Никлаус Вирт

Modula-2 и объектно-ориентированное программирование

Niklaus Wirth (1990) Modula-2 and Object-Oriented Programming // Microprocessors and Microsystems, Vol.14, No.3, p.149-152.

Р. Богатырев, перевод с англ.

Как это ни печально, но в области компьютерных наук слишком уж господствуют модные поветрия. Появляясь, как правило, в период обострения проблем, они преподносятся как сильнодействующее средство для тяжелобольных и живут за счет надежд тех, кто находится в отчаянном положении. В программном обеспечении и в программировании вообще часто употребляется термин "кризис программного обеспечения", о котором впервые открыто заговорили в 1968 году и который сделал популярным структурное программирование. Был признан тот факт, что сложное программное обеспечение может быть понято только тогда, когда оно упорядочено и структурировано. Разработка огромных систем, где задействованы армии "аналитиков", со всей очевидностью доказывает необходимость координации работ, документирования и соблюдения соглашений, которые должны быть представлены в виде спецификаций интерфейсов. Руководство (management) становится доминирующим фактором, и все упомянутые аспекты так или иначе покрываются новой волной, которая носит название "программная инженерия" (software engineering). Все это настоятельно требует по-настоящему профессионального подхода.

Самый последний из выдвинутых лозунгов — это объектно-ориентированное программирование. Оно выражает принципиально новый взгляд на системы, фокусируясь на децентрализованном управлении, и берет свое начало в системном программировании. Всякое подобное течение имеет свои законные причины и цели и может подвергаться исследованию на свою пригодность с точки зрения конкретных критериев. Такое исследование необходимо еще и для того, чтобы не идти на поводу у негативных аспектов модного направления, не применять его там, где это не нужно, и перестать опасаться того, что могут назвать устаревшим. Необходимо правильно понимать особенности и основы новой дисциплины. Иначе не дисциплина будет служить нам, а мы станем ее рабами.

Что такое "объектно-ориентированный" ?

Глубинный смысл этой концепции с точки зрения автора заключается в децентрализованном управлении. Первым примером, наглядно поясняющим идею, может служить операционная система. Обычно она состоит из центральной процедуры, которая воспринимает ввод с клавиатуры и передает управление процедуре, отвечающей за интепретацию команд. Еще более простым примером может служить операционная система обычного настольного калькулятора, которая выбирает процедуру, соответствующую нажатой функциональной кнопке. Современные рабочие станции с их средствами поддержки многочисленных окон (т.н. визуализаторов — viewers) требуют более изощренного подхода. Как правило, запуск операции инициируется нажатием кнопки мышки. То действие, которое при этом будет предпринято, зависит от положения отображаемого курсора. Оно априори системе не известно и зависит от типа того визуализатора, на котором оказался курсор. Предполагается, что каждый визуализатор следует своему собственному режиму интепретации команд. Другими словами, он рассматривается как объект со своим поведением. Такая схема реализуется путем поиска того дескриптора, который отвечает за визуализатор, определяемый текущим положением курсора, и последующей передачей управления специальной процедуре. Эта процедура приписана данному дескриптору и носит название "обработчик" (handler). Естественно, что различные визуализаторы (и их типы) должны обладать различными обработчиками. Вместо того, чтобы все управление сконцентрировать в одном диспетчере (в котором жестко заданы пути передачи управления), оно распределяется по обработчикам, личность и номер которых не заданы в тексте программы.

Кстати, подобный взгляд на структуру, обладающую своими процедурами интепретации собственных данных, совпадает с понятием абстрактного типа данных. Описание типа не только специфицирует тип и структуру данных, но также его функцию и допустимые операции. Говорят, что переменные являются экземплярами типа. В среде объектно-ориентированных программистов набор объектов с идентичной структурой данных и обработчиком называется "классом", а объект является экземпляром класса подобно тому, как переменная является экземпляром типа.

Часто бывает нужно строить новый класс на базе уже имеющегося данного класса, в том смысле, что экземпляры нового класса будут использовать свойства (т.е. атрибуты и действия) объектов данного класса, но при этом обладают также дополнительными свойствами. Тем самым, они становятся особыми членами исходного класса и формируют подкласс. Типичным примером могут служить подклассы визуализаторов — текста, графики, изображений. Они обладают всеми свойствами визуализаторов и, кроме того, имеют дополнительные операции, специфичные для обработки текста, графики и изображений. Некоторым программистам нравится проводить аналогии между миром компьютерных систем и миром людей. При таком подходе объектно-ориентированная система сравнивается с человеческим обществом. Причина этого антропоморфического взгляда, который автор находит скорее вводящим в заблуждение, нежели полезным, кроется в том, что подкласс наследует свойства суперкласса.

А потому в техническом жаргоне программистов должно найтись место для "субъекта наследования". Да и термин "субъектно-ориентированный" был бы куда более подходящим для популярного антропоморфического взгляда, чем термин "объектно-ориентированный". К тому же, в традиционной трактовке именно субъект проявляет свое поведение, получая адресованные ему сообщения, тогда как объект играет пассивную роль.

Далеко не случайно, что парадигма объектно-ориентированного программирования — здесь мы подчиняемся традиции и сохраняем неправильное название — берет свое начало в области моделирования систем с дискретными событиями. В этой области возникает потребность представлять абстрактные агенты, обладающие своими свойствами и своим поведением. Такие абстракции первоначально выражались на языках Simula-1 [1] и Simula-67 [2]. Основное внимание уделялось моделированию коллективных и параллельных действий классов таких агентов с использованием интерпретатора в рамках одного-единственного процессора. А потому понятие процессов, или, выражаясь более точно, сопрограмм (coroutines), остается неразрывно связанным с языком Simula. Понятие объекта было унаследовано и сыграло ключевую роль в языке Smalltalk [3]. В то же время парадигма имитации и квазипараллельности была отброшена; по крайней мере, были устранены ее основы.

Какие возможности делают язык "объектно-ориентированным"?

Стоит заметить, что приложения, в которых имеет смысл рассматривать вещи с объектно-ориентированной точки зрения, обычно используют значительное количество элементов данных, большинство которых существует непродолжительное время. Поэтому наипервейшее требование к формулированию подобных приложений — это возможность использовать динамические структуры данных, обычно выражаемые в виде записей, связанных через указатели. Здесь необходимыми механизмами являются динамическое выделение памяти под данные и (предпочтительно автоматическая) утилизация этой памяти.

Помимо этого мы можем выделить следующие два важных требования.

1. Должна существовать возможность определять шаблоны (объектов), состоящие из переменных и процедур. Шаблонам присваивают имя класса, а экземпляры класса называют объектами. Процедуры, определенные для конкретного класса, называют методами, а о вызове метода говорят как об отправке сообщения.

2. Должна существовать возможность строить новые классы из уже существующих. Производный класс (derived class) связан с тем классом, на основе которого он построен, за счет того, что он наследует переменные своего предка, наследует или заменяет его процедуры и возможно добавляет новые переменные и процедуры. Производный класс совместим с производящим классом (deriving class) в том смысле, что экземпляр производного класса может подставляться на место любого объекта производящего класса.

Эти правила совместимости подразумевают, что процедура данного объекта может быть вызвана без ссылки на ее точное представление, поскольку данный объект может быть экземпляром многих производных классов. Это проясняет, почему вместо термина "вызов процедуры" используется термин "отправка сообщения": ведь значение сообщения известно, а интерпретирующая процедура — нет. Диспетчеризация сообщения может теперь происходить в том месте, где фактическая процедура неизвестна, в частности, в модуле, который в иерархии лежит ниже того, где определена процедура. Поэтому подобные вызовы называют отложенными (upcall).

Modula-2 в объектно-ориентированном программировании

Теперь мы перейдем к изучению применимости Modula-2 [4] к объектно-ориентированному программированию. Предварительное требование наличия динамических структур данных здесь удовлетворяется, хотя в большинстве реализаций и отсутствует автоматическая утилизация памяти.

Первое требование также выполняется за счет наличия процедурного типа. Объекты представляются в виде записей, а их методы — как поля процедурного типа. Отправка сообщения выглядит в виде косвенного вызова процедуры через такую процедурную переменную.

Второе требование, однако, не выполняется. Дело в том, что невозможно построить такой тип T1 на основе типа T0, что T0 остается совместимым с T1 (помимо тривиальных случаев совпадения и диапазонов значений).

На этом мы можем закончить данный раздел статьи. Однако, имеет смысл выяснить, можно ли использовать с языком Modula-2 объектно-ориентированный стиль программирования, и если можно, то какими средствами надежности при этом нам придется жертвовать. Как отмечается в работе [5], один из возможных подходов состоит в использовании лазейки в языке Modula-2 (а именно, в модуле SYSTEM). Решение, в частности, связано с типом ADDRESS: с учетом потребности в производном типе описание базового типа дается вместе с дополнительным полем типа ADDRESS с именем, скажем, "ext".

Помимо неизбежности лишнего разыменования при доступе к дополнительным полям, существенный недостаток этого подхода состоит в том, что нарушается способность компилятора осуществлять проверку типов. Программа, использующая подобный прием, потенциально столь же ненадежна, сколь и любой ассемблерный код. Надежность типов — это слишком важное свойство языка высокого уровня, чтобы его можно было принести в жертву.

Да и низкоуровневые средства были заложены в язык Modula-2 с тем, чтобы ими можно было воспользоваться лишь в крайнем случае, когда необходим доступ к специфическим машинным ресурсам, и с учетом того, что они будут изолированы в небольших модулях-драйверах. Автор выступает против их использования в качестве основных инструментов на протяжении всей программы.

Традиционная терминология	Объектно-ориентированная терминология
тип	класс
переменная	объект, экземпляр
процедура	метод
вызов	сообщение
расширение	наследование

Табл.1. Соотношение традиционной и объектно-ориентированной терминологий

Исходя из такого взгляда на проектирование языка не надо прибегать ни к правкам, ни к уловкам, а следует искать решение, которое сможет объединить в себе новое требование с уже существующими свойствами и сумеет подчиниться концепции проверки типов при текстуальном просмотре (точнее говоря, проверки на этапе компиляции).

Расширение Modula-2 для объектно-ориентированного программирования

Идея видоизменения языка Modula-2 с тем, чтобы сделать его приемлемым для объектно-ориентированного программирования, заключается в расширении языка за счет новых свойств, которые должны аккуратно вписаться в существующую структуру, позволяя получить четкое и сжатое описание на основе ясно понимаемых математических концепций. Если сконцентрироваться только на фундаментальных требованиях, то единственным новым механизмом станет средство для построения классов различных объектов.

В соответствии с основным принципом проектирования языка, а именно, введением как можно меньшего числа концепций, и с учетом явной схожести, если не идентичности, типов и классов, мы уравниваем объект с экземпляром (типа запись) и класс с типом. Производный класс (подкласс) теперь соответствует расширению типа запись [6]. Если тип запись рассматривается как растягивание координатного пространства (декартовое произведение подпространств), то расширение увеличивает размерность пространства. Соответственно, присваивание экземпляра производного класса (т.е. переменной расширенного типа) экземпляру суперкласса (т.е. переменной базового типа) просто соответствует проекции на подпространство значения переменной, полученной за счет "растяжения" базового типа.

Пример:

```
TYPE Point2 = RECORD x,y: INTEGER END;
     Point3 = RECORD (Point2) z: INTEGER END;

VAR  p2: Point2; p3: Point3;
```

Присваивание `p2 := p3` соответствует `p2.x := p3.x` и `p2.y := p3.y`.

Естественно, что концепция расширения также применима и к типу указатель. Тем самым, появляется возможность конструировать разнородные структуры, соответствующий ссылочный тип которых связан с узлом типа R. Узлы структуры могут иметь различные расширения R: скажем, R1, R2, R3.

Очевидно, что возникает потребность определять фактический (расширенный) тип узла, на который ссылается указатель (связанный с R). Язык Oberon реализует это в виде "пробы типа" (type test), которая рассматривается как булевский множитель с оператором IS.

Следует особенно подчеркнуть, что важнейшее достоинство подобного подхода в языке Oberon [7, 8] состоит в том, что концепции типов и классов объединены и что при этом удалось избежать сосуществования двух различных понятий, которые по сути выражают одну и ту же концепцию.

Вопросы "синтаксиса"

Если говорить о представлении методов через поля записи, имеющие процедурный тип, то следует сделать некоторые дополнительные пояснения. Объявление класса в объектно-ориентированных языках выглядит подобно объявлению записи с дополнительными объявлениями процедур (или их заголовков). Такой подход обладает некоторыми достоинствами и ведет к определенным последствиям. В языках Modula-2 и Oberon соответствующее поле процедурного типа предполагает особую роль переменной и, следовательно, фактическая процедура должна присваиваться этой переменной явно всякий раз, когда порождается новый экземпляр записи. Это можно рассматривать либо как нагрузку (и как источник ошибок), либо как дополнительную степень свободы (и мощи). К тому же большинство типичных приложений привязывает одну и ту же процедуру (обработчик) ко всем экземплярам класса: взгляд с позиции методов ориентирован на классы; взгляд с позиции Oberon'а — на экземпляры.

Пример:

```
CLASS Viewers =
```

```
        RECORD
          x, y, w, h: INTEGER;
          METHOD restore (T: Text)
          BEGIN ...
          END restore
        END;

TYPE  Viewers =
        RECORD
          x, y, w, h: INTEGER;
          restore: PROCEDURE (T: Text)
        END;

v := Viewers.New (X,Y,W,H); NEW (v);
v.x := X; v.y := Y; v.w := W; v.h := H;
v.restore := Restore;
```

В реализации с акцентом на класс (class-centred) каждый экземпляр будет содержать скрытый указатель на одну и ту же таблицу ссылок на процедуры. В реализации Oberon'a каждый экземпляр содержит прямые (или дублированные) ссылки на установленные процедуры. Естественно, что это нежелательно, особенно когда их много.

Другим достоинством взгляда с акцентом на класс и описания тел процедур внутри описания класса является способность непосредственно ссылаться на поля объекта (x,y,w,h) изнутри процедуры. Это ведет к следующей удобной формулировке отправки сообщения ("restore") объекту ("v"):

```
v.restore(T)
```

Эта форма полностью согласуется с символикой для присваивателей полей (v.x). В приведенном вызове "v" играет роль двойного параметра. Он одновременно действует и как квалификатор имени метода (через класс объекта "v"), и как параметр вызова, а именно, переменной "v".

В языке Oberon такая сокращенная форма невозможна, и обе роли параметра четко разграничиваются:

```
v.restore(v,T)
```

Различия между подходами с акцентом на классы (class-centred) и с акцентом на экземпляры (instance-centred) по отношению к описанию методов можно рассмотреть под другим углом. В первом случае методы объявляются как (процедурные) константы, во втором — как (процедурные) переменные. Ограниченные возможности первого подхода уже проявились в отношении подклассов, т.е. расширений типов. Обычно подкласс делает методы отличными от методов своего суперкласса.

И так как методы задаются в виде констант, вызывается новая реализация, которая переопределяет описание суперкласса. В соответствующих реализациях переопределение (перегрузка) осуществляется за счет выделения своей таблицы методов для каждого подкласса. В ориентированном на экземпляры подходе языка Oberon не требуется ни подобный дополнительный механизм, ни дополнительная символика переопределения, так как все это достигается через обычное явное присваивание.

Недавно был продуман и реализован диалект языка Oberon, поддерживающий подход с ориентацией на классы [9]. Он показал, что дополнительная сложность компилятора остается в устойчивых границах. Вопрос скорее о том, компенсируют ли удобства в использовании символики концептуальные усложнения, и этот вопрос остается открытым.

Объектно-ориентированные языки обычно заставляют объекты быть динамически порождаемыми записями со связанными с ними указателями. В языке Oberon подобное ограничение должно было бы быть определено через явное исключение из правил, поскольку статические, также как и динамические переменные могут принадлежать типу запись и потому быть расширяемыми. На деле же наличие концепции расширения типа оказалось также чрезвычайно полезным в случае статических переменных, передаваемых (по ссылке) в процедуры в качестве параметров. Упомянутое выше ограничение будет приводить к использованию динамического выделения

памяти в тех случаях, когда переменная концептуально должна быть описана как статическая и локальная, и сделано это будет из-за ее недолговечности. В свою очередь, такое решение может серьезно сказаться на эффективности реализации. Подход языка Oberon оказался на поверку очень удачным, тогда как интерпретация каждой переменной в качестве объекта стала заметной ошибкой.

Заключение

Помимо удобных синтаксических конструкций объектно-ориентированный язык отводит важное место объявлениям процедур, связанными со структурами данных (записями), и предоставляет возможность описывать структуры (расширения), которые получаются из других структур и которые совместимы по типу со своими прародителями.

В языке Modula-2 можно использовать объектно-ориентированную парадигму, если прибегнуть к низкоуровневым средствам и пожертвовать наиболее важным достоинством языка высокого уровня — гарантией целостности типов. И хотя язык Modula-2 допускает использование этой парадигмы, он ее не поддерживает.

Язык Oberon развивает Modula-2 необходимым механизмом расширения типов. Однако, между объектно-ориентированными средствами Oberon'a и теми, что имеются обычно в объектно-ориентированных языках программирования, существуют некоторые различия. Наиболее принципиальное состоит в том, что в других языках процедуры (названные методами) выступают в роли констант в описании типа запись (названного классом), тогда как в Oberon'e они выступают как переменные (поля записи). Следовательно, в первом случае гарантируется, что методы будут одними и теми же для всех экземпляров класса, в то время как в Oberon'e они могут варьироваться от экземпляра к экземпляру (и их нужно явным образом устанавливать всякий раз, когда порождается новый экземпляр). Как результат, в Oberon'e нет надобности иметь дополнительное языковое средство под обеспечение разных процедур для расширенных типов (подклассов), тогда как в обычных объектно-ориентированных языках переопределение методов (установленных как константы) требует дополнительной концепции перегрузки (overriding).

Как результат, Oberon концептуально проще, и его реализации не нагружаются дополнительными механизмами работы с классами. С другой стороны, объектно-ориентированные языки могут предоставлять несколько более удобную символику и обеспечивать безопасность за счет гарантии неизменности объявленных методов для всех экземпляров класса, что выражается в улучшенной эффективности отложенных вызовов. Правда, это стоит рассматривать как незначительное преимущество, поскольку есть уверенность в том, что использование объектно-ориентированной парадигмы должно быть осознанным и там, где это в самом деле нужно. При проектировании операционной системы [10] мы обнаружили, что практически вся система была успешно запрограммирована в традиционном стиле, а объектно-ориентированный стиль затронул лишь систему визуализаторов, которая предоставляла распределенное управление. И как совет — не стоит злоупотреблять отложенными вызовами.

Весьма значительный вклад в эффективность вносит обобщение концепции расширения типа (подкласса) для статических переменных, в частности, при их использовании в качестве параметров процедур.

Наиболее важная особенность Oberon'a состоит в том, что он поддерживает объектно-ориентированное программирование в структуре самого языка, которая также полностью поддерживает традиционный стиль. Это гарантирует полную проверку целостности типов. Таким образом, Oberon отличается от других языков, поскольку он вырос из того убеждения, что проектирование языка должно стремиться к упрощению через интеграцию крайне схожих концепций, а не к усложнению за счет добавления новых средств, очень похожих на те, что уже имеются.

Литература

[1] Dahl O.-J., Nygaard K. (1966) Simula: an Algol-based simulation language // Communications of the ACM, Vol.9, No.9, p.671-678.

[2] Birtwistle G. et al. (1973) Simula Begin // Auerbach.

[3] Goldberg A., D.Robson D. (1983) Smalltalk-80: The Language and its Implementation // Addison-Wesley.

[4] Wirth N. (1982) Programming in Modula-2 // Springer-Verlag.

[5] Blaschek G. (1989) Implementation of objects in Modula-2 // Structured Programming, Vol.1, No.3, p.147-156.

[6] Wirth N. (1988) Type Extensions // ACM Transactions on Programming Languages and Systems, Vol.10, No.2, p.204-214.

[7] Wirth N. (1988) From Modula to Oberon // Software — Practice and Experience, Vol.18, No.7, p.661-670.

[8] Wirth N. (1988) The programming language Oberon // Software — Practice and Experience, Vol.18, No.7, p.671-690.

[9] Moessenboeck H., Templ J., Griesemer R. (1989) Object Oberon. An Object-Oriented Extension of Oberon // ETH, Zurich, Dept.Informatik, Technical Report No.109.

[10] Gutknecht J., Wirth N. (1989) The Oberon System // Software — Practice and Experience, Vol.19, No.9, p.857-893.