גירסה 1.00 – 5.2.2005

Visitor Design Pattern

ניר אדר

מסמך זה הורד מהאתר http://underwar.livedns.co.il.
אין להפיץ מסמך זה במדיה כלשהי, ללא אישור מפורש מאת המחבר.
מחבר המסמך איננו אחראי לכל נזק, ישיר או עקיף, שיגרם עקב השימוש במידע המופיע במסמך, וכן לנכונות התוכן של הנושאים המופיעים במסמך. עם זאת, המחבר עשה את מירב המאמצים כדי לספק את המידע המדויק והמלא ביותר.

כל הזכויות שמורות ל**ניר אדר**

Nir Adar
Email: underwar@hotmail.com
Home Page: http://underwar.livedns.co.il

אנא שלחו תיקונים והערות אל המחבר.

Visitor Design Pattern

מסמך זה מציג את תבנית התכן Visitor ודוגמא לשימוש בה. מהי תבנית תכן? תבנית תכן היא פיתרון כללי לבעיה, שניתן להתאים אותו לבעיות רבות בתחום התכנות. המושג הומצא לראשונה בהקשר לתחום הבניין, אך כיום, בעקבות ספר שנכתב על ידי ארבעה אנשים המכונים The gang of four, המושג מתייחס בעיקר לפתרונות שונים לבעיות בתחום התוכנה והתכנות מונחה העצמים.
Visitor אינה התבנית הפשוטה ביותר, אך בה מסמך זה יעסוק. בעתיד אני מתכוון לכתוב מסמך על מספר Design Patterns נוספים, ויתכן שאשלב גירסה כלשהי של מסמך זה בתוכו, ולכן אשמח לשמוע הערות על מסמך זה.
הקוד במסמך זה יודגם בשפת C# אבל כמובן שהרעיון מתאים גם עבור שפות אחרות.

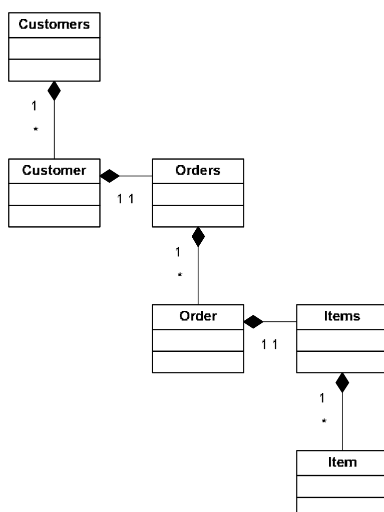
הגדרת התבנית ומטרתה

הצורך: אנחנו מעוניינים להגדיר פעולה שתבוצע על מספר מחלקות, אשר לא בהכרח יש ביניהן קשר. כמו כן, איננו מעוניינים שהפעולה תהיה חלק מהממשק של המחלקות (או שלחילופין – איננו מסוגלים שהפעולה תהיה חלק מהממשק, למשל אם לא נתון לנו מימוש המחלקות).

התבנית Visitor מאפשרת לנו להגדיר פעולה מבלי לשנות את המחלקות עליהן היא מופעלת.
התבנית יעילה גם במקרה בו ניתן לייצג את האובייקטים כגרף, כאשר אובייקטים מכילים אובייקטים אחרים.

דוגמא

נתחיל בתיאור דוגמא לבעיה ולפתרון שהתבנית מציעה, ולאחר מכן גם נתאר את התבנית באופן כללי.



נביט בתרשים המחלקות הבא.

Customers, Orders, Items הם מכולות,
ו-Cunstomer, Order, Item הם אלמנטים יחידים. נניח שאנחנו רוצים לממש, למשל, פעולה על העץ שהיא BuildReport.

ברור כי ביצירת דו"ח כל אלמנט ידווח על עצמו דברים שונים. לכ נכתוב פעולת BuildReport עבור כל אלמנט, שתדפיס את המידע הרלוונטי לגביו. בנוסף, הפעולה תגרום לסיור בבנים שלו ולהדפסה גם שלהם.

מיד נראה כיצד נממש דבר כזה, אבל לפני זה נענה על שאלה שיכולה להתעורר: מדוע לא להגדיר פונקציה אחת, מחוץ לכל המחלקות, בשם BuildReport שתבצע את כל הפעולות הנדרשות על העץ?

לדוגמא:

```
public string BuildReport(ICollection collection)
{
   StringWriter report = new StringWriter();

   foreach (object item in collection)
   {
      if (item is Customer)
      {
         Customer customer = item as Customer;
         report.WriteLine(customer.FirstName + ", " + customer.LastName);
         report.WriteLine(BuildReport(customer.Orders));
      }
      else if (item is Order)
      {
         Order order = item as Order;
         report.WriteLine(order.Date + " - " + order.Amount);
         report.WriteLine(BuildReport(order.Items));
      }
      else if (item is Item)
      {
         Item orderitem= item as Item;
         report.WriteLine(orderitem.Code + " - " + orderitem.Quantity);
      }
   }

   report.ToString();
}
```

פתרון זה עובד, אבל סובל ממספר בעיות:
עבור מבנה זה, שהוא פשוט יחסית, כבר הופיע לנו משפט if… else מורכב. במידה ובמערכת היו רכיבים נוספים, המשפט היה נעשה מורכב והיינו מקבלים "קוד ספגטי".
ייתרה מזאת, במידה והיו n אובייקטים במערכת, בחירת הפעולות המתאימות עבור האובייקט יתבצעו במקרה הגרוע ב-$O(n)$.

אם נפעל בשיטה שהצענו, הוספת מתודה BuildReport לכל אחד מהאלמנטים, אז הבעיה תהפוך לפשוטה יותר. אולם גם שיטה זו בעייתית. נניח כעת שנרצה להוסיף פעולה נוספת לכל האלמנטים – נצטרך שוב לערוך ולשנות את כל המחלקות שעבורן מוגדרת הפעולה.

הפתרון: הפרדת המתודות המתאימות לפעולות מהאיברים, והשמתן בממשק נפרד, שנכנהו Visitor. Visitor מגדיר למעשה פעולה כללית לביצוע על האיברים. ה-interface מגדיר פעולה עבור כל אחד מהאיברים בהם נבקר. מימוש של ה-interface צריך להגדיר מימוש לכל אחת מהפעולות הנ"ל.

לדוגמא:

```
public interface IVisitor
{
   void VisitCustomer(Customer customer);
   void VisitOrder(Order order);
   void VisitItem(Item item);
}
```

דרך נוספת נפוצה להגדרת הממשק היא להגדיר את כל המתודות בשם Visit (מתודות חופפות), כאשר ההבדל ביניהן הוא לפי הארגומנט שהן מקבלות.

מלבד ממשק זה קיים ממשק נוסף, שממנו יירשו כל האלמנטים בהם נבקר. אלמנטים היורשים מממשק זה הם אלמנטים שאנו מגדירים שהם "ניתנים לביקור".

```
public interface IVisitable
{
   void Accept(IVisitor visitor);
}
```

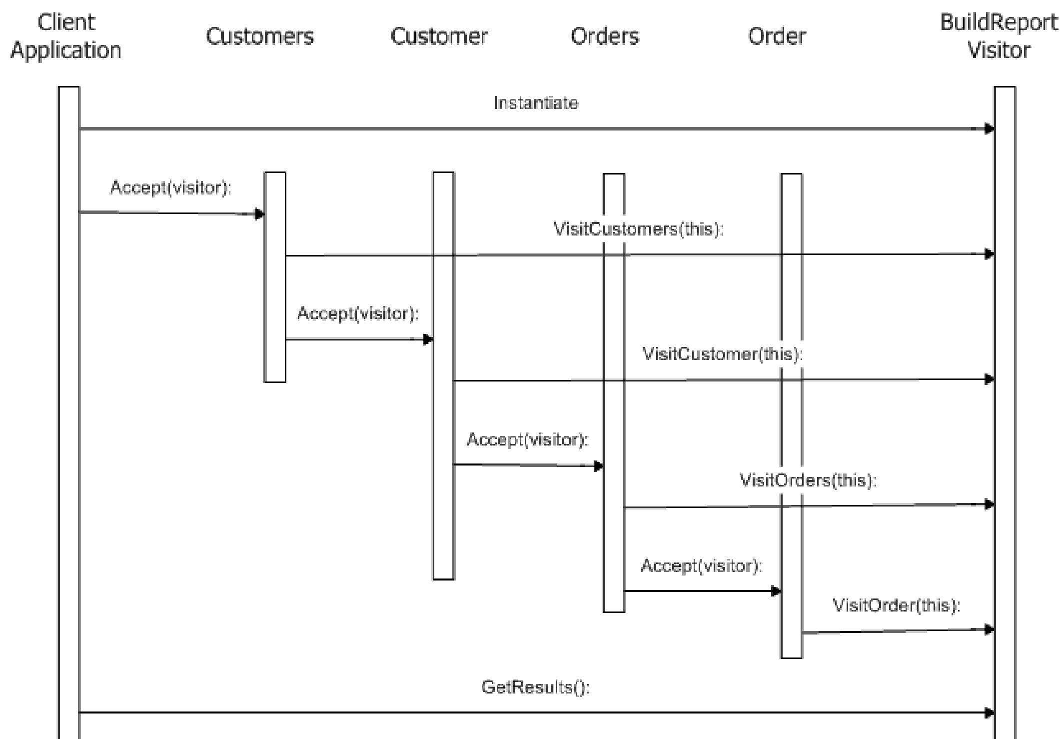
כל אלמנט קונקרטי יממש פעולה זו.
אלמנט קצה, כגון Customer, יכול להציע מימוש כזה:

```
public virtual void Accept(IVisitor visitor)
{
   visitor.VisitCustomer(this);
   _orders.Accept(visitor)
}
```

אלמנט מכולה, כגון Customers, יכול להציע את המימוש הבא:

```
public virtual void Accept(IVisitor visitor)
{
   visitor.VisitCustomers(this);
   foreach (IVisitable visitable in this._children)
      visitable.Accept(visitor);
}
```

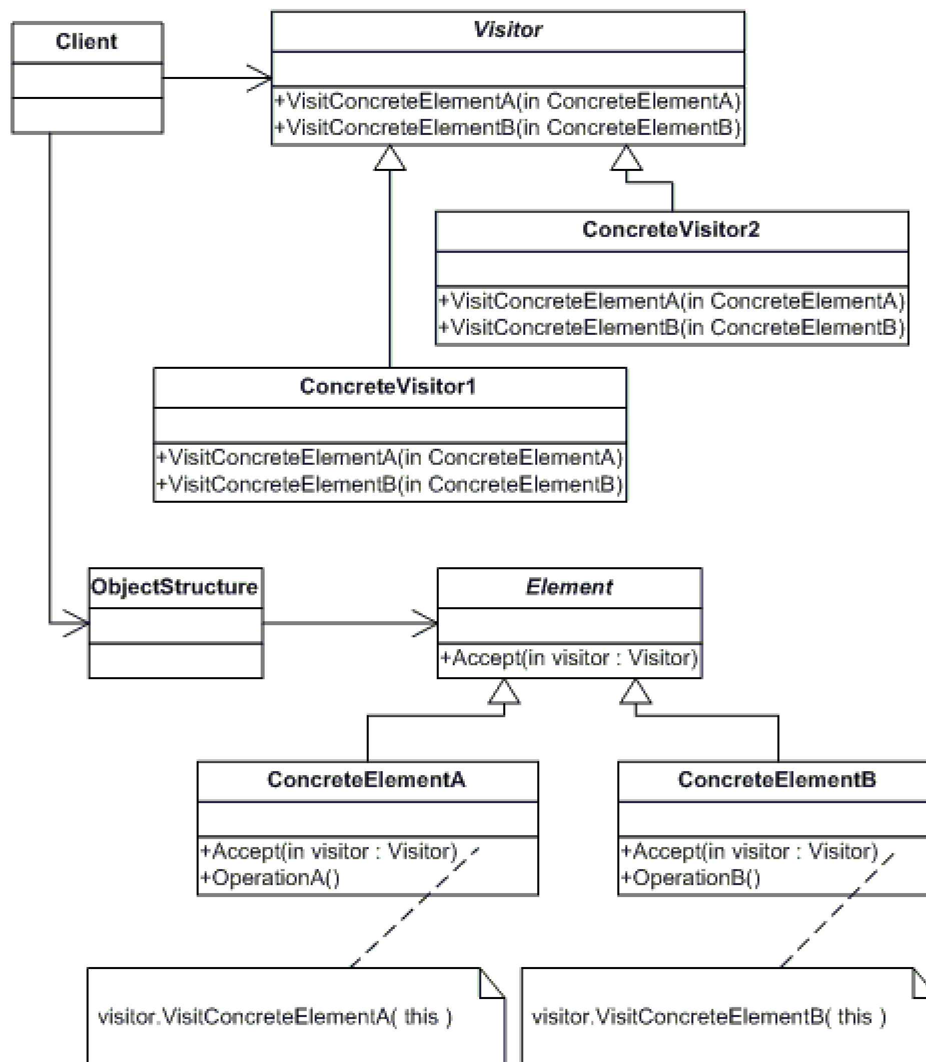
תרשים Sequence המתאים לביקור:



על ידי שימוש ב-()Visit, אנו משיגים את בחירת הפעולה לביצוע על האובייקט ב- $O(1)$.

התבנית הכללית

נציג כעת את התבנית באופן כללי. תרשים המחלקות המתאים הינו:



המשתתפים

Visitor

מגדיר פעולת Visit עבור כל מחלקה מסוג ConcreteElement. שם הפעולה והחתימה מזהים את המחלקה שתשלח את הודעת ה-Visit אל ה-Visitor. עובדה זו נותנת ל-visitor את היכולת לזהות את המחלקה של האלמנט אותו הוא הולך לבקר. לאחר שהוא זיהה אותו, הוא יכול לגשת לשדות של אותו אלמנט דרך ממשק האלמנט.

Concrete Visitor

ממש כל אחת מהפעולות שהוגדרו על ידי ה-Visitor. כל פעולה מממשת אלגוריתם שהוגדר עבור אותה מחלקה ספציפית המתאימה לפעולה.
המחלקה ConcreteVisitor שומר את הקשר האלגוריתם ושומרת את המשתנים המגדירים את המצב (state) שלו. למשל – המחלקה יכולה לשמור משתנה בו מתבצעת סכימה של תוצאה וכדו'.

Element

מגדיר פעולה בשם Accept המקבלת Visitor כארגומנט.

ConcreteElement

מממש פעולת Accept המקבלת Visitor כארגומנט.

ObjectStructure

מחלקה המכילה אוסף אלמנטים, ומאפשרת ל-Visitor לעבור על האלמנטים השונים שבה.

מסקנות

- הוספת פעולות חדשות למחלקות נעשית בקלות.
- ניתן לאחד קבוצות של פעולות דומות במחלקה בודדת שתהיה אחראית עליהן.
- הוספת אלמנט חדש לגרף המחלקות היא קשה, מכיוון שיהיה צורך לשנות את כל ה-visitors הקיימים.
- המחלקה visitor יכלה לשמור על המצב שלה במהלך הסיור על ידי שדות בתוכה.
- ה-visitors חייבים לממש את כל הממשק אפילו אם הם לא משתמשים בסוג מסויים של concrete element.

מקורות

1. Data & Object Factory (2002), "**Design Patterns: Visitor**"
2. Daniel Cazzulino (2003), "**Visitor Design Pattern: revisited for .NET**"
3. Antonio Garca, "**The Visitor Design Pattern**"