DOI: 10.15514/ISPRAS-2025-37(6)-3

Сравнение объектно-ориентированного и процедурно-параметрического полиморфизма

П.В. Косов, ORCID: 0000-0002-9035-312X <pvkosov@hse.ru>
А.И. Легалов, ORCID: 0000-0002-5487-0699 <alegalov@hse.ru>

Национальный исследовательский университет «Высшая школа экономики», Россия, 101000, г. Москва, ул. Мясницкая, д. 20.

Аннотация. Динамический полиморфизм часто применяется в ситуациях, связанных с определением и обработкой альтернативных ситуаций в процессе выполнения программ, обеспечивая гибкое расширение ранее написанного кода. Он широко используется в статически типизированных языках объектно-ориентированного программирования за счет совместного применения наследования и виртуализации. Языки программирования Go и Rust, также поддерживают динамический полиморфизм, используя для его реализации статическую утиную типизацию. Еще один подход предлагает процедурно-параметрическая парадигма программирования, обеспечивающая гибкое эволюционное расширение как вариантных данных, так и обрабатывающих их функций, включая ситуации, связанные с множественным полиморфизмом, которые возникают при реализации мультиметодов. В работе проводится сравнение возможностей динамического полиморфизма объектно-ориентированной и процедурно-параметрической парадигм, по поддержке гибкой разработки программного обеспечения. Сопоставляются базовые техники, обеспечивающие расширение функциональности программ, рассматриваются особенности реализации паттернов проектирования.

Ключевые слова: язык программирования; компиляция; процедурно–параметрическое программирование; полиморфизм; эволюционная разработка программного обеспечения; гибкая разработка программного обеспечения.

Для цитирования: Косов П.В., Легалов А.И. Сравнение объектно-ориентированного и процедурно-параметрического полиморфизмов. Труды ИСП РАН, том 37, вып. 6, часть 1, 2025 г., стр. 43–58. DOI: 10.15514/ISPRAS–2025–37(6)–3.

Comparison of Object-Oriented and Procedural-Parametric Polymorphism

P.V. Kosov ORCID: 0000-0002-9035-312X <pvkosov@hse.ru>
A.I. Legalov ORCID: 0000-0002-5487-0699 <alegalov@hse.ru>
Higher school of Economics, National research University,
20, Myasnitskaya st., Moscow, 101000, Russia.

Abstract. Dynamic polymorphism is widely used in situations involving the identification and processing of alternatives during program execution. Dynamic polymorphism allows to flexibly expand programs without changing previously written code. It is widely used in statically typed object-oriented programming languages by combining inheritance and virtualization. The programming languages Go and Rust also provide support for dynamic polymorphism, using static duck typing to implement it. Another approach to implementing dynamic polymorphism is offered by the procedural-parametric programming paradigm, which at the same time provides direct support for multimethods and flexible evolutionary expansion of both alternative data and their processing functions. The paper compares the capabilities of object-oriented and procedural-parametric paradigms to support agile software development. The basic techniques that ensure the expansion of the functionality of programs are compared. The features of the implementation of design patterns are considered.

Keywords: programming language; compilation; procedural-parametric programming; polymorphism; evolutionary software development; agile software development.

For citation: Kosov P.V., Legalov A.I. Comparison of object-oriented and procedural-parametric polymorphism. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, part 1, 2025, pp. 43-58 (in Russian). DOI: 10.15514/ISPRAS-2025-37(6)-3.

1. Введение

При разработке программного обеспечения учитываются различные критерии качества. Расширение программы без изменения ранее написанного кода или с его минимальными изменениями является одним из них. Его важность обуславливается как неполным знанием об окончательной функциональности программ во время их создания и начальной эксплуатации, так и необходимостью ускорить выход продуктов на рынок за счет реализации только базового набора функций с последующим его наращиванием. В подобных ситуациях добавление новых конструкций, изменяющих уже написанный код, зачастую ведет к появлению неожиданных ошибок и непредсказуемому поведению.

Интерес к эволюционному расширению программ возник достаточно давно. Например, в работе [1] рассмотрена технология вертикальных слоев, выстраиваемых на основе выделения функций, расширяющих «обедненную» версию программы. Была предложена практическая реализация вертикального слоения в виде концепции сосредоточенного описания рассредоточенных действий. В работах [2-6], наряду с более детальным анализом механизмов реализации технологии вертикального слоения, даны технические рекомендации по ее реализации.

Эволюционное расширение во многом реализуется за счет динамического связывания программных объектов, что обеспечивает гибкое изменение структуры программы во время выполнения. При этом могут использоваться как явная, так и неявная обработка подключаемых альтернативных объектов. Последняя определяет динамический полиморфизм. Он отличается от статического полиморфизма, при котором выявление альтернатив осуществляется во время компиляции.

Изначально реализация динамического полиморфизма, была предложена для объектно-ориентированной (ОО) парадигмы. В ОО языках со статической типизацией ключевым решением стало совместное использование механизмов наследования и виртуализации. Наследование обеспечило идентичность интерфейсов родительского и дочерних классов, а

виртуализация позволила подменять методы в дочерних классах. В качестве примера можно привести языки C++ [7], Java [8] и другие.

Поддержку динамического полиморфизма, основанную на статической утиной типизации, стали включать и в процедурные языки. В языке Go [9], реализован механизм интерфейсов, позволяющий использовать в качестве обработчиков альтернатив функции, связанные со структурами данных. Близкий механизм на основе типажей реализован в языке программирования Rust [10]. Вместе с тем, представленные выше подходы напрямую не поддерживают эволюционного расширения программ в случае множественного полиморфизма, связанного, например, с реализацией мультиметодов. Помимо этого, даже в случае одиночного полиморфизма, определяемого также как монометод, изменение функциональности достаточно часто ведет к модификации интерфейсов и классов. Достижение необходимого эффекта, связанного с безболезненным расширением программ, возможно только за счет написания дополнительного кода, формирующего необходимые композиции и обертки над программными объектами, которые не связаны с вычислениями, определяемыми прикладной задачей. Подобные решения, в частности, предлагаются объектно-ориентированными паттернами проектирования [11].

Для инструментальной поддержки эволюционно расширяемого множественного полиморфизма была предложена процедурно–параметрическая (ПП) парадигма программирования [12], использующая альтернативные методы для повышения возможностей процедурного подхода. Используемые для ее реализации технические решения базируются на оригинальном параметрическом механизме формирования отношений между данными и обрабатывающими их процедурами, который может быть реализован различными способами [13]. Они обеспечивают безболезненное и независимое расширение как данных, так и функций в статически типизированных языках программирования. Для первоначальной апробации идеи был разработан язык O2M, расширяющий язык программирования Оберон-2 [14]. Проведенные на его основе эксперименты позволили определить возможности процедурно–параметрического программирования (ППП). Было показано, что подход может быть интегрирован как в уже существующие языки процедурного и функционального программирования, так и использоваться при разработке новых языков. Также показано, что ПП парадигма обеспечивает более гибкое расширение программ по сравнению с другими методами поддержки динамического полиморфизма для статически типизированных языков программирования [15].

В работе рассматривается интеграция ПП механизмов в язык программирования C [16], который входит в пятерку наиболее популярных языков по различным рейтингам. Язык обладает гибкостью и относительной простотой, широко используется в предметных областях, где ОО подход не является эффективным. Вместе с тем, многие решения, применяемые при разработке программ на языке C, могут приводить к ошибкам за счет отсутствия в ряде случаев контроля за типами данных во время компиляции. Такие ситуации возникают из-за того, что язык поддерживает произвольные преобразования типов, связанные с их разыменованием. Несмотря на то, что это зачастую позволяет достичь высокой производительности, подобный стиль кодирования снижает надежность программ. Другой проблемой является отсутствие в языке жесткой зависимости между альтернативными данными и идентифицирующими их признаками (например, в объединениях), которые могут произвольно вводиться и использоваться программистом. Это тоже приводит к ошибкам при разработке программ, выявляемым только во время выполнения.

Многие из этих проблем эффективно решаются в ОО языках программирования, а также в языках Go и Rust за счет использования полиморфизма. ПП парадигма, наряду с решением этих проблем [17], позволяет также эффективно поддерживать множественный полиморфизм и эволюционное расширение мультиметодов. Проведенное моделирование вносимых в язык изменений, реализованное на языке программирования C в рамках операционной системы Linux, подтверждает возможности такой реализации [18].

В результате проведенных расширений компилятора CLang [19] сформировано надмножество языка, названное процедурно-параметрическим C (procedural-parametric C, PPC) [20]. Это обеспечило проведение экспериментов по созданию процедурно-параметрических программ и позволило провести сопоставление методов гибкого программирования, предлагаемых данной парадигмой с методами и техникой кодирования, применяемых в других подходах.

ОО подход в настоящее время является доминирующим в различных предметных областях. Поэтому представляет интерес сравнение его возможностей с возможностями, представляемыми ПП парадигмой программирования. Примеры, демонстрирующие возможности ПП подхода и его сравнение с другими парадигмами, представлены в [21].

2. Особенности процедурно–параметрической парадигмы

При описании особенностей процедурно–параметрической парадигмы используются следующие понятия [18]:

- параметрическое обобщение – структура, объединяющая различные альтернативы и выступающая в качестве общего родительского типа для всех альтернативных подтипов;
- специализация параметрического обобщения – структура, добавляющая к обобщению одну из сформированных альтернатив, в которой основа специализации выступает в качестве подтипа;
- основа специализации – любая абстракция данных, допускаемая в качестве расширения альтернативы;
- экземпляр параметрического обобщения или специализированная переменная – переменная, имеющая тип обобщения и подтип одной из альтернативной специализации;
- обобщающая функция – функция, определяющая общий интерфейс для множества альтернативных специализаций;
- обработчик параметрической специализации или специализированная функция – функция, осуществляющая обработку одной из комбинаций специализаций и допускающая в качестве аргументов от одной и более специализаций, подставляемых вместо обобщающих аргументов;
- вызов параметрической функции – полиморфный вызов обобщающей функции с аргументами, являющимися специализациями.

Варианты специализаций, реализованные в языке PPC, приведены на рис. 1.

Параметрическое обобщение (или просто «обобщение») используется для объединения в единую категорию различных альтернатив. В отличие от типа `union` языка программирования C, заключающего альтернативы внутри описания, что не позволяет их расширять без изменения внутренней структуры, в PPC расширение обобщения альтернативами осуществляется децентрализовано. В первоначальном определении обобщение может содержать несколько альтернатив или не содержать ни одной альтернативы, определяя некоторую основу для последующего их подключения. Синтаксически обобщение определяется как структура, содержащая общие данные, которые также могут отсутствовать. Для задания альтернатив, используются угловые скобки. Например, описание пустой обобщенной фигуры может выглядеть следующим образом:

```
typedef struct Figure {}<> Figure;
```

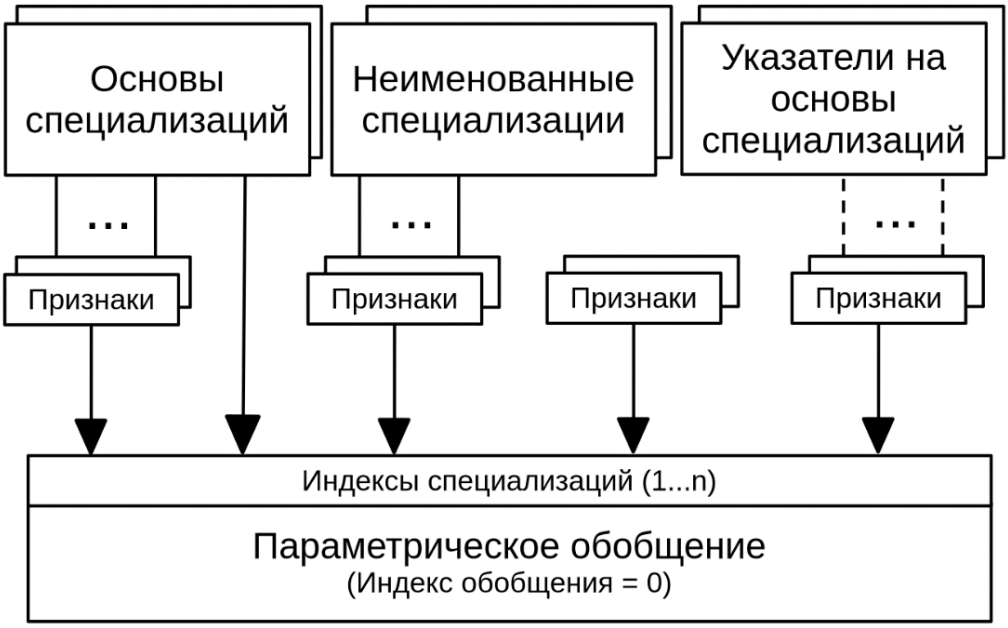

Рис. 1. Варианты формирования специализаций в языке программирования PPC.
Fig. 1. Options for the formation of specializations in the PPC programming language.

Использование описателя `typedef` не является обязательным. Однако в этом случае удобнее задавать описание типов структур только по именам. Обобщение может расширяться за счет добавления новых альтернатив, каждая из которых определяет одну из специализаций этого обобщения. В отличие от объединения языка C, расширение обобщения может осуществляться в различных единицах компиляции с использованием основ специализаций.

В качестве этих основ могут выступать именованные и неименованные типы данных, пустые типы и указатели на именованные типы данных. Именованные типы могут быть представлены базовыми типами и структурами. Например, в качестве такой структуры могут выступать явно задаваемые описания таких геометрических фигур, как прямоугольник и треугольник:

```
typedef struct Rectangle { int x, y; }    Rectangle;
typedef struct Triangle  { int a, b, c; } Triangle;
```

Добавление их в обобщение с формированием соответствующих специализаций будет выглядеть следующим образом:

```
Figure + < rect: Rectangle; >; // Создается фигура как прямоугольник
Figure + < Triangle; >;        // Создается фигура как треугольник
```

Формируемые при этом типы специализаций состоят из типа и подтипа. Тип определяется обобщением (`Figure`), а подтипы задаются либо явно признаками, либо непосредственно типом подключаемой основой специализации. Использование признаков обеспечивает уникальную идентификацию подтипа даже в том случае, когда одна и та же основа специализации многократно используется в качестве подтипа некоторого обобщения. Для приведенного обобщения в результате сформируются следующие типы специализаций:

```
Figure.rect        // Специализация фигуры - прямоугольника
Figure.Triangle    // Специализация фигуры - треугольника
```

Допускается формирование специализации с использованием неименованных структур. В этом случае для идентификации подтипа всегда используется признак:

```
Figure + < point: struct {int x, y;}; >; // Специализация фигуры - точки
```

Признак также используется при создании специализаций при отсутствии типа, что обеспечивается использованием ключевого слова `void`, а также в том случае, когда в качестве основы специализации выступает указатель на именованный тип. Например, таким образом можно описать набор специализаций, задающих дни недели. При этом в основной структуре можно указывать дополнительный параметр, определяющий номер недели:

```
// Дни недели
   struct WeekDay {int week_number;}
        <Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday: void;>
        const;
```

Ключевое слово `const` запрещает дальнейшее эволюционное расширение данного обобщения. При этом создаются специализации следующих типов:

```
WeekDay.Monday, WeekDay.Tuesday, WeekDay.Wednesday...
```

3. Сравнение объектно–ориентированного и процедурно–параметрического механизмов

Отличие подходов к реализации между объектно-ориентированным и процедурно–параметрическим полиморфизмом представлено на рис 2.

Объектно-ориентированный полиморфизм (рис. 2a) обеспечивается сочетанием наследования и виртуализации. Виртуализация при этом обычно реализуется за счет таблиц виртуальных методов (VT). Таблица базового класса Base, выступающего в роли обобщения, содержит указатели на один или несколько методов (F), которые обычно переопределяются в виртуальных таблицах производных классов, являющихся специализациями обобщения ($Child_1$–$Child_n$). При этом каждый производный класс, расширяя базовый класс, имеет собственный тип, что выражается в уникальном имени класса ($Child_i$). Производные классы формируются независимо друг от друга. Это позволяет эволюционно расширять альтернативные программные объекты, каждый из которых, при наличии одинаковых интерфейсов, может иметь иную функциональность, обеспечивая тем самым реализацию динамического ОО полиморфизма. Однако добавление нового виртуального метода требует модификации всей иерархии классов. Также данный механизм напрямую не поддерживает мультиметоды, для реализации которых ОО решением является использование диспетчеризации, не способствующей безболезненному расширению классов.

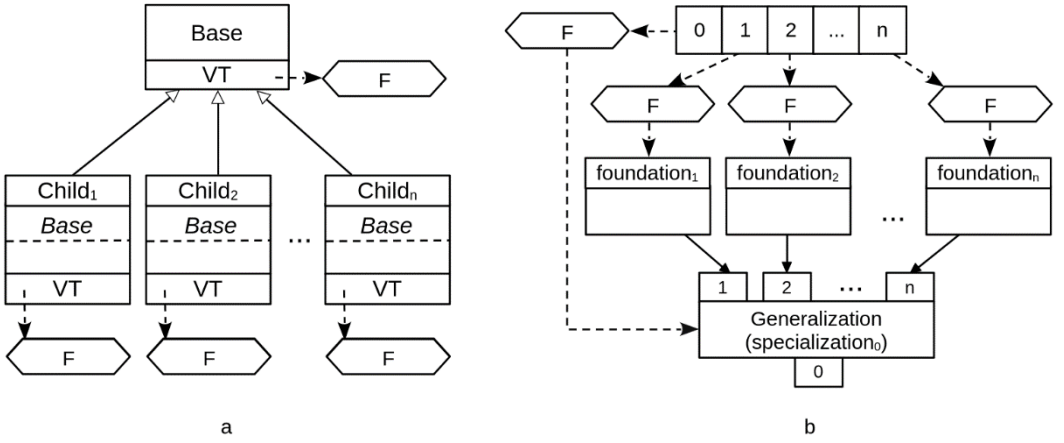

Рис. 2. Реализация ОО (a) и ПП (b) полиморфизма.
Fig. 2. Implementation of object-oriented (a) and procedural-parametric (b) polymorphis.

Процедурно–параметрический полиморфизм (рис. 2b) обеспечивает формирование альтернативных специализаций ($specialization_1$ – $specialization_n$), используя для этого основы специализаций ($foundation_1$ – $foundation_n$), которые являются подтипами данных. Добавление альтернативных специализаций к обобщающему их типу данных (Generalization) может осуществляться в произвольные моменты времени и в различных единицах компиляции. Формируемые при этом специализации обобщения принадлежат к тому же типу, что и обобщающий тип, являясь его подтипами. Само обобщение также трактуется как одна из специализаций ($specialization_0$), которая содержит только свои внутренние данные. Обработка сформированных альтернатив осуществляется с использованием внешних независимых специальных функций – обработчиков специализаций, расширяющих обобщающую их функцию. Они также могут добавляться независимо друг от друга. Механизм идентификации специализаций локализован внутри каждого обобщения с использованием внутренней индексации (параметрических индексов). Этот же механизм используется для автоматического выбора обработчиков специализаций через параметрические таблицы, каждая из которых связана со своим набором обобщающих функций. Добавление новых обобщающих функций и их обработчиков специализаций может осуществляться в любой момент времени независимо от уже существующих данных и функций, что характерно для процедурного подхода. Эти функции могут содержать произвольное число обобщающих аргументов, обеспечивая прямую реализацию эволюционно расширяемых мультиметодов [18].

3.1 Сопоставление базовых технических приемов

К базовым техническим приемам следует отнести формирование абстракций данных (классов) и функций (методов), обеспечивающих поддержку динамического полиморфизма. Их характеристики во многом определяют гибкую разработку программ и возможность их эволюционного расширения при добавлении новой функциональности. Непосредственное использование этих технических приемов позволяет не писать дополнительный код, напрямую не связанный с задачей предметной области.

Отличие между ОО и ПП подходами можно рассмотреть на простом примере обобщенной фигуры и таких ее специализаций как прямоугольник и треугольник. Пусть при использовании ОО парадигмы создается абстрактный базовый класс фигуры, содержащий чистый метод вычисления периметра. Помимо этого, пусть в данном классе присутствует общая переменная, задающая цвет фигуры и метод, обеспечивающий вывод этого параметра. Для демонстрации виртуального метода, не являющегося чистым, будем использовать вариант, проверяющий, что фигура является прямоугольником, который переопределяется в соответствующем производном классе.

```
class Figure {
  int color;
public:
  ...
  int GetColor() {return color;}
  virtual double Perimeter() = 0;
  virtual bool isRectangle() {return false;}
  ...
};
```

От этого класса формируются производные классы прямоугольника и треугольника.

```
class Rectangle: public Figure {
  int x, y;
public:
  ...
  virtual double Perimeter();
  virtual bool isRectangle() {return true;}
  ...
```

```
};
class Triangle: public Figure {
  int a, b, c;
public:
  ...
  virtual double Perimeter();
  ...
};
```

Реализация методов вычисления периметров может быть вынесена в независимые единицы компиляции.

```
double Rectangle::Perimeter() {
  return (double)(2*(x + y));
}
double Triangle::Perimeter() {
  return (double)(a + b + c);
}
```

При процедурно-параметрическом подходе в качестве альтернативы базовому классу можно создать обобщенную структуру, общее поле которой содержит цвет. Для запрета использования обобщения, не содержащего основы специализации, используется приравнивание к нулю.

```
typedef struct Figure { int color; }<> Figure = 0;
```

Аналогией подклассам является построение специализаций. Используем описанные выше структуры прямоугольника и треугольника, добавив их в обобщение с явными признаками.

```
Figure + < rect: Rectangle; >; // Фигура как прямоугольник
Figure + < trian: Triangle; >; // Фигура как треугольник
```

Вместо виртуальных методов формируются обработчики специализаций, которые являются расширениями обобщающей функции. Обобщенные аргументы этой функции переопределяются на специализации. За счет этого реализуется процедурно-параметрический полиморфизм [18], который для функции от одного полиморфного аргумента практически эквивалентен объектно-ориентированному полиморфизму. Эквивалентом чистого метода является абстрактная функция, которая должна быть обязательно переопределена на соответствующий обработчик для каждой специализации. У абстрактной обобщающей функции тело отсутствует.

```
double FigurePerimeter<Figure *f>() = 0; // Абстрактная обобщающая функция
// Обработчики специализаций
double FigurePerimeter<Figure.rect *f>() {
  return (double)(2*(f->@x + f->@y));
}
double FigurePerimeter<Figure.trian *f>() {
  return (double)(f->@a + f->@b + f->@c);
}
```

Для идентификации того, что осуществляется обращение к данным, предоставляемым основами специализаций, после ссылки «->» указывается признак основы «@». Возможно также использование обобщающих функций, эквивалентных виртуальным методам базового класса, содержащим реализацию. В этом случае аналогичная обобщающая функция ведет себя как обработчик по умолчанию для тех специализаций, которые не имеют собственного обработчика. Описанный выше виртуальный метод базового класса, проверяющий, является ли фигура прямоугольником, при ПП подходе будет реализован следующим образом: обобщающая функция будет выполнять проверку на прямоугольник для всех фигур, исключая сам прямоугольник.

```
_Bool isRectangle<Figure *f>() {return 0;}
```

Для прямоугольника будет сформирован соответствующий обработчик специализации.

```
_Bool isRectangle<Figure.rect *f>() {return 1;}
```

Как и в случае ОО подхода в этом случае не нужно реализовывать все обработчики специализаций. Достаточно реализовать только те из них, для которых требуется изменить функциональность.

Добавление новых функций для специализаций может осуществляться с передачей параметров через обычный список аргументов. Через этот же список специализации могут передаваться в функции, реализующие действия для переменных, расположенных внутри обобщений. Например, для вывода цвета фигуры может быть реализована обычная функция, получающая в качестве аргумента указатель на обобщение:

```
int GetColor(Figure *f) {return f->color;}
```

Так как специализации являются подтипами обобщения, они могут передаваться в эту функцию в качестве параметров.

Рассмотренные ситуации показывают, что процедурно-параметрический подход поддерживает основные методы формирования полиморфных отношений, аналогичные тем, что реализуются в ОО языках программирования со статической типизацией. Реализация полиморфизма в качестве расширения языка программирования С позволяет создавать более гибкий и надежный код за счет того, что вместо разыменования типов и прямого анализа вариантов при обработке альтернатив можно использовать обертки из обобщений и специализаций, обрабатываемые через вызов обобщающих функций [17]. Разбиение программы на отдельные функции, не содержащие общего анализа альтернатив, обеспечивают доступность кода для автоматического анализа, аналогичного по сложности анализу для языков С и С++.

Помимо этого, процедурно-параметрический полиморфизм обеспечивает более гибкое расширение кода по сравнению с ОО полиморфизмом. Это связано с использованием раздельного расширения данных и полиморфных функций. В случае ОО подхода добавление нового виртуального метода в базовый класс может привести к модификации наследуемых от него подклассов, особенно в том случае, если добавляется абстрактный метод.

Так как функции отделены от данных, то вопросы, как и в языке С, связанные с построением интерфейсов и использованием для их подключения множественного наследования, теряют смысл. Для группирования функций и их ограниченного использования соответствующими единицами компиляции достаточно сформировать соответствующие заголовочные файлы, содержащие прототипы необходимых обобщающих функций.

3.2 Мультиметоды и диспетчеризация

При процедурно-параметрическом подходе, мультиметоды могут быть непосредственно описаны в коде. Для этого в обобщающей функции достаточно указать несколько обобщенных аргументов [18]. Однако в ряде случаев, как и в статически типизированных ОО языках, для их реализации можно использовать диспетчеризацию. Такой прием может оказаться полезным, когда количество полиморфных аргументов в мультиметоде достаточно велико, а вариантов обработки комбинаций аргументов немного. Также при большом числе аргументов можно сочетать диспетчеризацию и мультиметоды для различных комбинаций аргументов.

В качестве примера, демонстрирующего возможности реализации диспетчеризации, можно рассмотреть мультиметод, осуществляющий анализ вложенности первой геометрической фигуры во вторую. Ранее данный пример рассматривался для демонстрации возможностей эволюционного расширения мультиметодов при процедурно-параметрическом подходе [18]. При первоначальном наличии только двух альтернативных специализаций (прямоугольника

и треугольника) вместо непосредственного описания обобщающей функции, определяющей мультиметод, формируются две обобщающие функции от одного полиморфного аргумента:

```
// Обобщающая функция, задающая вход в первую фигуру.
// Вторая фигура передается как обычный параметр
void Multimethod<Figure* f1>(Figure* f2) {} = 0;
```

Обработчики конкретных специализаций определяют полиморфный вход во вторую фигуру, относительно известной им специализации первого аргумента, запуская обобщенные функции, выполняющие обработку в зависимости от передаваемых в них специализаций:

```
// Обработчик специализации, когда первая фигура - прямоугольник
void Multimethod<Figure.rect* r1>(Figure* f2) {
  MultimethodFirstRect<f2>(r1);
}

// Обработчик специализации, когда первая фигура - треугольник
void Multimethod<Figure.trian* t1>(Figure* f2) {
  MultimethodFirstTrian<f2>(t1);
}
```

Обобщенные вспомогательные функции для обработки второго аргумента получают уже известную специализацию определяющие первый аргумент:

```
// Обобщающая функция, задающая вход во вторую фигуру,
// Когда первая фигура уже определена и это прямоугольник
static void MultimethodFirstRect<Figure* f2>(Figure.rect* r1) = 0;

// Обобщающая функция, задающая вход во вторую фигуру,
// Когда первая фигура уже определена и это треугольник
static void MultimethodFirstTrian<Figure* f2>(Figure.trian* t1) = 0;
```

Для каждой из четырех возможных комбинаций создается свой обработчик специализации для второго аргумента:

```
// Обработчик специализации для двух прямоугольников
static void MultimethodFirstRect<Figure.rect* r2>(Figure.rect* r1) {
  printf("Rectangle - Rectangle Combination\n");
}
// Обработчик специализации для прямоугольника и треугольника
static void MultimethodFirstRect<Figure.trian* t2>(Figure.rect* r1) {
  printf("Rectangle - Triangle Combination\n");
}
// Обработчик специализации для треугольника и прямоугольника
static void MultimethodFirstTrian<Figure.rect* r2>(Figure.trian* t1) {
  printf("Triangle - Rectangle Combination\n");
}
// Обработчик специализации для двух треугольников
static void MultimethodFirstTrian<Figure.trian* t2>(Figure.trian* t1) {
  printf("Triangle - Triangle Combination\n");
}
```

В целом можно отметить, что диспетчеризация внешне похожа на ОО решение. Однако при добавлении нового подкласса, например, для круга, в ОО подходе приходится изменять ранее написанные родительский класс и его подклассы. При процедурно-параметрическом программировании новые альтернативы добавляются без изменения ранее написанного кода. К еще одной положительной черте ПП подхода можно отнести то, что передаваемый пользователю интерфейс мультиметода, реализованного через диспетчеризацию, можно ограничить только внешней обобщающей функцией:

```
// Сигнатура для входа в диспетчеризацию мультиметода
void Multimethod<Figure* f1>(Figure* f2);
```

Промежуточные функции могут быть скрыты в отдельных единицах компиляции. Возможно как их отсутствие в заголовочных файлах, так и использование дополнительного ключевого слова `static`. В случае ОО подхода все промежуточные методы засоряют интерфейсы базового и производных классов, не используясь при этом во внешних взаимодействиях.

3.3 ПП реализация паттернов ОО проектирования

Для сопоставления с возможностями ОО программирования проведена реализация классических паттернов ОО проектирования, представленных в работе [11]. Практически для всех паттернов была реализована аналогичная функциональность и обеспечены соответствующие критерии качества. Независимость данных и функций в языке С в большинстве случаев позволили избавиться от ряда классов, ориентированных только на поддержку полиморфных интерфейсов, заменив их на эволюционно расширяемый перечислимый тип. Это же позволило обеспечить более гибкое расширение как полиморфных функций, так и данных без изменения ранее написанного кода практически для всех паттернов, в которых при ОО подходе присутствуют соответствующие ограничения. В ряде случаев появилась возможность использования более простых и эффективных решений, обуславливаемых как с особенностями традиционного процедурного подхода, так спецификой, добавляемой процедурно–параметрическим механизмом.

В качестве примера, демонстрирующего особенности использования процедурно-параметрического подхода, можно рассмотреть паттерн Visitor. Полностью его ОО реализация на языке C++ приведена в [22]. Упрощенно данный паттерн может быть представлен следующим кодом:

```
class ConcreteComponentA;
class ConcreteComponentB;
class Visitor { // Абстрактный базовый класс для посетителя
 public:
  // Требует изменения интерфейса при добавлении новых конкретных компонент
  virtual void VisitConcreteComponentA
       (const ConcreteComponentA *element) const = 0;
  virtual void VisitConcreteComponentB
       (const ConcreteComponentB *element) const = 0;
};
class Component {
 public:
  virtual ~Component() {}
  virtual void Accept(Visitor *visitor) const = 0;
};
class ConcreteComponentA : public Component {
 public:
  void Accept(Visitor *visitor) const override {
    visitor->VisitConcreteComponentA(this);
  }
  void ExclusiveMethodOfConcreteComponentA() const {...}
};
class ConcreteComponentB : public Component {
 public:
  void Accept(Visitor *visitor) const override {
    visitor->VisitConcreteComponentB(this);
  }
  void SpecialMethodOfConcreteComponentB() const {...}
};
class ConcreteVisitor1 : public Visitor {
 public:
  void VisitConcreteComponentA
       (const ConcreteComponentA *element) const override {
    element->ExclusiveMethodOfConcreteComponentA();
    // ... Специфические манипуляции A в ConcreteVisitor1
  }
```

```
    void VisitConcreteComponentB
        (const ConcreteComponentB *element) const override {
      element->SpecialMethodOfConcreteComponentB();
      // ... Специфические манипуляции B в ConcreteVisitor1
    }
};

class ConcreteVisitor2 : public Visitor {
 public:
    void VisitConcreteComponentA
        (const ConcreteComponentA *element) const override {
      element->ExclusiveMethodOfConcreteComponentA();
      // ... Специфические манипуляции A в ConcreteVisitor2
    }
    void VisitConcreteComponentB
        (const ConcreteComponentB *element) const override {
      element->SpecialMethodOfConcreteComponentB();
      // ... Специфические манипуляции B в ConcreteVisitor2
    }
};
```

Базовый класс посетителя является интерфейсом и переопределен в подклассах. Очевидно, что добавление нового конкретного компонента ведет к изменению этого интерфейса и его подклассов. Также можно отметить использование диспетчеризации.

Прямая процедурно-параметрическая имитация этого паттерна, также может быть выполнена с использованием диспетчеризации:

```
// Конкретные компоненты могут содержать любые данные
typedef struct ConcreteComponentA {} ConcreteComponentA;
// Функция, выполняемая конкретным компонентом A
const char* ExclusiveMethodOfConcreteComponentA(ConcreteComponentA* c) {
  return "A";
}

typedef struct ConcreteComponentB {} ConcreteComponentB;
// Функция, выполняемая конкретным компонентом B
const char* SpecialMethodOfConcreteComponentB(ConcreteComponentB* c) {
  return "B";
}

// Посетитель как обобщение, имитирующее перечислимый тип
typedef struct Visitor {}<> Visitor;
// Обобщающие функции, реализуемые для различных компонентов
void VisitConcreteComponentA<Visitor* v>(const ConcreteComponentA *c) = 0;
void VisitConcreteComponentB<Visitor* v>(const ConcreteComponentB *c) = 0;

// Компоненты также могут быть подтипами обобщения
typedef struct Component {}<> Component;
// Обобщающая функци доступа к компоненту с передачей посетителя
void Accept<Component* c>(Visitor *v) = 0;

// Каждый специализированный Компонент должен реализовать Accept,
// чтобы он вызывал метод посетителя, соответствующий типу компонента.
Component + <ConcreteComponentA;>;
// Доступ к компоненту A
void Accept<Component.ConcreteComponentA* c>(Visitor *v) {
  VisitConcreteComponentA<v>(&(c->@));
}
Component + <ConcreteComponentB;>;
// Доступ к компоненту B
void Accept<Component.ConcreteComponentB* c>(Visitor *v) {
  VisitConcreteComponentB<v>(&(c->@));
}
```

```
// Создание конкретных посетителей и обработчиков ими конкретных компонент.
Visitor + <ConcreteVisitor1: void;>;
void VisitConcreteComponentA
     <Visitor.ConcreteVisitor1* v>(ConcreteComponentA *c) {
  const char* str = ExclusiveMethodOfConcreteComponentA(c);
  printf("%s + ConcreteVisitor1\n", str);
}
void VisitConcreteComponentB
     <Visitor.ConcreteVisitor1* v>(ConcreteComponentB *c) {
  char* str = SpecialMethodOfConcreteComponentB(c);
  printf("%s + ConcreteVisitor1\n", str);
}
Visitor + <ConcreteVisitor2: void;>;
void VisitConcreteComponentA
     <Visitor.ConcreteVisitor2* v>(ConcreteComponentA *c) {
  char* str = ExclusiveMethodOfConcreteComponentA(c);
  printf("%s + ConcreteVisitor2\n", str);
}
void VisitConcreteComponentB
     <Visitor.ConcreteVisitor2* v>(ConcreteComponentB *c) {
  char* str = SpecialMethodOfConcreteComponentB(c);
  printf("%s + ConcreteVisitor2\n", str);
}
```

Однако, вместо абстрактного интерфейса используется эволюционно расширяемый перечислимый тип, сформированный на основе обобщения и обеспечивающий гибкое расширение в комбинации с обобщающей функцией и обработчиками специализаций для каждого конкретного компонента. При этом добавление новых данных позволяет безболезненно расширять Посетителя без изменения ранее написанного кода.

Диспетчеризацию также можно заменить на мультиметод, что позволяет получить более простое решение:

```
typedef struct ConcreteComponentA {} ConcreteComponentA;
// Функция, выполняемая конкретным компонентом A
const char* ExclusiveMethodOfConcreteComponentA(ConcreteComponentA* c) {
  return "A";
}
typedef struct ConcreteComponentB {} ConcreteComponentB;
// Функция, выполняемая конкретным компонентом B
const char* SpecialMethodOfConcreteComponentB(ConcreteComponentB* c) {
  return "B";
}
typedef struct Component {}<> Component;
Component + <ConcreteComponentA;>;
Component + <ConcreteComponentB;>;
typedef struct Visitor {}<> Visitor;
Visitor + <ConcreteVisitor1: void;>;
Visitor + <ConcreteVisitor2: void;>;
// Посетитель и компонент образуют мультиметод
void VisitComponent<Visitor* v, Component *c>() = 0;
// Обработчики специализаций, реализованные через мультиметод
void VisitComponent
     <Visitor.ConcreteVisitor1* v, Component.ConcreteComponentA *c>() {
  const char* str = ExclusiveMethodOfConcreteComponentA(&(c->@));
  printf("%s + ConcreteVisitor1\n", str);
}
void VisitComponent
     <Visitor.ConcreteVisitor1* v, Component.ConcreteComponentB *c>() {
  const char* str = SpecialMethodOfConcreteComponentB(&(c->@));
  printf("%s + ConcreteVisitor1\n", str);
}
void VisitComponent
     <Visitor.ConcreteVisitor2* v, Component.ConcreteComponentA *c>() {
  const char* str = ExclusiveMethodOfConcreteComponentA(&(c->@));
```

```
    printf("%s + ConcreteVisitor2\n", str);
}
void VisitComponent
     <Visitor.ConcreteVisitor2* v, Component.ConcreteComponentB *c>() {
  const char* str = SpecialMethodOfConcreteComponentB(&(c->@));
  printf("%s + ConcreteVisitor2\n", str);
}
```

Представленное решение по сути является прямой реализацией вариантов обработки нужных компонент, позволяя быстро и безболезненно добавлять как новые компоненты, так и варианты их обхода без изменения ранее написанного кода.

4. Заключение

Проведенное сравнение показывает, что процедурно-параметрическая парадигма программирования обеспечивает более гибкую разработку программ по сравнению с ОО подходом. При этом реализация соответствующей надстройки над языком программирования С позволяет разрабатывать более надежные и эволюционно расширяемые программы даже в рамках этого языка. Помимо этого, обеспечивается добавление новых альтернативных данных, а также функций, позволяющих безболезненно для ранее написанного кода использовать множественный полиморфизм.

Список литературы / References

[1]. Фуксман, А. Л. Технологические аспекты создания программных систем. / А. Л. Фуксман – М.: Статистика, 1979. – 184 с.

[2]. Горбунов-Посадов, М. М. Система открыта, но что-то мешает. / М. М. Горбунов-Посадов // Открытые системы. 1996. № 6. С. 36–39.

[3]. Горбунов-Посадов, М. М. Конфигурационные ориентиры на пути к многократному использованию. / М. М. Горбунов-Посадов – ИПМ им. М.В.Келдыша РАН. Препринт № 37, 1997 г.

[4]. Горбунов-Посадов, М. М. Облик многократно используемого компонента. / М. М. Горбунов-Посадов // Открытые системы. 1998. № 3. С. 45–49.

[5]. Горбунов-Посадов, М. М. Расширяемые программы. / М. М. Горбунов-Посадов – М.: Полиптих, 1999.

[6]. Горбунов-Посадов, М. М. Эволюция программы: структура транзакции. / М. М. Горбунов-Посадов // Открытые системы. 2000, № 10. С. 43–47.

[7]. M. Gregoire, Professional C++, John Wiley & Sons. 2018, p. 1122.

[8]. E. Sciore, Java Program Design, Apress Media. 2019, p. 1122.

[9]. A. Freeman, Pro Go: The Complete Guide to Programming Reliable and Efficient Software Using Golang, Apress. 2022, p. 1105.

[10]. J. Blandy, J. Orendorff, and L. F. Tindall, Programming Rust, O'Reilly Media. 2021, p. 735.

[11]. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns. Elements of Reusable Object-Oriented Software, Addison-Wesley Professional. 1994, p. 416.

[12]. Легалов А.И. Процедурно–параметрическая парадигма программирования. Возможна ли альтернатива объектно-ориентированному стилю? - Красноярск: 2000. Деп. рук. № 622-В00 Деп. в ВИНИТИ 13.03.2000. - 43 с. Доступна в сети Интернет: http://www.softcraft.ru/ppp/pppfirst/, accessed 30.08.2024.

[13]. Легалов И.А. Применение обобщенных записей в процедурно–параметрическом языке программирования // Научный вестник НГТУ, 2007. № 3 (28). С. 25–38.

[14]. Легалов А.И. Швец Д.А. Процедурный язык с поддержкой эволюционного проектирования. --- Научный вестник НГТУ, № 2 (15), 2003. С. 25-38.

[15]. Легалов А.И., Косов П.В. Эволюционное расширение программ с использованием процедурно–параметрического подхода // Вычислительные технологии. 2016. Т. 21. № 3. С. 56–69.

[16]. Прата С. Язык программирования С. Лекции и упражнения, 5-е издание. : Пер. с англ. – М.: Издательский дом «Вильямс», 2013. – 960 с.

[17]. Легалов А.И., Косов П.В. Процедурно-параметрический полиморфизм в языке с для повышения надежности программ. // XIV Всероссийское совещание по проблемам управления, ВСПУ-2024,

Москва 17-20 июня 2024 г. С. 2827-2833. https://vspu2024.ipu.ru/preprints/2827.pdf, accessed 30.08.2024.

[18]. Легалов А.И., Косов П.В. Расширение языка С для поддержки процедурно–параметрического полиморфизма. Моделирование и анализ информационных систем. 2023;30(1):40-62. doi: 10.18255/1818-1015-2023-1-40-62.

[19]. Clang: A c language family frontend for llvm. [Online]. Available: https://clang.llvm.org/, accessed 05.06.2025.

[20]. Язык процедурно-параметрического программирования РРС. Репозиторий компилятора языка на Gitverse: https://gitverse.ru/kpdev/llvm-project. Ветка pp-extension-v2, accessed 05.06.2025.

[21]. Примеры на Гитхаб, написанные с использованием процедурно–параметрической версии языка С: https://github.com/kreofil/evo-situations, accessed 05.06.2025.

[22]. Реализация паттерна ОО проектирования Visitor на языке программирования С++: https://refactoringguru.cn/ru/design-patterns/visitor/cpp/example, accessed 02.09.2025.

Информация об авторах / Information about authors

Павел Владимирович КОСОВ – аспирант факультета компьютерных наук НИУ «Высшая школа экономики». Его научные интересы связаны с разработкой языков программирования и компиляторов, оптимизацией LLVM, эволюционной разработкой программного обеспечения.

Pavel Vladimirovich KOSOV is a postgraduate student of the Faculty of Computer Science at the Higher School of Economics. His research interests are related to the development of programming languages and compilers, LLVM optimization, and evolutionary software development.

Александр Иванович ЛЕГАЛОВ – доктор технических наук, профессор факультета компьютерных наук НИУ «Высшая школа экономики». Его научные интересы: технологии программирования, эволюционная разработка программного обеспечения, архитектурно-независимое параллельное программирование, языки программирования.

Alexander Ivanovich LEGALOV – Dr. Sci. (Tech.), Professor of the Faculty of Computer Science at the Higher School of Economics. His research interests include software engineering, evolutionary software development, architecture-independent parallel programming, programming languages.