Конопко Кирилл Сергеевич

НЕКОТОРЫЕ ПРОБЛЕМЫ СОЗДАНИЯ И ИСПОЛЬЗОВАНИЯ DSL И КАК ОНИ РЕШАЮТСЯ В СРЕДЕ РАЗРАБОТКИ JETBRAINS MPS

Аннотация

В статье рассматриваются проблемы, связанные с широким применением языков предметной области (DSL). Обсуждается связанное с этим понятие языкоориентированного программирования. Неоднозначность синтаксиса программы, написанной на нескольких языках, видится как критическая проблема. Объясняется один из подходов к преодолению этого ограничения и рассматривается реализация этого подхода на примере среды разработки языков JetBrains MPS.

Ключевые слова: MPS, DSL, LOP, языкоориентированное программирование, языки, среда разработки, синтаксис, AST, абстрактное синтаксическое дерево, клеточный редактор.

DSL И ОГРАНИЧЕНИЯ ИХ ИСПОЛЬЗОВАНИЯ

Существует большое количество языков программирования. Многие из них являются языками общего назначения, то есть языками, на которых можно написать любую программу для любого или, по крайней мере, весьма широкого круга задач. Помимо этих языков, уже достаточно давно известны языки, специализированные под конкретную предметную область, но непригодные для решения задач из других предметных областей. Такие языки называются языками предметной области, Domain-Specific Languages, или сокращённо DSL. Примерами существующих известных DSL могут служить такие языки, как SQL (язык запросов к базам данных), таким языком можно считать Prolog, который, хотя и позиционируется как язык общего назначения, исключительно удобен для решения только определённого круга задач, связанных со значительным использованием средств логического вывода. Также в какой-то мере DSL можно считать небольшой язык регулярных выражений, использующийся для поиска подстрок в строке во многих языках общего назначения. Всяческие языки конфигурирования также являются DSL, во многих визуальных средствах создания экранных форм так или иначе существуют языки описания содержимого экранных форм и расположения в них элементов управления (например, такой язык был в Delphi, в котором окно и его содержимое описывалось в отдельном от кода программы файле. Именно этот файл редактировался визуальным редактором Delphi, но можно было его редактировать и руками).

Многие задачи программирования так или иначе покрывают собой несколько предметных областей, начиная с разделения предметной области отображения графического интерфейса (то есть того, как приложение выглядит для пользователя и с ним взаимодействует) и области бизнес-логики (то есть того, что приложение делает), при программировании которых, очевидно, перед программистом стоят совершенно различные задачи, и продолжая тем, что во многих случаях бизнес-логика сама покрывает несколько областей, некоторые из которых достаточно узки: например, обращение к базе данных, или навигация по дереву или графу определённой структуры, или поиск подстроки по регулярному выражению и т. п.

© К.С. Конопко, 2009

Но, несмотря на достаточно большое количество узких предметных областей внутри задачи, решаемой почти любой программой, DSL распространены отнюдь не столь широко. Чаще всего задачи узкой предметной области решаются с помощью различных библиотек: таковы, например, графические библиотеки типа Swing для Java или GTK, библиотеки для работы с базами данных и т. п. Стиль программирования, при котором под каждую либо большинство узких предметных областей создаётся свой либо берётся уже готовый DSL, называется *языкоориентированным программированием* (термин принадлежит Сергею Дмитриеву [1]). Стиль этот распространён пока крайне нешироко.

Почему же DSL, которые, казалось бы, удобнее библиотек – намерение программиста в них выражается короче, яснее и в терминах предметной области – распространены весьма ограниченно? Иными словами, какие недостатки или ограничения DSL не дают их использовать столь же повсеместно, сколь повсеместно используются библиотеки для узких предметных областей?

НЕСОВМЕСТИМОСТЬ С ОСНОВНЫМ ЯЗЫКОМ ПРОГРАММЫ

Первое ограничение связано с совместимостью языка общего назначения, на котором написана значительная часть программы, и DSL, на которых написаны куски программы для определённых узких предметных областей. Так, код, написанный на SQL, не является органичной частью того языка, из которого он вызывается, а передаётся как строка в качестве аргумента специальной функции, которая осуществляет взаимодействие с базой данных. То же самое и с регулярными выражениями: во многих языках общего назначения, использующих встроенные функции работы с регулярными выражениями, регулярное выражение не интегрировано в язык, а передаётся в функцию как строка, то есть, с точки зрения основного языка, регулярные выражения или код на SQL является не конструкцией языка, а данными, с которыми он работает: создаёт, модифицирует, передаёт как аргументы функциям. Основной же язык, таким образом, является для таких DSL метаязыком (с языками же конфигурации или, например, описания графического интерфейса ситуация немного иная, но схожая: написанные на них конфигурации и описания уже изначально сами по себе являются данными, то есть не задают никакой семантики сами по себе, и лишь программа, использующая эти описания, понимает, как трактовать эти данные и каким образом действовать согласно этим данным).

Таким образом, между основным языком программы и DSL стоит некий барьер: разработчику приходится каждый раз при использовании DSL данные превращать в действия программы, и обратно, то, что является действием, записывать как данные. Фактически, программа выступает здесь как интерпретатор кода на DSL. Поскольку действия могут быть сложными, с большим количеством условий, проверок и ветвлений, для каждого из которых приходится преобразовывать код в данные и обратно, использование DSL становится зачастую неоправданным с точки зрения удобства. Таким образом, для удобства разработчика код, написанный на DSL, должен быть на том же метауровне, что и основной код программы, то есть DSL должен быть выражен конструкциями языка, а не данными. Получается, что удобный DSL должен быть не отдельным языком, никак не связанным с другими, а именно расширением основного языка, то есть дополнительным набором конструкций, ключевых слов и понятий, совместимым с основным языком программы. Удобство DSL при этом сохранится, поскольку он продолжает выражать задачу в терминах предметной области, но, кроме того, такой DSL уже является столь же просто используемым, как и вызов функции из библиотеки.

НЕОДНОЗНАЧНОСТЬ СИНТАКСИСА ПРОГРАММЫ

Второе ограничение обусловлено необходимостью иметь совместимость между DSL-языками, в случае если несколько DSL-

языков расширяют основной язык и используются в одной программе совместно. Особенно это ясно в том случае, когда используются DSL разных независимых поставщиков, не обязанных ничего знать друг о друге.

Самая очевидная проблема здесь – это проблема однозначности текстового синтаксиса. Никто не гарантирует даже того, что сами ключевые слова двух разных DSL (имеющие, скорее всего, различный смысл) будут различными. Даже если ключевые слова будут различными, однозначное понимание текста программы, написанной с использованием нескольких DSL, гарантировать невозможно. Рассмотрим для иллюстрации несколько искусственный пример: пусть есть два DSL-языка, где символ # в первом языке является символом, несущим синтаксическую нагрузку – он обозначает идентификаторы (например, локальные переменные), чтобы отличать их от строковых констант, которые поэтому в первом языке не заключаются в кавычки, – и второй язык, где кавычки используются для обозначения строковых констант, а последовательность символов без кавычек является идентификатором. Тогда в программе, использующей оба этих языка, выражение #s == hello, равно как и выражение s == “hello” являются двусмысленными: первое может трактоваться либо как сравнение содержимого переменной со строковой константой, либо же как сравнение содержимого двух переменных; второе также – либо как сравнение содержимого переменной со строковой константой, либо же как сравнение двух строковых констант. Можно, разумеется, придумать и другие примеры. Итак, как мы видим, неоднозначность текста программы является серьёзным препятствием к свободному использованию произвольного числа DSL-расширений языка в произвольных комбинациях.

АБСТРАКТНЫЙ И КОНКРЕТНЫЙ СИНТАКСИС

Для того чтобы осознать, что же именно мешает объединять синтаксисы различных языков в одной программе, надо сперва понять, что само слово «синтаксис» применительно к языку программирования может обозначать две, хотя и связанные, однако же различные вещи. Разработчики языков (компиляторов, интегрированных систем разработки) чётко разделяют абстрактный и конкретный синтаксис языка (и программы на этом языке). Абстрактный синтаксис программы отражает её структуру: содержание конструкций, каковы их атрибуты, каковы связи между ними. Итак, структура программы представляется в виде абстрактного синтаксического дерева, то есть дерева, узлами которого являются концепции языка. Конкретный же синтаксис отвечает за то, как декодировать программу из текста, и более тесно связан с формальной граматикой языка, чем со структурой программы. В узлах конкретного синтаксического дерева, не являющихся листьями, находятся нетерминалы, а в листьях – терминалы формальной граматики рассматриваемого языка.

Сутью синтаксиса программы является именно абстрактный её синтаксис, в то время как конкретный синтаксис во многом обусловлен удобством редактирования текстового представления программы (то есть исходного кода). Отсюда видно, что одно и то же абстрактное синтаксическое дерево (то есть одна и та же по сути программа) может иметь различные конкретные представления: например, выражение может быть представлено как показано на рис. 1, где все выражения суть записи одного и того же различными способами.

Теперь, оперируя понятиями абстрактного и конкретного синтаксиса, мы можем

```
x = 1 + 2,
(= (x, (+ (1, 2)))),
<assignment><varUsage name =”x” role=”leftHandSide”/><plus role=”rightHandSide”><const value=”1” role=”leftOperand”><const value=”2” role=”rightOperand”></plus></assignment>
```

Рис. 1

сказать, что абстрактный синтаксис неоднозначен быть не может, потому что сама по себе каждая отдельная конструкция языка уникальна – например, ей можно присвоить уникальный идентификатор. Проблема же синтаксической совместимости различных DSL проистекает от того, что различные уникальные абстрактные деревья могут при каких-то их конкретных синтаксисах быть представлены одинаковым текстом, и, таким образом, конкретные синтаксисы становятся несовместимыми. Пока разработчик языка имеет произвол в выборе конкретного синтаксиса DSL, то есть произвол в выборе последовательности символов, уже не содержащей однозначных сведений о том, какую именно конструкцию она обозначает, проблема неоднозначности текста программы, использующей несколько независимых DSL, сохраняется.

БЕЗ КОНКРЕТНОГО СИНТАКСИСА

Решение задачи состоит в том, чтобы отказаться от текстового представления конкретного синтаксиса как такового. Предлагается напрямую редактировать абстрактное синтаксическое дерево (abstract syntax tree, AST), минуя стадию кодирования и декодирования его в текст и обратно. Таким образом, проблема неоднозначности конкретного синтаксиса программы на нескольких языках снимается из-за отсутствия этого самого конкретного синтаксиса.

Разумеется, никто не предлагает редактировать какое-то графическое визуальное представление AST как дерево, в виде набора прямоугольников, овалов или иных графических примитивов, соединённых линиями и стрелками. Это было бы крайне неудобно для программиста и вообще очень медленно для ввода программы. Лучше всего, если бы абстрактное синтаксическое дерево редактировалось как текст – или почти как текст, так как это и привычно программисту, и быстро по скорости ввода программы. Идея же заключается в том, что редактироваться должна не просто строка текста, а именно абстрактное синтаксическое дерево: то есть визуальные элементы, составляющие картинку, выглядящую на экране как текст программы и как текст же редактируемую, должны всегда содержать в себе ссылку на те узлы дерева, которые они обозначают. Таким образом, визуальные элементы, представляющие узлы одной синтаксической конструкции, никак не могут быть перепутаны с другими, пусть так же выглядящими или содержащими те же буквы визуальными элементами, представляющими узлы дерева совершенно другой синтаксической конструкции. Представление дерева на экране является уже не плоским текстом, а, говоря в терминах MVC, View (представлением) и Controller (обычно переводят как «поведение») для AST, которое в таком случае является Model (моделью). При таком подходе конкретный синтаксис будет заменён редактором абстрактного синтаксического дерева. Редактор этот в принципе может быть произвольным, но лучше всего, как было сказано выше, если он будет максимально напоминать текстовый.

ПОДХОД, РЕАЛИЗОВАННЫЙ В JETBRAINS MPS

В JetBrains MPS применён как раз такой подход. JetBrains MPS (Meta-Programming System, система метапрограммирования) – это интегрированная среда разработки языков (например DSL-языков), одновременно являющаяся средой разработки программ на этих языках. Мартин Фаулер в своей статье [2] называет эту и подобные системы Language Workbench, что можно примерно перевести как верстак или рабочее место (для разработки) языков.

Описание языка в MPS состоит из многих так называемых «аспектов», главнейшими из которых являются: структура, описывающая собственно структуру AST, редактор, описывающий то, как будет выводиться и редактироваться AST, генератор, задающий семантику языка посредством указания тех конструкций другого языка, в которые будут превращаться конструкции нашего языка при генерации из программ на нём целевого кода. Также очень важным аспектом языка является система типов, ука-

зывающая то, какой тип должны иметь выражения, переменные и другие конструкции в этом языке. Хотя в принципе без системы типов, в отличие от структуры, редактора и генератора, при создании языка можно обойтись, на практике знание о типах выражений очень часто используется при генерации и довольно часто в редакторе, поэтому система типов тоже крайне важна.

СТРУКТУРА ЯЗЫКА

Описание структуры языка состоит из описания видов узлов AST, то есть различных синтаксических конструкций. В MPS виды узлов называются «*концептами*», каждый узел абстрактного синтаксического дерева является «экземпляром» некого концепта. Концепты могут наследовать друг друга, и, таким образом, они составляют иерархию концептов. Например, концепты «Плюс» и «Минус» могут наследоваться от концепта «Бинарная операция», который, в свою очередь, может являться наследником концепта «Выражение».

Описание концепта содержит в себе описание возможных детей, ссылок и свойств его экземпляров.

Свойства – это возможные строковые, булевские либо числовые атрибуты узла синтаксического дерева. Например, у переменной есть имя, у строковой константы или числовой константы есть соответственно строковое или численное значение. В концептах «Переменная», «Строковая константа» и «Числовая константа» будет указано, что у них могут быть соответствующие свойства. Описание свойства состоит из его имени и типа значения (строковое, численное или булевское).

Также должно быть описано, какие дети могут быть в дереве у узлов-экземпляров данного концепта. Например, у двоичной операции есть правый и левый операнд, у Java-классов и классов других объектно-ориентированных языков есть поля и методы, у скобочного выражения есть выражение внутри скобок и т. п. Описание возможных детей состоит из имени связи, называемой «*ролью*», концепта ребёнка и *кардинальности* связи, то есть количества детей, которые могут быть у экземпляров данного концепта в данной роли. В MPS кардинальностей всего четыре: 0..1, 1, 0..n и 1..n, в зависимости от того, одного или многих детей допускает эта роль и может ли не быть ребёнка в этой роли вообще. У двоичной операции, таким образом, есть связь «левый операнд» и связь «правый операнд», обе имеют кардинальность 1, у обоих концепт ребёнка «Выражение». У класса есть связь «поле» с кардинальностью 0..n и концептом ребёнка «Поле» и связь «метод» с кардинальностью также 0..n и концептом ребёнка «Метод».

Ссылки – это связи узла с другими узлами, не являющимися его детьми. Например, у использования переменной есть ссылка на её декларацию, у вызова метода есть ссылка на декларацию метода и т. п. Ссылки концепта описываются в MPS очень похоже на описание детей, за исключением того, что для ссылок допустимы только кардинальности 0..1 и 1.

На рис. 2 можно увидеть, как в MPS описана структура концептов «Использование переменной» и «Бинарная операция».

Кроме этих трёх частей – свойств, детей и ссылок – в описание концепта также входят и некоторые другие, гораздо менее важные и не являющиеся необходимыми, части. О них можно прочитать в документации MPS [3].

Концепты, которые могут служить корнями синтаксического дерева, называются корневыми. Корневой концепт подобен единице программы, целиком редактирующейся в одном редакторе. В текстовых программах таковы классы Java, модули Pascal, вообще исходный код, содержащийся в одном файле. Несколько корневых концептов образуют модель. Модель – это отдельная достаточно автономная часть программы. Это может быть вообще вся программа, или библиотека, или нечто подобное пакету(package) в Java, например. У каждой модели указываются те языки, которые она использует, и другие модели, которые она импортирует. Узел в модели может быть экземпляром только тех концептов, кото-

рые находятся в языках, которые использует модель. Узлы могут ссылаться только на узлы из их собственной модели и из тех моделей, которые их собственная модель импортирует.

РЕДАКТОР ЯЗЫКА

Редактор AST в MPS решает вышерассмотренную задачу – редактирование напрямую синтаксического дерева, при этом ведет себя похоже на текстовый редактор. Для этого редактор AST был сделан клеточным, то есть всё редактируемое дерево отображается в виде большой клетки, внутри которой расположены более мелкие клетки, представляющие части этого дерева, то есть клетки для детей корневого узла, детей этих детей и т. д. Клетки бывают двух видов: это «атомарные» клетки и клетки-коллекции, которые состоят из других, дочерних клеток и располагают их на экране в соответствии с указанной им политикой расположения, каковых три: горизонтальное расположение, вертикальное расположение и заворачивающее расположение(горизонтальное с переносом на следующий ряд клеток, не помещающихся в заданную предельную ширину редактора). Атомарные клетки бывают константными (клетки для скобок, ключевых слов, специальных символов и т. п.), то есть их содержимое не меняется, и могут отображать редактируемый текст (например некоторое свойство узла синтаксического дерева, которое можно поменять через эту клетку). Каждая клетка содержит ссылку на тот узел синтаксического дерева, который она отображает. Кроме того, все клетки могут содержать реакцию на нажатие определённых клавиш, таких как пробел, возврат каретки, клавиш insert, delete и т. п., и таким образом изменять сообразно с этим синтаксическое дерево.

Описание редактора для некоторого концепта есть описание того, из каких клеток будут состоять редакторы его экземпляров. Это описание является в достаточной степени WYSIWYG, так как оно представляет собой визуальную схему расположения

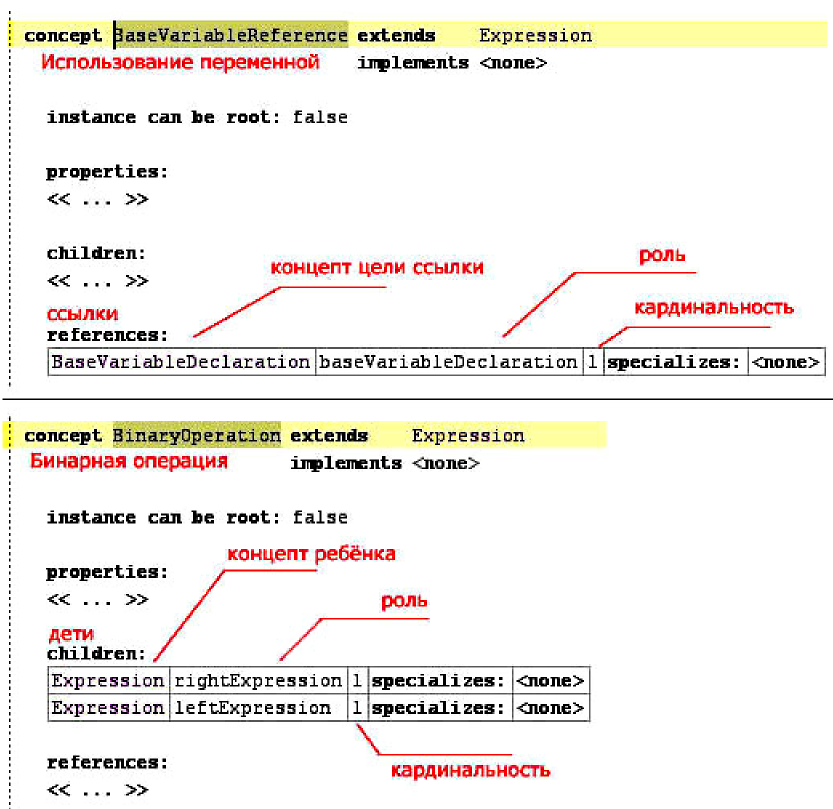

Рис. 2

клеток редактора для экземпляров этого концепта. Мы будем называть элементы этой схемы моделями клеток или мета-клетками, так как они сами по себе визуально являются клетками, описывающими те клетки, которые из них получаются.

Мета-клетки могут представлять клетки для свойств, детей и ссылок. Из мета-клетки свойства получается клетка для редактирования этого свойства; из мета-клетки для ребёнка с кардинальностью 1 или 0..1 получается клетка, содержащая редактор для ребёнка в этой роли; из мета-клетки для ссылки получается клетка, отображающая какие-либо части узла по ссылке (надо указать, какие). Мета-клетка-константа порождает константную клетку с указанным текстом. Мета-клетка-коллекция содержит другие мета-клетки, и в ней указано расположение (горизонтальное, вертикальное или заворачивающее); из неё порождается клетка-коллекция, содержащая клетки, которые получаются из её дочерних мета-клеток и расположены, как указано.

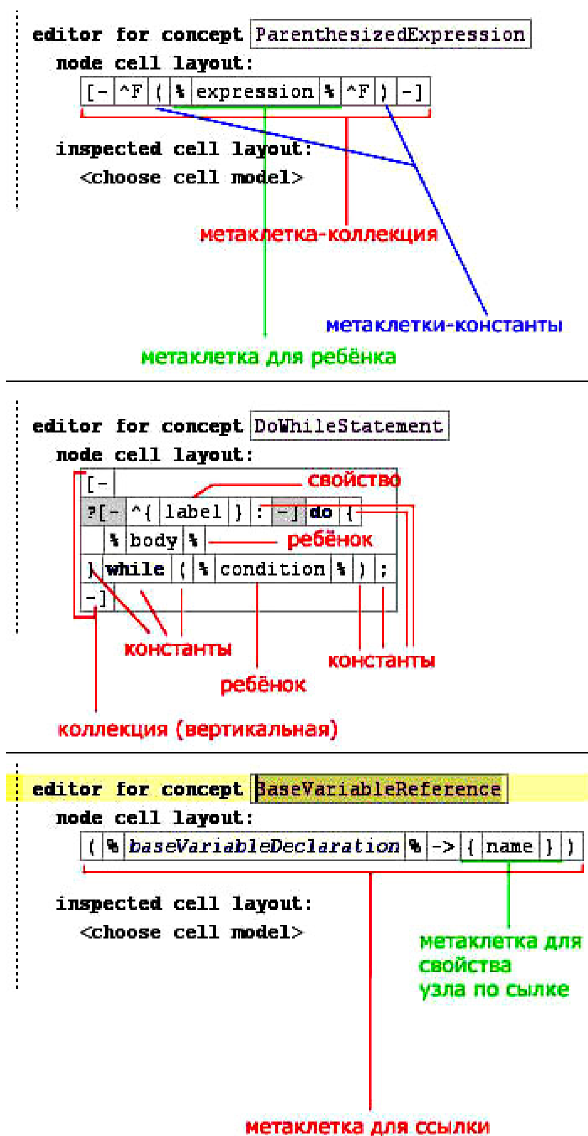

Рис. 3

Мета-клетка для ребёнка с кардинальностью 0..n и 1..n порождает клетку-коллекцию, содержащую редакторы для всех детей узла в указанной роли, расположенные, как указано в мета-клетке. Если каретка будет внутри этой клетки, то при нажатии Enter/Insert создастся ещё один ребёнок в указанной роли после/перед выделенным ребёнком. Если указать в мета-клетке символ-разделитель, редакторы для детей будут отделены клеткой-константой с этим символом, также при нажатии клавиши с этим символом будет создаваться новый ребёнок после выделенного. Например, редактор для параметров метода должен описываться именно такой мета-клеткой с разделителем «,».

Есть и другие, более специальные виды мета-клеток, которые в данной статье рассматриваться не будут. Они тоже описаны в документации MPS [3].

На рис. 3 можно увидеть, как в MPS описаны редакторы для концептов «Скобки», «Цикл делай-пока» и «Использование переменной».

Остался нерассмотренным ещё такой вопрос: допустим, у узла нет ребёнка в данной роли, или нет в некоторой роли ссылки, или есть, но мы хотим заменить ребёнка на другого или переставить ссылку на другой узел. Как в MPS устанавливается нужный ребёнок или цель ссылки посредством редактора синтаксического дерева? Основной механизм для этого – меню подстановки. У клеток для детей и ссылок есть специальная функциональность, которая вызывается нажатием клавиш «Ctrl»+«Space». Это сочетание клавиш активизирует меню, содержащее пун-

кты, с помощью которых можно выбрать нужного ребёнка или цель ссылки. Каждый пункт меню содержит строчку, которая его обозначает. Таким образом, выбрать пункт меню можно как навигацией по нему, так и набирая строчку или префикс строчки этого пункта. По умолчанию, меню для ребёнка содержит список всех доступных (то есть находящихся в доступных языках этой модели) концептов, унаследованных от того концепта, который указан в описании связи для этого ребёнка. Например, меню для правого операнда бинарной операции будет содержать, кроме концепта «Выражение», ещё концепты «Числовая константа», «Строковая константа», «Бинарная операция», «Плюс», «Минус» и многие другие. При выборе пункта меню создаётся новый экземпляр выбранного концепта и устанавливается ребёнком в соответствующей роли. Меню для ссылки содержит список всех доступных (то есть находящихся в этой модели или в импортированных моделях) узлов, являющихся экземплярами концепта этой ссылки и его наследников. При выборе пункта меню цель ссылки заменяется на выбранный узел. Подобная функциональность известна во многих IDE как автодополнение (auto-completion).

В специальном аспекте языка можно изменять меню по-умолчанию для определённых клеток, добавлять и удалять пункты меню для детей и ссылок, такие как концепты и узлы, и добавлять свои оригинальные пункты меню, действующие при их выборе иным помимо рассмотренных, произвольно сложным образом. Для ссылок в специальном аспекте языка можно также указывать область видимости их целей, тем самым отфильтровывая неподходящие узлы из меню. Есть возможность навешивать такие меню на произвольные клетки, предоставляя в них свои собственные оригинальные пункты с произвольным поведением.

Кроме того, такие меню активируются, помимо Ctrl-Space, неявно (меню при этом не показывается, но происходит выбор из него): по потере выделения клеткой (редактор пытается выбрать пункт меню по строке в этой клетке) и при так называемых правых и левых трансформациях, то есть когда программист нажимает пробел или набирает текст перед первым/после последнего символа клетки. Какие именно меню редактор будет использовать при трансформации, также можно задать в соответствующем аспекте языка. Например, если нажать после числовой константы «2» символ «+», то выражение преобразуется в сложение «2 + ..», в котором правый операнд пока пустой. По строчке «+» был неявно выбран пункт меню, который и осуществил подобное преобразование.

ГЕНЕРАТОР ЯЗЫКА

Генератор содержит набор правил, которые преобразуют AST программы на одном языке в AST программы на другом языке, для которого уже задана семантика. В MPS таким языком (почти всегда) является Java, для которой есть компилятор и виртуальная машина. Если один язык расширяет другой язык, можно сгенерировать из программы на более высокоуровневом языке программу на более низкоуровневом языке, а не генерировать из высокоуровневого языка сразу код на Java. Таким образом, в генерации программы может участвовать несколько генераторов, применяющихся на разных этапах. Для более тонкой настройки разработчик языка может указать те генераторы, перед которыми и после которых должен применяться его генератор.

ЗАКЛЮЧЕНИЕ

Были рассмотрены некоторые из ограничений на использование и разработку DSL, объясняющих то, почему языки предметной области, то есть DSL, не используются в программировании столь же широко, сколь широко распространены сами эти узкие предметные области; иными словами, рассмотрены некоторые причины ныне неширокого распространения языкоориентированного программирования. Сформулированы проблемы синтаксического харак-

тера, стоящие перед теми, кто хочет обойти эти ограничения. Был объяснён один из подходов к решению этих проблем и проиллюстрирована практическая реализация такого подхода на примере среды разработки JetBrains MPS.

К сожалению, ограниченный объём статьи не позволил рассмотреть те проблемы и их решения, которые связаны с разработкой генераторов языков, и генераторы лишь упомянуты, дабы не оставлять читателя в полном неведении относительно одного из ключевых аспектов языка. Система типов для программы на нескольких языках и интереснейшие задачи, связанные с нею, не освещены в статье вообще – по той же причине нехватки места.

Автор, один из разработчиков JetBrains MPS, надеется, что по мере развития и внедрения средств разработки языков (language workbenches) языкоориентированный подход к программированию будет использоваться всё более широко, и в перспективе станет в один ряд с такими заслуженными концепциями, как ООП, функциональное программирование, шаблоны проектирования и иными.

Автор благодарен своему коллеге Сергею Дмитриеву, которому, собственно, и пришла в голову идея языкоориентированного программирования, своему коллеге Константину Соломатову, без которого MPS наверняка не стал бы законченным продуктом, а также Ольге Зданович, Елене Циммерман и Владимиру Тимофееву, чья непоколебимая вера в успех вдохновляла меня и на разработку MPS, и на написание этой статьи.

Литература

1. Language Oriented Programming: The Next Programming Paradigm. By Sergey Dmitriev // http://www.jetbrains.com/mps/docs/Language_Oriented_Programming.pdf

2. Language Workbenches: The Killer-App for Domain Specific Languages? By Martin Fowler // http://www.martinfowler.com/articles/languageWorkbench.html

3. MPS User’s Guide // http://www.jetbrains.net/confluence/display/MPS/MPS+User%27s+Guide

Abstract

In this article some difficulties of domain-specific languages’ (DSLs) widespread application are discussed. Related notion of language-oriented programming is explained. Ambiguity of a concrete syntax of a program written in several different languages is considered to be a critical issue. One approach to overcome such an obstacle is explained and its implementation in Jetbrains MPS is overviewed.

Наши авторы, 2009.
Our authors, 2009.

Конопко Кирилл Сергеевич,
старший программный разработчик JetBrains,

Cyril.Konopko@jetbrains.com.