И. Г. Головин, А. В. Столяров

Объектно-ориентированный подход к мультипарадигмальному программированию

Настоящий документ представляет собой черновую версию статьи, опубликованной в журнале «Вестник МГУ», сер. 15 (вычислительная математика и кибернетика), №1 за 2002 год, стр. 46–50. Просьба при цитировании ссылаться на опубликованную версию с использованием следующей библиографической информации:

И. Г. Головин, А. В. Столяров. Объектно-ориентированный подход к мультипарадигмальному программированию. *Вестник Московского Университета, сер.15 вычисл. матем. и киберн., 2002, №1*, , стр. 46–50.

Для пользователей LaTeX приводится библиографическая информация в формате `bibtex`:

```
@ARTICLE{golovinstolyarov:multiparadigm,
        AUTHOR={{И.~Г.~Головин, А.~В.~Столяров}},
        TITLE="Объектно-ориентированный подход к
               мультипарадигмальному программированию",
        JOURNAL="Вестник Московского Университета",
        VOLUME="15",
        NUMBER="1",
        YEAR = "2002",
        PAGES = "46--50",
}
```

Объектно-ориентированный подход к мультипарадигмальному программированию

Игорь Геннадьевич Головин Андрей Викторович Столяров

1 Введение

При реализации большинства современных программистских проектов используется один избранный язык программирования: чаще всего - представитель группы императивных объектно-ориентированных языков, таких как C++[1], Java, Delphi или Ada95[2]. Императивная составляющая такого языка позволяет описывать работу программы в терминах, приближенных к возможностям компьютера (последовательное выполнение инструкций по модификации значений в оперативной памяти). Объектно-ориентированная составляющая [3] позволяет строить модели предметной области в терминах объектов - "черных ящиков", некоторым образом меняющих свое состояние в ответ на получение сообщений.

В то же время существуют языки, предоставляющие совершенно иные изобразительные средства и стимулирующие мышление в иных терминах. Так, функциональные языки (Lisp[4], Hope[5] и др.) предлагают представление программы в виде системы взаимнорекурсивных функций, построенных на обращениях к неким базовым ("системным") функциям. Язык Refal[6] также предоставляет возможность описания программы как системы функций, но основным механизмом работы отдельной функции является сопоставление выражений с образцами и преобразование выражений. Системные функции в Refal'е также присутствуют, но играют скорее вспомогательную роль. Наконец, языки логического программирования[7], и прежде всего Prolog (а также Datalog, Loglisp и др.), предлагают описывать программу в виде набора логических фактов и связок.

Часто в рамках одного проекта возникает потребность использовать различные выразительные средства, наиболее подходящие для решения частных подзадач. Так, если в некотором проекте требуется подсистема лексического анализа, для ее реализации был бы идеальным язык Refal. Символьные преобразования математических формул удобно задавать с помощью языка Lisp. Для решения переборной задачи методом проб и ошибок лучше всего подойдет Prolog или другой логический язык с встроенным механизмом бектрекинга. Использование этих языков для решения соответствующих подзадач могло бы существенно снизить трудозатраты на создание всей программы и повысить ее качество.

К сожалению, технические трудности, возникающие при интеграции языков, имеющих разную природу, на практике сводят на нет преимущества многоязыкового (мультипарадигмального) программирования. Решению данной проблемы и посвящена настоящая работа.

2 Обзор основных существующих подходов

2.1 Применение нескольких систем программирования

Наиболее простым с технической точки зрения подходом к решению проблемы многоязыковых проектов можно считать использование нескольких не связанных между собой систем программирования, каждая из которых поддерживает один из используемых языков. Как правило, на уровне объектного кода такие системы между собой не совместимы, так что сборка модулей воедино не представляется возможной. Кроме несовместимости по сборке, сложности возникают в соглашениях о связях, в способах представления данных и т.д. Поэтому проект реализуется в виде набора (пакета) отдельных программ, каждая из которых может быть написана на своем языке программирования.

При этом проблема интеграции языковых средств уступает место проблеме интеграции разнородных частей пакета программ. Известны различные попытки создания единого подхода к связыванию таких компонент, в частности - технологии CORBA и COM. Однако такие технологии достаточно сложны сами по себе. Создание соответствующих CORBA-объектов и COM-компонент по накладным трудозатратам может быть сравнимо с выигрышем от применения альтернативных языков программирования, что делает многоязыковой подход неоправданным. То же можно сказать и относительно дополнительных затрат технических ресурсов; неизбежно усложняются runtime-библиотеки, возникают потери по быстродействию и объему памяти. Во многих случаях расходы, связанные с CORBA/COM, сравнимы с расходами на решение самой задачи.

Наиболее простым подходом к построению интегрированных пакетов программ является, очевидно, так называемый Unix-стиль, при котором каждая программа имеет поток стандартного ввода, стандартного вывода и имеется возможность перенаправлять эти потоки в файлы или замыкать на другие программы. Единообразие интерфейсов программ позволяет выстраивать системы произвольной сложности из элементов в виде отдельных программ, которые не предназначались для такого использования специально, а лишь следовали соглашению о наличии стандартных потоков ввода-вывода. Чтобы избежать проблем, связанных с различием внутренних представлений данных, используется универсальное представление информации в виде текстов. Это позволяет использовать одни и те же программы как в качестве элементов системы, так и по отдельности. Но и этот подход налагает серьезные ограничения на разработку. Так, программы вынуждены обращаться к другим программам через потоки ввода-вывода, что не всегда удобно, не говоря уже о необходимости постоянно выполнять конверсию из внутреннего представления данных во внешнее и обратно.

Возможен и несколько иной подход к интеграции систем программирования. Допустим, основной проект написан на C++ и необходимо использовать модуль, написанный, например, на языке Lisp. Рассмотрим Lisp-интерпретатор, написанный на C++ (очевидно, что это возможно). Оформим такой интерпретатор в виде отдельного модуля, а необходимый в проекте код на языке Lisp - в виде текстовой константы языка C++. При инициализации программы передадим Lisp-код в интерпретирующий модуль, в результате чего получим готовую к работе виртуальную Lisp-машину с уже заданной программой. В процессе работы можно передавать в текстовом виде запросы к полученной Lisp-машине, получая в ответ результаты, опять таки, в текстовом виде.

Сформулируем основные недостатки рассмотренной методики.

- При таком решении модуль, написанный в альтернативной системе программирования, не может вызывать функции основной программы, и наоборот. Также отсутствует возможность совместного использования глобальных переменных.

- При применении интерпретаторов на этапе выполнения приходится выполнять лексический и синтаксический анализ кода, что снижает эффект от применения в проекте компилируемых языков.

- При организации обмена данными через текстовое представление мы вынуждены анализировать текстовые результаты во всех программах пакета, в том числе и на базовом языке (например, C++). Необходимо отметить, что лексический и синтаксический анализ - это отдельный класс задач, для которых был бы удобнее, например, язык Refal. При таком же решении вынести анализ текста в код на другом языке, очевидно, невозможно.

2.2 Создание нового языка

Другим подходом к решению проблемы мультипарадигмального программирования является создание некоего нового языка программирования, в который вносятся изобразительные средства большего числа различных парадигм. В качестве характерных примеров можно назвать языки Leda[8], Oz[9] и другие. Как показывает практика, такие языки не достигают уровня промышленных инструментов, натыкаясь в своем развитии на так называемый барьер внедрения. Создание нового языка даже при наличии эффективной реализации не влечет само по себе его широкого внедрения в практику. Специалисты начинают изучать новые инструменты только в случае очевидности и несомненной значительности их преимуществ перед существующими и изученными, что само по себе еще необходимо донести до сообщества. Кроме того, необходимо создание адекватной среды разработки, включающей системы программирования для различных платформ, широкие библиотеки и т.д. Наконец, язык, в котором на базовом уровне присутствуют возможности, относящиеся к нескольким парадигмам, неизбежно оказывается чрезмерно сложным.

2.3 Расширение существующего языка

При этом подходе выбирается некоторый язык-лидер и производится его расширение путем введения дополнительных изобразительных средств, соответствующих парадигмам, в исходном языке отсутствовавшим. Удачным примером такой эволюции может служить язык C, давший начало языку C with classes, а он, в свою очередь - языку C++[10].

Возможно расширение и путем создания надстроек над языком. В этом случае исходный код пишется с использованием некоторого набора конструкций, в базовый язык не входящих, а трансляция осуществляется в два этапа - сначала препроцессированием чужеродные конструкции заменяются кодом на исходном языке, затем полученный текст транслируется обычным транслятором. Эта технология широко применяется, например, для встраивания языка запросов на SQL в императивные языки. Также существуют встраиваемые варианты языков Lisp, Scheme[11] и других. Препроцессор фактически заменяет конструкции встроенного языка вызовами интерпретатора этого языка.

О недостатках такой ситуации, связанных с внесением интерпретатора в компилируемую программу, уже говорилось выше.

3 Непосредственная интеграция

Предлагаемый подход к решению проблемы мультипарадигмального программирования также основывается на выборе языка-лидера как основного языка проекта. В качестве языка-лидера предлагается использовать объектно-ориентированный язык программирования. Вместо синтаксического расширения языка средствами препроцессора и тому подобных инструментов мы рассматриваем альтернативные парадигмы как специфические предметные области, подлежащие моделированию объектно-ориентированными изобразительными средствами языка-лидера (в виде соответствующей библиотеки классов).

В пользу выбора объектно-ориентированной парадигмы говорят следующие факторы. Объектно-ориентированная парадигма популярна и имеет эффективные реализации в рамках различных языков и для различных платформ. Современные объектно-ориентированные языки программирования имеют мощные средства развития, богатые библиотеки классов для различных проблемных областей. Кроме того, объектная парадигма хорошо сопрягается с межпроцессными средствами взаимодействия типа CORBA и COM.

Помимо общих достоинств объектно-ориентированного программирования, отдельные языки обладают достаточно мощными синтаксическими возможностями, чтобы позволить написание кода, не выходящего за рамки избранного языка, и в то же время являющегося наглядным отображением некоего кода на альтернативном языке.

Перечислим требования, которым должен удовлетворять конкретный язык программирования, избираемый в качестве лидера. Во-первых, этот язык должен включать понятие класса, поддерживающего основные свойства объектно-ориентированного программирования - инкапсуляцию, наследование и динамический полиморфизм. Все это необходимо для построения адекватной модели альтернативной парадигмы как предметной области. Во-вторых, такой язык должен быть достаточно универсальным и известным, чтобы его использование было допустимо в большинстве проектов.

Наконец, избранный язык-лидер должен иметь возможность переопределения стандартных операций, таких как арифметические знаки, и достаточно широкий набор самих таких операций. Это необходимо для наглядности отображения кода с альтернативного языка на основной.

Этим требованиям удовлетворяют, в частности, такие языки, как C++ и Ada95. В то же время Java, Delphi и некоторые другие языки удовлетворяют первым двум требованиям, но не дают возможности перекрытия стандартных символов операций.

Для экспериментальной реализации метода мы выбрали в качестве базового язык C++. В настоящее время реализованы библиотеки классов, моделирующие изобразительный стиль языков Lisp и Refal (библиотека InteLib[12]).

Рассмотрим примеры конструкций C++, использующих библиотеку InteLib и моделирующих S-выражения языка Lisp.

```
(L| 25, 36, 49)                  // (25 36 49)
(L| "I am the walrus", 1965)      // ("I am the walrus" 1965)
(L| 1, 2, (L| 3, 4), 5, 6)       // (1 2 (3 4) 5 6)
```

```
(L| (L| 1, 2), 3, 4)            // ((1 2) 3 4)
(L| MEMBER, 1, ~(L| 1, 3, 5))   // (member 1 '(1 3 5))
(L| 1 || 2)                     // (1 . 2)
((L| 1, 2, 3)|| 4)              // (1 2 3 . 4)
```

Поясним, что `L` - это экземпляр класса `LListConstructor`, операция `|` которого преобразует свой правый операнд в S-выражение - список из одного элемента; операция "запятая"добавляет новый элемент в конец существующего списка; унарная операция `~` (тильда) конструирует S-список вида `(QUOTE <операнд>)`, то есть заменяет традиционный лисповский апостроф. Традиционное лисповское понятие S-выражения реализовано в виде иерархии полиморфных классов.

Рассмотрим лисповское определение функции:

```
(defun isomorphic (tree1 tree2)
 (cond ((atom tree1) (atom tree2))
       ((atom tree2) NIL)
       (t (and (isomorphic (car tree1)
                           (car tree2))
               (isomorphic (cdr tree1)
                           (cdr tree2))))))
```

Эквивалентный код на языке C++ может выглядеть так:

```
#include "intelib.h"
LSymbol ISOMORPHIC("ISOMORPHIC");
void LispInit_isomorphic() {
  static LSymbol TREE1("TREE1");
  static LSymbol TREE2("TREE2");
  static LFunctionalSymbol<LFunctionDefun> DEFUN("DEFUN");
  static LFunctionalSymbol<LFunctionCond> COND("COND");
  static LFunctionalSymbol<LFunctionAtom> ATOM("ATOM");
  static LFunctionalSymbol<LFunctionAnd> AND("AND");
  static LFunctionalSymbol<LFunctionCar> CAR("CAR");
  static LFunctionalSymbol<LFunctionCdr> CDR("CDR");
  (L|DEFUN, ISOMORPHIC, (L|TREE1, TREE2),
    (L|COND,
      (L|(L|ATOM, TREE1), (L|ATOM, TREE2)),
         (L|(L|ATOM, TREE2), NIL),
         (L|T, (L|AND,
           (L|ISOMORPHIC, (L|CAR, TREE1),
                          (L|CAR, TREE2)),
           (L|ISOMORPHIC, (L|CDR, TREE1),
                          (L|CDR, TREE2)))))).Evaluate();
}
```

Здесь `Evaluate()` - это метод класса, представляющего S-выражение, который производит вычисление выражения в смысле языка Lisp.

Следует особо отметить, что приведенный выше фрагмент кода является кодом на языке C++, не требующим какого-либо дополнительного препроцессирования. Таким

образом, не выходя за пределы языка C++, мы получаем возможность применения парадигм языка Lisp в C++-проекте.

Подробное описание библиотеки InteLib, описывающей все необходимые классы и операции, можно найти в работе [13].

Аналогичная методика применяется для языка Рефал. Рассмотрим функцию языка Рефал, определяющую, является ли заданное выражение палиндромом.

```
Pal {             = True;
      s.1         = True;
      s.1 e.2 s.1 = <Pal e.2>;
      e.1         = False;  }
```

Соответствующий код на C++ будет выглядеть так:

```
#include "intelib_refal.h"
LSymbol PAL("PAL");
LSymbol True("True");
LSymbol False("False");
RfVariable_E e_1("e.1"), e_2("e.2");
RfVariable_S s_1("s.1");
void InitRefalFunctionPal()  {
   RFUNC(PAL) [(L)                ^ (L| "True")]
              [(L| s_1)           ^ (L| "True")]
              [(L| s_1, e_2, s_1) ^ (L| (R| PAL, e_2))]
              [(L| e_1)           ^ (L| "False")] ;
}
```

Здесь `R` - это экземпляр класса `RfListConstructor`, применяемый для создания "активных списков"языка Рефал.

4 Заключение

Предложенный и экспериментально подтвержденный метод позволяет, не выходя за пределы базового языка проекта (например, C++), использовать парадигмы альтернативных языков. Будучи свободным от основных недостатков традиционных подходов к мультипарадигмальному программированию и не требуя существенных затрат на внедрение, новый метод имеет шанс найти реальное применение в индустрии программного обеспечения.

В перспективе планируется создание библиотек классов, моделирующих другие парадигмы, в частности - парадигму логического программирования. Также возможно создание аналогичных библиотек для других базовых языков, прежде всего - для языка Ada95.

Список литературы

[1] Stroustrup B. The C++ Programming Language, 3rd edition. Addison-Wesley. Reading, Massachusets, 1997.

[2] Mitchell R. Abstract Data Types and Ada. Prentice Hall, 1996.

[3] Booch G. Object-Oriented Analysis and Design. Addison-Wesley. 1994.

[4] Steele G. L. Common Lisp the Language, 2nd edition. Digital Press, 1990.

[5] Field A., Harrison P.. Functional Programming. Addison-Wesley, Reading, Massachusets, 1988.

[6] Turchin V. REFAL-5, Programming Guide and Reference Manual. New England Publishing Co., Holyoke, 1989.

[7] Robinson J. Logic programming - past, present and future // New Generation Computing, 1, 1983, p.107-121.

[8] Budd T. Multy-Paradigm Programming in LEDA. Addison-Wesley, Reading, Massachusets, 1995.

[9] Müller M., Müller T., Van Roy P.. Multiparadigm programming in Oz. / D. Smith, O. Ridoux, and P. Van Roy, editors // Workshop on the Future of Logic Programming. International Logic Programming Symposium, 1995.

[10] Stroustrup B. The Design and Evolution of C++. Addison-Wesley. Reading, Massachusets, 1994.

[11] Kelsey R. at al. Revised^5 report on Algorithmic Language Scheme // ACM SIGPLAN Notices, 1998. 33(9). P. 26–76.

[12] Stolyarov A., Bolshakova E., Building Functional Techniques into an Object-oriented System. Knowledge-Based Software Engineering // Proceedings of the 4th JCKBSE, Brno, Czech Republic, 2000. IOS Press, P. 101–106.

[13] Столяров А. Интеграция изобразительных средств альтернативных языков программирования в проекты на C++. М., 2001. Деп. в ВИНИТИ. №2319-В2001.