מרכז ההדרכה 2000
תמיכה ועדכונים
עדכון מס' 23
יום רביעי 07 פברואר 2001

שימוש ב- ProxyPattern בתכנות מונחה עצמים

DesignPatterns (תבניות תכן) הוא מהתחומים החמים היום בתחום פיתוח מונחה עצמים. הצורך במתן פתרונות לבעיות חוזרות ונשנות בתכנון של מערכות תוכנה מונחות עצמים, הביא להגדרתם של בעיות ופתרונות "סטנדרטיים".

אחד ה- **DesignPatterns** הידועים הוא ה- **Proxy**, והוא משמש בבעיה הבאה: נתייחס למחלקת String הממשת את האופרטורים + (שרשור) ו- = (הצבה):

```
classString
{
public:
	String(constchar*s=	0);
	String(constString&right);
	constchar*get()const{returnstr;}
	char&operator[](intpos);
	Stringoperator +(constString&str)const;
	String&operator=(constString&right);
	~String(){if(str)deletestr;}
private:
	char*str;
};
```

מימוש נאיבי של האופרטורים הנ"ל וה- CopyC'tor יהיה כך:

- אופרטור החיבור מחזיר עצם String על המחסנית **ע"י ערך** (מדוע?):

```
StringString::operator +(constString&right)const
{
	Stringret;
	char*s;
	if(!right.str)
		return*this;
	if(!str)
		returnright;
	s=newchar[strlen(str)	+strlen(right.str)  + 1];
	strcpy(s,str);
	strcat(s,right.str);
```

```
        ret.str=s;
        returnret;
}
```

– אופרטור ההצבה מבצע העתקה של עצם המחרוזת הימני לשמאלי:

```
String&String::operator=(constString&right)
{
        if(&right==this)
                return*this;
        if(str)
                deletes tr;
        if(right.str)
        {
                str=newchar[strlen(right.str)    + 1];
                strcpy(str,right.str);
        }
        else
                str=  0;
        return*this;
}
```

– ה- Copyc'tor ממומש כך:

```
String::String(constString&right)
{
        if(right.str)
        {
                str=newchar[strlen(right.str)    + 1];
                strcpy(str,right.str);
        }
        else
                str=  0;
}
```

בעיית השיכפול

נבחן כעת את קוד המשתמש הבא:

```
Strings 1("hello");
Strings 2,s 3,s 4(s1);
s2=s  1;
s3=s  1 +s 2;
```

שאלות:

1) כמה עצמי String נוצרים במהלך התכנית?
2) כמה מחרוזות שונות יש בתכנית?

תשובות:

1) ראשית נוצרים ארבעה עצמי String המוגדרים בתחילת התכנית.

String s1("hello");
String s2,s3,s4(s1);

בפעולת החיבור, s1 +s 2, נוצר עצם זמני על המחסנית, והוא מוחזר בסוף:

```
String String::operator +(const String& right) const
{
	String ret;
	...
	return ret;
}
```

לכן, פעולה זו יוצרה בעצם שני עצמים: ret וההעתק שלו המוחזר ע"י ערך (By-value), ולכן נוצר ע"י ה- copy constructor.

2) עפ"י קוד המשתמש, בסוף התכנית ערכי המחרוזות הם

s1=s2=s4="hello"
s3="hellohello"

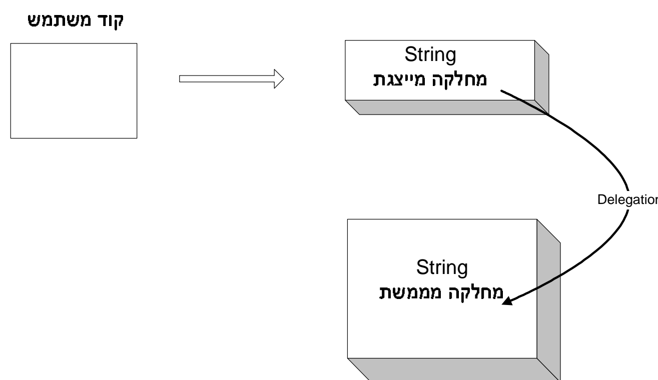
כלומר קיימות שתי מחרוזות שונות בתכנית.

מבחינת יעילות התכנית, אנו מגלים במחלקת המחרוזת שתי בעיות: עודף עצמים המיוצרים באופרטורים ובפונקציות המחזירות עצם ע"י ערך (אופרטור +), ושיכפול תוכן זיכרון של עצמי מחרוזת שבפועל הן בעלות ערך זהה (אופרטור =). הבעייה חריפה במיוחד עבור מחלקות כגון מחרוזות ומערכים, המחזיקים בממוצע חוצצי זיכרון גדולים.

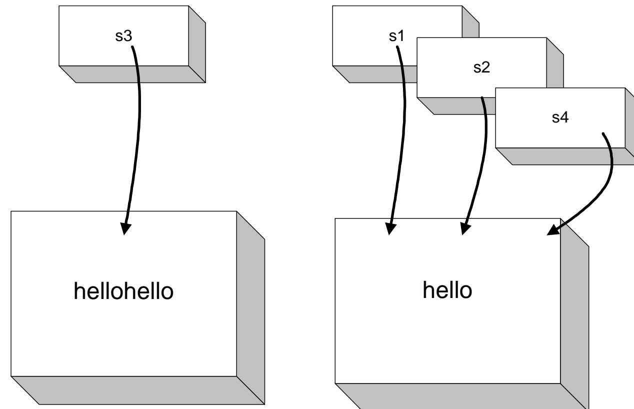
פתרון הבעייה: Proxy

פתרון הבעיה הכפולה, כפי שהוא מוכר בספרות Design Patterns, הוא שימוש ב- **Proxy**: ה- **Proxy** הוא מחלקה "קלה" המייצגת מחלקה "כבדה" יותר ומונעת העתקות מיותרות שלה.

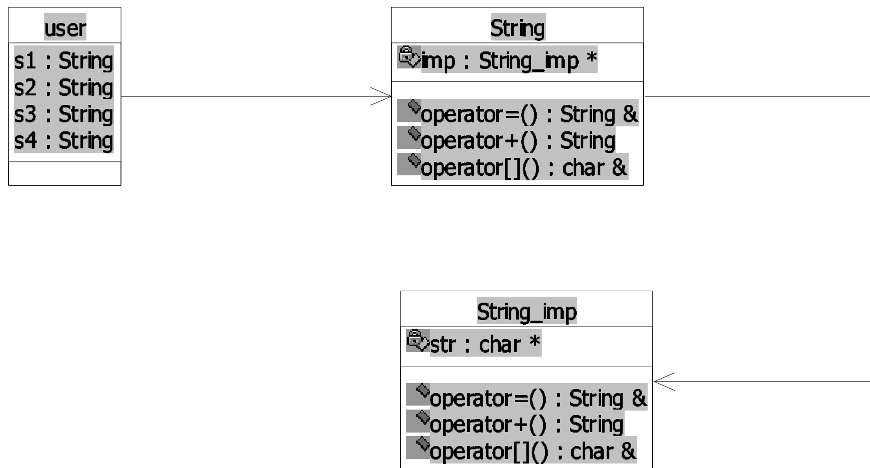
הרעיון: קוד המשתמש "מכיר" רק את המחלקה המייצגת (Proxy) והוא אינו יודע על קיומה של המחלקה האמיתית, המחלקה המממשת. כל הפעולות המבוצעות על המחלקה המייצגת **מואצלות** (**Delegated**) אל המחלקה המממשת:



כאשר מבוצעת העתקת מחרוזות, המחלקה המייצגת היא המועתקת ולא המחלקה המממשת: למשל, בתכנית המשתמש הנ"ל, תרשים העצמים יהיה



תרשים המחלקות הבא (תרשים UML) מראה את מבנה המחלקות והקשר ביניהן:



קוד המחלקות:

- המחלקה המממשת, מחלקת String_imp:

```
class String_imp
{
public:
	String_imp(constchar*str);
	String_imp(constString_imp&);
	constchar*get()const{returnstr;}
	char&operator[](intpos);
	String_imp*operator +(constString_imp&)const;
```

```
        String_imp&operator=(constString_imp&);
        ~String_imp(){deletestr;}
private:
        char*str;
};
```

מחלקה זו מממשת בפועל את פעולות הקצאות הזיכרון וההעתקות.

- המחלקה המייצגת, String :

```
class String
{
public:
        String(constchar*str=    0);
        String(constString&str)       :imp(str.imp){}
        constchar*get()const{returnimp?imp         ->get():"";}
        Stringoperator +(constString&str)const;
        char&operator[ ](intpos);
        String&operator=(constString&);
private:
        String_imp*imp;
        String(String_imp*other_imp)          {imp=other_imp;}
};
```

כפי שניתן לראות, המחלקה String מחזיקה מצביע לעצם מ- String_imp ומאצילה/שולחת אליו את בקשות הלקוח. האופרטור + מוגדר כך במחלקה המייצגת :

```
StringString::operator +(constString&str)const
{
        if(!str.imp)
                return*this;
        if(!imp)
                returnstr;
        return*imp   +*str.imp;
}
```

תרגיל : ממש/י האופרטור + עבור המחלקה המממשת, String_imp.

האופרטור = מוגדר כך במחלקה המייצגת :

```
String&String::operator=(constString&str)
{
        if(&str!=this)
        {
                if(imp)
                        imp->decRc();
                imp=str.imp;
                if(imp)
                        imp->incRc();
        }
        return*this;
```

```
}
```

התוצאה: העצמים היחידים המועתקים ו"מטיילים" על המחסנית הם עצמי String, אולם אלו עצמים קלים - למעשה ה- sizeof של עצם מהמחלקה String הוא sizeof של מצביע!

מניית מצביעים (ReferenceCount)

המחלקה String - על שני חלקיה - עדיין לא שלמה: עצמי המחלקה String_imp אינם משוחררים. לשם כך נצטרך להוסיף:

- מניית מצביעים (ReferenceCount) במחלקה המממשת.
  - מניית מצביעים משמעותה ספירת מספר המצביעים למחלקה נתונה.
  - עצם מהמחלקה המממשת יחזיק משתנה שלם שימנה את מספר המצביעים לעצם.
  - כאשר מצביע זה יגיע ל- 0, העצם ישחרר את עצמו מהזכרון
  - בכדי למנוע טעות אפשרית מצד קוד המשתמש, ה- destructor יוגדר כ- private
- המחלקה המייצגת תקדם את מונה המצביעים ב- constructor שלה, ותחסר אותו ב- destructor.

לאחר תוספת מניית המצביעים, כך ייראו המחלקות:

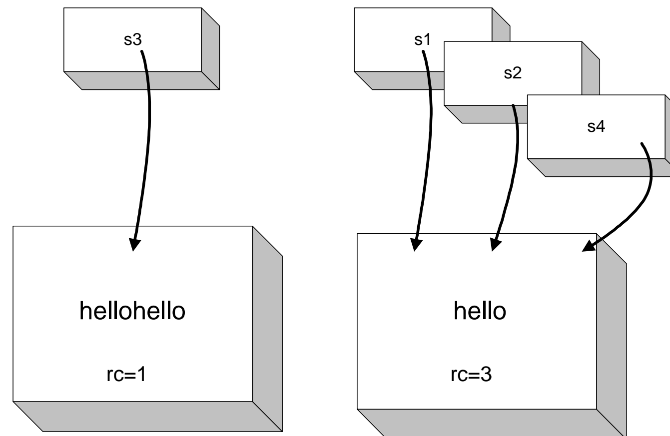
```
class String_imp
{
public:
	String_imp(constchar*str);
	String_imp(constString_imp&);
	constchar*get()const{returnstr;}
	char&operator[](intpos);
	String_imp*operator +(constString_imp&)const;
	String_imp&operator=(constString_imp&);
	voidincRc(){rc   ++;}
	voiddecRc(){if(    --rc<  1)deletethis;}
private:
	~String_imp(){deletestr;}
	char*str;
	intrc;
};

class String
{
public:
	String(constchar*str=    0)
	{
		if(str){
			imp=newString_imp(str);
			imp->incRc();
```

```
            }else
                    imp=  0;
        }
        String(constString&str):imp(str.imp){if(imp)       imp->incRc();}
        constchar*get()const{returnimp?imp        ->get():"";}
        Stringoperator +(constString&str)const;
        char&operator[](intpos);
        String&operator=(constString&);
        ~String(){if(imp)imp   ->decRc();}
private:
        String_imp*imp;
        String(String_imp*other_imp)          {imp=other_imp;}
};
```

הקוד המשתמש אינו מושפע מהשינוי - הוא אינו מודע לפיצול של מחלקת String למחלקה מייצגת ולמחלקה ממששת:

```
Strings 1("hello");
Strings 2,s 3,s 4(s1);
s2=s  1;
s3=s  1 +s 2;
```

תרשים העצמים הוא כעת :



שיכפול לפי צורך (Ondemand)

נחזור לקוד המשתמש הקודם:

```
Strings 1("hello");
Strings 2,s 3,s 4(s1);
s2=s  1;
s3=s  1 +s 2;
```

ונניח שכעת מבוצעת הפעולה

s1.toUpper();

פעולה זו משנה את הטקסט של המחרוזת s1 מ- "hello" ל- "HELLO". האם הפעולה תקינה?
לפי המודל לעיל, גם ערכה של המחרוזות s2,s 4 שונה!

הפתרון הוא לשכפל את עצם המחלקה המממשת (imp) ברגע שקוד המשתמש מנסה לבצע פעולה המשנה את מצב העצם. נגדיר במחלקה String_imp את הפונקציה getCopy() המחזירה העתק חדש של העצם (או את העצם עצמו, אם rc=1):

```
String_imp*String_imp:: getCopy ()
{
        if(rc==   1)
                returnthis;
        --rc;
        returnnewString_imp(*this);
}
```

במחלקה String נממש כך את הפונקציה toUpper():

```
voidString::toUpper()
{
        if(!imp)
                return;
        imp=imp  ->getCopy();
        imp->upper();
}
```

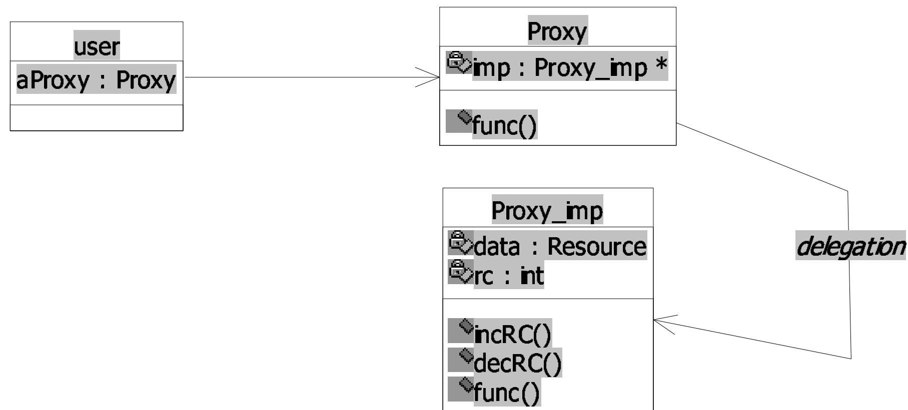
תפקידים נוספים של Proxy

כפי שראינו, ה- Proxy מונע את הצורך בשיכפול עצמים גדולים. Pattern זה יכול לשאת בתפקידים נוספים:

- VirtualProxy - זהו Proxy שאינו טוען את העצם שעליו הוא מצביע מהדיסק, עד לשימוש הראשון בו. זהו מנגנון שימושי עבור עצמים גדולים מאוד כגון תמונות בעלות רזולוציה גבוהה.
- RemoteProxy - מאפשר לגשת לעצם הנמצא על מחשב מרוחק. בתפקידו זה, ה- Proxy מעביר את קריאות קוד הלקוח לעצם דרך הרשת.

סיכום

Proxy הוא Pattern המהווה חוצץ בין קוד המשתמש לעצם "כבד", ומונע העתקות מיותרות שלו. תרשים המחלקות הכללי של ה- Proxy:



ספרות עזר

- **DesignPatterns:ElementsofReusableObject -OrientedSoftware,** byErich Gamma,RichardHelm, RalphJohnson ,and JohnVlissides .(GOF),Addison Wesley.October 1994.
- **ObjectOrientedAnalysisandDesignwithApplications** ,GradyBooch ,BenjaminCummings 1997
- **TheC ++ProgrammingLanguage** , 3rded.,BjarneStroustrup,Addison Wesley, 1997
- **PatternHatching:DesignPatternsApplied** ,JohnVlissides, Addison-Wesley(SoftwarePatternsSeries), 1998
- **ThedesignandevolutionofC** ++,BjarneStroustrup,AddisonWesley, 1994
- **PatternsofSoftware:TalesFromtheSoftwareCommunity** ,RichardP. Gabriel

כל הזכויות שמורות © מאיר סלע

מרכז ההדרכה 2000