Процедурно-параметрическая парадигма программирования. Возможна ли альтернатива объектно-ориентированному стилю?

Легалов А.И.

Красноярск: 2000. Деп. рук. № 622-В00 Деп. в ВИНИТИ 13.03.2000. - 43 с.

Введение

Предлагается парадигма программирования, определяющая новый стиль разработки программ, названный процедурно-параметрическим программированием (ППП). В основе парадигмы лежит параметрический полиморфизм, позволяющий процедурам принимать и обрабатывать вариантные типы данных, без алгоритмического выбора альтернатив внутри этих процедур. В процедурных языках программирования такие типы описываются объединениями (union в языках C, C++) или вариантными записями (в языке Паскаль). Алгоритмическая обработка вариантов осуществляется с применением условных операторов или переключателей. Данный подход является развитием методов процедурного программирования и служит альтернативой объектно-ориентированному программированию.

Следует уточнить понятие "***процедурное программирование***", используемое в работе. Оно не отождествляется со структурным программированием. Во-первых, структурное программирование в основном ассоциируется с алгоритмической декомпозицией. Вместе с тем, использование традиционных процедурных языков может сочетаться и с объектно-ориентированной методологией (ООМ) [7]. В этом случае, для доступа к структурам данных, можно использовать специально выделенные процедуры (специализированные процедуры). Остальные же процедуры, для доступа к этим же структурам данных, могут пользоваться интерфейсом, предоставляемым специализированными процедурами. Во-вторых, процедурная организация программ применяется не только в структурном программировании. Такое взаимодействие осуществляется в функциональном программировании (на уровне функций, обменивающихся данными) [6], параллельном программировании (на уровне взаимодействия процессов) [5] и т.д. Поэтому, при дальнейшем изложении, понятия "***процедурный подход***", "***процедурное программирование***" используются для обозначения всех парадигм программирования, связанных с организацией процесса вычислений на основе потоков управления, потоков данных и с непосредственным вызовом процедур, функций, процессов. Далее, говоря о процедурах, будем также подразумевать функции и процессы.

Процедурно-параметрическая парадигма программирования является расширением процедурного подхода. Она позволяет увеличить возможно-

сти последнего за счет поддержки полиморфизма данных. Применение предлагаемого подхода позволит наращивать функциональные возможности процедур без внесения в них внутренних алгоритмических изменений. ППП может использоваться как независимо, так и в сочетании с другими парадигмами программирования.

Различия процедурной и объектной парадигм

Различная популярность существующих парадигм программирования во многом объясняется их способностями поддерживать современные методологии разработки программного обеспечения. Основным критерием в оценке программных продуктов является сложность [1], а основными требованиями к методологиям разработки программного обеспечения являются: удобство сопровождения, возможность безболезненного наращивания уже существующей программы, способность разработанных программных объектов к повторному использованию. При этом, на второй план отступает такое требование, как быстрое проектирование первоначальной версии программы, потому что его воплощение обычно не позволяет соблюсти все остальные условия.

Предъявляемым требованиям соответствует объектно-ориентированная методология (ООМ), которая и получила наибольшее распространение. В настоящее время она успешно развивается по самым различным направлениям, затрагивая как анализ и проектирование программных систем, так и написание самих программ. Последнее определяется как объектно-ориентированное программирование (ООП) и связано с использованием соответствующих объектно-ориентированных (ОО) языков. Развитие ООП практически полностью вытеснило процедурное программирование из области, связанной с разработкой сложных программных систем.

Такое достижение в первую очередь связано с методами группировки элементарных программных объектов в более крупные наборы. Конструкции, используемые в ООП, отличаются от тех, что применяются в процедурном программировании. Это позволяет по-новому подходить к процессу разработки. В результате стало удобнее осуществлять сопровождение и модификацию программ.

Основным конструктивным элементом ООП является класс. Именно он определяет свойства, связанные с возможностью однократного определения типа обрабатываемого объекта и последующим многократным использованием его внутренних процедур для изменения своих данных без алгоритмической проверки типа. Эта группировка осуществляется на основе построения однозначных отношений между специализированными данными и обрабатывающими их процедурами. При процедурном подходе такая группировка обычно осуществляется внутри процедур, что ведет к использованию алгоритмических методов для ее формирования и обработ-

ки. ООП предлагает группировку процедур вокруг обрабатываемых ими альтернативных наборов данных, что позволяет обойтись только декларативными методами.

Рассмотрим специфику различных способов группировки на простом примере. Предположим, что нам необходимо осуществить вычисление выражения **F(X)**, где **F** - это обобщающая процедура, являющаяся одной из разновидностей:

$$\mathbf{F(X) = \{ f_1(X), f_2(X), \dots , f_m(X) \}},$$

$\mathbf{X = \{ x_1, x_2, \dots , x_n \}}$ - обобщающий аргумент одного из типов:

$$\mathbf{type(x_j) = t_j}, \text{ где } \mathbf{t_j \in T} \text{ и } \mathbf{T = \{t_1, t_2, \dots , t_n\}}.$$

При этом для выполнения функции $\mathbf{f_i}$, обрабатывающей аргумент типа $\mathbf{t_j}$, используется специализированная процедура $\boldsymbol{\varphi_{ij}(x_j)}$. Процессор, обрабатывающий такую полиморфную процедуру, может использовать два различных варианта последовательного доступа к альтернативным параметрам. В первом случае обработка может начаться с анализа значения функции, для которой будет осуществляться выбор типа операнда. На некотором С-подобном языке такой вариант анализа можно представить в виде следующей схемы:

```
switch(F) {
    case f1:
        switch(type(X)) {
            case t1: φ11(x1); break;
            case t2: φ12(x2); break;
                …
            case tn: φ1n(xn); break;
        }
    case f2:
        switch(type(X)) {
            case t1: φ21(x1); break;
            case t2: φ22(x2); break;
                …
            case tn: φ2n(xn); break;
        }
    …
    case fm:
        switch(type(X)) {
            case t1: φm1(x1); break;
            case t2: φm2(x2); break;
                …
```

```
                case tn: φmn(xn); break;
            }
}
```

Тот же самый результат на выходе можно получить, если в начале проанализировать тип аргумента, а затем лишь значение процедуры:

```
switch(type(X)) {
    case t1:
        switch(F) {
            case f1: φ11(x1); break;
            case f2: φ21(x1); break;
                …
            case fm: φm1(x1); break;
        }
    case t2:
        switch(F) {
            case f1: φ12(x2); break;
            case f2: φ22(x2); break;
                …
            case fm: φm2(x2); break;
        }
    …
    case tn:
        switch(F) {
            case f1: φ1n(xn); break;
            case f2: φ2n(xn); break;
                …
            case fm: φmn(xn); break;
        }
}
```

Первый способ анализа задает зависимость данных от значений процедур. Именно таким образом происходит группировка внутри процедурных программ. Группировка процедур в зависимости от типов данных является спецификой объектного программирования. Эта специфика практически не проявляется, если мы имеем дело с некоторым процессором, занимающимся анализом произвольных комбинаций процедур и данных, поступающих на его вход.

Однако при разработке программ возникает несколько иная ситуация. Значения альтернативных типов данных обычно определяются в ходе выполнения программы. Они формируются случайным образом из различ-

ных источников, что и определяет их непредсказуемость. Расположение же обобщающих процедур задается программистом на этапе разработки программы. Это ведет к естественному отбрасыванию альтернативы, обеспечивающей выбор процедуры. Такое представление программы ведет к тому, что процедуры, располагаемые в процедурной программе на строго определенных местах, не могут знать конкретных типов их аргументов, поступающих во время выполнения программы. В этой ситуации вполне естественно, что обобщающие процедуры содержат в своих телах код, осуществляющий анализ альтернативных аргументов.

В ОО программах процедуры классов тоже размещаются в местах, определенных алгоритмом обработки данных. Но их группировка вокруг обрабатываемых типов данных осуществляется на этапе разработки программы и в декларативной манере, что не требует создания в программе экземпляров конкретных объектов, позволяя, однако, хранить заготовки отношений между типами данных и процедурами в виде векторов отношений, определяющих отдельные классы $\mathbf{r_i} \in \mathbf{R}$:

$$r_1 = (t_1, \varphi_{11}, \varphi_{11}, \varphi_{m1}),$$
$$r_2 = (t_2, \varphi_{12}, \varphi_{22}, \varphi_{m2}),$$
$$\dots$$
$$r_n = (t_n, \varphi_{1n}, \varphi_{2n}, \varphi_{mn}).$$

Поэтому, когда в ходе выполнения программы формируется экземпляр объекта $\mathbf{x_i}$ типа $\mathbf{t_i}$, создается отношение $\mathbf{r_i}$, которое содержит и экземпляры процедур, обеспечивающих разнообразную обработку созданного объекта. Однозначная зависимость между данными и процедурами их обработки, сгруппированными в одном объекте, позволяет каждому экземпляру класса в любой момент обратиться к его же функции через существующее отношение без дополнительного выбора специализированной процедуры. Естественно, что в реальных системах программирования создание своего экземпляров процедур для каждого экземпляра данных не происходит. Там используется обращение через указатели к общему набору процедур.

Программные объекты процедурной и объектной парадигм, используемые для построения эквивалентных понятий

Основными понятиями, используемыми в любой парадигме программирования, являются: состояния программы, действия выполняемые программой. Их реализация осуществляется через такие элементарные программные объекты как данные и операции. Абстрагирование от конкретных экземпляров достигается за счет определения понятий "абстрактный тип данных" и "процедура" (понятие "функция" будем использовать как синоним). Вполне естественно, что эти элементарные понятия исполь-

зуются для построения составных программных объектов путем объединения в агрегаты и разделения по категориям. Категорию Г. Буч [1] называет иерархией типа "is-a". Она также связывается с таким понятием, как обобщение программных объектов. ***Агрегаты*** и ***обобщения*** используются при конструировании композиций данных и процедур. В каждой из существующих парадигм программирования вопросы конструирования решаются по-своему.

Конструирование агрегатов

Построение агрегатов при процедурном подходе опирается на использование следующих понятий:

- Агрегатам данных соответствуют абстрактные типы, представляемые структурами в языках C, C++ и обычными записями в языках Паскаль и Модула. В дальнейшем будем использовать для их обозначения понятие "***структура***".
- Агрегатам процедур соответствуют вложения в процедуры различных по иерархии объектов действия: операций, операторов, вызовов процедур. Обозначим данную абстракцию понятием "***независимая процедура***".

Процедурный подход предполагает независимость структур от процедур. Процедуры используют структуры в качестве элементов списка параметров, обеспечивая тем самым связь с различными экземплярами данных и их обработку. Следует отметить, что здесь и далее механизм взаимодействия кода и данных рассматривается упрощенно. Не учитываются внешние переменные, доступ к которым можно считать как частный случай доступа к аргументу из списка параметров.

Объектно-ориентированное программирование предлагает следующие варианты композиции для создания агрегатов:

- Основной агрегирующей единицей является оболочка, называемая ***классом***, в которой могут быть размещены данные, процедуры и другие классы. Класс называется также типом (в языке Оберон-2 [8]). В языке C++он может быть структурой. Однако, термин "***класс***" является наиболее распространенным. Поэтому, в дальнейшем будет использоваться именно он.
- Процедуры, размещаемые в классе и используемые для изменения своего внутреннего состояния, часто называются методами, функциями-членами класса. В Обероне-2 они называются связанными процедурами. Будем использовать термин "***процедура класса***".
- Данные, определяющие внутреннее состояние класса, обычно называются ***переменными класса***. В дальнейшем будем использовать именно это название.

- Термин "***оболочка класса***" (или просто "***оболочка***", если понятен контекст) будем использовать для обозначения класса в том случае, когда хотим убрать из рассмотрения его переменные и процедуры.

Следует также отметить, что разная терминология в ОО подходе усугубляется также различными способами организации классов. В языке программирования C++ процедура класса может располагаться как внутри оболочки, содержащей также переменные класса (внутреннее описание процедуры класса), так и вне ее. В последнем случае она представляет внешнее описание процедуры, которому должно предшествовать ее объявление внутри оболочки. В языке программирования Java процедура класса всегда располагается внутри оболочки. В ней же размещаются и переменные класса. В языке программирования Оберон-2 процедура класса размещается всегда вне его оболочки, однако, вместе с ней составляет единое целое, подчиняющееся законам полиморфизма и наследования, что, правда, описано, опять-таки, с использованием другой терминологии. Внутри оболочки располагаются только переменные класса, а сама оболочка называется типом данных.

Такая относительность в целом ведет к одинаковому набору связей между объектами класса, но разным образом сказывается на их размещении в тексте программы. Это приводит к тому, что изменение связанной процедуры в Обероне-2 не изменяет тип данных, к которому она привязана. В других же языках программирования происходят изменения внутри оболочки. В целом же можно считать, что изменения класса как совокупности оболочки, переменных и процедур происходят в любом случае.

Конструирование обобщений

Обобщение - это способ формирования альтернативных по свойствам программных объектов, принадлежащих к единой категории в некоторой системе классификации. Оно включает ***обобщение данных*** и ***обобщение процедур (процедурное обобщение)***. Обобщение данных состоит из ***основы обобщения***, к которой присоединяются различные специализации. ***Специализации обобщения*** - это различные понятия, принадлежащие к единой категории. В общем случае, специализации могут тоже являться обобщениями, которые определим как ***многоуровневые обобщения***. Многоуровневые обобщения могут быть иерархическими и рекурсивными. Обобщение является ***одноуровневым***, если оно рассматривается как конструкция, состоящая из основы обобщения и его специализаций.

Обработка обобщений данных осуществляется соответствующими процедурами, включаемыми в состав процедурного обобщения. Процедура, связанная с обработкой основы обобщения называется ***обобщающей процедурой***. Процедура, осуществляющая обработку отдельной специализации обобщения, называется ***специализированной***.

Вариантное обобщение

Процедурный подход строит обобщения данных с применением в качестве основы конструкций, обеспечивающих альтернативное толкование понятий (объединений в C, C++ или вариантных записей в Паскале и Модуле-2). Для обозначения процедурной основы обобщения будем использовать термин "***вариант***". Обобщение, применяемое в процедурном подходе и построенное на основе варианта, назовем ***вариантным обобщением***. С каждым вариантом связан набор специализаций обобщения, построенный на основе агрегатов и других обобщений (структур и вариантов). Эти специализации, используемые в процедурном подходе, определим как ***вариантные специализации***.

Особенностью вариантного обобщения является то, что его основа, как программный объект формируется после того, как описаны все ее специализации. Это связано с тем, что при формировании варианта необходимо знать типы специализаций. Любая вновь создаваемая вариантная основа может объединить произвольное число существующих специализаций. Таким образом, процесс построения вариантного обобщения совпадает по направленности с восходящим проектированием. В начале кодируются отдельные специализированные фрагменты, а уже затем формируется их совместное, более крупное, образование. Это никоим образом не говорит, что при процедурном подходе проблематично применять нисходящую разработку. Просто, нисходящее проектирование ведется, в данной ситуации, таким образом, что в начале осуществляется полная проработка всей иерархии обобщения, а уже только потом можно приступать к его кодированию. Возможно, что эта особенность кодирования вариантных обобщений в какой-то мере послужила популярности водопадной модели при структурном проектировании, оказавшейся малоэффективной при разработке сложных программ.

Обработка обобщений данных осуществляется независимыми процедурами, организующими доступ к внутренним переменным через экземпляр обобщения, получаемый, в качестве аргумента, списка параметров. Процедуры, обрабатывающие специализации обобщений, могут создаваться независимо от обобщающей процедуры. Они могут использоваться различными обобщающими процедурами. Каждая из таких процедур связана только со своими специализациями. В дальнейшем специализированные процедуры, используемые в процедурном подходе, будут также называться ***обработчиками вариантов***.

Процедуры, осуществляющие обработку всего вариантного обобщения, используют алгоритмический механизм анализа вариантов, который обычно строится с использованием условных оператором или переключателей. Анализ осуществляется всякий раз, когда запускается процедура, и заключается в проверке специализации обобщения. После определения специализации запускается соответствующий обработчик варианта. Обоб-

щающая процедура, осуществляющая обработку вариантного обобщения, называется ***вариантной процедурой***.

Использование независимых вариантных процедур для создания кода ведет к централизации процесса обработки обобщений, разделяя его на отдельные задачи. Каждая из процедур обеспечивает решение одной из специализированных задач. Процедуры, решающие разные задачи, совершенно не связаны друг с другом. Декомпозиция работ внутри каждой вариантной процедуры осуществляется в соответствии со специализацией обобщений, когда каждая специализация выполняется отдельным обработчиком варианта, вызываемым из вариантной процедуры.

Опорное обобщение

Опорное обобщение строится при использовании ОО подхода. Оно состоит из ***опорного класса***, выступающего в роли основы обобщения, и производных классов, выполняющих роли отдельных специализаций обобщения. Опорный класс реализуется как базовый класс или суперкласс.

В рассматриваемом контексте базовый класс отличается от обычного класса, введенного при определении агрегатов, наличием виртуальных процедур, допускающих переопределение в производных классах. Будем считать, что обычный класс не содержит виртуальных процедур. В отличие от него, базовый класс позволяет порождать производные классы с применением иерархии наследования и содержит только виртуальные процедуры. Такое допущение позволяет независимо рассматривать особенности процесса модификации ОО программ для различных конструкций и не вносит искажений в предмет исследований.

Процесс кодирования опорного обобщения начинается с формирования структуры опорного класса, который определяет единую основу для всех составляющих обобщения. В нее входят общие переменные, задающие унифицированную часть состояния, общий интерфейс, состоящий из виртуальных процедур класса. Каждая процедура класса определяет одну из функций обработки в соответствии с целевым назначением обобщения. Виртуальная процедура базового класса определяется как ***опорная процедура*** и играет роль обобщающей процедуры.

После построения опорного класса начинают формироваться все производные классы, каждый из которых доопределяет состояние в сторону своей специализации. Производные классы выступают в роли специализаций обобщения. При этом специализации могут добавляться к опорному обобщению в любое время, не изменяя его организации. Специализации обобщения, построенные на основе производных классов, назовем ***специализациями опоры***.

Направление процесса построения опорного обобщения совпадает с процессом нисходящего проектирования. Первоначально создается опорный класс, определяющий абстракцию верхнего уровня. После этого осу-

ществляется детализация, связанная с построением производных классов, решающих частные задачи. Многоэтапное уточнение может сопровождаться построением иерархий классов на основе наследования. Такое совпадение позволяет проводить программирование почти одновременно с проектированием, создавая каркас приложения или законченную версию с неполными функциональными возможностями. Наращивание функциональных возможностей осуществляется за счет новых специализаций на следующих витках процесса проектирования. Это объясняет популярность возвратной модели проектирования при использовании ООМ. Данный подход наиболее популярен в настоящее время.

Обработка обобщений обычно осуществляется виртуальными процедурами производных классов, которые переписываются в соответствии с их специализацией. Эти процедуры назовем ***расширяющими процедурами***. При этом процедура базового класса (опорная процедура) может использоваться там, где ее функциональных возможностей достаточно. В опорном обобщении осуществляется привязка процедур классов к объектам, что ведет к централизации интерфейса. Декомпозиция производится путем распределения каждой процедуры по своим специализациям, заданным производными классами.

Использование процедурного и ОО подходов

Отличие между процедурной и объектно-ориентированной парадигмами программирования по способу анализа альтернатив можно рассмотреть на примере простой программы нахождения действительных корней квадратного уравнения. Следует отметить, что в представленных примерах программ отсутствует код, несущественный для рассматриваемого вопроса. В листинге 1 приведена программа, написанная на языке программирования C++ с использованием процедурного подхода. Нумерация строк листингов введена для обеспечения на них ссылок из текста статьи.

Листинг 1. Использование процедурного подхода

```
1. #include <iostream>
2. #include <math.h>

3. using namespace std;

4. enum rootType {ONE, TWO, ZERO};

5.  struct TwoRoots {
6.      double x1;
7.      double x2;
8.  };
```

```
9.  struct Root {
10.     rootType key;
11.     union {
12.         char* mes;
13.         double x;
14.         TwoRoots x1x2;
15.      } value;
16. };

17. void EvalOneRoot(double a, double b, Root& root) {
18.     root.key = ONE;
19.     root.value.x = (-b) / (2 * a);
20. }

21. void EvalTwoRoots(double a, double b, double d, Root& root) {
22.     root.key = TWO;
23.     d = sqrt(d);
24.     root.value.x1x2.x1 = (-b - d) / (2 * a);
25.     root.value.x1x2.x2 = (-b + d) / (2 * a);
26. }

27. void EvalNoRoot(Root& root) {
28.     root.key = ZERO;
29.     root.value.mes = "No of roots!";
30. }

31. void SqRoot(double a, double b, double c, Root &root) {
32.     double d = b * b - 4 * a * c;
33.      if(d == 0) {
34.         EvalOneRoot(a, b, root);
35.     } else if(d > 0) {
36.         EvalTwoRoots(a, b, d, root);
37.     } else {
38.          EvalNoRoot(root);
39.     }
40. }

41. void Out(double x) {
42.     cout << "x = " << x << endl;
43. }

44. void Out(TwoRoots &x1x2) {
```

```
45.     cout << "x1 = " << x1x2.x1 << endl;
46.     cout << "x2 = " << x1x2.x2 << endl;
47. }

48. void Out(char* mes) {
49.     cout << "Confuse: " << mes << endl;
50. }

51. void Out(Root &root) {
52.     if(root.key == ONE) {
53.         Out(root.value.x);
54.     } else if(root.key == TWO) {
55.         Out(root.value.x1x2);
56.     } else {
57.         Out(root.value.mes);
58.     }
59. }

60. void main() {
61.     double a, b, c;
62.     Root root;
63.     cout << "Input a, b, c:\n";
64.     cin >> a >> b >> c;
65.     SqRoot(a, b, c, root);
66.     Out(root);
67. }
```

Типичным для процедурного подхода является использование алгоритмического анализа альтернатив. В представленной выше программе с их помощью осуществляется вывод требуемого значения корня (строки 51-59). В данном случае, для этого используется условный оператор, который, в результате, обращается к требуемой функции вывода.

Аналогичная программа, написанная с использованием объектно-ориентированного подхода, представлена в листинге 2.

Листинг 2. Использование объектно-ориентированного подхода

```
1.  #include <iostream>
2.  #include <math.h>

3.  using namespace std;

4.  class Root {
```

```
5.  public:
6.      virtual void Out() = 0;
7.  };

8.   class OneRoot: public Root {
9.   public:
10.     OneRoot(double a, double b) {
11.         x = (-b) / (2 * a);
12.     }
13.     void Out() {
14.         cout << "x = " << x << endl;
15.     }
16. private:
17.     double x;
18. };

19. class TwoRoots: public Root {
20. public:
21.     TwoRoots(double a, double b, double d) {
22.         d = sqrt(d);
23.         x1 = (-b - d) / (2 * a);
24.         x2 = (-b + d) / (2 * a);
25.     }
26.     void Out() {
27.         cout << "x1 = " << x1 << endl;
28.         cout << "x2 = " << x2 << endl;
29.     }
30. private:
31.      double x1;
32.      double x2;
33. };

34. class NoRoot: public Root {
35. public:
36.     NoRoot() {
37.          mes = "No of roots!";
38.     }
39.     void Out() {
40.         cout << "Confuse: " << mes << endl;
41.     }
42. private:
43.      char* mes;
44. };
```

```
45. void SqRoot(double a, double b, double c, Root** root) {
46.     double d = b * b - 4 * a * c;
47.     if(d == 0) {
48.         *root = new OneRoot(a, b);
49.     } else if(d > 0) {
50.         *root = new TwoRoots(a, b, d);
51.     } else {
52.          *root = new NoRoot;
53.     }
54. }

55. void main() {
56.     double a, b, c;
57.     Root* root;
58.     cout << "Input a, b, c:\n";
59.     cin >> a >> b >> c;
60.     SqRoot(a, b, c, &root);
61.     root->Out();
62.     delete root;
63. }
```

В отличие от предыдущей программы, процедуры вычисления корней и вывода результатов "разбросаны" по различным классам, каждый из которых обеспечивает поддержку только своей специализации. Отсутствует в явном виде и проверка типа корня при его выводе методом ***Out***, что показывает мощь механизма виртуальных процедур.

Преимущества ОО программирования по сравнению с другими методами

Хотя ООП является лишь одной из завершающих частей ООМ, оно вносит существенный вклад в общую популярность подхода из-за наличия в ОО языках механизмов, более эффективно представляющих абстракции. Эти механизмы проявляются через соответствующую организацию классов, поддерживающих инкапсуляцию, полиморфизм и наследование, что обеспечивает более надежный доступ через интерфейсы, скрывая особенности реализации.

Преимущества объектно-ориентированного программирования вытекает из общих преимуществ объектного подхода [1] и выражаются в следующем:

1. Объектная модель позволяет в полной мере использовать выразительные возможности объектных и объектно-ориентированных языков программирования.
2. Использование объектного подхода существенно повышает уровень унификации разработки и пригодность для повторного использования не только программ, но и проектов.
3. Использование объектной модели приводит к построению систем на основе стабильных промежуточных описаний, что упрощает процесс внесения изменений.
4. Объектная модель уменьшает риск разработки сложных систем, прежде всего потому, что процесс интеграции растягивается на все время разработки, а не превращается в единовременное событие.

Если же языки ОО программирования применяются вне рамок объектной модели, их эффективность резко падает.

Существует, правда, авторитетное мнение [2, 4], что ОО языки не внесли ничего нового, а абстрактные типы данных (АТД) существовали и до них, но эффективно не использовалась. Действительно, используя любой язык, предоставляющий в распоряжение программиста АТД, можно написать программу в ОО стиле. Однако, как и любая модель, данная программа будет имитировать объекты большим числом простых понятий. Следует также отметить, что никто, в общем-то, и не пытался писать программы на процедурных языках (даже при наличии АТД) в объектном стиле. Обходились переключателями и вариантными записями. Просто тогда еще не пришло время программировать подобным образом. А когда оно наступило, ОО языки уже заняли соответствующую нишу. Кроме того, всегда приятнее использовать меньшее число языковых конструкций при выражении одного и того же понятия. Это прекрасно демонстрируется переходом от ассемблеров к языкам высокого уровня.

Вместе с тем, сторонники "простых" подходов Вирта, да и он сам, не отрицают плюсов ООП. В работе [8], посвященной ОО программированию на языке Оберон-2, приведен список достоинств ООП, проявляемых на уровне кодирования:

1. Классы позволяют проводить конструирование из полезных компонент, обладающих простыми инструментами, что дает возможность абстрагироваться от деталей реализации.
2. Данные и операции вместе образуют определенную сущность, и они не «размазываются» по всей программе, как это нередко бывает в случае процедурного программирования.
3. Локализация кода и данных улучшает наглядность и удобство сопровождения программного обеспечения.
4. Инкапсуляция информации защищает наиболее критичные данные от несанкционированного доступа.

Этим правилам можно следовать и при написании процедурных программ. Однако, отсутствие соответствующих языковых ограничений делает их соблюдение полностью зависимым от воли и желаний программистов.

Вместе с тем, Мессенбок также отмечает ряд достоинств, присущих только ОО программам. Эти достоинства связаны с возможностью построения производных классов (расширением типов в терминологии Оберона-2),

1. Обработка разнородных структур данных. Программы могут работать, не утруждая себя изучением вида объектов. Новые виды могут быть добавлены в любой момент.
2. Изменение поведения во время выполнения. На этапе выполнения один объект может быть заменен другим. Это может привести к изменению алгоритма, в котором используется данный объект.
3. Реализация родовых компонент. Алгоритмы можно обобщать до такой степени, что они уже смогут работать более чем с одним видом объектов.
4. Доведение полуфабрикатов. Компоненты не надо подстраивать под определенное приложение. Их можно сохранять в библиотеке в виде полуфабрикатов (semifinished products) и расширять, по мере необходимости, до различных законченных продуктов.
5. Расширение каркаса. Независимые от приложения части предметной области могут быть реализованы в виде каркаса и в дальнейшем расширены за счет добавления частей, специфичных для конкретного приложения.

Также Мессенбок останавливается и на достоинствах многоразового использования, проявляющихся при ООП:

1. Сокращается время на разработку, которое с выгодой может быть отдано другим проектам.
2. Компоненты многоразового использования обычно содержат гораздо меньше ошибок, чем вновь разработанные, ведь они уже не раз подвергались проверке.
3. Когда некая компонента используется сразу несколькими клиентами, то улучшения, вносимые в ее код, одновременно оказывают свое положительное влияние и на множество работающих с ней программ.
4. Если программа опирается на стандартные компоненты, то ее структура и пользовательский интерфейс становятся более унифицированными, что облегчает ее понимание и упрощает ее использование.

Недостатки ООП

Если вновь обратиться к Г. Бучу [1] и посмотреть на отмеченные им недостатки, то окажется, что они несущественные:

- небольшое снижение производительности ОО программ;

- трудности первоначального освоения.

Ряд недостатков отмечается и в работе Мессенбока [8]. Он отмечает необходимость более широких познаний предметной области, по сравнению с процедурным подходом (хотя, и там полезно знать аналогичные вещи):

1. Необходимо понимать базовые концепции, такие как классы, наследование и динамическое связывание. Для программистов, уже знакомых с понятием модуля и с абстрактными типами данных, это потребует минимальных усилий. Для тех же, кто никогда не использовал инкапсуляцию данных, это может означать изменения мировоззрения и может отнять на изучение значительное количество времени.
2. Многоразовое использование требует от программиста познакомиться с большими библиотеками классов. А это может оказаться сложнее, чем даже изучение нового языка программирования. Библиотека классов фактически представляет собой виртуальный язык, который может включать в себя сотни типов и тысячи операций. В языке Smalltalk, к примеру, до того, как перейти к практическому программированию, нужно изучить значительную часть его библиотеки классов. А это тоже требует времени.
3. Проектирование классов - задача куда более сложная, чем их использование. Проектирование класса, как и проектирование языка, требует большого опыта. Это итеративный процесс, где приходится учиться на своих же ошибках.
4. Очень трудно изучать классы, не имея возможности их «пощупать». Только с приобретением маломальского опыта можно уверенно себя почувствовать при работе с использованием ООП.

Наряду с проблемами первоначального обучения им также рассматриваются недостатки, связанные непосредственно с проектированием и кодированием.

1. Поскольку детали реализации классов обычно неизвестны, то программисту, если он хочет разобраться в том или ином классе, нужно опираться на документацию и на используемые имена. И время, которое было сэкономлено на том, что удалось обойтись без написания собственного класса, должно быть отчасти потрачено (особенно вначале освоения) на то, чтобы разобраться в существующем классе.
2. Документирование классов - задача более трудная, чем это было в случае процедур и модулей. Поскольку любой метод может быть переопределен, в документации должно говориться не только о том, что делает данный метод, но также и о том, в каком контексте он вызывается. Ведь переопределенные методы обычно вызываются не клиентом, а самим каркасом. Таким образом, программист должен знать, какие условия выполняются, когда вызывается данный метод.

Для абстрактных методов, которые пусты, в документации должно даже говориться о том, для каких целей предполагается использовать переопределяемый метод.

3. В сложных иерархиях классов поля и методы обычно наследуются с разных уровней. И не всегда легко определить, какие поля и методы фактически относятся к данному классу. Для получения такой информации нужны специальные инструменты вроде навигаторов классов. Если конкретный класс расширяется, то каждый метод обычно сокращают перед передачей сообщения базовому классу. Реализация операции, таким образом, рассредоточивается по нескольким классам, и чтобы понять, как она работает, нам приходится внимательно просматривать весь код.
4. Методы, как правило, короче процедур, поскольку они осуществляют только одну операцию над данными. Зато количество методов намного выше. Короткие методы обладают тем преимуществом, что в них легче разбираться, неудобство же их связано с тем, что код для обработки сообщения иногда «размазан» по многим маленьким методам.
5. Абстракция данных ограничивает гибкость клиентов. Клиенты могут лишь выполнять те операции, которые предоставляет им тот или иной класс. Они уже лишены неограниченного доступа к данным. Причины здесь аналогичны тем, что вызвали к жизни использование высокоуровневых языков программирования, а именно, чтобы избежать непонятных программных структур.
6. Абстракцией данных не следует злоупотреблять. Чем больше данных скрыто в недрах класса, тем сложнее его расширять. Отправной точкой здесь должно быть не то, что клиентам не разрешается знать о тех или иных данных, а то, что клиентам для работы с классом этих данных знать не требуется.

Мессенбок отмечает также и ряд недостатков, связанных с эффективностью использования ОО программ. Правда, следует отметить, что им же описываются разумные решения для их преодоления.

1. Неэффективность на этапе выполнения. В языках типа Smalltalk сообщения интерпретируются во время выполнения программы путем осуществления поиска их в одной или нескольких таблицах и за счет выбора подходящего метода. Конечно, это медленный процесс. И даже при использовании наилучших методов оптимизации Smalltalk-программы в десять раз медленнее оптимизированных С-программ.
2. Неэффективность в смысле распределения памяти. Динамическое связывание и проверка типа на этапе выполнения требуют по ходу работы информации о типе объекта. Такая информация хранится в дескрипторе типа, и он выделяется один на класс. Каждый объект имеет невидимый указатель на дескриптор типа для своего класса.

Таким образом, в объектно-ориентированных программах требуемая дополнительная память выражается в одном указателе для объекта и в одном дескрипторе типа для класса.

3. Излишняя универсальность. Неэффективность может также означать, что программа имеет ненужные возможности. В библиотечном классе часто содержится больше методов, чем это реально необходимо. А поскольку лишние методы не могут быть удалены, то они становятся мертвым грузом. Это не воздействует на время выполнения, но влияет на возрастание размера кода. Одно из возможных решений - строить базовый класс с минимальным числом методов, а затем уже реализовывать различные расширения этого класса, которые позволят нарастить функциональность. Другой подход - дать возможность компоновщику удалять лишние методы.

При более детальном рассмотрении можно отметить еще ряд настораживающих моментов [3].

1. При создании методов, описывающих взаимодействие множества объектов, приходится выбирать, какой из объектов будет содержать данный метод, а какие будут участвовать на правах элементов сообщения. Такой выбор трудно бывает сделать однозначно, так как в различных ситуациях, при выполнении одного и того же действия, удобнее осуществлять вызов метода из различных объектов. Поэтому приходится прибегать к избыточному размещению одних и тех же методов в различных классах или вводить внешнюю процедуру - друга этих классов, что ведет к смешению стилей программирования.
2. Разрабатываемая объектно-ориентированная модель предметной области не всегда оказывается адекватной природе моделируемых объектов и категорий. Это ведет к неоднозначному восприятию программы различными лицами в ходе ее разработки и сопровождения. Например, если в программе используется объект "Книга", то он должен предоставить множество своих интерфейсов для доступа к отдельным страницам, абзацам, содержанию, строкам и т.д., вместо предоставления описания своей структуры. Развивая эту точку зрения, можно говорить о том, что каждая программа должна содержать методы, обеспечивающие доступ к ее символам, лексемам, правилам и прочим объектам, необходимым компилятору, выступающему в роли клиента. Я с содроганием представляю себя в роли экземпляра объекта "Оперируемый", добровольно предоставляющего хирургу свои внутренности (и, особенно, свои знания о них), методы самовскрытия живота (не путать с методом "Харакири"), ковыряния в нем с последующим зашиванием и самооставлением, на память, рваного рубца.

3. Проблема наращивания при изменении основополагающих понятий. Возникает, когда в уже разработанную и используемую программу требуется добавить новый объект, активно взаимодействующий с ранее созданными. Наличие тесных взаимодействий между данным объектом и уже существующими может привести к решению, связанному с его включением в соответствующую иерархию классов. Однако, возможное наличие у объекта новых методов и необходимость доступа к ним через базовый класс может потребовать изменения интерфейса базового класса, что ведет к переделке программы.
4. ОО подход ведет не только к "размазыванию" по родственным классам одного метода обработки. Он также "размазывает" процесс согласованного взаимодействия нескольких программных объектов, заменяя его методами, осуществляющими обмен сообщениями между классами. При этом для реализации одного согласованного процесса, требующего нескольких последовательных обменов данными, часто приходится создавать несколько мелких методов, последовательно передающих из класса в класс кусочки сообщений. Ситуация еще больше усложняется, когда взаимодействуют несколько классов. Поэтому, зачастую радикальным решением проблемы является создание процедуры (функции), являющейся другом всех этих классов, реализующей данный процесс централизованно. То есть, переход к процедурному программированию.
5. Отличие способа представления полиморфизма на уровне операций от способа его представления на уровне объектов. Любой язык программирования высокого уровня изначально поддерживает полиморфизм данных на уровне его операций. Записывая операцию сложения, программист сразу же подразумевает, что она, в качестве операндов, может использовать любые предопределенные значения (не обязательно, что они могут быть только числовыми). Этот полиморфизм операндов поддерживается автоматически, так как реализован на уровне системы. Вместо этого, полиморфизм классов реализуется через его виртуальные функции, что предопределяет другой образ мышления и, следовательно, другую реализацию. Это, конечно, не является тяжким грехом многих ОО языков, но вносит определенную долю сомнений в полную непогрешимость объектной модели.

Параметрическая обработка альтернатив

Представленное выше сравнение процедурного и объектного подходов, затрагивающее их конструктивное различие и сходство по формированию одинаковых категорий программных объектов (с использованием разных конструктивных элементов), позволяют сопоставить существую-

щие парадигмы программирования с подходом, предлагаемым в данной работе. Существует еще один вариант вычисления ***F(X)***. Мы можем предварительно установить новую функциональную зависимость между $\mathbf{f_i}$ и $\mathbf{t_j}$: $\mathbf{G(f_i, t_j)}$. Такая однозначная зависимость, назовем ее ***параметрической***, может быть сформирована, например, конкатенацией текущих значений функции и типа как двух слов фиксированной размерности. В этом случае воображаемый процессор может обрабатывать только один вид альтернатив.

$$
\begin{aligned}
&p = G(f_i, type(x_j))\\
&switch(p)\ \{\\
&\quad case\ p_{11}: \varphi_{11}(x_1);\ break;\\
&\quad case\ p_{12}: \varphi_{12}(x_2);\ break;\\
&\qquad \dots\\
&\quad case\ p_{1n}: \varphi_{1n}(x_n);\ break;\\
&\quad case\ p_{21}: \varphi_{21}(x_1);\ break;\\
&\quad case\ p_{22}: \varphi_{22}(x_2);\ break;\\
&\qquad \dots\\
&\quad case\ p_{2n}: \varphi_{2n}(x_n);\ break;\\
&\qquad \dots\\
&\quad case\ p_{m1}: \varphi_{m1}(x_1);\ break;\\
&\quad case\ p_{m2}: \varphi_{m2}(x_2);\ break;\\
&\qquad \dots\\
&\quad case\ p_{mn}: \varphi_{mn}(x_n);\ break;\\
&\}
\end{aligned}
$$

Использование параметрической зависимости между аргументами позволяет убрать иерархию из анализа вариантов и обеспечивает независимость одного аргумента от другого, в данном случае процедур от данных. Такой подход составляет основу параметрического полиморфизма и процедурно-параметрической парадигмы программирования.

Параметрическое программирование

Одним из простых приемов параметризации является использование многомерных массивов, в которых каждая из размерностей определяет один из варьируемых параметров. Значениями элементов такого массива являются выбираемые альтернативы. Для рассматриваемого примера будет строиться двухмерный массив, или таблица:

Значения функции F	**Значения типа данных T**			
	$\mathbf{t_1}$	$\mathbf{t_2}$	…	$\mathbf{t_n}$
$\mathbf{f_1}$	φ_{11}	φ_{12}	…	φ_{1n}

f_2	φ_{21}	φ_{22}	…	φ_{2n}
…	…	…	…	…
f_m	φ_{m1}	φ_{m2}	…	φ_{mn}

Для табличного доступа не имеет значения, в какой последовательности определяются и используются индексы элемента. Программная реализация выбора альтернатив с применением параметров позволяет получить еще один подход, который можно назвать процедурно-параметрическим программированием.

Моделирование параметрического выбора альтернатив

Параметрический подход можно смоделировать, используя процедурное программирование. Ниже представлена программа нахождения действительных корней квадратного уравнения, использующая параметрические отношения между альтернативными типами данных и функций. Для того, чтобы иметь возможность обрабатывать значения типов без использования механизма их идентификации во время выполнения, введем перечислимый тип данных, каждое значение которого соответствует одному из типов:

Вид корня	**Обозначение типа данных**	**Значение перечислимого типа**
Один корень	OneRoot	ONE
Два корня	RwoRoots	TWO
Нет корней	NoRoot	ZERO

Формирования значения перечислимого типа осуществляется вместе с вычислением соответствующего значения корня в одном из специализированных методов Eval… Таким образом, однозначно формируются значение корня заданного типа и соответствующий ему параметрический коэффициент. Параметризации подвергаются два вида функции программы: функции вывода корня (Out) и функции разрушения динамической структуры данных, хранящей корень (Destroy). Таблица, описывающая параметризируемые функции данной программы, будет выглядеть следующим образом:

Значения функции	**Значения типа данных (задается через перечислимый тип)**		
	ONE	TWO	ZERO
Out	OutOneRoot	OutTwoRoots	OutNoRoot
Destroy	DestroyOneRoot	DestroyTwoRoots	DestroyNoRoot

Параметрическая программа, написанная с использованием процедурного подхода, представлена в листинге 3.

Листинг 3. Моделирование параметрического полиморфизма

```
1.  #include <iostream>
2.  #include <math.h>

3.  using namespace std;

4.  int const rootNum = 3; // количество вычисляемых корней
5.  enum rootType {ONE = 0, TWO, ZERO};

6.  // Внешняя оболочка для присоединения нужной структуры,
7.  // определяющая параметрическое обобщение
8.   struct Root {
9.       rootType type;
10.     void* value;
11. };

12. struct OneRoot {
13.     double x;
14. };

15. void OutOneRoot(Root& root) {
16.      cout << "x = " << ((OneRoot*)(root.value))->x << endl;
17. }

18. void DestroyOneRoot(Root& root) {
19.      delete (OneRoot*)(root.value);
20. }

21. void EvalOneRoot(double a, double b, Root& root) {
22.     root.value = (void*)new OneRoot;
23.     ((OneRoot*)(root.value))->x = (-b) / (2 * a);
24.     root.type = ONE;
25.  }

26. struct TwoRoots {
27.     double x1;
28.     double x2;
29. };

30. void OutTwoRoots(Root& root) {
31.     cout << "x1 = " << ((TwoRoots*)(root.value))->x1 << endl;
```

```
32.     cout << "x2 = " << ((TwoRoots*)(root.value))->x2 << endl;
33. }

34. void DestroyTwoRoots(Root& root) {
35.     delete (TwoRoots*)(root.value);
36. }

37. void EvalTwoRoots(double a, double b, double d, Root& root) {
38.     root.value = (void*)new TwoRoots;
39.     d = sqrt(d);
40.     ((TwoRoots*)(root.value))->x1 = (-b - d) / (2 * a);
41.     ((TwoRoots*)(root.value))->x2 = (-b + d) / (2 * a);
42.     root.type = TWO;
43. }

44. struct NoRoot {
45.   char* mes;
46. };

47. void OutNoRoot(Root& root) {
48.   cout << "Confuse: " << ((NoRoot*)(root.value))->mes << endl;
49. }

50. void DestroyNoRoot(Root& root) {
51.   delete (NoRoot*)(root.value);
52. }

53. void EvalNoRoot(Root& root) {
54.     root.value = (void*)new NoRoot;
55.     ((NoRoot*)(root.value))->mes = "No of roots!";
56.     root.type = ZERO;
57. }

58. void SqRoot(double a, double b, double c, Root& root) {
59.     double d = b * b - 4 * a * c;
60.     if(d == 0) {
61.       EvalOneRoot(a, b, root);
62.     } else if(d > 0) {
63.       EvalTwoRoots(a, b, d, root);
64.     } else {
65.       EvalNoRoot(root);
66.     }
67. }
```

```
68.// А здесь осуществляется привязка функций
69.// к ячейкам соответствующей таблицы
70. void (*Out[rootNum])(Root& root) =
71.     {OutOneRoot, OutTwoRoots, OutNoRoot};
72. void (*Destroy[rootNum])(Root& root) =
73.     {DestroyOneRoot, DestroyTwoRoots, DestroyNoRoot};

74. void main() {
75.     double a, b, c;
76.     Root root;
77.     cout << "Input a, b, c:\n";
78.     cin >> a >> b >> c;
79.     SqRoot(a, b, c, root);
80.     Out[root.type](root);
81.     Destroy[root.type](root);
82. }
```

Параметрический полиморфизм

Можно говорить о параметрическом полиморфизме как методе, обеспечивающим полиморфизм аргументов процедур путем создания параметрического обобщения, объединяющего альтернативные специализации. В листинге 3 в роли параметрического обобщения выступают:

- перечислимый тип **RootType** (строка 5);
- структура **Root** (строки 8-11).

Параметризация процедуры осуществляется по одному или нескольким аргументам, имеющим тип, соответствующий одному из типов, используемому в параметрических обобщениях. В рассматриваемой процедурной программе параметризация реализована с использованием массивов указателей на специализированные функции (строки 70-73). Имена массивов таких указателей служат именами обобщающих функций.

Приведенный способ написания программ может использоваться при императивном и функциональном программировании, а также при программировании с использованием процессов, обеспечивая при этом полиморфизм аргументов соответствующих процедур без изменения внутренней организации обобщающей процедуры. А ведь именно необходимость коррекции таких процедур при сопровождении сложных программных систем и явилась причиной заката процедурного программирования, не выдержавшего конкуренции с ООП. Однако, для того, чтобы данный метод мог действительно применяться как парадигма программирования, необходимо разработать соответствующие языковые средства, скрывающие особенности реализации механизма параметрического полиморфизма.

Конструирование программных объектов при процедурно-параметрическом подходе

ППП предлагает ряд специфических способов конструирования составных программных объектов, отсутствующих в процедурном и объектном подходе. Прежде чем приступить к описанию программ на воображаемом процедурно параметрическом языке, рассмотрим основные понятия, используемые при написании ПП программ:

1. Конструирование агрегатов осуществляется так же, как и при процедурном подходе, с использованием тех же структур и внешних процедур.
2. Структуры данных, представляющие отдельные специализации обобщения (***специализированные структуры***). По своим свойствам и назначению они полностью эквивалентны специализированным структурам процедурного подхода.
3. Процедуры, предназначенные для обработки данных внутри специализированных структур (***специализированные обработчики***). Они также эквивалентны соответствующим процедурам, используемым в процедурных языках.
4. Обобщения данных строятся с использованием новых программных объектов, названных ***параметрическими обобщениями***. Они предназначены для группировки параметров, обеспечивающих однозначную связь с соответствующими специализированными структурами. Данный программный объект не имеет аналогов в процедурном и ОО программировании. Он решает задачи, схожие с задачами вариантов в процедурных языках и базовых классов в ОО языках.
5. ***Обобщающие параметрические процедуры*** (или просто ***параметрические процедуры***) поддерживают механизм параметрического полиморфизма. Они не имеют аналогов в других парадигмах программирования. Их основная задача - описание параметрических аргументов, обработчика по умолчанию. Похожие, но не совпадающие свойства, можно наблюдать у методов базовых классов.
6. ***Специализированные параметрические обработчики*** (или просто ***параметрические обработчики***) - процедуры, расширяющие функциональность обобщающей параметрической процедуры. Они непосредственно задают функциональность для каждой из специализаций.

Механизм параметрического полиморфизма может встраиваться в любой язык программирования, в том числе и объектно-ориентированный. Поэтому, к списку основных программных объектов, в качестве вспомогательных, могут добавиться и классы.

Языковая поддержка параметрического полиморфизма

В работе не ставится задача описания еще одного языка программирования. Ниже предлагаются те конструкции, которые, на мой взгляд, были бы полезны при расширении любого языка программирования средствами представления параметрического полиморфизма. Делается попытка привязки этих конструкций к языку программирования C++, используемому в работе при написании реальных программ. Назначения этих конструкций и базовые принципы процедурно-параметрического программирования излагаются в комментариях к листингу 4, в котором представлена параметрическая программа нахождения действительных корней квадратного уравнения, написанная с использованием воображаемого языка.

Листинг 4. Пример параметрической программы

```
1.  #include <iostream>
2.  #include <math.h>

3.  using namespace std;

4.  struct OneRoot {
5.     double x;
6.  };

7.  void OutOneRoot(OneRoot &root) {
8.       cout << "x = " << root.x << endl;
9.  }

10. struct TwoRoots {
11.     double x1;
12.     double x2;
13. };

14. void OutTwoRoots(TwoRoots &root) {
15.      cout << "x1 = " << root.x1 << endl;
16.      cout << "x2 = " << root.x2 << endl;
17. }

18. struct NoRoot {
19.     char* mes;
20. };

21. void OutNoRoot(NoRoot &root) {
22.      cout << "Confuse: " << root.mes << endl;
```

```
23. }

24. // Параметрическое обобщение по типам данных
25. common Root { OneRoot, TwoRoots, NoRoot };

26. // Обобщающая параметрическая функция
27. // и обработчик по умолчанию.
28. void Out[Root &r](void) {
29.     if(!r.value())
30.         throw RelationError("Забыли инициировать!");
31.     throw RelationError("Что-то еще забыли!");
32.     // Исключение RelationError надо где-то описать
33. }

34. // Внешнее описание параметрической специализации.
35. void Out[Root &r(OneRoot)](void) { OutOneRoot(r); }
36. void Out[Root &r(TwoRoots)](void) { OutTwoRoots(r); }
37. void Out[Root &r(NoRoot)](void) { OutNoRoot(r); }

38. EvalOneRoot(double a, double b, OneRoot &root) {
39.     root.x = (-b) / (2 * a);
40. }

41. TwoRoots* EvalTwoRoots(double a, double b, double d) {
42.     TwoRoots* root = new TwoRoots;
43.     d = sqrt(d);
44.     root->x1 = (-b - d) / (2 * a);
45.     root->x2 = (-b + d) / (2 * a);
46.     return root;
47. }

48. void EvalNoRoot(Root& root) {
49.     root.mes = "No of roots!";
50. }

51. void SqRoot(double a, double b, double c, Root& root) {
52.     double d = b * b - 4 * a * c;
53.     if(d == 0) {
54.         root.init(OneRoot); // Выстраиваем завершенное отношение
55.         EvalOneRoot(a, b, root); // Формируем его значение
56.     } else if(d > 0) {
57.         // Привязываем отношение к уже готовому значению
58.         // через псевдофункцию
```

```
59.         root.link(TwoRoots) = EvalTwoRoots(a, b, d);
60.      } else {
61.         // А здесь еще один вариант.
62.         Root r(NoRoot); // Формируем отношение, используя
63.               // конструктор по умолчанию (это его демонстрация).
64.         EvalNoRoot(r);  // Заполняем его значением
65.         root.init(NoRoot) = r; // Присваиваем значение
66.      }
67. }

68. void main(void) {
69.    double a, b, c;
70.    Root root; // Не может завершить, так как пока не знаем тип
71.    cout << "Input a, b, c:\n";
72.    cin >> a >> b >> c;
73.    SqRoot(a, b, c, root);
74.    Out[root]();
75. }
```

В строках 4-23 описаны специализированные структуры данных для различных вариантов корней и функции вывода этих специализаций. Данная часть программы практически ничем не отличается от любой из программ, разработанных с использованием процедурного подхода. В строке 25 приводится описание параметрического обобщения:

common Root { OneRoot, TwoRoots, NoRoot };

Оно начинается с ключевого слова **common**, а в фигурных скобках записаны обобщаемые типы данных. Именно использование имен типов данных позволяет скрыть особенности внутренней организации механизма параметризации. Транслятор сам может сгенерировать внутренние конструкции, похожие на структуру **Root** и перечислимый тип **RootType**, представленные в строках 6-12 листинга 3.

В строках 28-37 производится описание параметрической функции **Out**, осуществляющей вывод различных вариантов корней. Описание сигнатуры функции, а также ее тела, осуществляющего обработку по умолчанию, представлены в строках 28-33. Следует отметить, что параметрический аргумент также включается в сигнатуру функции. Описанию функции может предшествовать параметрический прототип, что вполне согласуется с концепцией прототипов языка C++. Он может быть представлен в следующем виде:

void Out[Root &](void);

В дальнейшем, имеющаяся в нем информация может использоваться в точках вызова обобщающей функции. Она позволяет также описать лю-

бую внешнюю специализацию обобщения без наличия описания самого обобщения.

Параметрические обработчики представлены в строках 35-37. Каждый из них задает вывод своего корня. Для того, чтобы отличать различные параметрические значения внешних специализаций, каждый параметрический аргумент указывается со значением, определяющим специализацию данной функции.

В теле обобщающей функции пишется код, задающий обработку по умолчанию. В общем случае, это может быть произвольная обработка данных. В рассматриваемом примере (строки 30-31) тело функции содержит генераторы исключений для двух ситуаций:

- в начале осуществляется проверка на завершенность обобщающей переменной;
- затем выбрасывается исключение, подтверждающее отсутствие обработчика специализации для данного параметра.

Под завершенностью обобщающей переменной подразумевается установка ее параметрического значения. Предполагается, что такая переменная может быть создана с конкретным начальным значением или без него. Во втором случае невозможно правильно выполнить функции, связанные с обработкой значения параметра. Планируется, что обобщение будет иметь стандартные функции (методы), предназначенные для работы с ним. Набор этих функций будет уточняться при разработке конкретного языка. Здесь же упоминаются только те из них, которые кажутся целесообразными при построении примера. Не вызывает сомнений, что состав функций будет также зависеть от организации разрабатываемого языка и его структур данных. В строке 29 используется функция value, которая и проверяет обобщение на завершенность, Функция возвращает булевское значение true, если значение есть, и false, если оно отсутствует. Обработка исключения приведена в качестве примера. Поэтому, организация самого исключения не описана.

Тело обобщающей функции может быть и пустым, то есть, не содержать ни одного оператора. Тогда вся обработка данных будет осуществляться только написанными обработчиками специализаций. Если, в данной ситуации, для какого-то параметра соответствующая функция написана не будет, то выполнится пустое тело. Следует отметить: если у обобщающей функции есть возвращаемое значение, то в ее теле должен обязательно быть хотя бы один оператор **return**, возвращающий это значение.

Кроме того, обобщающая функция может быть чистой (по аналогии с чистой функцией абстрактного класса). В этом случае, транслятор может отслеживать, чтобы для каждого из объявленных параметров отношения была написана своя функция, являющаяся специализацией обобщения. Описание чистой функции, для рассматриваемого примера, будет выглядеть следующим образом:

void Out[Root &r](void) = 0;

Представленная строками 51-67 функция вычисления значения корней демонстрирует различные способы формирования обобщений, в зависимости от специфических особенностей языковых конструкций. Предполагается, что незавершенное обобщение передается в функцию по ссылке. Внутри данной функции вычисляется конкретное значение, которое, через механизм параметризации, соединяется с обобщением. В каждом из вариантов вычисления корня, для примера, используется свой способ соединения, обеспечивающий формирование завершенного обобщения.

Следует также отметить возможность обобщения классов. Предположим, что один из корней описан классом X_Root, который содержит метод Out(), осуществляющий вывод сформированного значения. Тогда, при включении типа X_Root в обобщение, его можно было бы связать со следующей внешней параметрической специализацией:

void Out[Root &r(X_Root)](void) { r.Out(); }

Таким образом, параметрический полиморфизм прекрасно уживается с объектно-ориентированным подходом, обеспечивая надстройку над объектами даже в уже существующих языках программирования.

Установление параметрической привязки

При наличии у квадратного уравнения одного корня, его окончательное решение состоит из следующих действий (строки 54-55). В начале формируется завершенное обобщение (строка 54). Для этого используется внутренняя функция **init**, которая дополняет объявленное обобщение структурой, позволяющей хранить тип **OneRoot**. Такое добавление прозрачно для программиста. Технически оно может быть реализовано с применением ссылочного механизма аналогично тому, как это было сделано в программе, представленной в листинге 3. Прозрачность данного механизма позволяет программисту писать функцию нахождения единственного решения так, как будто она манипулирует с исходным типом **OneRoot** (строки 38-40). Завершенное обобщение, в качестве фактического параметра ведет себя эквивалентно специализированному значению (строка 55), предоставляя точке вызова функции **EvalOneRoot** непосредственную ссылку на структуру **OneRoot**.

Реализация функции **EvalTwoRoots** (строки 41-47) позволяет продемонстрировать другой вариант завершения обобщения. Данная функция формирует и заполняет структуру при наличии двух корней уравнения. Результат вычисления данных корней возвращается через указатель на структуру **TwoRoots**, динамически сформированную в ходе вычислений. Предполагается, что обобщение, чтобы не проводить дополнительных пересылок данных, должно иметь внутреннюю функцию link, позволяющую связать существующий экземпляр специализации с уже существующими

данным. Параметр функции link используется для инициализации обобщения. Чтобы связывание прошло правильно, необходимо совпадение значения данного параметра с типом связываемой структуры. Функция link используется как псевдопеременная, что позволяет применять ее в левой части оператора присваивания (использовать как lvalue). Вычисление корня и "привязывание" вычисленного значения к обобщению представлено в строке 58.

Строки 61-66 демонстрируют еще один возможный случай, когда нет корней. Этот вариант сделан несколько искусственно, но он позволяет показать начальную инициализацию обобщения путем использования внутреннего конструктора. Такая инициализация осуществляется во время автоматического создания обобщения в строке 61. Сама функция **EvalNoRoot** (строки 48-50) построена по тому же принципу, что и **EvalOneRoot**. Она принимает незавершенное обобщения по ссылке, считая, что это структура **NoRoot**, присваивает ей значение и возвращает управление (строка 64). После этого осуществляется присваивание сформированного временного обобщения обобщению, переданному в качестве параметра в функцию **SqRoot** (строка 65). Данная строка демонстрирует возможность присваивания друг другу эквивалентных структур. Естественно, что обобщение, стоящее в левой части присваивания должно быть инициализировано. Предполагается, что инициализация (как и связывание) может быть псевдопеременной. Поэтому, она выполняется в том же операторе, что и присваивание.

В главной функции (строки 68-75) осуществляется использование параметрического полиморфизма. В строке 70 объявляется незавершенное обобщение, значение которого определяется при вычислении корня (строка 73). После этого вступает в дело параметрический полиморфизм (строка 74). Вызываемая параметрическая функция выводит полученное значение. По завершении главной функции, автоматически созданный экземпляр обобщения также автоматически удаляется. При этом предполагается, что не имеет значения, является он завершенным или нет. Эти вопросы должны, на мой взгляд, решаться внутренними механизмами языковой среды.

Программирование при множественном полиморфизме

Множественный полиморфизм возникает в том случае, когда процедура **F** обрабатывает список параметров, содержащий несколько обобщающих аргументов:

$$\mathbf{F(X_1, X_2, \ldots, X_n)}.$$

В этой ситуации перебор возможных вариантов расширяется за счет анализа альтернатив, связанных с обработкой дополнительных параметров. Процедурное программирование решает эту задачу простым алгоритмическим анализом с использованием вложенных условных операторов или пе-

реключателей. Однако в больших программных проектах чаще используется ООП.

Использование объектно-ориентированного подхода

Приведенный фрагмент программы демонстрирует возможности ОО подхода на примере сложения разнотипных чисел. В каждом из классов, представлен операнд определенного типа. В нем же инкапсулируется операция сложения данного операнда с другими. При этом операция сложения разбивается на две составляющие. Одна, используя полиморфизм, передает первый операнд перегруженному методу ***Add1***, который использует полиморфизм для определения типа второго операнда и выполнения операции сложения в методе ***Add2***.

Листинг 5. ОО реализация множественного полиморфизма

```
1.  // Суммирование с использованием ОО подхода
2.  struct INT_EL;
3.  struct FLOAT_EL;

4.   struct ELEMENT {  // Базовый класс для элементов
5.       virtual void Out(void) = 0; // вывод
6.       // Определение типа первого операнда при сложении:
7.       virtual ELEMENT* Add1(ELEMENT*) = 0; //
8.       // Передача первого операнда и сложение со вторым:
9.       virtual ELEMENT* Add2(INT_EL*) = 0;
10.      virtual ELEMENT* Add2(FLOAT_EL*) = 0;
11. };

12. struct INT_EL: ELEMENT { // Целочисленный элемент
13.      int number;  // Значение элемента
14.      INT_EL(int n = 0) { number = n; } // конструктор класса
15.      void Out(void) { cout << "INT " << number << "\n"; }  // Вывод
16.      // Определение типа первого операнда при сложении:
17.     ELEMENT* Add1(ELEMENT* opd2) { return opd2->Add2(this); }
18.      // Передача первого операнда и сложение со вторым:
19.      ELEMENT* Add2(INT_EL* opd1)
20.           { return new INT_EL(opd1->number + this->number); }
21.     ELEMENT* Add2(FLOAT_EL* opd1)
22.           { return new FLOAT_EL(opd1->number + this->number); }
23. };

24. struct FLOAT_EL: ELEMENT { // действительный элемент
25.      float number;  // значение элемента
```

```
26.   FLOAT_EL(float n = 0) { number = n; } // конструктор класса
27.   // Вывод элемента класса:
28.   void Out(void) { cout << "FLOAT " << number << "\n"; }
29.   // Определение типа первого операнда при сложении:
30.   ELEMENT* Add1(ELEMENT* opd2) { return opd2->Add2(this); }
31.   // Передача первого операнда и сложение со вторым:
32.   ELEMENT* Add2(INT_EL* opd1)
33.        { return new FLOAT_EL(opd1->number + this->number); }
34.   ELEMENT* Add2(FLOAT_EL* opd1)
35.        { return new FLOAT_EL(opd1->number + this->number); }
36. };

37. // Функция, вычисления суммы элементов разного типа
38. ELEMENT* Addition(ELEMENT* opd1, ELEMENT* opd2) {
39.     return opd1->Add1(opd2);
40. }
```

Можно отметить несколько недостатков, присущих ОО при реализации множественного полиморфизма:

- операнд является объектом, не связанным с операциями над ним (представленная операция задана над отношениями операндов, а не отдельным операндом), поэтому инкапсуляция функции сложения в данном классе (как и любых других аналогичных функций) противоречит принципу модульности и независимости;
- в зависимости от количества операндов у добавляемых операций, приходится всякий раз вводить внутрь данного класса все новые и новые отношения, определяющие, в общем случае, n-полиморфизм, поэтому, наряду с отдельной функцией, задающей вычисления, будут существовать n-1 функций, задающих полиморфизм;
- хотя отношения между операндами для многих операций являются идентичными, каждая, вновь вводимая операция, чтобы обеспечить вычисление своей функции, требует введения нескольких дополнительных методов (например, операции, вычитания, умножения и т.д. нужно реализовать по той же схеме, что и сложение);
- дополнительные отношения между объектами различного типа обычно составляют независимые категории (например, целый - целый, целый - действительный и т.д.), что в рассмотренном разбиении на классы никак не отражается;
- добавление нового типа обрабатываемых элементов (например, действительных чисел двойной точности или комплексных чисел) ведет к переделке интерфейсов как базового, так и всех производных классов (необходимо добавить целую серию обработчиков, обеспечи-

вающих поддержку множественного полиморфизма для всех операций класса).

Эти недостатки особенно отчетливо проявляются в более сложных ситуациях, когда необходимо осуществить взаимодействие разнородных объектов, таких как окон и сообщений в системах диалоговых интерфейсов. Часто разработчики в этих случаях отказываются от объектной реализации и используют традиционный процедурный подход, осуществляя обработку событий функциями объектов диалогового интерфейса, применяя для обработки сообщений алгоритмический перебор альтернатив.

Параметрический полиморфизм при нескольких аргументах

Процедурно-параметрическое программирование позволяет легко осуществить параметризацию нескольких аргументов, обеспечивая при этом независимость отдельных специализированных процедур. Соответствующая программа, написанная на воображаемом языке, представлена в листинге 6.

Листинг 6. Реализация множественного полиморфизма при ППП

```
1.  // Использование параметрического полиморфизма
2.  // Описание структур данных, представляющих операнды.
3.  struct INT_EL { // целочисленный элемент
4.      int number;        // значение элемента
5.  };

6.  struct FLOAT_EL { // действительный элемент
7.      float number;        // значение элемента
8.  };

9.  // Перегруженные функции вывода элементов
10. void Out(INT_EL &value)
        { cout << "INTEGER " << value.number << "\n"; }
11. void Out(FLOAT_EL &value)
        { cout << "FLOAT " << value.number << "\n"; }

12. // Сложение различных комбинаций элементов
13. INT_EL *Add(INT_EL *e1, INT_EL *e2)
14.       { return new INT_EL(e1->number + e2->number); }
15. FLOAT_EL *Add(INT_EL *e1, FLOAT_EL *e2)
16.       { return new FLOAT_EL(e1->number + e2->number); }
17. FLOAT_EL *Add(FLOAT_EL *e1, INT_EL *e2)
18.       { return new FLOAT_EL(e1->number + e2->number); }
19. FLOAT_EL *Add(FLOAT_EL *e1, FLOAT_EL *e2)
```

```
20.       { return new FLOAT_EL(e1->number + e2->number); }

21. // Обобщение операндов различных типов
22. common ELEMENT { INT_EL, FLOAT_EL, };

23. // Параметрическое обобщение функции вывода
24. void Out[ELEMENT &e](void) = 0; // Тело отсутствует

25. // Параметрические специализации функции вывода
26. void Out[ELEMENT &e(INT_EL)](void) { Out(e); }
27. void Out[ELEMENT &e(FLOAT_EL)](void) { Out(e); }

28. // Использование полиморфизма перегрузки позволяет
29. // записать все специализации обобщений следующим образом:
30. // void Out[ELEMENT &e(*)](void) { Out(e); }
31. // Предполагается автоматическая генерация 2-х вариантов
32. // функции Out,написанных выше "ручным" способом

33. // Параметрическое обобщение функции сложения
34. ELEMENT *Addition[ELEMENT *e1][ELEMENT *e2]() = 0;

35. // А здесь представлены 4 необходимые специализации обобщения

36. ELEMENT *Addition[ELEMENT *e1(INT_EL)]
                        [ELEMENT *e2(INT_EL)]() {
37.     ELEMENT* s = new ELEMENT(INT_EL);
38.     *s = *Add(e1, e2);
39.     return s;
40. }

41. ELEMENT *Addition[ELEMENT *e1(INT_EL)]
                        [ELEMENT *e2(FLOAT_EL)]() {
42.     ELEMENT* s = new ELEMENT;
43.     s->link(FLOAT_EL) = Add(e1, e2);
44.     return s;
45. }

46. ELEMENT *Addition[ELEMENT *e1(FLOAT_EL)]
                        [ELEMENT *e2(INT_EL)]() {
47.     ELEMENT* s = new ELEMENT;
48.     s->init(FLOAT_EL) = *Add(e1, e2);
49.     return s;
50. }
```

```
51. ELEMENT *Addition[ELEMENT *e1(FLOAT_EL)]
                        [ELEMENT *e2(FLOAT_EL)]() {
52.     return (new ELEMENT)->link(FLOAT_EL) = Add(e1, e2);
53. }

54. // Использование перегрузки функции Add позволяет
55. // записать все специализации обобщения Addition таким образом:
56. // ELEMENT *Addition[ELEMENT e1(*)][ELEMENT e2(*)]() {
57. //      return (new ELEMENT)->link(*) = Add(e1, e2);
58. // }
59. // Предполагается автоматическая генерация 4-х вариантов.

60. // Функции, формирующие начальные значения параметров

61. ELEMENT *Constructor(int n = 0) { // Целочисленное обобщение
62.      INT_EL *e = new int(n);
63.      return (new ELEMENT)->link(INT_EL) = e;
64. };

65. ELEMENT *Constructor(float n = 0) { // Действительное обобщение
66.     FLOAT_EL *e = new float(n);
67.     return (new ELEMENT)->link(FLOAT_EL) = e;
68. };

69. // Главная функция иллюстрирует использование
70. // параметрического  полиморфизма от нескольких аргументов
71. void main() {
72.     ELEMENT i(INT_EL) = *Constructor(10);
73.     ELEMENT *j = Constructor(5);
74.     ELEMENT s = Addition[i, j]();
75.     Out[*s]();
76.     ELEMENT *f = Constructor(2.5);
77.     s = Addition[i, f]();
78.     Out[*s]();
79. }
```

Отметим ряд особенностей представленной программы. В строках с 1 по 20 приведены конструкции, характерные для процедурного программирования. Описаны два типа данных, две функции ввода этих данных и четыре функции, обеспечивающие всевозможные комбинации сложения операндов. Следует обратить внимание на использование перегрузки имен функций. Если типы параметров известны на этапе компиляции, то пере-

грузка функций обеспечивает для них подобие полиморфизма. Однако если типы обрабатываемых переменных определяются во время выполнения программы, то данный механизм перестает работать. В представленной программе динамический полиморфизм между аргументами обеспечивается посредством параметрического механизма. В строке 22 сформировано обобщение объявленных типов данных.

В строке 24 представлена чистая параметрическая функция вывода. Отсутствие у функции тела говорит о том, что в программе обязательно должны присутствовать все специализации обобщения. Они представлены в строках 26-27. Использование перегрузки функций, допустимое в таком языке, как C++, позволяет предложить параметрическую конструкцию, похожую на макрорасширения и шаблоны. Если все специализации обобщения отличаются по записи только значениями специализированных параметров, их можно записать в общем виде, подставив вместо значений перебираемых параметров "*". Возможный вариант сокращенной записи параметрической специализации **Out** представлен в закомментированной строке 30. Необходимость такой конструкции - это еще одна тема, связанная с разработкой или расширением конкретного языка программирования.

В строке 34 приводится параметрическое обобщение функции сложения. Ниже (строки 36-53) представлены все четыре внешние специализации. Для демонстрации уже рассмотренных конструкций, в каждой из них используется свой способ формирования результата. Выполнение сложения в каждой из специализаций осуществляется представленными ранее перегруженными функциями (строки 13-20). В строках 56-58 также представлен общий для всех параметрических специализаций расширяемый (шаблонизируемый) вариант, позволяющий одновременно задавать все четыре одинаковые параметрические специализации. В этом случае значения обоих параметров заменены "звездочками".

В данной программе перегрузка используется также в функциях-конструкторах, облегчающих создание обобщающих операндов. По сравнению с конструкторами классов эти внешние функции выглядят достаточно громоздко. Поэтому, при разработке языка, имеет смысл подумать о внутренних конструкторах обобщений.

В целом же, параметрических полиморфизм является хорошей альтернативой ОО и процедурному подходам при обработке нескольких обобщений в качестве аргументов функции.

Достоинства и недостатки ППП

Трудно объективно оценивать достоинства и недостатки того, что еще не существует в реальном воплощении. Однако, предварительный анализ позволяет сделать некоторые выводы. По крайней мере, можно

провести поверхностное сравнение новой парадигмы программирования с ООП и процедурным подходом.

Достоинства ППП

1. Изучение возможностей модификации программ, написанных с применением ППП, позволяет сделать вывод о том, что они практически не уступают возможностям модификации ОО программ. Вместе с тем, следует отметить, что эти возможности обеспечиваются противоположным способом, при котором разработка обобщений осуществляется на основе уже существующих специализаций. Этот путь полностью совпадает с тем, который используется в процедурном программировании. Отличие лишь в том, что вариантное обобщение процедурного подхода заменяется параметрическим. Результатом же такой замены является возможность эволюционного наращивания и модификации кода, что отсутствует в процедурном подходе.
2. Построение параметрических обобщений соответствует восходящему проектированию. Это позволяет расширять существующий код путем повышения его универсальности, а не путем специализации, присущей ОО подходу.
3. Построение новых параметрических обобщений может осуществляться на основе уже существующих обобщений, причем, самым различным способом. Старые обобщения можно объединять включением, что позволяет строить иерархические конфигурации, можно сливать конкатенацией, используя при этом теоретико-множественные операции (объединение, пересечение разность и т.д.). При этом вновь разрабатываемые обобщающие параметрические процедуры могут использовать, в качестве своих параметрических специализаций, уже существующие обобщающие процедуры, построенные для реализации ранее существующих параметрических отношений.
4. Использование теоретико-множественных операций позволяет создавать ограниченные обобщения, скрывающие ряд специализированных процедур обработки. Это дает возможность создавать интерфейсы для различных клиентов с разными функциональными характеристиками без какого-либо переписывания методов.
5. Одним из основных достоинств ППП является поддержка множественного полиморфизма, которая достигается использованием параметрического списка аргументов. Параметрический список задает многомерный массив отношений между различными специализациями, каждому вектору которого соответствует своя специализированная процедура. Это позволяет гибко модифицировать функциональные возможности программы, обеспечивающей взаимодействие

между собой многих обобщенных объектов. ООП напрямую не поддерживает множественный полиморфизм.

6. Использование процедурно-параметрического подхода существенно повышает уровень унификации разработки. Повторно можно использовать ранее разработанные структуры данных и процедуры их обработки, создавая обобщения более высокого уровня. В принципе, ничто не мешает формировать обобщения на уровне уже готовых программ, если будут разработаны соответствующие инструменты, например, на основе языков сценариев. В отличие от ООП, унификация осуществляется не сверху вниз, а снизу вверх, путем обобщения уже созданного и работающего.
7. Применение процедурно-параметрического подхода ведет к построению простых функциональных модулей нижнего уровня, предназначенных для выполнения специализированных задач в различном окружении. Эти модули могут обобщаться различными способами, используя, в том числе, для построения унифицированных модулей более высокого уровня. Стабильность низкоуровневых модулей снижает риск разработки более универсальных приложений.
8. Отсутствие непосредственной зависимости процедур от данных, присущей классам, позволяет формировать внешние интерфейсы независимо от привязки к внутренней структуре программных объектов. Одна и та же обобщающая параметрическая процедура может одновременно (через механизм ссылок) присутствовать в нескольких интерфейсах, каждый из которых предназначен для своей категории клиентов. Отсутствие привязки интерфейсов к организации процедур и данных позволяет гибко создавать и модифицировать программные компоненты.
9. Параметрическое обобщение позволяет объединять самые различные типы данных. Их совместное использование в качестве альтернатив происходит только при написании соответствующих процедур обработки. Такая возможность позволяет легко обобщать понятия, зависимость между которыми при первоначальном проектировании программы не была установлена. При этом не требуется изменения и повторного проектирования уже существующих механизмов.
10. Использование, по сравнению с ООП, отделенных друг от друга данных и процедур позволяет повысить их возможности по применению в более крупных конструкциях. Одна и та же процедура, посредством явных или неявных ссылок может использоваться в различных интерфейсах, модулях, виртуальных конструкциях (формируемых инструментальной системой программирования в качестве альтернативного вида, но не имеющих физических или логических аналогов в тексте программы) и т.д.

11. Процедурно-параметрическая парадигма может использоваться совместно с объектно-ориентированным подходом. При этом последняя может осуществлять эволюционную специализацию программы сверху вниз, а первая - расширять ее обобщающие возможности снизу вверх. Кроме этого, методы классов можно расширять не только с использованием виртуализации, но и за счет параметризации.

Недостатки ППП

1. К самому существенному, на данный момент, недостатку следует отнести отсутствие практического подтверждения возможности использования данного подхода при разработке больших программных систем. Чтобы избавиться от этого недостатка, необходимо: провести более глубокие теоретические исследования, разработать соответствующие инструментальные средства, приступить к разработке процедурно-параметрической методологии. Глядя на предстоящие направления работ, убеждаешься, что этот недостаток будет полностью устранен в последнюю очередь.
2. Опираясь на процедурный подход, ППП наследует его гибкость при доступе к данным. Это может привести к написанию ненадежных программ. Устранению этого недостатка может способствовать создание соответствующей методологии, которая многое может заимствовать у объектно-ориентированной.
3. Невозможность непосредственного представления объектных абстракций предметной области, что связано с процедурной ориентацией парадигмы. Этот недостаток можно преодолеть, с созданием инструментальных средств, поддерживающих навигацию по программе и создание различных метафор на основе разноплановых композиций программных объектов. Такие инструменты по своим функциональным возможностям должны быть аналогичны современным браузерам классов.

Заключение

Представленная в работе процедурно-параметрическая парадигма программирования позволяет по-новому взглянуть на разработку сложных программных систем, их сопровождение, повторное использование и наращивание. На мой взгляд, эти проблемы могут решаться с не меньшей эффективностью, чем при использовании ОО подхода. Вместе с тем, ППП является дальнейшим развитием процедурного программирования и расширяет его полиморфизмом данных без использования алгоритмического анализа альтернатив. Открываются новые пути дальнейших исследований и практических разработок, среди которых можно отметить следующие:

- разработка теоретического фундамента ППП;

- разработка методологии процедурно-параметрического проектирования и программирования;
- включение механизмов процедурно-параметрического полиморфизма, в существующие языки программирования;
- разработка новых языков, поддерживающих процедурно-параметрическое программирование;
- создание интегрированных средств разработки приложений, ориентированных на ППП;
- интеграция ППП с ООП.

Вызывает также удивление то, что рассматриваемый метод расширения процедурных языков не был предложен раньше. Ведь для написания параметрических программ достаточно иметь процедурный язык, работающий с указателями и поддерживающий абстрактные типы данных. Никлаус Вирт мог бы еще раз сказать, что все это уже было давно разработано. На мой взгляд, повальное увлечение борьбой со сложностью с использованием ООМ увлекло всех в другую сторону, заставив забыть о тех направлениях исследований и разработок в области процедурных языков, которые раньше широко проводились. Процедурно-параметрическое программирование дает еще один шанс для дальнейшего развития уже подзабытых технологий.

Литература

1. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++, 2-е изд./Пер. с англ. - М.: "Издательства Бином", СПб: "Невский диалект", 1998 г. - 560 с., ил.
2. Вирт Н. Долой "жирные" программы. "Открытые системы", №6 (20), 1996, стр. 26-31.
3. Легалов А.И. Сочетание процедурного и объектного подходов при разработке программ. - Вестник Красноярского государственного технического университета. Сб. научных трудов. Вып. 10. Красноярск, 1997. с. 102-109.
4. Пешио К. Никлаус Вирт о культуре разработки ПО. "Открытые системы", №1, 1998, с. 41-44.
5. Хоар Ч. Взаимодействующие последовательные процессы. Пер. с англ. - М.: Мир, 1989. -264 с.
6. Henderson P. Functional Programming, Application and Implementation. /Prentice-Hall, London, 1980.
7. Meyer M. Object-Oriented Software Construction. Second Edition. ISE Inc. Santa Barbara (California).
8. Moessenboeck H., Wirth N. The Programming Language Oberon-2. Institut fur Computersysteme, ETH Zurich July 1996.