электронный журнал
МОЛОДЕЖНЫЙ НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК
Издатель ФГБОУ ВПО "МГТУ им. Н.Э. Баумана". Эл No. ФС77-51038.

УДК 004.434

Применение предметно-ориентированных языков программирования (Domain Specific languages)

***Минакова С. В.**, студент*
Россия,105005, г. Москва, МГТУ им. Н.Э. Баумана,
кафедра «Системы обработки информации и управления»

Научный руководитель: Гапанюк Ю.Е.,к.т.н, доцент
кафедра «Системы обработки информации и управления»,
Россия,105005, г. Москва, МГТУ им. Н.Э. Баумана
gapyu@bmstu.ru

Ни для кого не секрет, что на данный момент одной из самых динамично развивающихся сфер человеческой деятельности является сфера информационных технологий (IT). Это легко объяснимо, ведь основной предмет сферы имеет свойство очень быстро, причем, чем больше становится информации, тем больше увеличивается темп ее роста. Учитывая динамичность роста информации, логично предположить, что не менее быстро должны развиваться средства ее обработки.

Этому условию в полной мере отвечают языки программирования, которых со времени создания первых программируемых машин было придумано более восьми тысяч, и каждый год их число увеличивается.

Рассмотрим кратко этапы развития языков программирования.

1 этап (1940—1950 гг.)	Использовались машинные коды, непосредственно воспринимаемые машиной, то есть состоящие из единиц и нулей. Составление программ для первых ЭВМ — утомительный и малопроизводительный процесс, где могло быть много ошибок.
2 этап (1950—1960 гг.)	Появились символические ассемблеры — условные мнемонические обозначения-автокоды. Программы на автокодах переводятся на машинный язык с помощью специальной программы, тоже называемой ассемблером. С появлением символических ассемблеров скорость разработки значительно увеличилась, а вероятность сделать ошибку – снизилась.

3 этап (1960—1970 гг.)	Были разработаны первые языки программирования высокого уровня, близкие по своей структуре к естественным языкам. Они настолько упростили процесс разработки, что на момент их возникновения, многие всерьез полагали, что профессии программиста положен конец, потому что пользователи смогут программировать сами (этого, однако, не случилось).
4 этап (1970—1980 гг.)	Началось развитие специализированных языков, таких, как языки систем управления базами данных, предназначенных для решения определенного класса задач.

Итак**,** Domain Specific language (предметно-ориентированный язык) — это язык программирования, специализированный для конкретной области применения, в противоположность языку общего назначения, применимому к широкому спектру областей и не учитывающему особенности конкретных сфер знаний.

Строго говоря, деление языков программирования на языки общего назначения и предметно-ориентированные условно.

Например, язык Си, хоть и считается языком общего назначения, преимущественно используется для написания драйверов, а язык общего назначения PHP — для написания сайтов. Таким образом, и Си и PHP условно являются DSL.

SQL, HTML, CSS — это тоже DSL, предназначенные специально для решения определенного круга задач. Согласитесь, было бы странно верстать веб-страницы на Си, а запросы к базе данных формулировать на CSS. Таким образом, мы постоянно работаем с DSL, просто не всегда осознаем это.

Задачи, решаемые с помощью DSL

Все названное выше представляет собой готовые DSL. Зачем же изобретать новые? На самом деле, этого лучше не делать. Если для решения вашей задачи уже есть готовый, проверенный временем, стабильный DSL, нет повода не использовать его. Но иногда вы можете столкнуться с задачами, для решения которых готового DSL нет.

DSL почти всегда является в том или ином виде попыткой избавиться от шаблонного кода. Например, CSS нужен для того, чтобы не вводить постоянно стили прямо в HTML, а Си — чтобы не писать на программы на ассемблере, ассемблер, в свою очередь — чтобы не писать в байткодах, байткоды — чтобы не вбивать вручную единицы

и нули, то есть первая задача, решаемая с помощью DSL – это сокращение шаблонного (повторяющегося) кода.

DSL также могут помочь, когда код на нем надо писать непрограммистам. Например, если ваш DSL описывает последовательность фильтров, которые нужно применить к цифровому изображению или аудиозаписи, то вряд ли пользователь DSL-я окажется опытным Ruby/C++/Scala/Haskell/Lisp разработчиком. Таким образом, вторая задача, решаемая DSL – это возможность создания приложений в терминах конкретной предметной области.

Для решения этих задач используются разные типы DSL

Типы DSL

Существует два типа DSL: и внешний (external) и внутренний/встроенный (internal)

Внешними называют DSL, которые написаны на языке, отличном от основного языка программного приложения. Примерами DSL такого типа могут служить "малые языки" Unix (lex и yacc) и конфигурационные XML файлы.

Самый большой плюс внешних DSL состоит в том, что его формат будет ограничен лишь вашим умением создать транслятор, который сможет считать конфигурационный файл и выдать некий выполняемый код. То есть вы можете выразить свою предметную область в самой простой и доступной для чтения и редактирования форме. Отсюда же следует и очевидный недостаток внешних DSL - вам придется писать этот транслятор. Если язык несложен, это будет нетрудно. Более сложный язык потребует больших усилий, но с этим тоже можно справиться, так как на данный момент уже существуют и генераторы лексических анализаторов, и другие инструменты для создания компиляторов, которые облегчают работу со сложными языками.

Внутренний DSL - это использование возможностей синтаксиса одного языка программирования для создания иллюзии использования другого, нового языка. Для выполнения внутренний DSL не требует стороннего компилятора, исполняется при исполнении основного программного кода на общем языке программирования. Он используется некоторыми общими программными языками для расширения возможностей программ.

По своей реализации внутренний DSL довольно мало отличается от просто библиотеки (классов, функций или макросов), но является самостоятельным решением, более высокоуровневым и сложным. Поэтому создавать внутренний DSL стоит, только если

стало очевидно, что при использовании библиотеки решение какой-то задачи упрощается ненамного.

Рассмотрим пример. Допустим, вы занимаетесь поддержкой некоторого написанного вами на C++ приложения на. Пусть это будет интернет – магазин, торгующий скейтбордами. В некоторый момент, ваш заказчик решает расширить ассортимент товаров и торговать не только скейтбордами, но еще и самокатами. Вам придется поменять архитектуру вашего приложения таким образом, чтобы оно поддерживало работу с самокатами тоже. С одной стороны принцип выбора и заказа товаров остается неизменным. Между товарами есть определенные сходства, то есть они оба, например, имеют запчасти, но есть и различия. Таким образом, система в целом остается такой же, но часть функций должна поддерживать новые типы объектов, или должны появиться отдельные функции для новых типов объектов.

Пусть есть некая функция, которая написана для скейтборда:

```
public void func(Skate s)
{
  SomeUniversalCode
 someSkateSpecificCode
}
```

Теперь она должна работать и для нового типа - самокат.

Есть несколько способов приспособления уже имеющихся функций под новый объект.

Самый просто способ – скопировать метод, заменив внутри специфичный для старого объекта код.

1.Копирование кода(copy/paste)

```
  public void func(scooter s )
{
  SomeUniversalCode
 someScooterSpecificCode
}
```

Однако, если эта функция вызывается рядом других функций, придется менять и их тоже, фактически дублируя вручную весь зависимый от типа предмета код.

2.Использование If/else

```
public void func (Object o) {
  SomeUniversalCode
   if(o instanceof scooter) {
    someSkateSpecificCode
   } else if(o instanceof scooter) {
    someScooterSpecificCode
   }
}
```

Это немного лучше, чем copy/paste, но заметно увеличивает исходный код функции. Если новых типов объектов несколько – мы получим плохо читаемый шаблонный код, состоящий из множества if/else или switch/case.

3. Использование абстрактного метода

```
abstract void specific()
public void func(Vehicle v) {
  SomeUniversalCode
   v.specific()
}
```

В этом месте вызывается абстрактный метод, который реализован для каждого типа. В принципе это может оказаться приемлемым вариантом, а может и существенно усложнить систему. Когда имеется многоэтажные иерархия наследования с множеством переопределенных методов, зачастую нелегко разобраться какой конкретно метод вызывается.

4.Использование DSL

DSL разрабатывается таким образом, что все особенности типов можно описать. В генераторе кода пишутся шаблоны, которые применяются к описанию типов и получается код как в copy/paste

Шаблон:

```
public void func("TYPE" v) {
 SomeUniversalCode
   "SPECIFIC CODE"
}
```

DSL:

```
type Skateboard:
   property A, ( description, value, links ...)
type Scooter:
   property B,
   property C,
```

Дальше для каждого типа из DSL шаблон трансформируется в конкретный код. В целом, подход следующий — генерируется много простого и хорошо читаемого кода, но файлов получается много, и они могут быть по несколько тысяч строк, то есть объем кода получается довольно большой, но значительная часть этого кода генерируется автоматически, а не пишется вручную, что существенного ускоряет коррекцию архитектуры приложения.

Разработка DSL

Ввиду различий внешних и внутренних DSL, подходы для их разработки также различны

Основные подходы для разработки внутренних DSL.Метод макрорасширений

Во многие языки программирования, например C/C++, LISP, встроены макроязыки, которые позволяют осуществлять замену специально оформленных фрагментов программы на другой текст в соответствии с заданными шаблонами. Для этого в тексте программы должны быть описаны как сами шаблоны, так и случаи их использования, что выполняется с помощью макроконструкций. Макропроцессор, который запускается до компилятора, выполняет заданные правила подстановки шаблона и избавляет от макроконструкций текст программы. Это делает макроязыки удобным и естественным способом задания небольших DSL на основе универсальных языков программирования (содержащих макроязыки).

Например, в C++ макросы создаются с помощью директивы #define, и принимают параметры подобно функциям. После директивы #define указывается имя макроса, за

которым в скобках (без пробелов) параметры, отделенные запятыми и определение макроса, отделенное пробелом:

```
//Объявление
#define Some_lenght = 1436537

//использование
MaxLenght = Some_lenght + 200;
```

Это удобно, особенно если выражение Some_lenght используется в коде несколько раз в разных местах, ведь запомнить условное обозначение проще, чем числовое значение.

Несмотря на множество примеров таких реализаций и простоту метода, он имеет ряд серьезных недостатков:

1. Отсутствие синтаксических проверок для новых конструкций, например проверок соответствия типов параметров.
2. При переводе препроцессором конструкций нового DSL в конструкции базового языка теряется их «привязка» к исходному тексту программы, что затрудняет понимание программистами сообщений об ошибках компиляции и усложняет пошаговую отладку программы.
3. Никто не мешает пользователю ввести свои макрокоманды, возможно переопределив при этом уже имеющиеся, что приводит к серьезным ошибкам в работе программы.
4. Пользователь так реализованного DSL не ограничен в возможностях базового языка и, значит, может создавать программы, выходящие за пределы данной предметной области, т.е. у данного подхода отсутствует семантическая замкнутость.

Основные подходы для разработки внутренних DSL.Метод синтаксических расширений

Метод синтаксических расширений основан на идее введения в существующий язык новых конструкций и семантических правил их трансляции в базовый язык. Проекции новых конструкций в базовый язык задаются с помощью правил синтаксических расширений. Идея синтаксических расширений – это программа, оперирующая другой программой. В отличие от макрорасширений, здесь при определении новых конструкций нам доступна информация о грамматике базового языка. Мы, например, можем указать, что параметром этой нашей новой конструкции может быть любое выражение базового языка или только переменные целого типа. В

макрокомандах нет возможности указывать такую информацию, они ничего «не знают» о том языке, в который они будут раскрыты.

Пример синтаксического расширения на языке C#

```
namespace ExtensionMethods
{
    public static class MyExtensions
    {
        // метод WordCount принимает на вход строку, возвращает число слов, т.е. число подстрок, разделённых пробелом, точкой или вопросительным знаком.
        public static int WordCount(this string str)
        {
            return str.Split(new char[] { ' ', '.', '?' },
                             StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}

/* Метод-расширение WordCount появится в области видимости, если подключить соответствующее пространство имён: */
using ExtensionMethods;

/* Теперь его можно вызвать: */
string s = "Hello Extension Methods";
int i = s.WordCount(); // i получит значение 3
int j = "Съешь же ещё этих мягких французских булок, да выпей чаю.".WordCount(); // j получит значение 10
```

Однако в методе синтаксических расширений до настоящего времени не решена проблема более строгого семантического контроля в новых конструкциях. И еще один недостаток данного подхода— это наследование полученным DSL тех принципов и парадигм, которые присущи базовому языку.

Основные подходы для разработки внешних DSL

Как было сказано, выше, при создании внешнего DSL, возникает создание транслятора DSL, то есть программы, осуществляющей преобразование DSL в язык, на котором он написан.

В общем случае, трансляторы реализуются в виде компиляторов или интерпретаторов.

Компиляторы выполняют полное преобразование всего исходного текста на основной язык, то есть фактически получают новый код, а интерпретаторы подвергают преобразованию только отдельные операторы исходных текстов, не создавая нового кода.

Поскольку язык внешних DSL существенно отличается от языка, но котором они написаны, для внешних DSL обычно пишутся компиляторы.

Две основные парадигмы, использующиеся при разработке компиляторов, — это атрибутные грамматики (attribute grammars) и переписывание (tree/term rewriting).

Атрибутные грамматики

Атрибутные грамматики были введены Дональдом Кнутом в конце 60-х как продолжение идеи вычисления семантики целого выражения через семантику его подвыражений. Атрибутная грамматика — это контекстно-свободная грамматика с атрибутами, которые определяют семантику конструкций языка и записываются в виде БНФ-правил.

Главная идея, лежащая в основе понятия атрибутной грамматики, состоит в том, что значения символов сопоставляются вершинам дерева вывода. Отношения между входными и выходными значениями выражаются по принципу «от правила к правилу»; при этом значения находятся в вершинах дерева.

Рассмотрим пример: Опишем язык L = $\{a_n b_n c_n \mid n > 0\}$, т.е. состоящий из предложений вида abc, aabbcc, aaabbbccc итд. Для его формального описания используем атрибутную грамматику. Синтаксис языка L в виде БНФ будет выглядеть так:

```
S ::= A B C
A ::= a | aA
B ::= b | bB
C ::= c | cC
```

На рисунке 1 представлен синтаксис языка L в соответствии с составленной БНФ

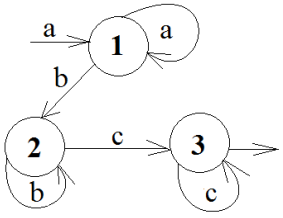

Рис. 1. Граф синтаксиса языка L в соответствии с БНФ

Заметим, что такое описание является неполным - нигде не указано, что количество символов a,b,c в конструкции должно быть одинаковым.

Дополним синтаксис семантической информацией : каждому грамматическому Символу x, кроме стартового, соответствует атрибут Nx. Значение атрибута Nx – это чи Рисунок 1, граф синтаксиса языка в соответствии с БНФ Например, атрибут Na соответствует символу a. Правило вычисления атрибута Na

```
Na(a) = 1
Na(aA) = 1 + Na(A)
```

Атрибуты Nb и Nc соответствуют символам B и C, правила их вычисления аналогичны правилу вычисления атрибута Na. Теперь мы можем ввести семантическое правило, обеспечивающее одинаковое количество символов в предложениях языка.

```
Na(A) = Nb(B) = Nc(C)
```

Таким образом, атрибутная грамматика для языка L имеет следующий вид:

```
S ::= ABC Na(A) = Nb(B) = Nc(C)
A ::= a Na(A’) := 1
| aA Na(A’) := 1 + Na(A)
B ::= b Nb(B’) := 1
| bB Nb(B’) := 1 + Nb(B)
C ::= c Nc(C’) := 1
| cC Nc(C’) := 1 + Nc(C)
```

Граф, полученный в результате введения семантических правил, представлен на рисунке 2

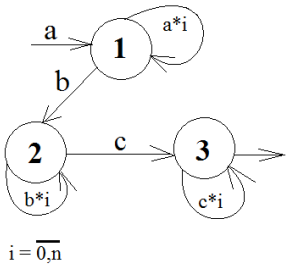

Рис. 2. Граф синтаксиса языка L в соответствии с БНФ введенными семантическими правилами

При анализе программы, заданной атрибутной грамматикой строится дерево разбора. Для нашего примера разбор строки представлен на рисунке 3.

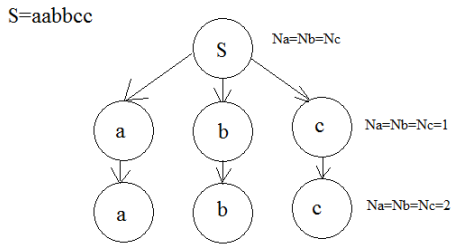

Рис. 3. Дерево разбора строки

Заметим, что правила вычисления атрибутов не могут быть применены, пока не будет готово предложение языка. При построении предложений можно руководствоваться только продукционными BNF-правилами. Поэтому так устроенный алгоритм генерации требует фильтрации построенных предложений для выбора из них только семантически корректных, что эквивалентно написанию семантического анализатора.

Системы переписывания

Существует большое количество методов, использующих процедуры последовательной замены частей формул в соответствии с определенными правилами, которые называются правилами переписывания. Впервые правила переписывания были введены в лямбда-исчислении А.Черчем. Простейшим примером переписывания в графе является замена при оптимизации в дереве программы умножения на два сложением. Вносить небольшие изменения в DSL, описанный с помощью какой-либо системы переписывания, существенно проще, чем в атрибутные грамматики.

Одной из наиболее интересных систем переписывания является TXL - язык трансформационного программирования. Основная парадигма TXL заключается в трансформации входных данных в выходные используя набор правил преобразования, описывающих как различные части входных данных должны изменяться на выходе.

Выводы и перспективы

Использование DSL имеет следующий ряд преимуществ:

- DSL дает возможность создания решений в терминах предметной области, благодаря чему специалисты в данной предметной области могут создавать и модифицировать DSL программы, не вдаваясь в подробности исходного ЯП, на котором написан DSL.
- при проектировании в похожей предметной области можно использовать готовый DSL;
- программы, написанные с использованием DSL лаконичны. Написание DSL с использованием терминов предметной области дает возможность в дальнейшем читать программу достаточно легко;
- происходит повышение надежности, эффективности и качества сопровождения. Поскольку на уровне модели операции осуществлять легче, они более эффективны и подвержены меньшему количеству ошибок, чем те же операции на уровне кода;

К недостаткам использования DSL относятся:

- необходимость обучения пользователей (в случае внешнего DSL) и других программистов, работающих с вашим приложением (в случае внутреннего DSL);
- трудность изначального определения области действия DSL;
- сложность сохранения равновесия между конструкциями, использующимися в DSL и конструкциями языка программирования общего назначения.

Несмотря на недостатки предметно-ориентированных языков программирования, их количество день ото дня увеличивается и программы, написанные на DSL уже во многих областях частично замещают программы, написанные на языках общего назначения. Однако, разнообразие DSL в одной предметной области может приводить к определенным затруднениям. Я полагаю, что, DSL, также, как их предшественники в развитии – языки общего назначения высокого уровня, будут развиваться до тех пор, пока для каждой предметной области не сформируется свой устойчивый язык, учитывающий основные особенности предметной области и достаточно гибкий, чтобы его можно было специализировать для каждого конкретного случая.

Список литературы

1. Фаулер М., Парсонс Р., Специализированные языки программирования. М.: Вильямс, 2011. 413 с.

2. Флауэр Д., Ваш первый специализированный язык программирования. Режим доступа: http://www.codeproject.com/Articles/22650/Irony-NET-Compiler-Construction-Kit (дата обращения 25.02.2015).
3. Креншоу Д. Пишем компилятор. Режим доступа: http://bookz.ru/authors/djek-kren6ou/davaite-_281/page-11-davaite-_281.html (дата обращения 20.03.2015).
4. Архипова В.М. Генерация тестов для модулей проверки статической семантики в компиляторах // Труды института системного программирования РАН. Вып. 8(1), 2004. С. 59-76.