PLC-2017

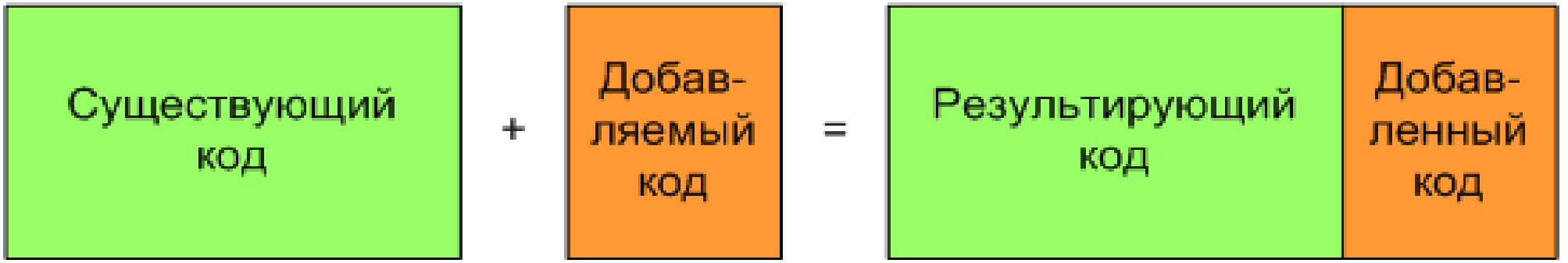
СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ
SIBERIAN FEDERAL UNIVERSITY

# Эволюционная разработка программ с применением процедурно-параметрической парадигмы

*Легалов А.И. (legalov@mail.ru)*

# Специфика эволюционного расширения

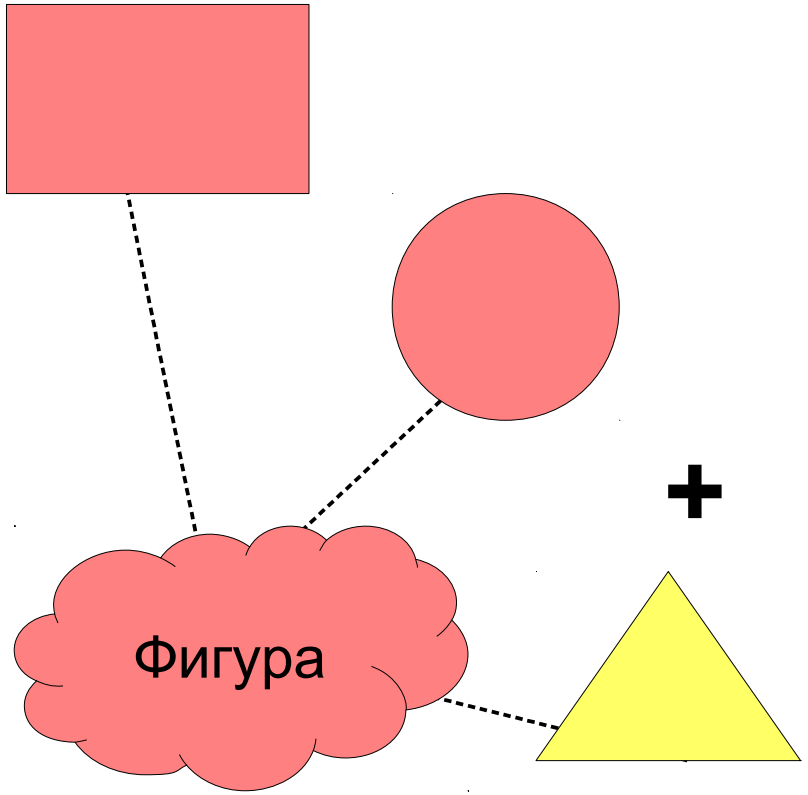

а) Изменение существующего кода




б) Эволюционное расширение

# Специфика эволюционного расширения

Специализации




**Обобщение**

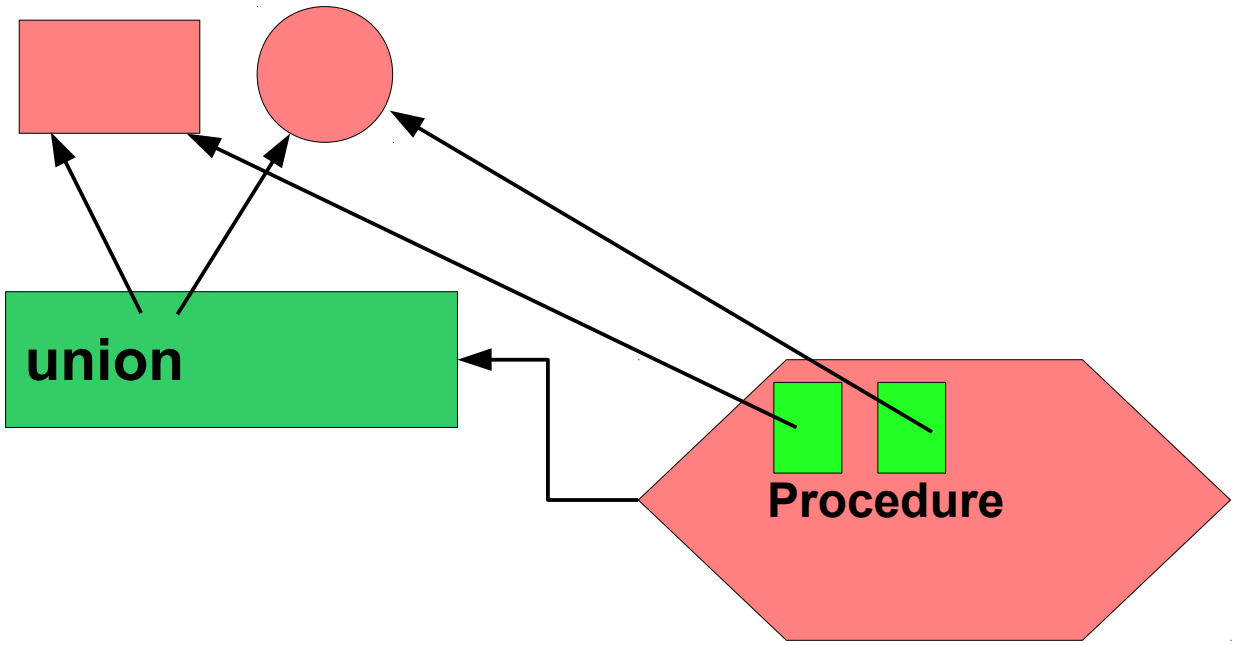
Обработчики специализаций

- Вывод прямоугольника
- Периметр прямоугольника
- Площадь прямоугольника
- Вывод круга
- Периметр круга
- Площадь круга
- Вывод треугольника
- Периметр треугольника
- Площадь треугольника
- Вывод фигуры
- Периметр фигуры
- Площадь фигуры

**Обобщающие процедуры**

# Процедурное программирование

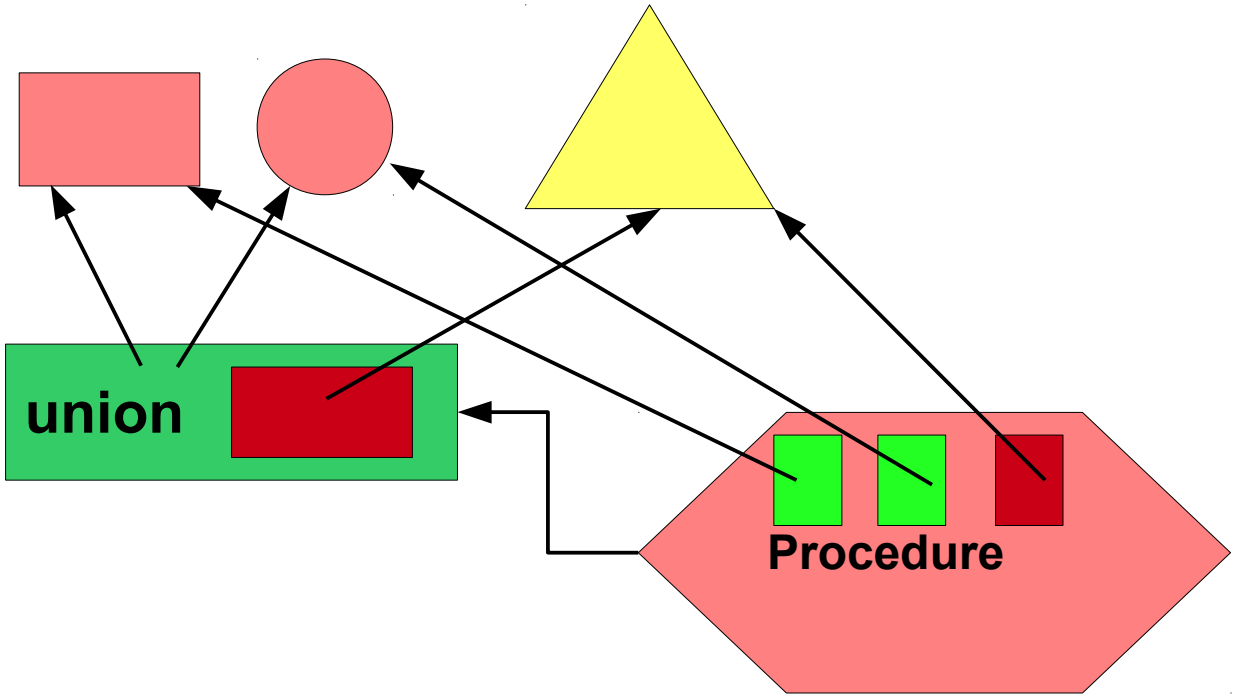
```
struct Rectangle { int x; int y; };
struct Circle { int r; };

enum key { rectangle, circle };

struct Figure {
    key k;
    union {
        Rectangle r;
        Circle c;
    };
};

void Procedue(Rectangle& r);
void Procedue(Circle& r);

void Procedure(Figure& f) {
    switch(key) {
    case rectangle: P
        Procedure(f.r);
    break;
    case circle:
        Procedure(f.c);
    break;
    }
}
```




- Безболезненное добавление новых процедур.
- Специализации можно использовать в разных обобщениях.
- Изменение кода при добавлении новых специализаций (можно избежать).
- Изменение обобщающих процедур при добавлении новых специализаций и их обработчиков.

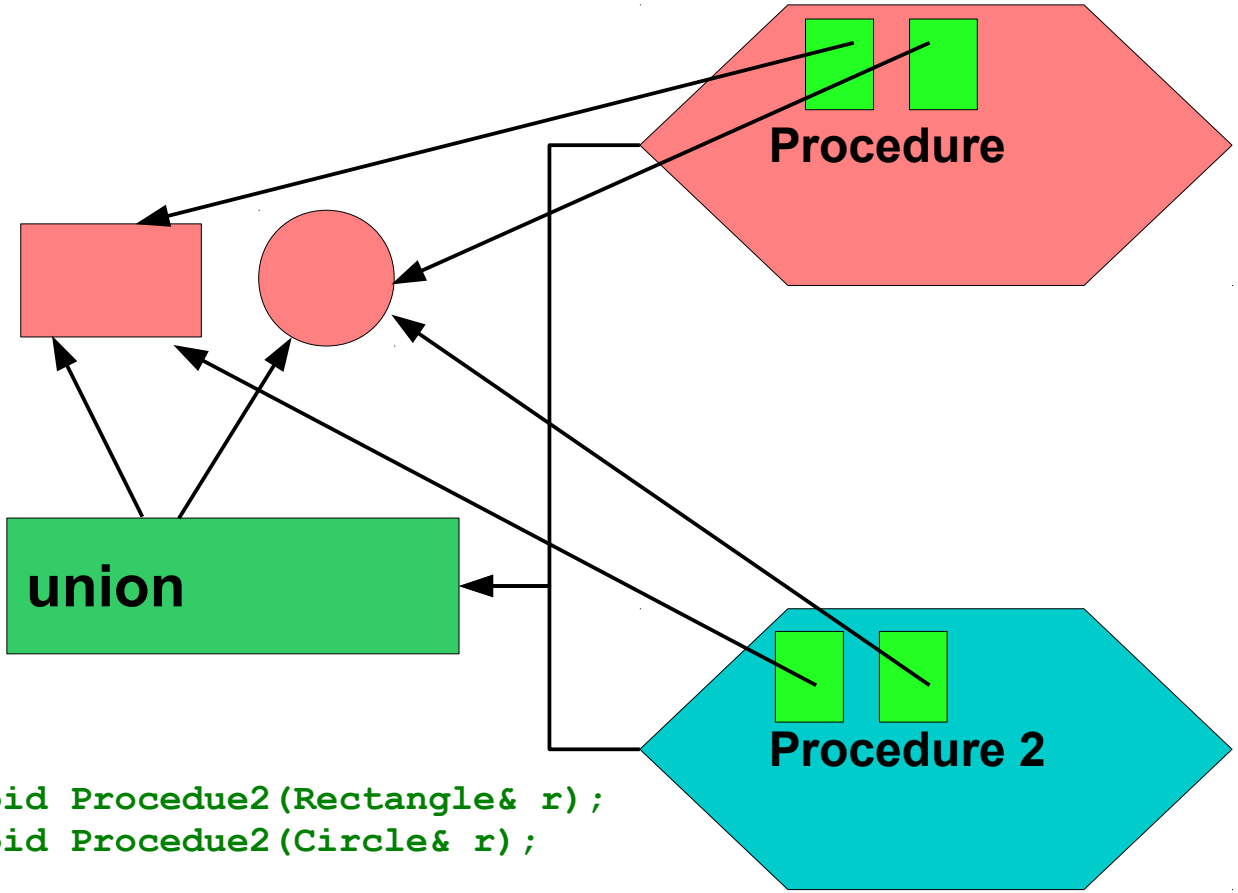
# Процедурное программирование. Добавление специализаций

```
struct Rectangle { int x; int y; };
struct Circle { int r; };
struct Triangle { int a; int b; int c;};

void Procedue(Rectangle& r);
void Procedue(Circle& r);
void Procedue(Triangle& r);

enum key { rectangle, circle, triangle };

struct Figure {
     key k;
     union {
          Rectangle r;
          Circle c;
          Triangle t;
     };
};

void Procedure(Figure& f) {
     switch(key) {
     case rectangle: P
          Procedure(f.r);
     break;
     case circle:
          Procedure(f.c);
     break;
     case triangle:
          Procedure(f.t);
     break;
     }
}
```



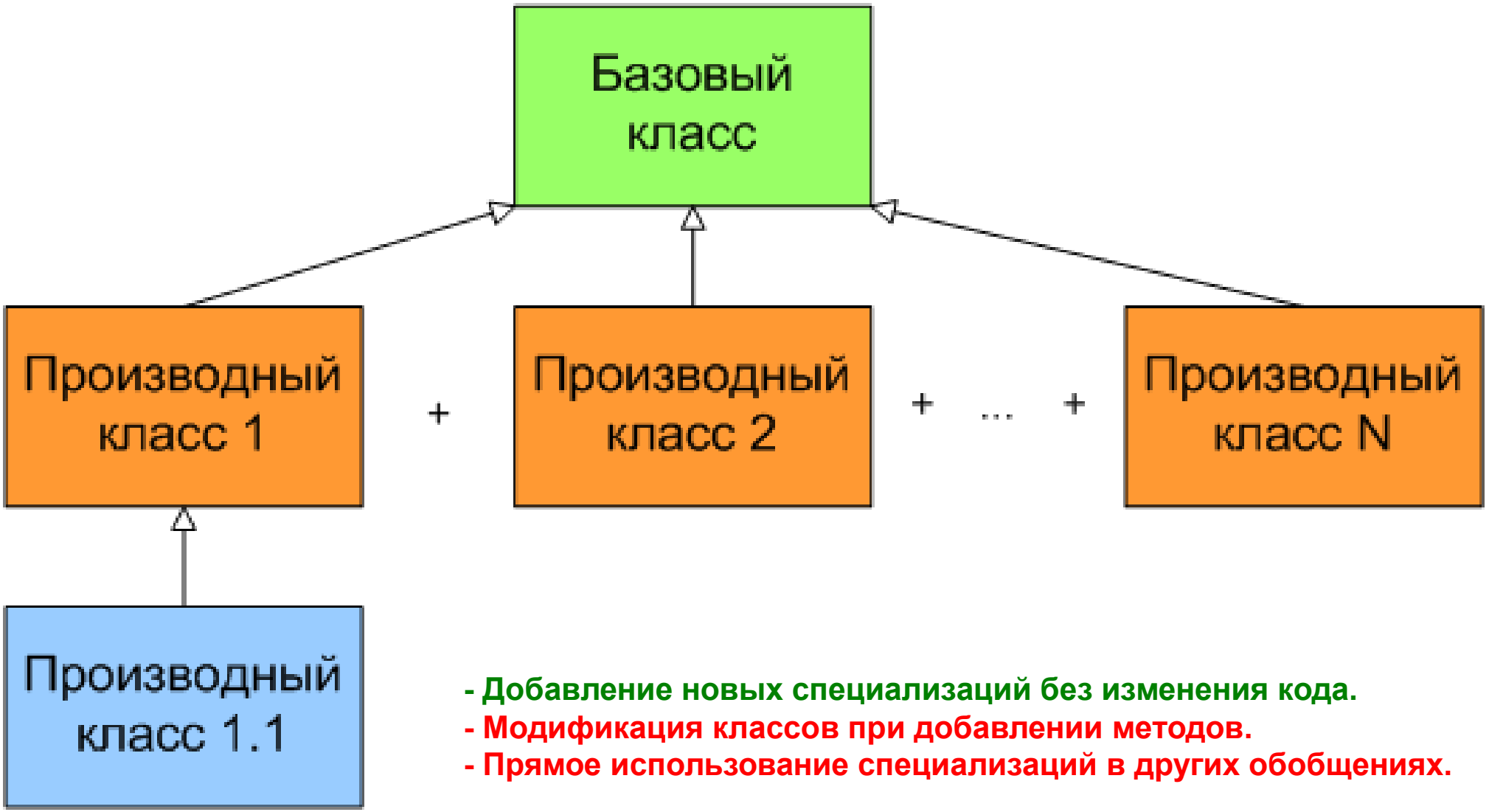
# Процедурное программирование. Добавление процедуры

```
struct Rectangle { int x; int y; };
struct Circle { int r; };

enum key { rectangle, circle };

struct Figure {
     key k;
     union {
          Rectangle r;
          Circle c;
     };
};

void Procedue(Rectangle& r);
void Procedue(Circle& r);

void Procedure(Figure& f) {
     switch(key) {
     case rectangle: P
          Procedure(f.r);
     break;
     case circle:
          Procedure(f.c);
     break;
     }
}
```




```
void Procedue2(Rectangle& r);
void Procedue2(Circle& r);

void Procedure2(Figure& f) {
     switch(key) {
     case rectangle: P
          Procedure2(f.r);
     break;
     case circle:
          Procedure2(f.c);
     break;
     }
}
```

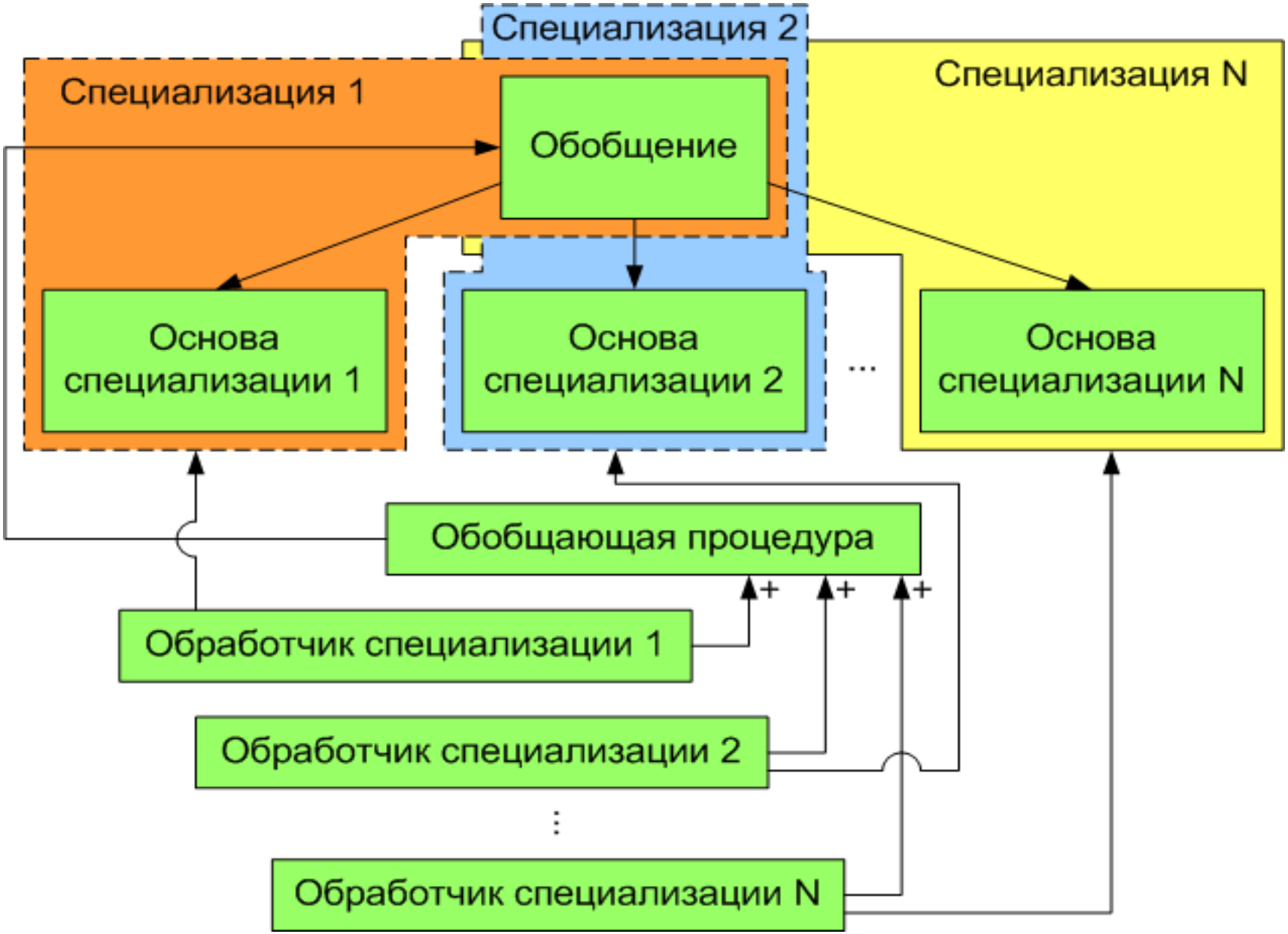
# Объектно-ориентированное программирование




- Добавление новых специализаций без изменения кода.
- Модификация классов при добавлении методов.
- Прямое использование специализаций в других обобщениях.

# Процедурно-параметрический полиморфизм

# Безболезненное расширение обобщений и процедур?
# → процедурно-параметрический стиль



# Обобщенная запись

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

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

VAR v(y): T0
    ...
v.x := 10;
v(y) := 3.14;
```

***или***

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

```
...
Circle* = RECORD
r* : INTEGER; // радиус круга
END;

// Процедура вывода
PROCEDURE Output*(VAR c: Circle);
BEGIN ... END Output;
...

Rectangle* = RECORD
x*, y* : INTEGER;
END;

// Процедура вывода
PROCEDURE Output*(VAR r: Rectangle);
BEGIN ... END Output;
```

```
struct Circle {
    // Радиус круга
    int r;
};

void Out(ofstream &ofst, Circle& c) {
    ofst << "Circle: r = " << c.r << endl;
}

struct Rectangle {
    // Стороны прямоугольника
    int x;
    int y;
};

void Out(ofstream &ofst, Rectangle& r) {
    ofst << "Rectangle: x = " << r.x << ",
                        y = " << r.y << endl;
}
```

```
Figure* = CASE TYPE OF
  Rectangle | Triangle
END;

//Обобщенная параметрическая процедура вывода фигуры
PROCEDURE Output* {VAR f: Figure} := 0;

/* Вывод обобщенного прямоугольника */
PROCEDURE Output {VAR r: Figure(Rectangle)};
BEGIN MRect.Output(r); END Output;

/* Вывод обобщенного треугольника */
PROCEDURE Output {VAR t: Figure(Triangle)};
BEGIN MTrian.Output(t); END Output;
```

```
// Figure,h
struct Figure {
    int mark;
};

// Счетчик зарегистрированных фигур
extern int figuresCounter;

typedef void (*OutFigureFunc)(ofstream &ofst, Figure& f);
extern OutFigureFunc outFigure[];

// Вывод обобщенной фигуры в поток
void OutFigure(ofstream &ofst, Figure& f);

==========================================================
// Figure.cpp
int figuresCounter = 0;

// Описание переменной, используемой
// для регистрации функций вывода фигур-специализаций
OutFigureFunc outFigure[10];
```

```
// FigTriangle.h
struct FigTriangle: Figure {
    Triangle t;
};
// Задается признак фигуры-треугольника
const int triangleMark = 0;
// Вывод специализации фигуры-треугольника в поток
void Out(ofstream &ofst, FigTriangle& ft);
// Вывод специализации фигуры-треугольника в поток как фигуры
void OutFigTriangle(ofstream &ofst, Figure& f);
```

==========

```
// FigTriangle.h
// Вывод специализации фигуры-треугольника в поток
void Out(ofstream &ofst, FigTriangle& ft) {
    ofst << "Triangle is as Specialization of Figure: a = " << ft.t.a
         << ", b = " << ft.t.b << ", c = " << ft.t.c << endl;
}
// Вывод специализации фигуры-треугольника в поток как фигуры
void OutFigTriangle(ofstream &ofst, Figure& f) {
    if(f.mark == triangleMark) {
        Out(ofst, static_cast<FigTriangle&>(f));
    }
    else {
        cerr << "OutFigTriangle: Incorrect convertion Figure to FigTriangle" << endl;
        throw; // Exeption;
    }
}
```

```
namespace {
   // Класс, обеспечивающий формирование нужных связей
   // в своем конструкторе.
   class Register {
   public:
      Register();
   };

   Register::Register() {
      figuresCounter++;
      // Регистрация функции создания фигуры
      createFigureUseFileMark[triangleMark]
                                    = CreateFigTriangleUseFileMark;
      // Регистрация функции ввода
      inFigureValue[triangleMark] = InFigTriangleValue;
      // Регистрация функции вывода
      outFigure[triangleMark] = OutFigTriangle;
   }

   // Объект, обеспечивающий регистрацию
   Register trianRegister(InFigTriangleValue, OutFigTriangle");
}
```

# Записи с уже существующими обобщениями

```
ColoredFigure = RECORD
  color: INTEGER;
  CASE Figure
END;
```

**В отличие от концепции базового типа, расширяемого за счет добавления в производных типах, использование параметрического обобщения предполагает сохранение исходного типа, а альтернативные специализации вводятся как его уточнения.**

**Внешне все специализации имеют единый тип, а их разнообразное толкование используется только внутри него. Это позволяет убрать глобальную идентификацию типов, применяемую при расширении записей или наследовании, и обеспечивает поддержку концепции строгой типизации.**

**==> Возможность табличных обращений**

**Косвенная адресация --- Индексация**

# Возможности обобщенных записей

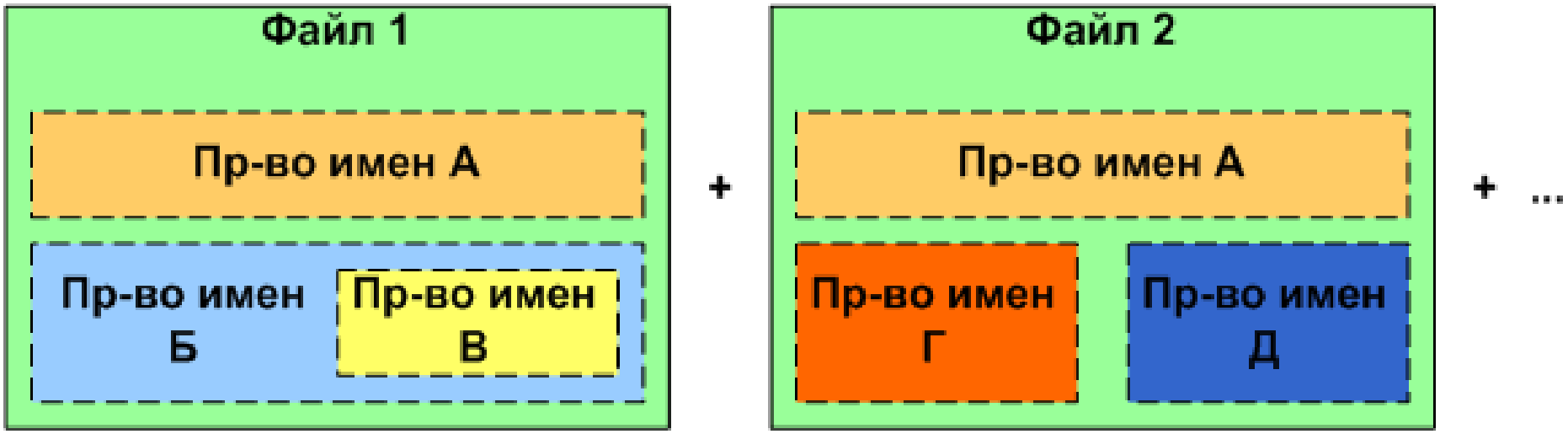
```
// Идентификация по признаку
T = RECORD x, y: INTEGER; CASE OF END;
T += t0: BOOLEAN;
T += t1: RECORD r: REAL; s: CHAR END;

v0: T(t0); v1:T(t1);
v0.x, v1.y, v0(), v1().r …

// Формирование статических цепочек
T += t3: T;
T(t3), T(t3)(t3),
T(t3)(t3)(t3)(t3)(t2)(t00), …
```

# Статика вместо динамики

```
// Декоратор точки
Decorator = CASE TYPE OF END;
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;
// Общий декоратор
Decorator += PointDecorator | ColorDecorator
           | AngleDecorator | Figure;

// Цветная фигура на плоскости через декораторы
NewFigure = PointDecorator(AngleDecorator
                 (ColorDecorator(
                      ColorDecorator(Figure))));

VAR   circle: NewFigure(Circle);
      rectangle: NewFigure(Rectangle);
```

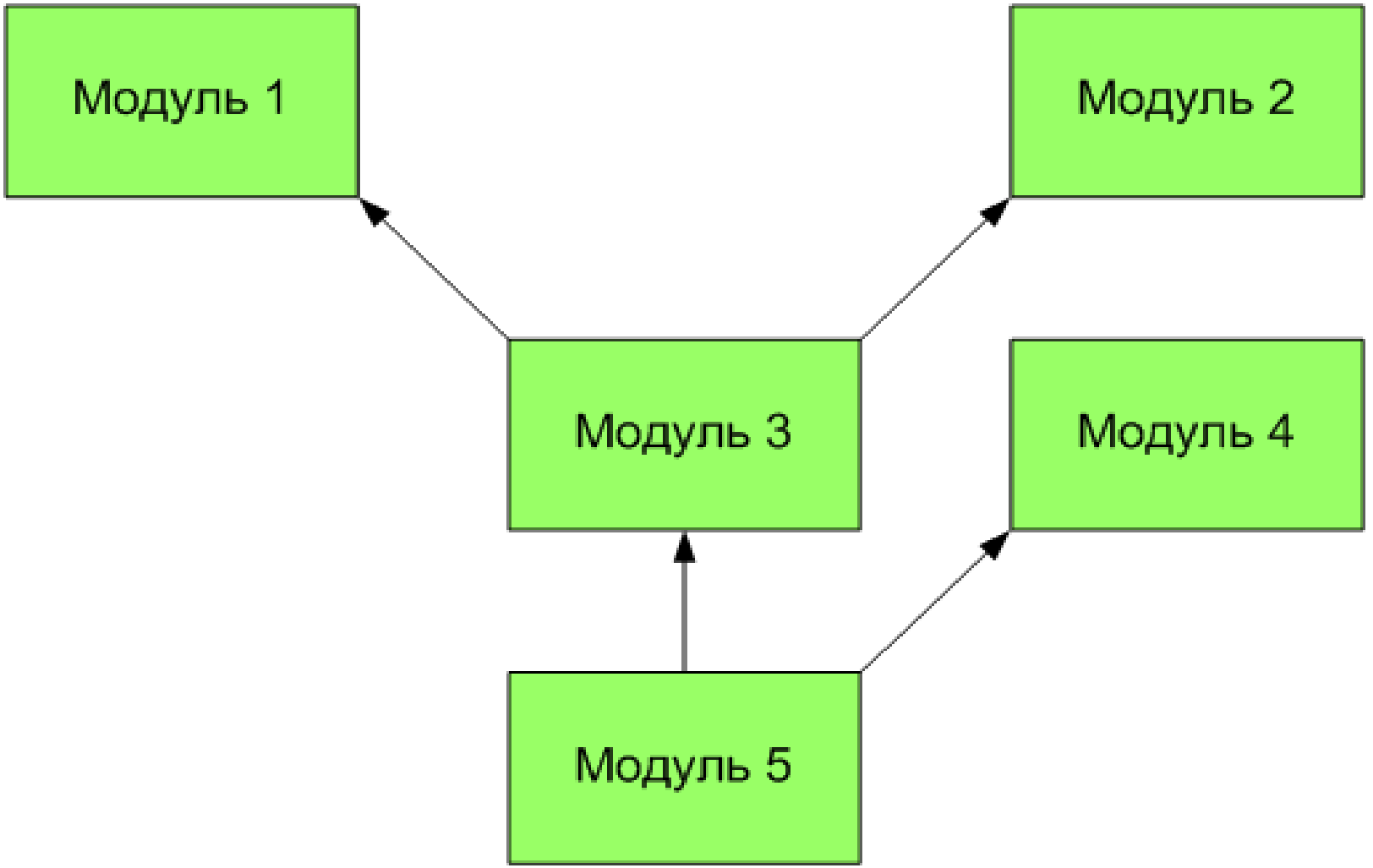
# Подключаемые модули

# Пространства имен

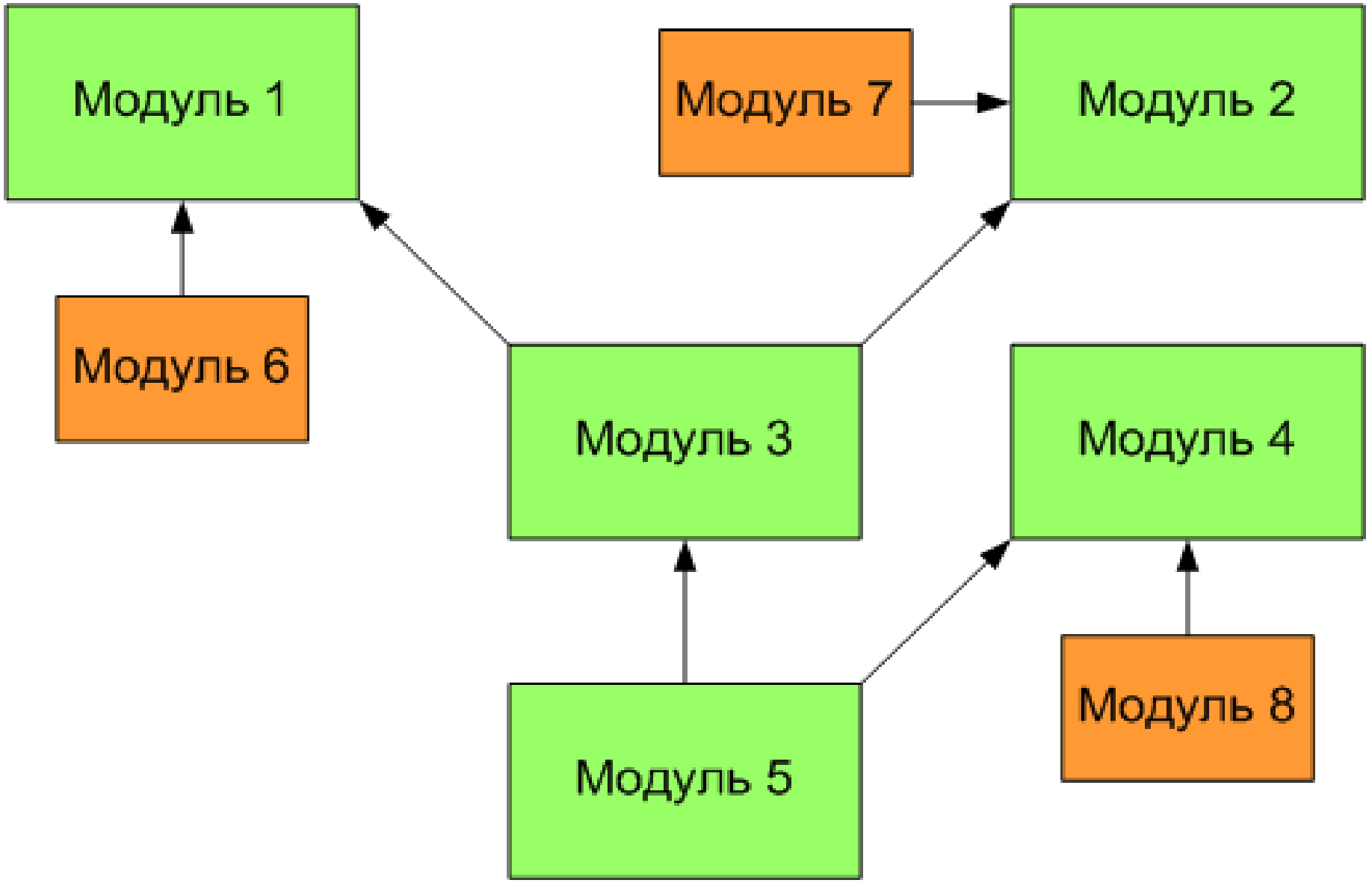



C++, C#

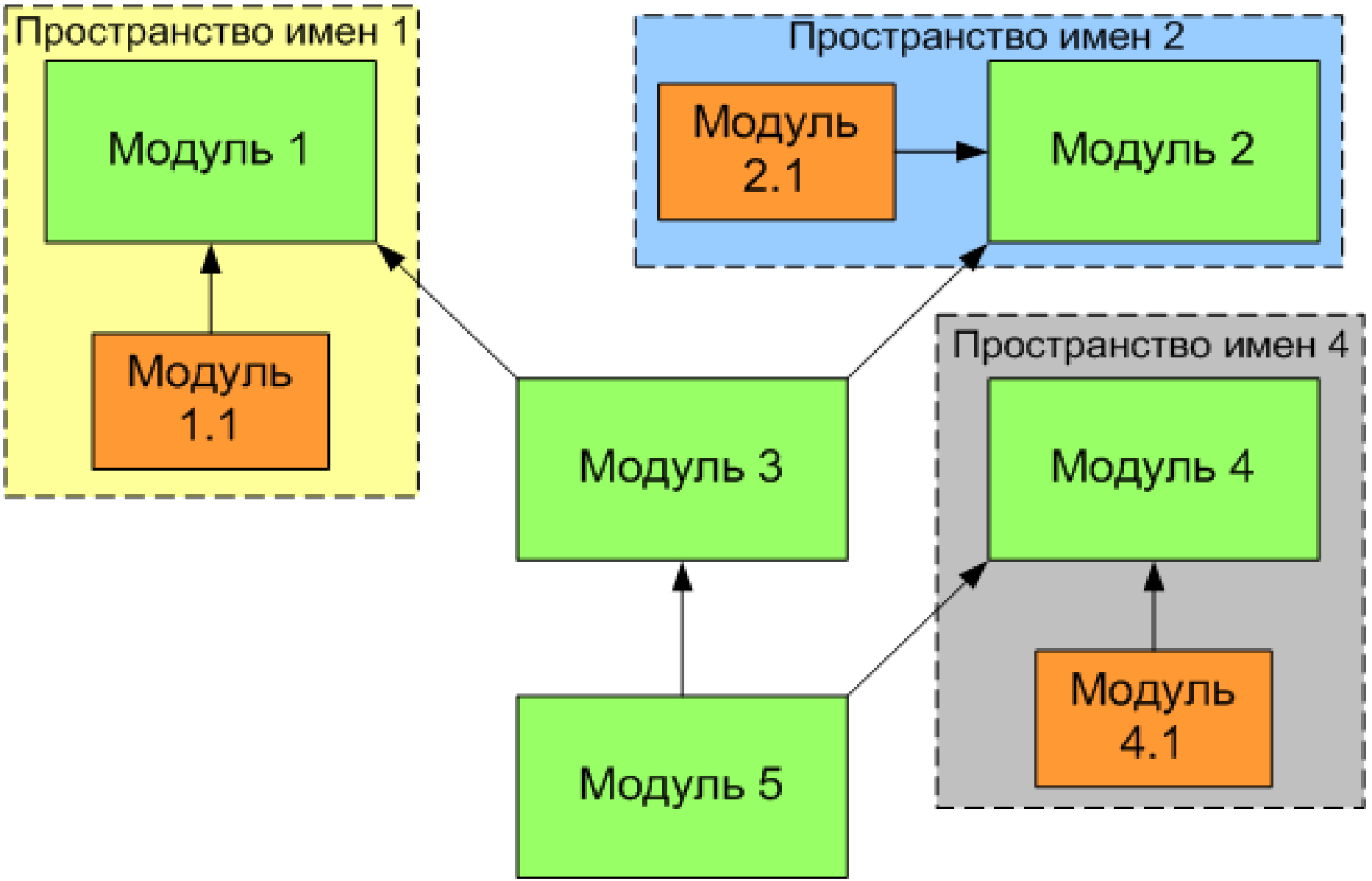
# Традиционная модульная структура



# Расширение иерархической модульной структуры



# Подключаемые модули



# Модуль с прямоугольником

```
// 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 с треугольником

```
// 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.
```

# Модуль с обобщенной фигурой

```
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.
```

# Модуль с процедурой, реализующей мультиметод

```
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.
```

# Клиентский модуль, использующий обобщающие процедуры

```
MODULE Mclient; // Клиентский модуль, обрабатывающий обобщения
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 специализации

```
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.
```

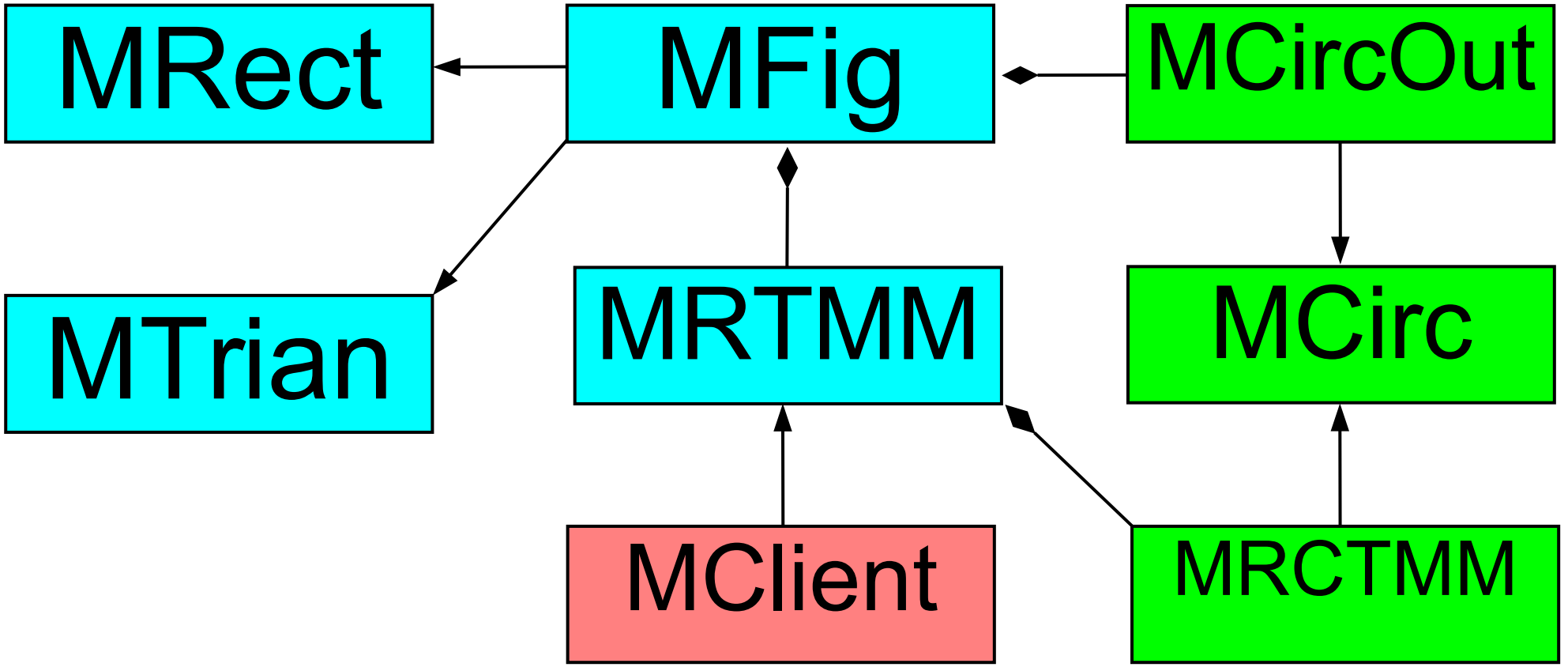
## Расширение процедуры вывода после добавления специализации

```
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 MCircle.Output(c) END Output;
END MCircOut.
```

# Расширение мультиметода после добавления специализации

```
MODULE MRCTMM (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.
```

# Зависимости между модулями в ходе эволюционного расширения



# Эволюционное расширение с поддержкой динамического связывания

| Ситуация | Возможность расширения | | |
|---|---|---|---|
| | **ПП** | **ООП** | **ППП** |
| **1** Расширение обобщений новыми специализациями | **нет** | **есть** | **есть** |
| **2** Добавление новых процедур | **есть** | **нет** | **есть** |
| **3** Добавление новых полей в существующие типы данных | **косвенное, для расширяемых типов** | **косвенное, при наличии RTTI** | **косвенное, при использовании обобщенной записи** |
| **4** Добавление процедур, осуществляющих обработку одной специализации | **есть** | **нет** | **есть процедурный и параметрический** |
| **5** Создание нового обобщения на основе существующих специализаций | **есть** | **косвенное** | **есть** |
| **6** Добавление мультиметодов | **есть** | **нет** | **есть** |
| **7** Изменение мультиметодов при добавлении новых специализаций | **нет** | **нет** | **есть** |

# Благодарю за внимание!

# Дополнительная информация

Раздел сайта, посвященный процедурно-параметрическомму программированию - http://www.softcraft.ru/ppp/

Эволюция мультиметодов при процедурном подходе - http://www.softcraft.ru/coding/evp/

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

Легалов А.И., Косов П.В., Легалов И.А. Использование процедур с одинаковой сигнатурой для эволюционного расширения программ./ Доклады АН ВШ РФ, № 1 (26), 2015. - С. 41-51.

Legalov A, Kosov P. Evolutionary software development using procedural-parametric programming. / CEE-SECR '13 Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia. ACM New York, NY, USA ©2013. ISBN: 978-1-4503-2641-4. Article No. 3.