Почему объектно-ориентированный подход победил процедурный?

Распространенность парадигм программирования во многом обуславливается их способностями поддерживать современные методологии разработки программного обеспечения. Основной показатель программных продуктов - **сложность**, а основными требованиями к методологиям разработки являются:
удобство сопровождения, возможность безболезненного наращивания уже существующей программы, способность разработанных программных объектов к повторному использованию.

На второй план отступает такое требование, как **быстрое проектирование первоначальной (полнофункциональной) версии программы**, потому что его воплощение обычно не позволяет соблюсти все остальные условия.

Процесс разработки программного обеспечения не заканчивается выпуском одного релиза. Он сводится к **итеративному расширению предыдущих версий**, что, в некоторой степени, и помогает решать проблему сложности.

Техника эволюционного развития реализована в **возвратном проектировании**. Она же используется при экстремальном программировании (**X-programming**).

1. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++, 2-е изд./Пер. с англ. - М.: "Издательства Бином", СПб: "Невский диалект", 1998 г. - 560 с., ил.
2. Бек К. Экстремальное программирование./Пер. с англ. – СПб.: Питер,
3. Бек К. Экстремальное программирование: разработка через тестирование./Пер. с англ. – СПб.: Питер, 2003. – 224 с.

Эволюционная разработка – один из основных факторов, определяющих создание современного программного обеспечения

Окончание очередного витка эволюционного цикла разработки версии продукта, приводит к получению кода, написанного на некотором языке программирования. **Новый виток** спирали требует расширения и модификации этого кода, то есть его **эволюции**.

1. Его нельзя переделать «мастером», его невозможно автоматически сгенерировать при повторном проектировании.
2. При этом желательно как можно меньше изменять уже написанный код, используя методы, позволяющие его наращивать.
3. Лучше всего, если только добавлять новый код.

   То есть (учитывая метаморфозы терминологии), эволюционное программирование можно ассоциировать с **эволюционным кодированием**.

Основные виды отношений между программными объектами

Отличие парадигм программирования заключается в различных способах реализации моделей состояния и поведения. Эти различия проявляются через отношения между алгоритмами и данными, способами описания алгоритмов и способами описания данных.

Известная всем метафора:

Алгоритмы + Данные = Программы

Отражает только простейшие отношения, существующие между данными и методами их обработки.

Более точным отражением является текущего состояния является:

Программа =

Конструктив (Данные,

Алгоритмы)

В борьбе конструктивов и проявляется формирование современных принципов эволюционного программирования.

ООП предложило более гибкие конструктивы для построения эволюционно расширяемых программ

Абстрагирование от конкретных экземпляров достигается за счет введения специальных понятий, определяющих как семантические артефакты, например, ***"абстрактный тип данных"*** и ***"процедура"*** (понятие ***"функция"*** используется как синоним процедуры).

Также выделяются базовые понятия используются для конструирования составных программных объектов путем объединения в ***агрегаты*** и разделения по **категориям**. **Категорию** Г. Буч называет иерархией типа **"is-a".** Она также трактуется как ***обобщение*** программных объектов.

Цикритзис Д., Лоховски Ф. Модели данных. Пер. с англ. - М.: Финансы и статистика, 1985. - 344 с.

Агрегаты **и** ***обобщения*** используются при конструировании **композиций из данных и процедур**. В каждой из существующих парадигм программирования вопросы такого конструирования композиций решаются **по-своему**, что и вносит определенные **отличительные черты**.

Артефакт (по Вирту) – создаваемый программистом программный объект с применением средств системы программирования.
Программный объект – любая конструкция допускаемая в системе (языке) программирования (никоим образом не должен ассоциироваться только с ООП).

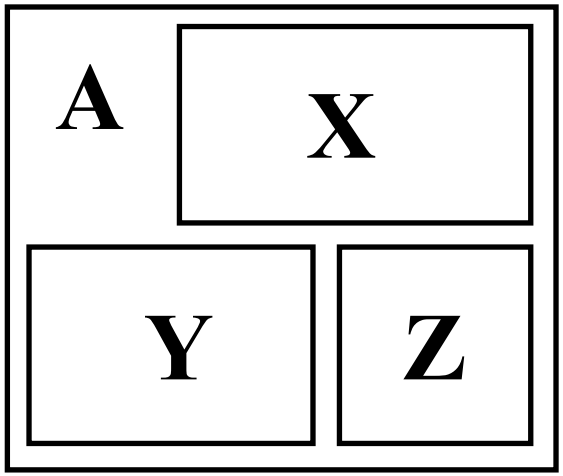
Конструирование агрегатов

Агрегация (агрегирование) - это абстрагирование, посредством которого один объект конструируется из других [Цикритзис].

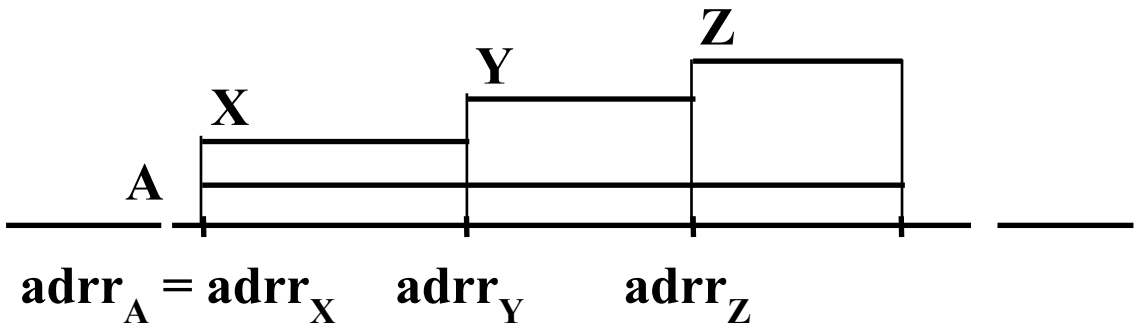
Агрегирование обеспечивает формирование программных объектов одним из способов:

- **непосредственным включением**,
- **косвенным (ссылочным) связыванием**,
- **с применением наследования (расширения)**,
- **образного восприятия**.

Наиболее типичным является восприятие агрегата как единой абстракции, сформированной **непосредственным включением** используемых в нем программных объектов. В нашем сознании он видится как единый, **монолитный ресурс**, занимающий некоторое **неразрывное пространство**. Цельность и законченность данного объекта не требует выполнения ***дополнительных алгоритмов***, связанных с формированием его структуры. Можно сразу приступать к алгоритмическому использованию объекта, например, его инициализации.



а) условное изображение на плоскости

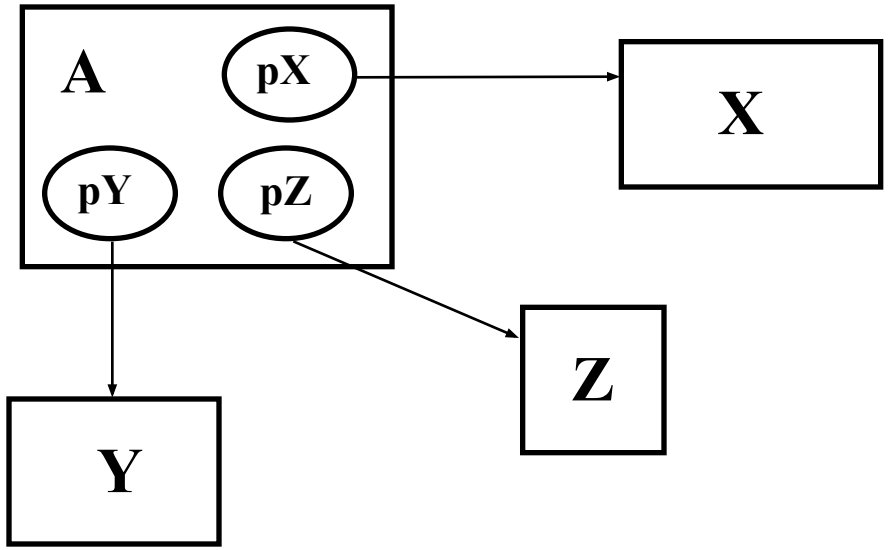


б) отображение на одномерное адресное пространство

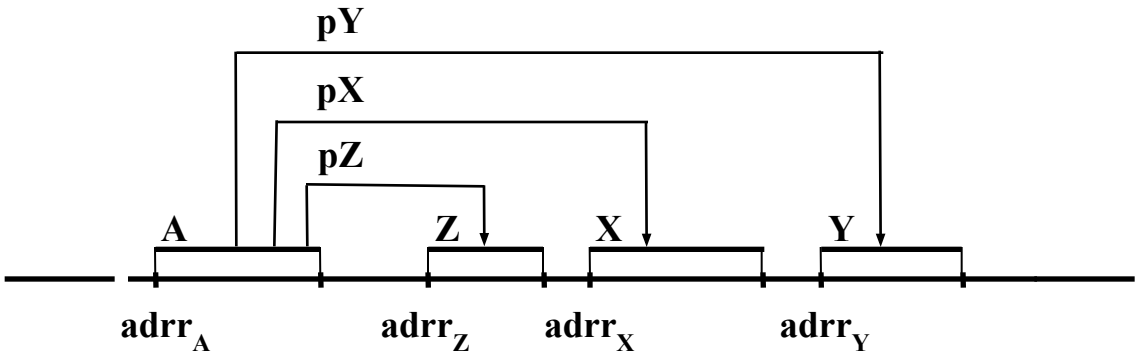
Агрегат A из элементов X, Y, Z, построенный с использованием непосредственного включения.

Косвенное связывание позволяет формировать агрегаты из **отдельных элементов**, уже располагаемых в пространстве. Взаимосвязь осуществляется **алгоритмическим заданием значений ссылок**. Спецификой является **выполнение кода обеспечивающего конструирование объекта** из разрозненных экземпляров абстракций. Однако этот код выполняется только один раз, после чего работа с агрегатом осуществляется точно так же, как и при включении.

В ряде случаев, когда элементы и агрегат создаются статически, инициализация связей может быть проведена во время трансляции программы.



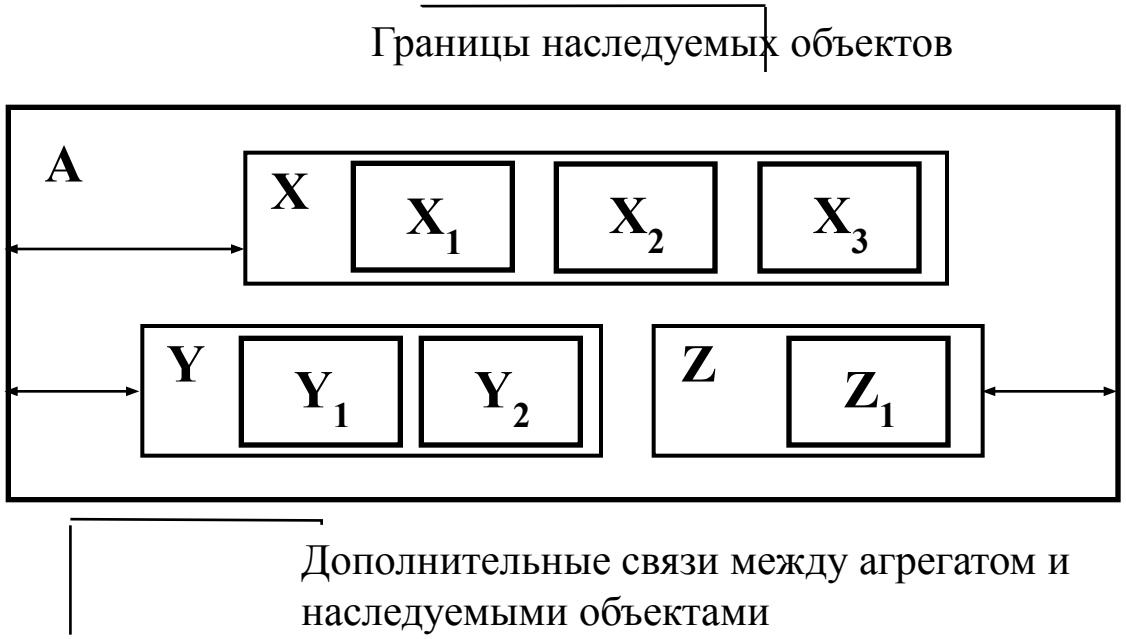
а) условное изображение на плоскости



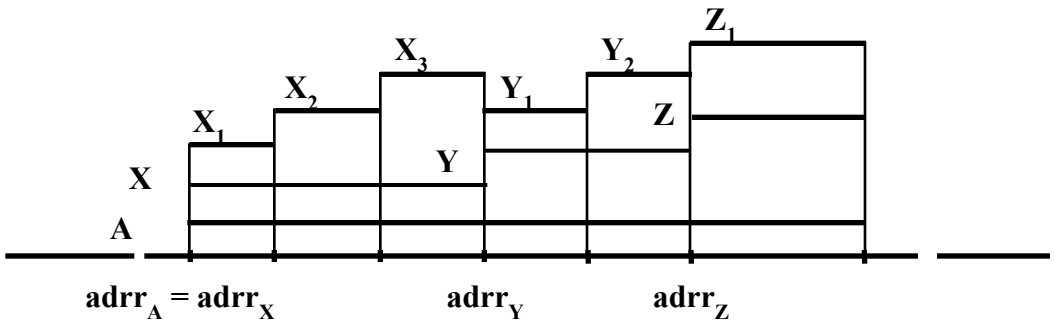
б) отображение на одномерное адресное пространство

Агрегат A из элементов X, Y, Z, построенный с использованием косвенного связывания

Наследование позволяет создавать структуру объекта, эквивалентную той, которая формируется непосредственным включением. Но поддерживает свойства **конкатенации**, в результате чего обращение к элементам подключаемых абстракций осуществляется **напрямую**. **Сохраняется информация о базовых абстракциях**, которая может использоваться для разрешения проблем неоднозначности доступа к элементам сформированного агрегата при совпадении их имен. В отличие от непосредственного включения, каждому наследуемому элементу "предоставляется" дополнительная информация о формируемом агрегате. Использование дополнительной информации позволяет правильно манипулировать агрегатом, используя лишь сведения об одном из его базовых (включаемых) элементов. Например, через базовый элемент можно определить тип агрегата, его размер, выполнить операцию удаления (реализация к конструктиве **полиморфизма**).



а) условное изображение на плоскости



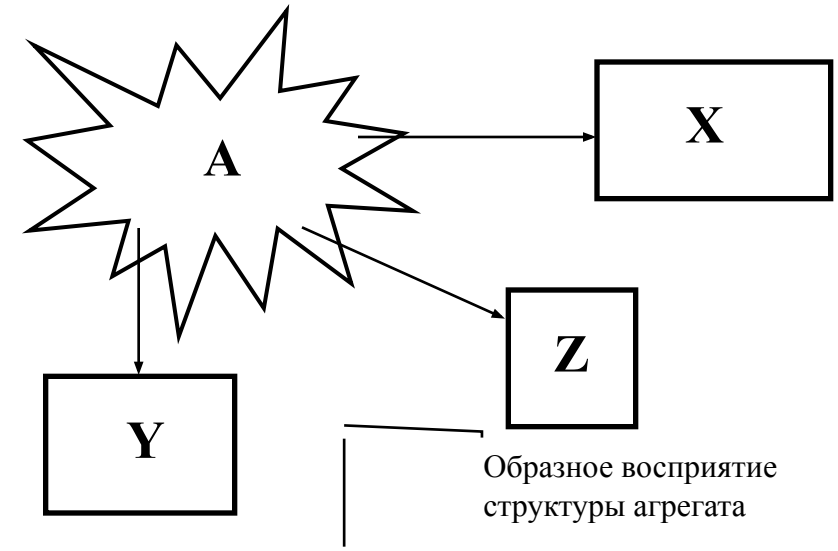
б) отображение на одномерное адресное пространство

Агрегат A из элементов X, Y, Z, построенный с использованием наследования

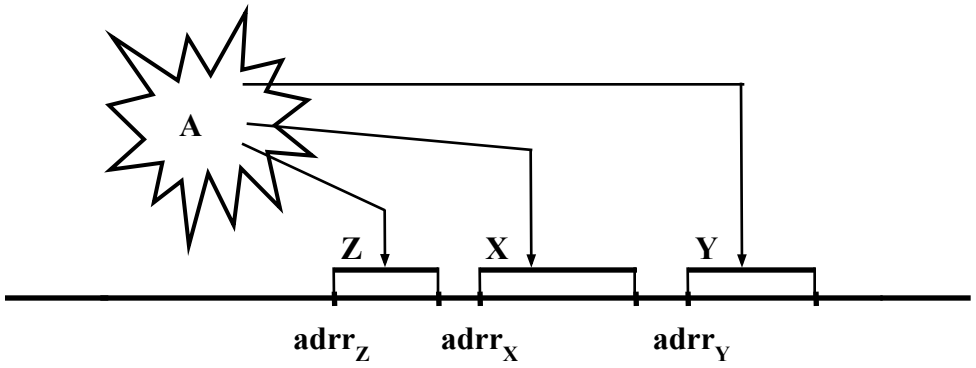
Образное агрегирование связано с отсутствием специально созданной абстракции, соответствующей формируемому агрегату. Вместо этого **агрегат воссоздается только в мысленном восприятии программиста**, а на уровне программы имеются его разрозненные элементы, обрабатываемых как единое целое.

Например, точку на плоскости можно представить как две независимые целочисленные переменные x и y.

Такое агрегирование уходит корнями в далекое прошлое (эпоху Фортрана и Алгола-60), но и сейчас встречаются программисты, которым "лень" вписать лишнюю абстракцию. Это приводит к определенным проблемам, связанным с **мобильностью** и **повторным использованием** кода (но иногда так хочется поскорее написать программу, что не остается времени на раздумья о стиле!).



а) условное изображение на плоскости



б) отображение на одномерное адресное пространство

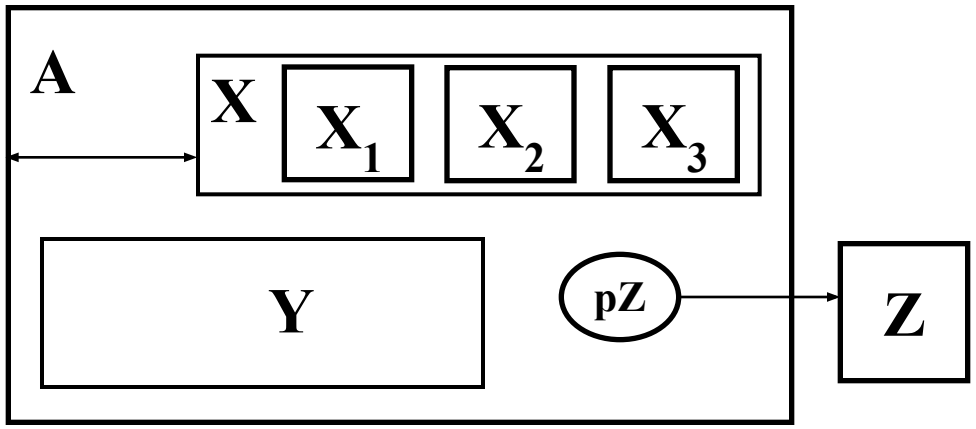
Агрегат A из элементов X, Y, Z, построенный на основе образного восприятия

Наряду с "**чистыми**" методами, агрегаты могут строиться по **смешанному** принципу, когда одновременно применяются различные комбинации методов агрегирования.
В качестве примера представлен агрегат, выстроенный с применением включения, косвенного связывания и наследования.

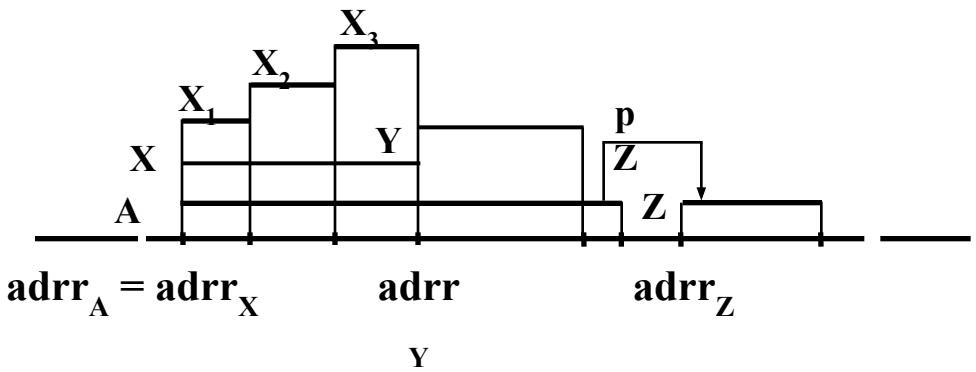
Такой подход является **наиболее популярным** в настоящее время, так как позволяет рационально использовать различную технику в зависимости от организации и назначения агрегируемых элементов.

Дополнительные связи между агрегатом и наследуемыми объектами

Границы наследуемых объектов



а) условное изображение на плоскости



б) отображение на одномерное адресное пространство

Агрегат A из элементов X, Y, Z, построенный с использованием наследования, непосредственного включения и косвенного связывания

К специфике различных парадигм программирования можно отнести способы построения агрегатов и их использования. В частности, при **процедурном программировании осуществляется разделение на агрегаты данных и процедур.**

Объектно-ориентированный подход использует комбинированное агрегирование.

Ниже рассмотрены особенности построения и использования агрегатов для этих парадигм.

Процедурное агрегирование

Агрегатам данных соответствуют конструктивные типы данных. Они состоят из объектов данных, подключаемых непосредственно или с использованием ссылок (указателей). Будем использовать понятие "***запись***". Процедурный подход предполагает независимость записей (**R**) от процедур (**P**). Запись **R** состоит из элементов данных: $\mathbf{R=(D_1, D_2, ..., D_k)}$, называемых ***полями записи***.

Агрегатам процедур соответствуют вложения в тела процедур различных по иерархии объектов действия: операций, операторов, вызовов процедур, осуществляющих обработку отдельных элементов (полей) записи. Обозначим данную абстракцию понятием "***независимая процедура***". Доступ к различным экземплярам записи, определяющим агрегируемую абстракцию, осуществляется через один из элементов списка формальных параметров.

Пример использования процедурного агрегирования

```
//---------------------------------------------------------
// Простейший контейнер на основе одномерного массива
//---------------------------------------------------------
// Контейнер должен знать о фигуре
#include "shape_atd.h"
//---------------------------------------------------------
// Данные контейнера
struct container
{
    enum {max_len = 100}; // максимальная длина
    int len; // текущая длина
    shape *cont[max_len];
};
```

```
//-------------------------------------------------
// Процедуры должны знать о контейнере и фигуре,
// доступной через модуль, описывающий контейнер
#include "container_atd.h"
//-------------------------------------------------
// Инициализация контейнера
void Init(container &c)
{
    c.len = 0;
}
//-------------------------------------------------
// Очистка контейнера от элементов (освобождение памяти)
void Clear(container &c)
{
    for(int i = 0; i < c.len; i++)
    {
      delete c.cont[i];
    }
    c.len = 0;
}
```

```
//-----------------------------------------------------
// Необходим прототип функции, формирующей фигуру при вводе
shape *In();
//-----------------------------------------------------
// Ввод содержимого контейнера
void In(container &c) {
    cout
      << "Do you want to input next shape"
         " (yes=\'y\', no=other character)? "
      << endl;
    char k;
    cin >> k;
    while(k == 'y') {
      cout << c.len << ": ";
      if((c.cont[c.len] = In()) != 0){
        c.len++;
      }
      cout
        << "Do you want to input next shape"
           " (yes=\'y\', no=other character)? "
        << endl;
      cin >> k;
    }
}
```

```
//-----------------------------------------------
// Необходим прототип функции вывода отдельной фигуры
void Out(shape &s);
//-----------------------------------------------
// Вывод содержимого контейнера
void Out(container &c)
{
    cout << "Container contents " << c.len
         << " elements." << endl;
    for(int i = 0; i < c.len; i++)
    {
      cout << i << ": ";
      Out(*(c.cont[i]));
    }
}
```

Объектно-ориентированное агрегирование

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

- Основной агрегирующей единицей является виртуальная или реальная ***оболочка*** **(С)**, Реально существующая оболочка, вместе с размещенными в ней программными объектами, чаще всего называется ***классом***. В нем могут быть размещены как данные, так и процедуры. Оболочка в явном виде может не присутствовать в языке и проявляться через связывание процедур с типами данных, как в языке программирования Оберон-2 [MoessenboeckWirth]. Но при этом подчеркивается объектно-ориентированная направленность механизма связывания процедур.

Обычные процедуры, размещаемые в классе и используемые для изменения своего внутреннего состояния, часто называются методами, функциями-членами класса. Будем использовать термин "***процедура класса***". Виртуальные процедуры, переопределяемые в производных классах, будут рассмотрены ниже как составляющие обобщений.

- Данные, определяющие внутреннее состояние класса, обычно называются ***переменными класса***.
- Термин "***оболочка класса***" (или просто "***оболочка***", если понятен контекст) будем использовать для обозначения класса в том случае, когда хотим исключить из рассмотрения его переменные и процедуры. Оболочка, при доступе извне, выступает в роли посредника к программным объектам, расположенным внутри класса.

Пример объектно-ориентированного агрегирования

```
//-------------------------------------------------
// Простейший контейнер на основе одномерного массива
//-------------------------------------------------
// Контейнер должен знать о фигуре
#include "shape_atd.h"
//-------------------------------------------------
// Описание контейнера
class container
{
    enum {max_len = 100}; // максимальная длина
    int len; // текущая длина
    shape *cont[max_len];
public:
    void In();      // ввод фигур в контейнер
    void Out();     // вывод фигур в выходного потока
    double Area(); // подсчет суммарной площади
    void Clear();  // очистка контейнера от фигур
    container();   // инициализация контейнера
    ~container() {Clear();} // утилизация контейнера
};
```

```
//-------------------------------------------------
-
// Необходимо знать описание контейнера и методов фигуры,
// доступных через container_atd.h
#include "container_atd.h"
//-------------------------------------------------
-
// Прототип обычной внешней функции, формирующей фигуру
// при вводе
shape *In();
//-------------------------------------------------
-
// Инициализация контейнера
container::container(): len(0) { }
//-------------------------------------------------
-
// Очистка контейнера от элементов (освобождение памяти)
void container::Clear()
{
    for(int i = 0; i < len; i++)
    {
      delete cont[i];
```

```
//-------------------------------------------------
-
// Ввод содержимого контейнера
void container::In() {
    cout
      << "Do you want to input next shape"
         " (yes=\'y\', no=other character)? "
      << endl;
    char k;
    cin >> k;
    while(k == 'y')
    {
      cout << len << ": ";
      if((cont[len] = simple_shapes::In()) != 0)
      {
        len++;
      }

      cout
        << "Do you want to input next shape"
           " (yes=\'y\', no=other character)? "
        << endl;
      cin >> k;
    }
}
```

```
//-------------------------------------------------------
-
// Вывод содержимого контейнера
void container::Out() {
    cout << "Container contents " << len
         << " elements." << endl;
    for(int i = 0; i < len; i++) {
      cout << i << ": ";
      cont[i]->Out();
    }
}
//-------------------------------------------------------
-
// Вычисление суммарной площади для фигур, размещенных
// в контейнере
double container::Area() {
    double a = 0;
    for(int i = 0; i < len; i++) {
      a += cont[i]->Area();
    }
    return a;
}
```

Отличие методов агрегирования

ОО агрегирование не имеет никаких преимуществ перед процедурным агрегированием

ОО подход лучше описывает окружающий нас мир – заблуждение.

```
-------------------------------------------------
class simple {
  int v;
public:
  simple(int val);
  void out();
};
-------------------------------------------------s
truct simple {
  int v;
};
simple *create_simple (int val);
void destroy_simple(simple* s);
void out (simple &s);
```

Процедурный подход позволяет лучше скрывать данные [Meyers]

```
struct simple;
simple *create_simple (int val);
void destroy_simple(simple* s);
void out (simple &s);
```

Резюме

ОО агрегирование обеспечивает более компактное написание программ за счет поддержки рациональных методов создания артефактов.

Процедурное и ОО агрегирование практически ничем не отличаются между собой, если не считать за отличие размещение процедур.

Между тем, процедурное агрегирование обеспечивает более гибкую реализацию ассоциаций, связанных с конструированием эволюционирующих программ.

При процедурном проектировании агрегатов обеспечивается лучшее сокрытие данных

Конструирование обобщений

Обобщение – это композиция альтернативных по своим свойствам программных объектов, принадлежащих к единой категории в некоторой системе классификации.

Выделяются:
обобщение данных и ***обобщение процедур (процедурное обобщение)***.
Обобщение данных состоит из ***основы обобщения***, к которой присоединяются различные ***основы специализаций***.

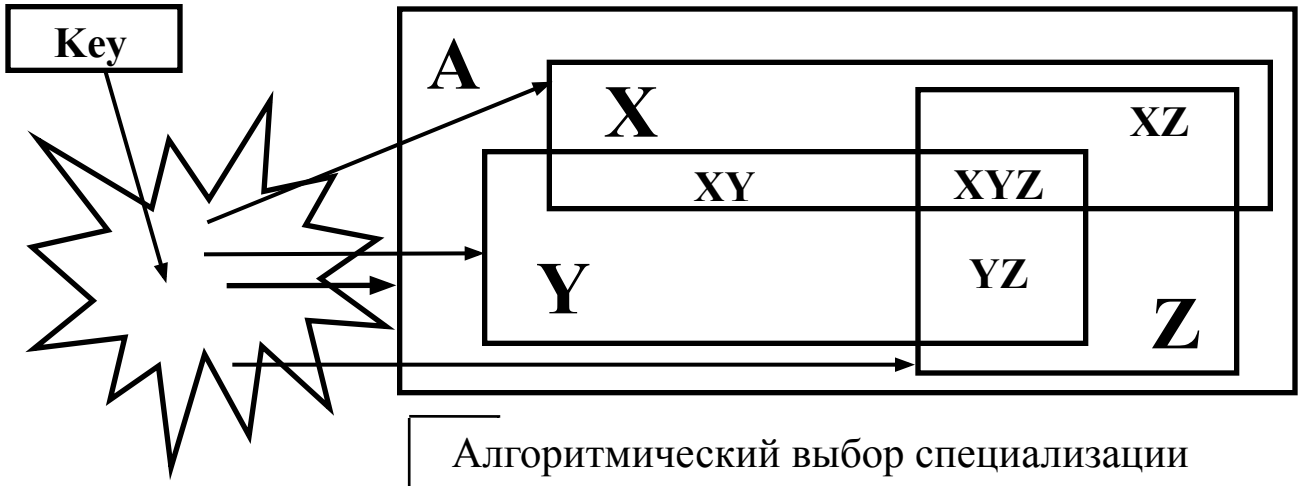
Основа специализации – программный объект, который может использоваться в качестве **специализации** того или иного **обобщения**.
Для данных –любой тип данных.
Для процедур – любая процедура.

Специализации обобщения – отдельные альтернативные понятия, принадлежащие к единой категории (присоединенные **основы специализаций**).

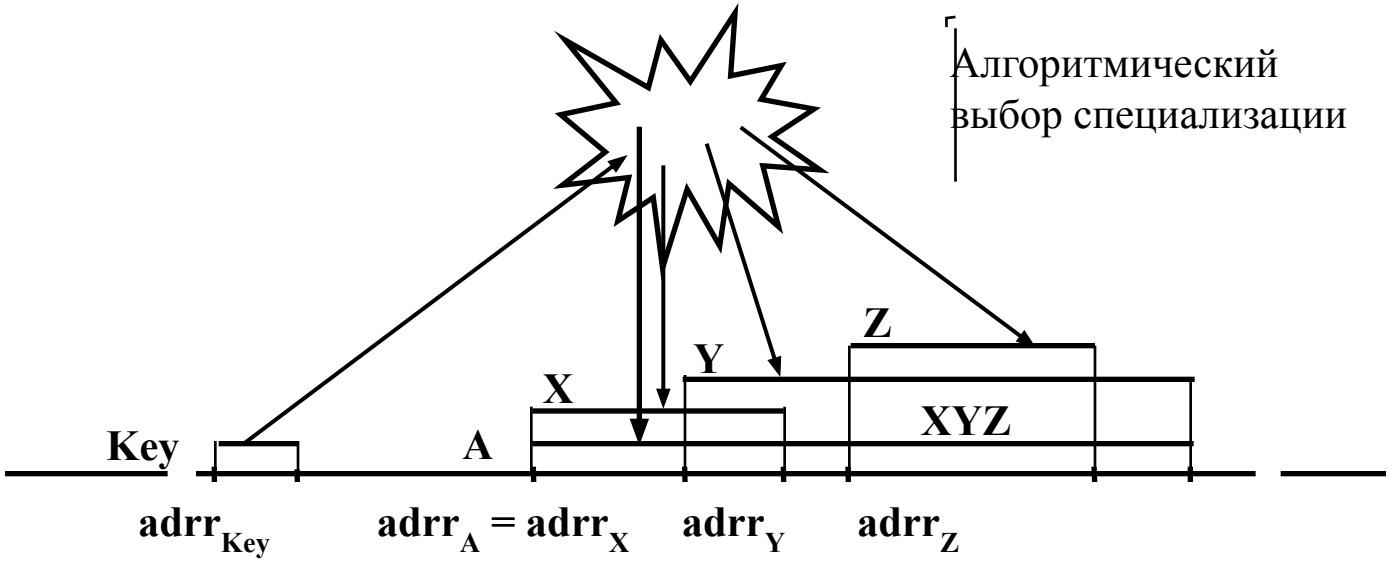
Обработка обобщений данных осуществляется соответствующими процедурами, включаемыми в состав процедурного обобщения. Процедура, связанная с обработкой основы обобщения называется ***обобщающей процедурой***.
Процедура, осуществляющая обработку отдельной специализации обобщения, называется ***специализированной***.

Методы формирования обобщений

При обобщении **на основе общего ресурса** происходит размещение специализаций в едином адресном пространстве.
При этом обычно существует часть ресурса, одновременно перекрываемая всеми размещаемыми программными объектами.
Но чаще всего обобщения на основе общего ресурса строится таким образом, что начальный адрес для всех размещаемых объектов является одинаковым.

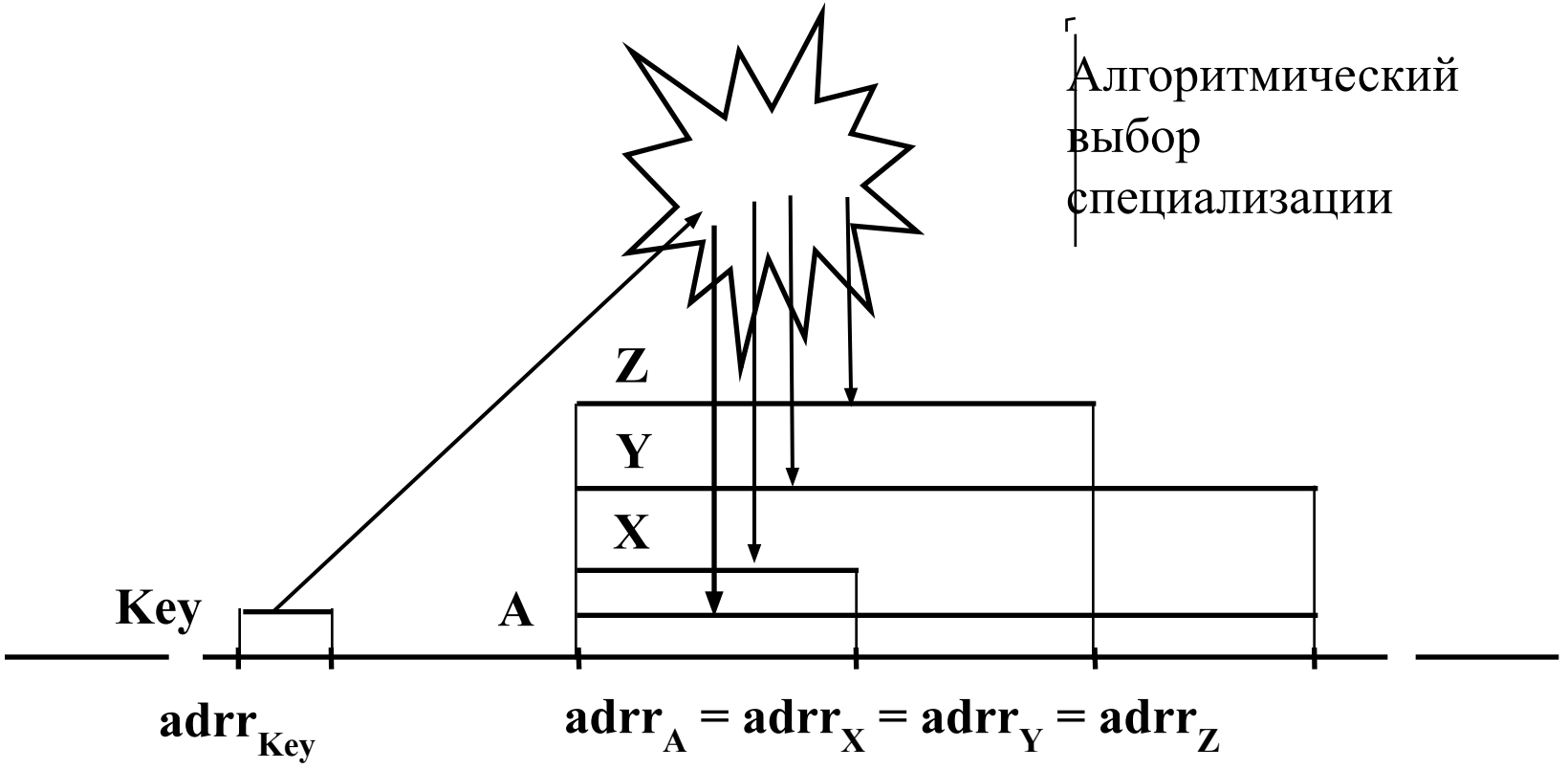


а) условное изображение на плоскости



б) отображение на одномерное адресное пространство

Обобщение А, построенное с использованием объединения на основе общего ресурса (начало)



в) отображение на одномерное адресное пространство при размещении специализаций по одному начальному адресу

Обобщение А, построенное с использованием объединения на основе общего ресурса (окончание)

Наличие общего ресурса для нескольких абстракций позволяет использовать такое обобщение в **двух различных целях**:

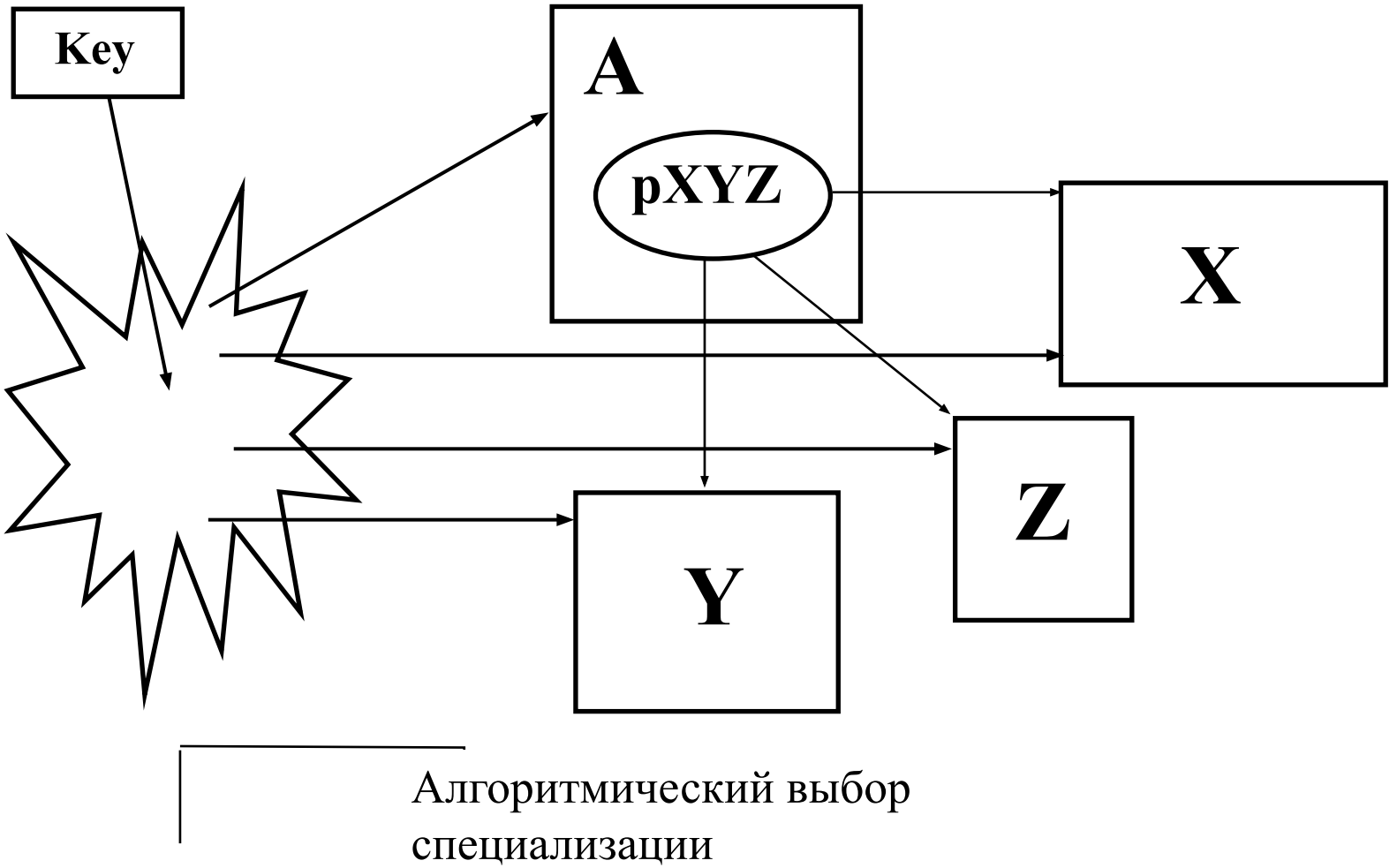
1. **Хранение одного из альтернативных объектов**. Выделенное пространство предоставляется экземпляру абстракции только одного типа, который хранит свое значение, никоим образом по семантике не пересекающееся с семантикой альтернативных объектов.
2. **Использование различных трактовок одного и того же пространства ресурсов**. В частности, для математических операций целое число длиной два байта может интерпретироваться как единое слово, для операций сохранения-восстановления в двоичном формате, как младший и старший байты. Для операций сравнения и сдвига бывает необходимо отделять знаковый бит от значения. Для каждого из этих случаев возможна своя специализация. А вместе они могут формировать обобщение.

Альтернативное связывание

Предполагает независимое размещение специализаций в рассматриваемом обобщении.

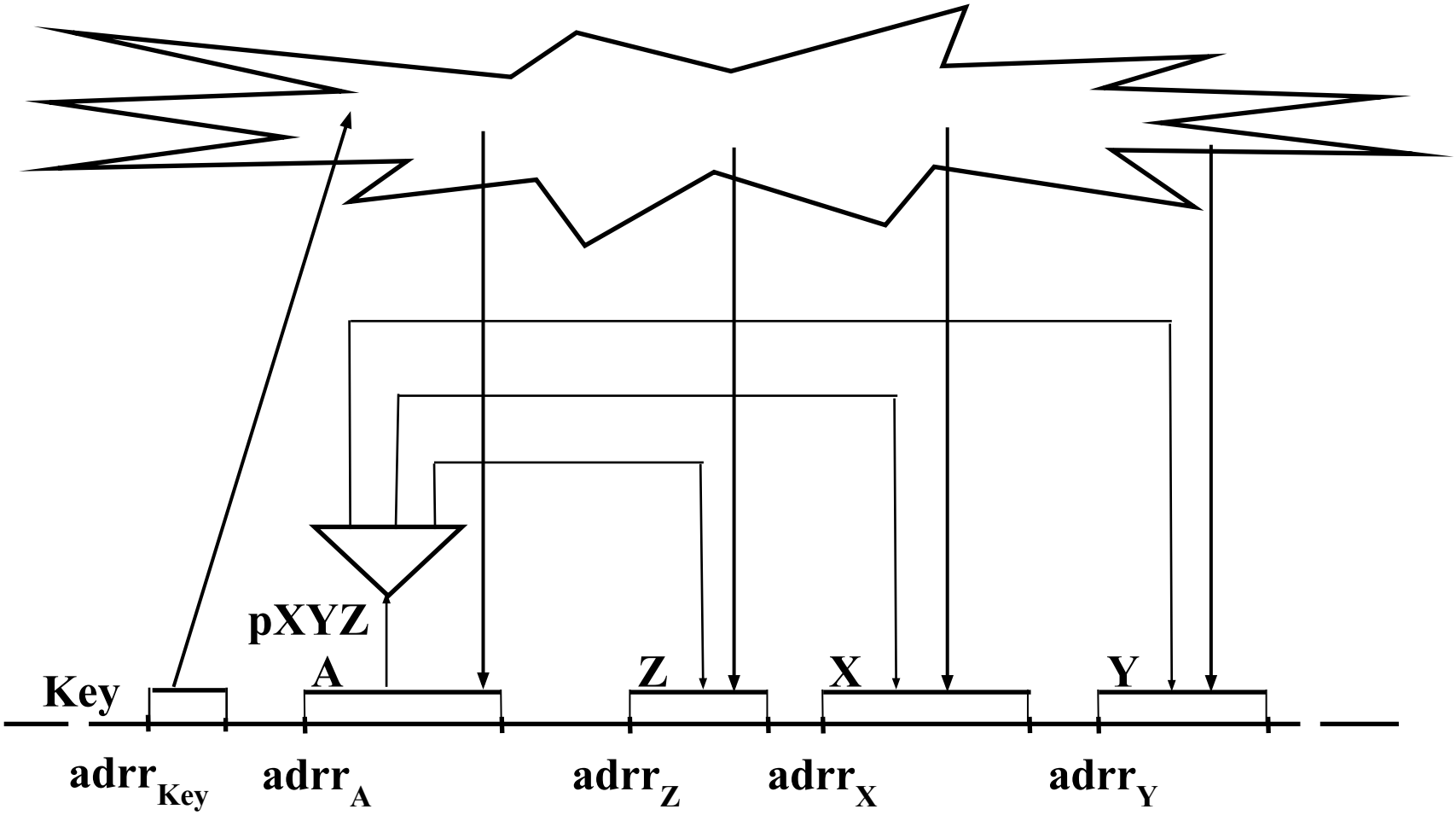
Использование ссылки или указателя обеспечивает доступ только к одному **избранному** объекту. Как и при агрегировании, альтернативное связывание обычно осуществляется алгоритмически. Выполнение алгоритма связывания, происходит для отдельного экземпляра обобщения только один раз.

Одновременно с выбором специализации осуществляется и установка ключа, указывающего признак специализации. Дальнейшая работа с обобщением осуществляется на основе предварительного анализа значения ключа. Это значение позволяет семантически правильно обработать специализацию через "обезличенный" указатель.



а) условное изображение на плоскости

Обобщение A, построенное с использованием динамического вариантного связывания (начало).



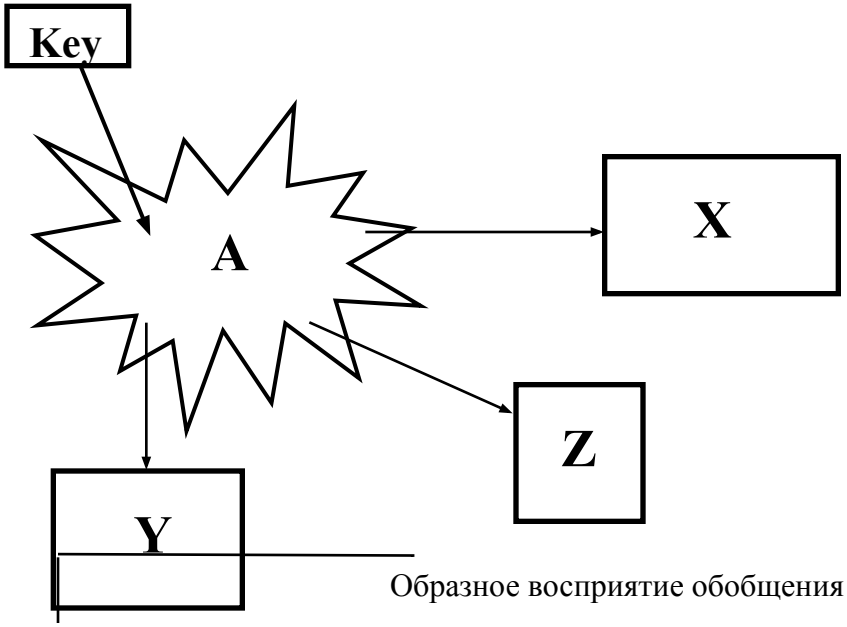
б) отображение на одномерное адресное пространство

Обобщение A, построенное с использованием динамического вариантного связывания (окончание).

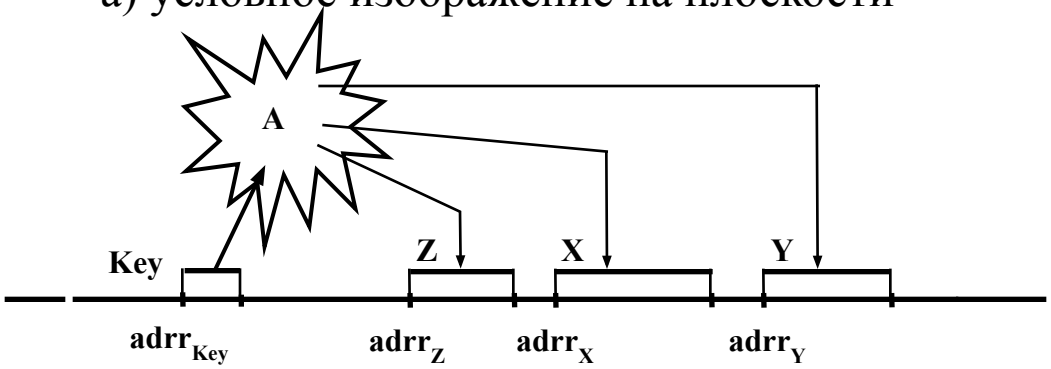
Образное обобщение

Как и образное агрегирование, связано с мысленными ассоциациями программиста. Используя ключевой признак, можно связать с отдельными специализациями подмножества разрозненных объектов программы, семантическая связь между которыми поддерживается на уровне алгоритма. При этом одни и те же программные объекты могут использоваться в альтернативных специализациях, имея при этом различную семантическую трактовку.

Как и в случае с образным агрегированием, такой подход к написанию программ является "пережитком прошлого". Достаточно часто он использовался и для экономии памяти, когда одни и те же переменные интерпретировались различным образом.



а) условное изображение на плоскости



б) отображение на одномерное адресное пространство

Обобщение A из элементов X, Y, Z, построенное на основе образного восприятия

Вариантное обобщение

Вариант **- основа обобщения данных** в **процедурном подходе**.

Обобщение, применяемое в процедурном подходе и построенное на основе варианта, назовем ***вариантным обобщением***.

С каждым вариантом связан набор специализаций обобщения, построенный на основе уже существующих абстракций (основ специализаций). Определим их как ***вариантные специализации***.

Обработка вариантных обобщений осуществляется **независимыми процедурами**, организующими доступ к внутренним переменным через экземпляр обобщения, получаемый, в качестве одного из **аргументов списка параметров**.

Процедуры, обрабатывающие специализации обобщений, могут создаваться независимо от обобщающей процедуры. Они могут использоваться различными обобщающими процедурами.

Каждая из таких процедур связана только со своими специализациями.

В дальнейшем специализированные процедуры, используемые в процедурном подходе, будут называться ***обработчиками вариантов***.

Процедуры, осуществляющие обработку всего вариантного обобщения, используют алгоритмический механизм анализа вариантов по ключевому параметру, содержащему признак текущей **вариантной специализации**. Алгоритм анализа обычно строится с использованием условных оператором или переключателей. Анализ осуществляется всякий раз, когда запускается процедура, и заключается в проверке ключа, задающего признак специализации обобщения. После определения специализации запускается соответствующий **обработчик варианта**. Обобщающая процедура, осуществляющая обработку вариантного обобщения, называется ***вариантной процедурой***.

Использование независимых вариантных процедур для создания кода ведет к ***централизации*** процесса обработки обобщений, выделя в нем отдельные задачи. ***Каждая из процедур обеспечивает решение одной из задач***. Процедуры, решающие разные задачи, совершенно не связаны друг с другом. Декомпозиция работ внутри вариантной процедуры осуществляется в соответствии с вариантной специализацией. ***Каждая из подзадач выполняется отдельным обработчиком варианта.***

Построение вариантного обобщения на основе общего ресурса

Рассмотрим создание обобщенной геометрической фигуры, используемой для представления прямоугольника или треугольника. Первоначально создаются абстракции данных, определяющие конкретные геометрические фигуры (основы специализаций).

```
//--------------------------------
// прямоугольник
struct rectangle {
    int x, y; // ширина, высота
};

//--------------------------------
-
// треугольник
struct triangle {
    int a, b, c; // стороны
};
```

С каждой из специализаций связывается набор обработчиков:

Процедура динамического создания прямоугольника:

```
rectangle *Create_rectangle(int x, int y)
```

Процедура инициализации уже созданного прямоугольника:

```
void Init(rectangle &r, int x, int y)
```

Процедура ввода параметров прямоугольника из входного потока:

```
void In(rectangle &r)
```

Процедура вывода параметров прямоугольника:

```
void Out(rectangle &r)
```

Процедура вычисления площади прямоугольника:

```
double Area(rectangle &r)
```

Процедура динамического создания треугольника:

```
triangle *Create_triangle
        (int a, int b, int c)
```

Процедура инициализации уже созданного треугольника:

```
void Init(triangle &t,
        int a, int b, int c)
```

Процедура ввода параметров треугольника из входного потока:

```
void In(triangle &t)
```

Процедура вывода параметров треугольника:

```
void Out(triangle &t)
```

Процедура вычисления площади треугольника:

```
double Area(triangle &t)
```

```
//-------------------------------------------------
// Процедуры, обеспечивающие работу с прямоугольником
//-------------------------------------------------

// Динамическое создание прямоугольника по двум сторонам
rectangle *Create_rectangle(int x, int y)
{
    rectangle *r = new rectangle;
    r->x = x;
    r->y = y;
    return r;
}

// Инициализация уже созданного прямоугольника
// по двум сторонам
void Init(rectangle &r, int x, int y)
{
    r.x = x;
    r.y = y;
}
```

```
// Ввод параметров прямоугольника
void In(rectangle &r)
{
    cout << "Input Rectangle: x, y =
";
    cin >> r.x >> r.y;
}

// Вывод параметров прямоугольника
void Out(rectangle &r)
{
    cout << "It is Rectangle: x = "
         << r.x << ", y = "
         << r.y << endl;
}

// Вычисление площади прямоугольника
double Area(rectangle &r)
{
    return r.x * r.y;
}
```

```
//-------------------------------------------------
// Процедуры, обеспечивающие работу с треугольником

// Динамическое создание треугольника по трем сторонам
triangle *Create_triangle(int a, int b, int
c)
{
    triangle *t = new triangle;
    t->a = a;
    t->b = b;
    t->c = c;
    return t;
}

// Инициализация уже созданного треугольника
// по трем сторонам
void Init(triangle &t, int a, int b, int c)
{
    t.a = a;
    t.b = b;
    t.c = c;
}
```

```
// Ввод параметров треугольника
void In(triangle &t)
{
    cout << "Input Triangle: a, b, c = ";
    cin >> t.a >> t.b >> t.c;
}

// Вывод параметров треугольника
void Out(triangle &t)
{
    cout << "It is Triangle: a = "
      << t.a << ", b = " << t.b
      << ", c = " << t.c << endl;
}

// Вычисление площади треугольника
double Area(triangle &t)
{
    // полупериметр
    double p = (t.a + t.b + t.c) / 2.0;
    return
        sqrt(p * (p-t.a) * (p-t.b) * (p-t.c));
}
```

Вариантное обобщение на основе прямого включения строится с применением объединения:

```
// Обобщение на основе
// разделяемого (общего) ресурса
union {
     rectangle r;
     triangle t;
};
```

Объединение включается в агрегат, который также содержит признаки специализаций и ключевую переменную, предназначенную для хранения текущего признака каждого экземпляра.

```
// структура, обобщающая все имеющиеся фигуры
  struct shape {
    // значения ключей для каждой из фигур
    enum key {RECTANGLE, TRIANGLE};
    key k; // ключ
    // Обобщение на основе разделяемого
    // (общего) ресурса
    union { // используем простейшую
реализацию
      rectangle r;
      triangle t;
  };
};
```

Вариантные процедуры

Динамическое создание обобщенной фигуры:

```
shape *Create_shape_rectangle(int x, int y)
shape *Create_shape_triangle
        (int a, int b, int c)
```

Инициализация обобщенной фигуры:

```
void Init_rectangle(shape &s, int x, int y)
void Init_triangle
        (shape &s, int a, int b, int c)
```

Создание обобщенной фигуры и ввод ее типа:

```
shape* In()
```

Вывод варианта, определяемого экземпляром обобщенной фигуры:

```
void Out(shape &s)
```

Вычисления площади заданного экземпляра вариантного обобщения:

```
double Area(shape &s)
```

```
//-----------------------------------------------------
// Процедуры, обеспечивающие работу с обобщенной фигурой
//-----------------------------------------------------

// Сигнатуры, необходимые обработчикам вариантов.
void Init(rectangle &r, int x, int y);
void Init(triangle &t, int a, int b, int c);
void In(rectangle &r);
void In(triangle &t);
void Out(rectangle &r);
void Out(triangle &t);
double Area(rectangle &r);
double Area(triangle &t);

// Динамическое создание обобщенного прямоугольника
shape *Create_shape_rectangle(int x, int y) {
    shape *s = new shape;
    s->k = shape::key::RECTANGLE;
    Init(s->r, x, y);
    return s;
}
```

```
// Инициализация обобщенного прямоугольника
void Init_rectangle(shape &s, int x, int y) {
    s.k = shape::key::RECTANGLE;
    Init(s.r, x, y);
}

// Динамическое создание обобщенного треугольника
shape *Create_shape_triangle(int a, int b, int c) {
    shape *s = new shape;
    s->k = shape::key::TRIANGLE;
    Init(s->t, a, b, c);
    return s;
}

// Инициализация обобщенного треугольника
void Init_triangle(shape &s, int a, int b, int c)
{
    s.k = shape::key::TRIANGLE;
    Init(s.t, a, b, c);
}
```

```
// Ввод параметров обобщенной фигуры из
// стандартного потока ввода
shape* In() {
    shape *sp;
    cout << "Input key: for Rectangle is 1, "
            "for Triangle is 2, else break: ";
    int k;
    cin >> k;
    switch(k) {
    case 1:
      sp = new shape;
      sp->k = shape::key::RECTANGLE;
      In(sp->r);
      return sp;
    case 2:
      sp = new shape;
      sp->k = shape::key::TRIANGLE;
      In(sp->t);
      return sp;
    default:
      return 0;
    }
}
```

```
// Вывод параметров текущей фигуры в стандартный
// поток вывода
void Out(shape &s) {
    switch(s.k) {
    case shape::key::RECTANGLE:
      Out(s.r);
      break;
    case shape::key::TRIANGLE:
      Out(s.t);
      break;
    default:
      cout << "Incorrect figure!" << endl;
    }
}

// Нахождение площади обобщенной фигуры
double Area(shape &s) {
    switch(s.k) {
    case shape::key::RECTANGLE:
      return Area(s.r);
    case shape::key::TRIANGLE:
      return Area(s.t);
    default:
      return 0.0;
    }
}
```

Полная версия примера в pp_examp1.zip.

Резюме

Вариантные обобщения на основе общего ресурса формируются при написании программы и распознаются во время трансляции.

Это позволяет заранее распределить память и обеспечить ***быстрый и непосредственный доступ*** к отдельным экземплярам.

К недостаткам можно отнести неэффективное использование пространства памяти при **различных размерах** специализаций.

Процесс построения языковой структуры вариантного обобщения совпадает по направленности с ***восходящим проектированием***.

Это никоим образом не говорит, что при процедурном подходе проблематично применять нисходящую разработку. Проектирование, **при статических вариантных обобщениях**, ведется таким образом, что в начале осуществляется полная проработка всей иерархии обобщения, а уж только потом можно приступать к его кодированию.

Возможно, что эта особенность кодирования вариантных обобщений в какой-то мере послужила популярности водопадной модели, оказавшейся малоэффективной при разработке сложных программ.

Построение вариантного обобщения на основе альтернативного связывания

Позволяет обойтись без дополнительных обобщающих конструкций, что обеспечивает большую гибкость при эволюционном наращивании процедурной программы.
Можно посмотреть, как изменится метод решения задачи, если использование общего ресурса заменить вариантным связыванием.
Сохраним существующие специализации без изменения, но заменим обобщение:

Полная версия примера в pp_examp1b.zip.

```
// структура, обобщающая все имеющиеся фигуры
   struct shape {
      // значения ключей для каждой из фигур
      enum key {RECTANGLE, TRIANGLE};
      key k; // ключ
      // используемые альтернативные указатели
      union { // используем простейшую
реализацию
         rectangle *pr;
         triangle *pt;
      };
   };
```

Построение вариантного обобщения на основе образного восприятия

Для иллюстрации образного восприятия вариантов, введем в нашу программу следующие изменения:

- Создадим контейнер, используя указатели на любой тип данных (***void***). Будем подразумевать, что каждый элемент контейнера указывает на одну из разработанных фигур.
- Для идентификации специализаций, вместо перечислимого типа, используем целые числа, образно ассоциируя их с соответствующими фигурами (1 - прямоугольник, 2 - треугольник). Тогда, появление новой фигуры просто сведется к образной ассоциации с еще одним числом.

Полная версия примера в pp_examp1c.zip.

```
struct container {
    enum {max_len = 100}; // максимальная длина
    int len; // текущая длина
    // Контейнер обазных указателей на специализации,
    // построенные на основе распределенного ключа
    void* cont[max_len];
};

//-----------------------------------------------
// Использование образного восприятия обобщения на основе
// ключа, разнесенного по отдельным дополнительным
// специализациям

// шаблонная структура, обобщающая все имеющиеся фигуры
struct shape {
    // значения ключей для каждой из фигур
    int k;
    // образый ключ устанавливается для вводимой фигуры.
    // Можно легко ошибиться, но гибкость - прекрасная!
    // А связь с фигурами отсутствует!
    // Предполагается, что они пропишутся отдельно
    // и будут связаны
    // через образное наложение одинаковых структур памяти
  };
```

```
// структура, обобщающая прямоугольник
struct r_shape {
    // значения ключей для каждой из фигур
    int k; // образый ключ  = 1
    // А здесь будет храниться
прямоугольник
    rectangle r;
};

// структура, обобщающая треугольнк
struct t_shape {
    // значения ключей для каждой из фигур
    int k; // образый ключ  = 2
    // А здесь будет храниться треугольник
    triangle t;
};
```

```
void Out(rectangle &r);
void Out(triangle &t);

void Out(shape *ps)
{
  switch(ps->k) {
  case 1: // прямоугольник
    Out(((r_shape*)ps)->r);
    break;
  case 2: // треугольник
    Out(((t_shape*)ps)->t);
    break;
  default:
    cout << "Incorrect figure!" <<
endl;
  }
}
```

Пример программы на языке программирования Оберон (процедурный стиль)

Язык содержит расширяемые типы данных, обеспечивает проверку типов во время выполнения, в нем реализована автоматическая сборка мусора.
Ограничения на работу с указателями и использование только расширяемых типов (отсутствуют вариантные записи предшественников) позволили сократит число вариантов написания программы.

```
(*****************************************************
figure.o: содержит модуль figure с процедурами и типами,
осуществляющими обработку обобщенных геометрических фигур
*****************************************************)

MODULE figure;

TYPE
   (* Структуры данных описывающие обобщенную
      геометрическую фигуру *)

   (* Указатель на обобщенную геометрическую фигуру *)
  Pfig* = POINTER TO fig;

   (* Запись, определяющая структуру обобщенной
      геометрической фигуры *)
  fig* = RECORD
     (* запись является базовой для всех прочих *)
     (* поля отсутствуют *)
  END;

END figure.
```

```
(**********************************************
rectangle.o: содержит модуль rectangle с процедурами и
типами, осуществляющими обработку прямоугольников
**********************************************)
MODULE rectangle;
IMPORT In, Out, FileIO, figure;
TYPE
   (* Структуры данных описывающие прямоугольник *)
   (* Указатель на прямоугольник *)
   Prect* = POINTER TO rect;
   (* Запись, определяющая структуру прямоугольника *)
   rect* = RECORD(figure.fig)
     x*, y* : INTEGER (* стороны прямоугольника*)
   END;
(* Процедура динамического создания и инициализации
нового прямоугольника *)
PROCEDURE New*(x, y: INTEGER): Prect;
VAR tmp : Prect;
BEGIN
     NEW(tmp);
     tmp^.x := x;
     tmp^.y := y;
     RETURN tmp
END New;
```

```
(* Процедура инициализации уже существующего
прямоугольника *)
PROCEDURE Init*(VAR r: rect; x, y: INTEGER);
BEGIN
    r.x := x;
    r.y := y;
END Init;

(* Процедура вычисления периметра прямоугольника *)
PROCEDURE Perimeter*(VAR r: rect) : REAL;
VAR
BEGIN
  RETURN (r.x + r.y) * 2
END Perimeter;

(* Процедура ввода прямоугольника из входного потока *)
PROCEDURE Input*(VAR r: rect);
BEGIN
  Out.String("x = "); In.Int(r.x);
  Out.String("y = "); In.Int(r.y);
END Input;
```

```
(* Процедура вывода *)
PROCEDURE Output*(VAR r: rect);
BEGIN
  Out.String("Rectangle: x = "); Out.Int(r.x, 0);
  Out.String(", y = "); Out.Int(r.y, 0);
  Out.String(",   perimemter = "); Out.Real(Perimeter(r),
0);
  Out.Ln;
END Output;

(* Процедура Заполнения прямоугольника из файла *)
PROCEDURE FileInput*
     (VAR inFile : FileIO.TFile; VAR r: rect);
VAR
  f : INTEGER;
BEGIN
  f := inFile.ReadInt(r.x);
  f := inFile.ReadInt(r.y);
END FileInput;
```

```
(* Процедура вывода в файл, использующая передачу
параметра-переменной *)
PROCEDURE FileOutput*
            (VAR outFile : FileIO.TFile; VAR r: rect);
VAR
  flag : INTEGER;
BEGIN
  flag := outFile.WriteString("Rectangle: x = ");
  flag := outFile.WriteInt(r.x, 7);
  flag := outFile.WriteString(", y = ");
  flag := outFile.WriteInt(r.y, 7);
  flag := outFile.WriteString(",   perimemter = ");
  flag := outFile.WriteReal(Perimeter(r), 7);
  flag := outFile.Ln();
END FileOutput;

END rectangle.
```

Точно также устроен модуль triangle, обеспечивающий обработку треугольника…

```
(**********************************************
figureproc.o: содержит модуль figureProc с процедурами,
осуществляющими обработку специализаций, связанных с
обобщенной геометрической фигурой. Эти процедуры лучше
разместить после того, как будут созданы специализации, т.
к. их структуру необходимо знать во время обработки.
**********************************************)
MODULE figureProc;
IMPORT In, Out, FileIO, figure, rectangle, triangle;

(* Процедура, вычисляющая переметр обобщенной фигуры *)
PROCEDURE Perimeter*(VAR f: figure.fig) : REAL;
VAR
  p : REAL;
BEGIN
  WITH f: rectangle.rect DO p := rectangle.Perimeter(f)
     | f: triangle.trian DO p := triangle.Perimeter(f)
  ELSE p := 0
  END;
  RETURN p

END Perimeter;
```

```
(* Процедура, осуществляющая создание и ввод одной из
специализаций после предварительного анализа признака
вводимой фигуры *)
PROCEDURE Input*() : figure.Pfig;
VAR
  tag : INTEGER;
  pf : figure.Pfig;
  pr : rectangle.Prect;
  pt : triangle.Ptrian;
BEGIN
  Out.String("Input figure tag (1 - rectangle; 2 -
triangle; else - NIL): ");
  Out.Ln;
  In.Int(tag);
  IF tag = 1 THEN
    NEW(pr); rectangle.Input(pr^); pf := pr;
  ELSIF tag = 2 THEN
    NEW(pt); triangle.Input(pt^); pf := pt;
  ELSE
    pf := NIL;
  END;
  RETURN pf

END Input;
```

```
(* Процедура, осуществляющая вывод специализаций через
параметр-переменную *)
PROCEDURE Output*(VAR f: figure.fig);
BEGIN
  WITH f: rectangle.rect DO rectangle.Output(f)
     | f: triangle.trian   DO triangle.Output(f)
  ELSE Out.String("Invalid figure"); Out.Ln
  END
END Output;
```

```
(* Процедура Заполнения фигуры из файла *)
PROCEDURE FileInput*(VAR inFile : FileIO.TFile) :
figure.Pfig;
VAR
  flag : INTEGER;
  tag : INTEGER;
  pf : figure.Pfig;
  pr : rectangle.Prect;
  pt : triangle.Ptrian;
BEGIN
  flag := inFile.ReadInt(tag);
  IF tag = 1 THEN
    NEW(pr); rectangle.FileInput(inFile, pr^); pf := pr;
  ELSIF tag = 2 THEN
    NEW(pt); triangle.FileInput(inFile, pt^); pf := pt;
  ELSE
    pf := NIL;
  END;
  RETURN pf
END FileInput;
```

```
(* Процедура вывода в файл, использующая передачу
параметра-переменной *)
PROCEDURE FileOutput*(VAR outFile : FileIO.TFile; VAR f:
figure.fig);
VAR
  flag : INTEGER;
BEGIN
  WITH f: rectangle.rect DO rectangle.FileOutput(outFile,
f)
     | f: triangle.trian DO triangle.FileOutput(outFile,
f)
  ELSE
    flag := outFile.WriteString("Invalid figure");
    flag := outFile.Ln();
  END
END FileOutput;

END figureProc.
```

```
(* test.o: *)
MODULE figureTest;
IMPORT In, Out, FileIO, Console,
       figure, rectangle, triangle, figureProc,
container;
VAR
  flag : INTEGER;
  c:  container.cont;
  pf: figure.Pfig;
  inFile, outFile : FileIO.TFile;
  str : ARRAY 100 OF CHAR;
BEGIN
  In.Open;
  (* Открытие файла, используемого для чтения исходных
     данных *)
  Out.String("Input name of source file: ");
  In.String(str);
  inFile.Assign(str);
  inFile.Reset;
  (* Открытие файла для записи результатов *)
  Out.String("Input name of output file: ");
  In.String(str);
  outFile.Assign(str);
  outFile.Rewrite;
```

```
  (* Предварительная инициализаци контейнера *)
  container.Init(c);
  (* Тестовый вывод содержимого контейнера до работы *)
  Out.String("Initialed Container is:");
  container.Output(c);
  Out.Ln;
  (* Чтение из файла в контейнер *)
  flag := container.FileInput(inFile, c);
  Out.String("Result Container is:"); Out.Ln;
  container.Output(c);
  flag := outFile.WriteString("Result Container is:");
  flag := outFile.Ln();
  container.FileOutput(outFile, c);

  (* Закрытие используемых файлов *)
  inFile.Close;
  outFile.Close;

  (* Закрытие используемых файлов *)
  inFile.Close;
  outFile.Close;

END figureTest.
```

Отличительные особенности объектно-ориентированной альтернативы

Основной спецификой ОО подхода является возможность группирования процедур вокруг обрабатываемых ими данных. Она, в сочетании с динамическим связыванием объектов, обеспечивает возможность выбора альтернатив **без дополнительного анализа признака специализации**.

При процедурном же подходе анализ принадлежности артефактов к определенному типу обычно осуществляется внутри процедур, что ведет к использованию алгоритмических методов обработки альтернатив.

При ООП методы классов **напрямую** привязываются к обрабатываемым данным. Эта привязка осуществляется во написания кода (в декларативной) манере **посредством инструментальной поддержки** механизма **виртуализации**.

В программе не требуется создание экземпляров конкретных объектов, но заготовки отношений между типами данных и процедурами уже хранятся в виде векторов отношений, в каждом из классов $\mathbf{C}_i \in \mathbf{C}$:

$$C_1 = (t_1, f_{11}, f_{11}, f_{m1}),$$

$$C_2 = (t_2, f_{12}, f_{22}, f_{m2}),$$

Поэтому, когда в ходе выполнения программы формируется экземпляр объекта $\mathbf{D_i}$ с данными типа $\mathbf{t_i}$, создается и экземпляр отношения $\mathbf{C_i}$, которое содержит и процедуры $\mathbf{f_{1n}}$, $\mathbf{f_{2n}}$, $\mathbf{f_{mn}}$, обеспечивающие требуемую обработку созданного объекта.

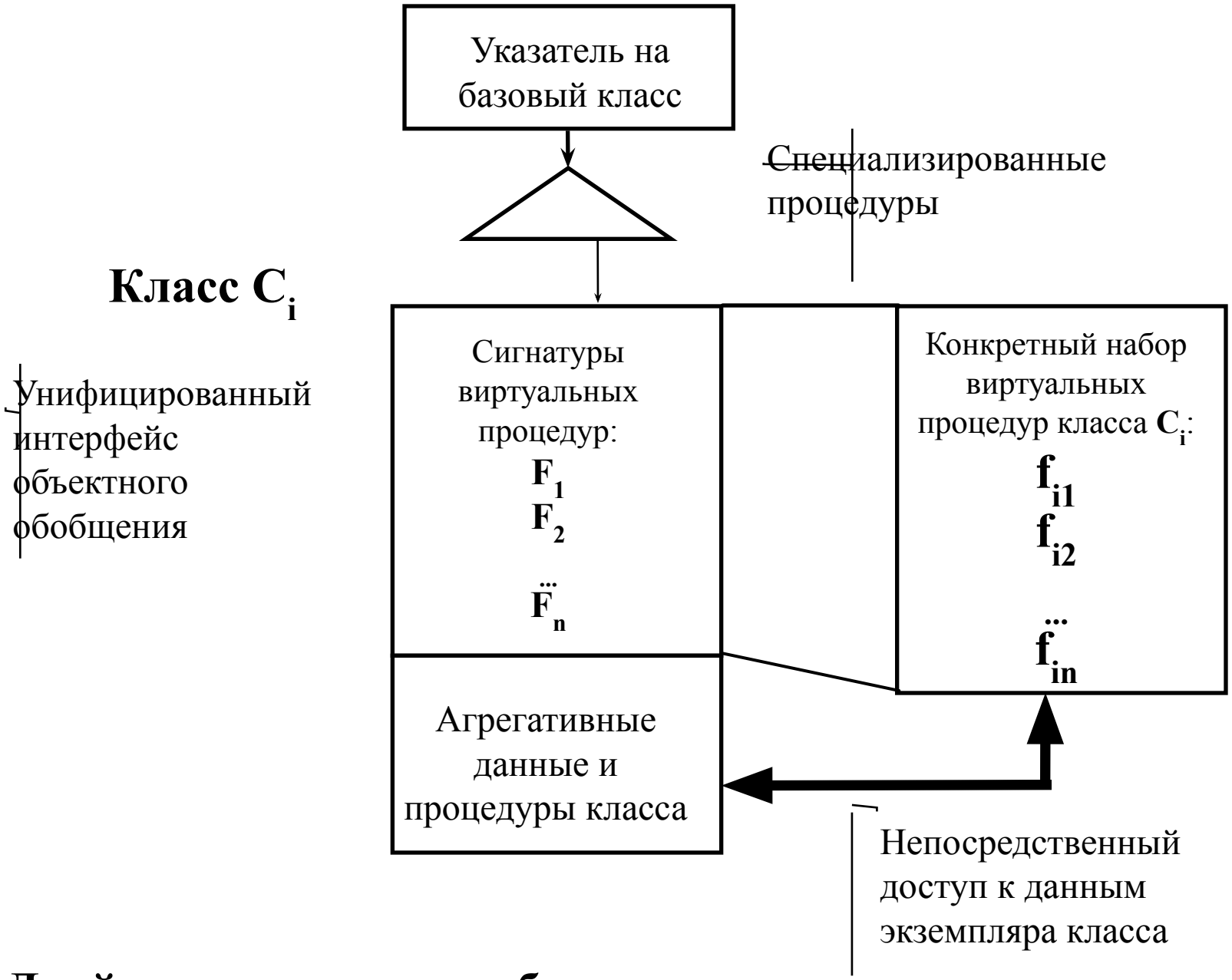
Однозначная зависимость между данными и процедурами их обработки, сгруппированными в одном объекте, позволяет каждому экземпляру класса в любой момент обратиться к его же методу через существующее отношение **без дополнительного анализа признака** при выборе специализированной процедуры.

Достоинства ОО альтернатив проявляются в «физике» конструкции, а не в преимуществах «идеологии».

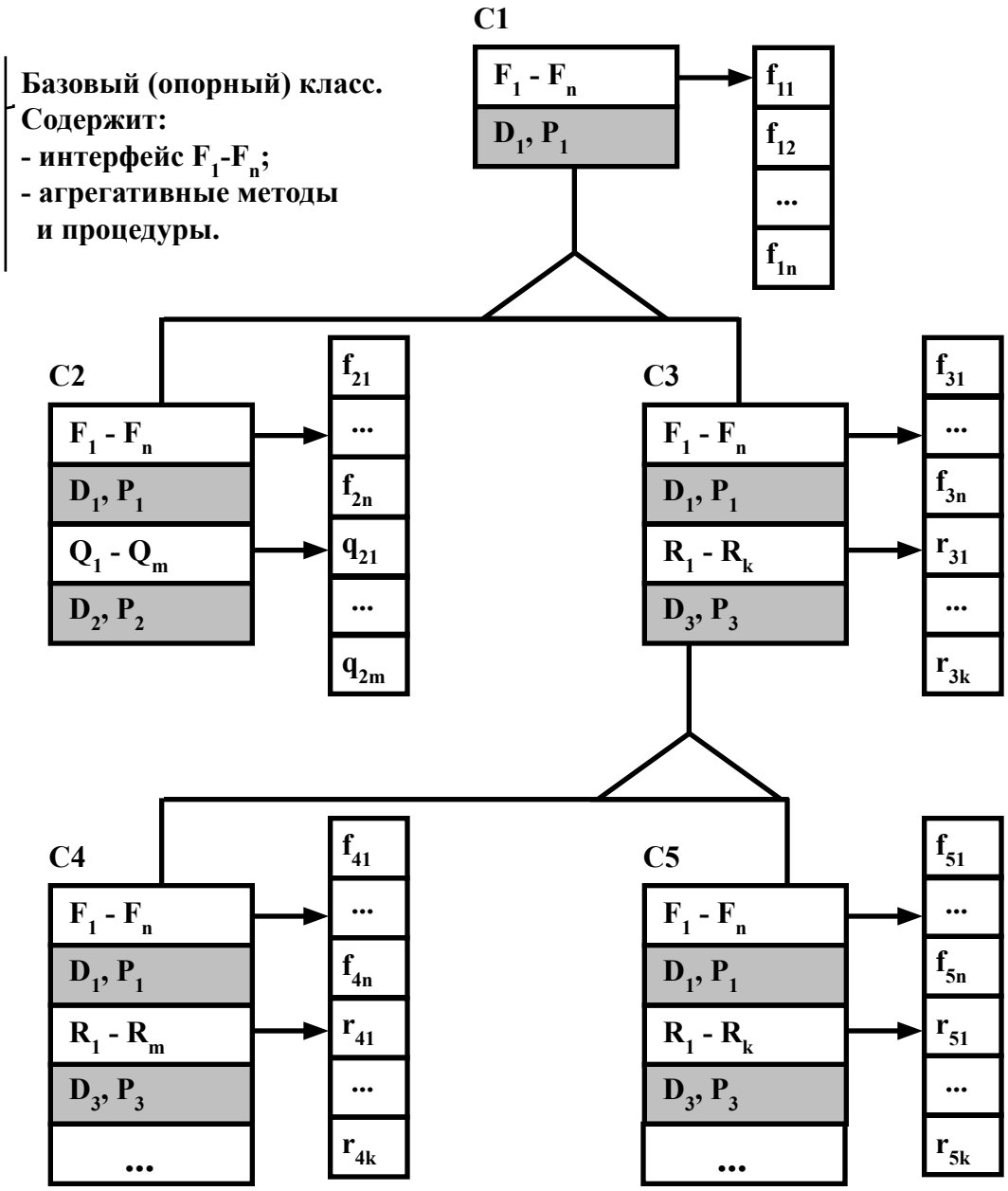
Объектное обобщение

При ОО методе организации альтернатив, не требуется анализа признаков специализаций. Используется зависимость двух статически связанных агрегатов, динамически подключаемых к указателю на альтернативу.

Первый агрегат, являясь экземпляром производного или базового класса, содержит данные и процедуры, непосредственно размещенные в его теле. Для различных производных классов состав этих данных и процедур может отличаться, так как не они определяют облик формируемого обобщения.



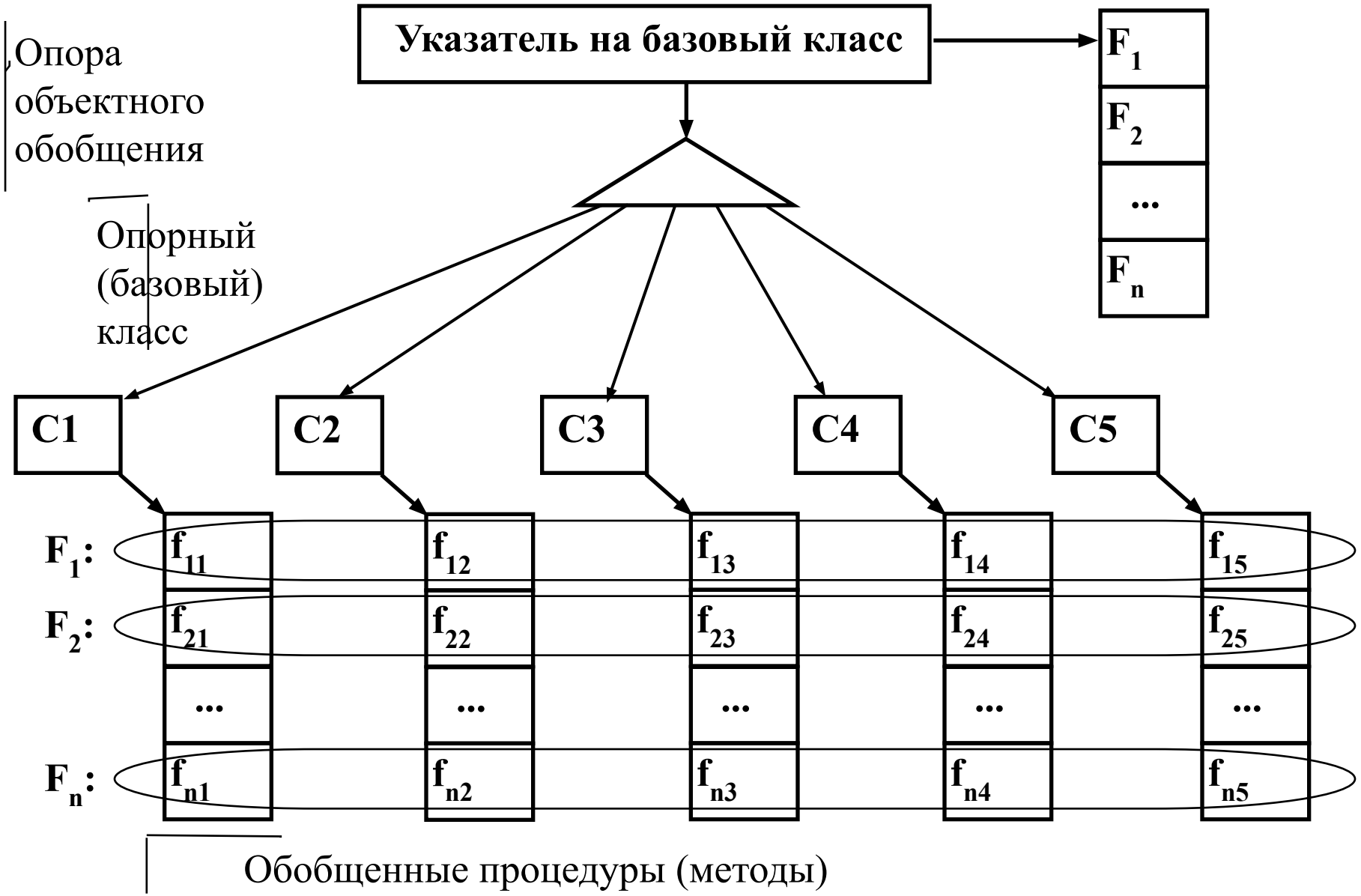
Двойное связывание, обеспечивающее трактовку класса как альтернативы объектного обобщения.



Иерархия классов, построенная с применением наследования и виртуализации

Основная задача выбора альтернативы заключается в вызове соответствующей **связанной процедуры**.
Она решается следующим образом. Через указатель на экземпляр базового класса, обеспечивающий динамическое связывание с любым экземпляром производного класса, осуществляется доступ к интерфейсу объекта.

Назовем такой указатель ***опорой объектного обобщения***.



Динамическое подключение альтернатив в объектном обобщении

Таким образом, в объектном обобщении **не нужен специализированный ключ**, обеспечивающий анализ существующей альтернативы, так как решение проблемы безусловного выбора осуществляется через использование внутренней структуры объекта.

Вместе с тем, можно считать, что **такой ключ у каждого класса существует**. Им является совокупность всех **специализированных процедур**, подключаемых через статически связанный агрегат.

Пример использования объектного обобщения

Полная версия примера в oop_examp1.zip

```
// Класс, обобщающий все имеющиеся фигуры.
// Является абстрактным, обеспечивая, тем самым,
// проверку интерфейса

class shape {
public:
  virtual void In() = 0;  // ввод данных
  virtual void Out() = 0; // вывод данных
  virtual double Area() = 0; // площадь
};
```

Опорные процедуры:
virtual void In()
virtual void Out()
virtual double Area()

Все эти виртуальные абстракции начинают обрастать конкретным содержанием только при реализации конкретных специализаций. При этом, разработка специализаций может осуществляться независимо и поэтапно, так как между ними отсутствует какая-либо взаимосвязь. Поэтому, в начале разработаем прямоугольник и его методы.

```
// прямоугольник
class rectangle: public shape {
    int x, y; // ширина, высота
public:
  // переопределяем интерфейс класса
  void In();     // ввод данных из стандартного потока
  void Out();    // вывод данных в стандартный поток
  double Area(); // вычисление площади фигуры
  rectangle(int _x, int _y);
  rectangle() {} // создание без инициализации.
};
```

```
// Динамическое создание прямоугольника по двум сторонам
rectangle::rectangle(int _x, int _y): x(_x), y(_y) {}
// Ввод параметров прямоугольника
void rectangle::In() {
    cout << "Input Rectangle: x, y = ";
    cin >> x >> y;
}
// Вывод параметров прямоугольника
void rectangle::Out() {
    cout << "It is Rectangle: x = " << x
         << ", y = " << y << endl;
}
// Вычисление площади прямоугольника
double rectangle::Area() {
    return x * y;
}
```

```
class triangle: public shape {
    int a, b, c; // стороны
public:
  void In();     // ввод данных из стандартного потока
  void Out();    // вывод данных в стандартный поток
  double Area(); // вычисление площади фигуры
  triangle(int _a, int _b, int _c); // создание
  triangle() {} // создание без инициализации.
};
```

```
// Инициализация по трем сторонам
triangle::triangle(int _a, int _b, int _c):
  a(_a), b(_b), c(_c) {}

// Ввод параметров треугольника
void triangle::In() {
    cout << "Input Triangle: a, b, c = ";
    cin >> a >> b >> c;
}

// Вывод параметров треугольника
void triangle::Out() {
    cout << "It is Triangle: a = " << a << ", b = "
         << b << ", c = " << c << endl;
}

// Вычисление площади треугольника
double triangle::Area() {
    double p = (a + b + c) / 2.0; // полупериметр
    return sqrt(p * (p-a) * (p-b) * (p-c));
}
```

Пример программы на языке программирования Оберон 2 (ОО стиль)

Язык дополнительно содержит процедуры, связанные с типом, что обеспечивает полиморфизм процедур, аналогичный полиморфизму методов ОО языков. Процедуры при этом находятся вне типов (но в одном с ним модуле).

```
(*********************************************************
figure.o: содержит модуль figure с процедурами и типами,
осуществляющими обработку обобщенных геометрических фигур
*********************************************************)
MODULE figure;
IMPORT Out, FileIO;
TYPE
  (* Указатель на обобщенную геометрическую фигуру *)
  Pfig* = POINTER TO fig;

  (* Запись, определяющая структуру обобщенной
     геометрической фигуры *)
  fig* = RECORD
    (* запись является базовой для всех прочих *)
    (* поля отсутствуют *)
  END;
```

```
(* Процедура, вычисляющая переметр обобщенной фигуры *)
PROCEDURE (VAR f: fig) Perimeter* (): REAL;
BEGIN
  RETURN 0
END Perimeter;

(* Процедура, осуществляющая вывод обобщенной фигуры *)
PROCEDURE (VAR f: fig) Output*;
BEGIN
  Out.String("Invalid figure"); Out.Ln
END Output;

(* Процедура вывода обобщенной фигуры в файл *)
PROCEDURE (VAR f: fig) FileOutput*
                              (VAR outFile : FileIO.TFile);
VAR
  flag : INTEGER;
BEGIN
    flag := outFile.WriteString("Invalid figure");
    flag := outFile.Ln();
END FileOutput;

END figure.
```

```
(*****************************************************
rectangle.o: содержит модуль rectangle с процедурами и
типами, осуществляющими обработку прямоугольников
*****************************************************)
MODULE rectangle;
IMPORT In, Out, FileIO, figure;
TYPE
   (* Указатель на прямоугольник *)
   Prect* = POINTER TO rect;
   (* Запись, определяющая структуру прямоугольника *)
   rect* = RECORD(figure.fig)
     x*, y* : INTEGER (* стороны прямоугольника*)
   END;

(* Процедура динамического создания и инициализации
нового прямоугольника *)
PROCEDURE New*(x, y: INTEGER): Prect;
VAR tmp : Prect;
BEGIN
     NEW(tmp);
     tmp^.x := x;
     tmp^.y := y;
     RETURN tmp
END New;
```

```
(* Процедура инициализации уже существующего
прямоугольника *)
PROCEDURE  (VAR r: rect) Init*(x, y: INTEGER);
BEGIN
    r.x := x;
    r.y := y;
END Init;

(* Процедура вычисления периметра прямоугольника -
наследника фигуры *)
PROCEDURE (VAR r: rect) Perimeter* () : REAL;
VAR
BEGIN
  RETURN (r.x + r.y) * 2
END Perimeter;

(* Процедура ввода прямоугольника из входного потока *)
PROCEDURE (VAR r: rect) Input*;
BEGIN
  Out.String("x = "); In.Int(r.x);
  Out.String("y = "); In.Int(r.y);
END Input;
```

```
(* Процедура вывода прямоугольника - наследника фигуры *)
PROCEDURE (VAR r: rect) Output*;
BEGIN
  Out.String("Rectangle: x = "); Out.Int(r.x, 0);
  Out.String(", y = "); Out.Int(r.y, 0);
  Out.String(",   perimemter = ");
Out.Real(r.Perimeter(), 0);
  Out.Ln;
END Output;

(* Процедура Заполнения прямоугольника из файла *)
PROCEDURE (VAR r: rect) FileInput*(VAR inFile :
FileIO.TFile);
VAR
  f : INTEGER;
BEGIN
  f := inFile.ReadInt(r.x);
  f := inFile.ReadInt(r.y);
END FileInput;
```

```
(* Процедура вывода в файл прямоугольника - наследника
фигуры *)
PROCEDURE (VAR r: rect) FileOutput*(VAR outFile :
FileIO.TFile);
VAR
  flag : INTEGER;
BEGIN
  flag := outFile.WriteString("Rectangle: x = ");
  flag := outFile.WriteInt(r.x, 7);
  flag := outFile.WriteString(", y = ");
  flag := outFile.WriteInt(r.y, 7);
  flag := outFile.WriteString(",   perimemter = ");
  flag := outFile.WriteReal(r.Perimeter(), 7);
  flag := outFile.Ln();
END FileOutput;

END rectangle.
```

Точно также устроен модуль triangle, обеспечивающий обработку треугольника…

```
(**********************************************
figureproc.o:
**********************************************)
MODULE figureProc;
IMPORT In, Out, FileIO, figure, rectangle, triangle;
(* Создание и ввод одной из специализаций  после
предварительного анализа признака вводимой фигуры *)
PROCEDURE Input*() : figure.Pfig;
VAR
  tag : INTEGER;
  pf : figure.Pfig;
  pr : rectangle.Prect;
  pt : triangle.Ptrian;
BEGIN
  Out.String("Input figure tag (1 - rectangle; 2 -
triangle; else - NIL): ");
  Out.Ln;
  In.Int(tag);
  IF tag = 1 THEN
    NEW(pr); pr.Input; pf := pr;
  ELSIF tag = 2 THEN
    NEW(pt); pt.Input; pf := pt;
  ELSE
    pf := NIL;
  END;
  RETURN pf
END Input;
```

```
(* Процедура заполнения обобщенной фигуры из файла *)
PROCEDURE FileInput*
    (VAR inFile : FileIO.TFile) : figure.Pfig;
VAR
   flag : INTEGER;
   tag : INTEGER;
   pf : figure.Pfig;
   pr : rectangle.Prect;
   pt : triangle.Ptrian;
BEGIN
   flag := inFile.ReadInt(tag);

   IF tag = 1 THEN
     NEW(pr); pr.FileInput(inFile); pf := pr;
   ELSIF tag = 2 THEN
     NEW(pt); pt.FileInput(inFile); pf := pt;
   ELSE
     pf := NIL;
   END;
   RETURN pf

END FileInput;

END figureProc.
```

Отличие методов обобщения

Основным неоспоримым **достоинством** объектного обобщения является поддержка **быстрого и безболезненного**, для уже написанного кода, **добавления новых альтернатив** в существующие обобщающие процедуры. Там, где процедурный подход ведет к поиску и редактированию фрагментов программы, ООП довольствуется только созданием и легкой притиркой новых классов.

Рассмотрим добавление круга в уже написанную программу.

Полная версия примера в pp_examp1_1.zip

```
//-----------------------------------------
// структура, обобщающая все имеющиеся фигуры
// Изменилась, в связи с добавлением круга
struct shape {
     // значения ключей для каждой из фигур
     enum key {RECTANGLE, TRIANGLE, CIRCLE};
        // добавился признак круга
     key k; // ключ
     // используемые альтернативы
     union { // используем простейшую
реализацию
        rectangle r;
        triangle t;
        circle c;  // добавился круг
     };
```

```
// Ввод параметров обобщенной фигуры
// Изменилась из-за добавления круга
shape* In() {
     shape *sp;
     // Изменилась подсказка из-за добавления круга
     cout <<
       "Input key: for Rectangle is 1, for Triangle is 2,"
       " for Circle is 3 else break: ";
     int k;      cin >> k;
     switch(k) {
     case 1:
       sp = new shape;
       sp->k = shape::key::RECTANGLE;
       In(sp->r);       return sp;
     case 2:
       sp = new shape;
       sp->k = shape::key::TRIANGLE;
       In(sp->t);       return sp;
     case 3: // добавился ввод круга
       sp = new shape;
       sp->k = shape::key::CIRCLE;
       In(sp->c);
       return sp;
     default:       return 0;
     }
}
```

```
// Вывод параметров текущей фигуры
// Изменилась из-за добавления вывода круга
void Out(shape &s){
     switch(s.k) {
     case shape::key::RECTANGLE:
       Out(s.r);       break;
     case shape::key::TRIANGLE:
       Out(s.t);       break;
     case shape::key::CIRCLE: // добавился вывод круга
       Out(s.c);       break;
     default:
       cout << "Incorrect figure!" << endl;
     }
}
// Нахождение площади обобщенной фигуры
// Изменилась в связи с добавлением круга
double Area(shape &s){
     switch(s.k) {
     case shape::key::RECTANGLE:
       return Area(s.r);
     case shape::key::TRIANGLE:
       return Area(s.t);
     case shape::key::CIRCLE: // добавился круг
       return Area(s.c);
     default:       return 0.0;
     }
}
```

При разработке же больших программных систем обработка обобщений может осуществляться не одной сотней процедур, каждую из которых потребуется изменить.

А изменение написанного кода всегда связано с риском внести **дополнительные ошибки**. Кроме этого, изменения, связанные с добавлением сведений о новой специализаций могут затронуть различные единицы компиляции (без изменения располагаемых в них программных объектов), что тоже ведет к дополнительным затратам.

ОО подход, в аналогичной ситуации, позволяет провести добавление круга практически без изменений уже написанного кода.

При этом, все добавления, связанные с новой фигурой могут осуществляться во вновь создаваемых единицах компиляции. Процесс добавления новой фигуры полностью аналогичен разработке уже созданных и не требует специальных комментариев. В рассматриваемом случае необходимо только изменить процедуру, обеспечивающую ввод новой фигуры.

Хотя, и здесь можно было бы поступить более тонко.

Полная версия примера в oop_examp1_1.zip

```
// Ввод параметров обобщенной фигуры
// Изменяется в связи с добавлением круга
shape* In() {
     shape *sp;
     // Изменяется подсказка
     cout << "Input key: for Rectangle is 1,
       " for Triangle is 2,"
       " for Circle is 3, else break: ";
     int k; cin >> k;
     switch(k) {
     case 1:
       sp = new rectangle; sp->In(); return sp;
     case 2:
       sp = new triangle;  sp->In(); return sp;
     case 3: // добавлен ввод круга
       sp = new circle;    sp->In(); return sp;
     default:       return 0;
     }
}
```

ОО обобщение прекрасно поддерживает эволюционное расширение агрегатов, при котором используется только одна производная альтернатива, заменяющая ту, которая эксплуатировалась до нее.
Эта возможность вытекает из того, что обобщение - агрегат. А добавляемые или изменяемые в производных классах виртуальные методы можно трактовать как методы агрегата. Таким образом, применяя наследование, **мы можем легко расширять внутреннюю структуру и функциональность объекта**, предоставляемого клиентам. А **сохранение интерфейса обеспечивает прозрачную для клиента подмену одного объекта другим**. Методы любого производного класса могут использовать тот же интерфейс, что и методы ранее используемых базовых классов.
Главное для клиента - это отсутствие изменений в интерфейсе связываемого объекта.

Ограничения техники ООП

Проблемы с расширением функциональности альтернатив

При добавлении нового обработчика специализации необходимо включить в базовый класс новую виртуальную процедуру, расширяющую исходный интерфейс. Далее требуется вставить во все производные классы методы, осуществляющие непосредственное вычисление периметров.

При процедурном подходе таких проблем не возникает.

Следует отметить, что большая часть расходов ложится не на модификацию, а перекомпиляцию

Пример. Добавление вычисления периметра (ПП)

```
// Вычисление периметра прямоугольника double
Perimetr(rectangle &r) { return (r.x + r.y) * 2.0; }
// Вычисление периметра треугольника double
Perimetr(triangle &t) { return t.a + t.b + t.c; }

// Нахождение периметра обобщенной фигуры double
Perimetr(shape &s) {
  switch(s.k) {
    case shape::key::RECTANGLE: return Perimetr(s.r);
    case shape::key::TRIANGLE:  return Perimetr(s.t);
    default: return 0.0;
  }
}
// Вычисление суммарного периметра для фигур
double Perimetr(container &c) {
  double a = 0;
  for(int i = 0; i < c.len; i++) {
    a += Perimetr(*(c.cont[i]));
  }
  return a;
}
```

Пример. Добавление вычисления периметра (ООП)

```
// Класс, обобщающает все имеющиеся фигуры.
// Изменился в связи с добавлением вычисления периметра
class shape {
public:
 virtual void In() = 0; // ввод данных
 virtual void Out() = 0; // вывод данных
 virtual double Area() = 0; // вычисление площади фигуры
 // добавлено вычисление периметра фигуры
 virtual double Perimetr() = 0;
};
```

```
// Измененный прямоугольник (вычисляет периметр)
class rectangle: public shape {
int x, y; // ширина, высота
public:
void In(); // ввод данных из стандартного потока
 void Out(); // вывод данных в стандартный поток
 double Area(); // вычисление площади фигуры
 double Perimetr(); // добавлено вычисление периметра
 rectangle(int _x, int _y);
 rectangle() {}
}

// Измененный треугольник
class triangle: public shape {
 int a, b, c; // стороны
public:
 void In(); // ввод данных из стандартного потока
 void Out(); // вывод данных в стандартный поток
 double Area(); // вычисление площади фигуры
 double Perimetr();
 // добавлено вычисление периметра фигуры
 triangle(int _a, int _b, int _c);
 triangle() {}
};
```

Проблемы при добавлении специализированных действий

Пример. Надо выводить все прямоугольники, расположенные в контейнере. Соответствующая процедура должна "**выявлять**" прямоугольник из множества фигур всех видов и запускать специализированную процедуру вывода.

Использование процедурного подхода позволяет добавить новый метод без каких-либо проблем, так как его отличие от других обработчиков альтернатив заключается только в анализе одного признака. Ее легко сформировать путем вырезания лишнего кода из уже существующей процедуры вывода произвольной альтернативы.

Добавление в процедурную программу

```
// Вывод только обобщенного прямоугольника
// Процедура добавлена без изменений
// других объектов

bool Out_rectangle(shape &s) {
  switch(s.k) {
    case shape::key::RECTANGLE:
      Out(s.r);
      return true;
    default: return false;
  }
}
```

Объектно-ориентированный подход не позволяет, в данном случае, получить элегантное решение.

Возможные пути:

1. Использование динамического анализа типа объекта. Но это – **процедурный прием**, так как используются внешние свойства объекта.
2. Перенос специализированного обработчика в базовый класс и закрепления за ним функций ничего неделания, используя для этого, например, пустое тело. Основным недостатком примененного технического решения является "**разбухание**" интерфейсов базового и производных классов. Возникают проблемы, связанные модификацией базового класса и полной перекомпиляции всех зависимостей при добавлении **каждого нового специализированного метода**.

Добавление в ОО программу

```
// Класс, обобщающает все имеющиеся фигуры.
class shape {
public:
 virtual void In() = 0; // ввод данных
 virtual void Out() = 0; // вывод данных
 // Добавлен вывод только прямоугольника как заглушку
 // Метод не является чистым и вызывается
 // там где не переопределен
 virtual bool Out_rectangle() { return false; };
 virtual double Area() = 0; // вычисление площади фигуры
};
```

```
// Прямоугольник переопределяет вывод себя
class rectangle: public shape {
 int x, y; // ширина, высота
public:
 void In(); // ввод данных из стандартного потока
 void Out(); // вывод данных в стандартный поток
 // Переопределен вывод только прямоугольника
 bool Out_rectangle(); // вывод только прямоугольника
 double Area(); // вычисление площади фигуры
 rectangle(int _x, int _y);
 rectangle() {}
}; /



// Вывод только прямоугольника
bool rectangle::Out_rectangle() {
  Out(); return true;
};
```

Выводы

1. Независимо от техники и парадигм программирования создаются программные объекты и отношения между ними, с использованием таких понятий как **агрегат и обобщение**. К одному и тому же конечному результату можно прийти различными путями.
2. Процедурное агрегирование обладает большей гибкостью по сравнению с объектно-ориентированным, а метафора автономного полнофункционального объекта не всегда удобна, так как вступает в противоречие с ассоциациями, необходимыми при построении эволюционно наращиваемых программ.

3. Объектное обобщение обладает неоспоримыми преимуществами при создании **эволюционно расширяемых альтернатив и агрегатов с неизменяемым интерфейсом**.
4. Эволюционное программирование опирается на **универсальные приемы и методы** присущие **различным** парадигмам.
5. **Процедурное программирование слишком универсально и разнообразно по используемым техническим решениям, чтобы эффективно использоваться для написания прикладных программ!??**