Evolutionary software development using procedural-parametric programming

Alexander Legalov
Sibreian Federal University
Russia
+7 913 830 5976
legalov@mail.ru

Pavel Kosov
Sibreian Federal University
Russia
+7 923 374 3428
kosov_@mail.ru

Эволюционная разработка программ с применением процедурно-параметрического программирования

Александр Легалов
Сибирский федеральный университет
Российская федерация
+7 913 830 5976
legalov@mail.ru

Павел Косов
Сибирский федеральный университет
Российская федерация
+7 923 374 3428
kosov_@mail.ru

ABSTRACT

The article examines the methods of evolutionary software development with the use of the procedural parametric paradigm of programming. The basic program object of the paradigm are presented and their implementation in a programming language Alien is shown. The generalized records and generalizing parametric procedures are described as well as their using for the enlargement of already written a program without changing the previously written code. We consider a number of techniques that demonstrate the peculiarities of the proposed programming style.

The use of generalizing procedures combined with generalized records instead of extensible types provides support for the multiple polymorphism as a special case includes a single polymorphism that is used in the objective-oriented approach. The flexibility of the proposed programming style in case of adding new data types is considered on the example of multimethods. In order to enhance the language capability it is proposed to use plug-in modules.

We have carried out the analysis of the situations that arise in the software development process and compared the possibility of using evolutionary expansion in procedural, object-oriented and procedural parametric approaches. We have considered adding new specializations to the generalizations, adding new procedures, adding new data fields to existing types, adding a new procedure designed to process only one of the specialization, creating a new generalization based on the existing specializations, adding new multimethods to the program and changing multimethods.

The article shows that the use of parametric generalizations combined with generalizing parametric procedures leads to the creation of more flexible evolutionary-enlargement programs. The complexity of building for generalized records is comparable with the creation of enlargement records with the use of programming language Oberon or class hierarchies of object-oriented languages. The use of plug-in modules facilitates the control of the program increment process.

The development of the programming language Alien allows us to carry out the research of capabilities for procedural parametric paradigm and to define the further ways of its development. In particular, the tool support for the evolutionary enlargement of multimethods allows carrying out a flexible direct coding without additional cost in the case of degeneration of multimethods to a simple constructions. Many of these constructions are currently implemented using a pattern design. The procedural parametric paradigm in general allows us to describe many of the patterns directly, ie, without specifying complex relationships between classes and without their dynamic binding at run time. This increases the efficiency of the written code and reduces its size.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features – *abstract data types, control structures, data types and structures, modules, packages, procedures, functions, and subroutines.*

General Terms

Algorithms, Design, Reliability, Languages.

Keywords

Evolutionary software development; procedure-parametric programming; programming languages; programming methods, programming paradigms

АННОТАЦИЯ

В статье рассматриваются методы эволюционной разработки программного обеспечения с применением процедурно-параметрической парадигмы программирования. Представлены основные программные объекты парадигмы и их реализация в языке программирования Alien. Приведено описание обобщенных записей и обобщающих параметрических процедур, показано каким образом они могут использоваться для безболезненного расширения уже написанной программы. Рассмотрен ряд приемов, демонстрирующих особенности предлагаемого стиля. На примере мультиметодов показана гибкость подхода при добавлении новых типов данных. Для повышения возможностей языка предложено использовать подключаемые модули. Проведен анализ ситуаций,

возникающих при разработке программ, в ходе которого сравнивались возможности использования эволюционного расширения в процедурном, объектно-ориентированном и процедурно-параметрическом подходах. Показано, что применение процедурно-параметрической парадигмы ведет к повышению гибкости и надежности программ.

Keywords
Эволюционная разработка программного обеспечения, процедурно-параметрическое программирование, языки программирования, методы программирования, парадигмы программирования

1. ВВЕДЕНИЕ
Эволюционная разработка является неотъемлемой чертой многих программных проектов. Она хорошо согласуется с инкрементальным проектированием, позволяя добавлять новый код без изменения ранее написанного. Доминирующий в настоящее время объектно-ориентированный (ОО) подход обеспечивает частичную инструментальную поддержку безболезненному расширению программ за счет использования наследования и виртуализации. Вместе с тем, эволюционное расширение только за счет ОО программирования является затруднительным. Для достижения требуемых критериев качества приходится использовать дополнительные структурные и алгоритмические приемы, динамическое связывание объектов или переходить к мультипарадигменному стилю.

Сочетание парадигм во многом связано с существованием задач, при решении которых применяются мультиметоды. И хотя в данной ситуации возможно чистое ОО эволюционное решение, опирающееся на диспетчеризацию [1], более эффективными и удобными являются мультипарадигменные варианты [2, 3]. Следует отметить, что после долгого перерыва, прошедшего со времени выхода языка CLOS, в настоящее время снова появился интерес к инструментальной поддержке мультиметодов. В частности возможность их гибкого расширения реализована в языке программирования C#. Однако предложенные решения обладают невысокой эффективностью, так как базируются на динамической проверке типов во время выполнения программы.

Другим подходом, изначально ориентированным на эффективную поддержку мультиметодов, является процедурно-параметрическая (ПП) парадигма программирования [4], добавляющая новые возможности по расширению программ в процедурный стиль. Для исследования парадигмы был разработан язык программирования O2M [5]. В последующих работах возможности парадигмы были расширены за счет использования обобщенных записей [6] и подключаемых модулей [7]. Данные конструкции были включены во вновь разрабатываемый язык процедурно-параметрического программирования Alien. Как и O2M, новый язык использует синтаксис, аналогичный синтаксису языка программирования Oberon-2 [8]. Однако если O2M разрабатывался как расширение Oberon-2, то Alien является языком, ориентированным на поддержку только процедурного подхода. Это обусловлено тем, что предлагаемые ПП понятия полностью дублируют, а зачастую и перекрывают имеющиеся в Oberon-2 расширяемые типы данных и процедуры, связанные с типом.

Использование обобщающих процедур в сочетании с обобщенными записями (вместо расширяемых типов) обеспечивает поддержку множественного полиморфизма и как частный случай включает одиночный полиморфизм, используемый в ОО подходе. Кроме того, для исследования возможностей ПП подхода целесообразным является создание языка, ориентированного только на эту парадигму, что позволяет иначе взглянуть на ряд методов разработки программ. Вместе с тем следует отметить, что разделение процедур и данных, позволяет реализовать ПП стиль не только в процедурных, но и в функциональных языках программирования.

2. ОСОБЕННОСТИ ПРОЦЕДУРНО-ПАРАМЕТРИЧЕСКОЙ ПАРАДИГМЫ
2.1 Организация обобщенных данных
ПП парадигма программирования базируется на концепции параметрического обобщения, обеспечивающего основу для группировки альтернативных понятий, называемых специализациями. Синтаксис обобщения в общем случае может быть представлен следующим образом:

ТипОбобщение* = *CASE [TYPE] OF [[LOCAL]
[СписокСпециализаций]]
[ELSE Специализация] END.
***СписокСпециализаций* = (**
СписокПризнаков «:» (Тип | NIL)) | Тип
{ «|» (СписокПризнаков «:» (Тип | NIL)) | Тип } .
СписокПризнаков* = *идентификатор
{ «,» идентификатор }.
Специализация* = (*идентификатор «:» (Тип | NIL)) | Тип .

Специализации строятся на основе ранее определенных типов (основ специализаций) и могут являться как обычными записями, так и обобщениями. Их отличие друг от друга обеспечивается признаками, даже если типы совпадают. В качестве примера основ специализаций можно привести записи, определяющие треугольник и прямоугольник.

```
// Прямоугольник со сторонами x, y
Rectangle = RECORD x, y: INTEGER END
// Треугольник со сторонами a, b, c
Triangle = RECORD a, b, c: INTEGER END
```

На их основе может быть создано параметрическое обобщение, содержащее прямоугольник и треугольник в качестве специализаций:

```
Figure = CASE OF
       rect: Rectangle |
       trian: Triangle
END
```

Обобщение можно расширить новыми специализациями в дополнительных модулях программы без изменения ранее написанного кода. Синтаксис расширения:

Расширение = Обобщение "+=" СписокСпециализаций.

Например, при добавлении круга можно использовать следующий код:

```
// Круг радиуса r
Circle = RECORD r: INTEGER END
// Расширение обобщения
Figure += circ: Circle
```

В языке программирования Alien, параметрическое обобщение может входить в состав обобщенной записи, что позволяет задавать как общие, так и альтернативные поля при описании абстрактных типов данных. Синтаксис обобщенной записи:

ТипОбобщеннаяЗапись = RECORD
[СписокПолей {";" СписокПолей}]
(Обобщение | [CASE ИмяОбобщения] END).

В целом трактовка данной конструкции практически ничем не отличается от вариантной записи языков программирования Pascal и Modula-2. Основное отличие проявляется в возможности расширения за счет добавления новых альтернатив. При этом параметрическое обобщение располагается в конце записи и позволяет использовать общие поля во всех специализациях, добавляемых при расширении. Допустимость только одного обобщения позволяет применять признак специализации для характеристики всей записи. В примере:

```
T0 = RECORD x: INTEGER; CASE END;
T0 += y: REAL;
```

T0 – запись, содержащая пустое обобщение. К ней добавляется специализация с признаком ***y*** вещественного типа. В результате возможно существование двух альтернативных понятий: записи, состоящей только из целочисленного поля ***x***, и записи, которая помимо поля ***x*** содержит вещественное поле с признаком ***y***. Обращение к этому полю состоит из идентификатора переменной и идентификатора признака, задаваемого в круглых скобках. Например:

```
VAR v(y): T0;
      ...
v(y) = 3.14;
```

Обозначение ***v(y)*** в последней строке является избыточным, так как признак специализированной переменной уже зафиксирован в ее объявлении. Он распознается во время компиляции и является неизменным. Поэтому можно использовать альтернативное обозначение:

```
v() = 3.14;
```

Скобки при этом остаются. Они сигнализируют об использовании обобщенной части. Допускается создавать записи с уже существующими обобщениями. В качестве примера можно рассмотреть обобщение геометрической фигуры с добавлением к каждой из конкретных фигур целочисленного поля, информирующего, например, об ее цвете:

```
// Фигура с цветом
ColoredFigure = RECORD
      color: INTEGER
      CASE Figure
END;
```

Подход удобен, когда расширение обобщения желательно скрыть от клиентского модуля, что позволит использовать только те специализации, которые имеются в библиотечных модулях разработчика.

В отличие от концепции базового типа, расширяемого за счет новых, хоть и «родственных» производных типов, использование параметрического обобщения предполагает, что основной тип остается неизменным, а альтернативы трактуются как уточнения исходного типа. То есть, внешне все специализации имеют единый тип, а их разнообразное толкование используется только внутри него. Это позволяет избавиться от глобальной идентификации типов, применяемой при наследовании, и обеспечивает поддержку концепции строгой типизации, как при статическом, так и динамическом связывании. Использование локальной идентификации альтернатив в свою очередь позволяет реализовать табличный доступ к обработчикам специализаций обобщающих процедур, что значительно ускоряет их вызов.

Обобщенные записи можно применять вместо наследования. Пусть базовый тип будет выстроен как запись с пустым обобщением:

```
T = RECORD x, y: INTEGER; CASE END;
```

Тогда на ее основе можно выстроить две специализации с явным указанием признаков:

```
T += t0: BOOLEAN;
T += t1: RECORD r: REAL; s: CHAR END;
```

Используя построенную обобщенную запись можно получить переменные типа ***T***, с двумя специализациями ***t0*** и ***t1***:

```
v0: T(t0);
v1:T(t1);
```

В качестве примера можно привести следующие варианты доступа к полям этих переменных:

```
v0.x, v1.y, v0(), v1().r ...
```

Круглые скобки, помимо задания признаков, отделяют поля основной записи от полей специализаций, что позволяет использовать в обеих частях обобщенной записи одинаковые имена.

Обработка обобщенных записей аналогична методам обработки расширяемых записей языка программирования Oberon. Допускается статическое и динамическое создание специализаций, указателей на обобщенные и специализированные записи, явное приведение обобщенного типа к типу специализации, проверка типа специализации подключенной к указателю на обобщенную запись.

Проверка типа специализации осуществляется операцией **IS**, в которой первым операндом является указатель на обобщенную запись, а в качестве второго операнда выступает признак специализации. Если к указателю на обобщенную запись подключена проверяемая специализированная запись, то в качестве результата возвращается значение ***TRUE***. Во всех иных случаях возвращается ***FALSE***. Например:

```
VAR pv: POINTER T; ...
pv := NEW(T(t0)); ...
IF pv IS T(t0) THEN ... ELSE ...
```

Прямое преобразование типа специализации может применяться и к указателю на обобщенную запись. Оно

успешно завершается, если подключаемая специализация соответствует преобразуемому типу. В противном случае происходит аварийное завершение программы. Данная операция должна использоваться совместно с операцией ***IS***. Например:

```
VAR pv: POINTER T; v: T(t0);
v := NEW(T(t0)); ...
IF v IS T(t0) THEN v() := pv(t0) END; ...
```

Обобщенному полю специализированной переменной присваивается значение обобщенной части динамической переменной. Предварительная проверка типа позволяет определить, что динамическая переменная ***pv*** имеет специализацию ***t0***. В противном случае присваивание не осуществляется.

Обобщенную запись и ее специализации допускается использовать в операторах присваивания. В частности, при наличии эквивалентных специализаций в левой и правой частях операторов присваивания, осуществляется присваивание всех полей записи, стоящей справа от знака «**:=**», полям, расположенным в его левой части. Если специализации различны, то присваивание осуществляется только для общих полей обычной записи. Допускается также прямое присвоение полям специализации обобщенной записи полей основы специализации. Аналогичная ситуация возможна и при использовании в левой части оператора присваивания основы специализации, когда в правой его части располагается соответствующая специализированная запись. В этом случае поля основы заполняются соответствующими полями специализации.

2.2 Обобщающие параметрические процедуры

Для обработки обобщений используются обобщающие параметрические процедуры [5]. Синтаксис обобщающей параметрической процедуры:

ОбобщающаяПроцедура = PROCEDURE Имя
СписокОбобщающихПараметров
[ФормальныеПараметры]
((ТелоПроцедуры Имя) | ":=" " 0") .

Отличие от обычной процедуры заключается в присутствии списка обобщающих параметров:

СписокОбобщающихПараметров =
"{"ГруппаОбобщающих
{";" ГруппаОбобщающих } "}".
ГруппаОбобщающих = [VAR] Идентификатор
{ "," Идентификатор }":" ОбобщающийТип.

Тело содержит обработчик по умолчанию, если для всех обобщающих параметров существует тип по умолчанию. В противном случае оно содержит обработчик исключений. Тело этой процедуры может отсутствовать, что задается «приравниванием» его нулевому значению. Тогда необходимы обработчики специализаций для различных комбинаций обобщенных параметров.

Процедура вывода геометрической фигуры с одним полиморфным параметром может выглядеть следующим образом:

```
// Обобщающая процедура вывода фигуры
// Если нужен только общий интерфейс
PROCEDURE Out {VAR s: Figure} = 0;

// Если используется обработчик по умолчанию
PROCEDURE Out2 {VAR s: Figure}; BEGIN
      SendException('Incorrect parameter')
END P2
```

Обобщенные параметры задаются в фигурных скобках. Обычные параметры могут следовать за ними в обычных круглых скобках. Отсутствие (необязательное) тела у обобщающей процедуры указывает на ее абстрактную роль.

Реальные вычисления осуществляются в процедурах – обработчиках специализаций, которые могут располагаться в разных модулях. Они обеспечивают обработку различных комбинаций специализаций, сопоставляемых с обобщениями из списка параметров. Комбинация, на которую «настроен» конкретный обработчик, задается значениями признаков в соответствии со следующими синтаксическими правилами:

ОбработчикСпециализации = PROCEDURE Имя
СписокСпециализаций [ФормальныеПараметры]
ТелоПроцедуры Имя .
СписокСпециализаций = "{" ГруппаСпециализаций
{ ";" ГруппаСпециализаций } "}" .
ГруппаСпециализаций = [VAR] Идентификатор
{ "," Идентификатор }
":" ОбобщающийТип "(" [Признак] ")"
| [VAR] Идентификатор "(" [Признак] ")"
{ "," Идентификатор "(" [Признак] ")"} ":"
ОбобщающийТип.

Каждый элемент списка специализированных параметров должен задавать конкретное значение признака. Специализации должны поэлементно соответствовать параметрам обобщающей процедуры. В обязательном теле процедуры кодируется конкретный метод обработки. Можно использовать один из двух способов задания специализаций, более удобный в рассматриваемом контексте: несколько одинаковых специализаций в группе или несколько разных специализаций в группе.

Для описанных выше специализаций обработчики представлены в виде процедур:

```
// Вывод параметров прямоугольника
PROCEDURE Out {VAR r(rect): Figure};
BEGIN
  Out.String(«Rectangle: x = », );
  Out.Int(r.x, 0);
  Out.String(«, y = », );
  Out.Int(r.y, 0);
END Out;
// Вывод параметров треугольника
PROCEDURE Out {VAR t(trian): Figure};
BEGIN
  Out.String(«Triangle: x = », );
  Out.Int(t.a, 0);
  Out.String(«, b = », );
  Out.Int(t.b, 0);
  Out.String(«, c = », );
  Out.Int(t.c, 0);
END Out;
```

Появление дополнительной специализации для круга потребует нового обработчика, который для процедуры вывода будет выглядеть следующим образом:

```
// Вывод параметров круга
PROCEDURE Out {VAR c(circ): Figure};
BEGIN
  Out.String(«Circle: r = », );
  Out.Int(c.r, 0);
END Out;
```

Обобщенные записи могут использоваться в качестве параметров процедур аналогично тому, как используются параметрические обобщения.

2.3 Дополнительные особенности обобщенных записей

Построение новых специализаций может также осуществляться с использованием обобщенной записи, что позволяет формировать цепочки уточнений произвольной длины. Подобное введение новых специализаций возможно только в том случае, если предшествующий тип является обобщенным. Это позволяет контролировать процесс добавления новых уточнений. Например, для формирования новой ступени обобщения ***T*** необходимо включить в него обобщение ***T0***, содержащее свою «точку» для расширения, которую можно «подключить» к ***T***:

```
T0 = RECORD z: INTEGER; CASE OF END;
T += CASE OF t2: T0; END;
```

Тип ***T0*** можно будет уточнять, добавляя для этого к нему новые специализации, которые также могут содержать обобщения, обеспечивающие их дальнейшее уточнение:

```
T00 = RECORD a: INTEGER; CASE OF END;
T0 += t00: T00;
```

Использование данного приема позволяет выстраивать сложные зависимости между типами. Для приведенного примера могут быть сформированы следующие специализации:

- из ***T*** => ***T(t0)*** или ***T(t1)*** или ***T(t2)***,
- из ***T(t2)*** => ***T(t2)(t00)*** и т.д.

Допускается также рекурсивное подключение к существующим обобщениям других обобщений, включая и подключение самого себя. При необходимости это позволяет выстраивать длинные статические цепочки, формируемые во время компиляции программы. В качестве примера можно расширить тип T специализацией, построенной на основе этого же типа:

```
T += t3: T;
```

Тогда появляется возможность выстраивать следующие специализации:

```
T(t3), T(t3)(t3), T(t3)(t3)(t3)(t3)(t2), …
```

Это позволяет применять технику формирования структур данных, базирующуюся на параметрическом подключении в конец одной обобщенной записи другую. Так можно набирать длинные статические цепочки требуемой конфигурации из универсальных базовых элементов. Аналогичные по структуре элементы, но в динамическом режиме, реализуются паттерном «Декоратор» [9]. Один и тот же декорирующий тип может многократно использоваться при порождении нового типа. Еще один вариант цветной геометрической фигуры может быть построен следующим образом:

```
// Общий декоратор
Decorator = CASE TYPE OF
  PointDecorator | ColorDecorator |
  AngleDecorator | Figure

END;
// Декоратор Точки
PointDecorator = RECORD
  p:Point CASE Decorator
END;
// Декоратор Цвета
ColorDecorator = RECORD
  c:Color CASE Decorator
END;
// Декоратор угла
AngleDecorator = RECORD
  alpha:REAL; CASE Decorator
END;
// Цветная фигура через декораторы
NewFigure3 = PointDecorator(
  AngleDecorator(
    ColorDecorator(
      ColorDecorator(Figure))));
VAR circle: NewFigure3(Circle);
rectangle: NewFigure3(Rectangle);
```

Подобные приемы повышают гибкость процедурно-параметрической парадигмы и, в сочетании с множественным полиморфизмом обобщающих процедур, обеспечивают эффективное создание эволюционно расширяемых и повторно используемых программных конструкций.

3. ПОДКЛЮЧАЕМЫЕ МОДУЛИ

Для повышения гибкости эволюционного расширения используются подключаемые модули [7]. Они связываются с базовым модулем, образуя единое пространство имен. Это отличает предлагаемый подход от импорта модуля, при котором внутренние пространства имен не пересекаются. Главной особенностью подключаемого модуля является возможность более удобного размещения обобщенных записей и обобщающих параметрических процедур. Она достигается за счет описания в нем специализаций и их обработчиков. Синтаксис подключаемого модуля:

ПодключаемыйМодуль =
MODULE идентификатор "("
идентификатор ["*" \|"+" "] ")" ";"
[СписокИмпорта]
ПоследовательностьОбъявлений
[BEGIN ПоследовательностьОператоров]
END идентификатор ".".
СписокИмпорта = IMPORT Импорт {"," Импорт} ";".
Импорт = [идентификатор ":="] идентификатор.

Отличие от обычного модуля проявляется в указании имени модуля-родителя (базового модуля) в круглых скобках. Знак «*» или «+» за именем родителя определяет права доступа к интерфейсу родительского модуля из модулей,

импортирующих подключаемый модуль. Правила использования этих знаков совпадают с их применением в языке программирования Oberon.

3.1 Особенности использования подключаемых модулей

В качестве примера, раскрывающего особенности использования подключаемых модулей, можно рассмотреть «хаотическое» расширение программы, использующей мультиметод. Пусть, на начальном этапе необходимо создать только прямоугольник и треугольник, реализовать которые можно в двух модулях ***MRect*** и ***MTrian***.

```
// Модуль, описывающий прямоугольник
MODULE MRect; IMPORT In, Out;
  TYPE
    PRectangle* = POINTER TO Rectangle;
    Rectangle* = RECORD
    x*, y* : INTEGER // стороны
  END;
  // Процедура вывода
  PROCEDURE Output*(VAR r: Rectangle);
  BEGIN
    Out.String("Rectangle: x = ");
    Out.Int(r.x, 0); Out.String(", y = ");
    Out.Int(r.y, 0); Out.Ln;
  END Output;
  // Прочие процедуры
  ...
END MRect.
```

```
// Модуль, описывающий треугольник
MODULE MTrian; IMPORT In, Out;
  TYPE
    PTriangle* = POINTER TO Triangle;
    Triangle* = RECORD
      a*, b*, c* : INTEGER // стороны
    END;
  // Процедура вывода
  PROCEDURE Output*(VAR t: Triangle);
  BEGIN
    Out.String("Triangle: a = ");
    Out.Int(t.a, 0); Out.String(", b = ");
    Out.Int(t.b, 0); Out.String(", c = ");
    Out.Int(t.c, 0); Out.Ln;
  END Output;
  // Прочие процедуры
  ...
END MTrian.
```

На следующем этапе в модуле ***MFig*** формируется обобщенная геометрическая фигура и создается обобщающая процедура ее вывода, использующая процедуры вывода конкретных геометрических фигур.

```
MODULE MFig; IMPORT In, Out, MRect, MTrian;
  TYPE
    // Указатель на обобщенную фигуру
    PFigure* = POINTER TO Figure;
    // Обобщение геометрической фигуры
    Figure* = CASE TYPE OF
      MRect.Rectangle | MTrian.Triangle
  END;
  // Обобщенная процедура вывода фигуры
  PROCEDURE Output* {VAR f: Figure} := 0;
  // Обработчики специализаций
  // Вывод обобщенного прямоугольника
  PROCEDURE Output
    {VAR r: Figure(MRect.Rectangle)};
  BEGIN MRect.Output(r) END Output;
  // Вывод обобщенного треугольника
  PROCEDURE Output
    {VAR t: Figure(MTrian.Triangle)};
  BEGIN  MTrian.Output(t) END Output;
END MFig.
```

Добавление новых процедур, расширяющих возможности приложения, происходит без изменения написанных модулей. Можно создавать как обычные обобщающие процедуры, так и мультиметоды. Добавим мультиметод, проверяющий возможность размещения первой фигуры внутри второй. Создадим подключаемый модуль ***MRTMM*** для создаваемых обобщающих и специализированных процедур.

```
MODULE MRTMM (MFig*);
  IMPORT In, Out, MRect, MTrian;
  // Обобщающая процедура, задающая
  // интерфейс мультиметода
  PROCEDURE FirstInSecond*
    {VAR f1,f2: Figure}:BOOLEAN := 0;
  // Обработчики специализаций
  // прямоугольник внутри прямоугольника
  PROCEDURE FirstInSecond
    {VAR f1,f2: Figure(MRect.Rectangle)}
                                  :BOOLEAN;
  BEGIN
    Out.String
        ("Rectangle-Rectangle compare");
    Out.Ln;
    RETURN ((f1.x < f2.x)&(f1.y < f2.y)) OR
           ((f1.x < f2.y)&(f1.y < f2.x))
  END FirstInSecond;
  // прямоугольник внутри треугольника
  PROCEDURE FirstInSecond
    {VAR f1(MRect.Rectangle),
     f2(MTrian.Triangle): Figure}:BOOLEAN;
  BEGIN ... END FirstInSecond;
  // треугольник внутри прямоугольника
  PROCEDURE FirstInSecond
   {VAR f1(MTrian.Triangle),
   f2(MRect.Rectangle):MFig.Figure}:BOOLEAN;
  BEGIN ... END FirstInSecond;
  // треугольник внутри треугольника
  PROCEDURE FirstInSecond
    {VAR f1, f2:
      MFig.Figure(MTrian.Triangle)}:BOOLEAN;
  BEGIN ... END FirstInSecond;
END MRTMM.
```

Добавление новых геометрических фигур тоже протекает без каких-либо изменений существующих модулей. Например, включим в программу круг, который опишем в отдельном модуле ***MCirc***.

```
MODULE MCirc; IMPORT In, Out;
  TYPE
    PCircle* = POINTER TO Circle;
    Circle* = RECORD
      r* : INTEGER // радиус
    END;
  // Процедура вывода
```

```
  PROCEDURE Output*(VAR c: Circle);
  BEGIN
    Out.String("Circle: r = ");
    Out.Int(c.r, 0); Out.Ln;
  END Output;
  // Прочие процедуры
  ...
END MCirc.
```

Для расширения уже существующей обобщенной фигуры и обобщающих параметрических процедур можно создать дополнительные модули. Их количество определяется стратегией расширения. Например, для добавления специализации вывода обобщенной фигуры создадим модуль ***MCircOut***.

```
MODULE MCircOut (MFig);
  IMPORT In, Out, MCirc;
  TYPE
    // Расширение добавлением круга
    Figure += MCirc.Circle;
    // Вывод обобщенного круга
  PROCEDURE Output
    {VAR c: Figure(MCirc.Circle)};
  BEGIN MCircle.Output(c) END Output;
END MCircOut.
```

Проверку вложенности друг в друга различных фигур, учитывающую добавленный круг, можно расширить подключением модуля ***MRTCMM*** к модулю ***MRTMM***. Он также находится в пространстве имен модуля ***MFig***.

```
MODULE MRCTMM (MRTMM);
  IMPORT In, Out, MRect, MTrian, MCirc;
  TYPE
    // Расширение добавлением круга
    Figure += MCirc.Circle;
  // Обработчики специализаций
  // прямоугольник внутри круга
  PROCEDURE FirstInSecond
    {VAR f1(MRect.Rectangle),
       f2(MCirc.Circle): Figure}: BOOLEAN;
  BEGIN
    Out.String
        ("Rectangle in Circle compare");
    Out.Ln;
    RETURN ((f1.x*f1.x + f1.x*f1.x) <
            (f2.r*f2.r))
  END FirstInSecond;
  // треугольник разместится внутри круга
  PROCEDURE FirstInSecond
    {VAR f1(MTrian.Triangle),
         f2(MCirc.Circle): Figure}: BOOLEAN;
  BEGIN ... END FirstInSecond;
  // круг разместится внутри прямоугольника
  PROCEDURE FirstInSecond
    {VAR f1(MCirc.Circle),
         f2(MRect.Rectangle): Figure}:
BOOLEAN;
  BEGIN ... END FirstInSecond;
  // круг разместится внутри треугольника
  PROCEDURE FirstInSecond
    {VAR f1(MCirc.Circle),
         f2(MTrian.Triangle): Figure}:
                                  BOOLEAN;
  BEGIN ... END FirstInSecond;
  // круг разместится внутри круга
  PROCEDURE FirstInSecond {VAR
    f1, f2: Figure(MCirc.Circle)}:
                                  BOOLEAN;
  BEGIN
    Out.String("Circle in Circle compare");
    Out.Ln;
    RETURN f1.r < f2.r
  END FirstInSecond;
END MRTCMM.
```

Наряду с уменьшением связей внутри программы за счет их переноса во внешнее описание проекта увеличилась наглядность. Помимо этого, обработчики специализаций стали скрыты от общего использования, что повысило их защищенность.

4. РЕАЛИЗАЦИЯ ТИПИЧНЫХ СИТУАЦИЙ ЭВОЛЮЦИОННОГО РАСШИРЕНИЯ КОДА

Можно выделить ряд типичных ситуаций расширения кода, часто встречающихся на практике [10]:

- добавление в обобщения новых специализаций и, как следствие, расширение обрабатывающих их обобщающих процедур;
- добавление новых процедур, обеспечивающих дополнительную функциональность;
- добавление новых полей данных в существующие типы и изменение в соответствии с этим процедур, осуществляющих обработку измененных программных объектов;
- добавление новой процедуры, предназначенной для обработки только одной из специализаций некоторого обобщения;
- создание нового обобщения на основе существующих специализаций;
- добавление в программу мультиметодов, осуществляющих обработку двух или более обобщенных параметров;
- изменение мультиметодов при добавлении в обобщения, используемые в качестве аргументов мультиметодов, новых специализаций.

Эти варианты могут встречаться как раздельно, так и совместно. В последнем случае порождаются разнообразные комбинации, реализация которых требует нетривиальных алгоритмических решений. Сложность и гибкость этих решений зависит от используемой парадигмы программирования, которая в ряде случаев позволяет реализовать требуемую ситуацию с применением прямого использования предоставляемых ею инструментальных средств.

4.1 Добавление в обобщение новых специализаций

Процедурный подход. Обычно используются объединения (в языке C) или вариантные записи (языки Pascal, Modula-2), что не обеспечивает эволюционного расширения и требует

модификации уже написанного кода. Безболезненного расширения данных можно достичь использованием указателя на произвольный тип, что снижает безопасность программного кода. Использование расширений типов (Oberon, Oberon-2, Component Pascal, Ada) или наследования (C++) позволяет безболезненно добавить специализации. Обработчики специализаций обычно реализуются путем изменения условного оператора или эквивалентного ему оператора выбора (переключателя), что *не способствует эволюционному расширению написанного кода*.

ОО подход. В Oberon-2, Component Pascal и Ada применяется расширение типа совместно с процедурами, связанными с типом. В C++, C#, Java и других ОО языках создается производный класс с виртуальными методами, расширяющими соответствующие виртуальные методы базового класса. То есть, ООП поддерживает безболезненное расширение в данной ситуации.

ПП подход. Обеспечивает добавление новых специализаций через подключение специализации к обобщению за счет существующей в языке внешней операции внешнего добавления. Необходимые обработчики специализаций также легко расширяют уже существующие обобщающие процедуры за счет дописывания новых реализаций.

4.2 Добавление новых процедур

Процедурный и ПП подходы. Использование внешних процедур обеспечивает простое решение данной задачи. Подобный подход успешно применим практически во всех языках процедурного программирования.

ОО подход. Метод, добавляющий функциональность, приходится вставлять внутрь класса, что ведет к изменению последнего. Таким образом, ОО парадигма напрямую не поддерживает эволюционного добавления новых методов. Следует отметить что в современных языках программирования (C#) начинает использоваться частичное описание класса, что, решая в целом данную проблему, ухудшает восприятие программы на уровне кода и требует постоянной перекомпиляции на уровне исходных текстов. Также следует отметить, что подобную инструментальную поддержку можно использовать практически с любыми парадигмами программирования.

4.3 Добавление новых полей в существующие типы

Прямое добавление новых полей в запись или класс изменяет его независимо от стиля программирования. Поэтому, при необходимости модификации полей типа, используются обходные пути, опирающиеся на косвенное связывание через указатель на базовый тип, к которому подключается объект производного типа, содержащий дополнительные поля. При обработке такого объекта необходимо использовать явное приведение типа или дополнительные функции. Если эти операции приходится применять в ранее написанных модулях, то эволюционное расширение программы невозможно. Однако часто обработка модифицированного типа осуществляется только в добавленных единицах компиляции. Поэтому данный подход может применяться для всех рассматриваемых парадигм программирования.

Процедурный подход. В языке программирования Oberon удобно пользоваться расширением типов. В этом случае новый тип расширяет предшественника и через указатель на базовый тип может использоваться по целевому назначению. Реализованный в языке механизм проверки типа во время выполнения позволяет легко перейти от базового типа к производному. Вместе с тем, следует отметить, что традиционные процедурные языки (C, Pascal) не обеспечивают реализацию подобного приема из-за отсутствия дополнительной инструментальной поддержки проверки типа. Поэтому их использование в таком режиме невозможно без написания дополнительного кода.

ОО подход. Добавление новых полей, как и при расширении типов, осуществляется в производном классе. При использовании языков, поддерживающих динамическую идентификацию типа во время выполнения (C#, Java, Object Pascal), использование классов аналогично применению расширений типов, описанному для процедурных языков. В этом случае, наряду с данными можно включать и новые методы, осуществляющие их обработку. Следует однако отметить, что использование явной проверки типа во время выполнения в большей степени является процедурным, чем объектно-ориентированным приемом.

При отсутствии динамической проверки типа или отключения этого режима (например, в C++) использование косвенного добавления полей в класс становится невозможным. Для его реализации необходимы дополнительные алгоритмические приемы, например, включение в базовый класс виртуального метода, переопределяемого в производных классах, за счет чего обеспечивается требуемая идентификация. В данной ситуации использование объектно-ориентированного полиморфизма невозможно без дополнительной алгоритмической поддержки.

ПП подход. Использование косвенного добавления полей допускается при использовании обобщенных записей. В этом случае расширение заключается в добавлении специализации. Доступ к специализации поддерживается механизмом проверки признака специализации, по которому можно установить ее тип.

4.4 Добавление процедур для одной из специализаций

Ситуация возникает, когда необходимо выделить и обработать одну из специализаций. Например, при обработке элементов контейнера, ссылающегося на обобщенный тип, через который нужно обеспечить доступ только к одному из специализированных типов.

Процедурный подход. В случае расширений типа используются процедуры, в которых осуществляется явная проверка производного типа по указателю на базовый тип [11]. Метод легко добавляется в новом модуле. Не вызывает проблем использование этого же приема в процедурных языках, не имеющих поддержки полиморфизма типа (C, Pascal). В этом случае обобщение обычно строится таким образом, что имеет признак, который и используется при идентификации специализации.

ОО подход. Используются виртуальные методы, осуществляющие обработку специализированного объекта требуемого типа [11]. Однако такой метод должен добавляться в базовый класс, что не способствует

эволюционному расширению программы. Помимо этого, базовый класс перегружается дополнительной информацией о производном классе, что не способствует инкапсуляции. Поэтому чаще всего применяется процедурный прием, основанный на явной проверке типа.

ПП подход. Простейшим вариантом является использование процедурного подхода, примененного к обобщенному типу [4]. Отличие заключается в проверке признака специализации вместо проверки производного типа подключенного объекта. Помимо этого, возможно использование процедурно-параметрического полиморфизма, при котором отсутствует необходимость явной манипуляции типом объекта. В начале пишется обобщающая процедура с пустым телом (как и при ОО подходе но вне структуры данных). Для выполнения манипуляций с заданной специализацией пишется ее обработчик, который и вызывается при вызове обобщающей процедурой. В отличие от ОО подхода все добавления осуществляются в новых модулях, что обеспечивает эволюционное расширение. То есть, ПП парадигма предоставляет разнообразные способы для разрешения данной ситуации**.**

4.5 Создание обобщений из существующих специализаций

Процедурный подход. Зачастую в программе требуется сформировать новое обобщение при уже существующих специализациях. Процедурный подход в этом случае обеспечивает прямое решение на основе объединения (язык C) или вариантной записи (Pascal, Modula-2). Обобщение может формироваться в модулях, добавляемых в программу, что не вызывает проблем с эволюционным расширением. Вместе с тем, следует отметить, что языки, использующие для построения обобщений только расширение типа (Oberon, Oberon-2), не обеспечивают прямого решения этой задачи. Необходимы дополнительные построения.

ОО подход. Использование наследования обладает теми же недостатками, что и расширение типа. Поэтому новое обобщение обычно создается на основе нового базового класса и порождения от него производных классов, включающих в себя необходимые существующие специализации. Решение простое, но не является прямым.

ПП подход. Обобщения и обобщенные записи при своем создании допускают непосредственное включение в них специализаций, что обеспечивает прямое построение. Помимо этого, расширение за счет уже существующих специализаций возможно и при уже существующем обобщении. Это обеспечивает высокую гибкость и эволюционный рост для рассматриваемой ситуации.

4.6 Добавление мультиметодов

Процедурный подход. Так как мультиметод является обычной процедурой, его добавление не вызывает никаких проблем.

ОО подход. Реализация мультиметода в объектно-ориентированном подходе опирается на множественную диспетчеризацию, при которой между классами, играющими роль обобщенных аргументов мультиметода, возникают тесные взаимодействия, задаваемые через методы. Добавление мультиметода ведет к изменению класса, что не способствует эволюционному расширению.

ПП подход. Для гибкой реализации мультиметодов используются обобщающие параметрические процедуры. Являясь по сути внешними процедурами, они обеспечивают безболезненное расширение мультиметодов.

4.7 Изменение мультиметодов при добавлении специализаций

Процедурный подход. Используется явная проверка типов обобщенных аргументов мультиметода внутри условных операторов или операторов выбора. Поэтому добавление новой специализации изменяет один или несколько условных операторов, не обеспечивая эволюционного расширения в данной ситуации [11].

ОО подход. Добавление специализации связано с созданием нового производного класса [11]. Включение этого класса в общую группу, содержащую мультиметод, ведет, для обеспечения диспетчеризации, к добавлению дополнительных методов во все классы группы. Поэтому прямое добавление новых специализаций не поддерживается.

ПП подход. Парадигма изначально разрабатывалась для поддержки безболезненного расширения мультиметодов при добавлении новых специализаций [4] за счет описания всех комбинаций аргументов в обработчиках специализаций, размещаемых независимо от обобщающей процедуры. Метод основан на достаточно простом механизме построения параметрических отношений, одна из возможных реализаций которого приведена в [2].

4.8 Общая характеристика подходов

В таблице 1 собраны сведения, показывающие возможности каждой из рассмотренных парадигм по прямой поддержке эволюционного расширения.

Таблица 1. Возможности прямого эволюционного расширения кода в различных ситуациях

Ситуация	Подходы		
	Процедурный	***ОО***	***ПП***
1. Добавление специализации и ее обработчиков	нет	есть	есть
2. Добавление новых процедур	есть	нет	есть
3. Добавление новых полей в существующий тип	косвенное, для расширяемых типов	косвенное, при наличии RTTI	косвенное, при использовании обобщенной записи
4. Добавление обработчика для одной специализации	есть	нет	есть процедурный и параметрический
5. Создание обобщения на основе существующих специализаций	есть	косвенное	есть
6. Добавление в программу мультиметода	есть	нет	есть
7. Изменение мультиметода при добавлении специализаций	нет	нет	есть

Видно, что ОО подход обеспечивает прямое эволюционное расширение программы в наименьшем числе из рассмотренных ситуаций. Во многом это связано с тем, что методы располагаются внутри классов. Поэтому, в случае ситуации, связанной с изменением функциональности, приходится каждый раз модифицировать один или несколько классов. Для преодоления этого в языке программирования C# используется распределенное описание элементов класса в различных файлах, что однако не способствует целостности восприятия данной конструкции.

Вместе с тем, текущая популярность ОО парадигмы вполне объяснима. Во-первых, она обеспечивает добавление новых специализаций (ситуация 1). То есть, в той ситуации, считающейся одной из типичных. Во-вторых, в большинстве других случаев (ситуации 3, 4, 5, 6) на практике, даже при использовании ОО языков, применяются методы процедурного и мультипарадигменного программирования. Ситуация 2 зачастую сглаживается тщательной проработкой интерфейса на этапе проектирования. Помимо этого во многих случаях легко можно пойти на небольшие изменения классов (ситуации 2, 4, 6), так как подобная модификация больше касается процесса компиляции, а не изменения модульной структуры программы. Можно пойти и на алгоритмические ухищрения (ситуации 3, 4, 7), которые не приводят к большому увеличению кода, но сохраняют требуемую гибкость, описанную в ситуации 1.

5. ЗАКЛЮЧЕНИЕ

Применение параметрических обобщений в сочетании с обобщающими параметрическими процедурами ведет к более гибкому созданию эволюционно расширяемых программ. Трудоемкость построения обобщенных записей соизмерима с созданием расширяемых записей языка программирования Oberon или иерархий классов объектно-ориентированных языков. Использование подключаемых модулей облегчает контроль процесса наращивания программы. Разработка языка программирования Alien позволяет провести исследование возможностей процедурно-параметрической парадигмы и наметить дальнейшие пути ее развития. В частности, инструментальная поддержка эволюционного расширения мультиметодов позволяет гибко и без дополнительных алгоритмических затрат осуществлять прямое написание кода и для тех ситуаций, когда мультиметоды вырождаются в более простые конструкции. Многие из таких конструкций в настоящее время реализованы с применением паттернов проектирования. ПП парадигма в целом позволяет многие из паттернов описывать напрямую, то есть, без задания сложных отношений между классами и без динамического их связывания во время выполнения. Это повышает эффективность написанного кода и снижает его размер.

Работа поддержана грантом РФФИ № 13-01-00360 «Методы и средства эволюционной разработки программного обеспечения с применением процедурно-параметрической парадигмы программирования».

6. ЛИТЕРАТУРА

[1] Легалов А. И. ООП, мультиметоды и пирамидальная эволюция // Открытые системы. 2002. № 3. С. 41–45.

[2] Легалов А. И. Мультиметоды и парадигмы // Открытые системы. 2002. No 5. С. 33–37.

[3] Мейерс С. Наиболее эффективное использование C++. 35 новых рекомендаций по улучшению ваших программ и проектов: Пер. с англ. М.: ДМК Пресс, 2000. 304 с.

[4] Легалов А. И. Процедурно-параметрическая парадигма программирования. Возможна ли альтернатива объектно-ориентированному стилю? Красноярск, 2000. Рук. деп. в ВИНИТИ 13.03.2000. № 622-В00. 43 с.

[5] Легалов А. И., Швец Д. А. Язык программирования O2M. URL: http://www.softcraft.ru/ppp/o2m/o2mref.shtml

[6] Легалов И. А. Применение обобщенных записей в процедурно-параметрическом языке программирования // Науч. вестн. НГТУ. 2007. № 3 (28). С. 25–38.

[7] Легалов А. И., Бовкун А. Я., Легалов И. А. Расширение модульной структуры программы за счет подключаемых модулей / Докл. АН ВШ РФ. 2010. № 1 (14). С. 114–125.

[8] Свердлов С. З. Языки программирования и методы трансляции: Учеб. пособие. Спб.: Питер, 2007. 638 с.

[9] Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования: Пер. с англ. СПб.: Питер, 2001. 368 с.

[10] Легалов А.И., Легалов И.А., Солоха А.Ф. Эволюционное расширение программ при различных парадигмах программирования. - Труды XVI Байкальской Всероссийской конференции «Информационные и математические технологии в науке и управлении». Часть III. - Иркутск: ИСЭМ СО РАН, 2011. ISBN 978-5-93908-094-1. - С. 42-49.

[11] Легалов, А.И. Разнорукое программирование. URL: http://www.softcraft.ru/paradigm/dhp/index.shtml