מרכז ההדרכה עיטם 2000
תמיכה ועדכונים
עדכון מס' 95
דצמבר 2004

Decorator Pattern

עדכון זה מפורסם לרגל הוצאת הספר "**Design Patterns**", הכולל סקירה של ה- patterns התקניים כמו גם מספר טכניקות שימושיות נוספות. הדוגמאות בספר הן משפות התכנות **C++**, **Java**, **C#**. כמו כן מובאות מספר דוגמאות גם מ- **Python**, כשפת עצמים דינמית.

Design Pattrens הוא תחום המהווה גישה **מונחית-בעיות** במסגרת תיכון מונחה עצמים (Object Oriented Design). בגישה זו קיים קטלוג של תבניות בעיות וטכניקות תיכון לפתרונן. קטלוג ה-patterns מספק למעשה שפה "גבוהה" - שפת patterns - לתיאור ולתיעוד של מערכות עצמים מורכבות.

במסגרת מאמר זה נסקור את ה- **Decorator** - זהו Pattern שימושי בהרחבת פונקציונליות נתונה במחלקת הבסיס, באופן פולימורפי.

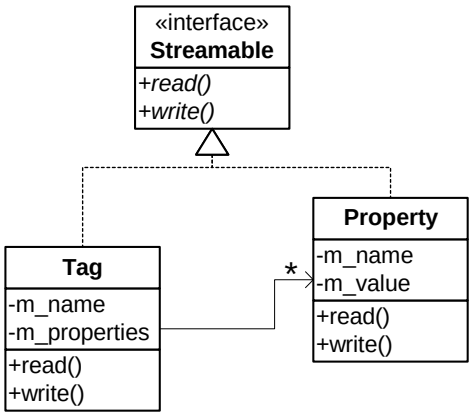
עוד על הספר ניתן לקרוא באתר האינטרנט בכתובת:

http://www.mh2000.co.il/dp

Decorator

בעיה

נתונה מערכת לתמיכה בשפת מסמכים כלשהי (כגון HTML, XML). מערכת זו כוללת בין היתר מחלקה המייצגת תג:



הממשק Streamable יאפשר קריאה וכתיבה של עצמי Tag ו- Property מ/-אל עצמי stream (שיכולים להיות ק/פ תקני, קבצים, דפי אינטרנט, ערוצי תקשורת וכו'). (מימוש הממשק Streamable יאפשר גם

שימוש פשוט באופרטורים "<<", "<<" תוך שימוש בטכניקת Double-Dispatch).

- מאפייני תג:
  - לתג יש שם
  - התג מעוטר בקצוותיו בסימנים מיוחדים, למשל [aTag], <aTag>, {aTag}.
  - תג יכול לכלול וקטור תכונות עם ערכים, לדוגמא: <font size=3 color=red>

קוד המחלקות:

```
class Streamable
{
public:
	virtual void read(istream &is) = 0;
	virtual void write(ostream &os) const = 0;
};

inline istream &operator>>(istream& is, Streamable &s)
		{ s.read(is); return is;}
inline ostream &operator<<(ostream& os, const Streamable &s)
		{ s.write(os); return os;}

class Property : public Streamable
{
	string m_name, m_value;
public:
	Property(string n, string v) : m_name(n), m_value(v) {}
	virtual void read(istream &is)
		{ char ch; is >> m_name >> ch >> m_value; }
	virtual void write(ostream &os) const
		{ os << m_name << "=" << m_value; }
};

class Tag : public Streamable
{
	string m_name;
	vector<Property> m_properties;
	static char SEP;
public:
	Tag() {}
	Tag(string n) : m_name(n) {}
	virtual void add(string n, string v)
		{ m_properties.push_back(Property(n,v)); }
	virtual void read(istream &is)
	{
		char ch;
		is >> m_name >> ch;
		for(int i=0; i<m_properties.size(); i++)
			is >> m_properties[i] >> ch;
	}
	virtual void write(ostream &os) const
	{
		os << m_name << SEP;
		for(int i=0; i<m_properties.size(); i++)
			os << m_properties[i] << SEP;
	}
};

char Tag::SEP = ' ';
```

דוגמא לקוד משתמש:

```
// simple tag with properties
Tag t("font");
t.add("size", "3");
t.add("color", "red");
cout << t << endl;
```

הפלט:

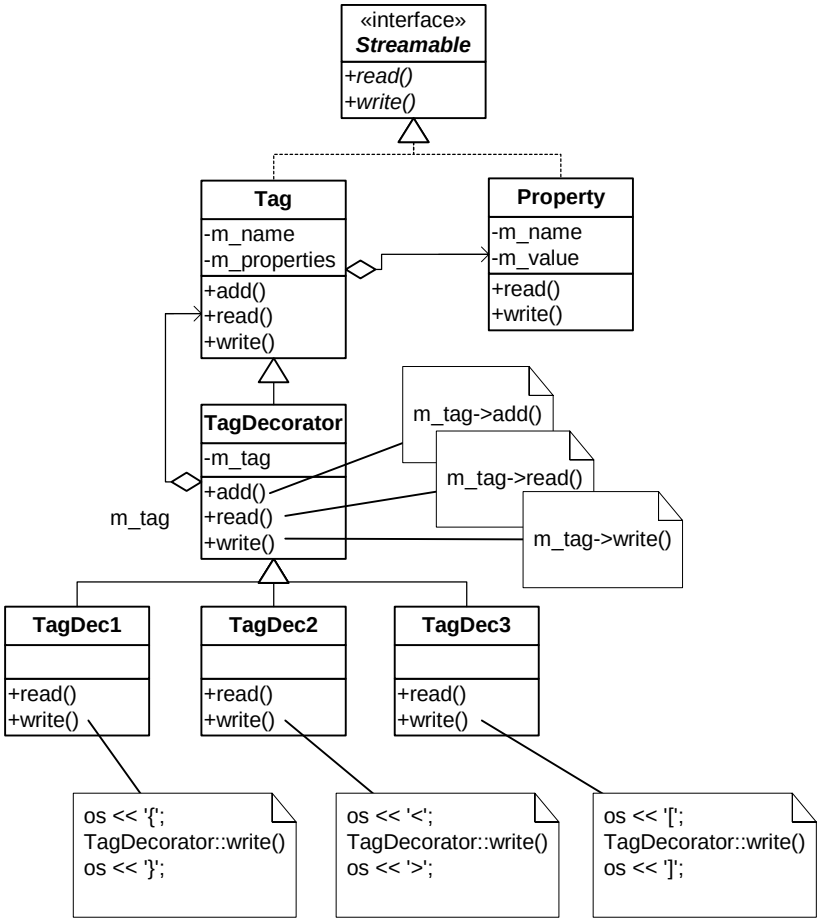
```
font size=3 color=red
```

המטרה:

לאפשר ייצוג תגים מהצורה [<aTag>], <[aTag]>, {<aTag>}, {[<aTag>]}.

פתרון

- נגדיר מחלקה אחת לכל סוג עיטור אפשרי.
- מחלקות העיטור (Decoration) יירשו ממחלקת בסיס משותפת, Decorator.
- המחלקה TagDecorator יורשת מהמחלקה Tag: היא מספקת מבנה הכלה רקורסיבית (בדומה ל Composite).



המחלקה TagDecorator מטפלת גם באיתחול המצביע למחלקת הבסיס, Tag. קוד המחלקה:

```
class TagDecorator : public Tag
{
        Tag * m_tag;
public:
```

```
        TagDecorator(Tag *pTag) : m_tag(pTag) {}
        TagDecorator(string n) { m_tag = new Tag(n);}
        void add(string n, string v) { m_tag->add(n,v); }
        virtual void read(istream &is)    { is >> *m_tag; }
        virtual void write(ostream &os) const { os << *m_tag;   }
};
```

המחלקות הנגזרות מ- TagDecorator מבצעות קריאה לפונקציה המתאימה של הבסיס, תוך ביצוע פעולות הדקורציה לפני ואחרי.

קוד המחלקה TagDec1 לדוגמא:

```
class TagDec1: public TagDecorator
{
public:
        TagDec1(Tag *pTag) : TagDecorator(pTag) {}
        TagDec1(string n) : TagDecorator(n) {}
        virtual void read(istream &is)
        {
                char ch;
                is >> ch; assert(ch=='{');
                TagDecorator::read(is);
                is >> ch; assert(ch=='}');
        }
        virtual void write(ostream &os) const
        {
                os << '{';
                TagDecorator::write(os);
                os << '}';
        }
};
```

באופן דומה מוגדרות שאר מחלקות העיטור.

דוגמא לקוד משתמש:

```
        // {hello}
        Tag *pTag1 = new TagDec1("hello");

        // <hello>
        Tag *pTag2 = new TagDec2("hello");

        // {<hello>}
        Tag *pTag3 = new TagDec1(new TagDec2("hello"));

        // [{<hello>}]
        Tag *pTag4 = new TagDec3(new TagDec1(new TagDec2("hello")));
        pTag4->add("size", "7");
        pTag4->add("color", "blue");

        // print all
        cout << *pTag1 << endl;
        cout << *pTag2 << endl;
        cout << *pTag3 << endl;
        cout << *pTag4 << endl;
```

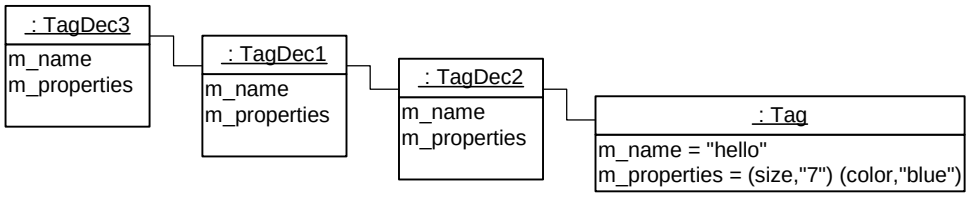
והפלט:

```
{hello}
<hello>
{<hello>}
[{<hello size=7 color=blue>}]
```

כדי להבין את מבנה העצמים, נתבונן בתרשים העצמים הנוצרים כתוצאה מהשורה

```
// [{<hello>}]
Tag *pTag4 = new TagDec3(new TagDec1(new TagDec2("hello")));
```

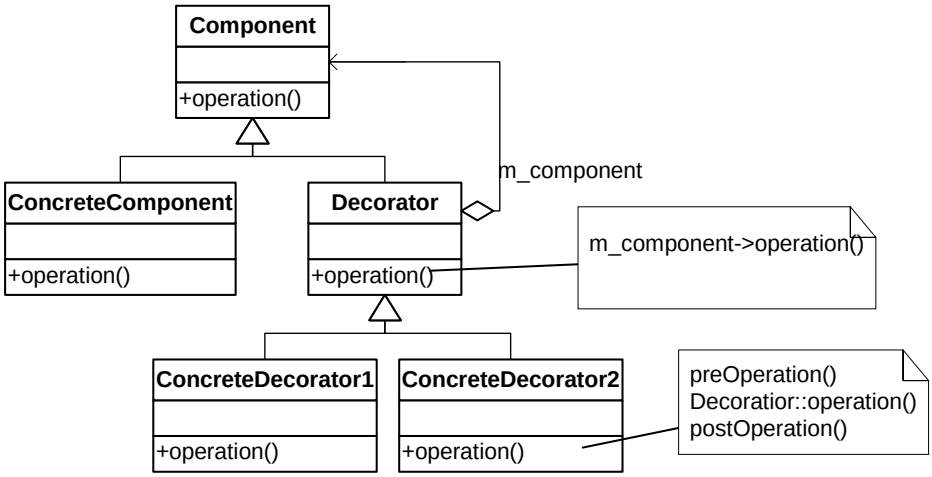
התרשים:



הכללה

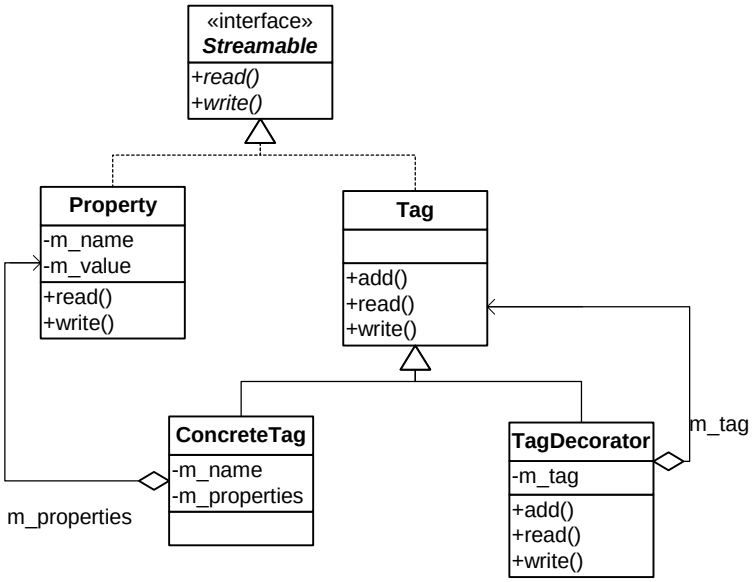
ה-**Decorator** מאפשר הרחבת פונקציונליות של עצם ע"י הוספת "עיטור" (דקורציה) לשירותים באופן דינמי. ה- Decorator משמש כמחלקת בסיס למחלקות העיטור, וממש את מנגנון האגרגציה (הרקורסיבית) וה Forwarding למחלקת הבסיס (בדומה ל- Composite).

תרשים מחלקות:



- יתרונות:
  - גמישות בקביעת קומבינציה של דקורציות
  - שינוי התנהגות של עצם באופן דינמי ע"י הוספת (או הסרת) עצמי דקורציה
- חסרונות:
  - מורכבות בהבנת הקוד
  - הרבה עצמים קטנים עם מידע מיותר. ניתן לפתור בעיה זו ע"י הגדרת ה- Component כאבסטרקטי, והגדרת הנתונים במחלקות הנגזרות בלבד.

לדוגמא, כדי לייעל את תכנית ה Tags, ניתן לבנות את צמרת עץ המחלקות כך:



כעת, כל עצם ממחלקה הנגזרת מ- TagDecorator הוא בעל sizeof של מצביע!

אופן יצירת העצמים:

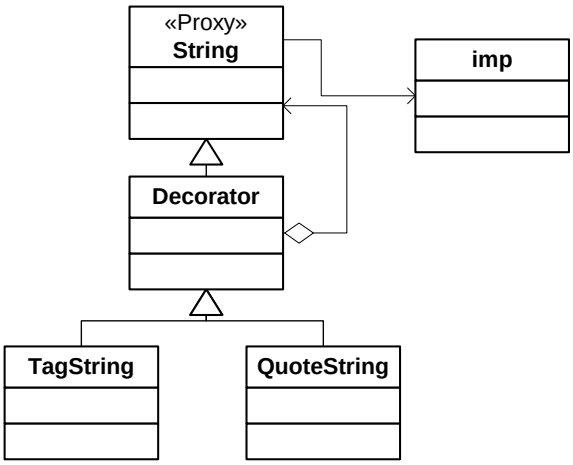
*Tag * ptag = new TagDec3(new TagDec2(new ConcreteTag));*

<u>שילוב עם Proxy</u>

נניח שנדרש לספק פעולות עיטור למחלקה String, שכזכור, מוגדרת בד"כ כ- **Proxy** (ראה/י מאמר מס. 23).

השאלה: כיצד ייראו שני ה- Patterns במשולב? כלומר, כיצד ניתן להגדיר מחלקת String הממומשת ע"י Proxy עם יכולות דקורציה בדומה ל- Tag לעיל?

תשובה: נגדיר Proxy Decorator - הפרדת מחלקת הבסיס של ה- Decorator למחלקה מייצגת (Proxy) ולמחלקה מממשת:



מכיוון שה- Proxy מסתיר את עובדת היותו כזה מכל העולם (כולל מחלקות נגזרות), מובטח לנו שה- Decorator לא מושפע ואינו צריך לעבור שינוי מיוחד בשל כך.

דוגמא: קלט/פלט ב- Java

קלט / פלט ב- Java, בדומה ל- C++, ממומש ע"י מנגנון זרמי קלט/פלט (**IO streams**). זרמי הקלט/פלט מסווגים עפ"י 3 קטגוריות עיקריות:

- עפ"י סוג המידע המועבר בהם - בתים או תווים.
- עפ"י תפקיד - ביצוע פעולה כלשהי על המידע כגון: דחיסה (ZIP), סינון, ספירת שורות וכו'.
- עפ"י מדיית הקלט/פלט (מקלדת/מסך, זכרון, קבצים, רשת).

במטרה לספק פתרון לכל קומבינציה של **נתונים - עיבוד - מדייה**, עושים שימוש ב- Decorator. לדוגמא, בכדי לבצע קלט טיפוסים בסיסיים (DataInputStream) עם חציצה (BufferedInputStream) מתוך קובץ (FileInputStream), יוצרים את העצם:

```
DataInputStream in = new DataInputStream(
                        new BufferedInputStream(
                                new FileInputStream("file.txt")));
```

באופן דומה, בכדי לבצע שמירה של עצמים (ObjectOutputStream) לקובץ (FileOutputStream), יוצרים את העצם:

```
ObjectOutputStream obj_out = new ObjectOutputStream(
                        new FileOutputStream("obj_file.dat");
```

דוגמא: קלט/פלט ב- #NET/C.

באופן דומה, גם #NET/C. עושה שימוש ב- Decorator בספריית הקלט/פלט שלה. לדוגמא, פתיחת Stream לקריאה עם הצפנה מסוג CryptoStream תראה כך:

```
CryptoStream s = new CryptoStream(
                new FileStream("file.dat", FileMode.Open),
                        new FromBase64Transform(),
                                CryptoStreamMode.Read);
```

דוגמא: Decorator ב- Python

ניתן לעשות שימוש בתמיכה המובנית של Python בהפנייה, להגדרת Decorator בפשטות.

דוגמא:

```
class Tag:
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return self.name

class TagDecorator(Tag):
    def __init__(self, name):
        self.tag = Tag(name)
    def __init__(self, tag):
        self.tag = tag

class TagDec1 (TagDecorator):
    def __str__(self):
        return "<" + self.tag.__str__() + ">"

class TagDec2 (TagDecorator):
    def __str__(self):
        return "{" + self.tag.__str__() + "}"

#client:
t1 = TagDec1(TagDec2("hello"))
print t1
```

```
t2 = TagDec1(TagDec2(TagDec1("hello")))
print t2
```

הפלט:

```
<{hello}>
<{<hello>}>
```

הסבר:

- מחלקת ה- Tag מוגדרת כאן בפשטות כמכילה את שם התג בלבד
- TagDecorator יורשת מ- Tag ע"י רישום שמה בהגדרה:

```
class TagDecorator(Tag):
```

- TagDecorator כוללת שני constructors: אחד המקבל מחרוזת ומייצר עצם Tag כ- member, והשני מקבל עצם תג קיים ומציב אותו ל- member שלו:

```
    def __init__(self, name):
        self.tag = Tag(name)
    def __init__(self, tag):
        self.tag = tag
```

- שתי המחלקות Dec1 ו- Dec2 יורשות מ- TagDecorator והן מגדירות אך ורק את הפונקציה __str__(), המחזירה את ייצוג המחרוזת שלהן.

כדאי לשים לב ל- 2 עובדות במימוש Decorator ב- Python:

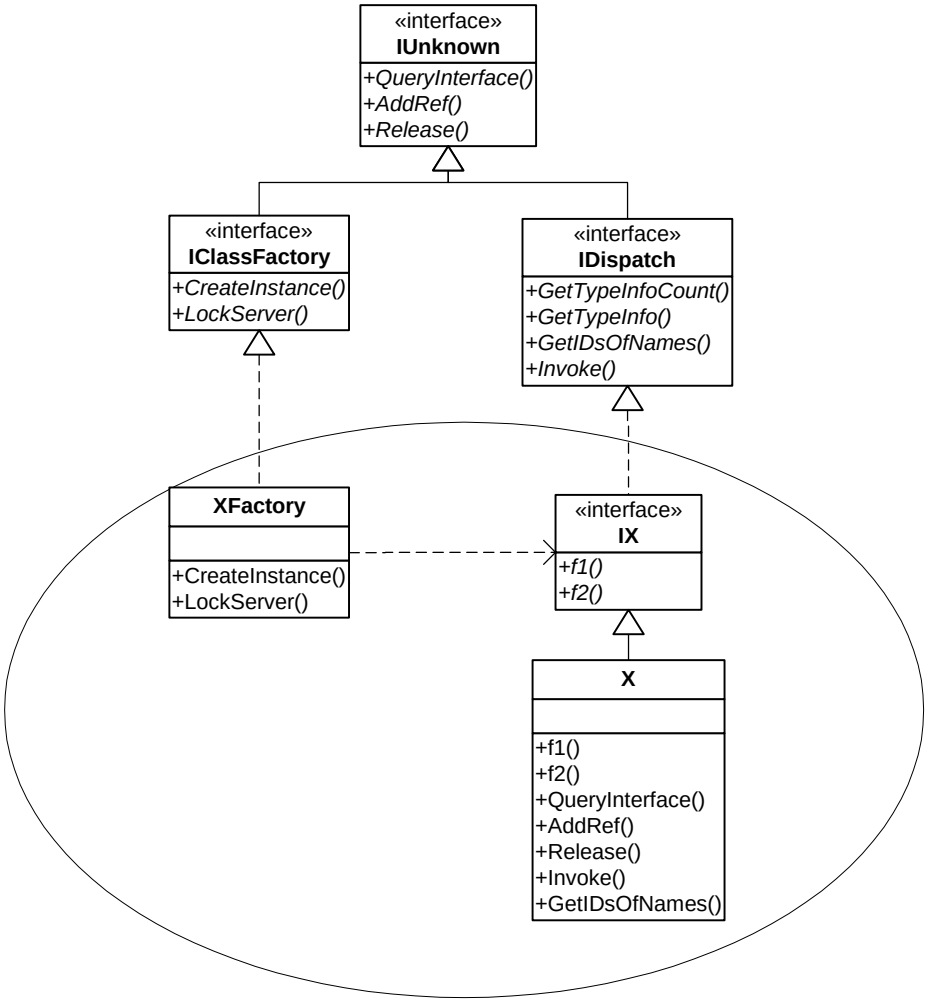
- מכיוון שהשדה name מוסף באופן דינמי רק ב- constructor שבו הוא מתקבל כפרמטר, ולא בשני, לא קיימת הבעיה שלעיל - הרבה עצמים קטנים עם מידע מיותר.
- ה- constructors נורשים וניתנים לשימוש במחלקות הנגזרות כאילו הוגדרו בהן! זה מאוד מפשט את כתיבתן.

דוגמא: COM-ATL

ATL, ספריית ה- Templates של Microsoft לפיתוח רכיבי COM, עושה שימוש מתוחכם ב- Decorator תוך שילוב Template.הטכניקה הבסיסית שבשימוש בספרייה היא שילוב של פולימורפיזם+Template.

לדוגמא, ללא ATL, אם רוצים לבנות מחלקת רכיב בשם X, עם השירותים f1() ו- f2(), יש צורך לממש את ממשקי הבסיס של COM:

COM



בתוך האליפסה מוצגים הממשקים והמחלקות שעל בונה הרכיב לממש.

ATL מממשת את שלושת ממשקי COM הבסיסיים:

- ממשק IUnknown ממומש ע"י **CComObjectRoot** (או באופן כללי יותר, ע"י **CComObjectRootEx**<ThreadModel>).

- ממשק IClassFactory ומודל האגרגציה ממומשים ע"י **CComCoClass**<>: זהו Template שהפרמטרים שלו הם מחלקת הרכיב ומזהה הרכיב:

```
template< class T, const CLSID* pclsid>
class CComCoClass;
```

- ממשק IDispatch ממומש בברירת מחדל ע"י מחלקת הבסיס **IDispatchImpl**<>:

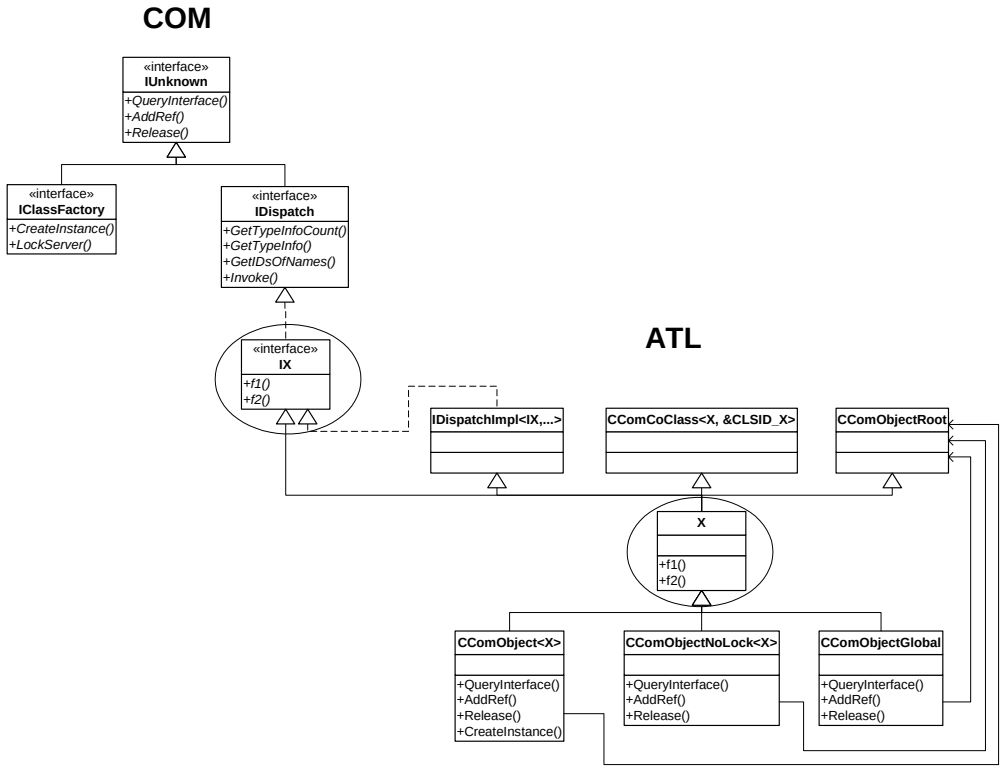
```
template<class T, const IID* piid, const GUID* plibid,...>
class IDispatchImpl : public T
```

- *templates של מחלקות מהצורה* ***CComObjectXXX***<> משמשים כ- **Decorators**: הם מממשים את הפונקציות של IUnknown בקומבינציות שונות (לפי מודל האגרגציה/נעילה/Threading), ע"י הפנייה (Forwarding) לבסיס, **CComObjectRoot**.

CComObject<> מממש גם את CreateInstance() של הממשק IClassFactory:

```
template <class Base>
class CComObject : public Base;
```

כך ניתן לבנות את הרכיב X, תוך שימוש ב- ATL :



(נדרש לממש רק את X ו- IX המוקפות באליפסה).

כך מוכרזת מחלקת הרכיב :

```
class X :
        public IX,
        public CComObjectRoot,
        public CComCoClass<X, &CLSID_X >,
        public IDispatchImpl<IX, &IID_IX, &LIBID_X>
{
        ...
public:
        STDMETHOD(f1)();
        STDMETHOD(f2)();
};
```

וקוד הלקוח נראה כך :

```
// client.cpp
HRESULT hret;
CLSID clsid;
wchar_t progid[] = L"XAPP.X.1";

hret = CLSIDFromProgID(progid, &clsid);
CoInitialize(0);

// create the COM object
IDispatch *pIDispatch = 0;
hret = CoCreateInstance(clsid, 0, CLSCTX_INPROC_SERVER, IID_IDispatch,
                (void **) &pIDispatch);

// What is returned from this call in pIDispatch is an instance of CComObject<X>
```

```
// can now use pIDispatch with both IUnknown/Automation interfaces
```

התוצאה היא שהמחלקה הנורשת ביותר אינה מחלקת המשתמש אלא template של ספריית ATL!

כל הזכויות שמורות © מרכז ההדרכה **עיטם** 2000