

# ФОРМАЛИЗАЦИЯ ПАРАДИГМЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ: КРИТИКА ОПРЕДЕЛЕНИЯ ГРАДИ БУЧА

ПИСКУНОВ А.Г.

17 мая 2007 г.

## АННОТАЦИЯ

Документ представляет собой попытку формализовать основные понятия объектно - ориентированного программирования. Даются определения таким основным понятиям ООП как класс, объект, наследование, инкапсуляция, полиморфизм через математические термины: множество, отношение, функция, абстрактный автомат. Строго различаются понятия тип (множество состояний абстрактного автомата) и класс (абстрактный автомат).

В работе существенную помощь оказали О.Головки и М.Николаев.

Д. Гильберт

Никто не может изгнать нас из рая, который создал Кантор.

## 1 ВВЕДЕНИЕ

Идея пошаговой разработки программ с последовательным уточнением и устранением разного рода неопределенностей подымается практически во всех работах по программированию, начиная с учебников, таких как [6], языков спецификаций [14] и заканчивая сложными современными технологиями [17] в которых используются автоматическое доказательство соответствия следующего шага предыдущему. Вслед за автором [10] будем рассматривать разработку программного обеспечения, как процесс преобразования текста с решением некоторой задачи из одного языка в другой язык с ее последовательной детализацией. При этом, в последовательности записи решений можно провести четкую границу между реализацией (т.е. записью решения на некотором (-ых) языке(-ах) программирования) и спецификацией (в самом крайнем случае это желание программиста сделать решить задачу).

Наличие удобных абстракций в языках спецификаций, позволяющую удобную ее запись в языке реализации должно существенно облегчать весь процесс разработки программного обеспечения. Например, если под программным обеспечением будет пониматься приложение, использующее РСУБД, то тогда, в языках реализации будет использоваться один из промышленных SQL, а в языках спецификаций может использоваться реляционная алгебра как формальная математическая теория. С другой стороны, несмотря на признание важности как можно более ранней формализации в [6] и очевидное соображение, что абстракцией от пары данные + программы может быть только пара значения + функции (то есть, абстрактный автомат [4]), для абстрактных типов данных не было замечено простой формальной математической модели.

Например, определения из [15], [24] мало похожи на математику. С другой стороны, в монографии [9] существенно используется понятие гибридного автомата. Однако это понятие избыточно переусложнено, не дается определение наследования автоматов и, собственно, автомат считается моделью не класса, а объекта.

В работах [17], [16] абстрактный тип, задаваемый схемами, очень близок к общепринятому понятию класса. Язык RSL поддерживает аналоги наследования классов - расширение схем-классов (extending), инкапсуляции - hiding и шаблонов - параметризованных схемы. Можно использовать объекты заданных схем-классов, но, по видимому, невозможно при помощи схем описывать формальные параметров функции и передавать в функции объекты. Также в языке RSL отсутствует операция прямого произведения множеств, понимаемая как операция над данными, а не как средство задания нового типа. Это приводит к существенно более громоздкому описанию схем РСУБД, чем при использовании реляционной алгебры или языка SQL ([22]).

Кроме того, смотри [19], [21], [18], [23], [11].

Далее будет предпринята попытка определить основные понятия ООП через автомат Мура.

Обозначения

- либо общеприняты и с ними можно познакомиться, к примеру, в таком введении в теорию множеств как [1];
- либо взяты из языка формальных спецификаций RSL [16]. Русскоязычные ресурсы по языку RSL: [3], [5];
- либо это обозначение операции диагонального произведения (операции соединения) из набора элементарных операций над частичными функциями. см. [8].

## 2 ОСНОВНЫЕ ОПРЕДЕЛЕНИЯ И РЕЗУЛЬТАТЫ

### 2.1 Основные определения

Полувывчислимая функция

(см. [8]) Функция  $f$  называется полувывчислимой, если существует программа

- вырабатывающая на выходе значение  $f(x)$  если на ее вход значения значения  $x$  из области определения функции  $\text{dom } f$ ;

- никогда не останавливающаяся при подаче на вход значения не принадлежащего области определения  $f$ :  $x \notin \text{dom } f$ .

### Автомат Мили

Пусть  $Q$  - множество состояний,  $X$  - множество входных сигналов,  $Y$  - множество выходных сигналов.

Тогда тройка  $(Q, \delta : Q \times X \rightarrow Q, \lambda : Q \times X \rightarrow Y)$  называется абстрактным автоматом Мили, где  $\delta$  - функция переходов,  $\lambda$  - функция выходов

### Автомат Мура

Пусть  $Q$  - множество состояний,  $X$  - множество входных сигналов,  $Y$  - множество выходных сигналов.

Тогда тройка  $(Q, \delta : Q \times X \rightarrow Q, \lambda : Q \rightarrow Y)$  называется абстрактным автоматом Мура, где  $\delta$  - функция переходов,  $\lambda$  - функция выходов

## 2.2 Утверждение

Для каждого автомата Мили можно построить имитирующий его поведение автомат Мура.

Доказательство смотри, например, в конце раздела 2.2 [4] .

### Диагональное произведение отображений

Пусть заданы отображения  $g_1 : X \rightarrow Y_1$ , ..., и  $g_k : X \rightarrow Y_k$ . Тогда отображение  $(g_1, \dots, g_k) : X \rightarrow Y_1 \times \dots \times Y_k$ , определенное условием  $(g_1, \dots, g_k)(x) = (g_1(x), \dots, g_k(x))$  для каждого  $x \in X$ , называется диагональным произведением отображений  $g_1, \dots, g_k$ . Сами отображения  $g_1, \dots, g_k$  называются компонентами диагонального произведения.

### Проекция

Отображение  $\pi_{X_j} : X_1 \times \dots \times X_k \rightarrow X_j$  заданное как  $\pi_{X_j}(x_1, \dots, x_k) = x_j$ , где  $1 \leq j \leq k$  и  $x_j \in X_j$  будет называться проекцией.

Обсуждение свойств диагонального произведения, композиции и проекции можно посмотреть в [1] .

## 2.3 Утверждение

Пусть  $T, X, Y$  некоторые множества, тогда каждое отображение  $f : T \rightarrow X \times Y$  можно представить как диагональное произведение композиций самой функции  $f$  и проекций множества значений на сомножители прямого произведения  $X \times Y$ .

Доказательство. Пусть для  $f$  выполняется  $f(t) = (x, y)$ ,  $t \in T$ ,  $x \in X$ ,  $y \in Y$ . Тогда, по определению проекции,  $(x, y) = (\pi_X(x, y), \pi_Y(x, y)) = (\pi_X(f(t)), \pi_Y(f(t)))$ . То есть, по определению,  $f$  есть диагональное произведение отображений  $\pi_X \circ f$  и  $\pi_Y \circ f$ .

## 2.4 Следствие

Любая функция  $f : T \rightarrow Q \times Y$  может быть выведена из двух функций  $f_s$  и  $f_o$ , причем множество значений первой, содержится в первом сомножителе -  $Q$ , то есть  $f_s : T \rightarrow Q$ , а множество значений второй, во втором сомножителе -  $f_o : T \rightarrow Y$ .

Обсуждение. В доказательстве леммы были использованы простейшие функции - проекция и элементарные операции над функциями - композиция и диагональное произведение (иначе соединение) см. раздел Частично рекурсивные функции в [8]. Следовательно, если функция  $f$  была частично рекурсивна, то функции - компоненты  $f_s = \pi_Q \circ f$  и  $f_o = \pi_Y \circ f$  тоже частично рекурсивны, а значит, полувывчислимы. Таким образом, можно считать что функция  $f$  определяется (выводится из) через более простые функции компоненты. Полувывчислимость функции будем понимать как синоним возможности написать эти функции на каком либо языке программирования.

### Функции состояния и выхода

Далее, для некоторых множеств  $Q, X, Y$  (причем  $Y$  нельзя представить в виде декартового произведения  $Q$  с каким либо множеством) функцию  $f_s : Q \times X \rightarrow Q$  будем называть функциями состояния, а функции  $f_o : Q \rightarrow Y$  функцией выхода.

### Замечание

Похоже что, в работе [17] (см. стр 47, 95) функции состояния обозначаются термином generator, функции выхода - observer, а множество  $Q$  - типом интересов.

## 2.5 Утверждение

Пусть есть множество  $Q$ . Два любых набора функций:

- состояния:  $\{ g_i : Q \times X_i \rightarrow Q \mid i \leq n \}$
- выхода:  $\{ f_j : Q \rightarrow Y_j \mid j \leq m \}$

для некоторых множеств  $Q$ ,  $X_i$ ,  $i \leq n$ ,  $Y_j$ ,  $j \leq m$  задают **автомат Мура**.

Доказательство. Пусть множество  $N = \{1, \dots, n\}$ . Тогда зададим функцию переходов

$\delta : Q \times N \times X_1 \times \dots \times X_n \rightarrow Q$  следующим образом:

$\delta ( q, i, x_1, \dots, x_n )$  is

case i of

1  $\rightarrow g_1(q, x_1)$ ,

2  $\rightarrow g_2(q, x_2)$ ,

...

n  $\rightarrow g_n(q, x_n)$

end

И функцию выходов  $\lambda : Q \rightarrow Y_1 \times \dots \times Y_m$  следующим образом

$\lambda ( q )$  is  $(f_1(q), \dots, f_m(q))$

Обозначив  $N \times X_1 \times \dots \times X_n$  через  $X$ , а  $Y_1 \times \dots \times Y_m$  - через  $Y$ , можно окончательно убедиться что полученная тройка  $(Q, \delta, \lambda)$  удовлетворяют определению автомата Мура по построению.

## 2.6 Следствие

Любое множество  $Q$  с любым набором функций/подпрограмм, содержащие это множество  $Q$  в сигнатуре, задает **автомат Мура**.

Класс, тип

Для любого автомата  $(Q, \delta, \lambda)$  множество его состояний  $Q$  будем называть типом. Термин класс будем считать синонимом термина [автомат](#).

Для некоторого автомата  $A$ , через  $Q(A)$  будем обозначать его тип.

Вследствие утверждения [2.6](#) классом можно называть также тройку  $(Q, \{ g_i : Q \times X_i \rightarrow Q \mid i \leq n \}, \{ f_i : Q \rightarrow Y_i \mid j \leq m \})$

Объект, процесс

Далее, для произвольного класса  $A = (Q, F_s, F_o)$

- в сигнатуре функций, в частности, из множеств  $F_s$  и  $F_o$ , вместо типа класса  $Q(A)$ , можно будет писать обозначение класса  $A$ ;
- также допускается запись  $a \in A$ , вместо  $a \in Q(A)$ . Она означает, что хотя величина  $a$  является таким же кортежем как любой элемент  $Q(A)$ , она не может использоваться ни в каких выражениях, кроме как в функциях с классом  $A$  (забегая вперед, с наследником класса  $A$ ) в сигнатуре.

Любая величина  $a$  определенная как  $a \in A$  будет называться объектом. Если во множестве функций состояния  $F_s$  существует хотя бы одна функция с пустым вторым сомножителем, то есть вида  $g : A \rightarrow A$ , то величина  $a$  будет называться процессом.

Заметим, что раз тип  $Q(A)$  и класс  $A$  - различные понятия, то можно говорить о объекте класса  $A$  и типа  $Q(A)$ .

#### Замечание

Толкование объекта как кортеж очень близко к толкованию объекта как запись в работе [\[20\]](#), в нашем случае роль меток значений играют функции состояния и выхода.

### 3 НАСЛЕДОВАНИЕ, ИНКАПСУЛЯЦИЯ, ПОЛИМОРФИЗМ

#### Ассоциативность

Отметим (см., например, [13], стр. 146, раздел Естественное соединение) Операция соединения и, следовательно, декартового произведения - ассоциативны:

$$(A \times B) \times C = A \times (B \times C) = A \times B \times C$$

Ассоциативность декартового произведения означает, что при  $X = A \times B$  функцию  $f : X \rightarrow T$ , можно рассматривать как функцию с сигнатурой  $f : A \times B \rightarrow T$ .

#### Лямбда выражения

Далее символ лямбда  $\lambda$  будет использоваться для записи лямбда выражений над функциями. Выражение

$$\lambda \text{ val}_1 : T_1, \dots, \text{val}_n : T_n \text{ :- expression}(\text{val}_1, \dots, \text{val}_n)$$

будет определять функцию с сигнатурой  $T_1 \times \dots \times T_n \rightarrow T$ , где  $T$  - есть **тип** выражения  $\text{expression}(\text{val}_1, \dots, \text{val}_n)$ .

Например, выражение  $\lambda x : \text{Int}, y : \text{Int} \text{ :- } x+y$  определяет функцию сложения двух целых чисел с сигнатурой  $\text{Int} \times \text{Int} \rightarrow \text{Int}$ .

#### 3.1 Наследование автоматов

Пусть есть **автомат**

$$B = (Q_2, \{ g_{2,i} : Q_2 \times X_{2,i} \rightarrow Q_2 \mid i \leq n_2 \}, \{ f_{2,i} : Q_2 \rightarrow Y_{2,i} \mid i \leq m_2 \})$$

и автомат

$A = (Q_1, \{ g_{1,i} : Q_1 \times X_{1,i} \rightarrow Q_1 \mid i \leq n_1 \}, \{ f_{1,i} : Q_1 \rightarrow Y_{1,i} \mid i \leq m_1 \})$ . Будем говорить, что автомат  $B$  наследует автомат  $A$  (**Класс**  $B$  наследует класс  $A$ ), если

- $Q_2$  есть декартовым произведением  $Q_l \times Q_1 \times Q_r$ , где  $Q_l$  и/или  $Q_r$  могут быть пустыми множествами;
- (условие наследования для функций состояния) существует частично определенное инъективное отображение  $\tau : \{i \leq n_1\} \rightsquigarrow \{i \leq n_2\}$

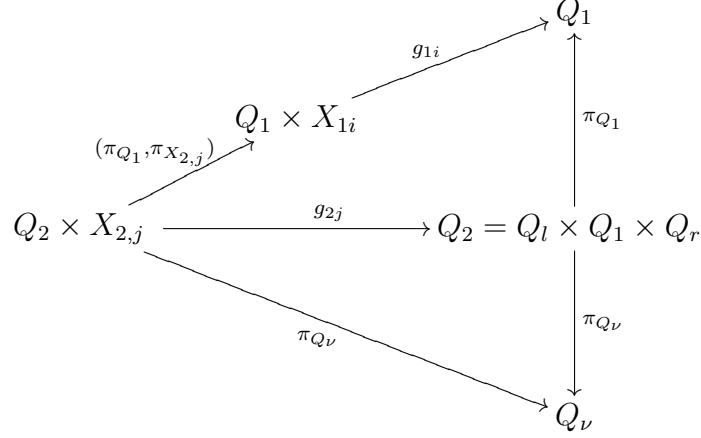


Рис. 1: Условие наследования функций состояния.  $\nu \in \{l, r\}$ ,  $X_{2,j} \stackrel{def}{=} X_{1,i}$ ,  $Q_2 \stackrel{def}{=} Q_l \times Q_1 \times Q_r$

} такое что, для любого  $i \leq n_1$  и любого  $(q_l, q_1, q_r) \in Q_2$  функция  $g_{2,\tau(i)}$  и функция  $g_{1,i}$  связаны следующим соотношением:

$$g_{2,\tau(i)} = \lambda (q_l, q_1, q_r, x) : Q_2 \times X_{2,\tau(i)} :- (q_l, g_{1,i}(q_1, x), q_r) ,$$

то есть, функция  $g_{2,\tau(i)}$  является диагональным произведением

$$g_{2,\tau(i)} = (\pi_{Q_l}, g_{1,i} \circ (\pi_{Q_1}, \pi_{X_{2,\tau(i)}}), \pi_{Q_r}). \text{ Отметим, что } X_{2,\tau(i)} = X_{1,i} \text{ (см. рис. 1):}$$

- (условие наследования для функций выхода) существует частично определенное инъективное отображение  $\rho : \{i \leq m_1\} \rightsquigarrow \{i \leq m_2\}$ , такое что  $f_{2,\rho(i)} = f_{1,i} \circ \pi_{Q_1}$ ,  $i \leq m_1$  причем,  $Y_{2,\rho(i)} = Y_{1,i}$  (см. рис. 2).

**Автомат** А будет называться родителем, автомат В - потомком. Если считать, что В подкласс, А - надкласс, а функции состояния, для которых выполняется условие наследования (то есть,  $g_{2,\tau(i)}$  и  $g_{1,i}$ ) и функции выхода, для которых выполняется условие наследования (то есть,  $f_{2,\rho(i)}$  и  $f_{1,i}$ ) разделяются (sharing) между классами, то это определение вполне соответствует определению наследования в смысле [19] :

$$\begin{array}{ccc}
 Q_2 & \xrightarrow{f_{2j}} & Y_j \\
 \downarrow \Pi & & \downarrow id \\
 Q_1 & \xrightarrow{f_{1i}} & Y_i
 \end{array}$$

Рис. 2: Условие наследования для функций выхода

Inheritance is the sharing of attributes between a class and its subclass. Multiple inheritance occurs when a subclass inherits attributes from more than one class. Single inheritance occurs when a subclass inherits attributes from only one class.

Где под атрибутом может пониматься как переменная так и метод.

### 3.2 Пример множественного наследования

Рассмотрим пример множественного наследования в терминах языка C++:

```

class A      { public: int a; };
class B      { public: int a; };
class C : public A, public B {      };

```

И запишем его в терминах **автоматов**:

$A = (A.Int, \{a.set = \lambda (x,y): A.Int \times Int \text{ :- } y\}, \{a.get = \lambda x: A.Int \text{ :- } x\})$ .

Функции (и тип) **класса** будем указывать

- явно задавая функцию, **класса** например, `A.a.get`;
- либо, в случае уникальности имени функции в каком либо тексте (спецификация или программе), просто задавая ее имя - `a.get`.

Причем давать имена `set` и `get`, вообще говоря, большой нужды нет, в связи с различными сигнатурами функций. Это сделано для дополнительного указания на аналоги операторов присвоения (`set`) и извлечения значения (`get`);

Далее, класс  $B = (B.Int, \{a = \lambda (x,y): B.Int \times Int \text{ :- } y \}, \{a = \lambda x: B.Int \text{ :- } x \})$  ;  
 и класс  $C = ( A.Int \times B.Int,$   
 $\{ a_1 = \lambda (a,b,x): A.Int \times B.Int \times Int \text{ :- } (A.a.set (a,x),b),$   
 $a_2 = \lambda (a,b,x): A.Int \times B.Int \times Int \text{ :- } (a, B.a(b,x)) \},$   
 $\{ a_1 = A.a.get \circ \pi_{A.Int}, a_2 = B.a \circ \pi_{B.Int} \} )$

По построению видно, что удовлетворены все условия определения наследования:

- **Тип автомата** наследника  $Q_2$ , в данном случае,  $A.Int \times B.Int$  содержит в качестве сомножителя тип любого из автоматов родителей  $Q_1$ ,  $A.Int$  равно как и  $B.Int$ ;
- Если в качестве родителя рассматривать **класс**  $A$ , то есть, под **типом**  $Q_1$  понимать  $A.Int$ , а под набором функций состояния  $\{g_{1,i}\}$  понимать множество  $\{A.a.set\}$ , то становится очевидно, что функциями  $A.a.set$  и  $a_1$  находятся в отношении

$$a_1 = \lambda (q_1, q_r, x): A.Int \times B.Int \times Int \text{ :- } (A.a.set (q_1, x), q_r),$$

левый сомножитель  $Q_l$  пуст, правый  $Q_r = B.Int$  ,  $g_{2,1}$  в данном примере это  $a_1$ :

$$g_{2,1} = \lambda (q_l, q_1, q_r, x) : Q_2 \times Int \text{ :- } (q_l, g_{1,1}(q_1, x), q_r)$$

Те же рассуждения годятся в случае, если в качестве родителя рассматривать класс  $B$ . Тогда пустым будет правый сомножитель  $Q_r$ ;

- Условия соотношения функций выхода родителей и потомка выполняются по построению.

Таким образом, получено, что записанные автоматы  $A$ ,  $B$ ,  $C$  удовлетворяют определению наследования, причем **автоматы**  $A$ ,  $B$  есть родителями, а автомат  $C$  - наследником. Еще раз отметим, проблемы множественного наследования от одного и того же класса,

```
class C : public A, public A { }
```

не представляет особых трудностей, если в качестве оператора разрешения области видимости :: языка  $C++$  пользоваться полным или частичным путем из имен классов родителей, для указания функций заданного

родителя. Естественно, в случае множественного наследования из одного и того же класса необходимо давать имя - синоним для каждого вхождения такого класса. Чтото вроде :

```
class C : public A, public A as B { } ;
```

### 3.3 Выражения над автоматами и инкапсуляция

**Наследование** автоматов, по сути, основано на декартовом произведении их типов и поэтому **класс**

```
class A      { public: int a; };
class C : public A, public A as B { float x; } ;
```

например, может записываться как  $C =$

$$A \times A \text{ as } B \times (\text{Real}, \\ \{a_1 = A.a, a = B.a, x = \lambda (a, b, x, \text{val}) : A.\text{Int} \times B.\text{Int} \times \text{Real} \times \text{Real} \\ \text{:} (a,b,\text{val})\}, \\ \{a_1 = A.a, a = B.a, x.\text{get} = \lambda (a, b, \text{val}) : A.\text{Int} \times B.\text{Int} \times \text{Real} \text{:} \text{val}\} \\ )$$

Где функции состояния и выхода  $A.a$  и  $B.a$  определены согласно определению наследования в классе потомка  $C$  из соответствующих функций состояния и выхода  $A.a$  (класса родителя  $A$ ) и  $B.a$  (класса родителя  $B$ ). Необходимо еще раз отметить, что в случае однозначного именованя, вообще говоря, незачем их специально именовать, Функция класса наследника по умолчанию получает имя от функции класса родителя. Далее, добавление функции для присвоения значения и извлечения значения для новой компоненты может происходить в сокращенной форме. По системе обозначений из [16] обозначить пару  $x, x.\text{get}$  можно , например, в следующем виде:  $x.\text{get} : \text{Real} \leftrightarrow x$ . Общий вид -  $v : T \leftrightarrow v$ , для некоторого типа  $T$ . Первое вхождение  $v$  обозначает функцию выхода  $v : Q \rightarrow T$ , второе вхождение - функцию состояния  $v : Q \times T \rightarrow Q$ . Любая из функций может быть пропущена. Введенные обозначения позволяют сократить определение **класса**  $C$  до минимума:

$$C = A \times A \text{ as } B \times ( x : \text{Real} \leftrightarrow x, \{a_1 = A.a\}, \{a_1 = A.a\} )$$

Инкапсуляция функций может выражаться через теоретико- множественную операцию разности множеств. Таким образом наследование

```
class A      { public: int a; };
class C : A, public A as B { public: float x; } ;
```

может быть записано как

$$C = A \times A \text{ as } B \times (x : \text{Real} \leftrightarrow x, \{a_1 = A.a\}, \{a_1 = A.a\}) \setminus (, \{a_1\}, \{a_1\} )$$

Внутри анонимного **класса** (определенного слева от символа  $\setminus$ ) можно пользоваться функциями  $a_1$ , а в самом классе  $C$  функций  $C.a_1$  не существует.

### 3.4 Полиморфизм

Как видно из обсуждения **наследования** и **инкапсуляции** - эти два термина имеют отношения к конструированию **автоматов**. В случае полиморфизма ситуация другая, так как он для конструирования классов не используется.

#### Замечание

Как отмечается в монографии [9] :

"Традиционный" полиморфизм заключается в том, что вместо декларированного объекта определенного класса фактически могут использоваться экземпляры любых производных классов от заданного. Полиморфизм "по интерфейсу" означает, что можно заменить локальный объект на любой другой, в котором существуют внешние переменные, соответствующие по идентификатору и типу значения внешним переменным заменяемого объекта.

Или то же в работе [19]

Given classes C and D where D is a subclass of C, then an instance of class D (d) may be used as an instance of class C. This is referred to as (subtyping) polymorphism.

Ограничимся "традиционным" полиморфизмом.

Полиморфизм функций автомата родителя и автомата наследника

Пусть есть автомат родитель A и автомат наследник B. В обоих

автоматах есть функция с одинаковым именем  $f$ . Для определенности, функцию из автомата потомка обозначим как  $f_B$ . Будем говорить, что функции  $f$  и  $f_B$  находятся в отношении полиморфизма, если

- В случае функций состояния ( $f_B : Q(B) \times X \rightarrow Q(B)$  и  $f : Q(A) \times X \rightarrow Q(A)$ ) не выполняется условие наследования функций состояния (см. рис. 1):

$$f_B \neq (\pi_{Q_i}, f \circ (\pi_{Q(A)}, \pi_X), \pi_{Q_r}) .$$

- В случае функций выхода ( $f_B : Q(B) \rightarrow Y$  и  $f : Q(A) \rightarrow X$ ) не выполняется условие наследования функций выхода (см. рис. 2):

$$f_B \neq f \circ \pi_{Q(A)} .$$

#### Полиморфизм функций автоматов наследников

Пусть автомат  $A$  является родителем автоматов  $B$  и  $C$ . Пусть во всех трех классах существует функция с одним именем  $f$  (как и раньше обозначения  $f_B$ ,  $f_C$  нужны для различения этих функций). Функции  $f_B$  и  $f_C$  находятся в отношении полиморфизма относительно класса  $A$ , если

$f_B$  и  $f$  находятся в отношении полиморфизма и  $f_C$  и  $f$  находятся в отношении полиморфизма.

#### Полиморфизм множеств функций автоматов-наследников

Если у классов  $B$  и  $C$  наследующих класс  $A$ , найдутся имена функций  $F = \{ f_1, \dots, f_n \}$ , находящихся в отношении полиморфизма относительно класса  $A$ , то классы  $B$  и  $C$  находятся в отношении полиморфизма относительно класса  $A$  и множества имен функций  $F$ .

Отношение полиморфизма относительно  $A$  и множества  $F$  позволяет записывать шаблон некоторой функции  $f$  общей для всех наследников класса  $A$ , если в  $f$  используются только функции с именами из множества  $F$ .

Рассмотрим следующий пример. В примере через  $\text{Real} : \text{Int} \rightarrow \text{Real}$  обозначена функция для преобразования целого числа в плавающее:

- $P = (\text{Bool}, \{x = \lambda (b, v) : \text{Bool} \times \text{Int} \text{ :- if } v = 0 \text{ then false else true end } \},)$
- $I = P \times (\text{Int}, \{x = \lambda (b, i, v) : Q(I) \times \text{Int} \text{ :- } (b, v) \},)$
- $R = P \times (\text{Real}, \{x = \lambda (b, r, v) : Q(R) \times \text{Int} \text{ :- } (b, \text{Real}(v)) \},)$
- и задана некоторая функция у которой в качестве области определения вместо типа появляется класс P:  
 $f : P \times \text{Int} \rightarrow P \quad f(p, v) \text{ is } P.x (p, v)$

Такое определение функции означает, что в этом примере задано 3 функции f:

- $f : Q(P) \times \text{Int} \rightarrow Q(P) \quad f(p, v) \text{ is } P.x (p, v)$
- $f : Q(I) \times \text{Int} \rightarrow Q(I) \quad f(p, v) \text{ is } I.x (p, v)$
- $f : Q(R) \times \text{Int} \rightarrow Q(R) \quad f(p, v) \text{ is } R.x (p, v)$

В каждой из которых используется функция x соответствующего типа для соответствующего автомата. И, естественно, запись  $f(p, 1)$ , для  $p \in Q(P)$  и для  $p \in Q(I)$  означает применение различных функций. Таким образом, разделение понятия тип и класс позволяет согласовать полиморфизм и проверку типов.

В примере классы P, I, R находятся в отношении полиморфизма относительно класса P и множества { x }.

#### Замечание

В тоже время в классе I (равно как и R), кроме его собственной функции  $x : Q(I) \times \text{Int} \rightarrow Q(I)$ , по определению наследования существует функция  $I.P.x : Q(I) \times \text{Bool} \rightarrow Q(I)$ , полученная из родительского автомата  $I.P.x = \lambda (b, i, v) : Q(I) \text{ :- } (P.x(b, v), i)$ .

## 4 ОБСУЖДЕНИЕ ОПРЕДЕЛЕНИЯ ГРАДИ БУ- ЧА

Попробуем применить следующее определение: согласно [2], гл. 2. объектно-ориентированным программированием назовем методологию программирования, основанную на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования. Программа будет объектно-ориентированной только при соблюдении всех трех указанных требований.

В данном определении можно выделить три части:

- Объектно ориентированное программирование использует в качестве базовых элементов объекты, а не алгоритмы (иерархия 'part of', которая была определена в главе 1 [2]);
- каждый объект является экземпляром какого-либо определенного класса;
- классы организованы иерархически (см. понятие об иерархии 'is a' там же).

### 4.1 Применение определения

Не будем останавливаться на том, что совокупность должна быть упорядочена. Вообще говоря, при перестановке двух объектов в программе, она может поменять свое поведение и, значит, станет другой программой. То есть, надо говорить не о совокупности, а о последовательности. Просто рассмотрим программу на любом языке с контекстно-независимой грамматикой. Согласно классификации Хомского см. [12], в этот класс грамматик попадают грамматики практически всех языков программирования. Причем, как показано в [7] выполняется лексический анализ, при котором текст разбивается на лексемы (третий тип языков по классификации Хомского - регулярные языки) и, затем, выполняется синтаксический анализ (parsing), при котором из лексем составляются операторы языка. Таким образом, можно утверждать что программа, которую можно считать объектом состоящую из других объектов, представляет собой следующую иерархию

- объекты класса `object`, как прародитель всех остальных классов;
- объекты класса символ (`character`);
- объекты класса лексема - `lexeme`, из которого наследуются классы число (`number`), лексема идентификатор (`identifier`), лексема ключевое слово (`key_word`) и лексема один символ (`one_char_lexeme`);
- объекты класса оператор - `statement`. Из объекта оператор наследуются класса оператор присваивания (`assignment_stat`), условный оператор (`if_stat`), оператор присваивания (`iteration_stat`);

Иерархия объектов изображена на схеме 3. Черными стрелками отображается отношение 'is a', синими - отношение 'part of'. Например, класс идентификатор наследует свойства класса лексема, и содержит объекты класса символ.

#### Замечание

Будем считать что рассматриваемый язык содержит, по крайней мере, в качестве лексем - числа, переменные и ключевые слова, и в качестве операторов: оператор присвоения, условный оператор и оператор цикла. Большой необходимости усложнять рассмотрение и делить программу на подпрограммы нет. При этом принципиально рассуждения не помешаются, а количество объектов увеличится, собственно рассуждения и иерархия объектов усложнятся.

Таким образом, все три требования Гради Буча удовлетворены:

- программа рассмотрена как последовательность базовых элементов - объектов;
- каждый объект является объектом определенного класса ( ;) - смайлик);
- классы организованы иерархически. Иерархия строилась с учетом обоих отношений, упомянутых Гради Бучем. А именно отношений 'is a' и 'part of';

Из всех проведенных рассуждений получаем следующий парадоксальный результат:

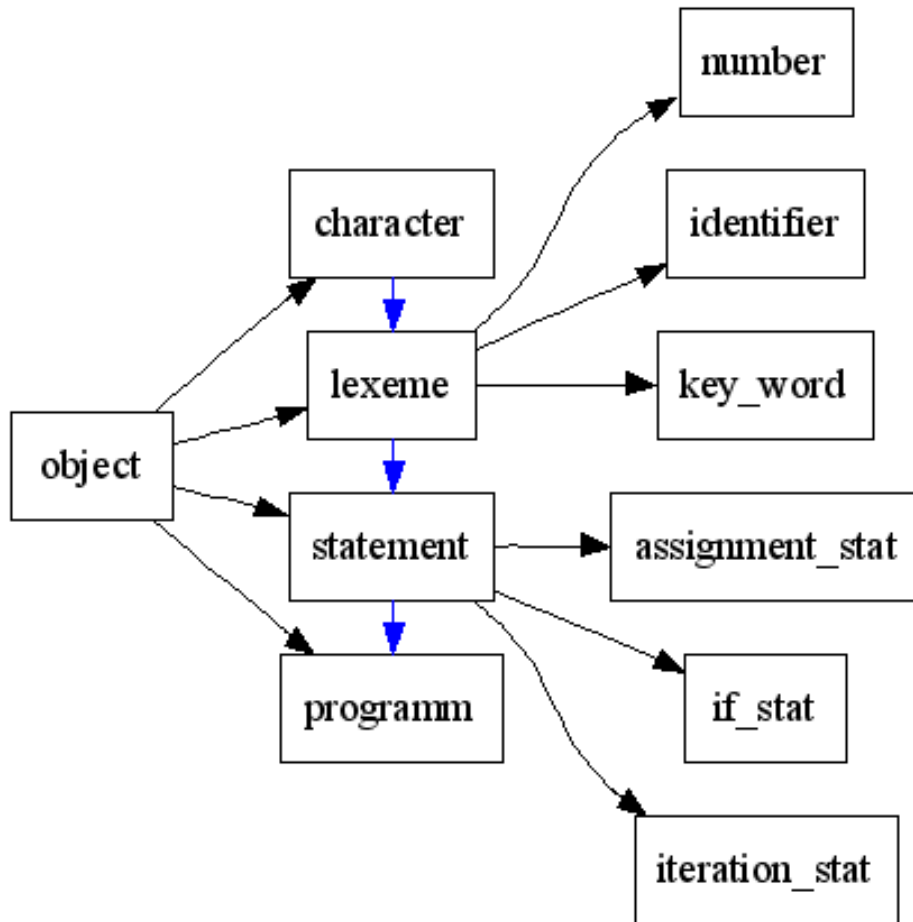


Рис. 3: Иерархия объектов программы языка с контекстно независимой грамматикой

## 4.2 Следствие

Согласно определению Гради Буча, любая программа, написанная на языке со контекстно независимой грамматикой без вызова подпрограмм и/или функций является программой, написанной по методологии объектно ориентированного программирования.

### 4.3 Опиять о использовании понятий отношение 'is a', тип и класс

Рассмотрим отношение 'is-a' (быть) между классами (объектами) позиционируемое в как критически важная часть ( см. глава Объектная модель, [2] ) объектно - ориентированного анализа и программирования и неразличения понятий типа и класса

Гради Буч:

Большинству смертных различать типы и классы просто противно и бесполезно.

....

Типизация - это способ защититься от использования объектов одного класса вместо другого, или по крайней мере управлять таким использованием.

На практике отношение 'is-a' с такой трактовкой типа приводит к немедленной путанице и противоречиям. В самом деле, если класс В является наследником класса А, то любой объект  $b \in B$ , по определению отношения 'is-a' должен быть признан объектом из класса А,  $b \in A$ . Таким образом приведение типа (А)  $b$ , ничего не должно менять в объекте  $b$ . Тип, то есть нечто не отличимое простым смертным от класса, не должен меняться с точки зрения простого смертного. Отсюда, получаем что операция "типизация" по Гради Бучу не должна защищать программиста от возможности использования переменных, отсутствующих в классе А, но присутствующих в классе В. По крайней мере, после присвоения переменной типа указатель на класс родителя адреса объекта класса потомка. Как в следующем примере:

```
class A      {public: int x;};
class B: public A {public: int y;};
B b;
A *a = &b; // объект b обязан не меняться при этом присвоении
a->y=13;
```

Таким образом

- либо "типизация" по Гради Бучу не будет защищать программиста от возможных ошибок и, следовательно, удовлетворять нуждам программиста;

- либо надо не признавать использование отношения 'is-a' в объектно-ориентированном программировании;
- либо надо отличать тип и класс в недвусмысленной и понятной форме.

Предложенное выше разделение понятия типов и классов позволяет устранить приведенное противоречие.

## Предметный указатель

автомат, 4, 6–11, 13

ассоциативность, 8

диагональное произведение отображений, 4

инкапсуляция, 12, 13

класс, 7, 8, 10–13, 15

лямбда выражение, 8

наследование, 8, 12, 13

объект, 7

полиморфизм, 13

процесс, 7

разность множеств, 12

тип, 7, 8, 11

функция выхода, 5

функция состояния, 5

## ССЫЛКИ

- [1] Кантор Георг и др. Теория множеств. <http://users.iptelecom.net.ua/~tchingiz/articles/setTheory.pdf>. 3, 4
- [2] Гради Буч. Объектно-ориентированный анализ и проектирование. <http://www.hardline.ru/1/5/1390/1789.html>. 16, 19
- [3] Кафедра системного программирования МГУ. Методы Формальной Спецификации Программ. <http://www.ispras.ru/~RedVerst/RedVerst/Lecturesandtrainingcourses/MSUcourseFormalspecificationofsoftware/RMain.html>. 3
- [4] Под редакцией М.А. Арбиба. Алгебраическая теория автоматов, языков и полугрупп. Москва, Статистика, 1975. 2, 4
- [5] А.К.Петренко Е.А. Кузьменкова. Формальная спецификация программ на языке rsl. <http://www.ergeal.ru/archive/cs/rsl/index.htm>. 3
- [6] А.Ахо, Д.Хопкрофт, Д. Ульман. Структуры данных и алгоритмы, 2003. Издательский дом Вильямс. 2
- [7] А.Ахо, Р.Сети, Д. Ульман. Компиляторы. Принципы, технологии, инструменты, 2003. Издательский дом Вильямс. 16
- [8] Манин Ю.И. Вычислимое и невычислимое, 1980. <http://agp1.nm.ru/arts/manin2.html>. 3, 5
- [9] Колесов Ю.Б. Объектно-ориентированное моделирование сложных динамических систем, 2004. <http://www.exponenta.ru/educat/systemat/kolesov/index.asp>. 2, 13
- [10] Костин Г.В. Процесс разработки ПО и ЯП, 2003. <http://bugtraq.ru/library/programming/languages.html>. 2
- [11] Зыков С. Современные языки программирования и .NET. Основы объектно - ориентированного подхода, 2003. <http://www.ict.edu.ru/ft/005130//index.html>. 3
- [12] Молчанов А.Ю. Системное программное обеспечение: Учебник для вузов, 2005. Издательский дом Питер. 16

- [13] К.Дейт. Введение в системы баз данных, 1998. 6-е издание. 8
- [14] Jonathan Bowen. Formal specification and documentation using z: A case study approach, 2003. <http://www.jpbowen.com/pub/zbook.pdf>. 2
- [15] Deborah J. Armstrong. The Quarks Object-Oriented Development, 2006. COMMUNICATIONS OF THE ACM, Vol 49, No 2. 2
- [16] Chris George. Introduction to RAISE, 2001. <http://users.iptelecom.net.ua/~agp1/arts/RAISE4.pdf>. 3, 12
- [17] Chris George. Introduction to RAISE. The RAISE Development Method, 2001. <http://users.iptelecom.net.ua/~agp1/arts/report249.pdf>. 2, 3, 6
- [18] Graeme Paul Smith. An Object-Oriented Approach to Formal Specification, 1992. <http://www.itee.uq.edu.au/~smith/papers/thesis.pdf>. 3
- [19] Jamie Shield. Towards an Object-Oriented Refinement Calculus, 2001. Thesis. 3, 9, 13
- [20] Luca Cardelli. A Semantics of Multiple Inheritance, 1988. Information and Computation 76, 138-164,1988. 7
- [21] Martin Abadi, Luca Cardelli, Ramesh Viswanathan. An Interpretation of Objects and Object Types. 3
- [22] A.G. Piskunov and V.L. Ilyuhin. The usage of formal specification languages for the design of rdbms. <http://users.iptelecom.net.ua/~agp1/ru/gsau.html>. 3
- [23] Thomas Studer von Werthenstein. Object-Oriented Programming in Explicit Mathematics: Towards the Mathematics of Objects, 2001. Bern, 3.April 2001. 3
- [24] Wikipedia. Object-oriented programming, 2006. [http://en.wikipedia.org/wiki/Object-oriented\\_programming](http://en.wikipedia.org/wiki/Object-oriented_programming). 2