכת זו בנויה על סמך מודל 3-Tier, שבו המערכת מופרדת לשולש שכבות. הפרדה זו מאפשר לנו בעתיד להחליף את המשק בין שתי שכבות מבלי להחליף את המשק בין שתי השכבות האחרות, וכן, להחליף את המימוש של אחת השכבות מבלי להחליף את המימוש של השכבות האחרות. כך למשל, אם נחליט שאנו מעוניינים לעבוד מול מודד נתונים שונה מ-SQL Server נוכל להחליף בקלות רק את המימוש של שכבת ה-Data מבלי לפגוע במימוש שכבות האחרות. דוגמה נוספת לשינוי היא החלפת משק משתמש מ-MySQL למשק MongoDB שכבת התצוגה מבלי לפגוע בשכבות האחרות.
2. היררכיית הירושה הקיימת בין הטפסים השונים במערכת מאפשר לנו להחיל שינויים על כל הטפסים במערכת (באמצעות המחלקה BasicForm) או רק על חלקם (באמצעות המחלקות: NavigationalForm, MenuForm, PopupForm). כך לדוגמה, אם נחליט שאנו רוצים לקבוע גודל אחיד לכל טפס ה-SubForm במערכת, נוכל לבצע שינוי זה במחלקה PopupForm ללא הצורך לשנות טופס אחר טופס.

3. לכל אורך התוכנית הוגדרו קבועים עבור הערכים אשר בהם אנו עושים שימוש. קבועים אלו יאפשרו לנו לשנות את הערכים במקומות אחד בלבד ללא הצורך לבצע חיפוש בתוכנית כולה, כדי למצוא היכן משתמשים בערכים אלו. כך למשל, אם נרצה בעתיד לשנות את סיסמתו של מנהל המערכת, יוכל לשנות את ערך הקבוע: `PASSWORD_ADMIN`.

4. שימוש המחלקה `ShoppingCart` באמצעות התבנית `Iterator` מאפשרת לנו לשנות את מבנה הנתונים של הפריטים בעגלת הקניות ללא הצורך לשנות את השימוש של המשתמשים במחלקה זו. מבנה הנתונים הנבחר לצורך מכן מימוש קניות הוא רשיימה, מכיוון שמבנה נתוניים זה הינו פשוט ומתאים לצרכים של עסקים קטנים (בדרך כלל עגלת קניות תכיל עד 50 פריטים). אם בעתיד נגלה צורך אצל משתמשי המערכת בעגלת קניות שתכיל אלפי או יותר פריטים לעגלה, נshall להחליף את מבנה הנתוניים במבנהיעיל יותר אשר יאפשר לנו לבצע חישובים כגון עלות כולל בזמן ריצהיעיל יותר.

5. שימוש אלגוריתמי בדיקת מורכבות הסיסמה באמצעות התבנית `Strategy` מאפשרת לנו להחליף את האלגוריתמים או להוסיף חדשים ללא הצורך לשנות את השימוש בשימושים של המחלקה. כך בעצם נוכל להתאים בקלות גרסאות שונות לעסקים שונים אשר לכל אחד מהם קיימת דרישת שונה לבדיקת המורכבות. לדוגמה: עסק כגון מספורה אשר מכיל 2 עובדים בלבד (מנהל המספורה שהוא גם מנהל המכירות וראש הצוות, וספר נוסף) אינו צריך מורכבות גבוהה של סיסמות ולכן יסתפק רק באלגוריתם בדיקת המורכבות הראשונית. לעומת זאת, עסק כגון מעבדת מחשבים אשר מכיל את כל סוג המשתמשים כאשר על ראש הצוות חל איסור לצפות בפרטי הלקוחות, ידרש מורכבות גבוהה של סיסמות ולכן נדרש לשלוח `checkComplexity` בבדיקה `UserValidator` את שלושת האלגוריתמים שיבדקו בראצף את מורכבות הסיסמה.

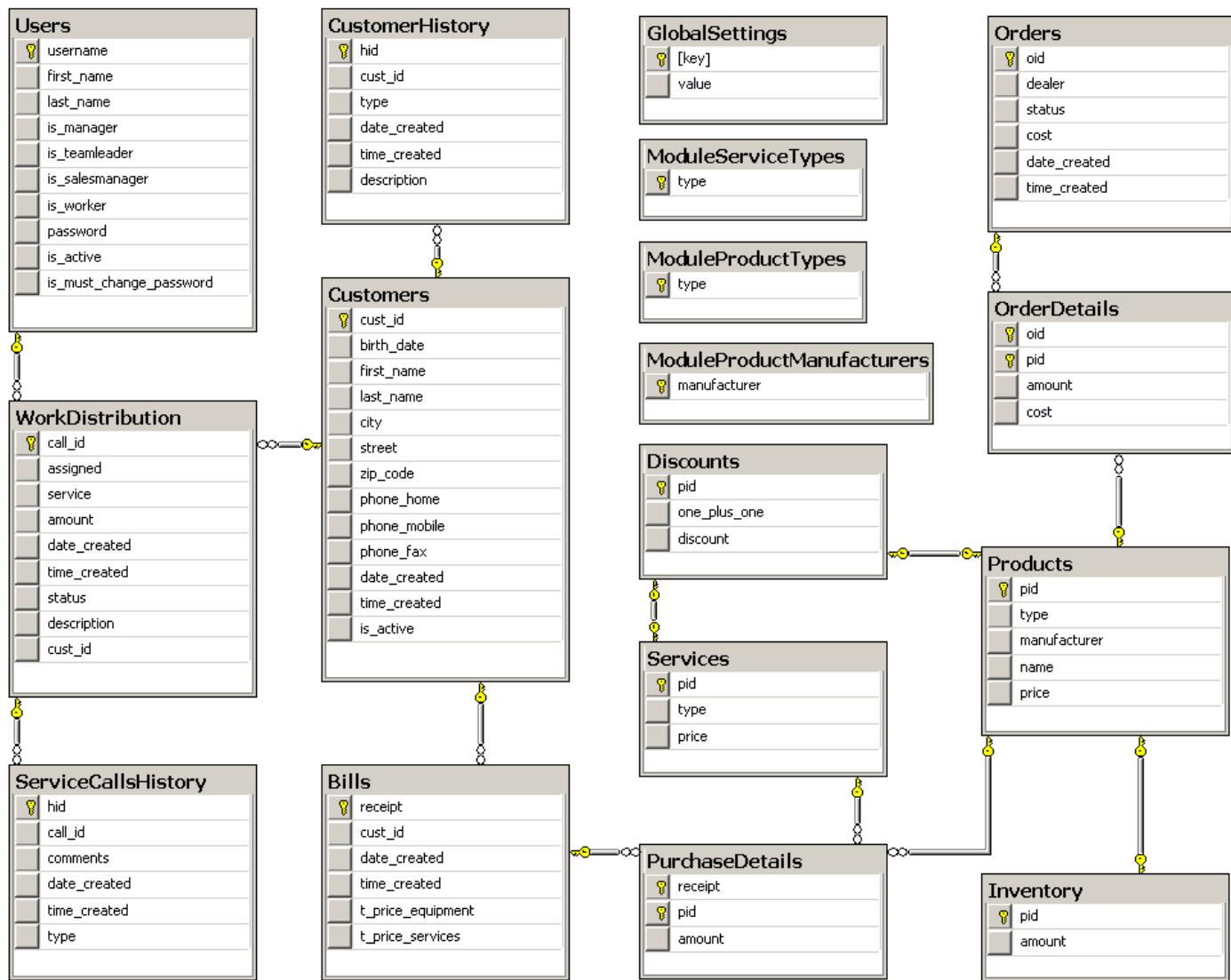
6. שימוש המחלקות `PageCache` ו-`PopupFrom` באמצעות תבנית העיצוב `Observer` מאפשרת לנו גמישות רבה לשינויים. נוכל להחילט בעתיד כי קיימים צריכים לשוגים שונים של טפסים (שונים מ-`PopupFrom`) להடען מול `PageCache` לגבי מצבו הנוכחי ולפעול בהתאם. נוכל לבצע שינוי זה בקלות ע"י כך שנשתמש את המחלקה המבוקשת באמצעות הממשק `Observer` כדי שה-`PageCache` יוכל לבצע רישום של דפים מהמחלקה זו ולעדכן אותם כאשר מצבו משתנה. כך למשל, נוכל להוסיף מצב חדש שנקרא `DISABLE_STATE` אשר יקרה לקבוצה מסוימת של טפסים לעבור במצב לא פעיל כאשר מתרחש אירוע חריג כלשהו בזיכרון המתמן.

7. במערכת זו קיימים שני סוגי של פריטים אשר ניתן לרכוש: מוצרים ושירותים. הגדרת שתי המחלקות עבור פריטים אלו תחת המחלקה האבסטרקטית `Purchasable` בהיררכיה הירושה של המערכת, מאפשרת לנו לרכוש את כל השיטות והתכונות המשותפות לפריטים אלו במחלקה זו. עיצוב זה גמיש לשינויים מכיוון שניתן יכולם לשנות את מימוש השיטות במחלקה זו, וכן השינוי יבוא לידי ביטוי גם עבור מוצרים וגם עבור שירותים במערכת. לדוגמה, השיטה `GetPriceIncVatAndDiscount` אשר ממחישה את מחיר המוצר לאחר

הוספת המע"מ וחישוב ההנחה, מוסיף קודם כל את המע"מ ורק לאחר מכן מחשבת את ההנחה. אם נרצה בעתיד לשנות את השימוש של השיטה כך שקדם תחשב ההנחה ורק לאחר מכן יתווסף המע"מ, אז שינוי זה יהיה תקף גם למוצרים וגם לשירותים. נוסף על כן, עיצוב זה מאפשר לנו להוסיף בעתיד סוג נוסף של פריט אשר ניתן לרכוש שאינו מוצר ואין שירות. מכיוון שעגלת הקניות מכילה רשימה של אובייקטים מטיפוס Purchasable, אז לא נדרש לשנות את השימוש של עגלת הקניות וייה ניתן להוסיף לעגלת הקניות פריטים מהסוג החדש.

8. הגדרת המערכת יכולה כמערכת גנריית המתאימה לסוגים רבים של עסקים קטנים ומימוש הפונקציונליות של טעינת קובץ המודול מאפשרים שימוש רב לשינויים עתידיים. ניתן להוסיף על הפונקציונליות של טעינת קובץ המודול, שבנוסף לטעינת רשימות היצרנים, סוג השירותים וסוגי המוצרים, שטעינת הקובץ תבצע התאמות נוספות לעסק המתאים. לדוגמה, ניתן לבנות טפסים נוספים אשר רלוונטיים רק לסוג עסק מסוים אשר הפקו להיות זמינים רק כאשר יטען קובץ המודול המתאים של אותו העסק.

(Data Base Diagram)



דיאגרמות מחלקות (Class Diagrams)

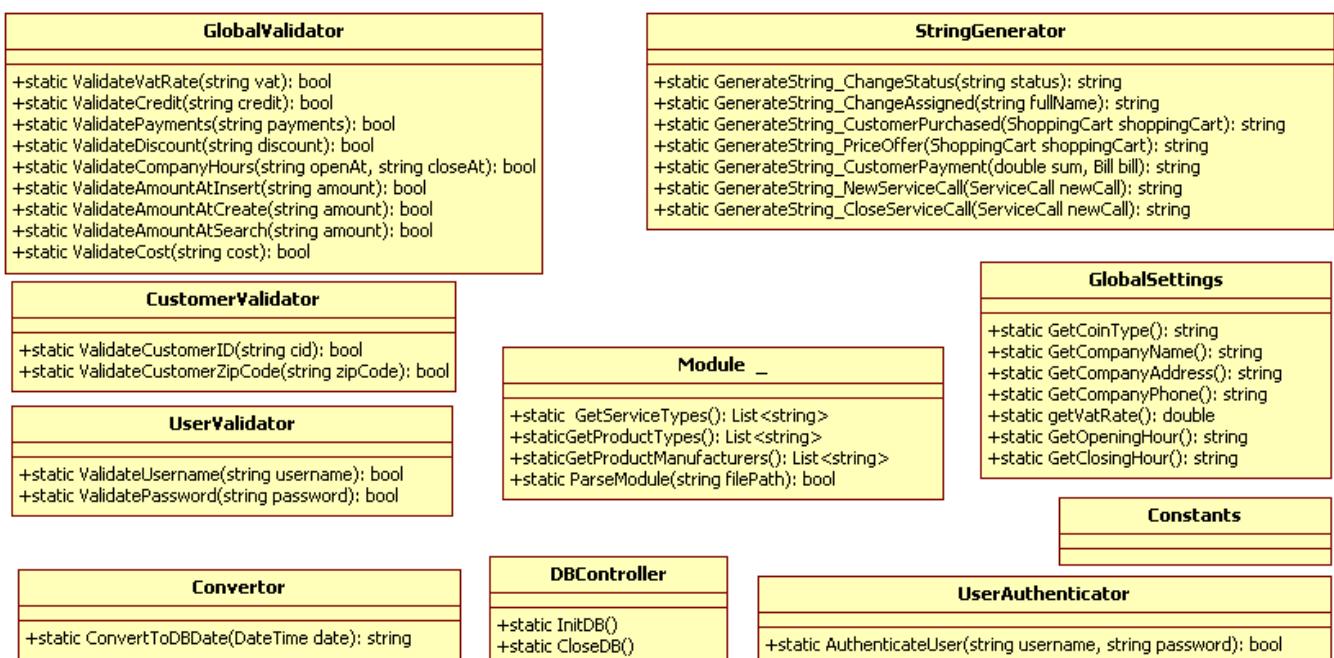
הערות כלליות עבור דיאגרמות המחלקות:

1. הסימן + מצבן גישה ציבורית (Public).
2. הסימן - מצבן גישה פרטית (Private).
3. הסימן # מצבן גישה מוגנת (Protected).
4. הבנאים אינם מופיעים בדיאגרמות.
5. שיטות עזר פרטיות אין מופיעות בדיאגרמות כדי לא להעמס יתר על המידה.

דיאגרמת מחלקות עבור הכלליות בשכבה ה-BI

הערות:

1. כל המחלקות אשר מופיעות בדיאגרמה זו הן כלליות ומילוט שיטות סטטיות בלבד.



דיאגרמת מחלקות עבור האובייקטים בשכבה ה-Bl

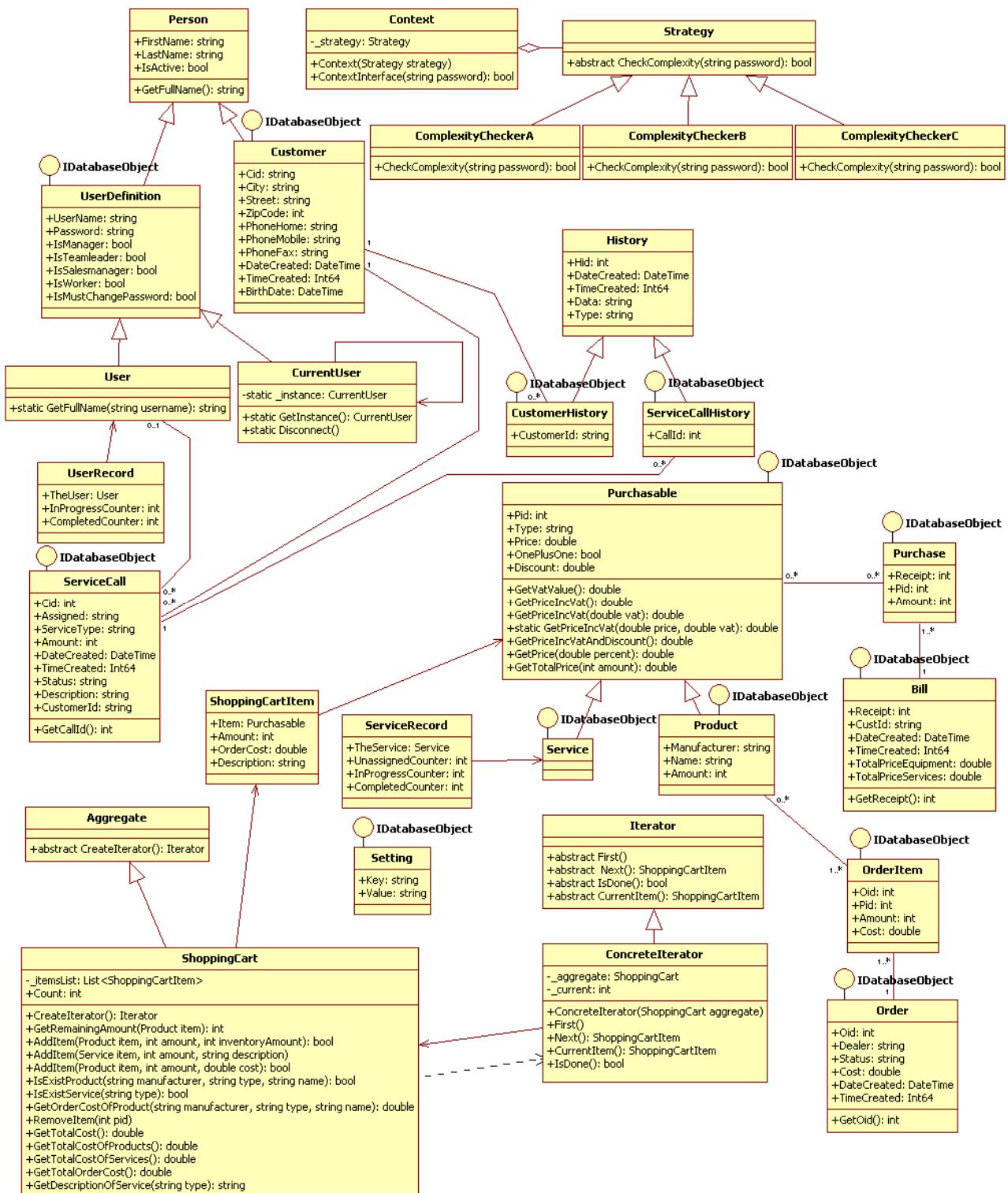
הערות:

1. כל מחלקה המממשת את הממשק `IDatabaseObject`, מכילה את השיטות הבאות:

```
bool InsertToDB(); bool RemoveFromDB(); bool UpdateDB(); bool IsExistInDB();
```

שיטות אלה לא מופיעות בדיאגרמה.

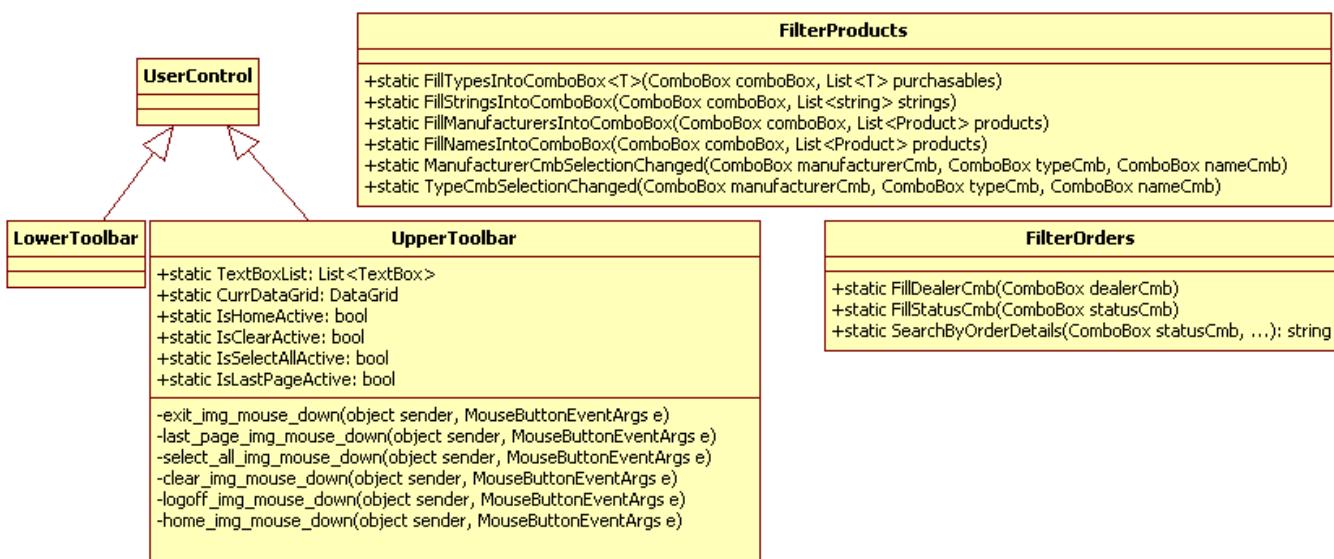
SmartBiz



דיאגרמת מחלקות עבור המחלקות הכלליות וה-UserControls בשכבה ה-PI

הערות:

1. בדיאגרמה זו ניתן לראות שתי המחלקות אשר מייצגות את סרגלי הכלים במערכת ירושת מהמחלקה `UserControl`.
2. המחלקות `FilterProducts` ו-`FilterOrders` הן מחלקות כלליות המכילות שיטות סטטיות בלבד.

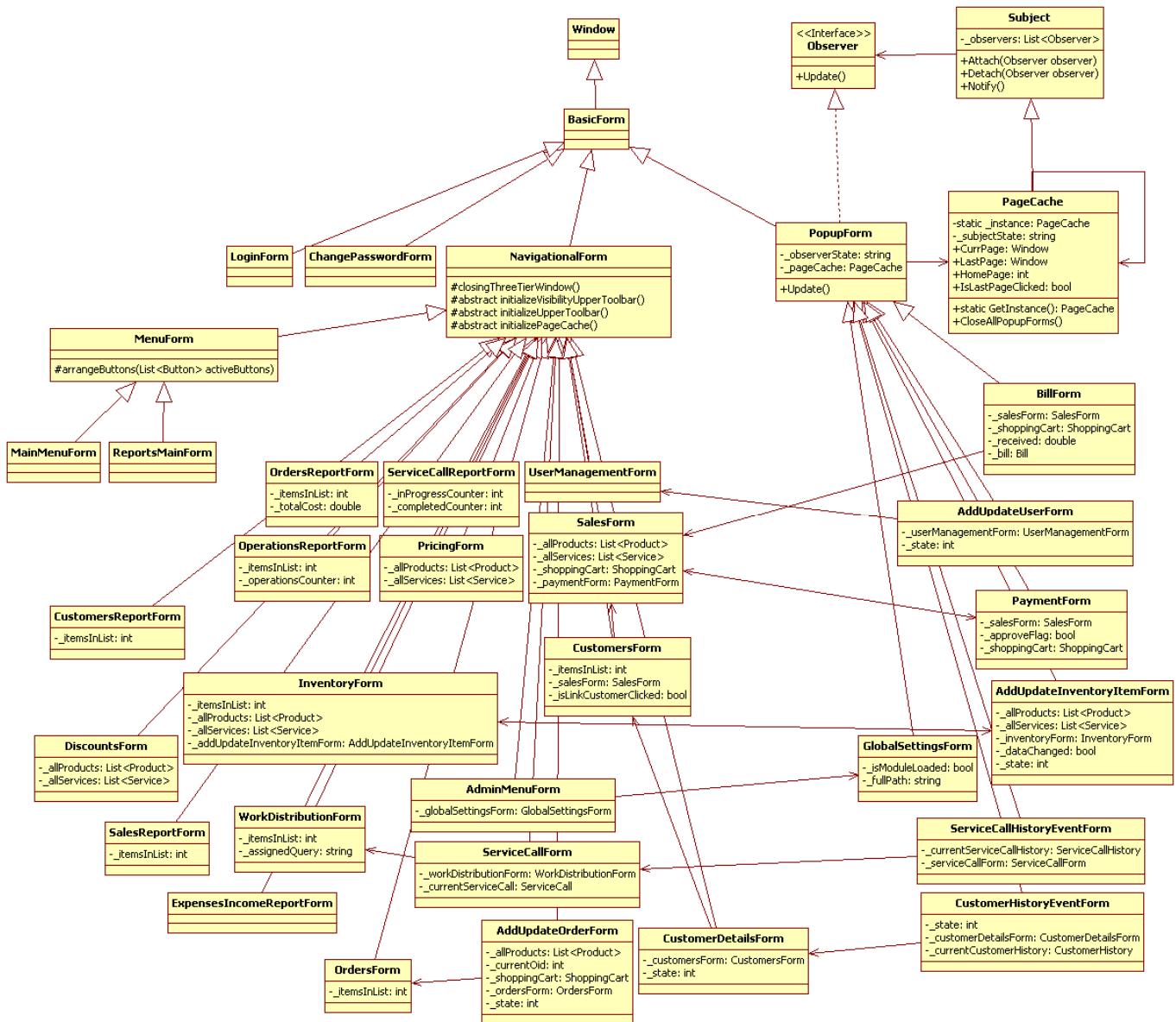


דיאגרמת מחלקות עבור הטפסים בשכבה ה-PI

הערות:

1. עקב ריבוי השיטות בכל מחלוקת של טופס קונקרטי, השיטות של מחלוקות אלה אינן מופיעות בדיאגרמה כדי לא להעמיס יתר על המידה.
2. בדיאגרמה זו ניתן לראות את היררכיית הירושה בין כל הטפסים במערכת.
3. בדיאגרמה זו ניתן לראות את הצלמידות בין טפסים מסוימים הקשורים ביניהם מבחינה לוגית, דבר זה בא לידי ביטוי כאשר מחלוקת אחת מחזיקה הפניה (Reference) למחלוקת השנייה.
4. כל הטפסים במערכת יורשים מחלוקת האבסטרקטית BasicForm היורשת מחלוקת האבסטרקטית Window.

SmartBiz

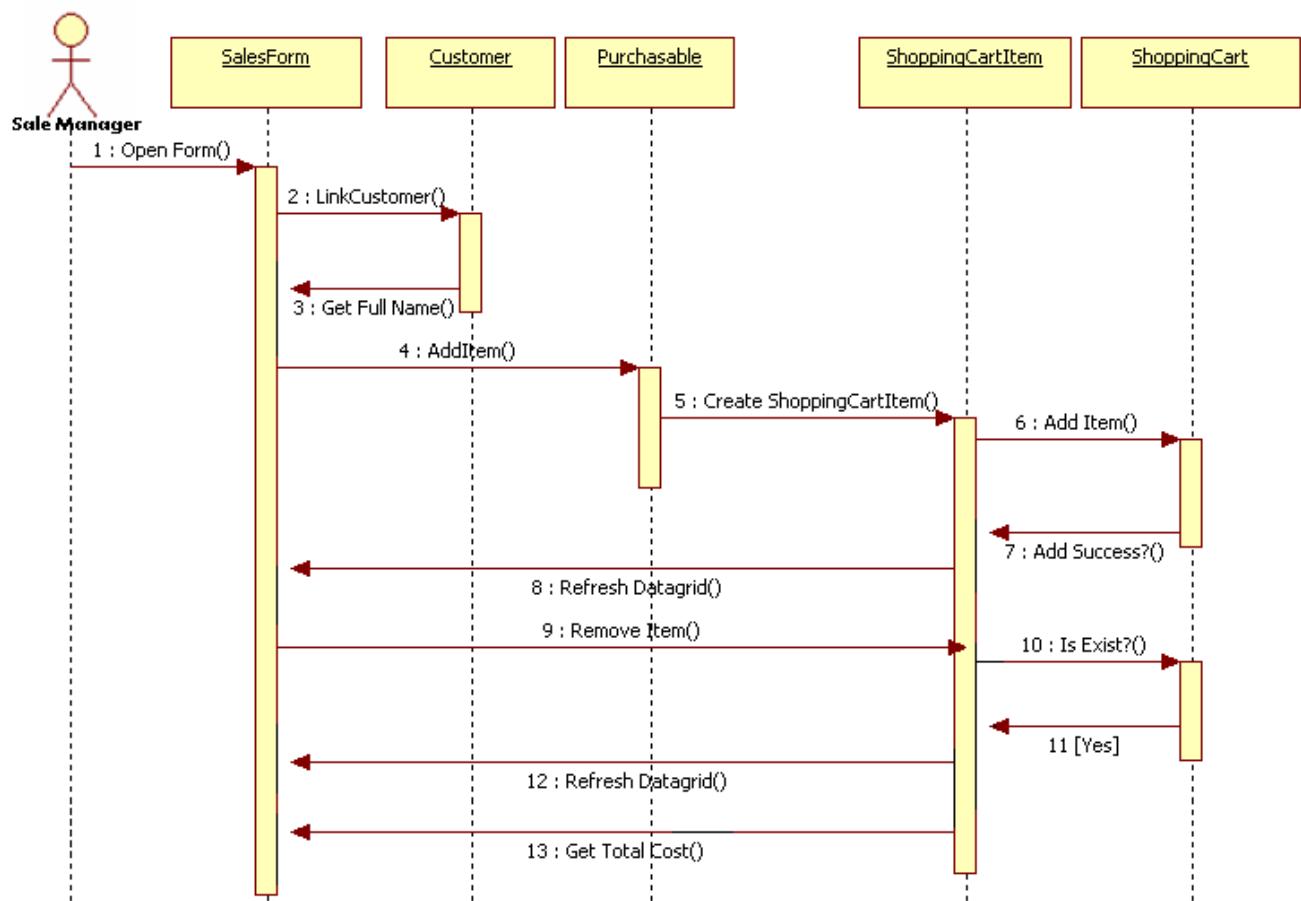


דיאגרמות רצף (Sequence Diagrams)

להלן מספר דיאגרמות רצף המתארות חלק מהתהליכיים המרכזיים במערכת. דיאגרמות אלה מתארות באופן כללי תהליכיים מרכזיים במערכת, "תכנים" שונים והבדלים בין דיאגרמות אלה לבין השימוש הסופי מכיוון שבפועל רצף האירועים ושליחת ההודעות בין האובייקטים השונים הינו מורכב יותר. לפיכך, דיאגרמות אלה מכילות אך ורק את האובייקטים וההודעות המשמעותיים ביותר בתהליך וזאת מטרה להמחיש את עיקרי הדברים ללא העמסת מידע יתר על המידה.

דיאגרמת רצף עבור הפקת הצעת מחיר ללקוח

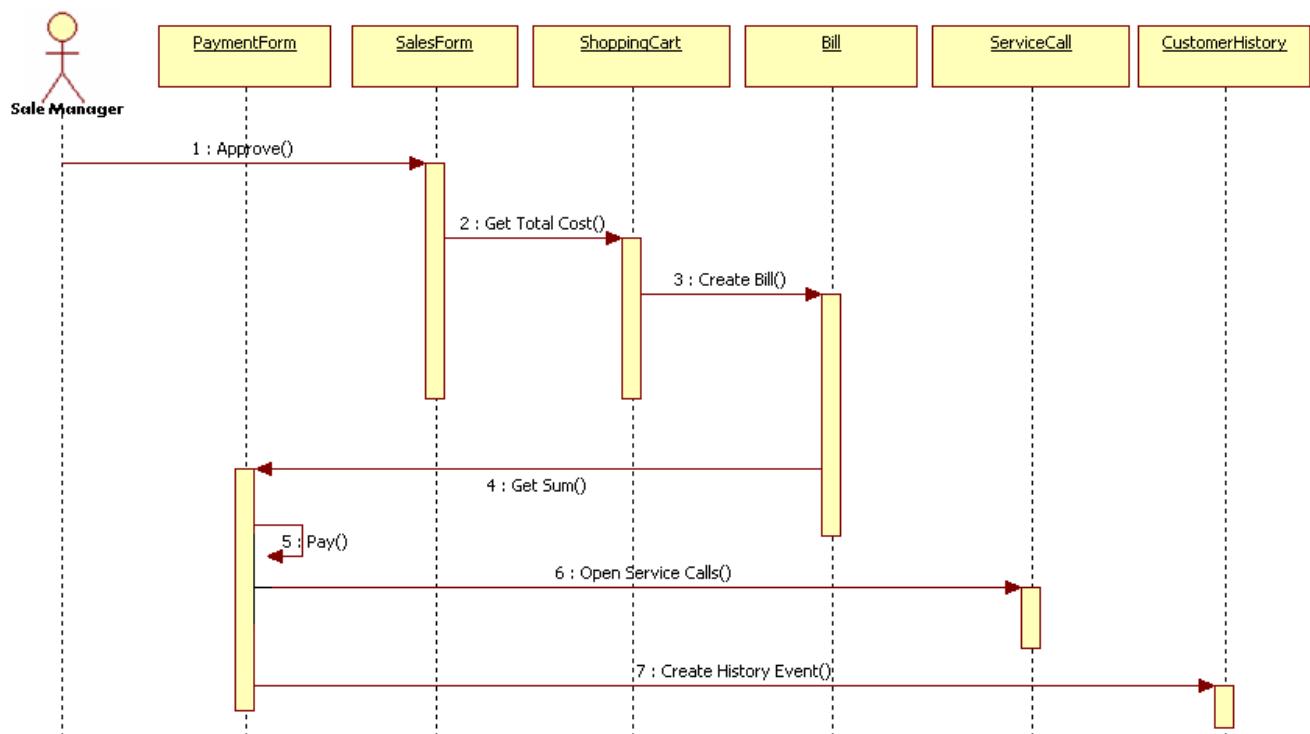
דיאגרמה זו מתארת את תהליך הפקת הצעת המחיר ללקוח. מנהל המכירות פותח את טופו המכירות, מקשר אליו את הלוקה המבוקש (אופציונלי), מוסיף פריטים (מוצרים ושירותים) לעגלת הקניות, מסיר פריטים מסוימים מעגלת הקניות. לבסוף, טופו המכירות יציג למנהל המכירות את תוכן העגלה ואת העלות הכוללת.



SmartBiz

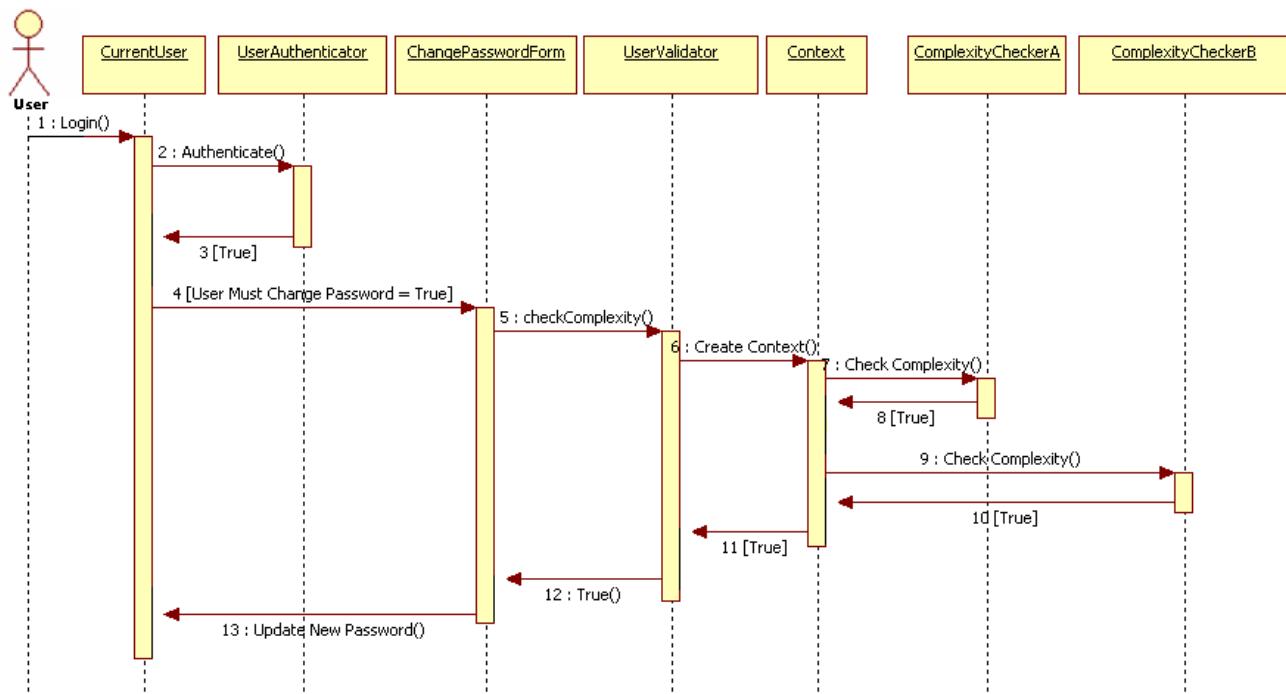
דיאגרמת רצף עבור ביצוע רכישה ע"י לקוחות

דיאגרמה זו מתרחשת את תהליך הרכישה. מנהל המכירות מאשר את הצעת המחיר ע"י להוצאה על כפטור ה-Approve, האובייקט Bill נוצר בהתאם לתוכן העגלה, טופס קבלת התקבולים נפתח, מנהל המכירות מזין את אמצעי התשלומים, ולבסוף אישור התשלום יגרום לפיתוחה קריאות השירות הרלוונטיות וליצירת אירוע היסטוריית לקוחות מתאים.



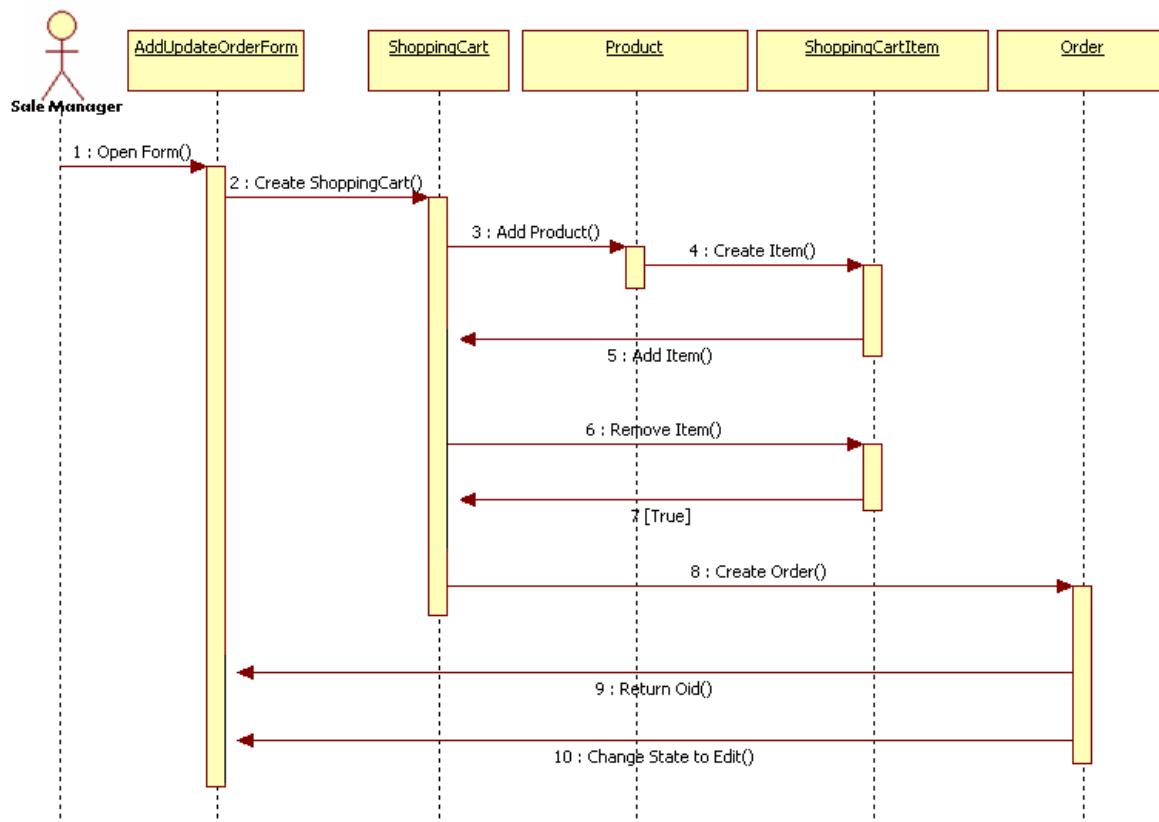
דיאגרמת רצף עבור החלפת סיסמה של משתמש

דיאגרמה זו מתארת את החלפת הסיסמה במערכת. משתמש כלשהו מקליד שם משתמש וסיסמה בטופס ההתחברות, נוצר אובייקט מסווג CurrentUser, מתבצע אימוט של שם המשתמש והסיסמה, אם ערכו של הדגל "המשתמש חייב לשנות את סיסמתו" הינו 'True' אז יפתח טופס שינוי הסיסמה, המשתמש מקליד את סיסמתו החדשה, מתבצעות מספר בדיקות מורכבות, ולבסוף אם הבדיקות עברו בהצלחה אז סיסמת המשתמש תटעדכן ב景德 הנתונים וטופס המסר הראשי ייפתח.



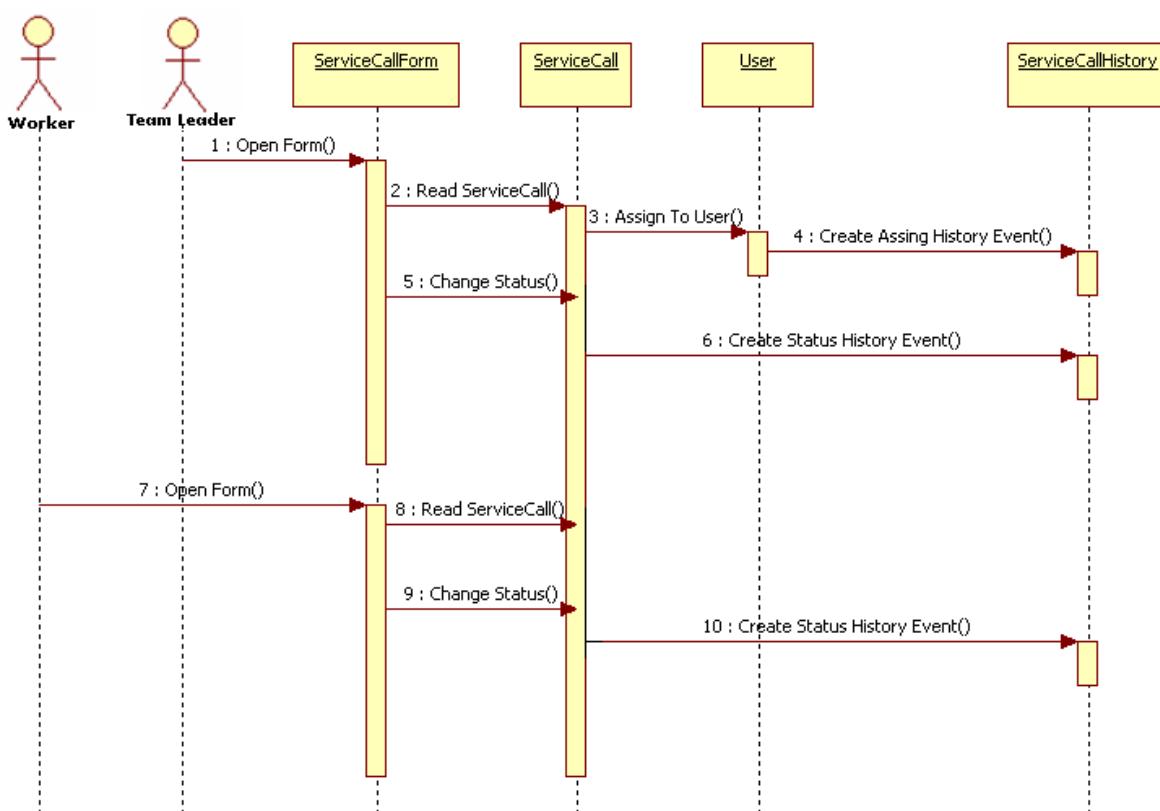
דיאגרמת רצף עבור ייצירת הזמנה חדשה במערכת

דיאגרמה זו מתארת את תהליך ייצירת הזמנה חדשה במערכת. מנהל המכירות פותח את הטופוף AddUpdateOrderForm, נוצר אובייקט מסווג ShoppingCart, מנהל המכירות מוסיף מוצרים אל עגלת הקניות תוך כדי שהוא קובע את עלות המוצר, מנהל המכירות מסיר מוצרים מעגלת הקניות, לחיצה על כפתור ה-Save יוצרת אובייקט מסווג Order, הטופוף AddUpdateOrderForm משנה את מצבו למצב עריכה כאשר כל פרטי הזמנה שנוצרה מופיעים בטופוף.



דיאגרמת רצף עבור הקצאה וטיפול בקריאה של שירות

דיאגרמה זו מתרחשת את תהליך ההקצאה והטיפול בקריאה של שירות לאחר שנפתחו במערכת. ראש הצוות פותח את הטופס ServiceCallFrom ע"י לחיצה כפולה על אחת הקריאה שטרם שייכו, ראש הצוות קורא את פרטי הקריאה המופיעים בטופס, ראש הצוות משנה את הSTATUS של הקריאה, נוצר אירוע אישור היסטורי מסוג שינוי STATUS. עובד הצוות פותח את הטופס ServiceCallFrom ע"י לחיצה כפולה על אחת הקריאה אשר שייכו אליו, עובד הצוות קורא את פרטי הקריאה המופיעים בטופס, מטפל בקריאה, עובד הצוות משנה את הSTATUS של הקריאה בהתאם לצורך, נוצר אירוע היסטורי מסוג שינוי STATUS.



File #0003243 belongs to Roei Daniel- do not distribute

נספח א – התקנת סביבת העבודה

הוראות התקנה

במסגרת הסדנה יעשה שימוש בוגון כלכלי פיתוח. באתר הקורס ניתן למצוא קישורים להורדת והתקנת כלים אלה. הכלים בהם יעשה שימוש הם :

- **Visual Studio 2012 Express Edition**
- **SQL Server 2012 Express Edition**
- **SQL Server 2012 Management Tool**
- **Star UML**

חברת מיקרוסופט מאפשרת לסטודנטים לקבל ללא תשלום כלים אלה בגרסת Professional באמצעות הצגת הוכחה שהיכנסם סטודנטים. (תעודת סטודנט)

התקנת התוכניות המופיעות בספר הלימוד

לכל פרק בספר יש תוכנית סיוכום לדוגמה שאינה מופיעה בספר עצמו, מפהת גודלה.

תוכניות אלה ניתנות להורדה מאתר האינטרנט של הספר שכותבו :

www.apress.com

שם עליכם לבחור בקישור CODE, ולחפש לפי שם הספר, כדי לקבל את דף הבית של הספר. תוכלו להוריד משם קובץ ZIP המכיל את התוכניות. במהלך הקריאה, הספר יפנה אותכם לעין בדוגמאות אלה.

File #0003243 belongs to Roei Daniel- do not distribute

נספח ב – קובץ מאמרים להעשרה

קובץ המאמרים באנגלית ממוספר במספרו המקורי

קובץ המאמרים בנספח זה הוא באנגלית

אנא הפכו את הספר לفتיחה אנגלית



Bibliography

[DeMarco79]: *Structured Analysis and System Specification*, Tom DeMarco, Yourdon Press Computing Series, 1979

[PageJones88]: *The Practical Guide to Structured Systems Design*, 2d. ed., Meilir Page-Jones, Yourdon Press Computing Series, 1988

Therefore the design in Figure 9-3 is probably better. It separates the two responsibilities into two separate interfaces³. This, at least, keeps the client applications from coupling the two responsibilities.

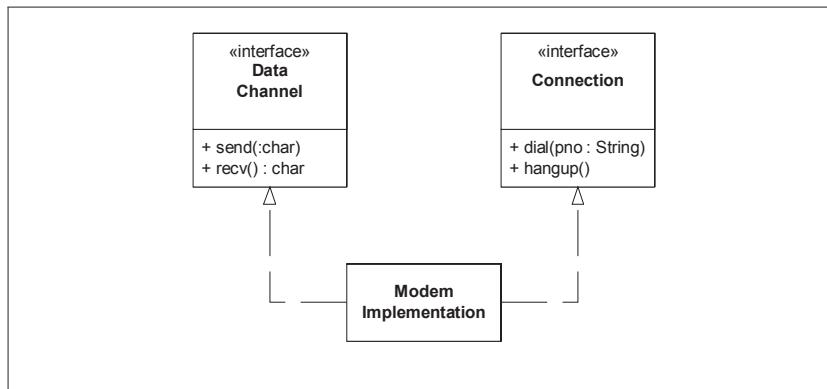


Figure 9-3
Separated Modem Interface

However, notice that I have recoupled the two responsibilities into a single `ModemImplementation` class. This is not desirable, but it may be necessary. There are often reasons, having to do with the details of the hardware or OS, that force us to couple things that we'd rather not couple. However, by separating their interfaces we have decoupled the concepts as far as the rest of the application is concerned.

We may view the `ModemImplementation` class as a kludge, or a wart; however, notice that all dependencies flow *away* from it. Nobody need depend upon this class. Nobody except `main` needs to know that it exists. Thus, we've put the ugly bit behind a fence. Its ugliness need not leak out and pollute the rest of the application.

Conclusion

The SRP is one of the simplest of the principle, and one of the hardest to get right. Joining responsibilities is something that we do naturally. Finding and separating those responsibilities from one another is much of what software design is really about. Indeed, the rest of the principles we will discuss come back to this issue in one way or another.

3. We'll see more of this in Chapter 13, when we study the Interface Segregation Principle (ISP).

A better design is to separate the two responsibilities into two completely different classes as shown in Figure 9-2. This design moves the computational portions of `Rectangle` into the `GeometricRectangle` class. Now changes made to the way rectangles are rendered cannot affect the `ComputationalGeometryApplication`.

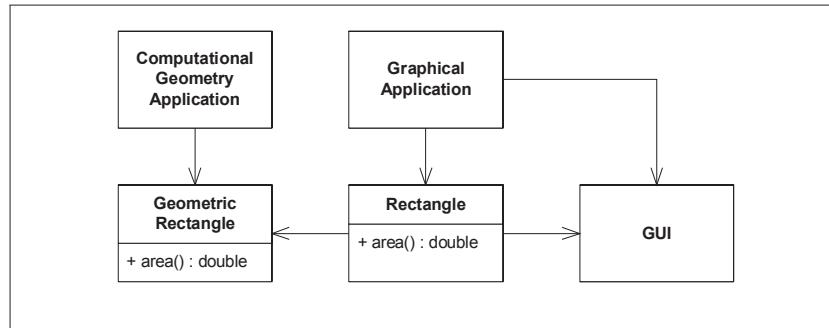


Figure 9-2
Separated Responsibilities

What is a Responsibility?

In the context of the Single Responsibility Principle (SRP) we define a responsibility to be “a reason for change.” If you can think of more than one motive for changing a class, then that class has more than one responsibility. This is sometimes hard to see. We are accustomed to thinking of responsibility in groups. For example, consider the `Modem` interface in Listing 9-1. Most of us will agree that this interface looks perfectly reasonable. The four functions it declares are certainly functions belonging to a modem.

Listing 9-1

```

Modem.java -- SRP Violation
interface Modem
{
    public void dial(String pno);
    public void hangup();
    public void send(char c);
    public char recv();
}

```

However, there are two responsibilities being shown here. The first responsibility is connection management. The second is data communication. The `dial` and `hangup` functions manage the connection of the modem, while the `send` and `recv` functions communicate data.

Should these two responsibilities be separated? Almost certainly they should. The two sets of functions have almost nothing in common. They’ll certainly change for different reasons. Moreover, they will be called from completely different parts of the applications that use them. Those different parts will change for different reasons as well.

Why was it important to separate these two responsibilities into separate classes? Because each responsibility is an axis of change. When the requirements change, that change will be manifest through a change in responsibility amongst the classes. If a class assumes more than one responsibility, then there will be more than one reason for it to change.

If a class has more than one responsibility, then the responsibilities become coupled. Changes to one responsibility may impair or inhibit the class' ability to meet the others. This kind of coupling leads to fragile designs that break in unexpected ways when changed.

For example, consider the design in Figure 9-1. The `Rectangle` class has two methods shown. One draws the rectangle on the screen, the other computes the area of the rectangle.

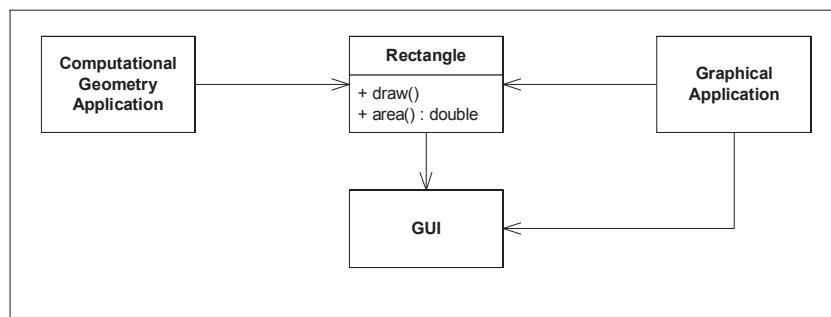


Figure 9-1
More than one responsibility

Two different applications use the `Rectangle` class. One application does computational geometry. It uses `Rectangle` to help it with the mathematics of geometric shapes. It never draws the rectangle on the screen. The other application is graphical in nature. It may also do some computational geometry, but it definitely draws the rectangle on the screen.

This design violates the SRP. The `Rectangle` class has two responsibilities. The first responsibility is to provide a mathematical model of the geometry of a rectangle. The second responsibility is to render the rectangle on a graphical user interface.

The violation of SRP causes several nasty problems. Firstly, we must include the `GUI` in the computational geometry application. If this were a C++ application, the `GUI` would have to be linked in, consuming link time, compile time, and memory footprint. In a Java application, the `.class` files for the `GUI` have to be deployed to the target platform.

Secondly, if a change to the `GraphicalApplication` causes the `Rectangle` to change for some reason, that change may force us to rebuild, retest, and redeploy the `ComputationalGeometryApplication`. If we forget to do this, that application may break in unpredictable ways.

9

SRP: The Single Responsibility Principle

None but Buddha himself must take the responsibility of giving out occult secrets...

— E. Cobham Brewer 1810–1897.
Dictionary of Phrase and Fable. 1898.

This principle was described in the work of Tom DeMarco¹ and Meilir Page-Jones². They called it *cohesion*. As we'll see in Chapter 21, we have a more specific definition of cohesion at the package level. However, at the class level the definition is similar.

SRP: The Single Responsibility Principle

***THERE SHOULD NEVER BE MORE THAN ONE REASON FOR A
CLASS TO CHANGE.***

Consider the bowling game from Chapter 6. For most of its development the Game class was handling two separate responsibilities. It was keeping track of the current frame, and it was calculating the score. In the end, RCM and RSK separated these two responsibilities into two classes. The Game kept the responsibility to keep track of frames, and the Scorer got the responsibility to calculate the score. (see page 85.)

1. [DeMarco79], p310
2. [PageJones88], Chapter 6, p82.

[OCP97]: The Open Closed Principle, Robert C. Martin...

[LSP97]: The Liskov Substitution Principle, Robert C. Martin

[DIP97]: The Dependency Inversion Principle, Robert C. Martin

[ISP97]: The Interface Segregation Principle, Robert C. Martin

[Granularity97]: Granularity, Robert C. Martin

[Stability97]: Stability, Robert C. Martin

[Liskov88]: Data Abstraction and Hierarchy...

[Martin99]: *Designing Object Oriented Applications using UML*, 2d. ed., Robert C. Martin, Prentice Hall, 1999.

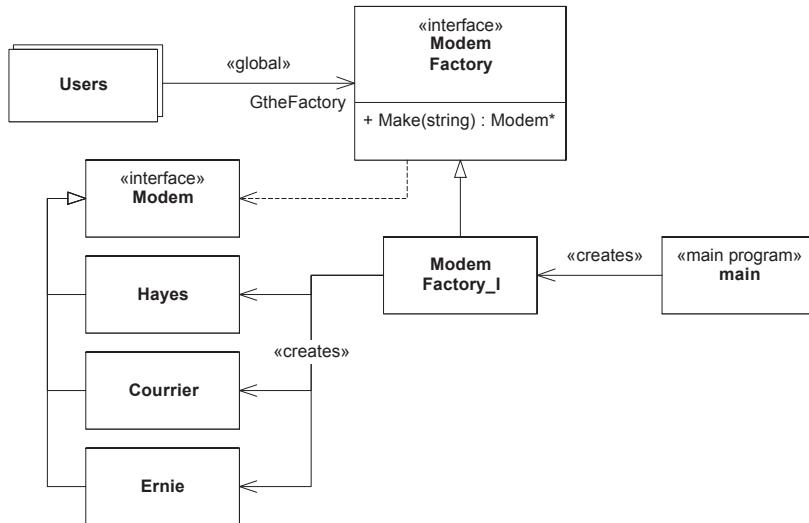


Figure 2-36
Abstract Factory

such architectures, and have been proven over time to be powerful aids in software architecture.

This has been an overview. There is much more to be said about the topic of OO architecture than can be said in the few pages of this chapter, indeed by foreshortening the topic so much, we have run the risk of doing the reader a disservice. It has been said that a little knowledge is a dangerous thing, and this chapter has provided a little knowledge. We strongly urge you to search out the books and papers in the citations of this chapter to learn more.

Bibliography

[Shaw96]: Patterns of Software Architecture (???), Garlan and Shaw, ...

[GOF96]: Design Patterns...

[OOSC98]: OOSC...

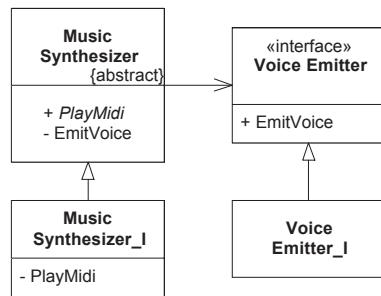


Figure 2-35
Hierarchy decoupled with Bridge

Abstract Factory

The DIP strongly recommends that modules not depend upon concrete classes. However, in order to create an instance of a class, you must depend upon the concrete class. ABSTRACTFACTORY is a pattern that allows that dependency upon the concrete class to exist in one, and only one, place.

Figure 2-36 shows how this is accomplished for the Modem example. All the users who wish to create modems use an interface called `ModemFactory`. A pointer to this interface is held in a global variable named `GtheFactory`. The users call the `Make` function passing in a string that uniquely defines the particular subclass of `Modem` that they want. The `Make` function returns a pointer to a `Modem` interface.

The `ModemFactory` interface is implemented by `ModemFactory_I`. This class is created by `main`, and a pointer to it is loaded into the `GtheFactory` global. Thus, no module in the system knows about the concrete modem classes except for `ModemFactory_I`, and no module knows about `ModemFactory_I` except for `main`.

Conclusion

This chapter has introduced the concept of object oriented architecture and defined it as the structure of classes and packages that keeps the software application flexible, robust, reusable, and developable. The principles and patterns presented here support



For example, consider a music synthesizer class. The base class translates MIDI input into a set of primitive `EmitVoice` calls that are implemented by a derived class. Note that the `EmitVoice` function of the derived class would be useful, in and of itself. Unfortunately it is inextricably bound to the `MusicSynthesizer` class and the `PlayMidi` function. There is no way to get at the `PlayVoice` method without dragging the base class around with it. Also, there is no way to create different implementations of the `PlayMidi` function that use the same `EmitVoice` function. In short, the hierarchy is just too coupled.

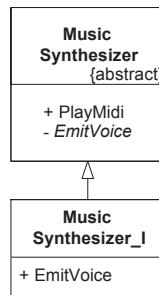


Figure 2-34
Badly coupled hierarchy

The BRIDGE pattern solves this problem by creating a strong separation between the interface and implementation. Figure 2-35 shows how this works. The `MusicSynthesizer` class contains an abstract `PlayMidi` function which is implemented by `MusicSynthesizer_I`. It calls the `EmitVoice` function that is implemented in `MusicSynthesizer` to delegate to the `VoiceEmitter` interface. This interface is implemented by `VoiceEmitter_I` and emits the necessary sounds.

Now it is possible to implement both `EmitVoice` and `PlayMidi` separately from each other. The two functions have been decoupled. `EmitVoice` can be called without bringing along all the `MusicSynthesizer` baggage, and `PlayMidi` can be implemented any number of different ways, while still using the same `EmitVoice` function.

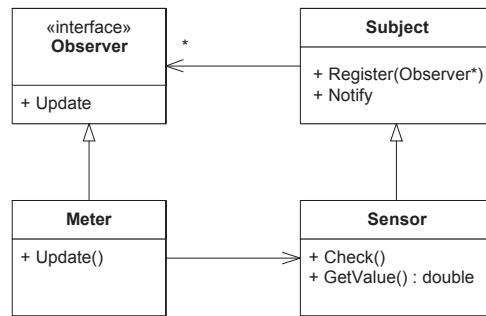


Figure 2-32
Observer Structure

that have been registered, calling `Update` on each. The `Update` message is caught by the **Meter** who uses it to read the new value of the **Sensor** and display it.

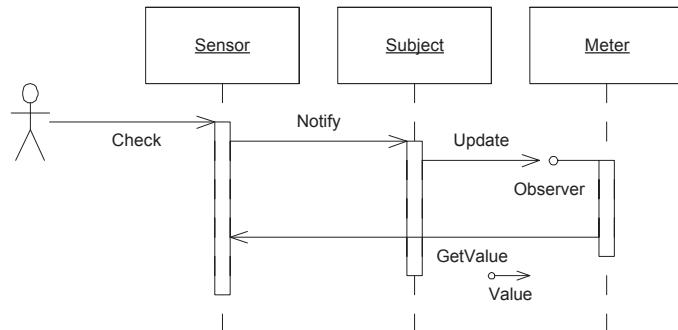


Figure 2-33

Bridge

One of the problems with implementing an abstract class with inheritance is that the derived class is so tightly coupled to the base class. This can lead to problems when other clients want to use the derived class functions without dragging along the baggage of the base hierarchy.

Adapter

When inserting an abstract interface is infeasible because the server is third party software, or is so heavily depended upon that it cannot easily be changed, an ADAPTER can be used to bind the abstract interface to the server. See Figure 2-31.

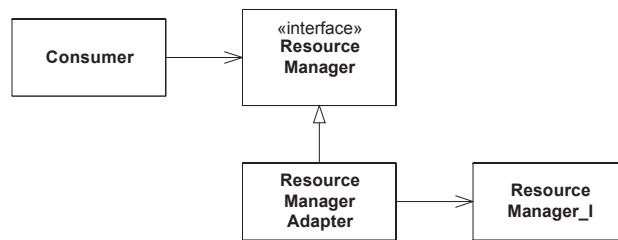


Figure 2-31
Adapter

The adapter is an object that implements the abstract interface to delegate to the server. Every method of the adpater simply translates and then delegates.

Observer

It often occurs that one element of a design needs to take some form of action when another element in the design discovers that an event has occurred. However, we frequently don't want the detector to know about the actor.

Consider the case of a meter that shows the status of a sensor. Every time the sensor changes its reading we want the meter to display the new value. However, we don't want the sensor to know anything about the meter.

We can address this situation with an OBSERVER, see Figure 2-32. The Sensor derives from a class named Subject, and Meter derives from an interface called Observer. Subject contains a list of Observers. This list is loaded by the Register method of Subject. In order to be told of events, our Meter must register with the Subject base class of the Sensor.

Figure 2-33 describes the dynamics of the collaboration. Some entity passes control to the Sensor who determines that its reading has changed. The Sensor calls Notify on its Subject. The Subject then cycles through all the Observers

Patterns of Object Oriented Architecture

When following the principles described above to create object oriented architectures, one finds that one repeats the same structures over and over again. These repeating structures of design and architecture are known as design patterns¹.

The essential definition of a design pattern is a well worn and known good solution to a common problem. Design patterns are definitively not new. Rather they are old techniques that have shown their usefulness over a period of many years.

Some common design patterns are described below. These are the patterns that you will come across while reading through the case studies later in the book.

It should be noted that the topic of Design Patterns cannot be adequately covered in a single chapter of a single book. Interested readers are strongly encouraged to read [GOF96].

Abstract Server

When a client depends directly on a server, the DIP is violated. Changes to the server will propagate to the client, and the client will be unable to easily use similar servers. This can be rectified by inserting an abstract interface between the client and the server as shown in Figure 2-30.

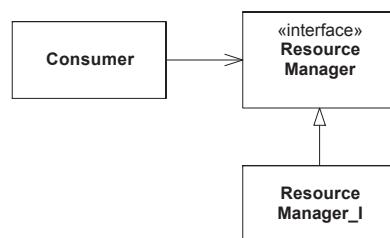
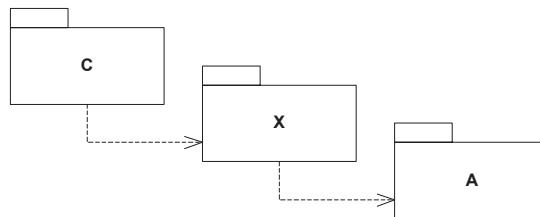


Figure 2-30
Abstract Server

The abstract interface becomes a “hinge point” upon which the design can flex. Different implementations of the server can be bound to an unsuspecting client.

1. [GOF96]

**Figure 2-29**

What do we put X on the A vs I Graph?

We can determine where we want Package X, by looking at where we don't want it to go. The upper right corner of the AI graph represents packages that are highly abstract and that nobody depends upon. This is the zone of uselessness. Certainly we don't want X to live there. On the other hand, the lower left point of the AI graph represents packages that are concrete and have lots of incoming dependencies. This point represents the worst case for a package. Since the elements there are concrete, they cannot be extended the way abstract entities can; and since they have lots of incoming dependencies, the change will be very painful. This is the zone of pain, and we certainly don't want our package to live there.

Maximizing the distance between these two zones gives us a line called the *main sequence*. We'd like our packages to sit on this line if at all possible. A position on this line means that the package is abstract in proportion to its incoming dependencies and is concrete in proportion to its outgoing dependencies. In other words, the classes in such a package are conforming to the DIP.

Distance Metrics. This leaves us one more set of metrics to examine. Given the A and I values of any package, we'd like to know how far that package is from the main sequence.

$$D \text{ Distance. } D = \frac{|A + I - 1|}{\sqrt{2}}. \text{ This ranges from } [0, \sim 0.707].$$

D' Normalized Distance. $D' = |A + I - 1|$. This metric is much more convenient than D since it ranges from [0,1]. Zero indicates that the package is directly on the main sequence. One indicates that the package is as far away as possible from the main sequence.

These metrics measure object oriented architecture. They are imperfect, and reliance upon them as the sole indicator of a sturdy architecture would be foolhardy. However, they can be, and have been, used to help measure the dependency structure of an application.

Nc Number of classes in the package.

Na Number of abstract classes in the package. Remember, an abstract class is a class with at least one pure interface, and cannot be instantiated.

A Abstractness. $A = \frac{Na}{Nc}$

The A metric has a range of $[0,1]$, just like the I metric. A value of zero means that the package contains no abstract classes. A value of one means that the package contains nothing but abstract classes.

The I vs A graph. The SAP can now be restated in terms of the I and A metrics: I should increase as A decreases. That is, concrete packages should be instable while abstract packages should be stable. We can plot this graphically on the A vs I graph. See Figure 2-28.

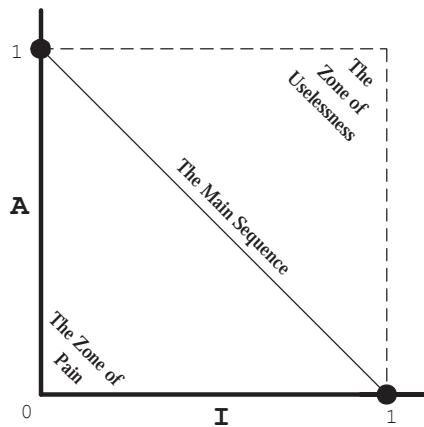


Figure 2-28
The A vs I graph.

It seems clear that packages should appear at either of the two black dots on Figure 2-28. Those at the upper left are completely abstract and very stable. Those at the lower right are completely concrete and very unstable. This is just the way we like it. However what about package X in Figure 2-29? Where should it go?

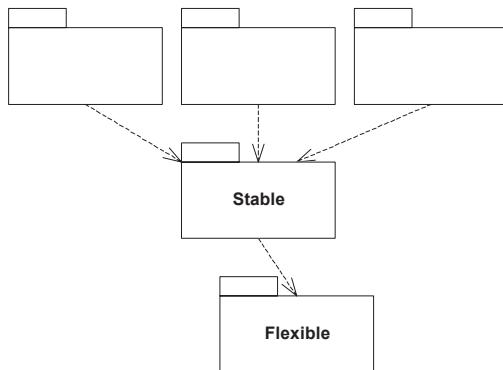


Figure 2-27
Violation of SDP.

We can envision the packages structure of our application as a set of interconnected packages with instable packages at the top, and stable packages on the bottom. In this view, all dependencies point downwards.

Those packages at the top are instable and flexible. But those at the bottom are very difficult to change. And this leads us to a dilemma: Do we want packages in our design that are hard to change?

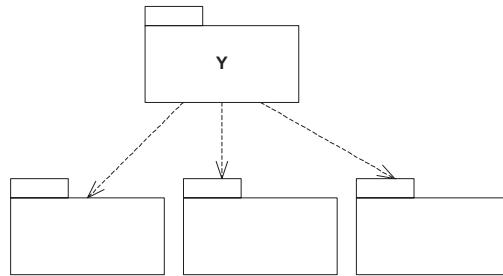
Clearly, the more packages that are hard to change, the less flexible our overall design will be. However, there is a loophole we can crawl through. The highly stable packages at the bottom of the dependency network may be very difficult to change, but according to the OCP they do not have to be difficult to extend!

If the stable packages at the bottom are also highly abstract, then they can be easily extended. This means that it is possible to compose our application from instable packages that are easy to change, and stable packages that are easy to extend. This is a good thing.

Thus, the SAP is just a restatement of the DIP. It states the packages that are the most depended upon (i.e. stable) should also be the most abstract. But how do we measure abstractness?

The Abstractness Metrics. We can derive another trio of metrics to help us calculate abstractness.

1. [Stability97]

**Figure 2-26**

Y is instable.

Ce Efferent Coupling. The number of classes outside the package that classes inside the package depend upon. (i.e. outgoing dependencies)

I Instability. $I = \frac{Ce}{Ca + Ce}$. This is a metric that has the range: [0,1].

If there are no outgoing dependencies, then I will be zero and the package is stable. If there are no incoming dependencies then I will be one and the package is unstable.

Now we can rephrase the SDP as follows: “Depend upon packages whose I metric is lower than yours.”

Rationale. Should all software be stable? One of the most important attributes of well designed software is ease of change. Software that is flexible in the presence of changing requirements is thought well of. Yet that software is unstable by our definition. Indeed, we greatly desire that portions of our software be unstable. We want certain modules to be easy to change so that when requirements drift, the design can respond with ease.

Figure 2-27 shows how the SDP can be violated. **Flexible** is a package that we intend to be easy to change. We want **Flexible** to be unstable. However, some engineer, working in the package named **Stable**, hung a dependency upon **Flexible**. This violates the SDP since the I metric for **Stable** is much lower than the I metric for **Flexible**. As a result, **Flexible** will no longer be easy to change. A change to **Flexible** will force us to deal with **Stable** and all its dependents.

The Stable Abstractions Principle (SAP)¹

Stable packages should be abstract packages.

tion for a very very long time. Thus stability has nothing directly to do with frequency of change. The penny is not changing, but it is hard to think of it as stable.

Stability is related to the amount of work required to make a change. The penny is not stable because it requires very little work to topple it. On the other hand, a table is very stable because it takes a considerable amount of effort to turn it over.

How does this relate to software? There are many factors that make a software package hard to change. Its size, complexity, clarity, etc. We are going to ignore all those factors and focus upon something different. One sure way to make a software package difficult to change, is to make lots of other software packages depend upon it. A package with lots of incoming dependencies is very stable because it requires a great deal of work to reconcile any changes with all the dependent packages.

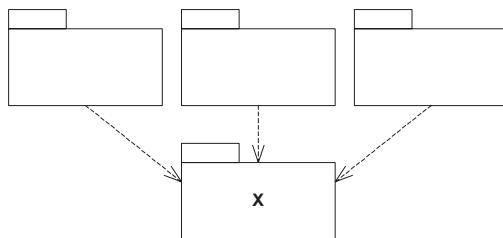


Figure 2-25
X is a stable package

Figure 2-25 shows X: a stable package. This package has three packages depending upon it, and therefore it has three good reasons not to change. We say that it is *responsible* to those three packages. On the other hand, X depends upon nothing, so it has no external influence to make it change. We say it is *independent*.

Figure 2-26, on the other hand, shows a very unstable package. Y has no other packages depending upon it; we say that it is irresponsible. Y also has three packages that it depends upon, so changes may come from three external sources. We say that Y is dependent.

Stability Metrics. We can calculate the stability of a package using a trio of simple metrics.

Ca Afferent Coupling. The number of classes outside the package that depend upon classes inside the package. (i.e. incoming dependencies)

ages. Interfaces are very often included in the package that uses them, rather than in the package that implements them.

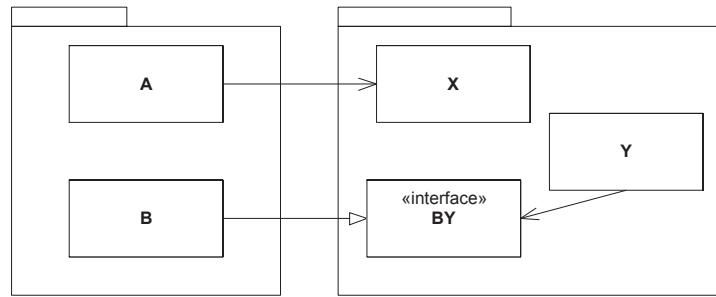
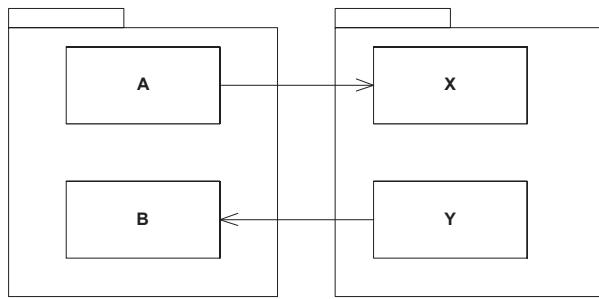


Figure 2-24

The Stable Dependencies Principle (SDP)¹

Depend in the direction of stability.

Though this seems to be an obvious principle, there is quite a bit we can say about it. Stability is not always well understood.

Stability. What is meant by stability? Stand a penny on its side. Is it stable in that position? Likely you'd say not. However, unless disturbed, it will remain in that posi-

1. [Stability97]

Figure 2-23 shows how to break the cycle by adding a new package. The classes that `CommError` needed are pulled out of `GUI` and placed in a new package named `MessageManager`. Both `GUI` and `CommError` are made to depend upon this new package.

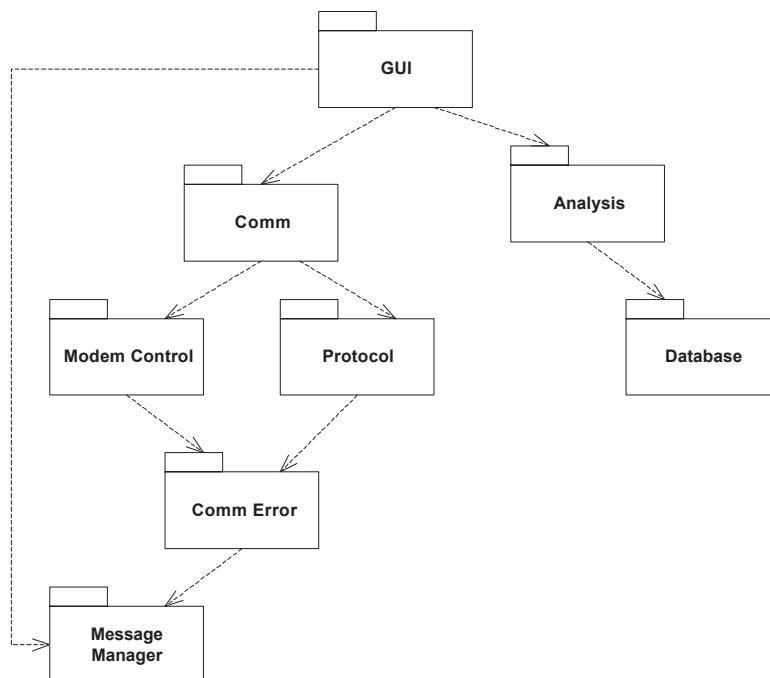


Figure 2-23

This is an example of how the package structure tends to jitter and shift during development. New packages come into existence, and classes move from old package to new packages, to help break cycles.

Figure 2-24 shows a before and after picture of the other technique for breaking cycles. Here we see two packages that are bound by a cycle. Class A depends upon class X, and class Y depends upon class B. We break the cycle by inverting the dependency between Y and B. This is done by adding a new interface, BY, to B. This interface has all the methods that Y needs. Y uses this interface and B implements it.

Notice the placement of `BY`. It is placed in the package with the class that uses it. This is a pattern that you will see repeated throughout the case studies that deal with pack-

A Cycle Creeps In. But now lets say that I am an engineer working on the `CommError` package. I have decided that I need to display a message on the screen. Since the screen is controlled by the `GUI`, I send a message to one of the `GUI` objects to get my message up on the screen. This means that I have made `CommError` dependent upon `GUI`. See Figure 2-22.

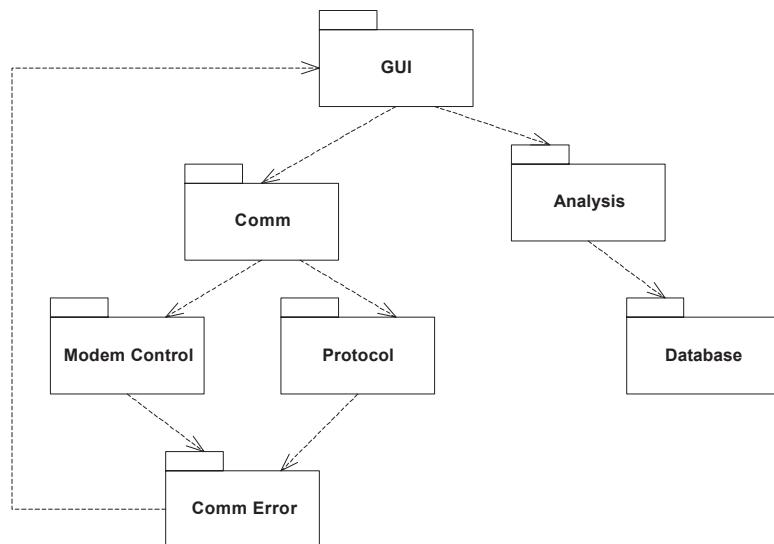


Figure 2-22
A cycle has been added.

Now what happens when the guys who are working on `Protocol` want to release their package. They have to build their test suite with `CommError`, `GUI`, `Comm`, `ModemControl`, `Analysis`, and `Database`! This is clearly disastrous. The workload of the engineers has been increased by an abhorrent amount, due to one single little dependency that got out of control.

This means that someone needs to be watching the package dependency structure with regularity, and breaking cycles wherever they appear. Otherwise the transitive dependencies between modules will cause every module to depend upon every other module.

Breaking a Cycle. Cycles can be broken in two ways. The first involves creating a new package, and the second makes use of the DIP and ISP.

together those classes that are related. Thus, engineers will find that their changes are directed into just a few package. Once those changes are made, they can release those packages to the rest of the project.

Before they can do this release, however, they must test that the package works. To do that, they must compile and build it with all the packages that it depends upon. Hopefully this number is small.

Consider Figure 2-21. Astute readers will recognize that there are a number of flaws in the architecture. The DIP seems to have been abandoned, and along with it the OCP. The GUI depends directly upon the communications package, and apparently is responsible for transporting data to the analysis package. Yuk.

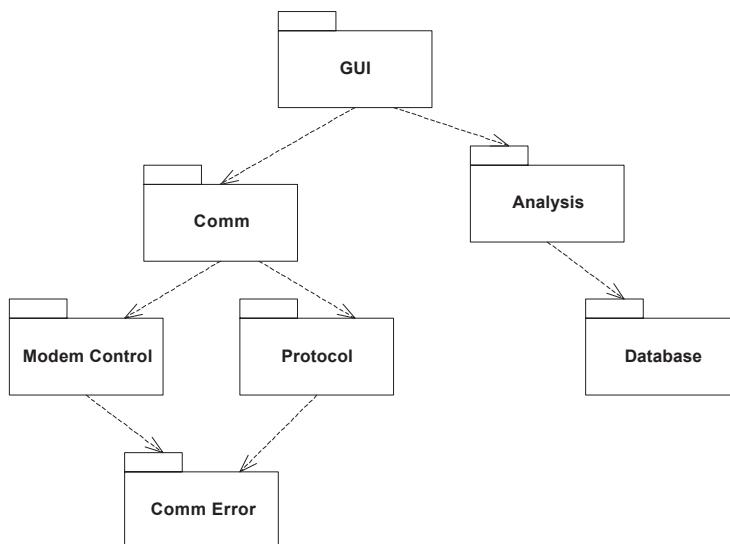


Figure 2-21
Acyclic Package Network

Still, let's use this rather ugly structure for some examples. Consider what would be required to release the `Protocol` package. The engineers would have to build it with the latest release of the `CommError` package, and run their tests. `Protocol` has no other dependencies, so no other package is needed. This is nice. We can test and release with a minimal amount of work.

age must verify that they work with the new package -- even if nothing they used within the package actually changed.

We frequently experience this when our OS vendor releases a new operating system. We have to upgrade sooner or later, because the vendor will not support the old version forever. So even though nothing of interest to us changed in the new release, we must go through the effort of upgrading and revalidating.

The same can happen with packages if classes that are not used together are grouped together. Changes to a class that I don't care about will still force a new release of the package, and still cause me to go through the effort of upgrading and revalidating.

Tension between the Package Cohesion Principles

These three principles are mutually exclusive. They cannot simultaneously be satisfied. That is because each principle benefits a different group of people. The REP and CRP makes life easy for reusers, whereas the CCP makes life easier for maintainers. The CCP strives to make packages as large as possible (after all, if all the classes live in just one package, then only one package will ever change). The CRP, however, tries to make packages very small.

Fortunately, packages are not fixed in stone. Indeed, it is the nature of packages to shift and jitter during the course of development. Early in a project, architects may set up the package structure such that CCP dominates and development and maintenance is aided. Later, as the architecture stabilizes, the architects may refactor the package structure to maximize REP and CRP for the external reusers.

The Package Coupling Principles.

The next three packages govern the interrelationships between packages. Applications tend to be large networks of interrelated packages. The rules that govern these interrelationship are some of the most important rules in object oriented architecture.

The Acyclic Dependencies Principle (ADP)¹

The dependencies between packages must not form cycles.

Since packages are the granule of release, they also tend to focus manpower. Engineers will typically work inside a single package rather than working on dozens. This tendency is amplified by the package cohesion principles, since they tend to group

1. [Granularity97]

The Release Reuse Equivalency Principle (REP)¹

The granule of reuse is the granule of release.

A reusable element, be it a component, a class, or a cluster of classes, cannot be reused unless it is managed by a release system of some kind. Users will be unwilling to use the element if they are forced to upgrade every time the author changes it. Thus, even though the author has released a new version of his reusable element, he must be willing to support and maintain older versions while his customers go about the slow business of getting ready to upgrade. Thus, clients will refuse to reuse an element unless the author promises to keep track of version numbers, and maintain old versions for awhile.

Therefore, one criterion for grouping classes into packages is reuse. Since packages are the unit of release, they are also the unit of reuse. Therefore architects would do well to group reusable classes together into packages.

The Common Closure Principle (CCP)²

Classes that change together, belong together.

A large development project is subdivided into a large network of interrelated packages. The work to manage, test, and release those packages is non-trivial. The more packages that change in any given release, the greater the work to rebuild, test, and deploy the release. Therefore we would like to minimize the number of packages that are changed in any given release cycle of the product.

To achieve this, we group together classes that we think will change together. This requires a certain amount of precience since we must anticipate the kinds of changes that are likely. Still, when we group classes that change together into the same packages, then the package impact from release to release will be minimized.

The Common Reuse Principle (CRP)³

Classes that aren't reused together should not be grouped together.

A dependency upon a package is a dependency upon everything within the package. When a package changes, and its release number is bumped, all clients of that pack-

1. [Granularity97]

2. [Granularity97]

3. [Granularity97]

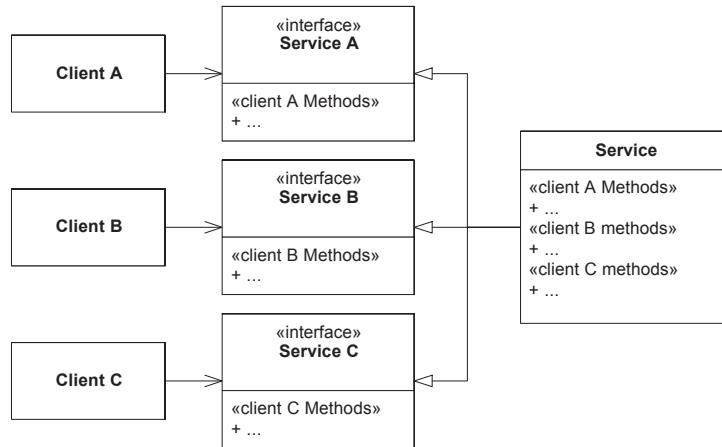


Figure 2-20
Segregated Interfaces

face that wish to access methods of the new interface, can query the object for that interface as shown in the following code.

```

void Client(Service* s)
{
    if (NewService* ns = dynamic_cast<NewService*>(s) )
    {
        // use the new service interface
    }
}
  
```

As with all principles, care must be taken not to overdo it. The specter of a class with hundreds of different interfaces, some segregated by client and other segregated by version, would be frightening indeed.

Principles of Package Architecture

Classes are a necessary, but insufficient, means of organizing a design. The larger granularity of packages are needed to help bring order. But how do we choose which classes belong in which packages. Below are three principles known as the *Package Cohesion Principles*, that attempt to help the software architect.

Figure 2-19 shows a class with many clients, and one large interface to serve them all. Note that whenever a change is made to one of the methods that ClientA calls, ClientB and ClientC may be affected. It may be necessary to recompile and redeploy them. This is unfortunate.

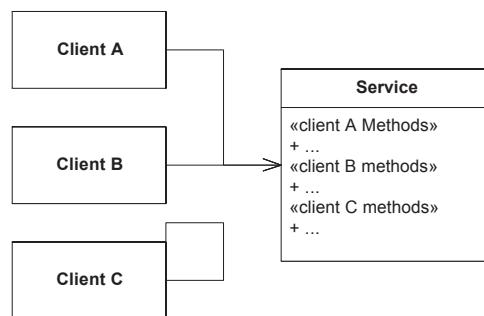


Figure 2-19
Fat Service with Integrated Interfaces

A better technique is shown in Figure 2-20. The methods needed by each client are placed in special interfaces that are specific to that client. Those interfaces are multiply inherited by the Service class, and implemented there.

If the interface for ClientA needs to change, ClientB and ClientC will remain unaffected. They will not have to be recompiled or redeployed.

What does Client Specific Mean? The ISP does not recommend that every class that uses a service have its own special interface class that the service must inherit from. If that were the case, the service would depend upon each and every client in a bizarre and unhealthy way. Rather, clients should be categorized by their type, and interfaces for each type of client should be created.

If two or more different client types need the same method, the method should be added to both of their interfaces. This is neither harmful nor confusing to the client.

Changing Interfaces. When object oriented applications are maintained, the interfaces to existing classes and components often change. There are times when these changes have a huge impact and force the recompilation and redeployment of a very large part of the design. This impact can be mitigated by adding new interfaces to existing objects, rather than changing the existing interface. Clients of the old inter-

Substrates such as COM enforce this principle, at least between components. The only visible part of a COM component is its abstract interface. Thus, in COM, there is little escape from the DIP.

Mitigating Forces. One motivation behind the DIP is to prevent you from depending upon volatile modules. The DIP makes the assumption that anything concrete is volatile. While this is frequently so, especially in early development, there are exceptions. For example, the `string.h` standard C library is very concrete, but is not at all volatile. Depending upon it in an ANSI string environment is not harmful. Likewise, if you have tried and true modules that are concrete, but not volatile, depending upon them is not so bad. Since they are not likely to change, they are not likely to inject volatility into your design.

Take care however. A dependency upon `string.h` could turn very ugly when the requirements for the project forced you to change to UNICODE characters. Non-volatility is not a replacement for the substitutability of an abstract interface.

Object Creation. One of the most common places that designs depend upon concrete classes is when those designs create instances. By definition, you cannot create instances of abstract classes. Thus, to create an instance, you must depend upon a concrete class.

Creation of instances can happen all through the architecture of the design. Thus, it might seem that there is no escape and that the entire architecture will be littered with dependencies upon concrete classes. However, there is an elegant solution to this problem named ABSTRACTFACTORY¹ -- a design pattern that we'll be examining in more detail towards the end of this chapter.

The Interface Segregation Principle (ISP)²

Many client specific interfaces are better than one general purpose interface

The ISP is another one of the enabling technologies supporting component substrates such as COM. Without it, components and classes would be much less useful and portable.

The essence of the principle is quite simple. If you have a class that has several clients, rather than loading the class with all the methods that the clients need, create specific interfaces for each client and multiply inherit them into the class.

1. [GOF96] p??

2. [ISP97]



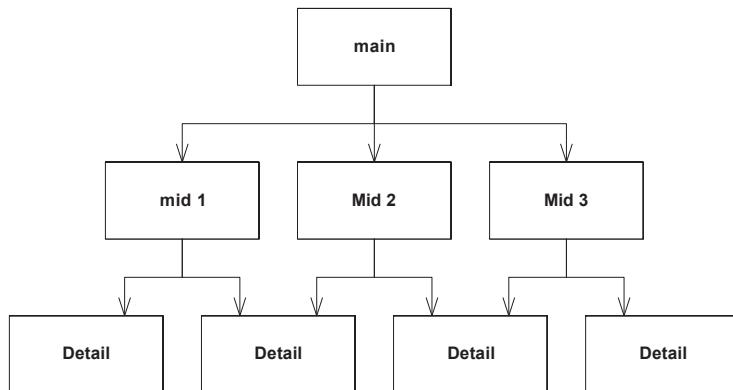


Figure 2-17
Dependency Structure of a Procedural Architecture

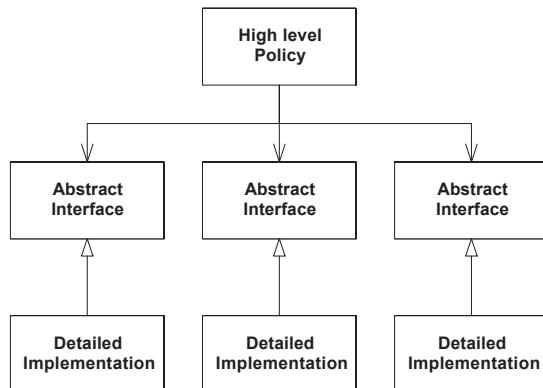


Figure 2-18
Dependency Structure of an Object Oriented Architecture

Clearly such a restriction is draconian, and there are mitigating circumstances that we will explore momentarily. But, as much as is feasible, the principle should be followed. The reason is simple, concrete things change a lot, abstract things change much less frequently. Moreover, abstractions are “hinge points”, they represent the places where the design can bend or be extended, without themselves being modified (OCP).

Listing 2-6**Ugly fix for LSP violation**

```

        assert(e.GetMajorAxis() == 3);
    }
    else
        throw NotAnEllipse(e);
}

```

Careful examination of Listing 2-6 will show it to be a violation of the OCP. Now, whenever some new derivative of `Ellipse` is created, this function will have to be checked to see if it should be allowed to operate upon it. *Thus, violations of LSP are latent violations of OCP.*

The Dependency Inversion Principle (DIP)¹

Depend upon Abstractions. Do not depend upon concretions.

If the OCP states the goal of OO architecture, the DIP states the primary mechanism. Dependency Inversion is the strategy of depending upon interfaces or abstract functions and classes, rather than upon concrete functions and classes. This principle is the enabling force behind component design, COM, CORBA, EJB, etc.

Procedural designs exhibit a particular kind of dependency structure. As Figure 2-17 shows, this structure starts at the top and points down towards details. High level modules depend upon lower level modules, which depend upon yet lower level modules, etc..

A little thought should expose this dependency structure as intrinsically weak. The high level modules deal with the high level policies of the application. These policies generally care little about the details that implement them. Why then, must these high level modules directly depend upon those implementation modules?

An object oriented architecture shows a very different dependency structure, one in which the majority of dependencies point towards abstractions. Moreover, the modules that contain detailed implementation are no longer depended upon, rather they *depend themselves* upon abstractions. Thus the dependency upon them has been *inverted*. See Figure 2-18.

Depending upon Abstractions. The implication of this principle is quite simple. Every dependency in the design should target an interface, or an abstract class. No dependency should target a concrete class.

1. [DIP97]

`Circle` does not honor the implied contract of `Ellipse`, it is not substitutable and violates the LSP.

Making the contract explicit is an avenue of research followed by Bertrand Meyer. He has invented a language named Eiffel in which contracts are explicitly stated for each method, and explicitly checked at each invocation. Those of us who are not using Eiffel, have to make do with simple assertions and comments.

To state the contract of a method, we declare what must be true before the method is called. This is called the precondition. If the precondition fails, the results of the method are undefined, and the method ought not be called. We also declare what the method guarantees will be true once it has completed. This is called the postcondition. A method that fails its postcondition should not return.

Restating the LSP once again, this time in terms of the contracts, a derived class is substitutable for its base class if:

1. Its preconditions are no stronger than the base class method.
2. Its postconditions are no weaker than the base class method.

Or, in other words, derived methods should *expect no more and provide no less*.

Repercussions of LSP Violation. Unfortunately, LSP violations are difficult to detect until it is too late. In the `Circle/Ellipse` case, everything worked fine until some client came along and discovered that the implicit contract had been violated.

If the design is heavily used, the cost of repairing the LSP violation may be too great to bear. It might not be economical to go back and change the design, and then rebuild and retest all the existing clients. Therefore the solution will likely be to put into an if/else statement in the client that discovered the violation. This if/else statement checks to be sure that the `Ellipse` is actually an `Ellipse` and not a `Circle`. See Listing 2-6.

Listing 2-6

Ugly fix for LSP violation

```
void f(Ellipse& e)
{
    if (typeid(e) == typeid(Ellipse))
    {
        Point a(-1,0);
        Point b(1,0);
        e.SetFoci(a,b);
        e.SetMajorAxis(3);
        assert(e.GetFocusA() == a);
        assert(e.GetFocusB() == b);
```

Still, if we ignore the slight overhead in space, we can make `Circle` behave properly by overriding its `SetFoci` method to ensure that both foci are kept at the same value. See Listing 2-5. Thus, either focus will act as the center of the circle, and the major axis will be its diameter.

Listing 2-5

Keeping the Circle Foci coincident.

```
void Circle::SetFoci(const Point& a, const Point& b)
{
    itsFocusA = a;
    itsFocusB = a;
}
```

Clients Ruin Everything. Certainly the model we have created is self consistent. An instance of `Circle` will obey all the rules of a circle. There is nothing you can do to it to make it violate those rules. So too for `Ellipse`. The two classes form a nicely consistent model, even if `Circle` has one too many data elements.

However, `Circle` and `Ellipse` do not live alone in a universe by themselves. They cohabit that universe with many other entities, and provide their public interfaces to those entities. Those interfaces imply a contract. The contract may not be explicitly stated, but it is there nonetheless. For example, users of `Ellipse` have the right to expect the following code fragment to succeed:

```
void f(Ellipse& e)
{
    Point a(-1,0);
    Point b(1,0);
    e.SetFoci(a,b);
    e.SetMajorAxis(3);
    assert(e.GetFocusA() == a);
    assert(e.GetFocusB() == b);
    assert(e.GetMajorAxis() == 3);
}
```

In this case the function expects to be working with an `Ellipse`. As such, it expects to be able to set the foci, and major axis, and then verify that they have been properly set. If we pass an instance of `Ellipse` into this function, it will be quite happy.

However, if we pass an instance of `Circle` into the function, it will fail rather badly.

If we were to make the contract of `Ellipse` explicit, we would see a postcondition on the `SetFoci` that guaranteed that the input values got copied to the member variables, and that the major axis variable was left unchanged. Clearly `Circle` violates this guarantee because it ignores the second input variable of `SetFoci`.

Design by Contract. Restating the LSP, we can say that, in order to be substitutable, the contract of the base class must be honored by the derived class. Since

The Circle/Ellipse Dilemma. Most of us learn, in high school math, that a circle is just a degenerate form of an ellipse. All circles are ellipses with coincident foci. This is-a relationship tempts us to model circles and ellipses using inheritance as shown in Figure 2-15.

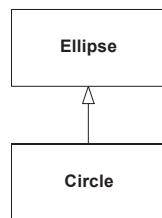


Figure 2-15
Circle / Ellipse Dilemma

While this satisfies our conceptual model, there are certain difficulties. A closer look at the declaration of `Ellipse` in Figure 2-16 begins to expose them. Notice that `Ellipse` has three data elements. The first two are the foci, and the last is the length of the major axis. If `Circle` inherits from `Ellipse`, then it will inherit these data variables. This is unfortunate since `Circle` really only needs two data elements, a center point and a radius.

Ellipse
- itsFocusA : Point - itsFocusB : Point - itsMajorAxis : double
+ Circumference() : double + Area() : double + GetFocusA() : Point + GetFocusB() : Point + GetMajorAxis() : double + GetMinorAxis() : double + SetFocus(a:Point, b:Point) + SetMajorAxis(double)

Figure 2-16
Declaration of Ellipse

The Liskov Substitution Principle (LSP)¹

Subclasses should be substitutable for their base classes.

This principle was coined by Barbara Liskov² in her work regarding data abstraction and type theory. It also derives from the concept of Design by Contract (DBC) by Bertrand Meyer³.

The concept, as stated above, is depicted in Figure 2-14. Derived classes should be substitutable for their base classes. That is, a user of a base class should continue to function properly if a derivative of that base class is passed to it.

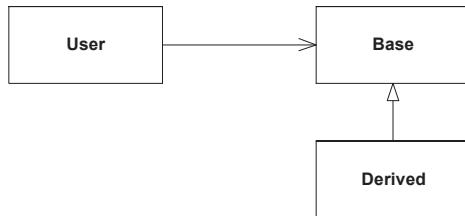


Figure 2-14
LSP schema.

In other words, if some function `User` takes an argument of type `Base`, then as shown in Listing 2-4, it should be legal to pass in an instance of `Derived` to that function.

Listing 2-4

User, Based, Derived, example.

```

void User(Base& b);

Derived d;
User(d);
  
```

This may seem obvious, but there are subtleties that need to be considered. The canonical example is the Circle/Ellipse dilemma.

1. [LSP97]

2. [Liskov88]

3. [OOSC98]



Listing 2-2**LogOn has been closed for modification**

```

virtual void Dial(const string& pno) = 0;
virtual void Send(char) = 0;
virtual char Recv() = 0;
virtual void Hangup() = 0;
};

void LogOn(Modem& m,
            string& pno, string& user, string& pw)
{
    m.Dial(pno);
    // you get the idea.
}

```

Static Polymorphism. Another technique for conforming to the OCP is through the use of templates or generics. Listing 2-3 shows how this is done. The LogOn function can be extended with many different types of modems without requiring modification.

Listing 2-3**Logon is closed for modification through static polymorphism**

```

template <typename MODEM>
void LogOn(MODEM& m,
            string& pno, string& user, string& pw)
{
    m.Dial(pno);
    // you get the idea.
}

```

Architectural Goals of the OCP. By using these techniques to conform to the OCP, we can create modules that are extensible, without being changed. This means that, with a little forethought, we can add new features to existing code, without changing the existing code and by only adding new code. This is an ideal that can be difficult to achieve, but you will see it achieved, several times, in the case studies later on in this book.

Even if the OCP cannot be fully achieved, even partial OCP compliance can make dramatic improvements in the structure of an application. It is always better if changes do not propagate into existing code that already works. If you don't have to change working code, you aren't likely to break it.

Of course this is not the worst attribute of this kind of design. Programs that are designed this way tend to be littered with similar if/else or switch statement. Every time anything needs to be done to the modem, a switch statement if/else chain will need to select the proper functions to use. When new modems are added, or modem policy changes, the code must be scanned for all these selection statements, and each must be appropriately modified.

Worse, programmers may use local optimizations that hide the structure of the selection statements. For example, it might be that the function is exactly the same for Hayes and Courier modems. Thus we might see code like this:

```
if (modem.type == Modem::ernie)
    SendErnie((Ernie&)modem, c);
else
    SendHayes((Hayes&)modem, c);
```

Clearly, such structures make the system much harder to maintain, and are very prone to error.

As an example of the OCP, consider Figure 2-13. Here the LogOn function depends only upon the Modem interface. Additional modems will not cause the LogOn function to change. Thus, we have created a module that can be extended, with new modems, without requiring modification. See Listing 2-2.

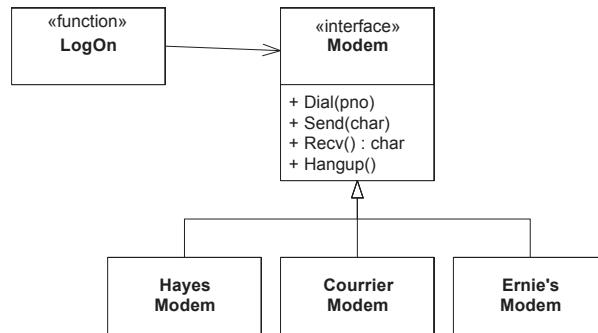


Figure 2-13

Listing 2-2

LogOn has been closed for modification

```
class Modem
{
public:
```



This may sound contradictory, but there are several techniques for achieving the OCP on a large scale. All of these techniques are based upon abstraction. Indeed, *abstraction is the key to the OCP*. Several of these techniques are described below.

Dynamic Polymorphism. Consider Listing 2-1. the `LogOn` function must be changed every time a new kind of modem is added to the software. Worse, since each different type of modem depends upon the `Modem::Type` enumeration, each modem must be recompiled every time a new kind of modem is added.

Listing 2-1

`Logon`, must be modified to be extended.

```
struct Modem
{
    enum Type {hayes, courier, ernie} type;
};

struct Hayes
{
    Modem::Type type;
    // Hayes related stuff
};

struct Courier
{
    Modem::Type type;
    // Courier related stuff
};

struct Ernie
{
    Modem::Type type;
    // Ernie related stuff
};

void LogOn(Modem& m,
           string& pno, string& user, string& pw)
{
    if (m.type == Modem::hayes)
        DialHayes((Hayes&)m, pno);
    else if (m.type == Modem::courrier)
        DialCourier((Courier&)m, pno);
    else if (m.type == Modem::ernie)
        DialErnie((Ernie&)m, pno)
    // ...you get the idea
}
```

project. If our designs are failing due to the constant rain of changing requirements, it is our designs that are at fault. We must somehow find a way to make our designs resilient to such changes and protect them from rotting.

Dependency Management

What kind of changes cause designs to rot? Changes that introduce new and unplanned for dependencies. Each of the four symptoms mentioned above is either directly, or indirectly caused by improper dependencies between the modules of the software. It is the dependency architecture that is degrading, and with it the ability of the software to be maintained.

In order to forestall the degradation of the dependency architecture, the dependencies between modules in an application must be managed. This management consists of the creation of dependency firewalls. Across such firewalls, dependencies do not propagate.

Object Oriented Design is replete with principles and techniques for building such firewalls, and for managing module dependencies. It is these principles and techniques that will be discussed in the remainder of this chapter. First we will examine the principles, and then the techniques, or design patterns, that help maintain the dependency architecture of an application.

Principles of Object Oriented Class Design

The Open Closed Principle (OCP)¹

A module should be open for extension but closed for modification.

Of all the principles of object oriented design, this is the most important. It originated from the work of Bertrand Meyer². It means simply this: We should write our modules so that they can be extended, without requiring them to be modified. In other words, we want to be able to change what the modules do, without changing the source code of the modules.

1. [OCP97]

2. [OO98]

Such software causes managers and customers to suspect that the developers have lost control of their software. Distrust reigns, and credibility is lost.

Immobility. Immobility is the inability to reuse software from other projects or from parts of the same project. It often happens that one engineer will discover that he needs a module that is similar to one that another engineer wrote. However, it also often happens that the module in question has too much baggage that it depends upon. After much work, the engineers discover that the work and risk required to separate the desirable parts of the software from the undesirable parts are too great to tolerate. And so the software is simply rewritten instead of reused.

Viscosity. Viscosity comes in two forms: viscosity of the design, and viscosity of the environment. When faced with a change, engineers usually find more than one way to make the change. Some of the ways preserve the design, others do not (i.e. they are hacks.) When the design preserving methods are harder to employ than the hacks, then the viscosity of the design is high. It is easy to do the wrong thing, but hard to do the right thing.

Viscosity of environment comes about when the development environment is slow and inefficient. For example, if compile times are very long, engineers will be tempted to make changes that don't force large recompiles, even though those changes are not optimal from a design point of view. If the source code control system requires hours to check in just a few files, then engineers will be tempted to make changes that require as few check-ins as possible, regardless of whether the design is preserved.

These four symptoms are the tell-tale signs of poor architecture. Any application that exhibits them is suffering from a design that is rotting from the inside out. But what causes that rot to take place?

Changing Requirements

The immediate cause of the degradation of the design is well understood. The requirements have been changing in ways that the initial design did not anticipate. Often these changes need to be made quickly, and may be made by engineers who are not familiar with the original design philosophy. So, though the change to the design works, it somehow violates the original design. Bit by bit, as the changes continue to pour in, these violations accumulate until malignancy sets in.

However, we cannot blame the drifting of the requirements for the degradation of the design. We, as software engineers, know full well that requirements change. Indeed, most of us realize that the requirements document is the most volatile document in the

ally the sheer effort required to make even the simplest of changes to the application becomes so high that the engineers and front line managers cry for a redesign project.

Such redesigns rarely succeed. Though the designers start out with good intentions, they find that they are shooting at a moving target. The old system continues to evolve and change, and the new design must keep up. The warts and ulcers accumulate in the new design before it ever makes it to its first release. On that fateful day, usually much later than planned, the morass of problems in the new design may be so bad that the designers are already crying for another redesign.

Symptoms of Rotting Design

There are four primary symptoms that tell us that our designs are rotting. They are not orthogonal, but are related to each other in ways that will become obvious. they are: rigidity, fragility, immobility, and viscosity.

Rigidity. Rigidity is the tendency for software to be difficult to change, even in simple ways. Every change causes a cascade of subsequent changes in dependent modules. What begins as a simple two day change to one module grows into a multi-week marathon of change in module after module as the engineers chase the thread of the change through the application.

When software behaves this way, managers fear to allow engineers to fix non-critical problems. This reluctance derives from the fact that they don't know, with any reliability, when the engineers will be finished. If the managers turn the engineers loose on such problems, they may disappear for long periods of time. The software design begins to take on some characteristics of a roach motel -- engineers check in, but they don't check out.

When the manager's fears become so acute that they refuse to allow changes to software, official rigidity sets in. Thus, what starts as a design deficiency, winds up being adverse management policy.

Fragility. Closely related to rigidity is fragility. Fragility is the tendency of the software to break in many places every time it is changed. Often the breakage occurs in areas that have no conceptual relationship with the area that was changed. Such errors fill the hearts of managers with foreboding. Every time they authorize a fix, they fear that the software will break in some unexpected way.

As the fragility becomes worse, the probability of breakage increases with time, asymptotically approaching 1. Such software is impossible to maintain. Every fix makes it worse, introducing more problems than are solved.

Design Principles and Design Patterns

Robert C. Martin
www.objectmentor.com

What is software architecture? The answer is multilayered. At the highest level, there are the architecture patterns that define the overall shape and structure of software applications¹. Down a level is the architecture that is specifically related to the purpose of the software application. Yet another level down resides the architecture of the modules and their interconnections. This is the domain of design patterns², packages, components, and classes. It is this level that we will concern ourselves with in this chapter.

Our scope in this chapter is quite limited. There is much more to be said about the principles and patterns that are exposed here. Interested readers are referred to [Martin99].

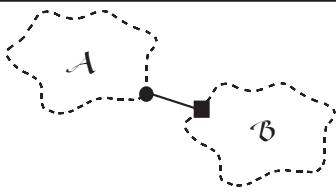
Architecture and Dependencies

What goes wrong with software? The design of many software applications begins as a vital image in the minds of its designers. At this stage it is clean, elegant, and compelling. It has a simple beauty that makes the designers and implementers itch to see it working. Some of these applications manage to maintain this purity of design through the initial development and into the first release.

But then something begins to happen. The software starts to rot. At first it isn't so bad. An ugly wart here, a clumsy hack there, but the beauty of the design still shows through. Yet, over time as the rotting continues, the ugly festering sores and boils accumulate until they dominate the design of the application. The program becomes a festering mass of code that the developers find increasingly hard to maintain. Eventu-

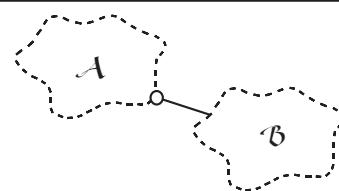
1. [Shaw96]

2. [GOF96]



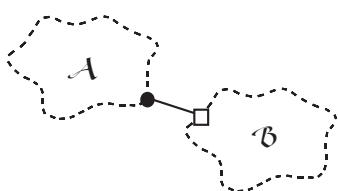
Contains - by value. This indicates that A and B have identical lifetimes. When A is destroyed, B will be destroyed too.

```
class A
{
    private:
        B itsB;
};
```



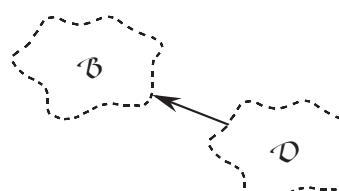
Uses. This indicates that the name of class B is used within the source code of class A.

```
class A
{
    public:
        void F(const B&);
};
```



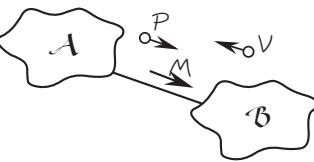
Contains - by reference. This is used to indicate that A and B have dissimilar lifetimes. B may outlive A.

```
class A
{
    private:
        B* itsB;
};
```



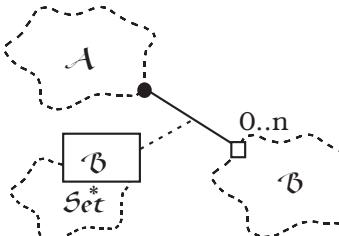
Inheritance. This indicates that B is a public base class of D.

```
class D : public B { ... };
```



Invoking a member function.

```
void A::F(B& theB)
{
    P p;
    V v = theB.M(p);
}
```



Using a container class. In this case a template for a "Set".

```
class A
{
    private:
        Set<B*> itsBs;
};
```



Listing 10 (Continued)

```
RTTI that does not violate the open-closed Principle.
{
    for (Iterator<Shape*>i(ss); i; i++)
    {
        Square* s = dynamic_cast<Square*>(*i);
        if (s)
            s->Draw();
    }
}
```

ever, nothing changes in Listing 10 when a new derivative of Shape is created. Thus, Listing 10 does not violate the open-closed principle.

As a general rule of thumb, if a use of RTTI does not violate the open-closed principle, it is safe.

Conclusion

There is much more that could be said about the open-closed principle. In many ways this principle is at the heart of object oriented design. Conformance to this principle is what yields the greatest benefits claimed for object oriented technology; i.e. reusability and maintainability. Yet conformance to this principle is not achieved simply by using an object oriented programming language. Rather, it requires a dedication on the part of the designer to apply abstraction to those parts of the program that the designer feels are going to be subject to change.

This article is an extremely condensed version of a chapter from my new book: *The Principles and Patterns of OOD*, to be published soon by Prentice Hall. In subsequent articles we will explore many of the other principles of object oriented design. We will also study various design patterns, and their strengths and weaknesses with regard to implementation in C++. We will study the role of Booch's class categories in C++, and their applicability as C++ namespaces. We will define what "cohesion" and "coupling" mean in an object oriented design, and we will develop metrics for measuring the quality of an object oriented design. And, after that, many other interesting topics.

Listing 9

RTTI violating the open-closed principle.

```
class Shape {}

class Square : public Shape
{
    private:
        Point itsTopLeft;
        double itsSide;
    friend DrawSquare(Square*);
};

class Circle : public Shape
{
    private:
        Point itsCenter;
        double itsRadius;
    friend DrawCircle(Circle*);
};

void DrawAllShapes(Set<Shape*>& ss)
{
    for (Iterator<Shape*>i(ss); i; i++)
    {
        Circle* c = dynamic_cast<Circle*>(*i);
        Square* s = dynamic_cast<Square*>(*i);
        if (c)
            DrawCircle(c);
        else if (s)
            DrawSquare(s);
    }
}
```

Listing 10

RTTI that does not violate the open-closed Principle.

```
class Shape
{
    public:
        virtual void Draw() const = 0;
};

class Square : public Shape
{
    // as expected.
};

void DrawSquaresOnly(Set<Shape*>& ss)
```

One complaint I could make about Listing 8 is that the modification of the time is not atomic. That is, a client can change the `minutes` variable without changing the `hours` variable. This may result in inconsistent values for a `Time` object. I would prefer it if there were a single function to set the time that took three arguments, thus making the setting of the time atomic. But this is a very weak argument.

It would not be hard to think of other conditions for which the `public` nature of these variables causes some problems. In the long run, however, there is no *overriding* reason to make these variables `private`. I still consider it bad *style* to make them `public`, but it is probably not bad *design*. I consider it bad style because it is very cheap to create the appropriate inline member functions; and the cheap cost is almost certainly worth the protection against the slight risk that issues of closure will crop up.

Thus, in those rare cases where the open-closed principle is not violated, the proscription of `public` and `protected` variables depends more upon style than on substance.

No Global Variables -- Ever.

The argument against global variables is similar to the argument against `public` member variables. No module that depends upon a global variable can be closed against any other module that might write to that variable. Any module that uses the variable in a way that the other modules don't expect, will break those other modules. It is too risky to have many modules be subject to the whim of one badly behaved one.

On the other hand, in cases where a global variable has very few dependents, or cannot be used in an inconsistent way, they do little harm. The designer must assess how much closure is sacrificed to a global and determine if the convenience offered by the global is worth the cost.

Again, there are issues of style that come into play. The alternatives to using globals are usually very inexpensive. In those cases it is bad style to use a technique that risks even a tiny amount of closure over one that does not carry such a risk. However, there are cases where the convenience of a global is significant. The global variables `cout` and `cin` are common examples. In such cases, if the open-closed principle is not violated, then the convenience may be worth the style violation.

RTTI is Dangerous.

Another very common proscription is the one against `dynamic_cast`. It is often claimed that `dynamic_cast`, or any form of run time type identification (RTTI) is intrinsically dangerous and should be avoided. The case that is often cited is similar to Listing 9 which clearly violates the open-closed principle. However Listing 10 shows a similar program that uses `dynamic_cast`, but does not violate the open-closed principle.

The difference between these two is that the first, Listing 9, *must* be changed whenever a new type of `Shape` is derived. (Not to mention that it is just downright silly). How-

In OOD, we expect that the methods of a class are not closed to changes in the member variables of that class. However we *do* expect that any other class, including subclasses *are closed* against changes to those variables. We have a name for this expectation, we call it: *encapsulation*.

Now, what if you had a member variable that you knew would never change? Is there any reason to make it `private`? For example, Listing 7 shows a class `Device` that has a `bool status` variable. This variable contains the status of the last operation. If that operation succeeded, then `status` will be `true`; otherwise it will be `false`.

Listing 7

```
non-const public variable
class Device
{
    public:
        bool status;
};
```

We know that the type or meaning of this variable is never going to change. So why not make it `public` and let client code simply examine its contents? If this variable really never changes, and if all other clients obey the rules and only query the contents of `status`, then the fact that the variable is `public` does no harm at all. However, consider what happens if even one client takes advantage of the writable nature of `status`, and changes its value. Suddenly, this one client could affect every other client of `Device`. This means that it is impossible to close any client of `Device` against changes to this one misbehaving module. This is probably far too big a risk to take.

On the other hand, suppose we have the `Time` class as shown in Listing 8. What is the harm done by the public member variables in this class? Certainly they are very unlikely to change. Moreover, it does not matter if any of the client modules make changes to the variables, the variables are supposed to be changed by clients. It is also very unlikely that a derived class might want to trap the setting of a particular member variable. So is any harm done?

Listing 8

```
class Time
{
    public:
        int hours, minutes, seconds;
        Time& operator=(int seconds);
        Time& operator+=(int seconds);
        bool operator< (const Time&);
        bool operator> (const Time&);
        bool operator==(const Time&);
        bool operator!=(const Time&);
};
```

Listing 6 (Continued)

Table driven type ordering mechanism

```

        if ((argOrd > 0) && (thisOrd > 0))
            done = true;
        }
        else // table entry == 0
            done = true;
    }
    return thisOrd < argOrd;
}

```

The only item that is not closed against the order of the various `Shapes` is the table itself. And that table can be placed in its own module, separate from all the other modules, so that changes to it do not affect any of the other modules.

Extending Closure Even Further.

This isn't the end of the story. We have managed to close the `Shape` hierarchy, and the `DrawAllShapes` function against ordering that is dependent upon the type of the shape. However, the `Shape` derivatives are not closed against ordering policies that have nothing to do with shape types. It seems likely that we will want to order the drawing of shapes according to some higher level structure. A complete exploration of these issues is beyond the scope of this article; however the ambitious reader might consider how to address this issue using an abstract `OrderedObject` class contained by the class `OrderedShape`, which is derived from both `Shape` and `OrderedObject`.

Heuristics and Conventions

As mentioned at the beginning of this article, the open-closed principle is the root motivation behind many of the heuristics and conventions that have been published regarding OOD over the years. Here are some of the more important of them.

Make all Member Variables Private.

This is one of the most commonly held of all the conventions of OOD. Member variables of classes should be known only to the methods of the class that defines them. Member variables should never be known to any other class, including derived classes. Thus they should be declared `private`, rather than `public` or `protected`.

In light of the open-closed principle, the reason for this convention ought to be clear. When the member variables of a class change, every function that depends upon those variables must be changed. Thus, no function that depends upon a variable can be closed with respect to that variable.

Listing 6

```
Table driven type ordering mechanism
#include <typeinfo.h>
#include <string.h>
enum {false, true};
typedef int bool;

class Shape
{
public:
    virtual void Draw() const = 0;
    virtual bool Precedes(const Shape&) const;

    bool operator<(const Shape& s) const
    {return Precedes(s);}

private:
    static char* typeOrderTable[];
};

char* Shape::typeOrderTable[] =
{
    "Circle",
    "Square",
    0
};

// This function searches a table for the class names.
// The table defines the order in which the
// shapes are to be drawn. Shapes that are not
// found always precede shapes that are found.
//
bool Shape::Precedes(const Shape& s) const
{
    const char* thisType = typeid(*this).name();
    const char* argType = typeid(s).name();
    bool done = false;
    int thisOrd = -1;
    int argOrd = -1;
    for (int i=0; !done; i++)
    {
        const char* tableEntry = typeOrderTable[i];
        if (tableEntry != 0)
        {
            if (strcmp(tableEntry, thisType) == 0)
                thisOrd = i;
            if (strcmp(tableEntry, argType) == 0)
                argOrd = i;
        }
    }
    if (thisOrd > argOrd)
        return true;
    else
        return false;
}
```

Listing 3

Shape with ordering methods.

```
class Shape
{
public:
    virtual void Draw() const = 0;
    virtual bool Precedes(const Shape&) const = 0;

    bool operator<(const Shape& s) {return Precedes(s);}
};
```

Listing 4

DrawAllShapes with Ordering

```
void DrawAllShapes(Set<Shape*>& list)
{
    // copy elements into OrderedSet and then sort.
    OrderedSet<Shape*> orderedList = list;
    orderedList.Sort();

    for (Iterator<Shape*> i(orderedList); i; i++)
        (*i)->Draw();
}
```

Listing 5

Ordering a Circle

```
bool Circle::Precedes(const Shape& s) const
{
    if (dynamic_cast<Square*>(s))
        return true;
    else
        return false;
}
```

It should be very clear that this function does not conform to the open-closed principle. There is no way to close it against new derivatives of Shape. Every time a new derivative of Shape is created, this function will need to be changed.

Using a “Data Driven” Approach to Achieve Closure.

Closure of the derivatives of Shape can be achieved by using a table driven approach that does not force changes in every derived class. Listing 6 shows one possibility.

By taking this approach we have successfully closed the `DrawAllShapes` function against ordering issues in general and each of the `Shape` derivatives against the creation of new `Shape` derivatives or a change in policy that reorders the `Shape` objects by their type. (e.g. Changing the ordering so that `Squares` are drawn first.)

Strategic Closure

It should be clear that no significant program can be 100% closed. For example, consider what would happen to the `DrawAllShapes` function from Listing 2 if we decided that all `Circles` should be drawn before any `Squares`. The `DrawAllShapes` function is not closed against a change like this. In general, no matter how “closed” a module is, there will always be some kind of change against which it is not closed.

Since closure cannot be complete, it must be strategic. That is, the designer must choose the kinds of changes against which to close his design. This takes a certain amount of prescience derived from experience. The experienced designer knows the users and the industry well enough to judge the probability of different kinds of changes. He then makes sure that the open-closed principle is invoked for the most probable changes.

Using Abstraction to Gain Explicit Closure.

How could we close the `DrawAllShapes` function against changes in the ordering of drawing? Remember that closure is based upon abstraction. Thus, in order to close `DrawAllShapes` against ordering, we need some kind of “ordering abstraction”. The specific case of ordering above had to do with drawing certain types of shapes before other types of shapes.

An ordering policy implies that, given any two objects, it is possible to discover which ought to be drawn first. Thus, we can define a method of `Shape` named `Precedes` that takes another `Shape` as an argument and returns a `bool` result. The result is `true` if the `Shape` object that receives the message should be ordered before the `Shape` object passed as the argument.

In C++ this function could be represented by an overloaded `operator<` function. Listing 3 shows what the `Shape` class might look like with the ordering methods in place.

Now that we have a way to determine the relative ordering of two `Shape` objects, we can sort them and then draw them in order. Listing 4 shows the C++ code that does this. This code uses the `Set`, `OrderedSet` and `Iterator` classes from the `Components` category developed in my book³ (if you would like a free copy of the source code of the `Components` category, send email to `rmartin@oma.com`).

This gives us a means for ordering `Shape` objects, and for drawing them in the appropriate order. But we still do not have a decent ordering abstraction. As it stands, the individual `Shape` objects will have to override the `Precedes` method in order to specify ordering. How would this work? What kind of code would we write in `Circle::Precedes` to ensure that `Circles` were drawn before `Squares`? Consider Listing 5.

3. *Designing Object Oriented C++ Applications using the Booch Method*, Robert C. Martin, Prentice Hall, 1995.

: The Open-Closed Principle

statements would be combined with logical operators, or that the case clauses of the switch statements would be combined so as to “simplify” the local decision making. Thus the problem of finding and understanding all the places where the new shape needs to be added can be non-trivial.

Listing 2 shows the code for a solution to the square/circle problem that conforms to the open-closed principle. In this case an abstract Shape class is created. This abstract class has a single pure-virtual function called Draw. Both Circle and Square are derivatives of the Shape class.

Listing 2

```
OOD solution to Square/Circle problem.
class Shape
{
public:
    virtual void Draw() const = 0;
};

class Square : public Shape
{
public:
    virtual void Draw() const;
};

class Circle : public Shape
{
public:
    virtual void Draw() const;
};

void DrawAllShapes(Set<Shape*>& list)
{
    for (Iterator<Shape*>i(list); i; i++)
        (*i)->Draw();
}
```

Note that if we want to extend the behavior of the DrawAllShapes function in Listing 2 to draw a new kind of shape, all we need do is add a new derivative of the Shape class. The DrawAllShapes function does not need to change. Thus DrawAllShapes conforms to the open-closed principle. Its behavior can be extended without modifying it.

In the real world the Shape class would have many more methods. Yet adding a new shape to the application is still quite simple since all that is required is to create the new derivative and implement all its functions. There is no need to hunt through all of the application looking for places that require changes.

Since programs that conform to the open-closed principle are changed by adding new code, rather than by changing existing code, they do not experience the cascade of changes exhibited by non-conforming programs.

Listing 1 (Continued)

Procedural Solution to the Square/Circle Problem

```
struct Square
{
    ShapeType itsType;
    double itsSide;
    Point itsTopLeft;
};

// These functions are implemented elsewhere
//
void DrawSquare(struct Square*)
void DrawCircle(struct Circle*);

typedef struct Shape *ShapePointer;

void DrawAllShapes(ShapePointer list[], int n)
{
    int i;
    for (i=0; i<n; i++)
    {
        struct Shape* s = list[i];
        switch (s->itsType)
        {
            case square:
                DrawSquare((struct Square*)s);
                break;

            case circle:
                DrawCircle((struct Circle*)s);
                break;
        }
    }
}
```

The function `DrawAllShapes` does not conform to the open-closed principle because it cannot be closed against new kinds of shapes. If I wanted to extend this function to be able to draw a list of shapes that included triangles, I would have to modify the function. In fact, I would have to modify the function for any new type of shape that I needed to draw.

Of course this program is only a simple example. In real life the `switch` statement in the `DrawAllShapes` function would be repeated over and over again in various functions all over the application; each one doing something a little different. Adding a new shape to such an application means hunting for every place that such `switch` statements (or `if/else` chains) exist, and adding the new shape to each. Moreover, it is very unlikely that all the `switch` statements and `if/else` chains would be as nicely structured as the one in `DrawAllShapes`. It is much more likely that the predicates of the `if`

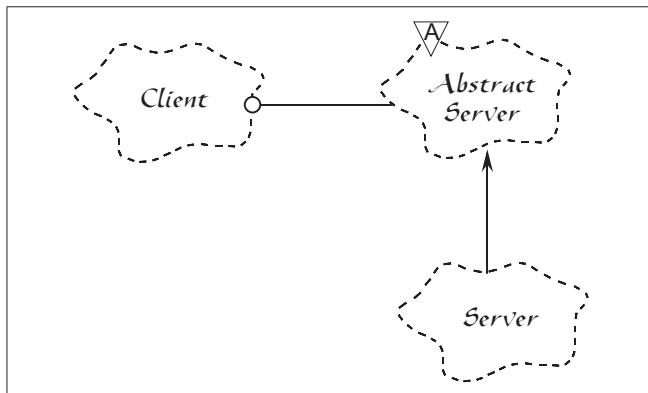


Figure 2
Open Client

The Shape Abstraction

Consider the following example. We have an application that must be able to draw circles and squares on a standard GUI. The circles and squares must be drawn in a particular order. A list of the circles and squares will be created in the appropriate order and the program must walk the list in that order and draw each circle or square.

In C, using procedural techniques that do not conform to the open-closed principle, we might solve this problem as shown in Listing 1. Here we see a set of data structures that have the same first element, but are different beyond that. The first element of each is a type code that identifies the data structure as either a circle or a square. The function DrawAllShapes walks an array of pointers to these data structures, examining the type code and then calling the appropriate function (either DrawCircle or DrawSquare).

Listing 1

Procedural Solution to the Square/Circle Problem

```

enum ShapeType {circle, square};

struct Shape
{
    ShapeType itsType;
};

struct Circle
{
    ShapeType itsType;
    double itsRadius;
    Point itsCenter;
};

```

1. They are “Open For Extension”.

This means that the behavior of the module can be extended. That we can make the module behave in new and different ways as the requirements of the application change, or to meet the needs of new applications.

2. They are “Closed for Modification”.

The source code of such a module is inviolate. No one is allowed to make source code changes to it.

It would seem that these two attributes are at odds with each other. The normal way to extend the behavior of a module is to make changes to that module. A module that cannot be changed is normally thought to have a fixed behavior. How can these two opposing attributes be resolved?

Abstraction is the Key.

In C++, using the principles of object oriented design, it is possible to create abstractions that are fixed and yet represent an unbounded group of possible behaviors. The abstractions are abstract base classes, and the unbounded group of possible behaviors is represented by all the possible derivative classes. It is possible for a module to manipulate an abstraction. Such a module can be closed for modification since it depends upon an abstraction that is fixed. Yet the behavior of that module can be extended by creating new derivatives of the abstraction.

Figure 1 shows a simple design that does not conform to the open-closed principle. Both the `Client` and `Server` classes are concrete. There is no guarantee that the member functions of the `Server` class are virtual. The `Client` class *uses* the `Server` class. If we wish for a `Client` object to use a different server object, then the `Client` class must be changed to name the new server class.

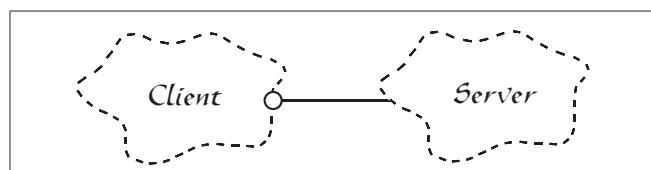


Figure 1
Closed Client

Figure 2 shows the corresponding design that conforms to the open-closed principle. In this case, the `AbstractServer` class is an abstract class with pure-virtual member functions. the `Client` class uses this abstraction. However objects of the `Client` class will be using objects of the derivative `Server` class. If we want `Client` objects to use a different server class, then a new derivative of the `AbstractServer` class can be created. The `Client` class can remain unchanged.

The Open-Closed Principle

This is the first of my *Engineering Notebook* columns for *The C++ Report*. The articles that will appear in this column will focus on the use of C++ and OOD, and will address issues of software engineering. I will strive for articles that are pragmatic and directly useful to the software engineer in the trenches. In these articles I will make use of Booch's notation for documenting object oriented designs. The sidebar provides a brief lexicon of Booch's notation.

There are many heuristics associated with object oriented design. For example, “all member variables should be private”, or “global variables should be avoided”, or “using run time type identification (RTTI) is dangerous”. What is the source of these heuristics? What makes them true? Are they *always* true? This column investigates the design principle that underlies these heuristics -- the open-closed principle.

As Ivar Jacobson said: “All systems change during their life cycles. This must be borne in mind when developing systems expected to last longer than the first version.”¹ How can we create designs that are stable in the face of change and that will last longer than the first version? Bertrand Meyer² gave us guidance as long ago as 1988 when he coined the now famous open-closed principle. To paraphrase him:

***SOFTWARE ENTITIES (CLASSES, MODULES, FUNCTIONS, ETC.)
SHOULD BE OPEN FOR EXTENSION, BUT CLOSED FOR
MODIFICATION.***

When a single change to a program results in a cascade of changes to dependent modules, that program exhibits the undesirable attributes that we have come to associate with “bad” design. The program becomes fragile, rigid, unpredictable and unreusable. The open-closed principle attacks this in a very straightforward way. It says that you should design modules that *never change*. When requirements change, you extend the behavior of such modules by adding new code, not by changing old code that already works.

Description

Modules that conform to the open-closed principle have two primary attributes.

-
1. Object Oriented Software Engineering a Use Case Driven Approach, Ivar Jacobson, Addison Wesley, 1992, p 21.
 2. Object Oriented Software Construction, Bertrand Meyer, Prentice Hall, 1988, p 23

Conclusion

The Open-Closed principle is at the heart of many of the claims made for OOD. It is when this principle is in effect that applications are more maintainable, reusable and robust. The Liskov Substitution Principle (A.K.A Design by Contract) is an important feature of all programs that conform to the Open-Closed principle. It is only when derived types are completely substitutable for their base types that functions which use those base types can be reused with impunity, and the derived types can be changed with impunity.

This article is an extremely condensed version of a chapter from my new book: *Patterns and Advanced Principles of OOD*, to be published soon by Prentice Hall. In subsequent articles we will explore many of the other principles of object oriented design. We will also study various design patterns, and their strengths and weaknesses with regard to implementation in C++. We will study the role of Booch's class categories in C++, and their applicability as C++ namespaces. We will define what "cohesion" and "coupling" mean in an object oriented design, and we will develop metrics for measuring the quality of an object oriented design. And, after that, many other interesting topics.



A Real Example.

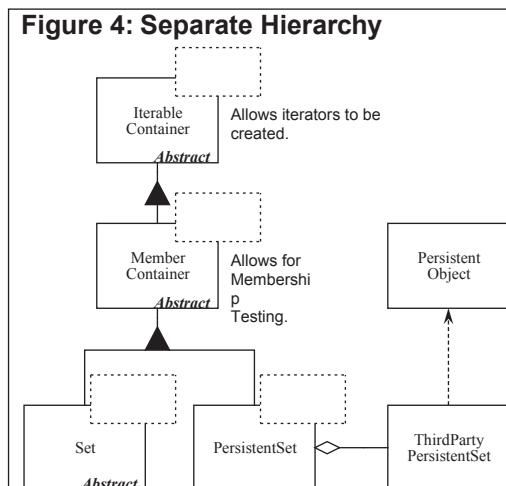
10

This solution may seem overly restrictive, but it was the only way I could think of to prevent PersistentSet objects from appearing at the interface of functions that would want to add non-persistent objects to them. Moreover it broke the dependency of the rest of the application upon the whole notion of persistence.

Did this solution work? Not really. The convention was violated in several parts of the application by engineers who did not understand the necessity for it. That is the problem with conventions, they have to be continually re-sold to each engineer. If the engineer does not agree, then the convention will be violated. And one violation ruins the whole structure.

An LSP Compliant Solution

How would I solve this now? I would acknowledge that a PersistentSet does not have an ISA relationship with Set; that it is not a proper derivative of Set. Thus I would Separate the hierarchies. But not completely. There are features that Set and PersistentSet have in common. In fact, it is only the Add method that causes the difficulty with LSP. Thus I would create a hierarchy in which both Set and PersistentSet were siblings beneath an abstract interface that allowed for membership testing, iteration, etc. (See Figure 4.) This would allow PersistentSet objects to be Iterated and tested for membership, etc. But would not afford the ability to add objects that were not derived from PersistentObject to a PersistentSet.



```

template <class T>
void PersistentSet::Add(const T& t)
{
    itsThirdPartyPersistentSet.Add(t);
    // This will generate a compiler error if t is
    // not derived from PersistentObject.
}
  
```

As the listing above shows, any attempt to add an object that is not derived from PersistentObject to a PersistentSet will result in a compilation error. (The interface of the third party persistent set expects a PersistentObject&).

derived from the abstract base class `PersistentObject`. I created the hierarchy shown in Figure 3.

On the surface of it, this might look all right. However there is an implication that is rather ugly. When a client is adding members to the base class `Set`, how is that client supposed to ensure that it only adds derivatives of `PersistentObject` if the `Set` happens to be a `PersistentSet`?

Consider the code for `PersistentSet::Add`:

```
template <class T>
void PersistentSet::Add(const T& t)
{
    PersistentObject& p =
        dynamic_cast<PersistentObject&>(t); // throw bad_cast
    itsThirdPartyPersistentSet.Add(p);
}
```

This code makes it clear that if any client tries to add an object that is not derived from the class `PersistentObject` to my `PersistentSet`, a runtime error will ensue. The `dynamic_cast` will throw `bad_cast` (one of the standard exception objects). None of the existing clients of the abstract base class `Set` expect exceptions to be thrown on `Add`. Since these functions will be confused by a derivative of `Set`, this change to the hierarchy violates the LSP.

Is this a problem? Certainly. Functions that never before failed when passed a derivative of `Set`, will now cause runtime errors when passed a `PersistentSet`. Debugging this kind of problem is relatively difficult since the runtime error occurs very far away from the actual logic flaw. The logic flaw is either the decision to pass a `PersistentSet` into the failed function, or it is the decision to add an object to the `PersistentSet` that is not derived from `PersistentObject`. In either case, the actual decision might be millions of instructions away from the actual invocation of the `Add` method. Finding it can be a bear. Fixing it can be worse.

A Solution that does *not* conform to the LSP.

How do we solve this problem? Several years ago, I solved it by convention. Which is to say that I did not solve it in source code. Rather I instated a convention whereby `PersistentSet` and `PersistentObject` were not known to the application as a whole. They were only known to one particular module. This module was responsible for reading and writing all the containers. When a container needed to be written, its contents were copied into `PersistentObjects` and then added to `PersistentSets`, which were then saved on a stream. When a container needed to be read from a stream, the process was inverted. A `PersistentSet` was read from the stream, and then the `PersistentObjects` were removed from the `PersistentSet` and copied into regular (non-persistent) objects which were then added to a regular `Set`.



A Real Example.

8

third parties. I did not want my application code to be horribly dependent upon these containers because I felt that I would want to replace them with better classes later. Thus I wrapped the third party containers in my own abstract interface. (See Figure 2)

I had an abstract class called `Set` which presented pure virtual `Add`, `Delete`, and `IsMember` functions.

```
template <class T>
class Set
{
public:
    virtual void Add(const T&) = 0;
    virtual void Delete(const T&) = 0;
    virtual bool IsMember(const T&) const = 0;
};
```

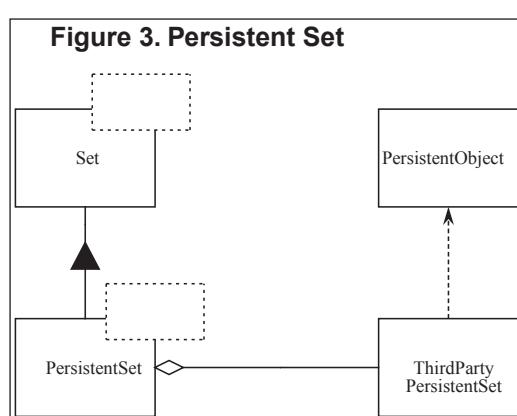
This structure unified the Unbounded and Bounded varieties of the two third party sets and allowed them to be accessed through a common interface. Thus some client could accept an argument of type `Set<T>&` and would not care whether the actual `Set` it worked on was of the Bounded or Unbounded variety. (See the `PrintSet` function listing.)

```
template <class T>
void PrintSet(const Set<T>& s)
{
    for (Iterator<T>i(s); i; i++)
        cout << (*i) << endl;
}
```

This ability to neither know nor care the type of `Set` you are operating on is a big advantage. It means that the programmer can decide which kind of `Set` is needed in each particular instance. None of the client functions will be affected by that decision. The programmer may choose a `BoundedSet` when memory is tight and speed is not critical, or the programmer may choose an `UnboundedSet` when memory is plentiful and speed is critical. The client functions will manipulate these objects through the interface of the base class `Set`, and will therefore not know or care which kind of `Set` they are using.

Problem

I wanted to add a `PersistentSet` to this hierarchy. A persistent set is a set which can be written out to a stream, and then read back in later, possibly by a different application. Unfortunately, the only third party container that I had access to, that also offered persistence, was not a template class. Instead, it accepted objects that were



We can view the postcondition of `Rectangle::SetWidth(double w)` as:

```
assert((itsWidth == w) && (itsHeight == old.itsHeight));
```

Now the rule for the preconditions and postconditions for derivatives, as stated by Meyer³, is:

...when redefining a routine [in a derivative], you may only replace its precondition by a weaker one, and its postcondition by a stronger one.

In other words, when using an object through its base class interface, the user knows only the preconditions and postconditions of the base class. Thus, derived objects must not expect such users to obey preconditions that are stronger than those required by the base class. That is, they must accept anything that the base class could accept. Also, derived classes must conform to all the postconditions of the base. That is, their behaviors and outputs must not violate any of the constraints established for the base class. Users of the base class must not be confused by the output of the derived class.

Clearly, the postcondition of `Square::SetWidth(double w)` is weaker than the postcondition of `Rectangle::SetWidth(double w)` above, since it does not conform to the base class clause “(itsHeight == old.itsHeight)”. Thus, `Square::SetWidth(double w)` violates the contract of the base class.

Certain languages, like Eiffel, have direct support for preconditions and postconditions. You can actually declare them, and have the runtime system verify them for you. C++ does not have such a feature. Yet, even in C++ we can manually consider the preconditions and postconditions of each method, and make sure that Meyer’s rule is not violated. Moreover, it can be very helpful to document these preconditions and postconditions in the comments for each method.

A Real Example.

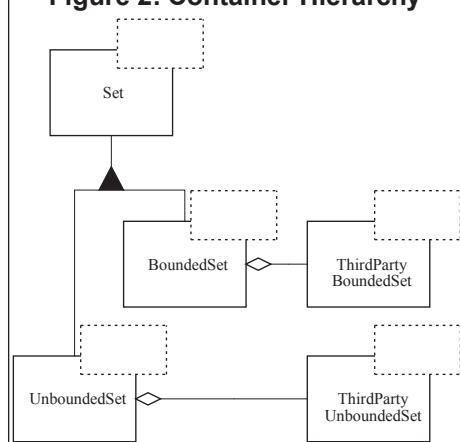
Enough of squares and rectangles! Does the LSP have a bearing on real software? Let’s look at a case study that comes from a project that I worked on a few years ago.

Motivation

I was unhappy with the interfaces of the container classes that were available through

3. *ibid*, p256

Figure 2: Container Hierarchy



The Liskov Substitution Principle

6

but cannot operate properly upon `Square` objects. These functions expose a violation of the LSP. The addition of the `Square` derivative of `Rectangle` has broken these functions; and so the Open-Closed principle has been violated.

Validity is not Intrinsic

This leads us to a very important conclusion. A model, viewed in isolation, can not be meaningfully validated. The validity of a model can only be expressed in terms of its clients. For example, when we examined the final version of the `Square` and `Rectangle` classes in isolation, we found that they were self consistent and valid. Yet when we looked at them from the viewpoint of a programmer who made reasonable assumptions about the base class, the model broke down.

Thus, when considering whether a particular design is appropriate or not, one must not simply view the solution in isolation. One must view it in terms of the reasonable assumptions that will be made by the users of that design.

What Went Wrong? (W³)

So what happened? Why did the apparently reasonable model of the `Square` and `Rectangle` go bad. After all, isn't a `Square` a `Rectangle`? Doesn't the ISA relationship hold?

No! A square might be a rectangle, but a `Square` object is definitely *not* a `Rectangle` object. Why? Because the *behavior* of a `Square` object is not consistent with the behavior of a `Rectangle` object. Behaviorally, a `Square` is not a `Rectangle`! And it is *behavior* that software is really all about.

The LSP makes clear that in OOD the ISA relationship pertains to *behavior*. Not intrinsic private behavior, but extrinsic public behavior; behavior that clients depend upon. For example, the author of function `g` above depended on the fact that `Rectangles` behave such that their height and width vary independently of one another. That independence of the two variables is an extrinsic public behavior that other programmers are likely to depend upon.

In order for the LSP to hold, and with it the Open-Closed principle, all derivatives must conform to the behavior that clients expect of the base classes that they use.

Design by Contract

There is a strong relationship between the LSP and the concept of Design by Contract as expounded by Bertrand Meyer². Using this scheme, methods of classes declare preconditions and postconditions. The preconditions must be true in order for the method to execute. Upon completion, the method guarantees that the postcondition will be true.

2. *Object Oriented Software Construction*, Bertrand Meyer, Prentice Hall, 1988

```

private:
    double itsHeight;
    double itsWidth;
};

class Square : public Rectangle
{
public:
    virtual void SetWidth(double w);
    virtual void SetHeight(double h);
};

void Square::SetWidth(double w)
{
    Rectangle::SetWidth(w);
    Rectangle::SetHeight(w);
}

void Square::SetHeight(double h)
{
    Rectangle::SetHeight(h);
    Rectangle::SetWidth(h);
}

```

The Real Problem

At this point in time we have two classes, `Square` and `Rectangle`, that appear to work. No matter what you do to a `Square` object, it will remain consistent with a mathematical square. And regardless of what you do to a `Rectangle` object, it will remain a mathematical rectangle. Moreover, you can pass a `Square` into a function that accepts a pointer or reference to a `Rectangle`, and the `Square` will still act like a square and will remain consistent.

Thus, we might conclude that the model is now self consistent, and correct. However, this conclusion would be amiss. A model that is self consistent is not necessarily consistent with all its users! Consider function `g` below.

```

void g(Rectangle& r)
{
    r.setWidth(5);
    r.setHeight(4);
    assert(r.getWidth() * r.getHeight() == 20);
}

```

This function invokes the `setWidth` and `setHeight` members of what it believes to be a `Rectangle`. The function works just fine for a `Rectangle`, but declares an assertion error if passed a `Square`. So here is the real problem: Was the programmer who wrote that function justified in assuming that *changing the width of a Rectangle leaves its height unchanged?*

Clearly, the programmer of `g` made this very reasonable assumption. Passing a `Square` to functions whose programmers made this assumption will result in problems. Therefore, there exist functions that take pointers or references to `Rectangle` objects,



The Liskov Substitution Principle

4

with the design. However, there is a way to sidestep the problem. We could override `SetWidth` and `SetHeight` as follows:

```
void Square::SetWidth(double w)
{
    Rectangle::SetWidth(w);
    Rectangle::SetHeight(w);
}

void Square::SetHeight(double h)
{
    Rectangle::SetHeight(h);
    Rectangle::SetWidth(h);
}
```

Now, when someone sets the width of a `Square` object, its height will change correspondingly. And when someone sets its height, the width will change with it. Thus, the invariants of the `Square` remain intact. The `Square` object will remain a mathematically proper square.

```
Square s;
s.setWidth(1); // Fortunately sets the height to 1 too.
s.setHeight(2); // sets width and height to 2, good thing.
```

But consider the following function:

```
void f(Rectangle& r)
{
    r.setWidth(32); // calls Rectangle::GetWidth
```

If we pass a reference to a `Square` object into this function, the `Square` object will be corrupted because the height won't be changed. This is a clear violation of LSP. The `f` function does not work for derivatives of its arguments. The reason for the failure is that `GetWidth` and `SetHeight` were not declared `virtual` in `Rectangle`.

We can fix this easily. However, when the creation of a derived class causes us to make changes to the base class, it often implies that the design is faulty. Indeed, it violates the Open-Closed principle. We might counter this with argument that forgetting to make `GetWidth` and `SetHeight` `virtual` was the real design flaw, and we are just fixing it now. However, this is hard to justify since setting the height and width of a rectangle are exceedingly primitive operations. By what reasoning would we make them `virtual` if we did not anticipate the existence of `Square`.

Still, let's assume that we accept the argument, and fix the classes. We wind up with the following code:

```
class Rectangle
{
public:
    virtual void SetWidth(double w) {itsWidth=w;}
    virtual void SetHeight(double h) {itsHeight=h;}
    double GetHeight() const {return itsHeight;}
    double GetWidth() const {return itsWidth;}
```

Square and Rectangle, a More Subtle Violation.

However, there are other, far more subtle, ways of violating the LSP. Consider an application which uses the `Rectangle` class as described below:

```
class Rectangle
{
public:
    void SetWidth(double w) {itsWidth=w;}
    void SetHeight(double h) {itsHeight=h;}
    double GetHeight() const {return itsHeight;}
    double GetWidth() const {return itsWidth;}
private:
    double itsWidth;
    double itsHeight;
};
```

Imagine that this application works well, and is installed in many sites. As is the case with all successful software, as its users' needs change, new functions are needed. Imagine that one day the users demand the ability to manipulate squares in addition to rectangles.

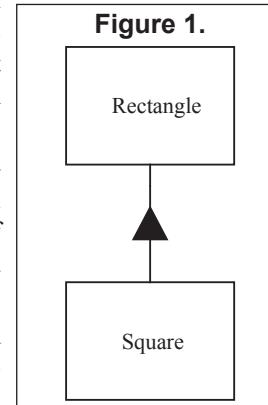
It is often said that, in C++, inheritance is the ISA relationship. In other words, if a new kind of object can be said to fulfill the ISA relationship with an old kind of object, then the class of the new object should be derived from the class of the old object.

Clearly, a square is a rectangle for all normal intents and purposes. Since the ISA relationship holds, it is logical to model the `Square` class as being derived from `Rectangle`. (See Figure 1.)

This use of the ISA relationship is considered by many to be one of the fundamental techniques of Object Oriented Analysis. A square is a rectangle, and so the `Square` class should be derived from the `Rectangle` class. However this kind of thinking can lead to some subtle, yet significant, problems. Generally these problem are not foreseen until we actually try to code the application.

Our first clue might be the fact that a `Square` does not need both `itsHeight` and `itsWidth` member variables. Yet it will inherit them anyway. Clearly this is wasteful. Moreover, if we are going to create hundreds of thousands of `Square` objects (e.g. a CAD/CAE program in which every pin of every component of a complex circuit is drawn as a square), this waste could be extremely significant.

However, let's assume that we are not very concerned with memory efficiency. Are there other problems? Indeed! `Square` will inherit the `SetWidth` and `SetHeight` functions. These functions are utterly inappropriate for a `Square`, since the width and height of a square are identical!”. This should be a significant clue that there is a problem



The Liskov Substitution Principle

FUNCTIONS THAT USE POINTERS OR REFERENCES TO BASE CLASSES MUST BE ABLE TO USE OBJECTS OF DERIVED CLASSES WITHOUT KNOWING IT.

The above is a paraphrase of the Liskov Substitution Principle (LSP). Barbara Liskov first wrote it as follows nearly 8 years ago¹:

What is wanted here is something like the following substitution property: If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T , the behavior of P is unchanged when o_1 is substituted for o_2 , then S is a subtype of T .

The importance of this principle becomes obvious when you consider the consequences of violating it. If there is a function which does not conform to the LSP, then that function uses a pointer or reference to a base class, but must *know* about all the derivatives of that base class. Such a function violates the Open-Closed principle because it must be modified whenever a new derivative of the base class is created.

A Simple Example of a Violation of LSP

One of the most glaring violations of this principle is the use of C++ Run-Time Type Information (RTTI) to select a function based upon the type of an object. i.e.:

```
void DrawShape(const Shape& s)
{
    if (typeid(s) == typeid(Square))
        DrawSquare(static_cast<Square&>(s));
    else if (typeid(s) == typeid(Circle))
        DrawCircle(static_cast<Circle&>(s));
}
```

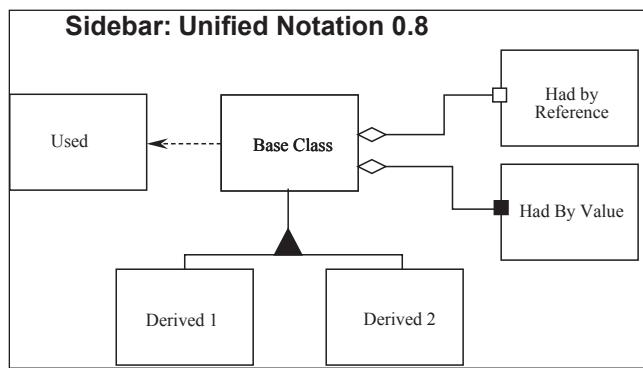
[Note: `static_cast` is one of the new cast operators. In this example it works exactly like a regular cast. i.e. `DrawSquare ((Square&) s) ;`. However the new syntax has more stringent rules that make it safer to use, and is easier to locate with tools such as grep. It is therefore preferred.]

Clearly the `DrawShape` function is badly formed. It must know about every possible derivative of the `Shape` class, and it must be changed whenever new derivatives of `Shape` are created. Indeed, many view the structure of this function as anathema to Object Oriented Design.

1. Barbara Liskov, "Data Abstraction and Hierarchy," *SIGPLAN Notices*, 23,5 (May, 1988).

The Liskov Substitution Principle

This is the second of my *Engineering Notebook* columns for *The C++ Report*. The articles that will appear in this column will focus on the use of C++ and OOD, and will address issues of software engineering. I will strive for articles that are pragmatic and directly useful to the software engineer in the trenches. In these articles I will make use of Booch's and Rumbaugh's new *unified* notation (Version 0.8) for documenting object oriented designs. The sidebar provides a brief lexicon of this notation.



Introduction

My last column (Jan, 96) talked about the Open-Closed principle. This principle is the foundation for building code that is maintainable and reusable. It states that well designed code can be extended without modification; that in a well designed program new features are added by adding new code, rather than by changing old, already working, code.

The primary mechanisms behind the Open-Closed principle are abstraction and polymorphism. In statically typed languages like C++, one of the key mechanisms that supports abstraction and polymorphism is inheritance. It is by using inheritance that we can create derived classes that conform to the abstract polymorphic interfaces defined by pure virtual functions in abstract base classes.

What are the design rules that govern this particular use of inheritance? What are the characteristics of the best inheritance hierarchies? What are the traps that will cause us to create hierarchies that do not conform to the Open-Closed principle? These are the questions that this article will address.



changed, ‘g’ and all clients of ‘g’ would have to recompile. This is more perverse than `g(ui,ui);!` Moreover, we cannot be sure that both arguments of ‘g’ will *always* refer to the same object! In the future, it may be that the interface objects are separated for some reason. From the point of view of function ‘g’, the fact that all interfaces are combined into a single object is information that ‘g’ does not need to know. Thus, I prefer the polyadic form for such functions.

Conclusion

In this article we have discussed the disadvantages of “fat interfaces”; i.e. interfaces that are not specific to a single client. Fat interfaces lead to inadvertent couplings between clients that ought otherwise to be isolated. By making use of the ADAPTER pattern, either through delegation (object form) or multiple inheritance (class form), fat interfaces can be segregated into abstract base classes that break the unwanted coupling between clients.

This article is an extremely condensed version of a chapter from my new book: *Patterns and Advanced Principles of OOD*, to be published soon by Prentice Hall. In subsequent articles we will explore many of the other principles of object oriented design. We will also study various design patterns, and their strengths and weaknesses with regard to implementation in C++. We will study the role of Booch’s class categories in C++, and their applicability as C++ namespaces. We will define what “cohesion” and “coupling” mean in an object oriented design, and we will develop metrics for measuring the quality of an object oriented design. And after that, we will discuss many other interesting topics.

```
GwithdrawlUI.RequestWithdrawlAmount();  
...  
};
```

One might be tempted to put all the globals in Listing 8 into a single class in order to prevent pollution of the global namespace. Listing 9 shows such an approach. This, however, has an unfortunate effect. In order to use UIGlobals, you must #include ui_globals.h. This, in turn, #includes depositUI.h, withdrawUI.h, and transferUI.h. This means that any module wishing to use any of the UI interfaces transitively depends upon all of them; exactly the situation that the ISP warns us to avoid. If a change is made to any of the UI interfaces, all modules that #include ui_globals.h are forced to recompile. The UIGlobals class has recombined the interfaces that we had worked so hard to segregate!

Listing 9

Wrapping the Globals in a class

```
// in ui_globals.h  
  
#include "depositUI.h"  
#include "withdrawlUI.h"  
#include "transferUI.h"  
  
class UIGlobals  
{  
public:  
    static WithdrawlUI& withdrawl;  
    static DepositUI& deposit;  
    static TransferUI& transfer  
};  
  
// in ui_globals.cc  
  
static UI Lui; // non-global object;  
DepositUI& UIGlobals::deposit = Lui;  
WithdrawlUI& UIGlobals::withdrawl = Lui;  
TransferUI& UIGlobals::transfer = Lui;
```

The Polyad vs. the Monad.

Consider a function ‘g’ that needs access to both the DepositUI and the TransferUI. Consider also that we wish to pass the UIs into this function. Should we write the function prototype like this: `void g(DepositUI&, TransferUI&);`? Or should we write it like this: `void g(UI&);`?

The temptation to write the latter (monadic) form is strong. After all, we know that in the former (polyadic) form, both arguments will refer to the *same object*. Moreover, if we were to use the polyadic form, its invocation would look like this: `g(ui, ui);` Somehow this seems perverse.

Perverse or not, the polyadic form is preferable to the monadic form. The monadic form forces ‘g’ to depend upon every interface included in UI. Thus, when WithdrawUI

```

        , public WithdrawlUI,
        , public TransferUI
{
    public:
        virtual void RequestDepositAmount();
        virtual void RequestWithdrawlAmount();
        virtual void RequestTransferAmount();
};
```

A careful examination of Listing 6 will show one of the issues with ISP conformance that was not obvious from the TimedDoor example. Note that each transaction must somehow know about its particular version of the UI. DepositTransaction must know about DepositUI; WithdrawlTransaction must know about WithdrawlUI, etc. In Listing 6 I have addressed this issue by forcing each transaction to be constructed with a reference to its particular UI. Note that this allows me to employ the idiom in Listing 7.

Listing 7

Interface Initialization Idiom

```

UI Gui; // global object;

void f()
{
    DepositTransaction dt(Gui);
}
```

This is handy, but also forces each transaction to contain a reference member to its UI. Another way to address this issue is to create a set of global constants as shown in Listing 8. As we discovered when we discussed the Open Closed Principle in the January 96 issue, global variables are not always a symptom of a poor design. In this case they provide the distinct advantage of easy access. And since they are references, it is impossible to change them in any way, therefore they cannot be manipulated in a way that would surprise other users.

Listing 8

Separate Global Pointers

```

// in some module that gets linked in
// to the rest of the app.

static UI Lui; // non-global object;
DepositUI& GdepositUI = Lui;
WithdrawlUI& GwithdrawlUI = Lui;
TransferUI& GtransferUI = Lui;

// In the depositTransaction.h module

class class WithdrawlTransation : public Transaction
{
public:
    virtual void Execute()
    {
        ...
    }
}
```

```
class class DepositTransation : public Transaction
{
public:
    DepositTransation(DepositUI& ui)
    : itsDepositUI(ui)
    {}

    virtual void Execute()
    {
        ...
        itsDepositUI.RequestDepositAmount();
        ...
    }

private:
    DepositUI& itsDepositUI;
};

class WithdrawlUI
{
public:
    virtual void RequestWithdrawlAmount() = 0;
};

class class WithdrawlTransation : public Transaction
{
public:
    WithdrawlTransation(WithdrawlUI& ui)
    : itsWithdrawlUI(ui)
    {}

    virtual void Execute()
    {
        ...
        itsWithdrawlUI.RequestWithdrawlAmount();
        ...
    }

private:
    WithdrawlUI& itsWithdrawlUI;
};

class TransferUI
{
public:
    virtual void RequestTransferAmount() = 0;
};

class class TransferTransation : public Transaction
{
public:
    TransferTransation(TransferUI& ui)
    : itsTransferUI(ui)
    {}

    virtual void Execute()
    {
        ...
        itsTransferUI.RequestTransferAmount();
        ...
    }

private:
    TransferUI& itsTransferUI;
};

class UI : public DepositUI,
```

: The Interface Segregation Principle

Figure 5
ATM Transaction Hierarchy

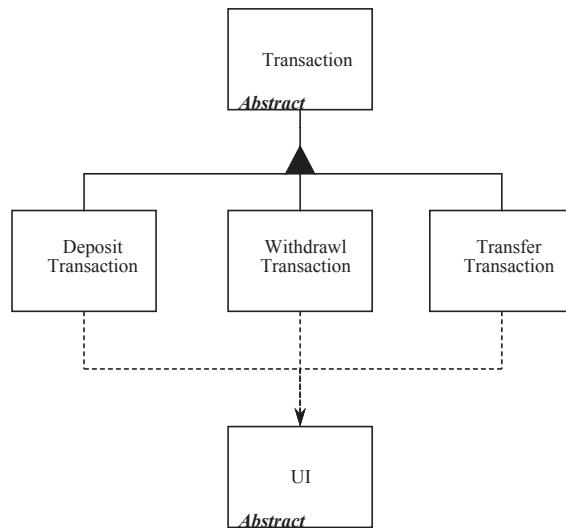
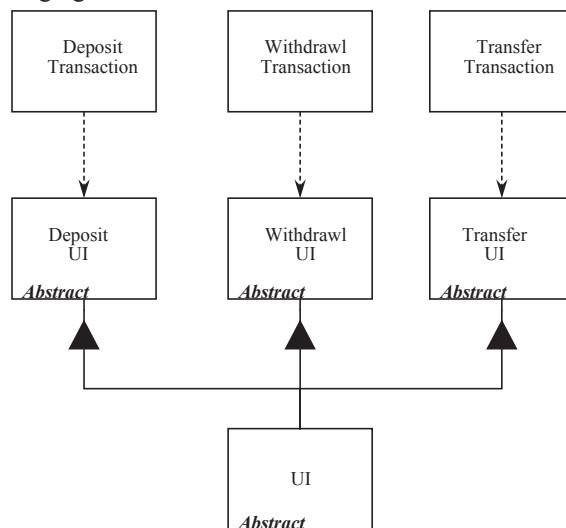


Figure 6
Segregated ATM UI Interface



Listing 6
Segregated ATM Interfaces

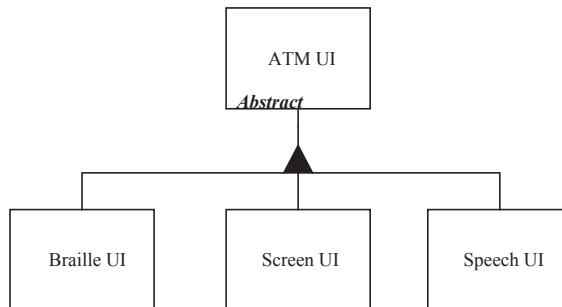
```

class DepositUI
{
public:
    virtual void RequestDepositAmount() = 0;
};
  
```

The ATM User Interface Example

Now let's consider a slightly more significant example. The traditional Automated Teller Machine (ATM) problem. The user interface of an ATM machine needs to be very flexible. The output may need to be translated into many different language. It may need to be presented on a screen, or on a braille tablet, or spoken out a speech synthesizer. Clearly this can be achieved by creating an abstract base class that has pure virtual functions for all the different messages that need to be presented by the interface.

Figure 4
ATM UI Hierarchy



Consider also that each different transaction that the ATM can perform is encapsulated as a derivative of the class Transaction. Thus we might have classes such as DepositTransaction, WithdrawlTransaction, TransferTransaction, etc. Each of these objects issues message to the UI. For example, the DepositTransaction object calls the RequestDepositAmount member function of the UI class. Whereas the TransferTransaction object calls the RequestTransferAmount member function of UI. This corresponds to the diagram in Figure 5.

Notice that this is precisely the situation that the ISP tells us to avoid. Each of the transactions is using a portion of the UI that no other object uses. This creates the possibility that changes to one of the derivatives of Transaction will force coresponding change to the UI, thereby affecting all the other derivatives of Transaction, and every other class that depends upon the UI interface.

This unfortunate coupling can be avoided by segregating the UI interface into individual abstract base classes such as DepositUI, WithdrawUI and TransferUI. These abstract base classes can then be multiply inherited into the final UI abstract class. Figure 6 and Listing 6 show this model.

It is true that, whenever a new derivative of the Transaction class is created, a corresponding base class for the abstract UI class will be needed. Thus the UI class and all its derivatives must change. However, these classes are not widely used. Indeed, they are probably only used by main, or whatever process boots the system and creates the concrete UI instance. So the impact of adding new UI base classes is contained.

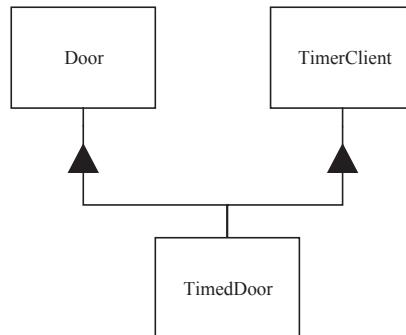
```
private:
    TimedDoor& itsTimedDoor;
};
```

However, this solution is also somewhat inelegant. It involves the creation of a new object every time we wish to register a timeout. Moreover the delegation requires a very small, but still non-zero, amount of runtime and memory. There are application domains, such as embedded real time control systems, in which runtime and memory are scarce enough to make this a concern.

Separation through Multiple Inheritance

Figure 3 and Listing 5 show how Multiple Inheritance can be used, in the *class form* of the ADAPTER pattern, to achieve the ISP. In this model, TimedDoor inherits from both Door and TimerClient. Although clients of both base classes can make use of TimedDoor, neither actually depend upon the TimedDoor class. Thus, they use the same object through separate interfaces.

Figure 3
Multiply Inherited Timed Door



Listing 5

Class Form of Adapter Pattern

```
class TimedDoor : public Door, public TimerClient
{
    public:
        virtual void TimeOut(int timeOutId);
};
```

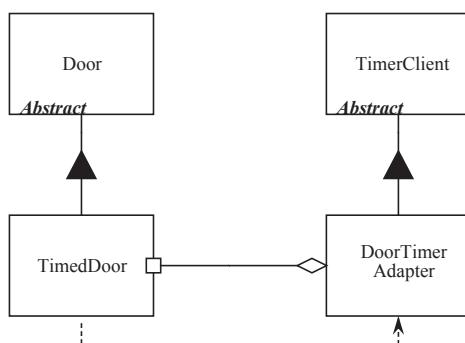
This solution is my normal preference. Multiple Inheritance does not frighten me. Indeed, I find it quite useful in cases such as this. The only time I would choose the solution in Figure 2 over Figure 3 is if the translation performed by the DoorTimerAdapter object were necessary, or if different translations were needed at different times.

Separation through Delegation

We can employ the *object form* of the ADAPTER² pattern to the TimedDoor problem. The solution is to create an adapter object that derives from TimerClient and delegates to the TimedDoor. Figure 2 shows this solution.

When the TimedDoor wants to register a timeout request with the Timer, it creates a DoorTimerAdapter and registers it with the Timer. When the Timer sends the TimeOut message to the DoorTimerAdapter, the DoorTimerAdapter delegates the message back to the TimedDoor.

Figure 2
Door Timer Adapter



This solution conforms to the ISP and prevents the coupling of Door clients to Timer. Even if the change to Timer shown in Listing 3 were to be made, none of the users of Door would be affected. Moreover, TimedDoor does not have to have the exact same interface as TimerClient. The DoorTimerAdapter can *translate* the TimerClient interface into the TimedDoor interface. Thus, this is a very general purpose solution. (See Listing 4)

Listing 4

Object Form of Adapter Pattern

```

class TimedDoor : public Door
{
public:
    virtual void DoorTimeOut(int timeOutId);
};

class DoorTimerAdapter : public TimerClient
{
public:
    DoorTimerAdapter(TimedDoor& theDoor)
        : itsTimedDoor(theDoor)
    {}

    virtual void TimeOut(int timeOutId)
    {itsTimedDoor.DoorTimeOut(timeOutId);}
}

```

2. Another GOF pattern. ibid.

and managers to the bone. When a change in one part of the program affects other completely unrelated parts of the program, the cost and repercussions of changes become unpredictable; and the risk of fallout from the change increases dramatically.

But it's just a recompile.

True. But recompiles can be very expensive for a number of reasons. First of all, they take time. When recompiles take too much time, developers begin to take shortcuts. They may hack a change in the “wrong” place, rather than engineer a change in the “right” place; because the “right” place will force a huge recompilation. Secondly, a recompilation means a new object module. In this day and age of dynamically linked libraries and incremental loaders, generating more object modules than necessary can be a significant disadvantage. The more DLLs that are affected by a change, the greater the problem of distributing and managing the change.

The Interface Segregation Principle (ISP)

CLIENTS SHOULD NOT BE FORCED TO DEPEND UPON INTERFACES THAT THEY DO NOT USE.

When clients are forced to depend upon interfaces that they don’t use, then those clients are subject to changes to those interfaces. This results in an inadvertent coupling between all the clients. Said another way, when a client depends upon a class that contains interfaces that the client does not use, but that other clients *do* use, then that client will be affected by the changes that those other clients force upon the class. We would like to avoid such couplings where possible, and so we want to separate the interfaces where possible.

Class Interfaces vs Object Interfaces

Consider the TimedDoor again. Here is an object which has two separate interfaces used by two separate clients; Timer, and the users of Door. These two interfaces *must* be implemented in the same object since the implementation of both interfaces manipulates the same data. So how can we conform to the ISP? How can we separate the interfaces when they must remain together?

The answer to this lies in the fact that clients of an object do not need to access it through the interface of the object. Rather, they can access it through delegation, or through a base class of the object.

Separate Clients mean Separate Interfaces.

Door and TimerClient represent interfaces that are used by completely different clients. Timer uses TimerClient, and classes that manipulate doors use Door. Since the clients are separate, the interfaces should remain separate too. Why? Because, as we will see in the next section, clients exert forces upon their server interfaces.

The backwards force applied by clients upon interfaces.

When we think of forces that cause changes in software, we normally think about how changes to interfaces will affect their users. For example, we would be concerned about the changes to all the users of TimerClient, if the TimerClient interface changed. However, there is a force that operates in the other direction. That is, sometimes it is the user that forces a change to the interface.

For example, some users of Timer will register more than one timeout request. Consider the TimedDoor. When it detects that the Door has been opened, it sends the Register message to the Timer, requesting a timeout. However, before that timeout expires the door closes; remains closed for awhile, and then opens again. This causes us to register a *new* timeout request before the old one has expired. Finally, the first timeout request expires and the TimeOut function of the TimedDoor is invoked. And the Door alarms falsely.

We can correct this situation by using the convention shown in Listing 3. We include a unique timeOutId code in each timeout registration, and repeat that code in the TimeOut call to the TimerClient. This allows each derivative of TimerClient to know which timeout request is being responded to.

Listing 3

Timer with ID

```
class Timer
{
    public:
        void Register(int timeout,
                      int timeOutId,
                      TimerClient* client);
};

class TimerClient
{
    public:
        virtual void TimeOut(int timeOutId) = 0;
};
```

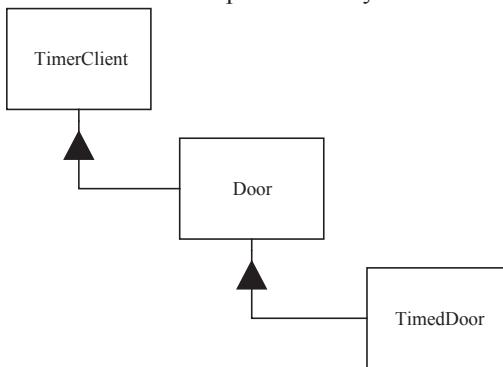
Clearly this change will affect all the users of TimerClient. We accept this since the lack of the timeOutId is an oversight that needs correction. However, the design in Figure 1 will also cause Door, and all clients of Door to be affected (i.e. at least recompiled) by this fix! Why should a bug in TimerClient have *any* affect on clients of Door derivatives that do not require timing? It is this kind of strange interdependency that chills customers

: The Interface Segregation Principle

When an object wishes to be informed about a timeout, it calls the Register function of the Timer. The arguments of this function are the time of the timeout, and a pointer to a TimerClient object whose TimeOut function will be called when the timeout expires.

How can we get the TimerClient class to communicate with the TimedDoor class so that the code in the TimedDoor can be notified of the timeout? There are several alternatives. Figure 1 shows a common solution. We force Door, and therefore TimedDoor, to inherit from TimerClient. This ensures that TimerClient can register itself with the Timer and receive the TimeOut message.

Figure 1
TimerClient at top of hierarchy



Although this solution is common, it is not without problems. Chief among these is that the Door class now depends upon TimerClient. Not all varieties of Door need timing. Indeed, the original Door abstraction had nothing whatever to do with timing. If timing-free derivatives of Door are created, those derivatives will have to provide nil implementations for the TimeOut method. Moreover, the applications that use those derivatives will have to #include the definition of the TimerClient class, even though it is not used.

Figure 1 shows a common syndrome of object oriented design in statically typed languages like C++. This is the syndrome of interface pollution. The interface of Door has been polluted with an interface that it does not require. It has been forced to incorporate this interface solely for the benefit of one of its subclasses. If this practice is pursued, then every time a derivative needs a new interface, that interface will be added to the base class. This will further pollute the interface of the base class, making it “fat”.

Moreover, each time a new interface is added to the base class, that interface must be implemented (or allowed to default) in derived classes. Indeed, an associated practice is to add these interfaces to the base class as *nil* virtual functions rather than pure virtual functions; specifically so that derived classes are not burdened with the need to implement them. As we learned in the second article of this column, such a practice violates the Liskov Substitution Principle (LSP), leading to maintenance and reusability problems.

words, the interfaces of the class can be broken up into groups of member functions. Each group serves a different set of clients. Thus some clients use one group of member functions, and other clients use the other groups.

The ISP acknowledges that there are objects that require non-cohesive interfaces; however it suggests that clients should not know about them as a single class. Instead, clients should know about abstract base classes that have cohesive interfaces. Some languages refer to these abstract base classes as “interfaces”, “protocols” or “signatures”.

In this article we will discuss the disadvantages of “fat” or “polluted” interface. We will show how these interfaces get created, and how to design classes which hide them. Finally we will present a case study in which the a “fat” interface naturally occurs, and we will employ the ISP to correct it.

Interface Pollution

Consider a security system. In this system there are Door objects that can be locked and unlocked, and which know whether they are open or closed. (See Listing 1).

Listing 1

```
Security Door
class Door
{
public:
    virtual void Lock() = 0;
    virtual void Unlock() = 0;
    virtual bool IsDoorOpen() = 0;
};
```

This class is abstract so that clients can use objects that conform to the Door interface, without having to depend upon particular implementations of Door.

Now consider that one such implementation. TimedDoor needs to sound an alarm when the door has been left open for too long. In order to do this the TimedDoor object communicates with another object called a Timer. (See Listing 2.)

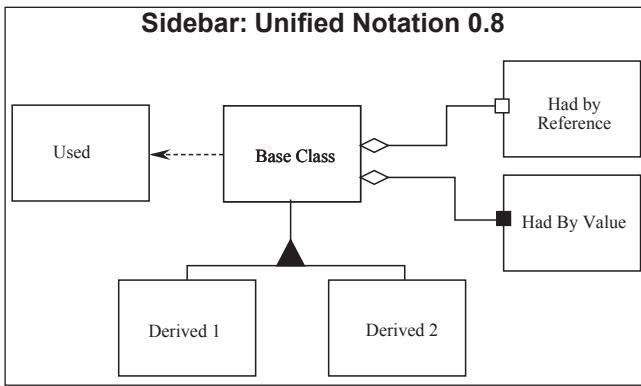
Listing 2

```
Timer
class Timer
{
public:
    void Register(int timeout, TimerClient* client);
};

class TimerClient
{
public:
    virtual void TimeOut() = 0;
};
```

The Interface Segregation Principle

This is the fourth of my *Engineering Notebook* columns for *The C++ Report*. The articles that appear in this column focus on the use of C++ and OOD, and address issues of software engineering. I strive for articles that are pragmatic and directly useful to the software engineer in the trenches. In these articles I make use of Booch's and Rumbaugh's new unified Modeling Langage (UML Version 0.8) for documenting object oriented designs. The sidebar provides a brief lexicon of this notation.



Introduction

In my last column (May 96) I discussed the principle of Dependency Inversion (DIP). This principle states that modules that encapsulate high level policy should not depend upon modules that implement details. Rather, both kinds of modules should depend upon abstractions. To be more succinct and simplistic, abstract classes should not depend upon concrete classes; concrete classes should depend upon abstract classes. A good example of this principle is the TEMPLATE METHOD pattern from the GOF¹ book. In this pattern, a high level algorithm is encoded in an abstract base class and makes use of pure virtual functions to implement its details. Derived classes implement those detailed virtual functions. Thus, the class containing the details depend upon the class containing the abstraction.

In this article we will examine yet another structural principle: the Interface Segregation Principle (ISP). This principle deals with the disadvantages of “fat” interfaces. Classes that have “fat” interfaces are classes whose interfaces are not cohesive. In other

1. *Design Patterns*, Gamma, et. al. Addison Wesley, 1995

change. And, since the abstractions and details are all isolated from each other, the code is much easier to maintain.

This article is an extremely condensed version of a chapter from my new book: *Patterns and Advanced Principles of OOD*, to be published soon by Prentice Hall. In subsequent articles we will explore many of the other principles of object oriented design. We will also study various design patterns, and their strengths and weaknesses with regard to implementation in C++. We will study the role of Booch's class categories in C++, and their applicability as C++ namespaces. We will define what "cohesion" and "coupling" mean in an object oriented design, and we will develop metrics for measuring the quality of an object oriented design. And after that, we will discuss many other interesting topics.



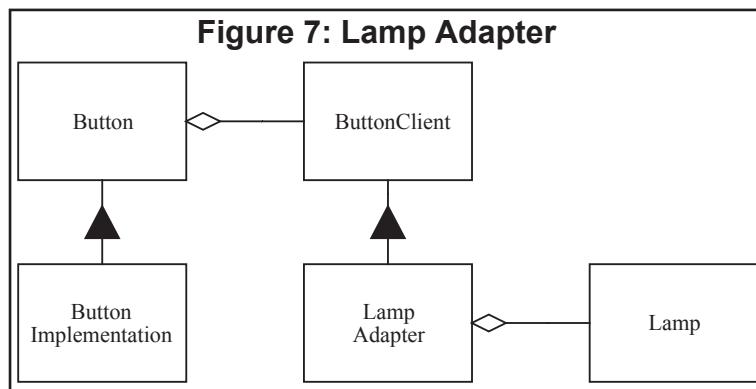
abstract button class³. The Button class knows nothing of the physical mechanism for detecting the user's gestures; and it knows nothing at all about the lamp. Those details are isolated within the concrete derivatives: ButtonImplementation and Lamp.

The high level policy in Listing 6 is reusable with any kind of button, and with any kind of device that needs to be controlled. Moreover, it is not affected by changes to the low level mechanisms. Thus it is robust in the presence of change, flexible, and reusable.

Extending the Abstraction Further

One could make a legitimate complaint about the design in Figure/Listing 6. The device controlled by the button must be derived from ButtonClient. What if the Lamp class comes from a third party library, and we cannot modify the source code.

Figure 7 demonstrates how the Adapter pattern can be used to connect a third party Lamp object to the model. The LampAdapter class simply translates the TurnOn and Turn-Off message inherited from ButtonClient, into whatever messages the Lamp class needs to see.



Conclusion

The principle of dependency inversion is at the root of many of the benefits claimed for object-oriented technology. Its proper application is necessary for the creation of reusable frameworks. It is also critically important for the construction of code that is resilient to

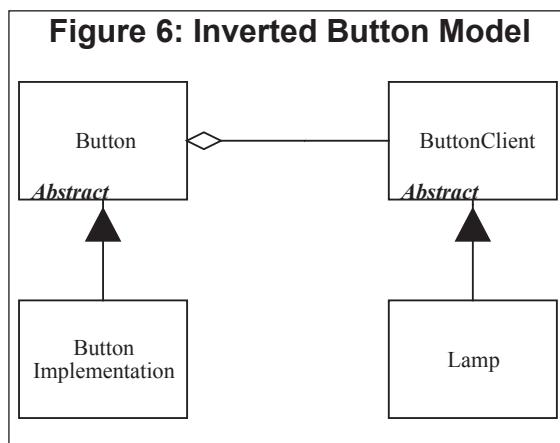
3. Aficionados of Patterns will recognize the use of the Template Method pattern in the Button Hierarchy. The member function: Button::Detect() is the template that makes use of the pure virtual function: Button::GetState(). See: *Design Patterns*, Gamma, et. al., Addison Wesley, 1995

Finding the Underlying Abstraction

What is the high level policy? It is the abstractions that underlie the application, the truths that do not vary when the details are changed. In the Button/Lamp example, the underlying abstraction is to detect an on/off gesture from a user and relay that gesture to a target object. What mechanism is used to detect the user gesture? Irrelevant! What is the target object? Irrelevant! These are details that do not impact the abstraction.

To conform to the principle of dependency inversion, we must isolate this abstraction from the details of the problem. Then we must direct the dependencies of the design such that the details depend upon the abstractions. Figure 6 shows such a design.

In Figure 6, we have isolated the abstraction of the Button class, from its detailed implementation. Listing 6 shows the corresponding code. Note that the high level policy is entirely captured within the



Listing 6: Inverted Button Model

```

-----buttonClient.h-----
class ButtonClient
{
public:
    virtual void TurnOn() = 0;
    virtual void TurnOff() = 0;
};
-----button.h-----
class ButtonClient;
class Button
{
public:
    Button(ButtonClient&);
    void Detect();
    virtual bool GetState() = 0;
private:
    ButtonClient* itsClient;
};
-----button.cc-----
#include button.h
#include buttonClient.h

Button::Button(ButtonClient& bc)
: itsClient(&bc) {}

-----button.h-----
void Button::Detect()
{
    bool buttonOn = GetState();
    if (buttonOn)
        itsClient->TurnOn();
    else
        itsClient->TurnOff();
}
-----lamp.h-----
class Lamp : public ButtonClient
{
public:
    virtual void TurnOn();
    virtual void TurnOff();
};
-----buttonImp.h-----
class ButtonImplementation
: public Button
{
public:
    ButtonImplementation(
        ButtonClient&,
        virtual bool GetState();
    );
};
  
```

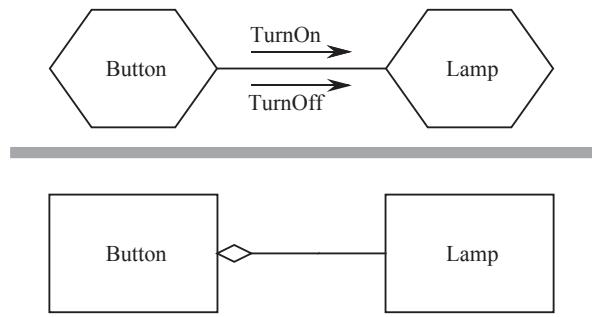
: The Dependency Inversion Principle

tor in a home security system. The Button object detects that a user has either activated or deactivated it. The lamp object affects the external environment. Upon receiving a TurnOn message, it illuminates a light of some kind. Upon receiving a TurnOff message it extinguishes that light. The physical mechanism is unimportant. It could be an LED on a computer console, a mercury vapor lamp in a parking lot, or even the laser in a laser printer.

How can we design a system such that the Button object controls the Lamp object? Figure 5 shows a naive model. The Button object simply sends the TurnOn and TurnOff message to the Lamp. To facilitate this, the Button class uses a “contains” relationship to hold an instance of the Lamp class.

Listing 5 shows the C++ code that results from this model. Note that the Button class depends directly upon the Lamp class. In fact, the button.cc module #includes the lamp.h module. This dependency implies that the button class must change, or at very least be recompiled, whenever the Lamp class changes. Moreover, it will not be possible to reuse the Button class to control a Motor object.

Figure 5, and Listing 5 violate the dependency inversion principle. The high level policy of the application has not been separated from the low level modules; the abstractions have not been separated from the details. Without such a separation, the high level policy automatically depends upon the low level modules, and the abstractions automatically depend upon the details.

Figure 5: Naive Button/Lamp Model**Listing 5: Naive Button/Lamp Code**

```

-----lamp.h-----
class Lamp
{
public:
    void TurnOn();
    void TurnOff();
};

-----button.h-----
class Lamp;
class Button
{
public:
    Button(Lamp& l) : itsLamp(&l) {}
    void Detect();
private:
    Lamp* itsLamp;
};

-----button.cc-----
#include "button.h"
#include "lamp.h"

void Button::Detect()
{
    bool buttonOn = GetPhysicalState();
    if (buttonOn)
        itsLamp->TurnOn();
    else
        itsLamp->TurnOff();
}
  
```

Using this model, Policy Layer is unaffected by any changes to Mechanism Layer or Utility Layer. Moreover, Policy Layer can be reused in any context that defines lower level modules that conform to the Mechanism Layer interface. Thus, by inverting the dependencies, we have created a structure which is simultaneously more flexible, durable, and mobile.

Separating Interface from Implementation in C++

One might complain that the structure in Figure 3 does not exhibit the dependency, and transitive dependency problems that I claimed. After all, Policy Layer depends only upon the *interface* of Mechanism Layer. Why would a change to the implementation of Mechanism Layer have any affect at all upon Policy Layer?

In some object oriented language, this would be true. In such languages, interface is separated from implementation automatically. In C++ however, there is no separation between interface and implementation. Rather, in C++, the separation is between the definition of the class and the definition of its member functions.

In C++ we generally separate a class into two modules: a .h module and a .cc module. The .h module contains the definition of the class, and the .cc module contains the definition of that class's member functions. The definition of a class, in the .h module, contains declarations of all the member functions and member variables of the class. This information goes beyond simple interface. All the utility functions and private variables needed by the class are also declared in the .h module. These utilities and private variables are part of the implementation of the class, yet they appear in the module that all users of the class must depend upon. Thus, in C++, implementation is not automatically separated from interface.

This lack of separation between interface and implementation in C++ can be dealt with by using purely abstract classes. A purely abstract class is a class that contains nothing but pure virtual functions. Such a class is pure interface; and its .h module contains no implementation. Figure 4 shows such a structure. The abstract classes in Figure 4 are meant to be purely abstract so that each of the layers depends only upon the *interface* of the subsequent layer.

A Simple Example

Dependency Inversion can be applied wherever one class sends a message to another. For example, consider the case of the Button object and the Lamp object.

The Button object senses the external environment. It can determine whether or not a user has “pressed” it. It doesn’t matter what the sensing mechanism is. It could be a button icon on a GUI, a physical button being pressed by a human finger, or even a motion detect-



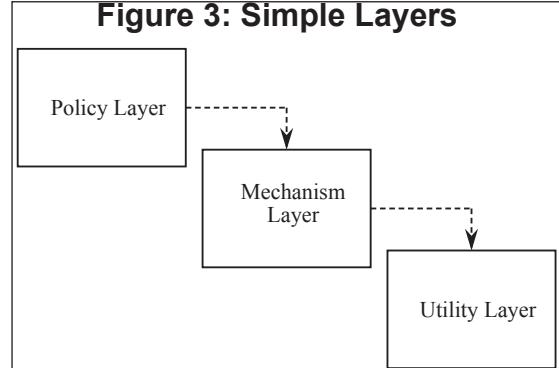
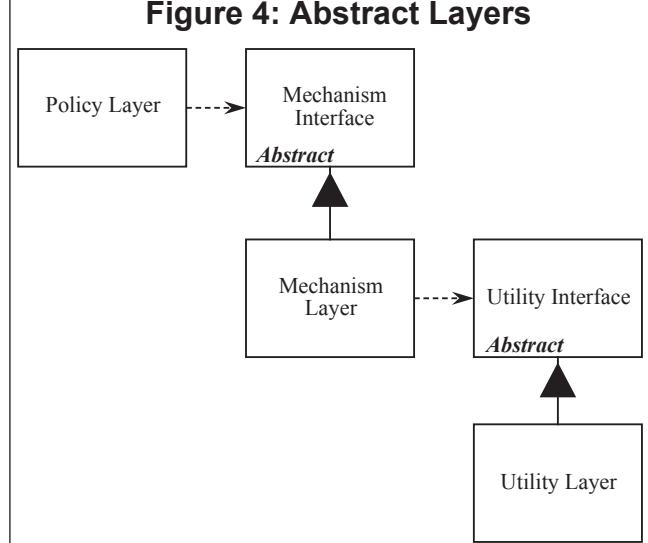
: The Dependency Inversion Principle

This is the principle that is at the very heart of framework design.

Layering

According to Booch², "...all well structured object-oriented architectures have clearly-defined layers, with each layer providing some coherent set of services though a well-defined and controlled interface." A naive interpretation of this statement might lead a designer to produce a structure similar to Figure 3. In this diagram the high level policy class uses a lower level Mechanism; which in turn uses a detailed level utility class. While this may look appropriate, it has the insidious characteristic that the Policy Layer is sensitive to changes all the way down in the Utility Layer. *Dependency is transitive*. The Policy Layer depends upon something that depends upon the Utility Layer, thus the Policy Layer transitively depends upon the Utility Layer. This is very unfortunate.

Figure 4 shows a more appropriate model. Each of the lower level layers are represented by an abstract class. The actual layers are then derived from these abstract classes. Each of the higher level classes uses the next lowest layer through the abstract interface. Thus, none of the layers depends upon any of the other layers. Instead, the layers depend upon abstract classes. Not only is the transitive dependency of Policy Layer upon Utility Layer broken, but even the direct dependency of Policy Layer upon Mechanism Layer is broken.

Figure 3: Simple Layers**Figure 4: Abstract Layers**

2. *Object Solutions*, Grady Booch, Addison Wesley, 1996, p54

declared in stdio.h. Moreover, the IO drivers that are eventually invoked also depend upon the abstractions declared in stdio.h. Thus the device independence within the stdio.h library is another example of dependency inversion.

Now that we have seen a few examples, we can state the general form of the DIP.

Listing 4: Copy using stdio.h

```
#include <stdio.h>
void Copy()
{
    int c;
    while((c = getchar()) != EOF)
        putchar(c);
}
```

The Dependency Inversion Principle

A. HIGH LEVEL MODULES SHOULD NOT DEPEND UPON LOW LEVEL MODULES. BOTH SHOULD DEPEND UPON ABSTRACTIONS.

B. ABSTRACTIONS SHOULD NOT DEPEND UPON DETAILS. DETAILS SHOULD DEPEND UPON ABSTRACTIONS.

One might question why I use the word “inversion”. Frankly, it is because more traditional software development methods, such as Structured Analysis and Design, tend to create software structures in which high level modules depend upon low level modules, and in which abstractions depend upon details. Indeed one of the goals of these methods is to define the subprogram hierarchy that describes how the high level modules make calls to the low level modules. Figure 1 is a good example of such a hierarchy. Thus, the dependency structure of a well designed object oriented program is “inverted” with respect to the dependency structure that normally results from traditional procedural methods.

Consider the implications of high level modules that depend upon low level modules. It is the high level modules that contain the important policy decisions and business models of an application. It is these models that contain the identity of the application. Yet, when these modules depend upon the lower level modules, then changes to the lower level modules can have direct effects upon them; and can force them to change.

This predicament is absurd! It is the high level modules that ought to be forcing the low level modules to change. It is the high level modules that should take precedence over the lower level modules. High level modules simply should not depend upon low level modules in any way.

Moreover, it is high level modules that we want to be able to reuse. We are already quite good at reusing low level modules in the form of subroutine libraries. When high level modules depend upon low level modules, it becomes very difficult to reuse those high level modules in different contexts. However, when the high level modules are independent of the low level modules, then the high level modules can be reused quite simply.



: The Dependency Inversion Principle

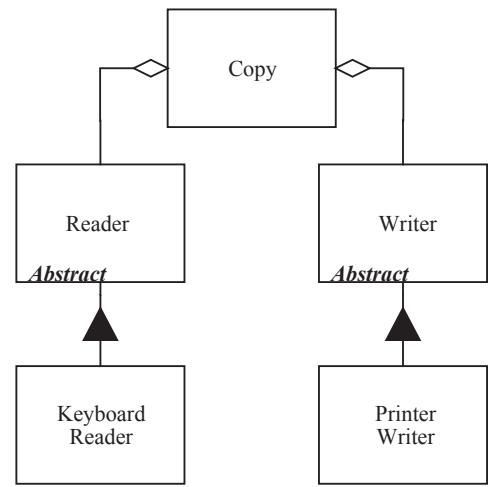
input device to any output device. OOD gives us a mechanism for performing this *dependency inversion*.

Consider the simple class diagram in Figure 2. Here we have a “Copy” class which contains an abstract “Reader” class and an abstract “Writer” class. One can easily imagine a loop within the “Copy” class that gets characters from its “Reader” and sends them to its “Writer” (See Listing 3). Yet this “Copy” class does not depend upon the “Keyboard Reader” nor the “Printer Writer” at all. Thus the dependencies have been *inverted*; the “Copy” class depends upon abstractions, and the detailed readers and writers depend upon the same abstractions.

Now we can reuse the “Copy” class, independently of the “Keyboard Reader” and the “Printer Writer”. We can invent new kinds of “Reader” and “Writer” derivatives that we can supply to the “Copy” class. Moreover, no matter how many kinds of “Readers” and “Writers” are created, “Copy” will depend upon none of them. There will be no interdependencies to make the program fragile or rigid. And Copy() itself can be used in many different detailed contexts. It is mobile.

Device Independence

By now, some of you are probably saying to yourselves that you could get the same benefits by writing Copy() in C, using the device independence inherent to `stdio.h`; i.e. `getchar` and `putchar` (See Listing 4). If you consider Listings 3 and 4 carefully, you will realize that the two are logically equivalent. The abstract classes in Figure 3 have been replaced by a different kind of abstraction in Listing 4. It is true that Listing 4 does not use classes and pure virtual functions, yet it still uses abstraction and polymorphism to achieve its ends. Moreover, it still uses dependency inversion! The Copy program in Listing 4 does not depend upon any of the details it controls. Rather it depends upon the abstract facilities

Figure 2: The OO Copy Program**Listing 3: The OO Copy Program**

```

class Reader
{
    public:
        virtual int Read() = 0;
};

class Writer
{
    public:
        virtual void Write(char) = 0;
};

void Copy(Reader& r, Writer& w)
{
    int c;
    while((c=r.Read()) != EOF)
        w.Write(c);
}
  
```

The two low level modules are nicely reusable. They can be used in many other programs to gain access to the keyboard and the printer. This is the same kind of reusability that we gain from subroutine libraries.

Listing 1. The Copy Program

```
void Copy()
{
    int c;
    while ((c = ReadKeyboard()) != EOF)
        WritePrinter(c);
}
```

However the “Copy” module is not reusable in any context which does not involve a keyboard or a printer. This is a shame since the intelligence of the system is maintained in this module. It is the “Copy” module that encapsulates a very interesting policy that we would like to reuse.

For example, consider a new program that copies keyboard characters to a disk file. Certainly we would like to reuse the “Copy” module since it encapsulates the high level policy that we need. i.e. it knows how to copy characters from a source to a sink. Unfortunately, the “Copy” module is dependent upon the “Write Printer” module, and so cannot be reused in the new context.

We could certainly modify the “Copy” module to give it the new desired functionality. (See Listing 2). We could add an “if” statement to its policy and have it select between the “Write Printer” module and the “Write Disk” module depending upon some kind of flag. However this adds new interdependencies to the system. As time goes on, and more and more devices must participate in the copy program, the “Copy” module will be littered with if/else statements and will be dependent upon many lower level modules. It will eventually become rigid and fragile.

Listing 2. The “Enhanced” Copy Program

```
enum OutputDevice {printer, disk};
void Copy(outputDevice dev)
{
    int c;
    while ((c = ReadKeyboard()) != EOF)
        if (dev == printer)
            WritePrinter(c);
        else
            WriteDisk(c);
}
```

Dependency Inversion

One way to characterize the problem above is to notice that the module containing the high level policy, i.e. the Copy() module, is dependent upon the low level detailed modules that it controls. (i.e. WritePrinter() and ReadKeyboard()). If we could find a way to make the Copy() module independent of the details that it controls, then we could reuse it freely. We could produce other programs which used this module to copy characters from any

: The Dependency Inversion Principle

predicted by the designers or maintainers, the impact of the change cannot be estimated. This makes the cost of the change impossible to predict. Managers, faced with such unpredictability, become reluctant to authorize changes. Thus the design becomes officially rigid.

Fragility is the tendency of a program to break in many places when a single change is made. Often the new problems are in areas that have no conceptual relationship with the area that was changed. Such fragility greatly decreases the credibility of the design and maintenance organization. Users and managers are unable to predict the quality of their product. Simple changes to one part of the application lead to failures in other parts that appear to be completely unrelated. Fixing those problems leads to even more problems, and the maintenance process begins to resemble a dog chasing its tail.

A design is immobile when the desirable parts of the design are highly dependent upon other details that are not desired. Designers tasked with investigating the design to see if it can be reused in a different application may be impressed with how well the design would do in the new application. However if the design is highly interdependent, then those designers will also be daunted by the amount of work necessary to separate the desirable portion of the design from the other portions of the design that are undesirable. In most cases, such designs are not reused because the cost of the separation is deemed to be higher than the cost of redevelopment of the design.

Example: the “Copy” program.

A simple example may help to make this point. Consider a simple program that is charged with the task of copying characters typed on a keyboard to a printer. Assume, furthermore, that the implementation platform does not have an operating system that supports device independence. We might conceive of a structure for this program that looks like Figure 1:

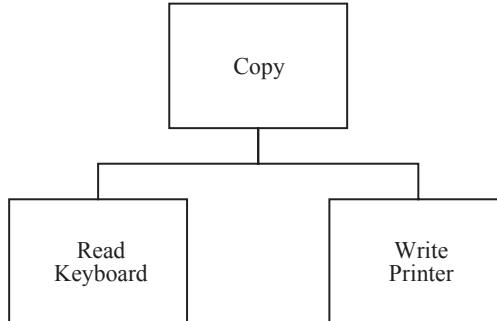
Figure 1. Copy Program.

Figure 1 is a “structure chart”¹. It shows that there are three modules, or subprograms, in the application. The “Copy” module calls the other two. One can easily imagine a loop within the “Copy” module. (See Listing 1.) The body of that loop calls the “Read Keyboard” module to fetch a character from the keyboard, it then sends that character to the “Write Printer” module which prints the character.

1. See: *The Practical Guide To Structured Systems Design*, by Meilir Page-Jones, Yourdon Press, 1988

What goes wrong with software?

Most of us have had the unpleasant experience of trying to deal with a piece of software that has a “bad design”. Some of us have even had the much more unpleasant experience of discovering that we were the authors of the software with the “bad design”. What is it that makes a design bad?

Most software engineers don’t set out to create “bad designs”. Yet most software eventually degrades to the point where someone will declare the design to be unsound. Why does this happen? Was the design poor to begin with, or did the design actually degrade like a piece of rotten meat? At the heart of this issue is our lack of a good working definition of “bad” design.

The Definition of a “Bad Design”

Have you ever presented a software design, that you were especially proud of, for review by a peer? Did that peer say, in a whining derisive sneer, something like: “Why’d you do it *that way?*”. Certainly this has happened to me, and I have seen it happen to many other engineers too. Clearly the disagreeing engineers are not using the same criteria for defining what “bad design” is. The most common criterion that I have seen used is the TNTWI-WHDI or “That’s not the way I would have done it” criterion.

But there is one set of criteria that I think all engineers will agree with. A piece of software that fulfills its requirements and yet exhibits any or all of the following three traits has a bad design.

1. It is hard to change because every change affects too many other parts of the system. (Rigidity)
2. When you make a change, unexpected parts of the system break. (Fragility)
3. It is hard to reuse in another application because it cannot be disentangled from the current application. (Immobility)

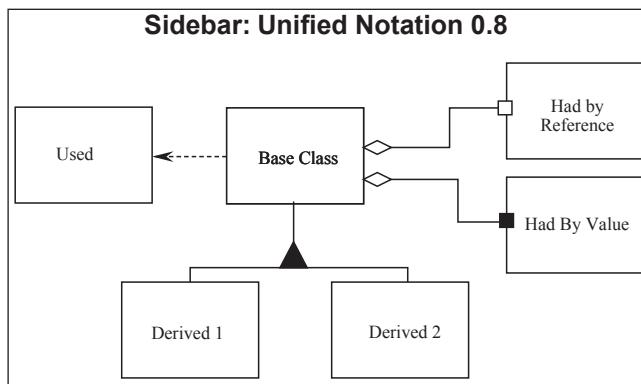
Moreover, it would be difficult to demonstrate that a piece of software that exhibits none of those traits, i.e. it is flexible, robust, and reusable, and that also fulfills all its requirements, has a bad design. Thus, we can use these three traits as a way to unambiguously decide if a design is “good” or “bad”.

The Cause of “Bad Design”.

What is it that makes a design rigid, fragile and immobile? It is the interdependence of the modules within that design. A design is rigid if it cannot be easily changed. Such rigidity is due to the fact that a single change to heavily interdependent software begins a cascade of changes in dependent modules. When the extent of that cascade of change cannot be

The Dependency Inversion Principle

This is the third of my *Engineering Notebook* columns for *The C++ Report*. The articles that will appear in this column will focus on the use of C++ and OOD, and will address issues of software engineering. I will strive for articles that are pragmatic and directly useful to the software engineer in the trenches. In these articles I will make use of Booch's and Rumbaugh's new *unified* notation (Version 0.8) for documenting object oriented designs. The sidebar provides a brief lexicon of this notation.



Introduction

My last article (Mar, 96) talked about the Liskov Substitution Principle (LSP). This principle, when applied to C++, provides guidance for the use of public inheritance. It states that every function which operates upon a reference or pointer to a base class, should be able to operate upon derivatives of that base class without knowing it. This means that the virtual member functions of derived classes must expect no more than the corresponding member functions of the base class; and should promise no less. It also means that virtual member functions that are present in base classes must also be present in the derived classes; and they must do useful work. When this principle is violated, the functions that operate upon pointers or references to base classes will need to check the type of the actual object to make sure that they can operate upon it properly. This need to check the type violates the Open-Closed Principle (OCP) that we discussed last January.

In this column, we discuss the structural implications of the OCP and the LSP. The structure that results from rigorous use of these principles can be generalized into a principle all by itself. I call it "The Dependency Inversion Principle" (DIP).