Projektowanie obiektowe oprogramowania
Zestaw 6

Wzorce czynnościowe

2020-04-07

Liczba punktów do zdobycia: **6/42**
Zestaw ważny do: 2020-04-28

1. (**1p**) (**Null Object**) Należy zaprojektować własny podsystem logowania, obsługujący zapis do pliku, na konsoli i brak logowania. Klient używa fabryki (singletona) do wydobycia odpowiedniego loggera. Brak logowania obsługiwany jest przez obiekt typu `Null Object`.

```
public interface ILogger
{
   void Log( string Message );
}

public enum LogType { None, Console, File }

public class LoggerFactory
{
   public ILogger GetLogger( LogType LogType, string Parameters = null )
   { ... }

   public static LoggerFactory Instance { ... }
}

// klient:

ILogger logger1 = LoggerFactory.GetLogger( LogType.File, "C:\foo.txt" );
logger1.Log( "foo bar" ); // logowanie do pliku

ILogger2 logger = LoggerFactory.GetLogger( LogType.None );
logger2.Log( "qux" );     // brak logowania
```

2. (**2p**) (**Interpreter**) Dostarczyć implementacji interpretera wyrażeń logicznych.

   Wyrażenia oparte powinny być na prostej gramatyce przewidującej binarne operatory koniunkcji i alternatywy logicznej i unarny operator negacji.

   Tokeny mogą być literałami `true`, `false` lub nazwami zmiennych. Kontekstem interpretera jest funkcja zadająca wartościowanie pewnych zmiennych - dla nazwy zmiennej funkcja zwraca informację o jej wartości logicznej.

   Interpreter powinien poprawnie wyliczać wartości wyrażeń, w których wszystkie symbole terminalne (zmienne) mają swoje wartości w zadanym kontekście oraz wyrzucać wyjątek jeśli podczas interpretacji jakaś zmienna nie ma wartości.

```
public class Context
{
   public bool GetValue( string VariableName ) { ... }
```

```
   public bool SetValue( string VariableName, bool Value ) { ... }
}

public abstract class AbstractExpression
{
   public abstract bool Interpret( Context context );
}

public class ConstExpression  : AbstractExpression { ... }
public class BinaryExpression : AbstractExpression { ... }
public class UnaryExpression  : AbstractExpression { ... }

// klient
Context ctx = new Context();
ctx.SetValue( "x", false );
ctx.SetValue( "y", true );

AbstractExpression exp = ....; // jakieś wyrażenie logiczne ze stałymi i zmiennymi

bool Value = exp.Interpret( context );
```

3. (**2p**) (**Visitor**) Dostarczyć implementacji visitora dla drzewa binarnego, który wyznacza głębokość drzewa. Jako podstawę implementacji wykorzystać kod visitora załączony w notatkach do wykładu.

   Uwaga! Wyznaczanie głębokości wymaga zmiany podejścia w stosunku do tego które jest zaimplementowane w notatkach - przy wyznaczaniu głębokości węzła trzeba bowiem znać głębokości lewego i prawego poddrzewa i przekazać tę informację "wyżej", w miejsce z którego wywoływane jest zapytanie o głębokość. To przekazywanie informacji nie jest w przykładowej implementacji z notatek przewidziane. Twórczy element tego zadania polega więc na wymyśleniu jak to zrobić i w której implementacji Visitora, spośród dwu zademonstrowanych w notatkach, będzie to łatwiejsze.

4. (**1p**) (**Visitor**) Przedstawioną na wykładzie (i załączoną w notatkach) implementację klasy `PrintExpressionVisitor`, dziedziczącej z `ExpressionVisitor` z biblioteki standardowej .NET, rozbudować o obsługę dowolnych trzech dodatkowych rodzajów odwiedzania (czyli formalnie - przeciążyć w sensowny sposób trzy wybrane metody `VisitXXX` z bazowej klasy Visitora, gdzie **XXX** odpowiada rodzajowi odwiedzanego wyrażenia).

   Zaimplementowane metody zilustrować stosownymi przykładami takich wyrażeń, dla których będzie można zaobserwować wyniki działania Visitora (na podobieństwo przykładu z wykładu w którym na konsoli widać efekty odwiedzania wyrażeń binarnych oraz funkcji).

   Uwaga! W dokumentacji można znaleźć przykłady tworzenia wyrażeń nietrywialnych (takich których nie da się napisać za pomocą składni lambda wyrażeń, a do ich wykonstruowania wymagane jest wykorzystanie metod tworzących, np. wyrażeń typu **LoopExpression** *tworzonych przy pomocy funkcji tworzącej* `Expression.Loop`.

Wiktor Zychla