УДК 004.432

РАСШИРЕНИЕ МОДУЛЬНОЙ СТРУКТУРЫ ПРОГРАММЫ ЗА СЧЕТ ПОДКЛЮЧАЕМЫХ МОДУЛЕЙ

А. И. Легалов,
д.т.н., профессор кафедры Вычислительной техники Сибирского федерального университета, г. Красноярск (legalov@mail.ru)
А. Я. Бовкун,
аспирант кафедры Вычислительной техники Сибирского федерального университета, г. Красноярск (bovkunalex@mail.ru)
И. А. Легалов,
к.т.н., доцент кафедры Информационных систем Сибирского федерального университета, г. Красноярск (igor@legalov.ru)

Исследуются подходы к организации модульной структуры и ее связи с эволюционной разработкой программного обеспечения. Для поддержки гибкого расширения программных объектов предлагается применение подключаемых модулей. Рассматриваются особенности использования подключаемых модулей в сочетании с процедурно-параметрической парадигмой программирования.

Ключевые слова: эволюционная разработка программ, модуль, процедурно-параметрическое программирование.

1. Введение

Эволюционная разработка больших программных систем приобретает все большую популярность. Это обусловлено различными факторами, связанными с развитием информационных технологий:

- современные методологии разработки программного обеспечения (ПО) ориентированы на инкрементальное наращивание кода;
- существующие системы программирования содержат средства, обеспечивающие поддержку эволюционного проектирования;
- эволюционное расширение программных систем экономически более выгодно, чем использование методов, ориентированных на постоянную модификацию уже написанного кода;
- использование эволюционного расширения программ уменьшает количество ошибок, вносимых в написанный и уже отлаженный код, который постоянно приходится модифицировать при использовании традиционных методов разработки программного обеспечения.

Существуют различные методы организации кода, обеспечивающие поддержку эволюционного расширения и базирующиеся на использовании таких абстракций данных как классы [1], расширяемые типы данных [2], обобщенные записи [3]. Их применение в совокупности с динамическим связыванием, использующим наследование и полиморфизм, позволяет создавать сложные, эволюционно расширяемые структуры [4]. Вместе с тем, поддержка эволюционной разработки также

обеспечивается и модульной структурой, широко применяемой при создании больших программных систем.

Модуль представляет собой функционально законченный фрагмент программы, оформленный в виде отдельного файла с исходным кодом или поименованной непрерывной его части [5], предназначенной для использования в других программах. Существуют разные подходы к организации модульной структуры программы. В ряде случаев используется жесткая концепция модульного программирования [6], нашедшая отражение в таких языках как Оберон-2 [7], Component Pascal [8], Python [9]. Во многих языках программирования модульная структура выстраивается произвольно, что объясняется видением соответствующих конструкций в качестве ограничителей пространств имен и единиц компиляции, а не средств для объединения программных объектов в функционально законченные компоненты. К таким языкам относятся C++ [10], C# [11]. В ряде языков логические модули отдельно не выделяются. Их роль выполняют физические единицы компиляции, как в языке C, или языковые конструкции, например, классы в Java [12].

Наибольшей гибкостью из рассмотренных вариантов обладает подход, базирующийся на пространстве имен и позволяющий произвольно накладывать логическую структуру модулей на физические единицы. Он широко применяется при эволюционной разработке программного обеспечения на языках программирования C++ и C#, так как позволяет легко расширять различные пространства имен во вновь добавляемых единицах компиляции.

Вместе с тем, модульное программирование, опирающееся на концепцию логического модуля, ограниченного физической единицей компиляции, не теряет своей популярности. Оно широко используется при программировании не только на компилируемых, но и интерпретируемых языках, например, Python [9]. Популярность обуславливается простотой формирования программы из законченных логических единиц, для объединения которых не требуется дополнительных компоновщиков. Это позволяет легко реализовать как статическую, так и динамическую сборку программы, а также повторно использовать модули при создании новых программ.

Однако у такого подхода имеются и недостатки, связанные с невозможностью безболезненного наращивания. При расширении функциональности зачастую приходится изменять уже написанные модули. Существует ряд методов, направленных на решение этой проблемы. В частности, в работе [13] предложено использовать механизм встраивания. Он обеспечивает создание новых модулей на основе наследования с переопределением имеющихся процедур и типов. Встраивание полезно, когда модули и классы используются в комбинации и необходимо их совместное расширение, или когда модули являются более подходящей средой, чем классы. Подход обеспечивает модификацию процедур и данных встраиваемого модуля, что позволяет эволюционно наращивать и изменять код. Однако данный прием требует

указания встраиваемого модуля в клиентской части программы, что ведет к модификации последней и не способствует полной поддержке эволюционного расширения.

Попытки организовать эволюционную разработку с сохранением традиционной для модульного программирования структурой программы представлены в работах [3, 14]. Подход базируется на использовании процедурно-параметрической парадигмы программирования, позволяющей добавлять в новых модулях дополнительные специализации обобщений, обобщенные записи и обобщающие параметрические процедуры. Несмотря на гибкость предлагаемого решения, отмечено, что использование модульной структуры программы ведет к созданию множества мелких вспомогательных модулей со своими пространствами имен, что изменяет целостное восприятие окончательного кода [14]. Помимо того, что универсальное размещение процедурно-параметрических конструкций в произвольных модулях затрудняет их идентификацию и ведет к дроблению программы на слишком мелкие модули, снижается надежность программирования за счет неконтролируемого использования ссылок на обобщающий тип. Стремление повысить надежность эволюционной разработки при использовании модульного программирования ведет к необходимости поиска более гибких решений.

2. Подключаемые модули

Для обеспечения эволюционного расширения программных объектов с сохранением существующих интерфейсов между модулями предлагается использовать подход, базирующийся на дополнительных модулях-расширителях, подключаемых к основному модулю. Идея подхода схожа с использованием механизма наследования вместо прямого включения типов во вновь формируемый программный объект. При наследовании создается описание нового типа, расширяющего базовый тип дополнительными данными, а принцип подстановки обеспечивает переопределение методов производного класса, которые могут обрабатывать эти дополнительные данные.

Использование подключаемых модулей отличается тем, что вместо множества экземпляров базового и производного классов в программе существуют только по одному экземпляру разных модулей. Разработанное расширение модуля может подключаться к уже существующему базовому модулю, образуя вместе с ним общее пространство имен, тогда как при традиционном импорте внутренние пространства имен модулей не пересекаются. Подключаемый модуль может импортироваться из других модулей, обеспечивая им передачу как своего интерфейса, так и интерфейса базового модуля. Однако подобное использование не эффективным, так как не обеспечивает неизменность клиента, изначально ориентированного на импорт базового модуля. Главной особенностью подключаемого модуля является возможность расширения обобщенных

записей и обобщающих параметрических процедур, используемых в процедурно-параметрическом программировании. Она достигается за счет описания в нем новых специализаций и их обработчиков.

Отличие подключаемого модуля от обычного проявляется в указании имени модуля-родителя (базового модуля) и прав доступа к интерфейсу родительского модуля из модулей, импортирующих подключаемый модуль (только для чтения или с возможностью расширения). Практически это позволяет организовать прямые взаимодействия, отличающиеся от существующих языковых структур, используемых в языках Оберон-2, O2M [14]. В сочетании с процедурно-параметрической парадигмой программирования [15] это облегчает контроль процесса наращивания программы. Добавление новых данных и процедур осуществляется прозрачно для клиентского кода, использующего параметрические обобщения, обобщенные записи и обработчики параметрических процедур.

3. Особенности использования подключаемых модулей

Подключаемые модули реализованы в языке программирования, являющемся модификацией языка O2M [14]. В качестве примера, раскрывающего особенности использования подключаемых модулей, можно рассмотреть новую реализацию программы, представленной в [14]. Пусть на начальном этапе необходимо создать только структуры, описывающие прямоугольник и треугольник, реализовать которые можно в двух традиционных модулях **MRect** и **MTrian**.

```
// Модуль, описывающий прямоугольник
MODULE MRect;
 IMPORT In, Out;
 TYPE
   PRectangle* = POINTER TO Rectangle;
   Rectangle* = RECORD
     x*, y* : INTEGER //стороны прямоугольника
   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
      // стороны треугольника
      a*, b*, c* : INTEGER
    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 также в обычном 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 геометрическую фигуру
   PFigure* = POINTER TO Figure;
   //Обобщение геометрической фигуры
   Figure* = CASE TYPE OF
     MRect.Rectangle | MTrian.Triangle
   END;
 //Обобщенная параметрическая 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 функциональные возможности приложения, происходит без изменения уже написанных модулей. При этом можно создавать как обычные, так и любые обобщающие процедуры, в том числе и мультиметоды. Рассмотрим, каким образом осуществляется добавление мультиметода, проверяющего возможность размещения первой фигуры внутри второй. Для этого создадим подключаемый модуль **MRTMM**, в котором опишем обобщающую процедуру и ее обработчики. Зададим возможность полного

экспорта данным модулем не только своего интерфейса, но и интерфейса родительского модуля. Это позволяет при импорте дочернего модуля получать интерфейсы всех видимых родительских модулей без дополнительных подключений.

```
MODULE MRTMM (MFig*);
 IMPORT In, Out, MRect, MTrian;
 //Чистая обобщающая процедура, задающая интерфейс
 PROCEDURE FirstInSecond* {VAR f1,f2: Figure}
                                          :BOOLEAN := 0;
 //Обработчики специализаций
 //прямоугольник разместится внутри прямоугольника
 PROCEDURE FirstInSecond
     {VAR f1,f2: Figure(MRect.Rectangle)}: BOOLEAN;
 BEGIN
   Out.String("Rectangle in 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 разместится внутри треугольника
 PROCEDURE FirstInSecond {VAR f1(MRect.Rectangle),
                f2(MTrian.Triangle): Figure}: BOOLEAN;
   BEGIN ... END FirstInSecond;
 //треугольник разместится внутри прямоугольника
 PROCEDURE FirstInSecond {VAR f1(MTrian.Triangle),
            f2(MRect.Rectangle): MFig.Figure}: BOOLEAN;
   BEGIN ... END FirstInSecond;
 //треугольник разместится внутри треугольника
 PROCEDURE FirstInSecond
        {VAR f1, f2: MFig.Figure(MTrian.Triangle)}:
BOOLEAN;
 BEGIN ... END FirstInSecond;
END MRTMM.
```

То, что этот модуль является подключаемым, определяется дополнительным параметром **MFig** в заголовке модуля **MRTMM**, следующем в скобах за именем модуля:

```
MODULE MRTMM (MFig*);
```

Звездочка (в стиле языка Оберон-2) указывает, что интерфейс родительского модуля будет виден и может экспортироваться из подключаемого модуля. Наличие общего пространства имен с родительским модулем позволяет напрямую обращаться к обобщению. Традиционный импорт модулей **MRect** и **MTrian** ведет к обращению с использованием префиксов модулей.

Если клиентский модуль **MClient** работает только с обобщенной фигурой, то он может напрямую импортировать модуль **MRTMM**, через который обеспечивается его обращение к интерфейсу модуля **MFig.**

Поэтому клиентского модуля позволяет осуществить сборку всех описанных модулей.

```
// Клиентский модуль, обрабатывающий обобщения
MODULE MClient;
 IMPORT MRTMM, Console;
 VAR
   bool: BOOLEAN; ...
   // Указатели на обобщенные фигуры формируемые
   // вне клиентского модуля
   r, t, c: MFig.PFigure;
BEGIN // Собственно обработка
   ...
  // Использование обобщенного вывода
  MFig.Output{r};
  MFig.Output{t};
  MFig.Output{c};
  // Использование мультиметода
  bool := MRTMM.FirstInSecond{r, r};
  bool := MRTMM.FirstInSecond{r, t};
  bool := MRTMM.FirstInSecond{r, c};
  bool := MRTMM.FirstInSecond{t, r};
  bool := MRTMM.FirstInSecond{t, t};
  bool := MRTMM.FirstInSecond{t, c};
  bool := MRTMM.FirstInSecond{c, r};
  bool := MRTMM.FirstInSecond{c, t};
  bool := MRTMM.FirstInSecond{c, c};
END MClient.
```

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**. Запретим в нем передачу интерфейса родителя другим модулям и закроем доступ к его обработчикам специализаций (отсутствие «*****» у имен **MFig** и **Output**).

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

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

```
MODULE MRTCMM (MRTMM);
 IMPORT In, Out, MRect, MTrian, MCirc;
 TYPE
   // Расширение обобщения добавлением круга
   Figure += MCirc.Circle;
   // Дополнительные обработчики специализаций
   // прямоугольник разместится 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.
```

Следует отметить неизменность клиентского модуля при добавлении новых модулей в программу. Схема зависимостей модулей программы для приведенного варианта представлена на рис. 1.

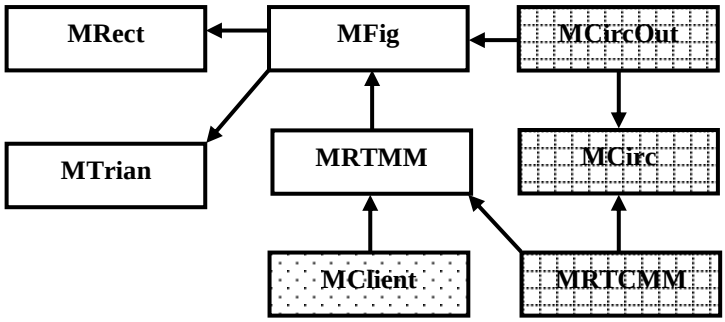

Рис. 1. Зависимости между модулями в программе обработки фигур

Наряду с уменьшением количества связей между модулями внутри программы, полученным за счет возможности использования подключаемых модулей для экспорта интерфейсов родительских модулей, увеличилась и наглядность кода, что обеспечено наличием единого пространства имен. Помимо этого, обработчики специализаций стали скрыты от общего использования, что повысило их защищенность.

4. Реализация подключаемых модулей

Использование дополнительных модулей-расширителей изменяет внутреннюю организацию процедурно-параметрической программы. Родительские и подключаемые модули порождают в ходе компиляции несколько единиц объектного кода:

- основную единицу, содержащую реализацию процедур и память, отводимую под переменные модуля;
- интерфейсную единицу, обеспечивающую описание внешних ссылок на программные объекты, экспортируемые в другие модули;
- параметрическую единицу, обеспечивающую связь обобщающих процедур родительского модуля с обработчиками специализаций, размещенных как в самом родительском модуле, так и в подключаемых модулях.

Помимо этого, в зависимости от режима компиляции, подключаемый модуль дополнительно может порождать дочернюю единицу, содержащую интерфейс для подключения расположенных в нем специализаций и их обработчиков к параметрической единице того

родительского модуля, который содержит соответствующие обобщения и обобщающие процедуры. Общая структура единиц кода, порождаемых при компиляции, приведена на рис. 2.

Рис. 2. Состав и структура единиц объектного кода, порождаемых при компиляции модуля

Основная единица объектного кода по своей внутренней организации практически не отличается от единиц, порождаемых модулями других языков программирования. Ее адресное пространство распределено между областями данных и кода. В области данных находятся переменные, описанные в модуле. В области кода размещаются все процедуры модуля. Обобщенные параметрические процедуры содержат код, обеспечивающий обращение к таблицам параметрической единицы, в которых во время компоновки происходит формирование окончательных связей с обработчиками специализаций. В этой же единице размещается код обработчиков специализаций, связанный с обобщенными процедурами как своего модуля, так и с обобщенными процедурами вышестоящих родительских модулей.

Интерфейсная единица объектного кода содержит ссылки на экспортируемые программные объекты модуля, размещенные в основной единице. Это могут быть адреса переменных, обычных процедур, обобщающих параметрических процедур. Помимо этого в ней описываются имена и видимые поля экспортируемых записей и обобщений.

Параметрическая единица объектного кода определяется спецификой процедурно-параметрического программирования. Она используется для обеспечения процесса раздельной компиляции модулей и содержит параметрические таблицы, обеспечивающие подключение обработчиков специализаций к обобщенной параметрической процедуре. При компиляции родительского модуля эти таблицы могут быть пустыми или заполняться только ссылками на обработчики специализаций, размещенные в самом модуле. Окончательное заполнение таблиц осуществляется во время компоновки, использующей информацию из дочерних единиц подключаемых модулей.

Дочерняя единица объектного кода порождается для подключаемых модулей. В ней содержится ссылочная информация об обработчиках специализаций, которая предназначена для использования во всех родительских модулях. То есть, в общем случае подключаемый модуль может создавать обработчики специализаций не только для своего родительского модуля, но и для модуля, который является родителем его родителя. Поддержка иерархии родителей обеспечивает гибкое расширение программы, не вынуждая программиста создавать подключаемые модули, непосредственно связанные с родительскими. Это хорошо согласуется с использованием общего пространства имен по всей цепочке подключения и позволяет использовать переменные, появившиеся при более поздних подключениях в обработчиках специализаций обобщенных параметрических процедур, созданных в вышестоящих родительских модулях.

Обработка информации, собранной в дочернем модуле, осуществляется в несколько шагов. Вначале компоновщик обеспечивает связывание дочерней единицы с параметрической единицей непосредственного родителя. Далее, при наличии связей с модулями, расположенными выше в иерархии подключений, осуществляется их связь по этой цепочке.

5. Связывание модулей в единый проект

Использование импорта обеспечивает прямое соединение иерархии модулей, что является достаточным для простых проектов. Однако при эволюционном расширении необходима дополнительная информация о подключаемых модулях, которые непосредственно не импортируются модулями основной иерархии. Для организации дополнительных связей используется проект, который определяет состав модулей разрабатываемой программы. Такое решение позволяет обойтись без модификации программы, затрагивая в случае изменений только конфигурационный файл проекта.

Заключение

Использование подключаемых модулей обеспечивает создание более понятной общей структуры программы за счет концентрации расширений процедурно-параметрических конструкций в подключаемых модулях. В

дальнейшем, если подключаемые модули окажутся небольшими по размер, а также если они не будут удаляться из программы, можно осуществить их автоматическое слияние с родительским модулем, используя переупаковку.

Добавление новых процедур, расширяющих функциональные возможности приложения, происходит без изменения уже написанных модулей. При этом можно создавать как обычные, так и любые обобщающие процедуры, в том числе и мультиметоды. В этом случае специализации, тип которых известен во время компиляции, обрабатываются так же, как и обычные переменные эквивалентного типа.

Для расширения уже существующего обобщенного программного объекта и обобщающих параметрических процедур можно создать дополнительные модули. Их количество определяется стратегией расширения.

К дополнительным положительным чертам предложенного механизма подключения модулей можно отнести возможность его использования в любых других модульных языках. Обеспечивая общее пространство имен с родительскими модулями, подключаемые модули позволяют надежно добавлять данные, процедуры и классы, обеспечивающие поддержку полиморфизма.

К недостаткам подхода следует отнести использование дополнительной инструментальной поддержки во время связывания и компоновки модулей в окончательную программу. Однако этот недостаток присущ практически всем современным системам программирования. В частности, при объектно-ориентированном подходе [1] изменение процесса компоновки направлено на идентификацию корректного использования классов, а при аспектно-ориентированном программировании [16] компоновщик должен анализировать правильность использования обертываемых методов.

ЛИТЕРАТУРА

[1] **Бадд Т.** Объектно-ориентированное программирование в действии. / Т. Бадд – СПб.: Питер. – 1997. – 464 с.

[2] **Вирт Н.** Алгоритмы и структуры данных. Новая версия для Оберона. Пер. с англ. / Н. Вирт – М.: ДМК-Пресс, 2010. – 272 с.

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

[4] **Гамма Э.** Приемы объектно-ориентированного проектирования. Паттерны проектирования. Пер. с англ. / Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес – СПб: Питер, 2007. – 366 с.

[5] **Модуль (программирование). Материал из Википедии – свободной энциклопедии** [Электронный ресурс] – Режим доступа: http://ru.wikipedia.org/wiki/Модуль_(программирование). – Загл. с экрана.

[6] **Grogono P.** Why One Source File Is Better Than Two. / P. Grogono // SEA 2000 – Software Engineering and Applications (Las Vegas, Nevada, USA, November 2000), (Proceedings of the LASTED International Conference). – ACTA Press. – 2000. – pp. 243 – 249.

[7] **Moessenboeck H.** The Programming Language Oberon-2. [Электронный ресурс] / H. Moessenboeck, N. Wirth // Institut fur Computersysteme, ETH Zurich July. – 1996. – Режим

доступа:
http://www-vs.informatik.uni-ulm.de:81/projekte/Oberon-2.Report/Oberon2-Report.ps. – Загл. с экрана.

[8] **BlackBox.** [Электронный ресурс] – Режим доступа:
http://www.oberon.ch/blackbox.html. – Загл. с экрана.

[9] **Саммерфильд М.** Программирование на Python 3. Подробное руководство. Пер. с англ. / М. Саммерфильд – Спб.: Символ-Плюс. – 2009. 608 с.

[10] **Страуструп Б.** Язык программирования C++. Третье издание. Пер. с англ. / Б. Страуструп – СПб.; М.: "Невский диалект" – "Издательство БИНОМ", 1999. – 991 с.

[11] **Троелсен Э. С.** C# и платформа .NET. Библиотека программиста. Пер. с англ. / Э.С. Троелсен – СПб.: Питер, 2003. – 800 с.

[12] **Нортон П.** Программирование на Java. Руководство П.Нортона (в 2-х томах). Пер. с англ. / П. Нортон – "СК-Пресс", 1998 – 900 с.

[13] **Radensky A.** Module embedding. [Электронный ресурс] / A. Radensky – Режим доступа: http://www1.chapman.edu/~radenski/research/papers/module.pdf. – Загл. с экрана.

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

[15] **Легалов А.И.** Процедурно-параметрическая парадигма программирования. Возможна ли альтернатива объектно-ориентированному стилю? / А.И. Легалов – Красноярск: 2000. Деп. рук. № 622-В00 Деп. в ВИНИТИ 13.03.2000. – 43 с.

[16] **Павлов В.** Аспектно-ориентированное программирование [Электронный ресурс] // В. Павлов – Режим доступа: http://www.javable.com/columns/aop/workshop/01/. – Загл. с экрана.

Legalov A.I., Bovkun A.J., Legalov I.A.

EXTENSION OF PROGRAM'S MODULAR STRUCTURE AT THE EXPENSE OF ATTACHABLE MODULES

Approaches to the modular structure and its relation to the evolutionary development of the software are investigated. The use of attachable modules to support a flexible expansion of program objects is proposed. Features of attachable modules using in conjunction with the procedure-parametric paradigm of programming are considered.

Keywords: evolution software development, module, procedure-parametric programming.