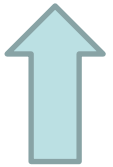
Языки и методы конструирования программ

Программирование 2
(Без раздела про C++
и дополнение про базы данных)

Содержание

Лекция 1. Модель вычислений фон Неймана и традиционные языки

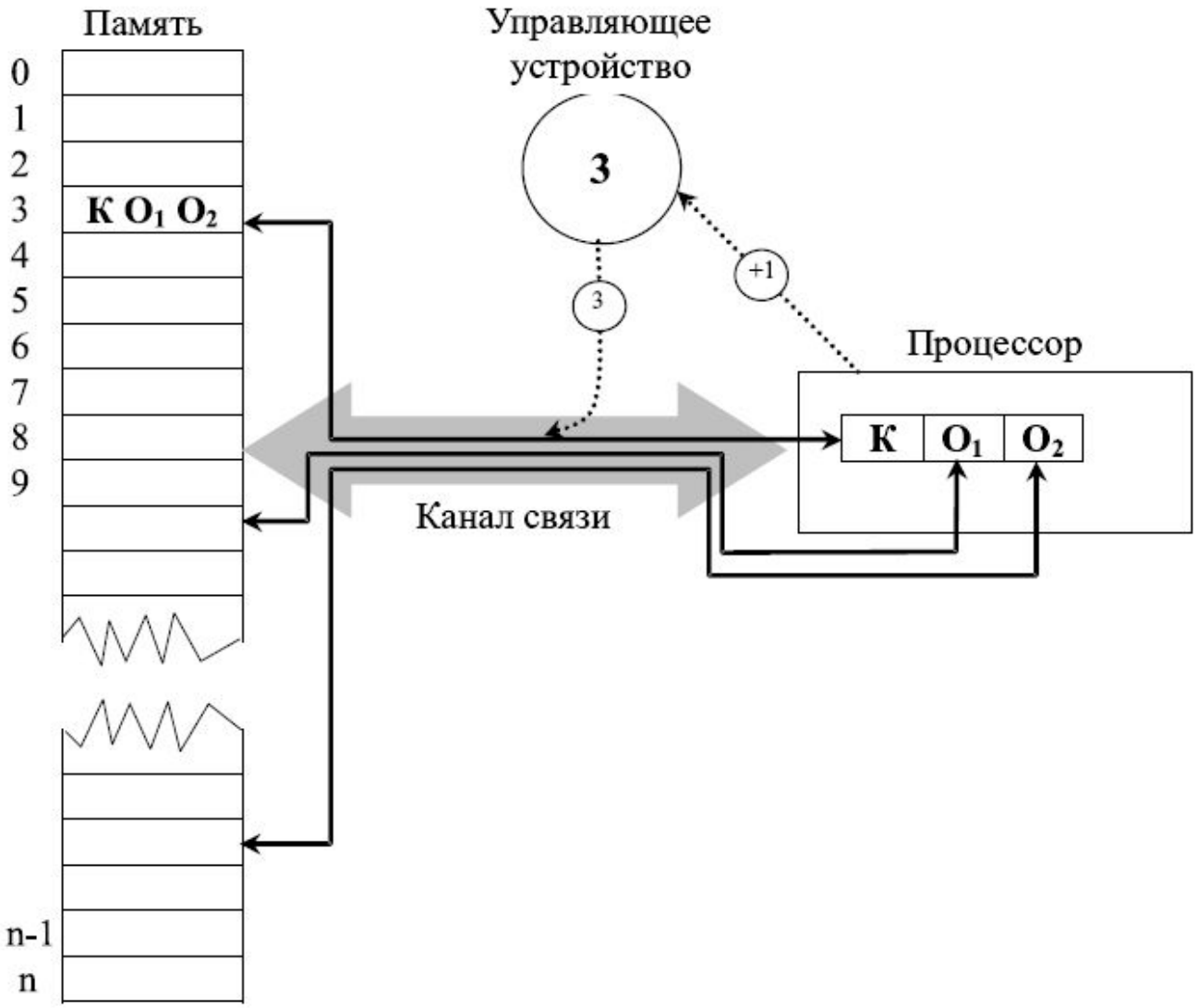


Каноническая архитектура фон Неймана

1. Три элемента вычислительной системы:
   - Память
   - Процессор
   - Управляющее устройство
2. Однородность памяти и адресация
   - Понятия ячейки, адреса и значения
3. Пассивность памяти и активность процессора
4. Роль устройства управления
5. Наличие канала связи между памятью и процессором. Работа канала осуществляется, когда
   - Требуется подать процессору команду для выполнения (активизируется устройством управления)
   - Процессору для выполнения команды требуется получить операнд (активизируется процессором)
   - При выполнении команды требуется изменение ячейки(активизируется процессором)

Дополнение канонической архитектуры: устройства ввода и вывода

Схема выполнения двухадресной



Модификация канонической схемы

a) У процессора появляются собственные ячейки со специальной адресацией, не требующие обращения к памяти. С их помощью можно уменьшать адресность команд машины, передавать промежуточные результаты вычислений от команды к команде, выполнять иные локальные действия. Они называются *быстрыми регистрами* процессора.

b) Уменьшается размер команд, в частности, их адресность. Наиболее употребляемые действия кодируются короткими последовательностями битов, отдается предпочтение одноадресным командам (это возможно благодаря регистрам процессора).

c) Команды кодируют довольно сложные действия над операндами, которые объединяют наиболее часто употребляемые последовательности элементарных операций.

d) Процессор снабжается памятью для команд, которые вероятно будут выполняться после текущей команды (эта память, называемая *кэшем команд*, заполняется в параллель с выполнением команды), а также для данных, в которой дублируются значения основной памяти с целью обеспечить следующие команды операндами (*кэш данных*).

e) Память разбивается по уровням доступа: регистры процессора, область быстрого доступа с непосредственной адресацией, область, адресация которой требует предварительного указания некоторого сегмента, ячейки которого адресуются непосредственно, и т. д. Другой вариант той же идеи — возможность переключения с сегмента на сегмент с сохранением более медленного доступа к ячейкам вне текущего сегмента.

Альтернатива канонической схемы

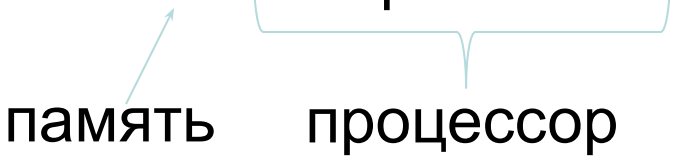
- Разрешить выполнение *всех команд*, для которых готовы операнды
- Замена хранения результата передачей его в качестве операнда ранее заблокированным командам
- Задача динамической коммутации
- *Data flow* vs. *Control flow*
- Несоответствие стиля альтернативных программ стилю, к которому привыкли программисты
- Активная (ассоциативная) память
- Консерватизм традиционных языков

Особенности традиционных языков

- Присваивание значений (переменная — аналог ячейки)
- Операторы (зависимость выполнения, последовательность)
- Структура управления (разветвления — наиболее употребительные приемы)
- Приведения
- Подпрограммы

Присваивание значений

a = <выражение>



Оператор

- Выполнение – команда
- Зависимость одного от другого (изменение памяти)

Структура управления

- Последовательность выполнения
- Принудительная передача управления
- Способ указания групп операторов – типовые приемы разветвления вычислений

Приведения

- Типов выражения и переменной в присваивании
- Округление и отбрасывание
- Контролируемые (явно указываемые) и по умолчанию
- Приведения указателей

Подпрограммы

- Типовой прием группировки команд
- Повторяемые действия
- Модульность
- Современное понимание (расширено)

Нарушения канона: побудительные причины

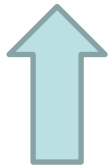
- Повышение эффективности
- Повышение выразительности
- Фиксация типовых приемов программирования
- Нарушение однородности памяти
  - Сегментация памяти
  - Быстрые регистры, кэширование
  - …
  - Содержательная трактовка хранимого
  - Выразительность
  - Надежность
- Определение традиционных и нетрадиционных языков

Традиционные языки (некоторые)

- Fortran (IV, 76, 90, 95, 2000, 2003, …)
- Algol 60
- Simula, Simula 67
- PL/1
- Algol 68
- Pascal
- C и др. ма
- Ada
- Объектно
  - Simula 6
    Object Pascal /Delphi/; *CLOS – нетрадиционный!*
- Java
  - Принципы «M машин x N языков» и «M машин + N языков»

a) Все реализуемые языки можно вложить в промежуточный язык, т. е. их модели вычислений совместимы, не противоречат друг другу;

b) Все целевые машины можно непротиворечиво представить в одной модели вычислений промежуточного языка так, чтобы трансляция программ для этой общей модели давала бы эффективный код для конкретных вычислителей



Лекция 2. Нетрадиционные модели вычислений

Повелительное и изъявительное наклонения

- Изъявительное наклонение и развитость языка
- Идеи внедрения изъявительного наклонения
  - Системы продукций
  - Системы функций
  - Коммутационные системы
  - Ассоциативные системы
  - Аксиоматические системы

Различные стили программирования

Системы продукций

- Соотношения записываются как правила вывода:

  Левая Часть => Правая Часть

- Обрабатываемые данные сопоставляются с *шаблонами* (части правила для распознавания, когда правило применимо).
  - Соответствие шаблону –> разрешение *применить правило:*
    - замена выделенного при сопоставлении фрагмента данных на что-то другое
    - однократное выполнение замены – атомарный акт вычисления

Системы функций

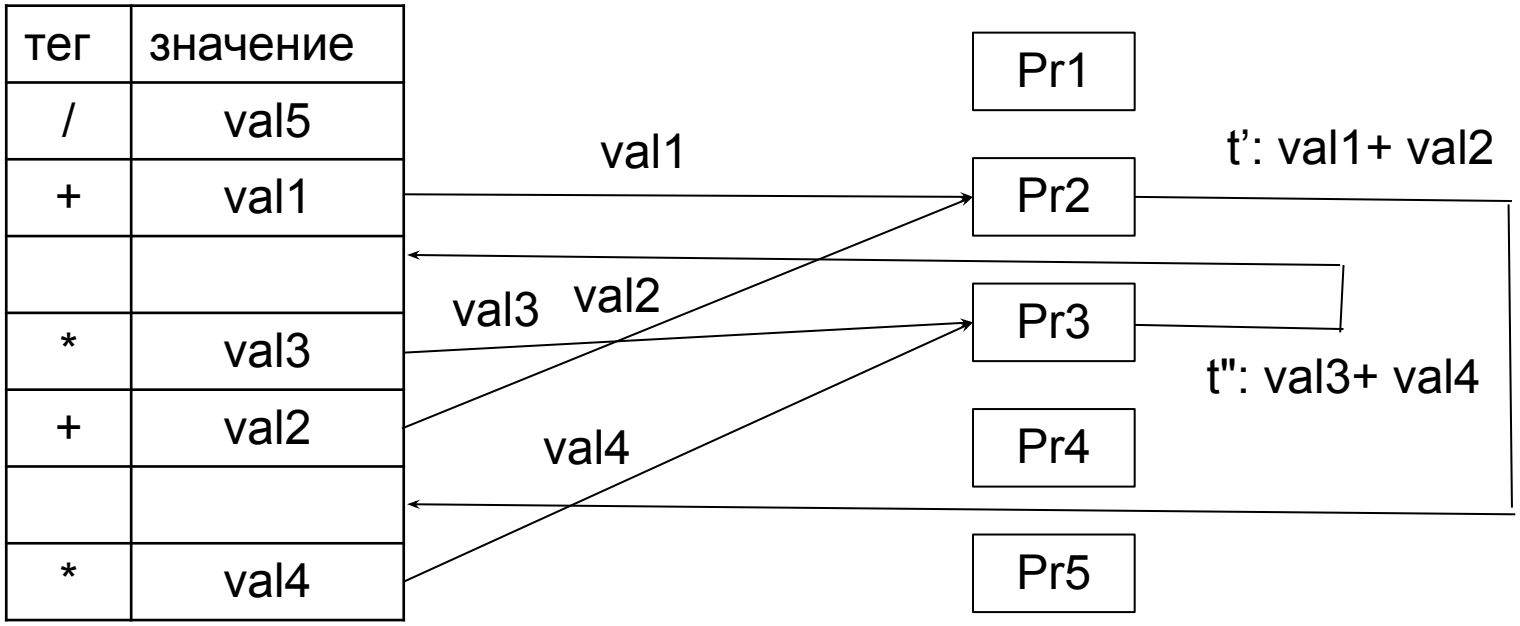
- Программа – соотношение между функциями, связанных между собой аргументами, которые в свою очередь могут быть функциями .
- Атомарный акт вычислений — это подготовка аргументов для использующей функции.
- Готовность аргументов – разрешение вычисления функции

Коммутационные системы

- Элемент системы – вершина графа (входные и выходные места)
- Дуги – каналы передачи значений
- Программа – граф с вершиной без входных мест (*генератор перерабатываемых данных*) и вершина без выходных мест (*получатель результата*)
- Вычисление – действие, связанное с вершиной или с дугой
- Активизация вычислений – поступление данных на входные места
- Возможна вложенность (структурная вершина)
- Если
  - граф без циклов, то коммутационная система – форма представления нерекурсивной системы функций
  - граф с циклами, то его можно проинтерпретировать как рекурсивную систему функций
- Но это не единственная вычислительная модель коммутационной системы (пример – сети Петри)
- Статическая коммутация vs. динамическая коммутация

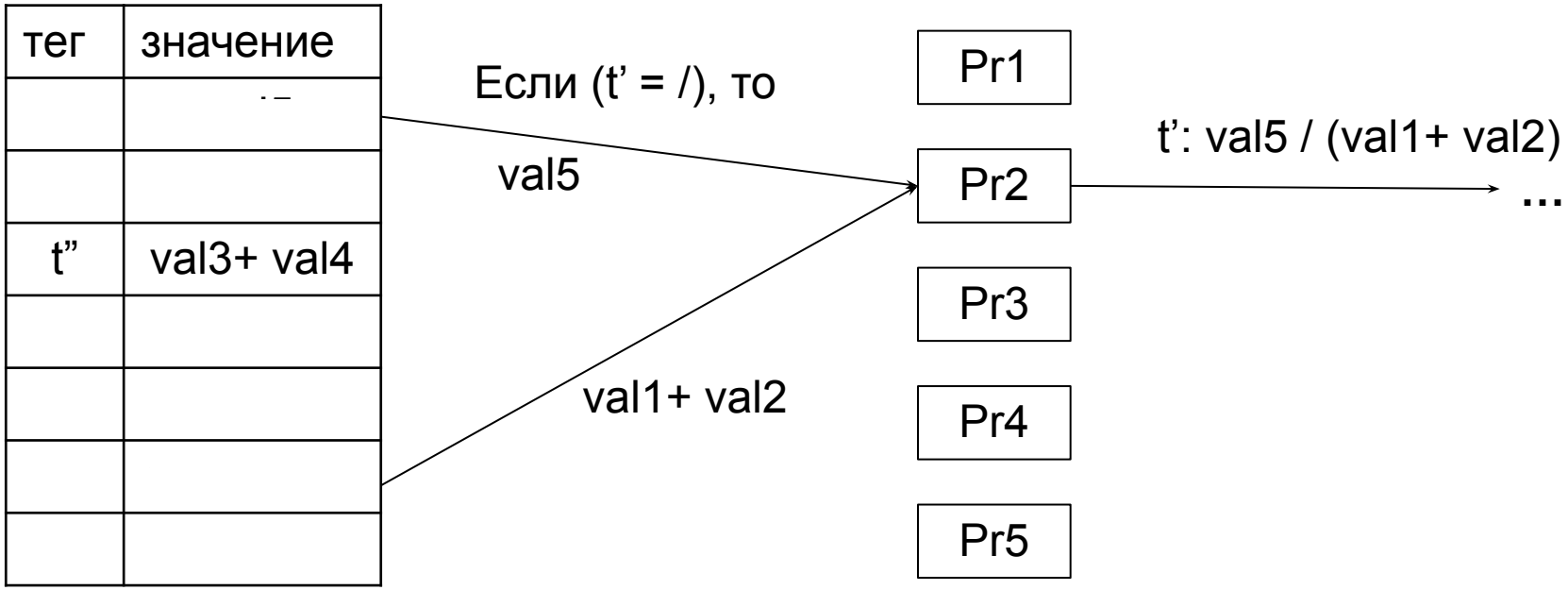
Ассоциативные системы

- Элементы системы — активные данные, представляющие собой пары: <значение, ключ>
- Спаривание элементов с одинаковыми ключами -> выполнение действия (в любом стиле)
- Результат выполнения действия – новые элементы
- Это иная форма коммуникационной схемы, более приспособленная к некоторым задачам (например, базы данных)



Ассоциативные системы

- Элементы системы — активные данные, представляющие собой пары: <значение, ключ>
- Спаривание элементов с одинаковыми ключами -> выполнение действия (в любом стиле)
- Результат выполнения действия – новые элементы
- Это иная форма коммуникационной схемы, более приспособленная к некоторым задачам (например, базы данных)



Аксиоматические системы

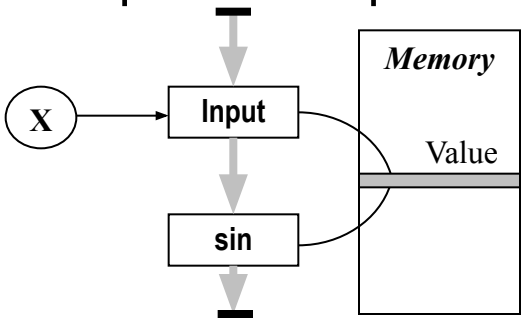
- Аксиомы
- Описание знаний на фиксированном языке
- Классическая логика – соотношение между данными
- Вывод теорем (конструктивный!), т.е. фактов – программа (элементарный акт?)
- Реализуемость – ?

Пример нарушения принципа обобщения без потерь: исчисление высказываний -> исчисление предикатов

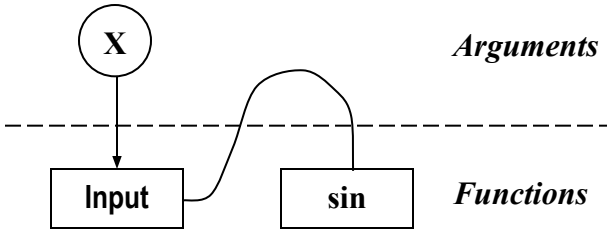
Programming styles and structuring. Program and data structuring from three points of view

Task: output (sin (input (x)))

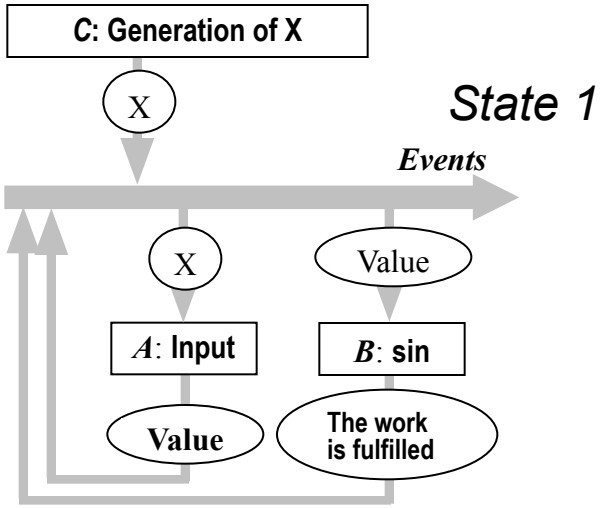
a) Operational point of view:

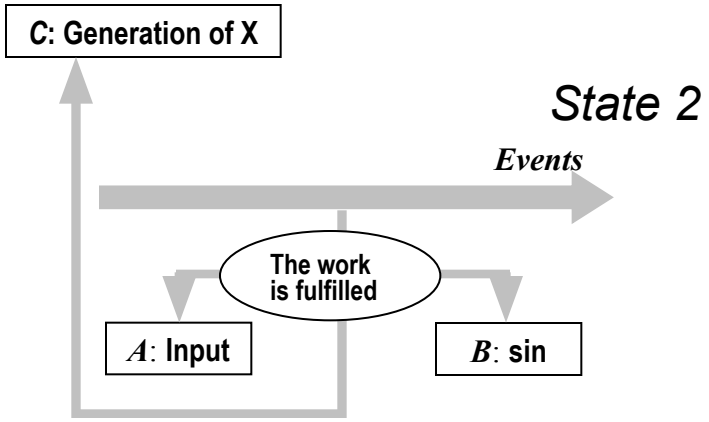


b) Functional point of view



c) Event-based point of view





a) **Memory as a unique centralized warehouse is required only for of the operational programming**
b) **Specifying arguments of functions only**
c) **Data are transferred as messages about the events the processes are waiting for**

Стили программирования

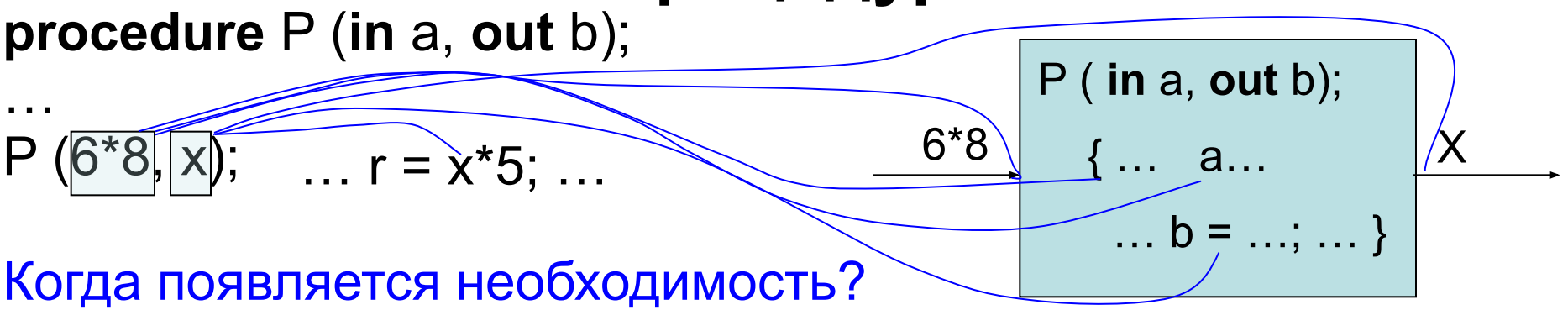
- Программирование от состояний;
- Структурное программирование;
- Сентенциальное программирование;
- Программирование от событий;
- Программирование от процессов и приоритетов;
- Функциональное программирование;
- Объектно-ориентированное программирование;
- Программирование от переиспользования;
- Программирование от шаблонов.

Лекция 3. Ленивые вычисления и функциональная модель

Ленивые и жадные вычисления

- Необходимость и возможность вычисления
- Принцип ленивости (рекурсивный ответ на вопрос):
  *Что из того, что требуется для продуцирования результатов, **может быть** и, следовательно, **должно быть вычислено**?*
- Принцип жадности:
  *Что **может быть** и, следовательно, **должно быть вычислено**, используя наличествующие сейчас данные?*
- Это управление вычислениями, берущее начало от данных
- Конкретное управление вычислениями, основывающееся на данных, — между строгими ленивым и жадным принципами
- Что эффективнее? — Когда как.
- Вырожденный случай — игнорирование обоих вопросов, соответствие управляющим структурам (детерминированный процесс)
- ***Необходимость*** и ***возможность***, их ограничения
- Функциональный язык — возможность всегда точно вычислить необходимость (в императивном случае это затруднительно).

Необходимости при выполнении процедуры



Когда появляется необходимость?

in parameters

- Реальная и форсированная необходимость
- Ingerman’s thunks (algol 60)

out parameters

- Только форсированная необходимость возможна в императивных языках

Что препятствует?

- Зависимость вычислений от контекста
- Возможность повторного присваивания

SISAL — язык однократными присваиваниями. Это паллиатив!

Метод обобщения специфичного в функциональном программировании

list of x = nil | cons x (list of x)

- Это синтаксическое представление утверждения, что список есть либо nil, либо результат применения функции cons к некоторому (абстрактному) x и уже построенному списку (list of x). Таким образом, список трех x-ов можно изобразить как

  cons x (cons x (cons x nil))

- Список можно трактовать как запись функции, в которой первый элемент есть ее наименование, а остальные элементы — ее аргументы, а точнее, как применение функции к ее аргументам (двуместная функция cons и нульместная функция nil)
- Обозначения:

  [ ] ↔ nil

  [1] ↔ cons 1 nil

  [1,2,3] ↔ cons 1 (cons 2 (cons 3 nil))

- Каким способом появились значки-функции без аргументов 1, 2, и 3, не имеет значения, лишь бы для них были определены нужные для нас другие функции, например, сложение, вычитание и т.п.

Метод обобщения специфичного (продолжение)

sum nil = 0
sum (cons num list) = num + sum list

specific to sum

sum = **reduce** add 0
add x y = x + y

(reduce f x) l

reduce f x nil = x
reduce f x (cons a l) = f a (reduce f x l)

reduce sum 0 (1 , 2 , 3 , [ ]) ⇒
1 + 2 + 3 + 0
reduce multiply 1(1 , 2 , 3 , [ ]) ⇒
1 * 2 * 3 * 1

product = reduce multiply 1
(reduce cons nil) α ⇒ α //copy of α
reduce (:) [ ] // — Haskell

reduce — функция с 3 аргументами, но она применяется только к 2 ⇒ результат — функция!

append a b = reduce cons b a
append [1,2] [3,4] = reduce cons [3,4] [1,2]
= (reduce cons [3,4]) (cons 1 (cons 2 nil))
f x a l
= cons 1 (reduce cons (cons 3 (cons 4 (cons nil)))) (cons 2 nil))
f x a l
= cons 1 (cons 2 (reduce cons (cons 3 (cons 4 (cons nil))) nil))
= cons 1 (cons 2 ((cons 3 (cons 4 (cons nil)))))

Gluing Functions Together: Composition and Map

A function to double all the elements of a list

```
doubleall = reduce doubleandcons nil
where
doubleandcons num list = cons (2*num) list
```

reduce f nil *gives expansion* f *to list*

specific to double

Further:

```
doubleandcons = fandcons double
where
double n = 2*n
fandcons f el list = cons (f el) list
```

An arbitrary function

Function composition — standard operator ".":

```
(f . g) h = f (g h)
```

So

```
fandcons f = cons . f
```

Next version of doubleall:

```
doubleall = reduce (cons . double) nil
```

Function map (for all the elements of list):

```
map f = reduce (cons. f) nil
```

Final version of doubleall :

```
doubleall = map double
```

This definition is correct:

fandcons f el = (cons . f) el
= cons (f el)
fandcons f el list = cons (f el) list

specific to double

One more example:

```
summatrix = sum . map sum
```

Lazy Evaluation: Scheme

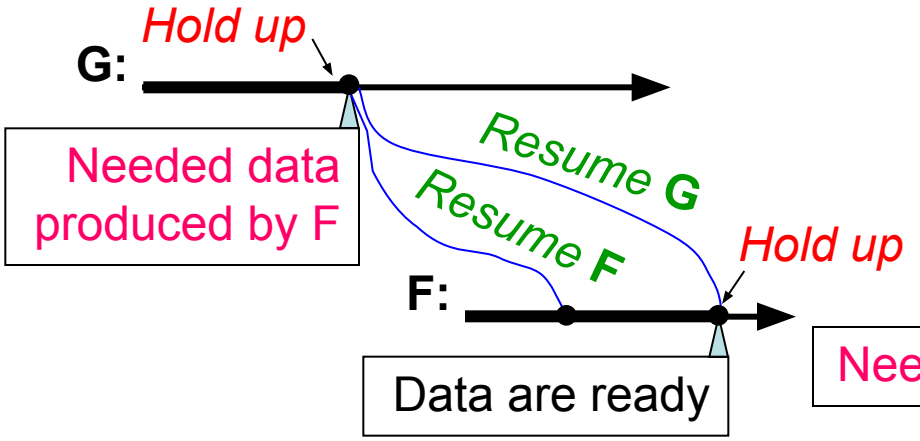
F and G — programs:

(G.F) input □ G (F input)

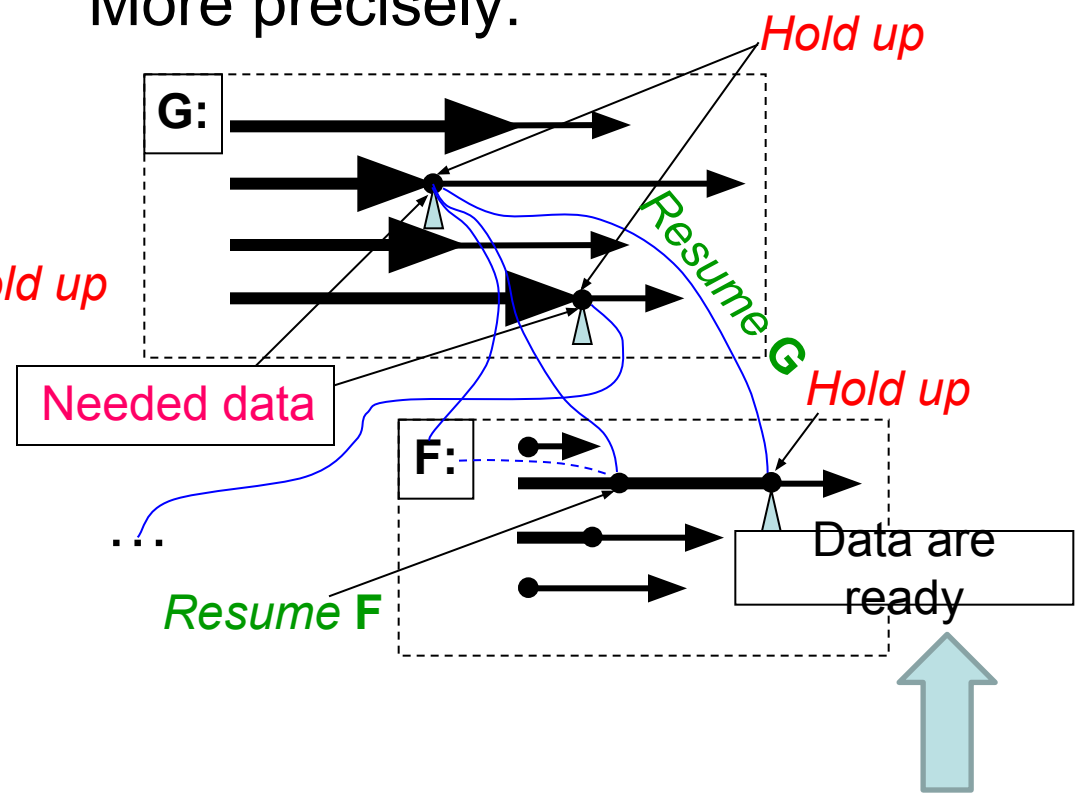
It is possible: F input → t_F; G t_F, but it is not good!

Using a temporary file

The attractive approach is to make requests for computation:



More precisely:



Лекция 4. Постулаты необходимости, их следствия. Особенности ленивых и жадных вычислений при решении различных задач

Постулаты необходимости

- Любое вычисление активизируется тогда и только тогда, когда оно (его результат) требуется для осуществимости одного или более других вычислений. Эта ситуация называется *необходимостью вычисления*.
- Любое вычисление перестает быть активным, т.е. *приостанавливается*, тогда и только тогда, когда необходимость в нем исчезает (например, требование удовлетворено).
- Вычисление программы в целом активизируется принудительно как *запрос* (необходимость) на получение *результатов* для внешнего использования (требование вывода продуцируемых данных).

Постулаты лишь фиксируют границы, в которых должна находиться любая конкретизация понятия необходимости

Следствия постулатов необходимости

1. То, что ленивые вычисления задают управление программы через потоки данных, следует из постулатов необходимости
2. Поток управления вычислением динамически формируется необходимостями вычислений составляющих программных единиц

Для императивных языков более слабое утверждение:

2. Если понятие необходимости определено корректно и вычисления программы в целом приводят к запланированным результатам, то **можно установить соответствие между последовательностью необходимостей и потоком управления**, заданным как последовательность выполняемых управляющих структур.

Жадные вычисления свойством, аналогичным (2), не обладают:

Для задания управления программы нужно не только определять наборы возможных действий, но и уметь их приоритезировать (из-за ресурсных ограничений)

Конкретизации необходимости

Для императивного языка

1. *Необходимость*, определяется правилами, задающими поток управления в программе (декларируется необходимость выполнения *следующего* оператора для всех языковых конструкций)
2. *Набор программных составляющих*, необходимых для выполнения определяется так, чтобы было справедливо:
   - в него попадают все элементы (вычислительно замкнутые программные фрагменты), вычисление которых стало возможным к этому моменту,
   - элементы этого набора вычислительно независимы друг от друга

Совместное вычисление

Для функциональных языков

Локальность всех вычислений

Завершение вычислений или их приостановка?

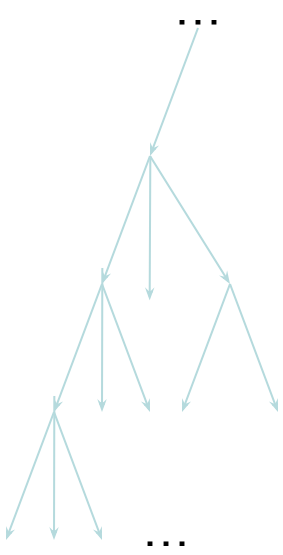
Возможность оперирования бесконечными структурами в функциональных языках

Функции высоких порядков

Пример-иллюстрация

- F строит «очень большое» дерево (например, всех возможных последовательностей ходов)
- G запрашивает поддерево фиксированной глубины, т.е. обращается к F при анализе какой-то вершины.

→ F достраивает дерево на запрошенную глубину.



- Если бы F завешалась, то требовалось бы строить уже пройденный путь дерева заново
- F никогда не вычисляется самостоятельно:

$$G \circ F \text{ /отношение запроса/}$$

Лекция 5. Решение численных задач в функциональном стиле

Метод Ньютона-Рафсона: вычисление квадратного корня

Функциональная программа не эффективна. Правда ли это?

Алгоритм:

- Начинать с первой аппроксимации **a0**
- Продолжать получение более точной аппроксимации, используя правило

  a(n+1) = (a(n) + N/a(n)) / 2

Если аппроксимации сходятся к пределу **a**, то **a = (a + N/a) / 2**

Таким образом **2a = a + N/a, a = N/a, a*a = N ⇒ a = squareroot(N)**

Императивная программа:

```
      X = A0
      Y = A0 + 2.*EPS
100 IF (ABS(X-Y).LE.EPS) GOTO 200
      Y = X
      X = (X + N/X) / 2.
      GOTO 100
200 CONTINUE
```

Мы хотим показать, что вместо этого кода можно:

- **построить простую функциональную программу**
- **получить метод ее улучшения**

В результате получится весьма выразительная программа!

Метод Ньютона-Рафсона: функциональная программа

- **Первый вариант:**

```
next N x = (x + N/x) / 2
          [a0, f a0, f(f a0), f(f(f a0)), ..]
repeat f a = cons a (repeat f (f a))
repeat (next N) a0
within eps (cons a (cons b rest))
    = b,                    if abs(a-b) <= eps
    = within eps (cons b rest),    otherwise
sqrt a0 eps N = within eps (repeat (next N) a0)
```

- **Улучшение:**

```
relative eps (cons a (cons b rest))
    = b,                    if abs(a-b) <= eps*abs(b)
    = relative eps (cons b rest), otherwise
relativesqrt a0 eps N = relative eps (repeat (next N) a0)
```

Численное дифференцирование

```
easydiff f x h = (f (x + h) - f (x)) / h
```

Проблема: при малых h ⇒ small (f (x + h) - f (x)) ⇒ ошибка!

```
differentiate h0 f x = map (easydiff f x) (repeat halve h0)
halve x = x/2
within  eps ( differentiate h0 f x )                    (1)
```

Последовательность аппроксимаций сходится, хотя не очень быстро.

```
elimerror n (cons a (cons b rest)) =
  = cons ((b*(2**n)-a)/(2**n-1)) (elimerror n (cons b rest))
```

при некотором n

```
order (cons a (cons b (cons c rest))) = round(log2((a-c)/(b-c)-1))
```

Общая функция, вычисляющая последовательность аппроксимаций:

```
improve
```

$$n \approx \log_2\left((a_{i+2} - a_i) / (a_{i+1} - a_i) - 1 \right)$$

Более эффективно:

```
within eps (
```

Пусть **A** – правильный ответ, а **B** – терм ошибки $B*h^n$.
Тогда $a(i) = A + B*2^n*h^n$ и $a(i+1) = A + B*(h^{**}n)$.

$$A = (a(i+1)*(2^{**}n) — a(i)) / 2^{**}n - 1$$

Используя ...ает свойст... ...лам можно ...

```
within eps (improve (improve (improve (differentiate h0 f x))))   (3)
```

Используя

```
super s = map second (repeat improve s)
second (cons a (cons b rest)) = b
```

Здесь repeat improve **применяется для получения все более улучшенной последовательности. Так довольно легко получен весьма сложный алгоритм**

```
within eps (super (differentiate h0 f x))                   (4)
```

Лекция 6. Ленивые вычисления: императивные примеры

Boolean выражения

$\Re \equiv \alpha\&\beta\&\gamma$: (α == false) ⇒ $\Re$ == false

(α == true) & (β == false) ⇒ $\Re$ == false

Необходимость вычислений может не появиться!

```
if  (  (precond) &&
       (init) &&
       (run) &&
       (close) )
{ printf ("OK!"); }
else …
```

Арифметические вычисления

x = (…) * 0; ⇒ x = 0;

Без вычисления (…)!

Ленивые и жадные вычисления при работе с файлами

cat File_F | grep WWW | head -1

Subject of next slide

Векторно-матричные выражения

```
vector a(n), b(n), c(n);
a = b + c + d;
```

Для матриц аналогично

The compiler does the following:

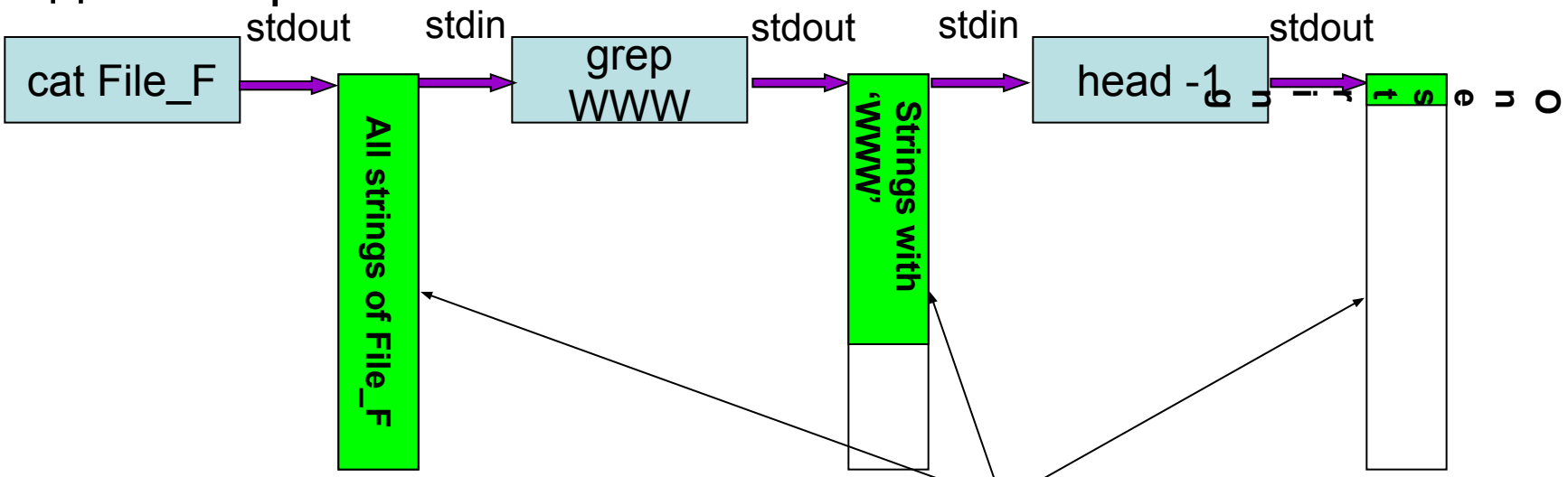
```
Vector* _t1 = new Vector(n);
    for(int i=0; i < n; i++) _t1(i) = b(i) + c(i);
Vector* _t2 = new Vector(n);
    for(int i=0; i < n; i++) _t2(i) = _t1(i) + d(i);
    for(int i=0; i < n; i++) a(i) = _t2;
delete _t2;
delete _t1;
```

Мы вынуждены создавать и удалять временные переменные!

```
for(int i=0; i < n; i++)   a(i) = b(i) + c(i) + d(i);
```

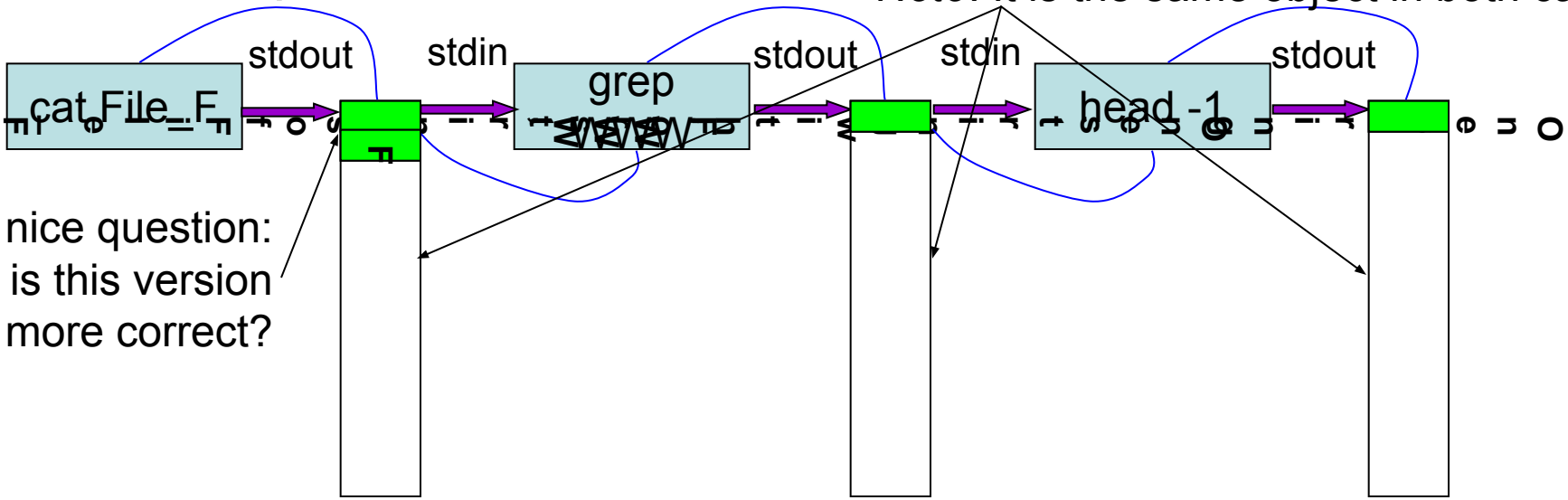
Анализ ленивого и жадного cat File_F | grep WWW | head -1

Жадный вариант:



Ленивый вариант:

Note: It is the same object in both cases!



A nice question:
is this version
more correct?

UNIX pipeline may be considered as optimization of lazy evaluation (in this case)!

Мемоизация

- **Программа F (n) = F (n-1) + F (n-2)**
  традиционный пример, когда рекурсия вредна
- Но не для функционального языка:
  - Локальность всех вычислений & независимость их от контекста
    повторение счета излишне. Можно «вспомнить» ранее посчитанное, если к такой возможности подготовиться – это мемоизация
- Прямолинейная – запоминать все
- Рациональная – запоминать необходимое!
- В императивных языках роль мемоизации – вспомогательные переменные.

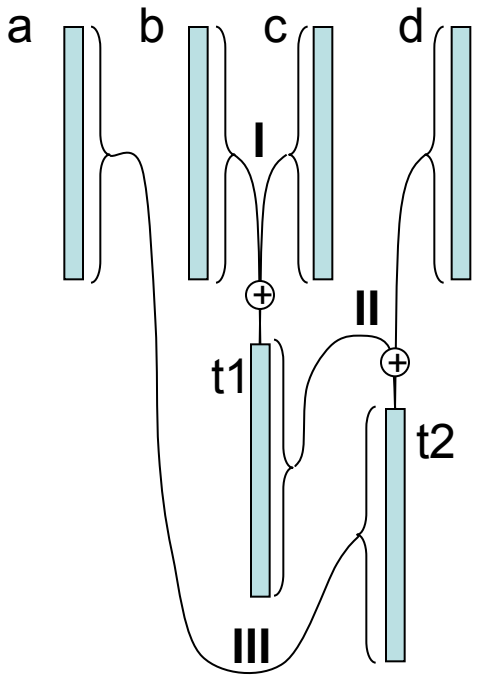
Анализ векторно-матричного примера

Consider the example more closely:

```
a = b + c + d;
t1 = b + c;
t2 = t1 + d;
a =  t2;
```

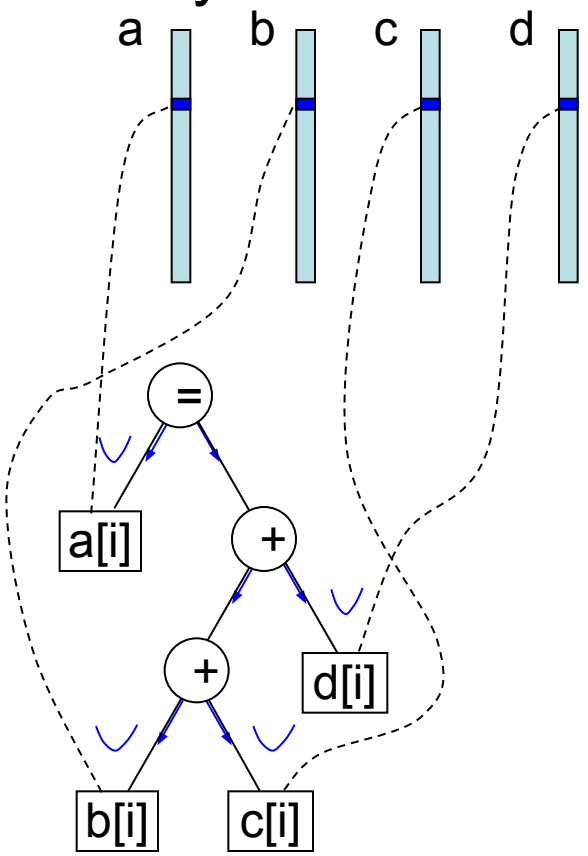
Order of calculations

Traditional scheme

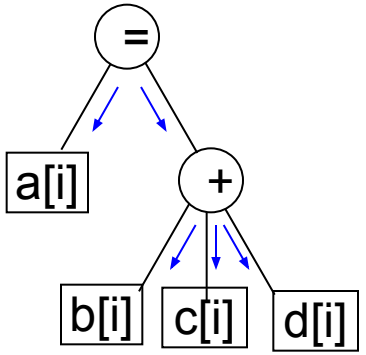


I ⇒ II ⇒ III (or **I ⇒ II U III**)

Lazy evaluation



Necessity of computation (↓) appears only when a [i] is needed



Лекция 7. Элементы сентенциального стиля программирования

Что такое сентенциальное программирование?

Sentence – предложение → грамматика, определяющая все правильные предложения

- Обрабатываемые данные имеют структуру предложения
- Обработку удобно связывать с такой структурой
- Примеры:
  - Синтаксический анализ и вычисление предложения
  - Логический вывод утверждений на основе фактов
- Распознавание элемента структуры → действие (преобразование)
- Возможность отложенных действий (планирование)
- Языковая поддержка
  - Lisp и другие функциональные языки: список – структура и данных, и программы. переработка данных, представленных в виде списков, как аргументов функции (дерево активизации функций) – частично
  - Snobol и другие языки обработки строк: *сопоставление с образцами*
  - Perl, Python и др. “скриптовые” языки – так называемые регулярные выражения
  - Prolog: данные – факты, как исходный материал для поиска
  - Рефал – язык, ориентированный на обработку древовидно структурированной информации и не только ее (сопоставление с образцами регулярные выражения и др.)

Синтаксический анализ и вычисление предложения

Грамматика

$E \rightarrow T_1 \mid E + T_2 \mid E - T_3$

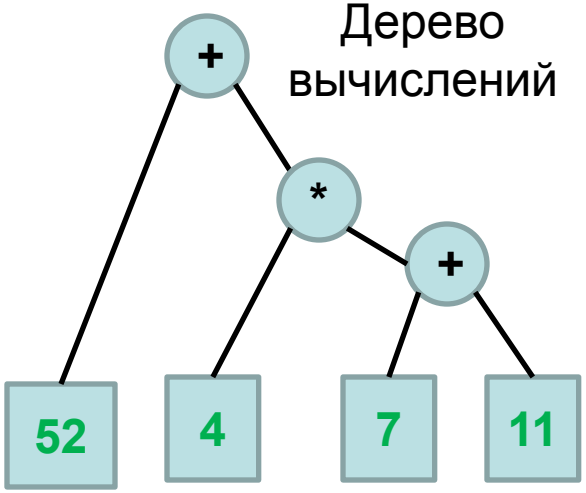
$T \rightarrow I_4 \mid T * I_5 \mid T / I_6$

$I \rightarrow (E)_7 \mid N_8$

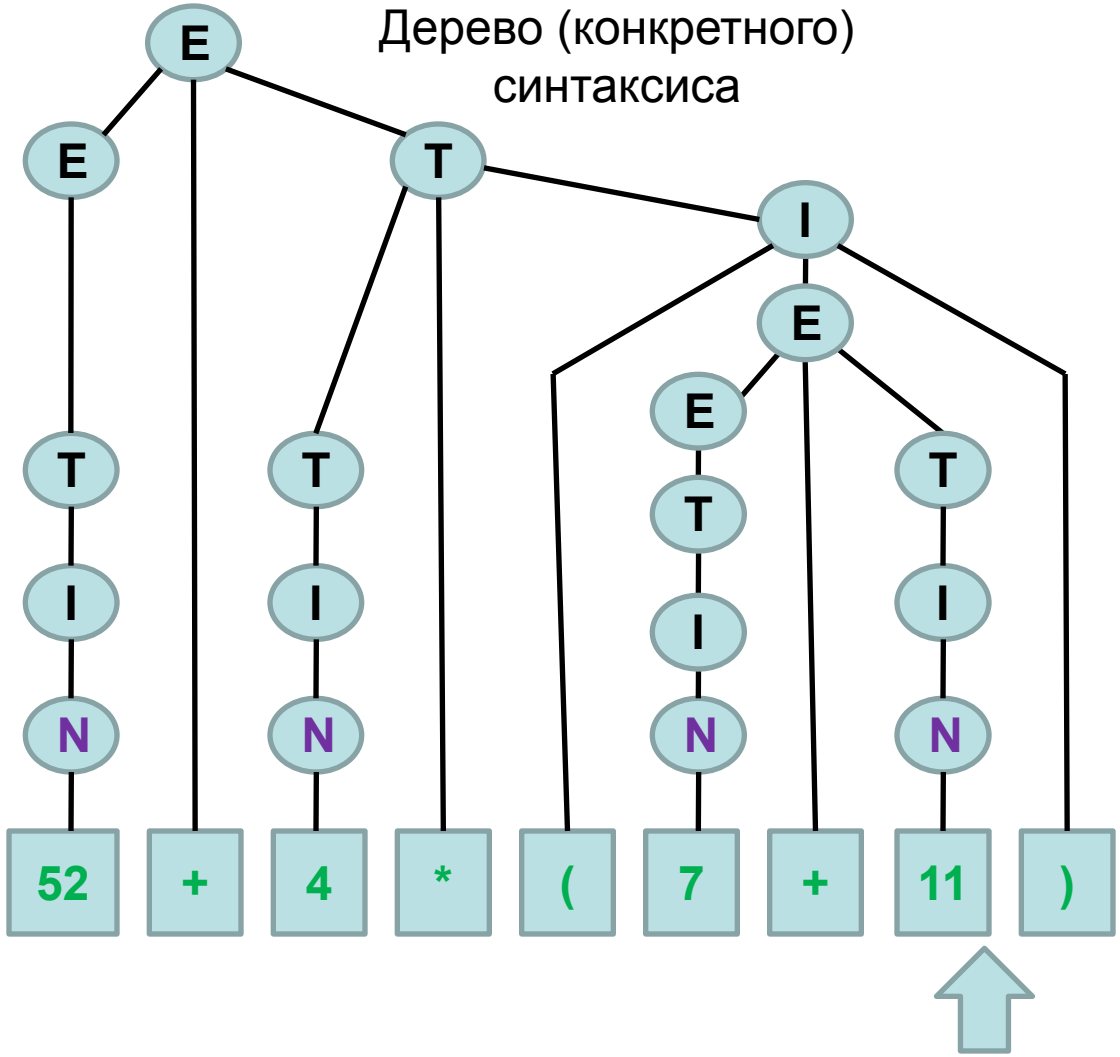
$N \rightarrow D \mid D\ N$

$N \rightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \mid 0$

Дерево вычислений



Дерево (конкретного) синтаксиса



Логический вывод утверждений на основе фактов и язык Prolog

- Семья (МАША и САША) – предикат, истинность которого *зависит* от многих фактов. Например:
  - Живут вместе (МАША и САША) – другой предикат
  - ~ Брат и сестра (МАША и САША) & Мужчина (САША) & …

- Правила вывода (например, $\dfrac{A,\ A \supset B}{B}$)
- *Цель* – доказываемый предикат (конъюнктивная форма, истинность конъюнктов – доказательство) – это *поле зрения*, т.е. подобласть данных (фактов), которые обрабатываются в текущий момент (оно обычно имеет нетривиальную (как в фон-неймановском случае) структуру)
- Поле зрения – аналог регистров процессора императивной модели вычислений
- Факты (аксиомы, леммы, …), на которые опирается доказательство, – *поле памяти программы*
- Это идея языка Prolog.

Фразовые формы, резолюция, унификация

$P1 \vee P1 \vee \ldots \vee Pn \vee N1 \vee N2 \vee \ldots \vee Nm$

Позитивные литералы Негативные литералы

$P1 \vee P1 \vee \ldots \vee Pn \leftarrow \sim N1 \vee \sim N2 \vee \ldots \vee \sim Nm$ (<заключение> ← <условие>)

Фраза Хорна: $P \leftarrow N1 \vee N2 \vee \ldots \vee Nm$ (то же, что $P \vee \sim N1 \vee \sim N2 \vee \ldots \vee \sim Nm$)

$P(a) \vee \sim Q(b,c)$ & $Q(b,c) \vee \sim R(b,c)$

b и c унифицированы, поэтому $P(a) \vee \sim Q(b,c)$ может быть резольвирована в

$P(a) \vee R(b,c)$ / R(b,c)— резольвента /

- $P(a) \vee \sim Q(b,c)$ & $Q(c,c) \vee R(b,c)$ не могут резольвироваться

Унификация переменных

$Q(x,y) \vee \sim R(x,y) \Leftrightarrow \forall x,y\ (Q(x,y) \vee \sim R(x,y))$

Переменная унифицируется с любой константой

$P(a) \vee \sim Q(b,c)$ & $Q(x,y) \vee R(x,y)$ резольвируется в $P(a) \vee R(b,c)$

P(a) — фраза без условия, $\sim P(a)$ фраза без заключения

Резолюция P(a) и $\sim P(a)$ — пустая фраза $\varnothing$

Вычисление, основанное на резолюции

Доказательство того, является ли некоторая фраза следствием теории или нет

Нисходящая (обратная) стратегия

T = { P(a) ∨ ~Q(b,c)
Q(x,y) ∨ ~R(x,y)
S(b)
R(a,b) }

Является ли P(a) следствием T?

Сначала

Правила:

a) В первой резолюции использовать добавленную фразу

b) В каждой следующей должна участвовать резольвента предыдущей

(a): ~P(a) &[1] ⇒ ~Q(a,b)
(b): ~Q(a,b)&[2] ⇒ ~R(a,b)
(b): ~R(a,b) &[4]⇒ ∅

Противоречие! P(a) доказано.

Добавим к T ~P(a):
T' = { P(a) ∨ ~Q(b,c)
[1]
Q(x,y) ∨ ~R(x,y)
[2]
S(b) [3]
R(a,b) [4]
~P(a) }

Пример Prolog программы:

```
хакер (джон)
получать_по_шее(X) :- хакер(X)
?- получать_по_шее(джон)
?- получать_по_шее(маша)
```

Какие ответы мы получим? Истина Ложь **или** Истина Недоказуемо **?**

Семантика Prolog'a

```
начальник(Фамилия, Оклад) :— служащий(Фамилия,
   Оклад), Оклад > 70000
```

Декларативная модель:

Некое лицо (Фамилия) является начальником, если он или она является служащим с окладом больше 70000

(связки: *если, и, или* — ничего более не требуется*)*

Процедурная модель:

Один из способов обнаружить начальника — это:
во-первых, отыскать служащего и,
во-вторых, проверить, превышает ли его оклад 70000

(важен порядок выполнения условий)

Модель абстрактного вычислителя

Интерпретация действий, связанных с выполнением запроса

(побочные эффекты, остановка, удаление / добавление фразы, … + прагматические соглашения)

Эти три взгляда на Prolog влияют на практику программирования!

Пример: обращение списка

```
% Метод 1 / с правом для присоединить
обр_порядок ([C|L1], L2) :-
   обр_порядок (L1,Выход), присоединить (Выход,[C],L2).
обр_порядок ([], []).
```

```
% Метод 2 %  Пример запроса:
обр_порядок (L1, [], L1). %  |?- обр_порядок
обр_порядок (L1, [X|L2], L3) :- %  ([], [a,b,c], R)
    обр_порядок ([X|L1,L2,L3).   %  R=[a,b,c]
```

Факт (unit clause) — фраза Хорна, не имеющая условий (без правой части)

Правило — фраза Хорна с одним или более условий (подцелей)

(<заключение> :— <подцель>{ , <подцель>}*)

Запрос (query, goal) — за счет унификации его параметры означиваются

Исчисление предикатов (1-го порядка) — за счет резолюций

Prolog’овские базы знаний

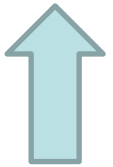
Что такое база знаний vs. база данных?

Представление знаний

- Вычислительные формализмы:
  - Дескриптивный язык +
  - Обрабатывающая структура формализма
- Этапы построения
  1. Анализ: поиск значимых сущностей и отношений между ними / *эксперт предметной области*
  2. Составление обозначений для сущностей и отношений / *программист*
  3. Семантическое определение: истинные и ложные реализации отношений / *эксперт предметной области*
  4. Аксиоматическое задание отношений фразами Prolog’а для отношений / *программист*
- Какие запросы правомерны?

Задания

- Написать на Prolog’е программу дифференцирования
- Где оканчивается адекватная применимость Prolog’а?
- Чем означивание (в двух его формах унификации и проецирования) отличается от присваивания и в чем они сходны?



Сопоставление с образцом

Сопоставление строки $\alpha \in T^*$ с образцом $\pi = \langle P_0, P_{01}, \ldots, P_{032} \rangle$, где

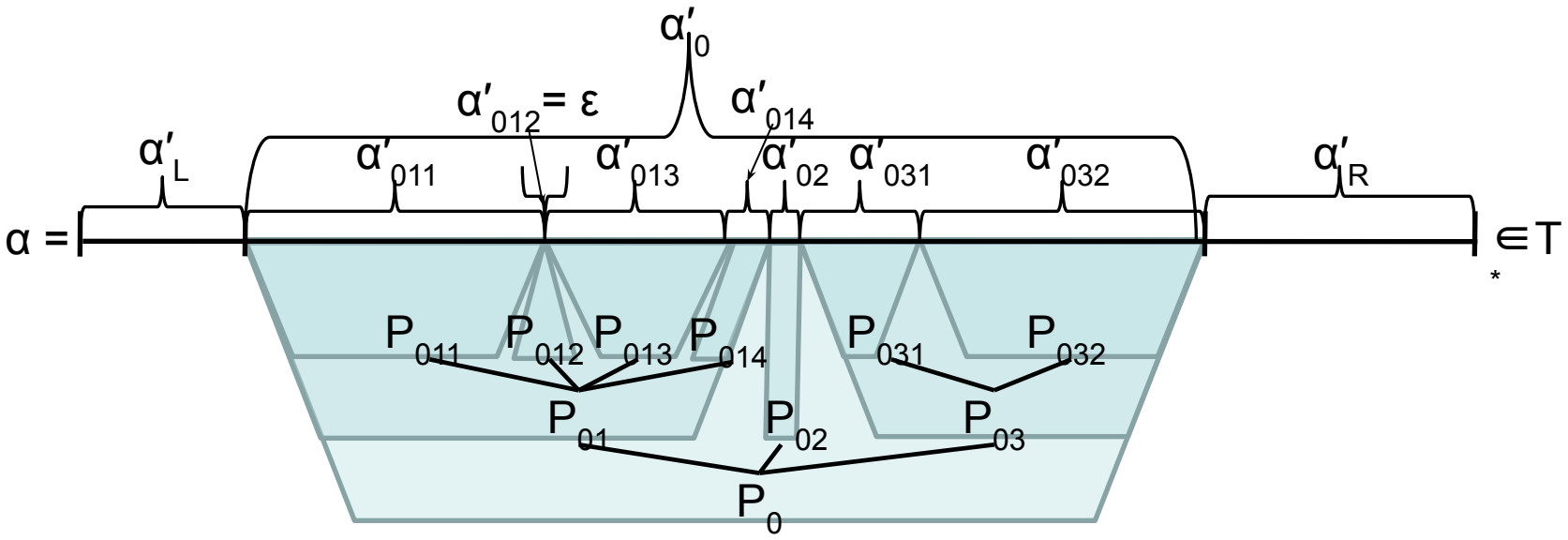
P_i, – составляющие (подтермы) элементы, из которых строится образец ($P_i \in (T \cup N \cup V)^*$,

N – имена элементов (подтермов) образца,

V – имена переменных, которые означиваются при сопоставлении,

это

1. *распознавание структуры* α, которая соответствует описанию, представленному образцом, – исход Yes; или выяснение, что такая структура не существует, – исход No.
2. *поиск* такой подстроки $\alpha'_0 = \alpha'_{011}\, \alpha'_{012}\, \alpha'_{02} \ldots \alpha'_{032}$ строки α, что
   - устанавливается соответствие между $P_0, P_{01}, \ldots, P_{032}$ и указанными подстроками α'_0. (рекурсивно)
   - все вхождения каждой из переменных из π получают одинаковые значения – *означивание переменных*, называемое *конкретизацией*



Сопоставление с образцом: примеры

Пусть $\alpha \in \{\text{‘a’, ‘b’, …, ‘z’}\}^* = T^*$

π = <‘a’> → α'_0 – любое одиночное вхождение ‘a’ в α

π = <{‘a’}*> → α'_0 – любая подстрока, состоящая из ‘a’ длины 0, 1, …

π = <{‘a’}$^+$> → α'_0 – любая подстрока, состоящая из ‘a’ длины 1, 2, …

π = <{‘a’}$^{= N}$> → α'_0 – любая подстрока, состоящая из ‘a’ длины N

N – переменная. Если она означенная, то ее значение определяет длину, если нет, то она означивается как длина α'_0.

π = <{‘a’}$^{= Na}$ X∈T*{‘b’}$^{= Nb}$ Y∈T*{‘c’}$^{= Nc}$, Na = Nb = Nc> → α'_0 – любая подстрока вида

a^N <любые символы> b^N <любые символы> c^N

Na, Nb, Nc, X и Y – переменные

π = <{‘a’}$^{= Na}$ X∈T*{‘b’}$^{= Nb}$ Y∈T*{‘c’}$^{= Nc}$, Na = Nb = Nc, Na → max>

→ α'_0 – та подстрока вида

a^N <любые символы> b^N <любые символы> c^N

для которой N наибольшее

Соглашения о стратегии сопоставления: *первое вхождение*, *все вхождения*, *максимальное вхождение* и др.

Детерминатив – элемент образца, который сопоставляется (обычно наиболее простым способом) в первую очередь с целью сузить число рассматриваемых вариантов

Рефал: основная идея

- Приспособить к практическим нуждам теоретический Алгорифм Маркова:

  $\{ \alpha_i \rightarrow \beta_i \}$, где $\alpha_i, \beta_i \in T^*$, T — алфавит символов
- Циклически повторяются в строгом порядке
  - *поиск* α_i (всегда с самого начала строки),
  - *замена* его на β_i в перерабатываемой строке.
- Завершение цикла предусматривается в двух случаях:
  - Когда ничто не может быть найдено, и
  - Когда выполняется продукция специального вида:

  $\alpha_i \rightarrow \cdot \beta_i$

Основные символы, структурированные строки

α_i строится как ***структурированная строка***, из следующего:

- ***символы*** T, не являющиеся скобками. Множество символов – **T** = T \ { **(**, **)** }
- ***конкретизационные скобки*** **k** и **.** (они могут сбалансировано появиться в строке в результате ее переработки)
- ***выражения*** — произвольные последовательности символов и скобок, сбалансированные по скобкам, в том числе и пустые последовательности, и отдельные символы. Множество всех выражений – **E** = $(\mathbf{T} \cup \{\mathbf{(},\mathbf{)}\} \cup \{\mathbf{k}, \mathbf{.}\})^*$, из которого удалены несбалансированные по скобкам строки
- ***термы*** — либо отдельные символы, либо выражения, заключенные в простые или конкретизационные скобки. Множество всех термов обозначается как **W** (**W** =(**T** ∪ **(E)**))
- ***символы переменных***, различаются по *видам*
  - s-*переменные*: S = {$\mathbf{s1},\ldots,\mathbf{sN_s}$} — могут принимать в качестве значения *только* символы из перерабатываемой строки;
  - w-*переменные*: W ={$\mathbf{w1},\ldots,\mathbf{wN_w}$} — могут принимать в качестве значения *только* термы;
  - v- *переменные*: V={$\mathbf{v1},\ldots,\mathbf{vN_v}$} — могут принимать в качестве значения *только* непустые выражения;
  - e-*переменные*: E={$\mathbf{e1},\ldots,\mathbf{eN_e}$} — могут принимать в качестве значения *произвольные* (в том числе и пустые) выражения

Рефал: продукции (операторы языка)

- Продукции Рефала принимают следующий вид:

$$\underline{\mathbf{k}}\alpha_i\underline{.}\rightarrow\beta_i \quad \text{(левая часть} \rightarrow \text{правая часть)}$$

где $\alpha_i \in (\mathbf{T} \cup \mathbf{E} \cup \mathbf{V} \cup S \cup W \cup V \cup E)^*$,

$\beta_i \in (\mathbf{T} \cup \mathbf{E} \cup \mathbf{V} \cup S \cup W \cup V \cup E\{\underline{\mathbf{k}}, \underline{.}\})^*$,

(α_i и β_i сбалансированы по скобкам).

- Перерабатываемая строка (выражение) обрамляется конкретизационными скобками (технический прием)
- Поиск $\alpha_i = X_1 \ldots X_r$, строки из левой части продукции, в перерабатываемой строке заменяется процедурой *проецирования* α_i, или построения *проекции* α_i на одну из подстрок перерабатываемой строки такую, что все переменные получают значения (означиваются)

Проецирование строки $\alpha_i = X_1 \ldots X_u \ldots X_r$ продукции $\underline{\mathbf{k}}\ \alpha_i\underline{.}\ \rightarrow \beta_i$ на перерабатываемую строку

Все следующие правила используются совместно

a) Ищется подстрока вида: $\underline{\mathbf{k}}\alpha'_i\underline{.}$ где $\alpha'_i \in (T \cup \{\underline{\mathbf{k}}, \underline{.}\})^*$;

b) $X_u \in \mathbf{T}$: α'_i представима как $\mu X_u \nu$, X_u представляет в проекции сам себя;

c) $X_u \in S$: α'_i представима как $\mu t \nu$, где $t \in \mathbf{T}$ (не скобка), и тогда $X_u \leftarrow t$;

d) $X_u \in W$: α'_i представима как $\mu \tau \nu$, где $\tau \in \mathbf{W}$ (τ — терм), и тогда $X_u \leftarrow \tau$;

e) $X_u \in V$: α'_i представима как $\mu \tau \nu$, где $\tau \in \mathbf{E} \setminus \{\varepsilon\}$ (τ — непустое выражение), и тогда $X_u \leftarrow \tau$;

f) $X_u \in E$: α'_i представима как $\mu \tau \nu$, где $\tau \in \mathbf{E}$ (τ — выражение, возможно пустое), и тогда $X_u \leftarrow \tau$;

g) все $X_1,\ldots,X_r$ должны найти свои образы в строке α'_i в соответствии с правилами (b—f), при этом значения, которые принимают одни и те же переменные в разных вхождениях, должны совпадать;

h) все символы строки α'_i должны быть сопоставлены символам и переменным $X_1,\ldots,X_r$ в соответствии с правилами (b—f).

Применение продукции $\underline{k}\alpha_i \rightarrow \beta_i$

a) построение проекции α_i на α'_i (одной из возможных),

b) построение β'_i по β_i путем замены всех вхождений переменных их значениями, полученными при проецировании α_i на α'_i;

c) замена в перерабатываемой строке ее подстроки α'_i строкой β'_i.

- Если данная продукция в данный момент неприменима, то делается попытка применить другую, текстуально следующую продукцию.
- Процесс применения продукций завершается, когда в перерабатываемой строке не остается конкретизационных скобок (***успешное завершение***), либо когда ни одна из продукций не может быть применена (***неудача***).

Разрешение неоднозначностей

При неоднозначном выборе проекции α_i на α'_i предпочтение отдается той проекции, удовлетворяющей одному из следующих критериев:

a) в конечном итоге выбор проекции приводит к успешному завершению процесса, т.е. неявно вводится недетерминизм через механизм *возвращения назад* (backtracking);

b) выбор проекции приводит к таким значениям переменных из списка символов $X_1,...,X_r$, которые оказываются короче для переменных с более ранними номерами, считая номера их первых вхождений, т.е. явно вводится прагматическое детерминирование процесса;

c) возможны иные критерии предпочтения.

Дополнение основного механизма

Цель: сужение вариантов проецирования, снижение возможной недетерминированности, как следствие, повышение эффективности вычислений, повышение наглядности описания обработки

Средство: пополнение словаря *детерминативами*. Это специальные символы, которые могут появляться в продукциях после **k** .

Собираются в **упорядоченные группы продукции**, имеющие один и тот же детерминатив

Процедура проецирования начинается с выбора группы продукций с детерминативом, выделенным в перерабатываемой строке

Примеры:

- **k**/DETERMINATIV/ … **.** → ……
- **k**ABCD+EF**.** → EFGH – замена “ABCD+EF” на “EFGH”
- **ks1**XYZ**.** → XYZ**s1** – перенос первого символа в конец

Использование детерминативов как своеобразных наименований процедур, в частности, внешних (на других языках)

Содержательный пример: дифференцирование (функция **k**/dif/**)**

1. **k**e1**.** → **k**/dif/**e1.**
2. **k**/dif/**v1**+**v2.** → (**k**/dif/**v1.**+**k**/dif/**v2.**)
3. **k**/dif/**v1***v2**.** → (**k**/dif/**v1.***(**v2**)+**k**/dif/**v2.***(**v1**))**.**
4. **k**/dif/(**e1**)**.** → **k**/dif/**e1.**
5. **k**/dif/**w1**x**.** → **w1**
6. **k**/dif/x**.** → 1
7. **k**/dif/**w1.** → 0

Продолжение примера

Дифференцирование **a*x+bx*(c+x)+d**

ka*x+bx*(c+x)+d. ⇒/1/

k/dif/a*x+bx*(c+x)+d. ⇒/2/

(**k**/dif/a*x.+**k**/dif/bx*(c+x)+d.) ⇒/2/

(**k**/dif/a*x.+(**k**/dif/bx*(c+x).+**k**/dif/d.)) ⇒/3/

((**k**/dif/a.*(x)+

((0*(x)+1*(a))

((0*(x)+1*(a))

((0*(x)+1*(a))

((0*(x)+1*(a))

((0*(x)+1*(a))

C

k

1. **k**e1. → **k**/dif/**e1.**
2. **k**/dif/**v1**+**v2.** → (**k**/dif/**v1.**+**k**/dif/**v2.**)
3. **k**/dif/**v1***v2. → (**k**/dif/**v1.***(**v2**)+**k**/dif/**v2.***(**v1**)).
4. **k**/dif/(**e1**). → **k**/dif/**e1.**
5. **k**/dif/**w1**x. → **w1**
6. **k**/dif/x. → 1
7. **k**/dif/**w1**. → 0

v1 = a*x	и	**v2** = bx*(c+x)+d	(продукция 2);
v1 = a*x+bx*(c+x)	и	**v2** = d	(продукция 2);
v1 = a	и	**v2** = x+bx*(c+x)+ d	(продукция 3);
v1 = a*x+bx	и	**v2** = (c+x)+ d	(продукция 3).

Внешние вычисления в Рефале

- Арифметические вычисления нерациональны:

  k/sum/**v1**+**v2**. → **k**/sum/ **k**/plus1/**v1**. + **k**/minus1/**v2**..

  k/sum/**v1**+1. → **k**/plus1/**v1**.

  k/sum/**v1**+0. → **v1**

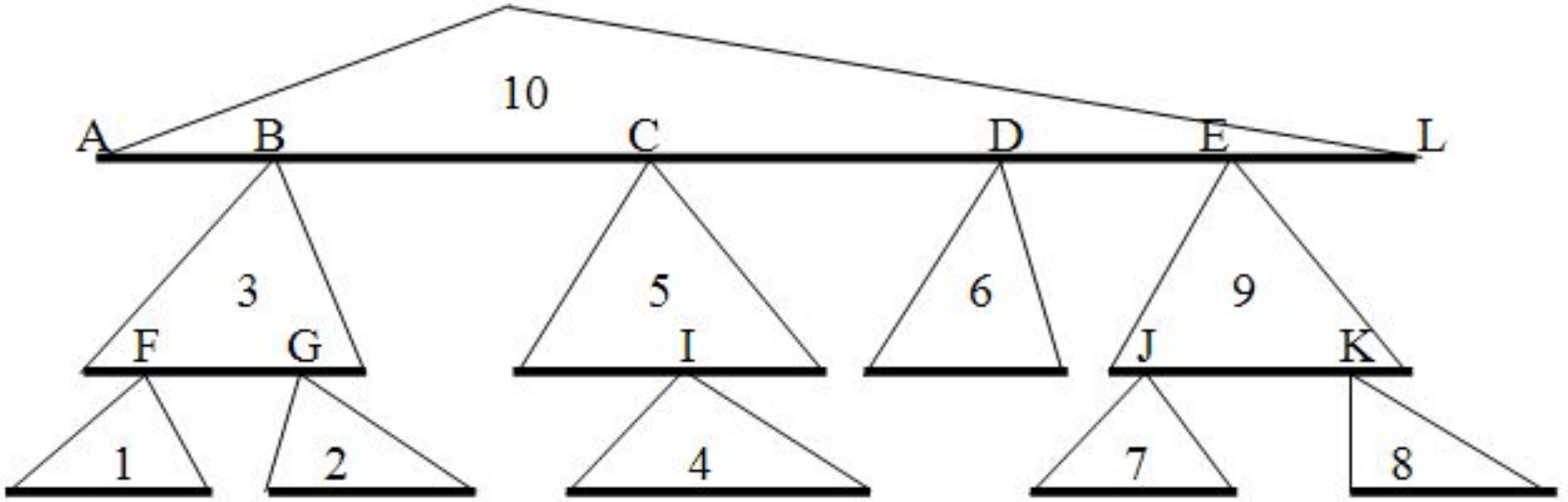
  … нужны и другие правила

- А как проще? Обратиться к другой модели вычислений:

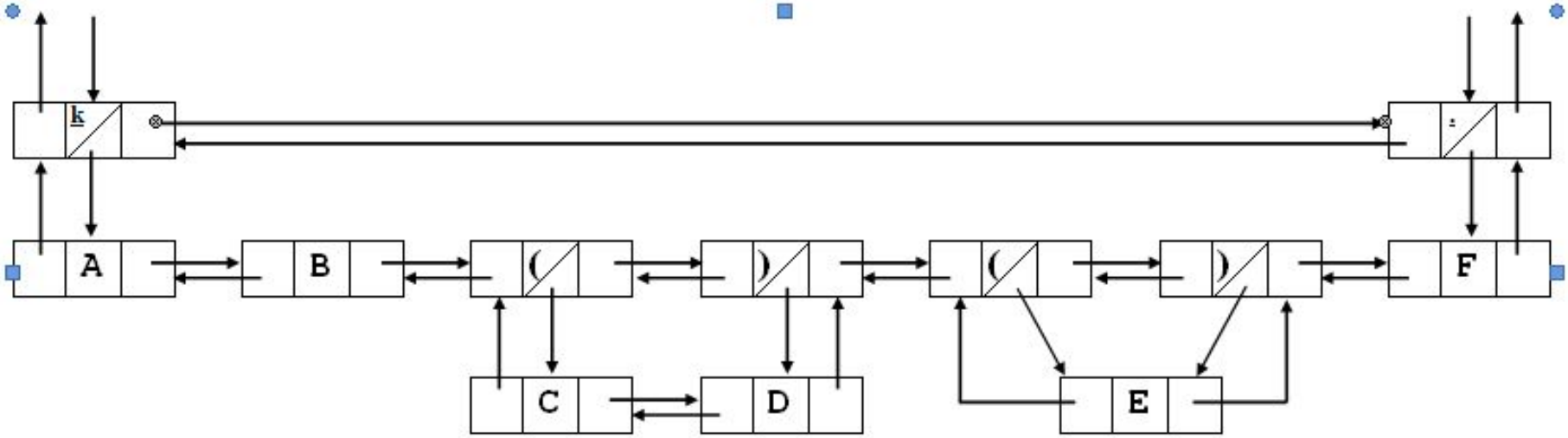
  k/sum/**v1+v2** → *результат*.

  - sum – имя внешней функции
  - **v1** и **v2** – входные параметры (для корректной работы должны быть означены как данные, соответствующие спецификациям sum)
  - *результат* – выходной параметр (приводится к строковому виду)
  - +, →, . и – можно рассматривать как оформление фактических параметров функции

Схема вычисления Рефал программы



Представление строки **k**AB(CD)(E)F**.** в поле зрения Рефал-машины



Лекция 9. Концепция «Model View Controller»

(что не удалось сделать в Delphi)

Система и ее декомпозиция

Система – набор связанных между собой и взаимодействующих компонент. Это *структура системы*

- Связь отражает возможность передачи информации
- Взаимодействие – передача информации, в частности:
  - Сигналы активизации компонент
  - Запросы и отклики на них
  - Передача данных
- Взаимодействие с окружением:
  - Передача информации от окружения – воздействие на компоненту
  - Передача информации окружению – воздействие на окружение

- *Функции системы* – все возможные ее влияния (действия) на окружение, а также действия, осуществляемые в ответ на воздействия окружения.

  Нужно всегда различать функции системы и функции компонент!

- *Состояния* системы и/или ее компонент – характеристика ее *поведения*, т.е. осуществимости тех или иных функций в данный момент. Состояние иногда рассматривается как часть структуры

При изучении, а тем более конструировании новой системы стоит *задача распознавания структуры системы, обеспечивающая выполнимость всех ее функций*

→ моделирование поведения системы

Декомпозиция и моделирование

Моделирование предполагает абстрагирование от несущественных *с точки зрения рассмотрения* деталей и выделение того, что рассматривается как главное

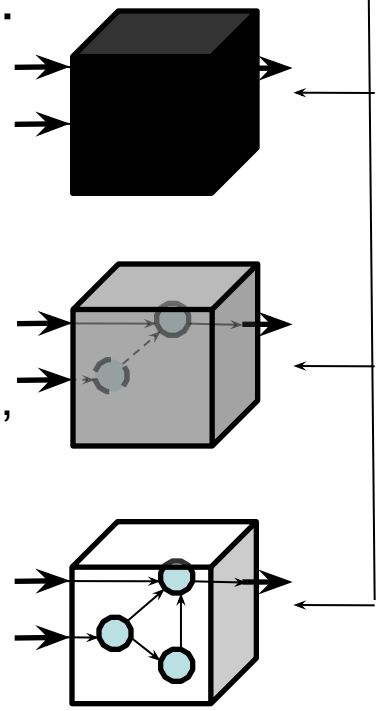
Моделирование зависит от

- Целей и решаемой задачи
- Текущего знания о системе
- Априорных установок

Где и какая декомпозиция здесь имеется ввиду?

Три подхода к изучению и конструированию системы:

- *Черный ящик*: про систему известно, какие функции она должна выполнять, характеристики функционирования . Задача – определить структуру (вариант структуры, удовлетворяющий некоторым требованиям)
- *Серый ящик*: помимо сведений о функциях известна информация о типе структуры, о некоторых ее элементах, возможно, о характеристиках функционирования и пр. Задача – реконструировать недостающую информацию, достроить систему
- *Белый ящик*: структура системы известна, есть сведения о взаимодействиях компонент. Задача – определить фактическое функционирование, возможно



Виды декомпозиции

- Стихийная (Почему это плохо? Когда приемлемо?)
- Концептуальная – уровень соглашений о системе
- Проектная – какие части выделяются в проектируемой системе, как они соотносятся с планируемыми функциями
- Организационная – разделение труда, обязанностей, ответственности и др.
- Технологическая – отображение компонентов на средства программирования
  - модульная
  - структурная (структура программной системы и структура перерабатываемых данных)
  - объектная
- Специальные – связаны с соглашениями о процессе разработки, о порядке использования ресурсов и пр.

Концепция Model View Controller

MVC – это:

- Концептуальная декомпозиция приложения, которая следует постулату разделения трех сущностей:
  - Понятия, объекты, действия и др., определяющие семантику вычислений, т. е. поведение системы на абстрактном уровне – ***Model*** (модель)
  - Понятия, объекты, компоненты интерфейса, полностью описывающие уровень взаимодействия пользователя с приложением, т.е. конкретный уровень – ***View*** (показ, представление, предъявление)
  - Понятия, объекты, компоненты управления поведением (изменение перерабатываемых данных и состояния системы) – ***Controller*** (контроллер, диспетчер)
- Шаблоны проектирования, библиотеки, языки поддерживающие концепцию
- Методический взгляд на разработку, соответствующий концептуальной декомпозиции и отвечающий на вопросы:
  - Что из того, что требуется разработать, относится к каждой из сущностей?
  - Что из этих сущностей для данной разработки является ключевой (самой сложной, неопределенной, определяющей остальное)?
  - Какие средства поддержки принятой концепции доступны в разработке?

Model View Controller

Diagrams:
- state,
- activity,
- concurrent,
- ER,
- …

Dataflow & control graphs
Code (API)

Model – структура системы и модель функционирования

Отвечает за функции системы: бизнес-логику, устройство баз данных, работу с ними и. др.
Задает уровень абстракции приложения как модель уровня конструирования

Отвечает за взаимодействие с Model и View, обрабатывает пользовательский ввод формирует запросы к View и передает данные из Model к View

Отвечает за UI
Задает конкретные представления приложения как модель уровня анализа

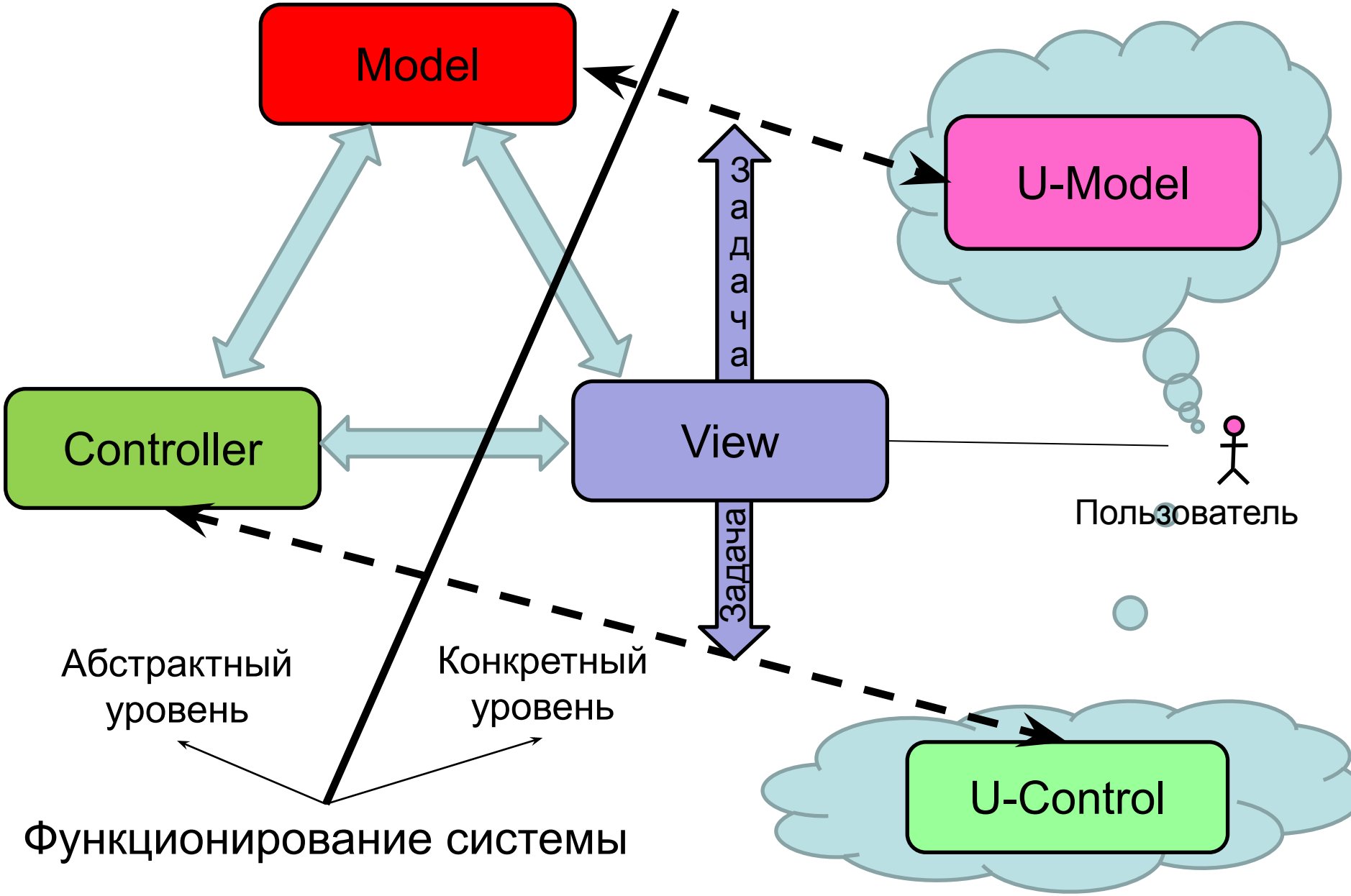
Controller – средства управления поведением

View – средства предъявления, показа

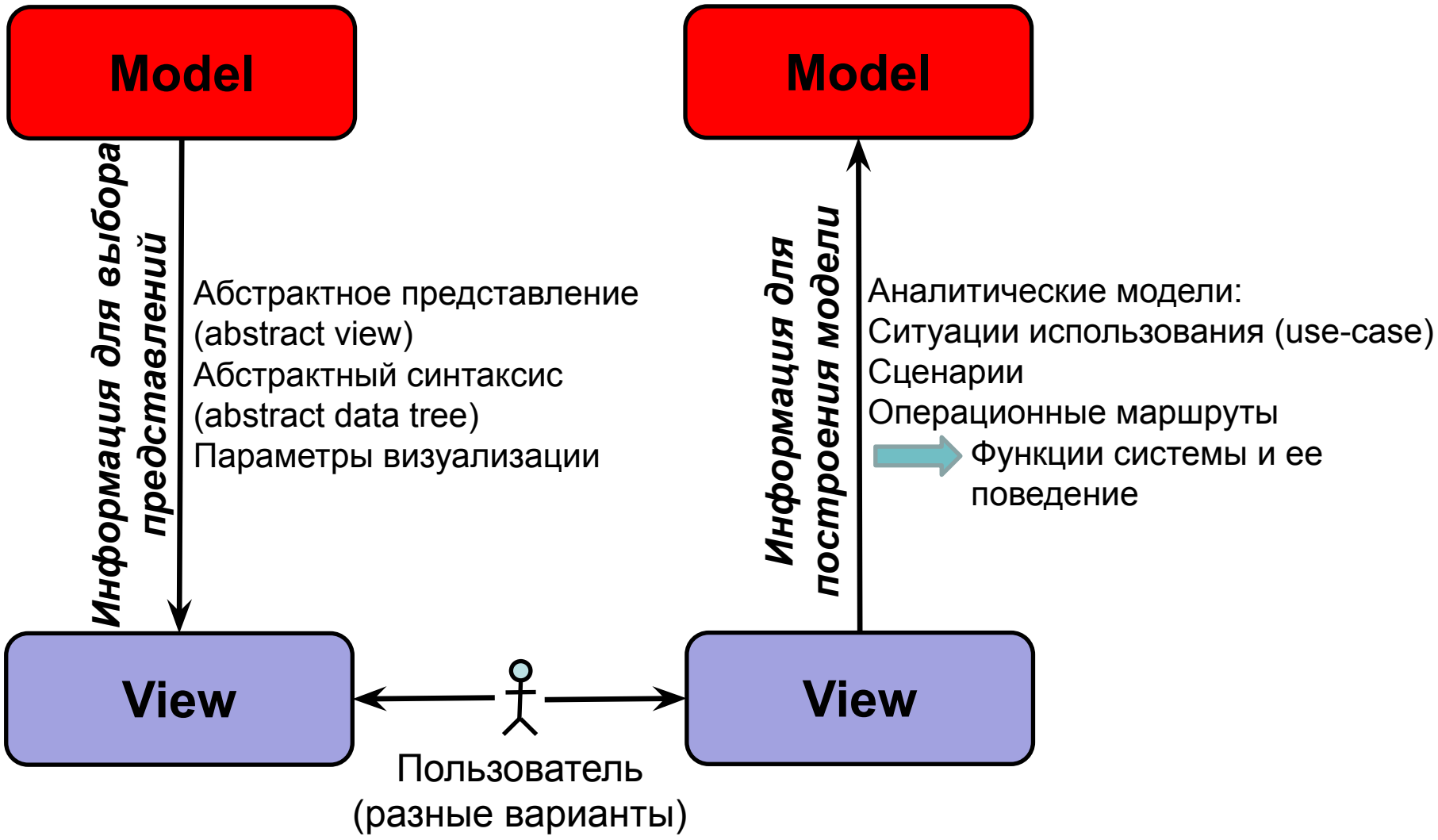
Запросы, команды, соответствующие внешним (пользовательским) воздействиям и обратно

Usage :
- use-case diagrams
- elements of UI
- info for controls

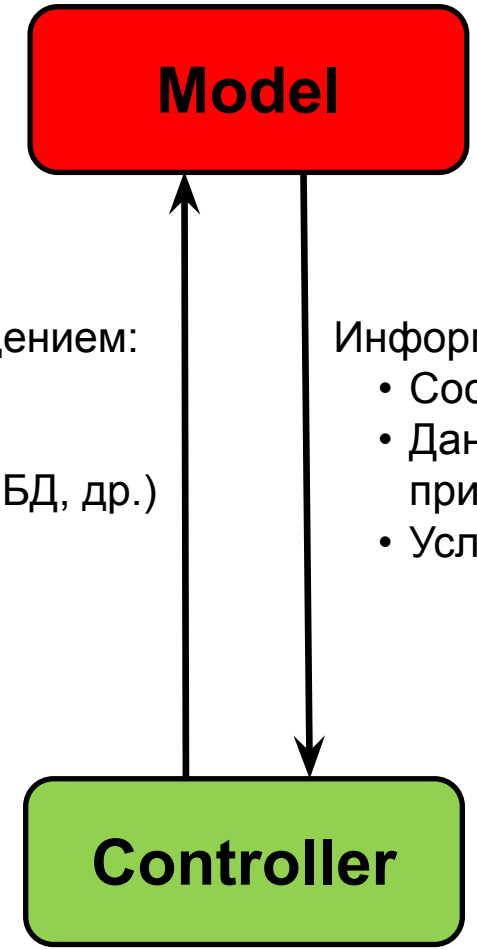
Зачем нужно выделять View?



Model ↔ View + User



Model ↔ Controller



Команды управления поведением:

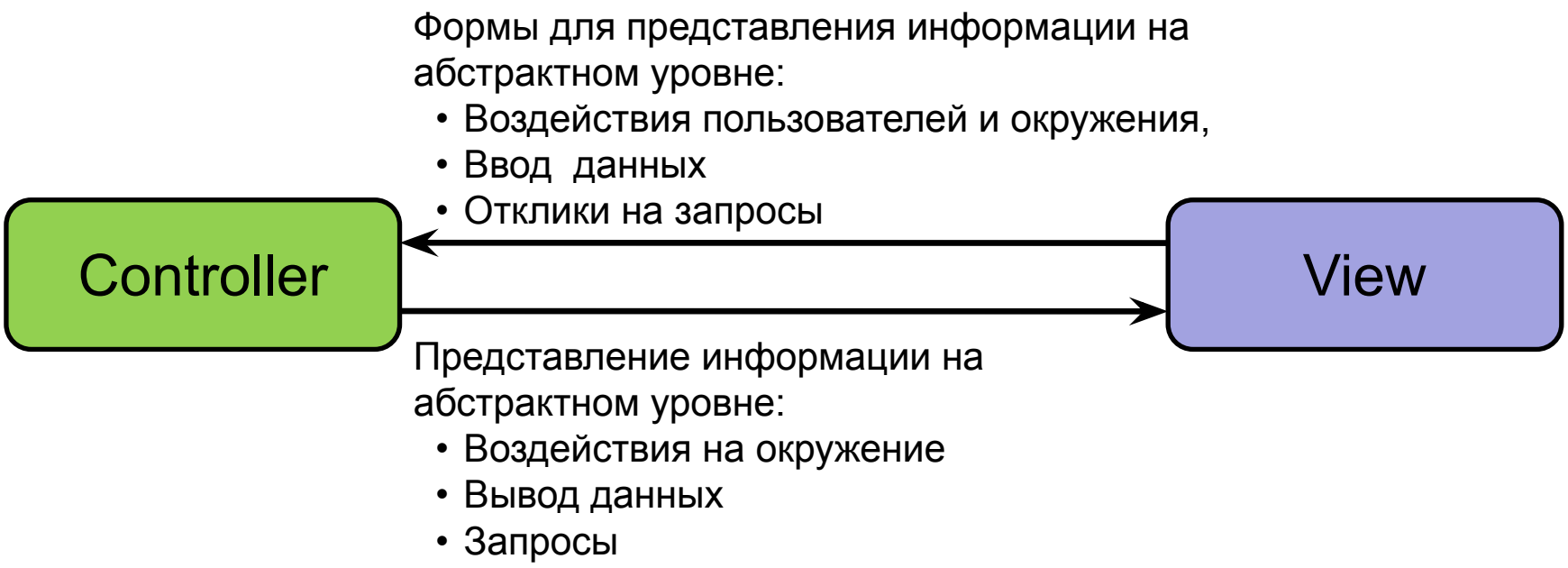
- Вводимые данные
- Сигналы от окружения
- Запросы (к обстановке, БД, др.)

Информация для управления:

- Состояние системы,
- Данные, перерабатываемые приложением,
- Условия корректности изменений

Controller ↔ View

Отображение динамики взаимодействий между абстрактным и конкретным уровнями



Когда применять MVC?

Прагматическая точка зрения:

- При реализации систем для разных пользователей (разные view, а функционирование подобно)
- Стандартизация интерфейса
- Стремление к переиспользованию
- Естественно разделение сущностей и их модульной инкапсуляции

Общая позиция:

- Никогда не игнорировать возможность концепции!
- Целесообразны следующие шаги:
  1. В начале проекта (фаза анализа) разделить три сущности
  2. Выявить, что из Model, View и Controller наиболее существенное и сложное
  3. Проанализировать аспект View и откорректировать сущности.
  4. Самое сложное – основа разработки!
- Сфера применения – разработка любого приложения!

Что не удалось сделать в Delphi?

Немного истории

- Delphi продукт года → другие системы
  (C/C++ Builder, MS Visual Studio и др.)
- Swing (Java) – одна из первых библиотек с осознанной поддержкой MVC

Delphi:

- Программирование от интерфейса (View)
- Использование палитры компонентов, инспектора объектов, поддержки проектов и др.
- Model – из области БД
- Control – стандартизованные средства управления

Почему не дошли до поддержки MVC?

- Просто разработчики не успевали!
- Затем – стихийный стандарт…

Лекция 10. Жизненный цикл программного обеспечения и его модели

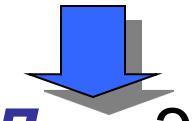
Мотивация изучения жизненного цикла и его моделей

- Жизненный цикл — база методологий
- Жизненные циклы технических и программных разработок (конструкционные материалы и окружение ПО, старение, продолжающаяся разработка (сопровождение):

- Причины изучения моделирования жизненного цикла:
  - для непрофессионалов — понимание реальных возможностей;
  - основа адекватного применения инструментов и методов разработки;
  - общие знания — ориентиры для планирования, аргументированность решений;
  - правильное распределение обязанностей в коллективе
- Соглашения, предписания, регламенты разработки и цена их игнорирования

Жизненный цикл программного обеспечения: определение

- Цикл разработки,
- Издержки после завершения разработки



Жизненный цикл — это проекция пользовательского понятия «время жизни» на понятие разработчика «технологический цикл (цикл разработки)».

Происхождение термина *жизненный цикл*

Задача методологии и жизненный цикл

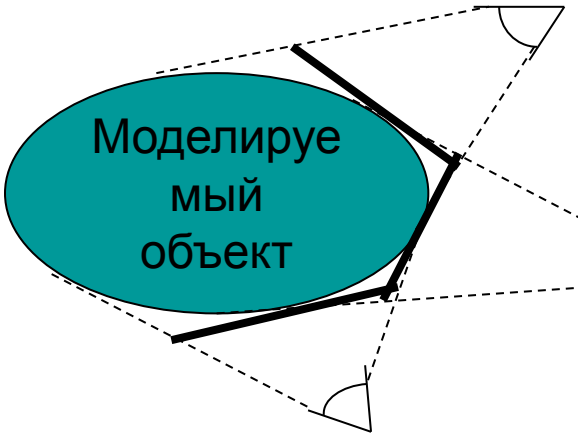
- Методологии — это инструменты, с помощью которых создание программного продукта превращается в упорядоченный процесс, а работа программиста становится более прогнозируемой и эффективной ⇒ ***планирование процесса***
- В материальном производстве: ***замысел → чертежи → проектные решения → точное воспроизводство плана***
  креативность
  - ⇒ Расчет времени и стоимости, определение требуемой квалификации и др.
  - ⇒ В традиционном производстве: ***рост технологичности & снижение креативности***
- Перенос в программирование. Разграничение двух видов деятельности:
  - ***проектирование*** (креативность)
  - ***производство*** (технологичность)

  Полностью избежать креативности не получится!
- Задача методологии: ***минимизировать творческий элемент в рутинных случаях*** ⇒ стремление разграничить:
  - план и конструирование программы
  - спецификации пользовательской потребности и план,
  - выбор инструментов работы программиста и саму работу

  ⇒ ***Появление соглашений, регламентов и предписаний, следование которым уменьшает вероятность ошибочных решений***
- Форма представления жизненных циклов в разных методологиях различна, но в основе любых представлений разработки и сопровождения программных изделий лежат общие процессы, которые ведут проекты от замыслов к удовлетворению пользовательской потребности.

Любая методология предписывает организацию этих общих процессов

Модели процесса и продукта



Модель процесса разработки:

- **Целенаправленное развитие объекта** под воздействием разработчиков
- **Ключевые понятия**: развитие, система деятельностей, субъект ⇔ объект, этапы деятельностей, инструменты деятельности, цели, результаты и продукты

Модели продукта (различные)**:**

- **Как устроен** (построен) продукт? **Для чего нужен**?
- **Один из типов моделей продукта**: проекция модели процесса, в которой игнорируется все, связанное с субъектом
  возможна реконструкция модели процесса, которая необязательно совпадает с исходной моделью процесса!

Иллюстративность модели: явное выделение некоторых аспектов для облегчения их понимания

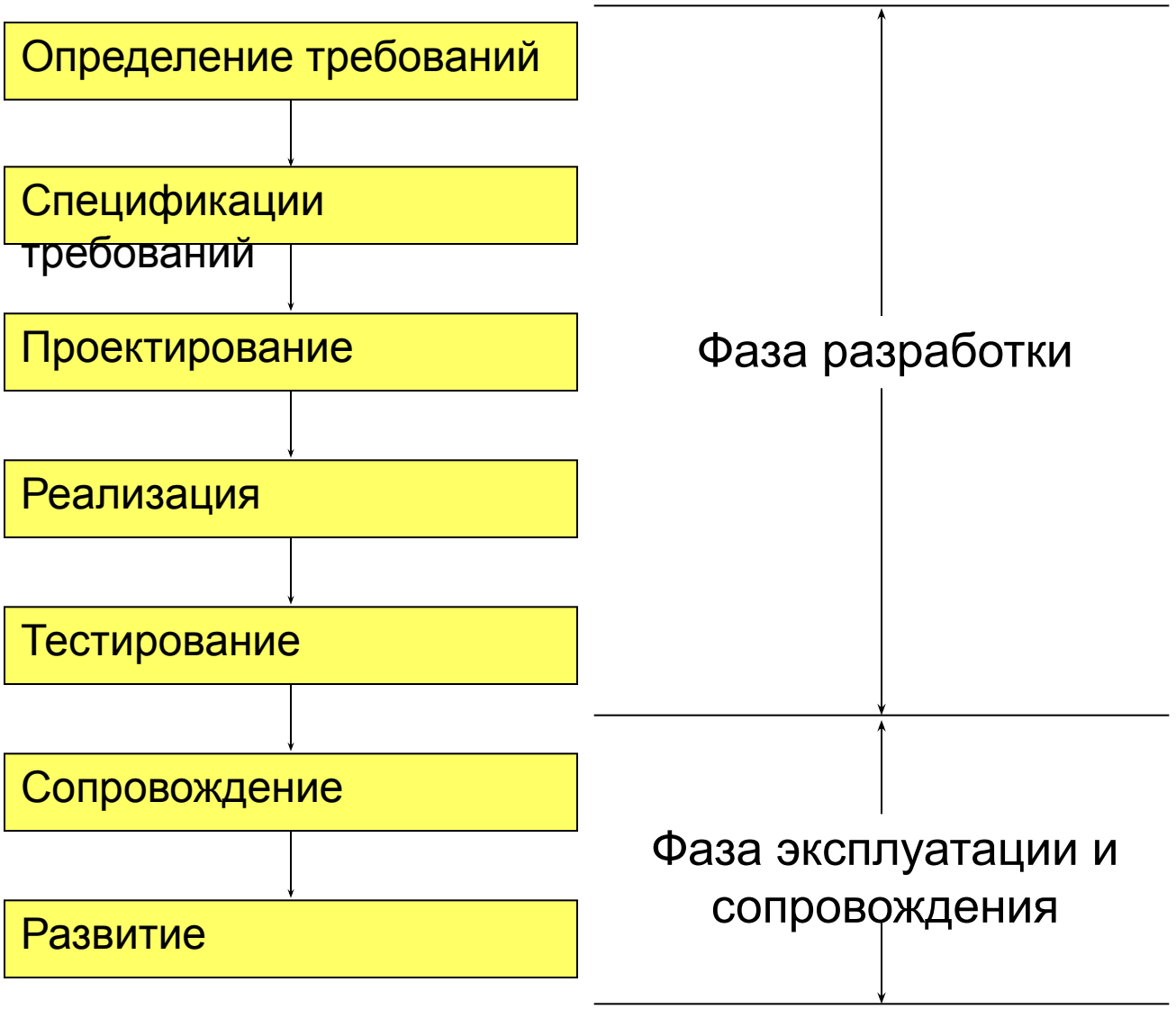
Инструментальность модели: использование ее в качестве инструмента некоторой деятельности (т.е. способствует целенаправленному развитию).

Нельзя смешивать иллюстративные и инструментальные модели

Вопросы в связи с моделью: **Что будет, если…?** и **Соответствует ли…?**

Лекция 11. Классические модели

Общепринятая модель жизненного цикла программного обеспечения

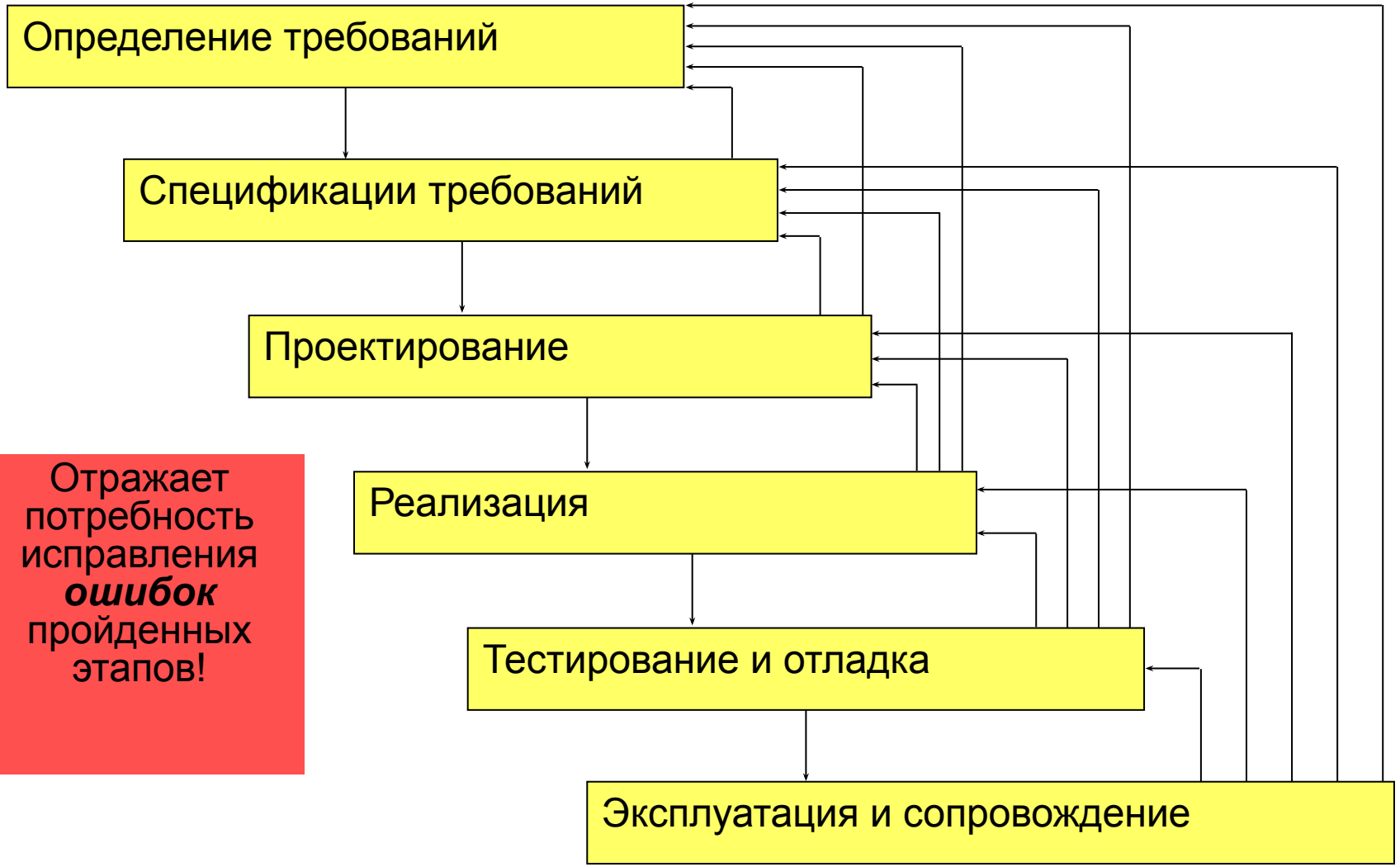


Общепринятая модель — источник базовых понятий

Начало разработки — *идентификация потребности*

- *Определение требований* — определяются: контекст задачи, ожидаемые функции ограничения. Проекта еще нет.
- *Спецификации системы в соответствии с требованиями* — Определяется поведение системы: ***что*** она должна делать. Фактическое начало работ
- *Проектирование (конструирование, дизайн)* — определяется декомпозиция системы /архитектура & результат ее построения/:
  как достигается соответствие спецификациям
- *Реализация (кодирование)* — разрабатываются модули (в модели нет этапа *интеграции*):
  что обеспечивает требуемый результат
- *Тестирование* — проверка соответствия спецификациям
- *Сопровождение* — поддержка использования продукта
- *Развитие* — поддержка эволюции системы (*новый проект*?)

Классическая итерационная модель



Отражает потребность исправления ***ошибок*** пройденных этапов!

Исправление ошибок или адаптивность проекта?

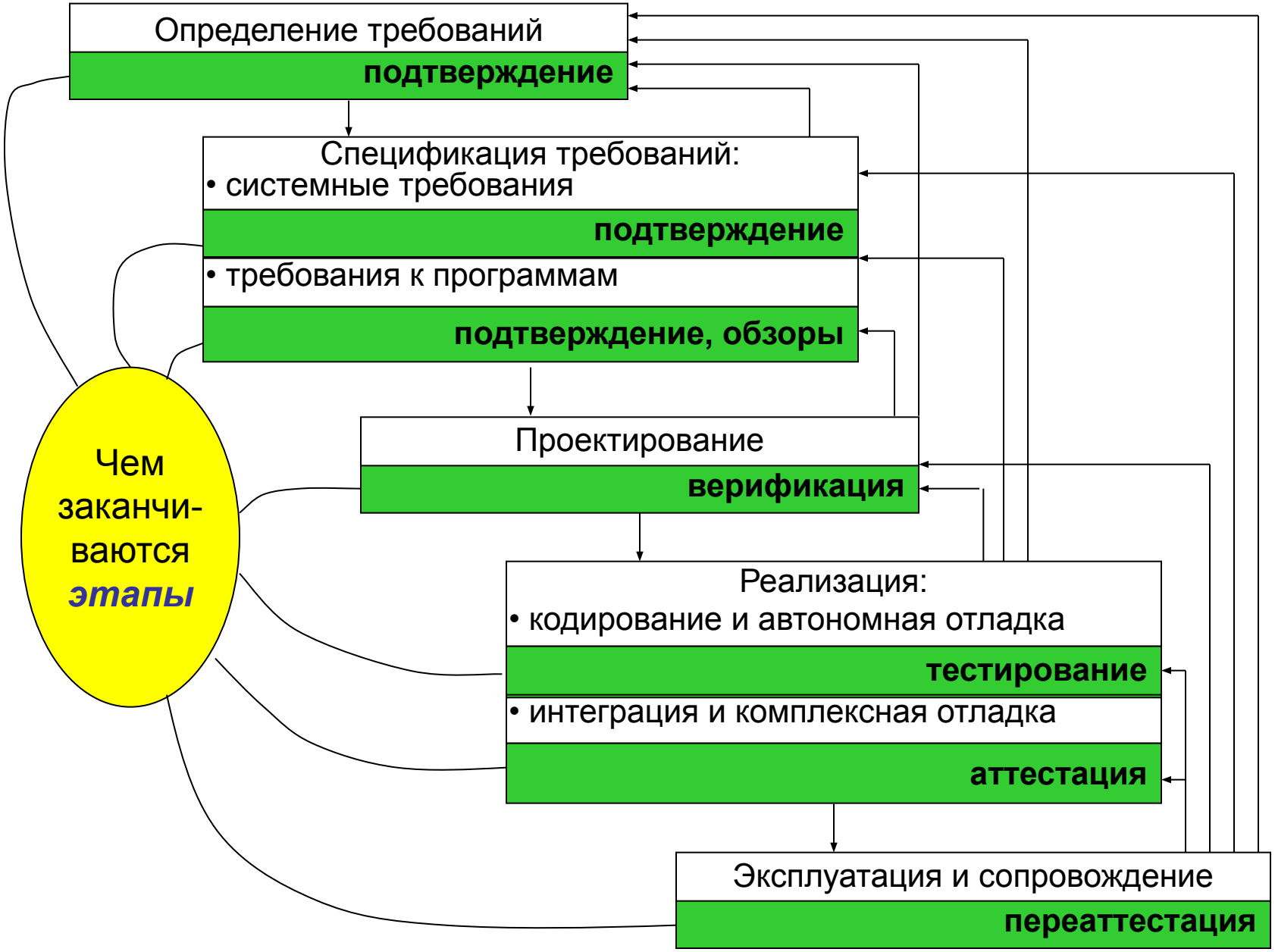
- Классическая итерационная модель — абсолютизация возможности возвратов для *исправления ошибок*, т.е.
- Идея завершенности пройденных этапов — предсказуемость
- Стратегия итеративного наращивания возможностей ослабляет требование завершенности
- ООП + CASE-системы — принципиальная возможность поддержки итеративного наращивания (не более!)
- Адаптивность проекта — альтернатива предсказуемости
- Адаптивность должна закладываться в проект на самых ранних этапах его развития

Задание: Приведите примеры, когда

a. адаптивность вредна для разработки,
b. поддержка адаптивности не помогает справиться со сложностью разработки.

Ответы обоснуйте!

Каскадная модель



Каскадная модель: новые понятия

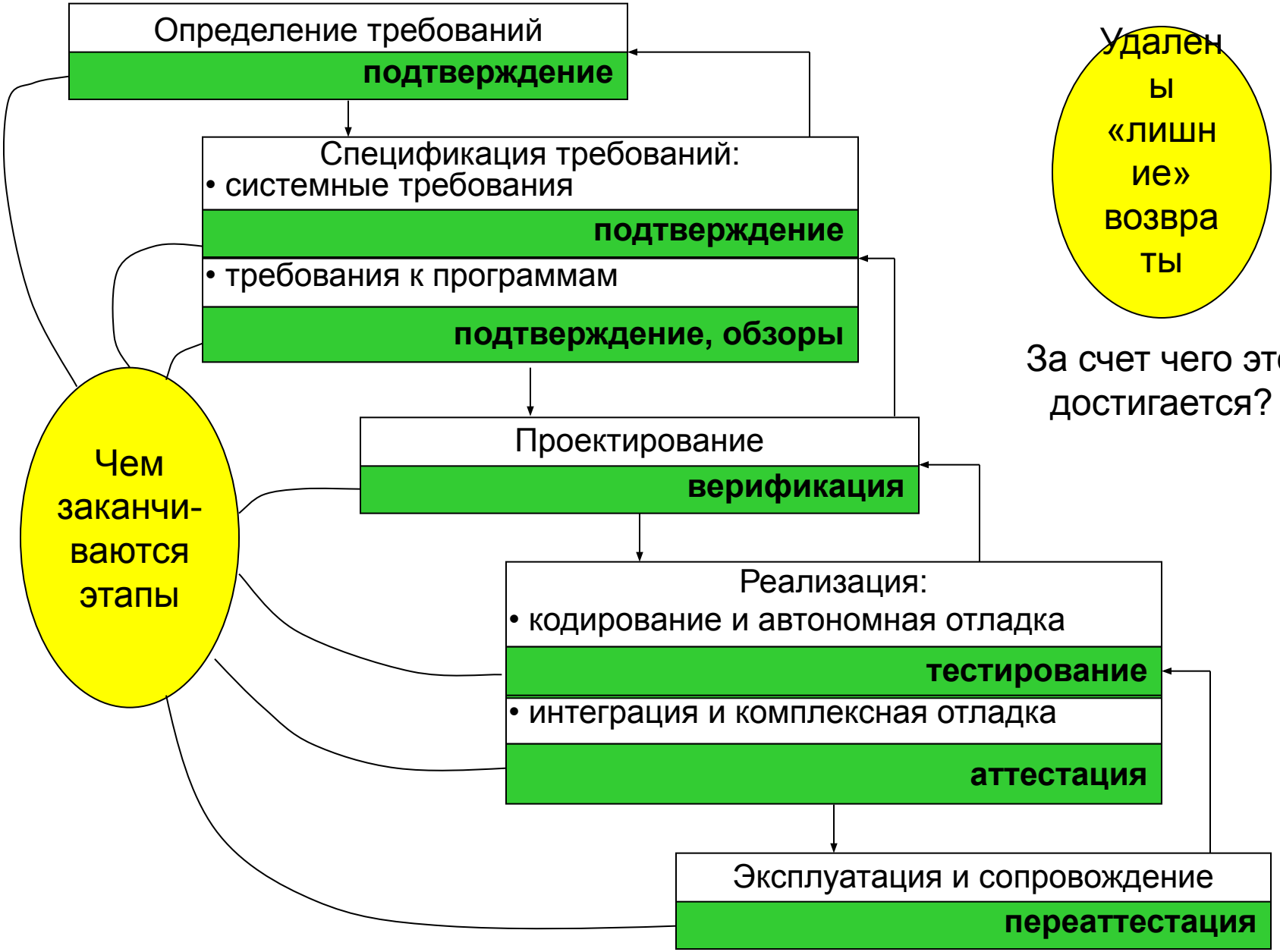
Характерные черты каскадной модели:

- завершение этапов *проверкой* полученных результатов
- циклическое *повторение* пройденных этапов

Чем заканчиваются этапы?

- *Подтверждение* — разного рода документальные согласования
- *Обзор* — документ, предоставляемый для согласования
- Проверка полученных результатов на соответствие целям:
  - *Верификация* — отвечает на вопрос, правильно ли создана (декомпозиция, программная система и др.)
  - *Аттестация* — отвечает на вопрос, правильно ли работает, т.е. будет использоваться (в первую очередь — программная система)
  - *Переаттестация* — аттестация, которая устанавливает насколько хорошо система соответствует пользовательским запросам

Строгая каскадная модель



Строгая каскадная модель: дополнительные требования к разработке проекта

- Минимизация возвратов за счет ликвидации переходов через уровни
  - ужесточение проверок (*барьеров*)
  - переход вниз сопровождается передачей исходного материала для следующего этапа — задание на разработку
- Причины невыполнения задания:
  - i. оно *противоречиво*, т.е. содержит несовместные или невыполнимые требования;
  - ii. не выработаны *критерии* для выбора одного из возможных вариантов решения

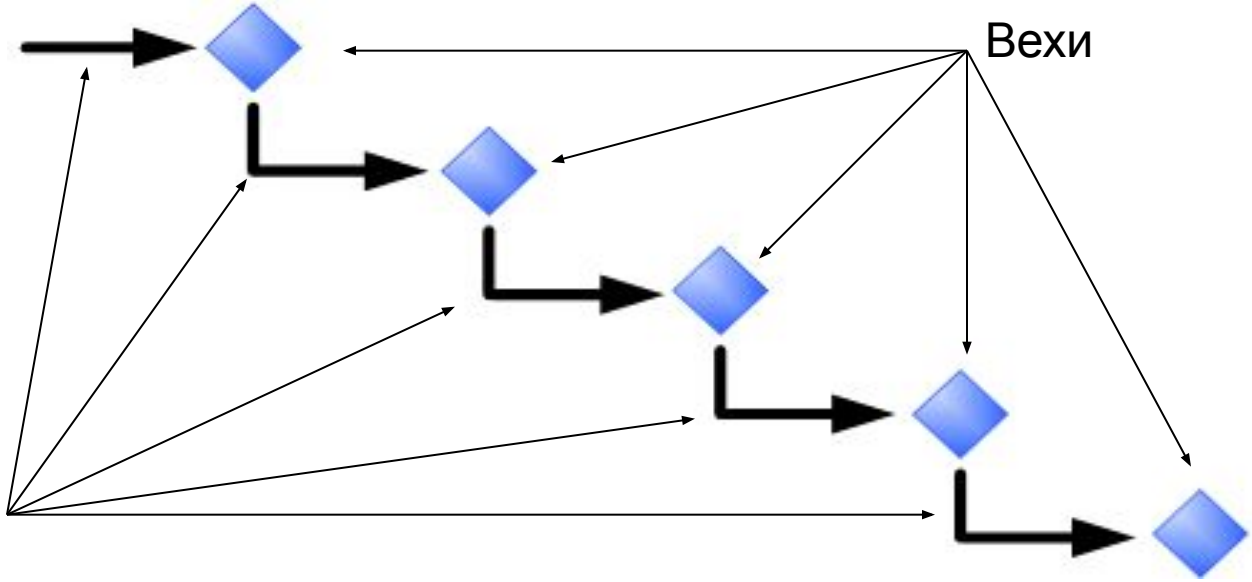
⇒ (i и ii) — ошибка предыдущего этапа → он возобновляется:

  - a. ошибка ликвидируется → переход к следующему этапу (через барьер)
  - b. невозможность исправления — это ошибка более раннего этапа → он возобновляется

- Два существенных момента (отражающих реальности разработок проектов):
  - точное разделение работ, заданий и ответственности разработчиков и тех, кто инициирует переход к следующему этапу — *шаг к осознанию фактического разделения труда*
  - малые циклы между соседними этапами, в результате достигается компромиссное задание — *совместное выполнение соседних этапов, т.е. их **перекрытие***

Однако в каскадной модели оба момента отражаются лишь

Каскадная модель MSF



Вехи (контрольные точки) используются в качестве точек оценки и перехода от одной фазы к другой.

Все задачи одной фазы должны быть завершены до того, как начнется следующая фаза.

Каскадная модель работает, когда на начальном этапе проекта можно четко определить *неизменный набор требований* к разрабатываемому решению.

Оценка: Слабее рассмотренной ранее строгой каскадной модели, но применимость характеризуется верно

Вопросы и задания

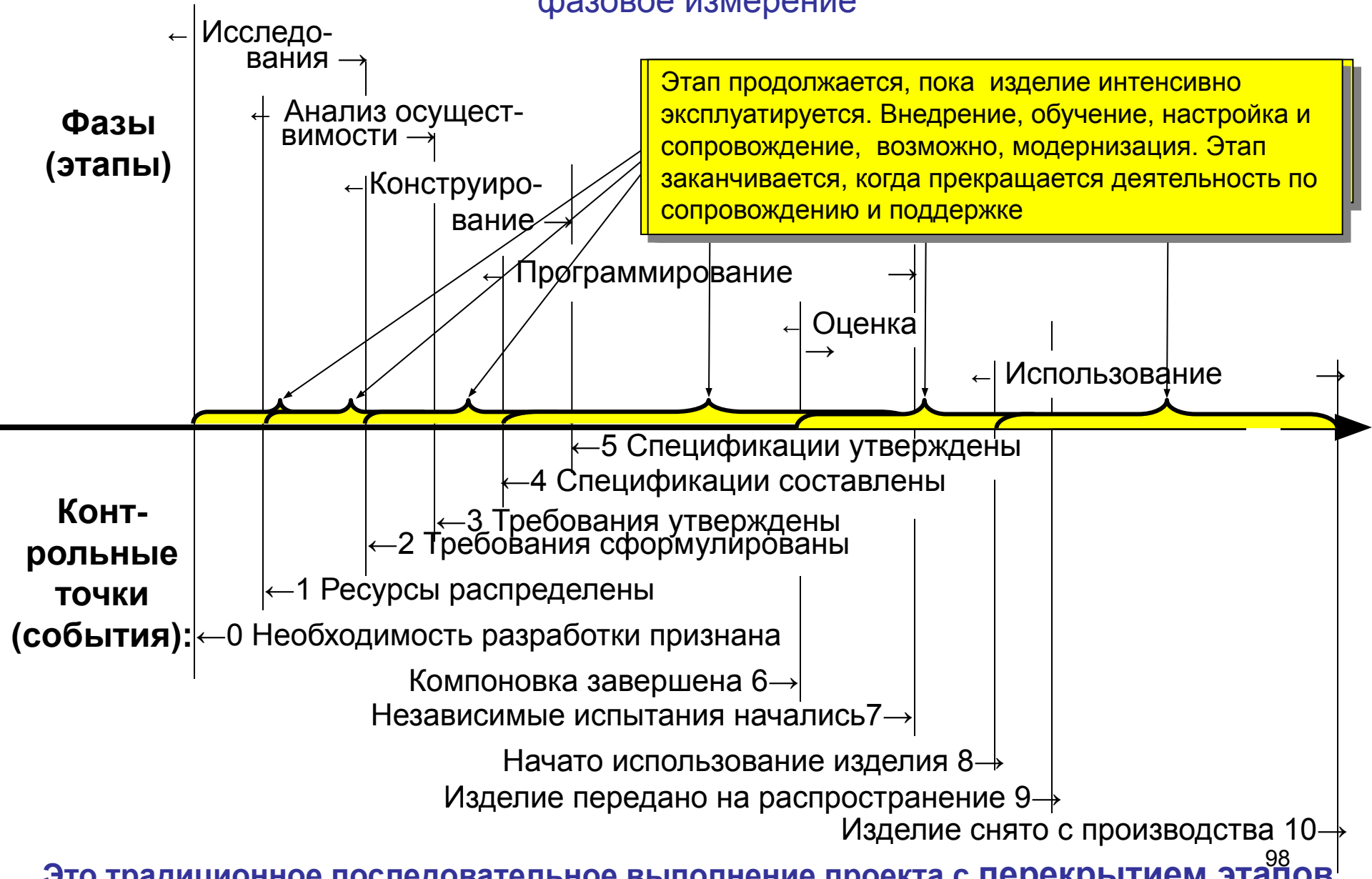
- Какие из рассмотренных моделей можно сделать инструментальными, а какие не допускают этого? Ответ обосновать.
- С какими видами документов, используемых в качестве барьеров вы сталкивались? Ответ поясните.

Лекция 12. Развитые модели жизненного цикла:

- Производственные функции в моделировании жизненного цикла: модель фазы — функции Гантера
- Моделирование жизненного цикла объектно-ориентированных программных проектов

Модель фазы—функции Гантера:

фазовое измерение



Это традиционное последовательное выполнение проекта с перекрытием этапов

Модель фазы—функции Гантера:
предпосылки функционального измерения (производственные функции — классы функций)

- Планирование
- Разработка
- Обслуживание
- Выпуск документации
- Испытания
- Поддержка
- Сопровождение

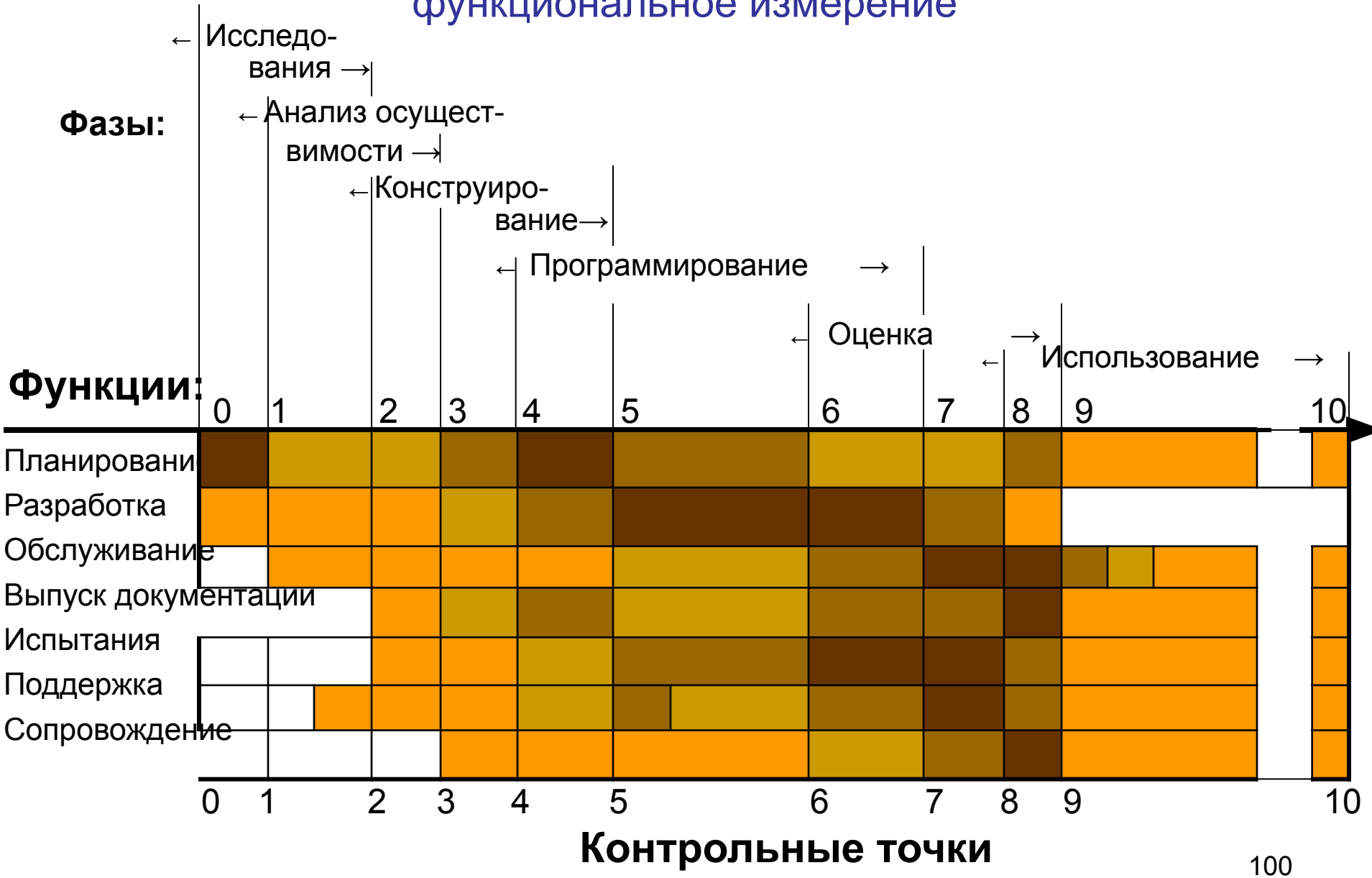
- Классы родственных функций можно считать выполняемыми в течение всего хода развития проекта;
- Содержание (цели) функции на различных этапах претерпевает изменение
- Интенсивность функции меняется как при переходе от этапа к этапу, так и в рамках работ одного этапа

Основной тезис: **На разных этапах функции**
имеют ***различное содержание***,
требуют ***различной интенсивности***,
при реализации проекта ***совмещаются***.

В конкретных проектах это понятие доопределяется (трактуется так, как полезно, например, как потребность или расходование ресурсов).

Модель фазы—функции Гантера:

функциональное измерение



Вариативность модели Гантера

- В зависимости от проекта функции можно трактовать свободно, дополнять другими классами функций, игнорировать некоторые из них и т.д.
- Можно рассматривать не только производственные функции, но и иное, полезное для управления (например, исполнителей проекта, трактуя интенсивность как занятость определенными заданиями)
- *Основной тези*
  — основа постро
  которое накладыва

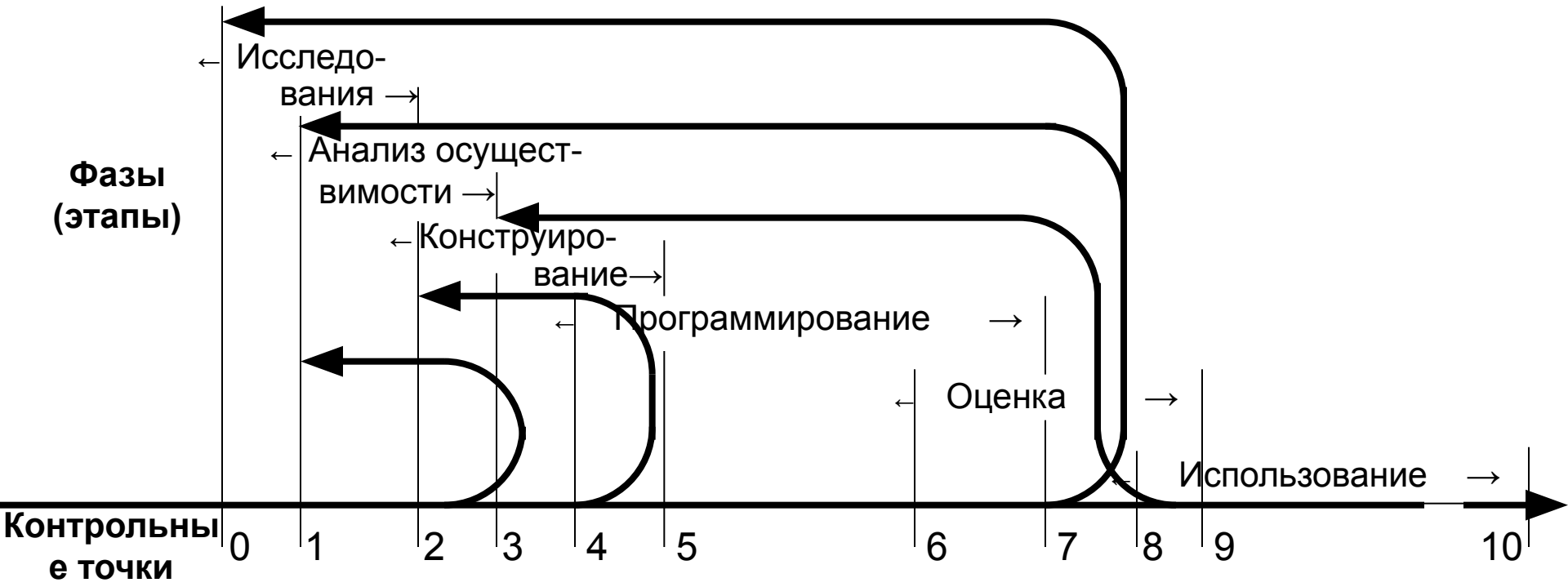
На разных этапах функции
имеют ***различное содержание***,
требуют ***различной интенсивности***,
при реализации проекта ***совмещаются***.

Матричная модель

- Элементы модели можно развивать, сохраняя требуемые связи моделируемой системы

⇒ Возможность превращения модели в инструментальную

Учет итеративности в модели фазы—функции



Расщепление линии развития проекта (жизненного цикла):

1. ***Приостановка процесса*** (в любой момент, если обеспечена корректность слияния) — традиционная реакция на ошибку
2. ***Действительное расщепление*** — появляются два (и более?) процесса. Для корректности нужно оценивать ресурсы, планировать новые контрольные точки и определять содержание существующих контрольных точек

- ***Слияние расщепленных процессов*** в случае 2 — должно планироваться!

⇒ **Действительное расщепление обязано быть регламентированным!**

Моделирование жизненного цикла объектно-ориентированных программных проектов

Принципы объектно-ориентированного проектирования

1. **Итеративность развития —** возможность перейти от последовательного развития к стратегии итеративного наращивания возможностей
2. **Изменение функциональности —** пересмотр требований при развитии проекта
3. **Формирование системы понятий проекта** — развивающийся *глоссарий проекта*
4. **Наращивание функциональности в соответствии со сценариями** — реализация выделенных сценариев; последующие итерации реализуют другие сценарии
5. **Ничто не делается однократно** — отказ от завершенности работ классических этапов, повторное прохождение их на новых итерациях (с новым набором сценариев)
6. **Оперирование на размножающихся фазах подобно** — обычные этапы при выполнении любой итерации развития проекта:
   - *Определение требований, или планирование итерации;*
   - *Анализ;*
   - *Моделирование пользовательского интерфейса /новое/;*
   - *Конструирование;*
   - *Реализация (программирование);*
   - *Тестирование;*
   - *Оценка результатов (итерации)*

Вне итераций:

1. **Начальная фаза проекта**: требования, ближайшая и перспективные задачи, критерии оценки результатов;
2. **Фаза завершения проекта**: поставка и сопровождение + выделение переиспользуемых компонентов

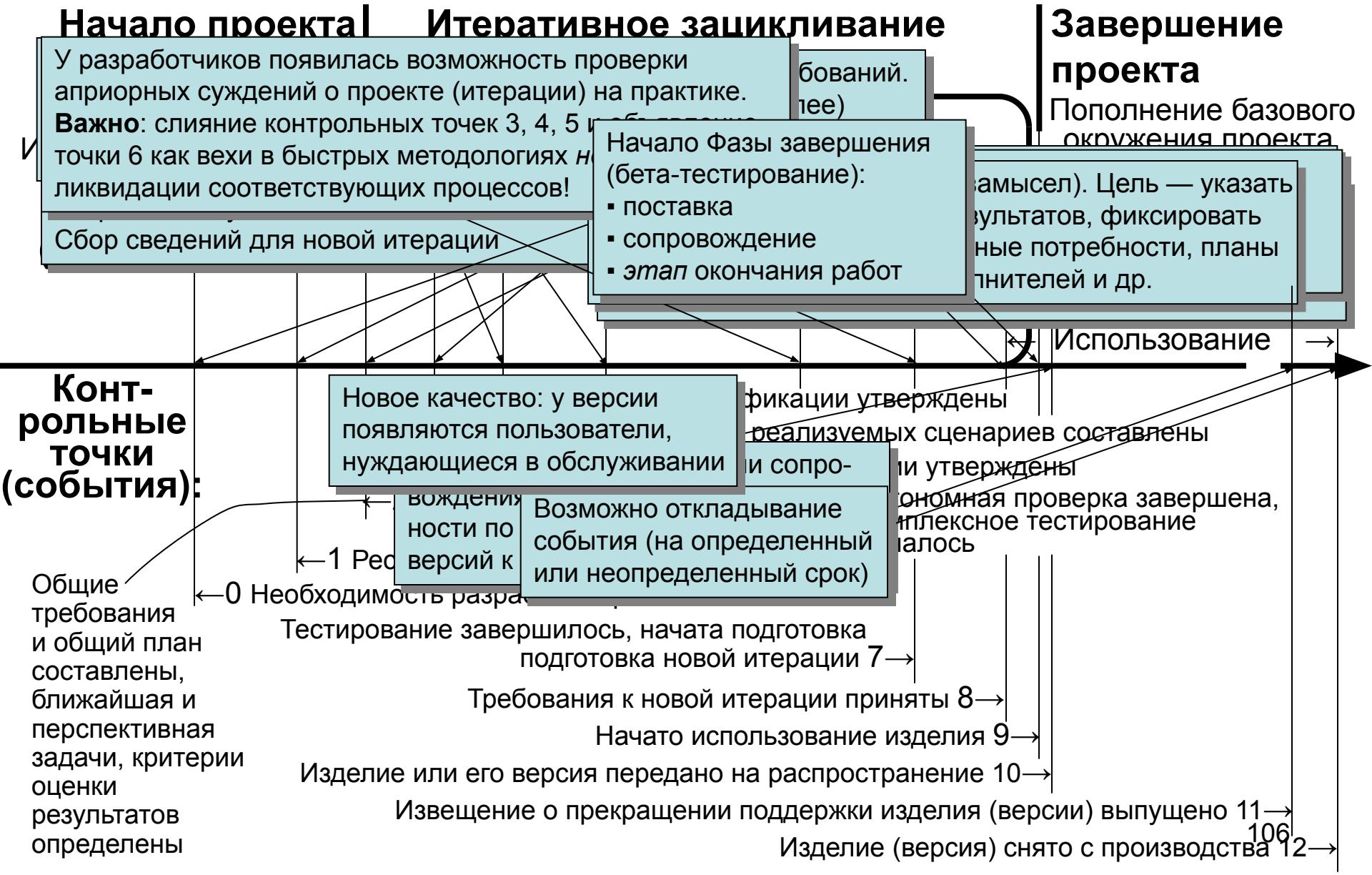
Моделирование при объектно-ориентированном проектировании

1. **Распределение реализуемых требований по итерациям**: Совокупность сценариев, реализуемых на очередной итерации + набор ранее реализованных сценариев образуют *законченную*, хотя и *неполную версию системы*, предлагаемую пользователям
   — *модели уровня анализа*
2. ***Особый стиль наращивания возможностей системы и ее развития***: Основа декомпозиции проекта при ООП подходе — набор связанных различными отношениями классов; новая итерация расширяет этот набор. Это расширение строится на базе построения
   — *моделей уровня конструирования*

Моделирование — *организационно-техническая* (*производственная*) *функция* всего процесса развития проекта, а не один из этапов!

Следствие:
Пополнение базового окружения проекта — дополнительный этап (вложенный в оценку), содержание которого — анализ возможного переиспользования накапливаемых компонентов ПО как для проекта, так и для будущего

Жизненный цикл при объектно-ориентированном развитии проекта (фазовое измерение)



Контрольные точки и вехи

- ***Контрольные точки*** (**check points**) — точки линии жизни жизненного цикла проекта, в которых возникают определенные события. Эти события рассматриваются как *существенные*, поскольку их необходимо отслеживать с целью управляемого развития проекта (такого, которое оставляет траекторию в рамках области допустимых операционных маршрутов)
- ***Вехи*** (**mail stones**) — это контрольные точки, прохождение которых сопровождается определенными планируемыми мероприятиями. Без успешного (результат соответствует цели) проведения таких мероприятий, прохождение *вехи* блокируется с целью выполнения активностей, направленных на исправление ситуации.
- ***Планирование*** получения ***результата*** и ***оценка полученного результата*** — основное содержание деятельности, связанной с вехами
- ***Конкретизация контрольных точек и вех*** — существенная задача, которую приходится решать в рамках выполнения функции планирования. Эта *конкретизация* делается на основе знания специфики выполняемого проекта и процесса его выполнения (т.е. приятой для проекта методологии). Специфика проекта и процесса определяет необходимость и количество вех.

Общие требования, общий план, ближайшая и перспективные задачи

Для каждой итерации должны быть определены:

- *Общие требования* — что требуется от проекта в целом в данный момент
- *Общий план* — как предполагается достигать цели (стратегия)
- *Ближайшая задача* — набор конкретных реализуемых требований и сценариев ← критерии предпочтения того, что планируется реализовывать
- *Перспективные задачи* — те, которые рассматриваются (в данный момент) как основа для планирования дальнейших итераций (в проектах жесткой отчетности)
- ***Критерии предпочтения***:
  1) Актуальность для пользователя
  2) Полнота и функциональная замкнутость предлагаемых средств
     - Функциональная полнота
     - Реализационная полнота
     - Интерфейсная полнота
  3) Системная значимость (внутрипроектные предпочтения)
  4) Демонстрационная значимость
  5) Скорость реализации

Характеристики значимости:

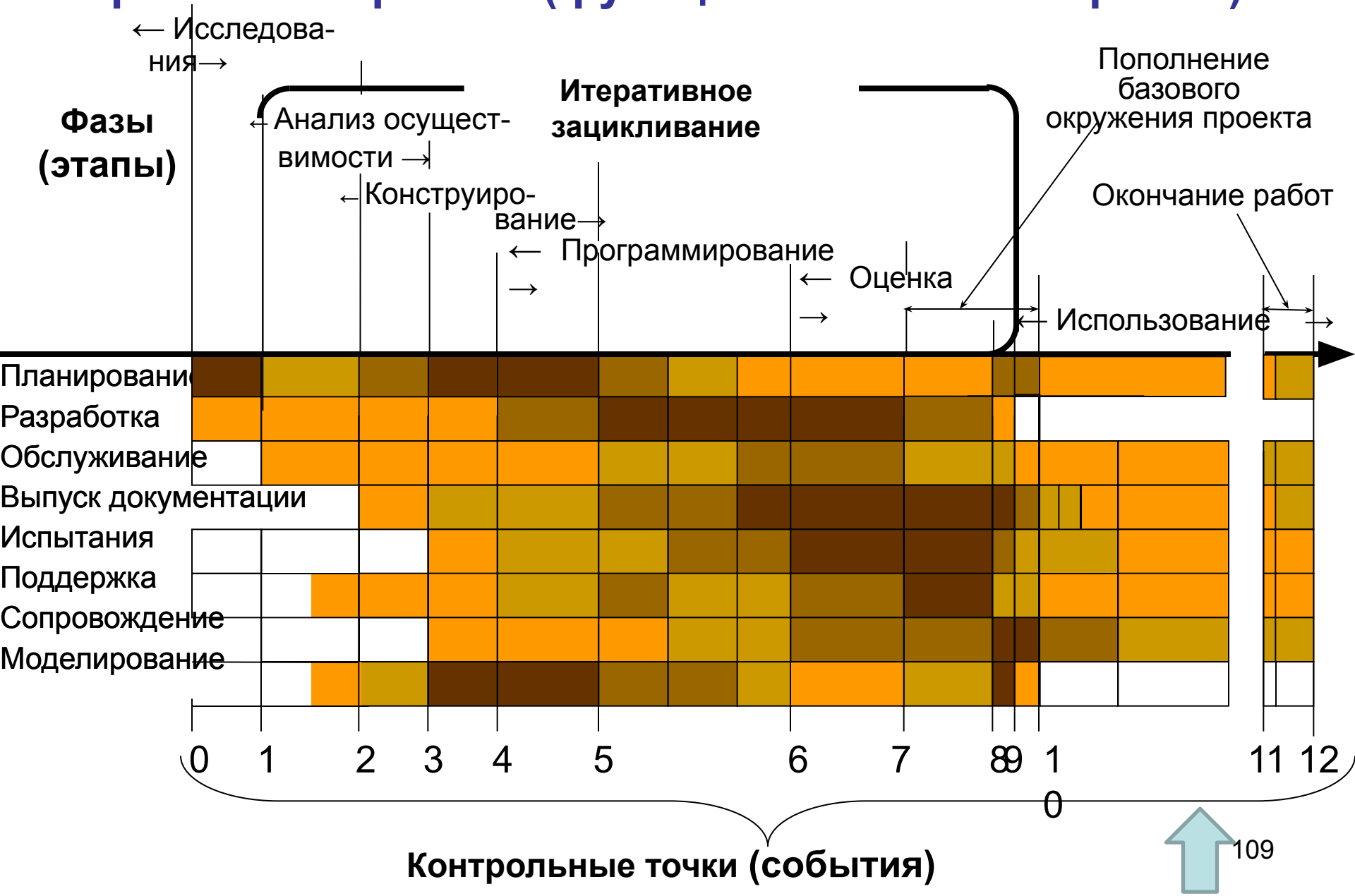
Всегда лучше то, что актуально!
Автоматизация деятельности в целом. Растет по мере увеличения объема уже выполненных работ
Реализационные предпочтения. Конкурирует с (1). Более значим для начальных итераций
Конкурирует с (1), (2) и (3) Максимально значимо для начальных итераций
Лучше то, что быстрее. Если время фиксировано, то для реализации определяется пул работ.

Возможные ограничения: время, объем работ, затраты (треугольник менеджмента)

Иногда время не критерий, а ограничение

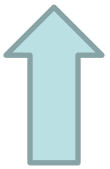
Минимизация реализуемого является критерием лишь для некоторых методологий!

Жизненный цикл при объектно-ориентированном развитии проекта (функциональное измерение)



Дополнительные лекции

Лекция А. Введение в базы данных: мотивация СУБД



Структурированные файлы и базы данных

- Файл как последовательность *записей* → справедливая мысль о связи понятий файла и записи

  модель файлов с буферной переменной:

  тип элементов файла = тип записей:

  "безликие" данные на внешних носителях обретают *структуру*

Если при обработке естественно выделять в две фазы, разделенные во времени:

- *формирование* структурированных данных (например, большого объема)
- *оперирование* со сформированной совокупностью данных

то использовать структурированные файлы удобно

Структурированные файлы и базы данных

- Файл как последовательность *записей* → справедливая мысль о связи понятий файла и записи

  модель файлов с буферной переменной:

  тип элементов файла = тип записей:

  "безликие" данные на внешних носителях обретают *структуру*

Если при обработке естественно выделять в две фазы, разделенные во времени:

- *формирование* структурированных данных (например, большого объема)
- *оперирование* со сформированной совокупностью данных

то использовать структурированные файлы удобно

Комплекс программ для поддержки работы приемной комиссии вуза

- Роли: абитуриент, секретарь, экзаменатор, посетитель и привилегированный посетитель, администратор, ?
- Функции:
  - Создание записи для посетителя, который становится абитуриентом;
  - Пополнение записи для абитуриента сведениями о сдаче экзамена;
  - Исключение записей абитуриентов, получивших неудовлетворительные оценки за экзамены (очевидно, что это не уничтожение информации — как добиться?);
  - Формирование списков абитуриентов, получивших проходной средний балл;
  - Формирование списков абитуриентов, зачисленных в вуз
- Дополнительные запросы. Например:
  - Качество подготовки выпускников в школах города: какие школы лучше готовят учеников к поступлению в вуз,
  - Эффективность подготовительных курсов и др.

Общие требования:

- Поступающая информация имеет определенную *структуру*;
- Она должна *накапливаться* для обработки (при первоначальном вводе, после сдачи очередного экзамена и др.);
- Информация *обновляется* (например, удаляются записи, относящиеся к сдавшим экзамен неудовлетворительно);
- Необходима *защита данных* от несанкционированного

Это (и другое) то, чем занимаются разработчики баз данных, когда приступают к проектированию

Проектирование БД и задача унификации

- Какое отношение проектирование имеет к файлам?
  - Система файлов – среда, в которой реализуется хранение информации баз данных.
  - Другая сторона проектирования БД: как пользовательское представление баз данных проецируется на уровень хранения информации.
- Решение этой задачи допускает стандартизацию → *системы управления базами данных* (СУБД). Их назначение – обеспечивать разработчиков информационных систем всем необходимым для единообразного и эффективного представления данных и средств доступа к ним.
- Однако далеко не всегда универсальное решение можно считать удовлетворительным
  - когда требуется точный учет специфики предметной области, приходится организовывать хранение данных и доступ к ним на основе **непосредственного представления базы данных структурами файловой системы** (пример АСУТП).

Автоматизация работы приемной комиссии: структура информации об абитуриенте

```
type TAbit = record
                    ...
             end;

var  FileAbitInf : file of TAbit;
```

- Файловое представление:
- не единственное, к тому же не самое удобное (что это?). Суть – файл как средство хранения данных на внешних носителях.
- не однозначное: можно по-разному организовывать файлы для хранения данных (в соответствии с разными *моделями баз банных*).
- С помощью типа `Tabit` нужно обеспечить *способ задания однозначного соответствия между сведениями о конкретном абитуриенте и записями файла* – это ближайшая задача

Требования к типу `Tabit`

Фамилия непригодна для *однозначного соответствия:*

- существуют однофамильцы;
- поиск записи, содержащей строковое поле (такой тип, по-видимому, будет у поля фамилии), более *медленный*, чем поиск по числовому полю;

→ для эффективного и однозначного соответствия между сведениями о абитуриенте и записями файла требуется числовое поле, которое мы назовем регистрационным номером (RegNum).

- Нужна *уникальность этих номеров* для каждого абитуриента

  если регистрацией занимается одновременно несколько членов комиссии, то необходима генерация новых номеров по запросам:

  - программа, блокирующая одновременное обращение к ней пользователей,
  - специальное соглашение о номерах.

  Последнее хуже, т.к. не исключает ошибок пользователя, но зато проще.

Вы

Задание этого типа эквивалентно описанию **array** [0..IName] of Char
(нулевой элемент массива – фактическое число символов)
Другие вар

Столько позиций резервируется для задания изображений имен. Это означает, при составлении программы надо

Пол задается как лит

льше или

Это ***производное поле***, оно принимает значение **True**, когда абитуриент становится студентом, т.е. когда
Exam. B >= проходной балл.

- Хранить его или вычислять? Для пользователя оно *всегда* хранится
- Взаимные производные поля (нельзя сказать, что хранить, а что нет)

Exam.B — тоже производное поле!

строение и

для

se

```
    Sex    : Char;
    Address: record
                 Ind : Word; { Почтовый индекс }
                 Twn, Str : String [IName]; { Город, улица }
                 H, F: Word; { Дом, квартира }
             end;
    PreCourses : Boolean;
    Exam  : record
                 Ex1, Ex2, Ex3, Ex4 : 1..5;
                 B : Real; { Средний балл }
            end;
    Student : Boolean;
end { описания типа TAbit};
```

Выбор структуры для типа `TAbit`

```
const lName = 10;
type  TAbit =
    record
      RegNum : Word;  { Положительное целое }
      FirstName, SecondName, LastName: String[lName];
      Sex    : Char;
      Address: record
                        Ind : Word; { Почтовый индекс }
                        Twn, Str : String [lName]; { Город, улица }
                        H, F: Word; { Дом, квартира }
                      end;
          PreCourses : Boolean;
          Exam  :  record
                        Ex1, Ex2, Ex3, Ex4 : 1..5;
                   B : Real; { Средний балл }
                end;
          Student: Boolean;
end  {описания типа TAbit};
```

Анализ выбранной структуры для типа `Tabit`

- Разумна ли выбранная структура? **Нет!**
  - Для поиска абитуриентов из одного места проживания просматриваются все записи и для каждой – сравнение Address с заданным значением.
  - Лучше перечень населенных пунктов (возможно, с индексами), выделить в специальный файл, а в записях FileAbitInf оставлять только номер
    - Кон
    - Сок

  Снова БД

Это даст сокращение размеров базы данных (в частности, за счет однофамильцев, тезок, земляков и т.п.).
Нужно ли это реализовывать, без обстоятельного анализа будущего применения системы сказать трудно

- Модификация:

```
Address :   record
              Location : Word; { Индекс записи в новом файле FileLocations }
            Str  : String [IName]; { Улица }
              H, F : Word; { Дом, квартира}
            end;
var FileLocations :  file of record
                       Ind  : Word;{ Почтовый индекс }
                       Twn      : String [IName]; { Город }
                     end;
```

- Радикальное решение для типа String [IName]:

  в FileAbitInf и в FileLocations использовать поля с индексами соответствующих строк, а сами строки хранить **в еще одном специальном файле**

Обсуждение модернизации типа TAbit

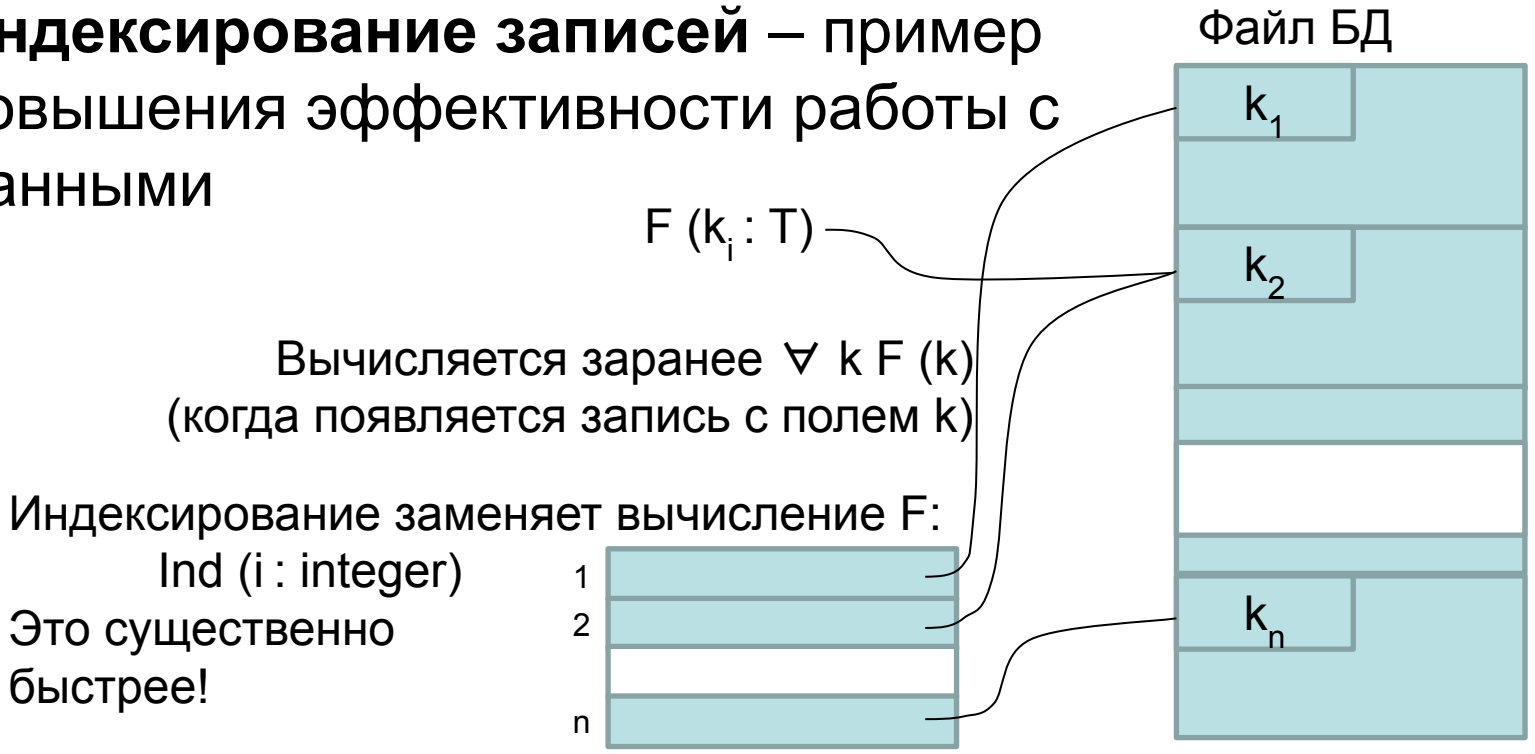
- Целесообразность разбиения информационного массива БД на несколько файлов:
  - для повышения эффективности
  - для обеспечения выборочной защиты данных
  - для повышения эффективности поиска информации:
    - Запросы к БД с поиском записи с заданным значением поля FirstName – **вычисление функции с аргументом строкового типа**, вырабатывающей номер записи с полем FirstName, равным аргументу функции, или выделенное значение (0 – нет такой записи).
    - Такая функция реализует **ключевой поиск**
      - Поле (набор полей), по которому ведется ключевой поиск, называется **ключевым**,
      - Возможные значения аргумента функции — **ключи**
- Ускорение ключевого поиска – **индексирование** записей:
  - Построение специального **индексного файла** с заранее вычисленной функцией ключевого поиска

    F (Key – ключ) = N – указатель на запись в файле или 0

    $\forall$ k $\in$ K (Ind (k) = n | $\neg$ (n = 0) $\supset$ F (n) – указатель на запись в основном файле БД)

Почему нужны СУБД

Индексирование записей – пример повышения эффективности работы с данными



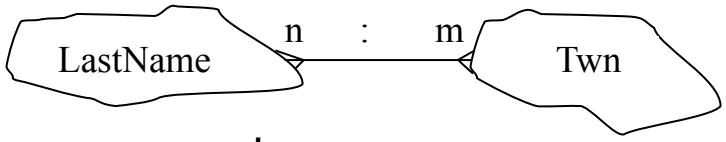
Требуется, чтобы значения всех ключевых полей различались

- Специальная организации индексных файлов
- В промышленных СУБД – фирменный секрет

Непосредственное конструирование информационных систем, т.е. без использования СУБД, довольно трудоемко

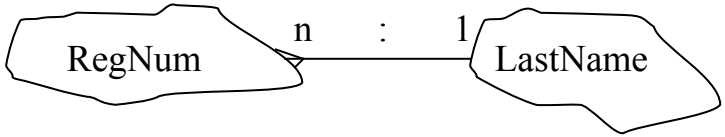
Отношения между данными

1. Отношение «многие ко многим»:



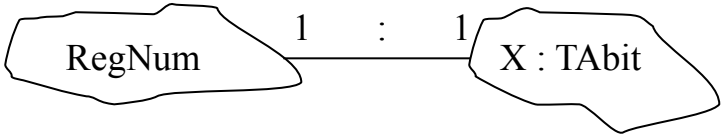
Могут существовать абитуриенты с одинаковыми фамилиями, приехавшие из одного или из разных городов, и из одного и того же города могут приехать абитуриент, имеющие одинаковые фамилии

2. Отношение «один ко многим»:



RegNum определяет LastName однозначно, но для одного и того же LastName возможны разные RegNum (однофамильцы).

3. Отношение «один к одному»:



RegNum определен так, чтобы он *взаимно-однозначно* идентифицировал каждую запись целиком

- Отношение 1:n – намек, что что-то может быть вынесено в отдельный файл.
- Отношение n:m – указание, что для запроса, связывающих эти сущности придется определять дополнительную структуру данных
- Отношение 1:1 – возможность склейки таблиц

Лекция В. Модели баз данных

Модели баз данных.Что это?

Данные: хранятся, появляются, уничтожаются, предоставляются – *пассивные*

Сведения: формируются, сообщаются, передаются – *предъявляемые*

Информация: извлекается (генерируется) из сведений и данных, используется в некоторой деятельности (деятельностях) – *активная*

Данные имеют структуру. В зависимости от точки зрения на них, от использования они структурируются по-разному: одновременно имеют разные структуры.

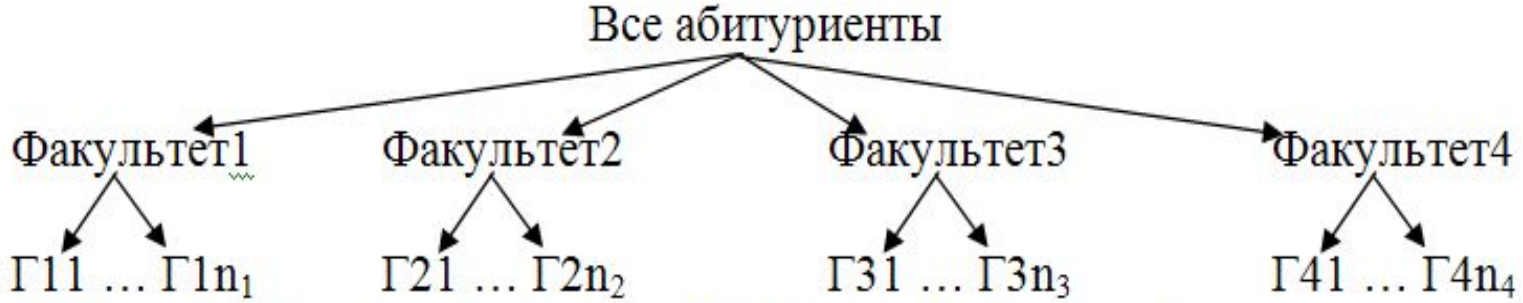
- Физическая структура данных
- Логическая структура данных

2. Хранимые данные: файлы, записи в машинном представлении информации – *модели уровня хранения*
3. Данные, которые размещаются в БД и извлекаются для внешнего использования – *модели уровня конструирования информационных систем и обработки запросов*
4. Представление, воспринимаемое пользователем – *модели пользовательского уровня*

- **Модель базы данных** – это структуры данных, которые поддерживаются СУБД на логическом уровне (основа конструирования конкретных баз данных и информационных систем)

Иерархическая модель

- Понятия иерархий и отношений, задающих иерархии
- Иерархии для накопления данных, а также поиска и извлечения их (для дальнейшей обработки)
- Два варианта иерархической модели:
  - для поиска листьев, представляющих данные, атрибуты которых размещаются в нелистовых узлах, — они общие для поддерева (отношение «содержит»)
  - данные размещаются в узлах (есть содержательное отношение, задающее иерархию, по которому строится дерево поиска)
- Примеры, когда использовать поиск по дереву эффективно
- Пример с абитуриентами:



Найти всех из одного города — неэффективный запрос!

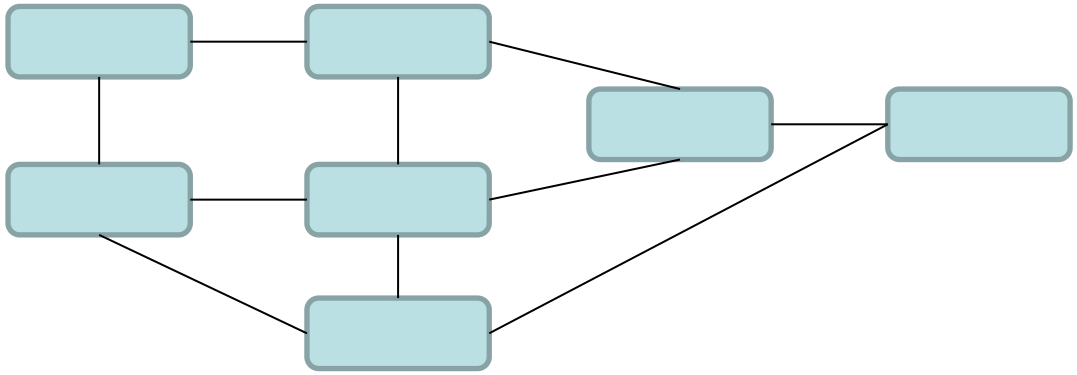
- *Прошитые деревья* (ссылки между узлами – для разных целей, несколько деревьев, связанных ссылками)

— ответ на недостаточность этой структуры и шаг к следующей модели

Сетевая модель

Используется граф: вершины данные, дуги используются для навигации (узнали гипертекстовый html?)

1. Есть естественная (сетевая) структура данных. Например, разные отношения.
2. Такая структура строится, исходя из запросов, которые предполагаются для данной прикладной области. Для новых типов запросов надо сеть достраивать. При добавлении данных сеть увеличивается.
3. Семантическая сеть.



Реляционная модель: неформальное определение

- Дейт:
  - вся информация в базе данных представлена в таблицах;
  - поддерживается три реляционные операции:
    - выборка,
    - проецирование,
    - объединение.

  } с помощью которых обеспечивается весь доступ к данным (физическую запись знать не надо) (*)

- Правила Кодда (12 критериев) — их, излагаем неформально:

Чтобы считаться реляционной, СУБД должна:

1. Предоставлять всю информацию в виде *таблиц* (как у Дейта);
2. Поддерживать *логическую структуру* данных, независимую от их физического представления;
3. Языки структурирования, запросов и модификации данных должны быть *высокого уровня* (например, SQL). Здесь про ЯОД, ЯМД и др. языки, в частности, ЯОХД;
4. Поддерживать *реляционные операции* (*), а также *теоретико-множественные операции* ($\cup$, $\cap$, $\setminus$ — как минимум);
5. Поддерживать *виртуальные таблицы* (термины: view, курсор);
6. Различать *неизвестные, невозможные значения* и *пропуски в данных* (что это?);
7. Обеспечивать механизмы:

1) поддержки целостности, 2) авторизации, 3) транзакций, 4) восстановления данных.

Список не взаимно независимый!

Таблицы и базы данных (1)

таблица	строка	столбец	Пользовательские термины
table	row	column	
отношение	кортеж	атрибут	«Академические» термины
relation	tuple	attribute	
файл	запись	поле	Термины из обработки данных
file	record	field	

База данных — набор связанных таблиц:

Строка описывает *объект*, или *сущность* (entity)

Столбец описывает *свойство*, или *атрибут* объекта

Первичный ключ – свойство, которое определяет, идентифицирует запись (объект, сущность)

имя таблицы + первичный ключ + столбец => значение

Пользовательские таблицы — данные и

Системные таблицы — описание базы данных (системные каталоги, мета данные и др.)

К правилам Кодда:

8. Обеспечивать возможность доступа к *любым* таблицам, в т.ч. к системным, причем точно такую же возможность, что и к пользовательским таблицам.

Независимость данных (2)

Независимость:

- на логическом уровне
- на физическом уровне

Изменение взаимосвязей между таблицами не влияет на функционирование (можно разбивать таблицы по строкам, столбцам, сливать их — старые запросы выполняются, как раньше)

Независимость логической структуры от способа хранения (в частности, перемещения, индексирования и т.д. ничего не меняют)

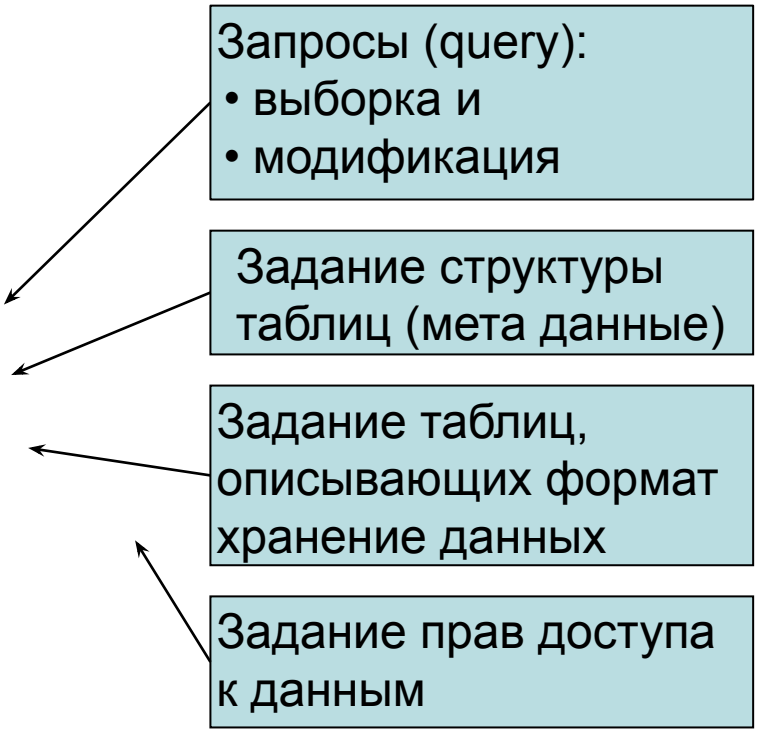
Почти!

Это источник различий одного и того же стандарта реляционных СУБД

Языки высокого уровня (3)

- Манипулирование данными (ЯМД);
- Определение данных (ЯОД);
- Определение хранимых данных (ЯОХД)
- Администрирование (управление)

Все это есть в SQL



Примеры

Манипулирование данными:

- *выборка*

```
select * from publishers
```

```
 pub_id   pub_name  address    city   state
-------------------------------------------------------
   0736   New Age Books 1 1st St. Boston MA
```

Снова тот же `select`:

```
 pub_id   pub_name  address    city   state
-------------------------------------------------------
   0736   New Age Books 1 1st St. Boston MA
   0897   Binnet&Hardley   12 2nd Ave.   Washington    DC
   1345   Algodata Info  10 3rd Dr. Btrkley CA
   0010   Pragma  45 10th ln. Chicago ÍL
```

publishers:

pub_id	pub_name	address	city	state
0736	New Age Books	1 1st St.	Boston	MA
0897	Binnet&Hardley	12 2nd Ave.	Washington	DC
1345	Algodata Info	10 3rd Dr.	Btrkley	CA
0010	Pragma	45 10th ln.	Chicago	ÍL

- *Проецирование*: задание того, какие столбцы хочется просматривать:

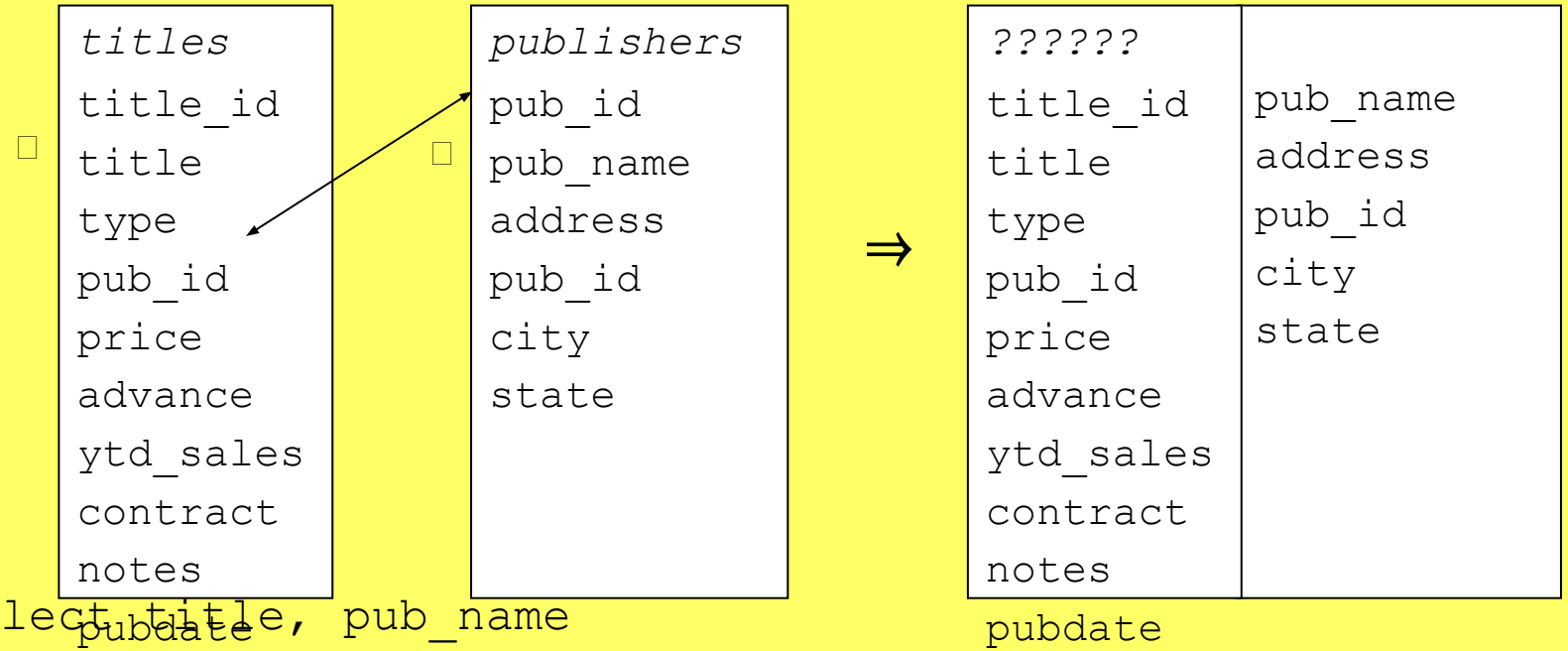
```
select pub_id, pub_name from publishers
```

Результат – таблица:

pub_id	pub_name
0736	New Age Books
0897	Binnet&Hardley
1345	Algodata Info
0010	Pragma

```
where publishers.pub_id = titles.pub_id
```

Результат — новая таблица, состоящая из строк, в которых произошло совпадение




```
select title, pub_name
from titles, publishers
where publishers.pub_id = titles.pub_id
```

Результат:

```
title  pub_name
-------------------------------------------------------------
TTTT lll rrr      New Age Books
Jjj lll rrr       New Age Books
EEE yyy        New Age Books
BBBBB1     Binnet&Hardley
    BBBBB2     Binnet&Hardley
    BBBBB3     Binnet&Hardley
AAA book1      Algodata Info
AAA book2      Algodata Info
```

Реляционные и теоретико-множественные операции (4 - продолжение)

А почему бы не использовать объединенную таблицу?

- Ответ:
  a) дублирование данных (примере – из таблиц *titles*, *publishers* и *??????*). Ключи дублировать можно и нужно!!!
  b) трудно составить согласованные со смыслом таблицы (какой сущности соответствует *??????* ?)
  c) возможны противоречия (из a)

Вывод: Нужно проектировать базу данных, т.е. ее таблицы !!!

Виртуальные таблицы(5)

Альтернативный способ просмотра данных: образование *виртуальной таблицы*, или *курсора* (view), или *производной таблицы*

```
create view Books_and_Pubs
as
    select title, pub_name
    from titles, publishers
    where publishers.pub_id = titles.pub_id
```

Трудности достичь эффективной реализации *оперирования с виртуальными таблицами точно также*, как с обычными таблицами (хотя это требование следует из правил Кодда) → нарушение стандарта

Неизвестные, невозможные значения и пропуски в данных (6)

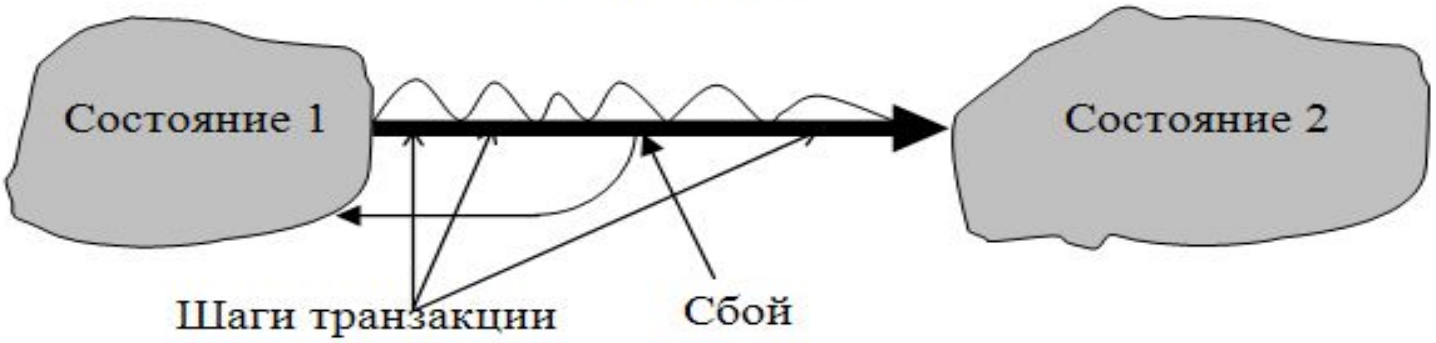
Это еще одна проблема для стандартизации (см. дополнение 8)

Безопасность: авторизация **(7.2)**

- Права доступа и роли
  → авторизация — механизм «знания» системой имен ее пользователей
- Понятие владельца данных
- В SQL выделены средства определения
  - владельца (тот, кто создает таблицу, но можно делегировать права);
  - права на чтение
  - права на модификацию (чтение и запись) таблицы или отдельного столбца.
- Запрет на доступ к отдельным строкам моделируется.

Безопасность: целостность и ограничения на целостность **(7.1, 7.3, 7.4)**

1. Причины рассогласования (противоречивости, некорректности) данных:
   - сбой в системе (программный, аппаратный);
   - логические ошибки (некорректное проектирование).
2. Управление транзакциями:
   - Транзакция выполняется с начала до конца:



3. *Объектная целостность*: ни один первичный ключ не может иметь нулевое значение (почему?)
4. *Ссылочная целостность*: непротиворечивость повторяющихся частей (почему?)

Безопасность: целостность и ограничения на целостность **(продолжение)**

Вариант	Плюсы	Минусы
Проверка целостности по вторичными ключам возлагается на СУБД	• универсальность решения: система заставит что-то делать, если целостность может нарушиться; • автоматическая согласованность с транзакциями.	исчезает содержательная трактовка опасной ситуации ⇒ все равно надо что-то делать
Проверка возлагается на приложение	разнообразие возможных реакций исходя из ситуаций, известных приложению (примеры: фиктивный издатель, отложенная реакция и др.)	можно вовсе забыть про реакцию или сделать не ту.
Вынужденная проверка в приложении	СУБД не поддерживает контроль данного вида ограничений, потому *приходится* отслеживать самому	

Лекция С. Проектирование баз данных

Общие положения

Выбор:

- таблиц,
- столбцов таблиц,
- взаимосвязей между таблицами и столбцами таблиц

Логическая структура не должна выбираться, исходя из хранения и предъявления данных. Хотя реляционная СУБД это обеспечивает, при проектировании БД есть возможность «забегать вперед»

Тем не менее, вопросы эффективности решаются

Дейт:

> «Создать нужную структуру базы данных зачастую проще, чем строго сформулировать, какой она должна быть»
>
> > в простых случаях — да, то в сложных без специальной работы с требованиями не обойтись

Последовательность шагов

1. Исследовать *информационную среду*, которую нужно моделировать:
   - откуда поступают данные?
   - как они вводятся, и кто это делает?
   - какие параметры будут наиболее критичными с точки зрения *времени реакции* и *надежности*?
   - какие виды извлечения информации нужны?
2. Создать список объектов вместе с их:
   - свойствами (здесь – гипотезы, которые потом корректируются: как часто объект будет использоваться, с какими другими объектами он связан, др.)
   - атрибутами (типы атрибутов, какие атрибуты за что отвечают и др.)
3. Можно начинать как с объектов, так и с атрибутов (не смешивать подходы!), но в обоих случаях нужно ответить на вопросы:
   - действительно ли выбранные атрибуты подходят для описания данного объекта?
   - нужны ли еще атрибуты или объекты?

Объекты – таблицы
(1 объект –строка)
Атрибуты — столбцы

Все принимаемые решения — записывать!

В частности, на этом этапе составляется *макет таблиц* и связей: ER-диаграммы.

Последовательность шагов

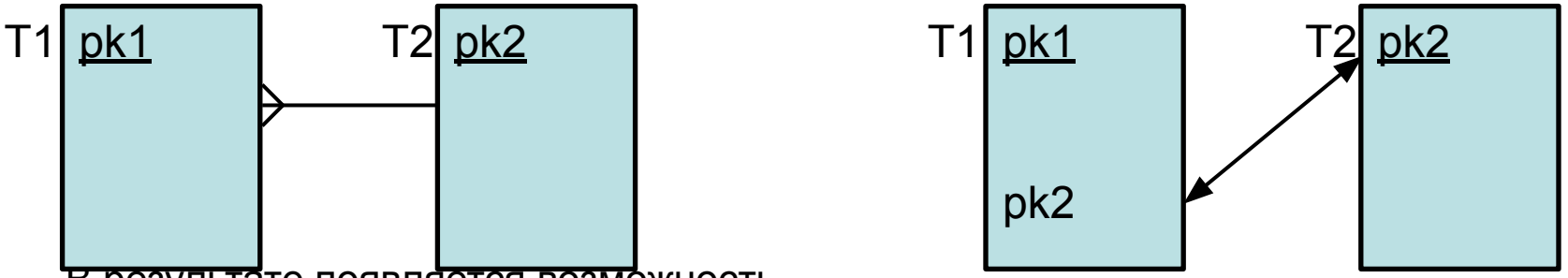
4. Убедиться, что есть атрибут (или группа атрибутов), однозначно идентифицирующая любую строку каждой таблицы, т.е., что есть *первичные ключи*.

   Если нет — добавить искусственно.

5. Рассмотреть зависимости один ко многим:

   Есть ли возможности для объединения связанных таблиц?

   Для этого добавить внешние ключи:



   В результате появляется возможность

   «*отобрать все из T1, у кого внешний ключ равен первичному из T2*»

6. Проверить нормализацию, если нужно, то

   исправить или *обосновать отклонение* от нее.

   Проследить, как нормализация выполняется на логическом уровне.

7. Ответить на вопрос: удовлетворяет ли вас результат?

   Нет — переделать, уточнить, дополнить и т.д.

Хорошая и плохая структура базы данных

- Что такое «хорошая структура» базы данных?
  - максимально простое взаимодействие;
  - гарантии непротиворечивости;
  - максимальная производительность.
- Что такое «плохая структура» базы данных?
  - непонимание результатов запросов;
  - риск противоречивости данных;
  - избыточность;
  - сложность изменение структуры уже созданных (и заполненных) таблиц.

Нужен критический анализ построенного (всегда)

Проект БД «Книги, авторы, издатели» (1)

1. Вопросы, которые могут задавать пользователи, — самые разные:
   - Кто из авторов проживает в Калифорнии?
   - Какие книги стоят больше XX $?
   - Кто написал самое большое число книг?
   - Чем мы обязаны автору XXX? (!!!)
   - Какова средняя стоимость книг по психологии?
   - Как продаются книги по программированию?
   - …

Нужно только суметь разграничить, что будет, и что не будет доступно из конструируемой базы данных

2. Что можно извлечь об информационной среде, изучая ее при опросе будущих пользователей:

Важно!

   - автор может написать несколько книг;
   - книга может быть написана несколькими авторами;
   - порядок фамилий авторов на первой странице книги является важным, т.к. влияет на получаемый ими гонорар;
   - редактор может работать над несколькими книгами, и у книги может быть несколько редакторов;
   - в заказе на покупку может быть несколько книг;
   - …

Проект БД «Книги, авторы, издатели» (2)

3. Объекты: Свойства: Взаимосвязи:

авторы имя — у книги есть один или
адрес — и т.д.
телефон
…

книги название
авторы
стоимость
…

издательства название
адрес
…

редакторы имя
адрес
телефон
книги

Пока здесь не учитывается весь круг вопросов, связанных с продажами и заказами на продажу. В последствии соответствующее дополнение будет сделано, а то,
на сколько это будет легко, скажет, хороша ли была выбрана первоначальная структура базы данных

4. Макет БД и поиск первичных ключей (*специальный столбец*) про фамилии и
кность исп я номера с полиса

Authors
au_id
au_lname
au_fname
address
phone

Titles
title_id
name
price
type
advance

Publishers
pub_id
name
address

Editors
ed_id
ed_lname
ed_fname
address
phone

Проект БД «Книги, авторы, издатели» (3)

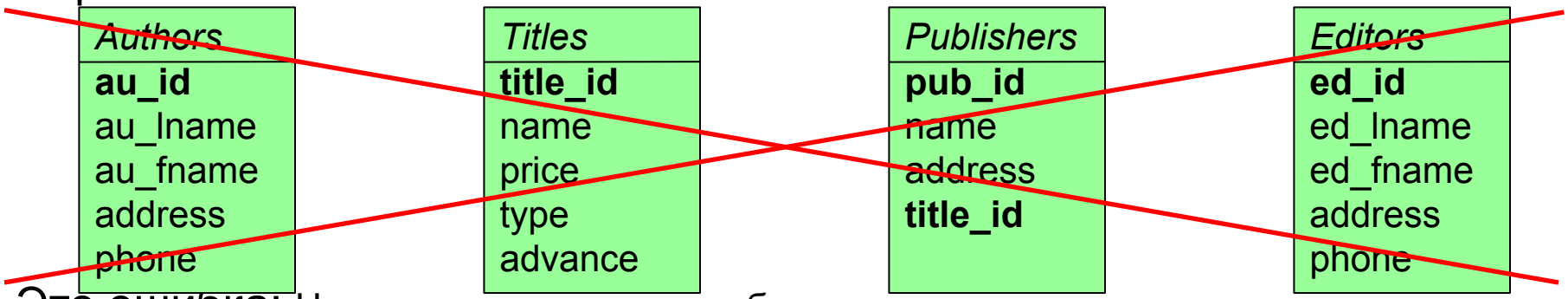
5. Отношения один ко многим

Задача: связать *Titles* и *Publishers*.

В результате анализа информационной среды выяснено требование, реализовать запросы, в которых название книги фигурирует совместно с издателем.

Требуется обеспечить возможность объединения этих таблиц

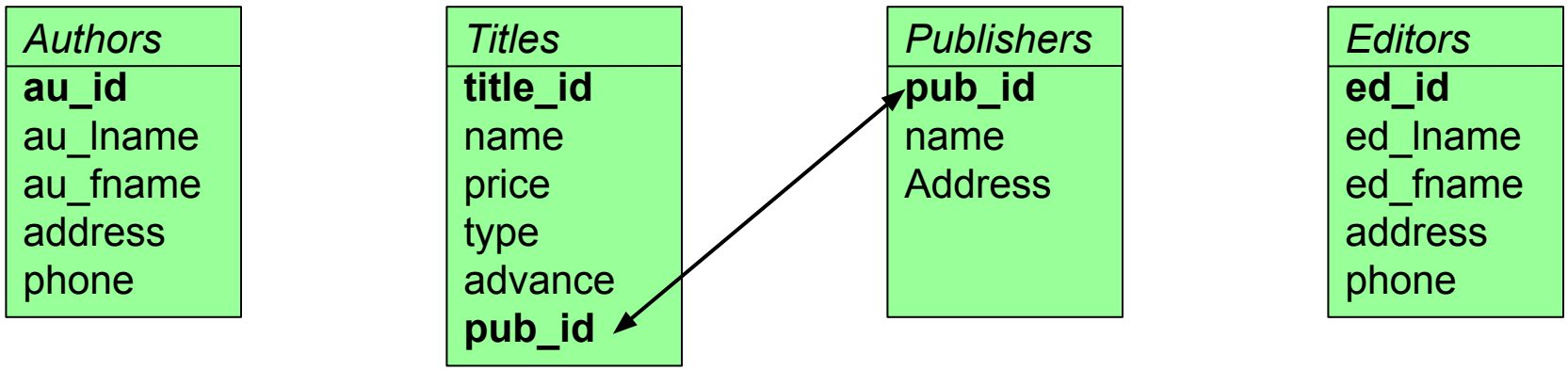
Первая попытка:



Это ошибка: На все названия книг столбцов не напасешься.

Трудно модифицировать данные (нереально определять для новой книги столбец)

Решение обратное: Снабдить Titles *внешним ключом*



Проект БД «Книги, авторы, издатели» (4)

Анализ решения (целостность):

- обе таблицы содержат по одной строке на объект;
- `pub_id` повторяется в `titles`, т.к. издательство издает много книг, но это не дублирование данных, а дублирование ключей!
- установленную связь можно использовать для объединения таблиц:

Пример

```
select title, pub_name \\ Что угодно
from titles, publishers
where publishers.pub_id = titles.pub_id
```

- НО! Са… …нных
  - уда…
- Нужно…
  - есл… …ть все заг…
  - если добавляется книга, то надо найти или добавить издателя.
- Кто подобные ограничения вводит и/или отслеживает?

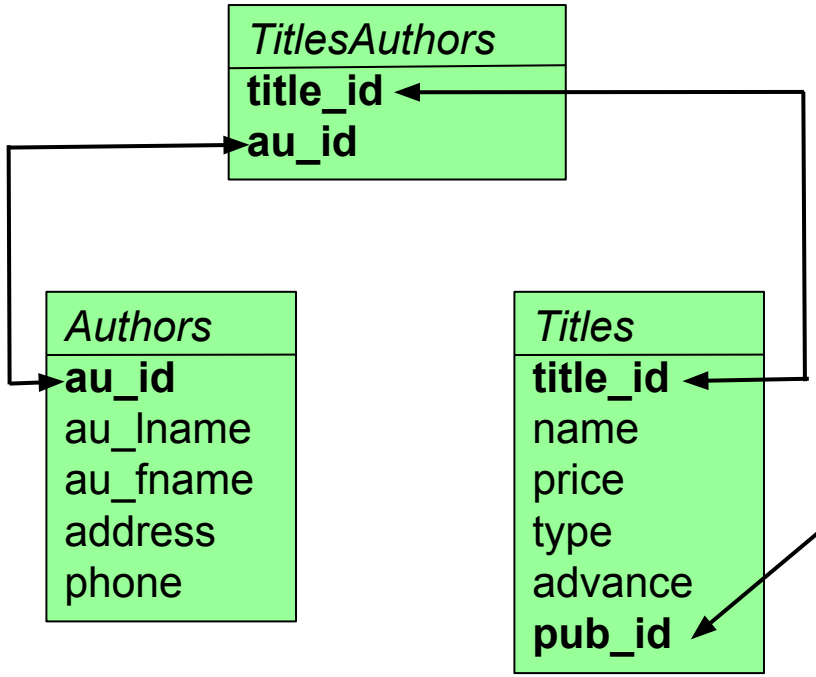
  Вопрос имеет далеко не однозначный ответ.
  - Из правил Кодда он не следует:
    - Ограничения хранятся в словарях, а не в приложениях, но это о *хранении*, а не об *отслеживании*.
    - Если СУБД *может* поддерживать отслеживание, то это не значит, что им пользуются конструктор базы данных и запросов.
  - Сопоставление возможностей СУБД и приложений в части поддержки ограничений целостности см. в предыдущей лекции

Проект БД «Книги, авторы, издатели» (5)

6. Отношения многие к многим

Автор может писать много книг, а книгу могут писать многие авторы
Учет этого отношения → нужна **реализация объединения таблиц**

Таблица связей, или *ассоциация*:



- Это пример *составного первичного ключа*.
- Можно рассматривать таблицу-ассоциацию как объект, связанный с книгами (авторами) отношением *один ко многим*:
  - одна ассоциация — одна книга (один автор),
  - книга (автор) может входить в несколько ассоциаций.

  → Нужна соответствующая пара ограничений на целостность (как выше).
- Связь и *придуманный* объект могут иметь содержательный смысл и, в частности, свои атрибуты (пример – накладная)

Ассоциация – это два отношения один к многим

Отношения один к многим и многие к многим и понятия *главной* и *детализирующей* таблиц

Проект БД «Книги, авторы, издатели» (6)

- *Анализ может*
  указать явно на то, что требуется реализация запросов, которые требуют объединение таблиц
- *Но он не может*
  показать, что такого сорта запросы не появятся при развитии конструируемой информационной системы в будущем
- Потребность расширения запросов
  → преобразование структуры существующих таблиц
  → потеря эффективности
  → нужно постараться ликвидировать причины возможных потерь производительности. (примеры: незамеченные отношения многие ко многим и один ко многим)
- Важно угадать и постараться ликвидировать причины, из-за которых возможны потери производительности (эффективности системы и работников при реализации новых возможностей)

Проект БД «Книги, авторы, издатели» (7)

7. **Отношения один к одному**

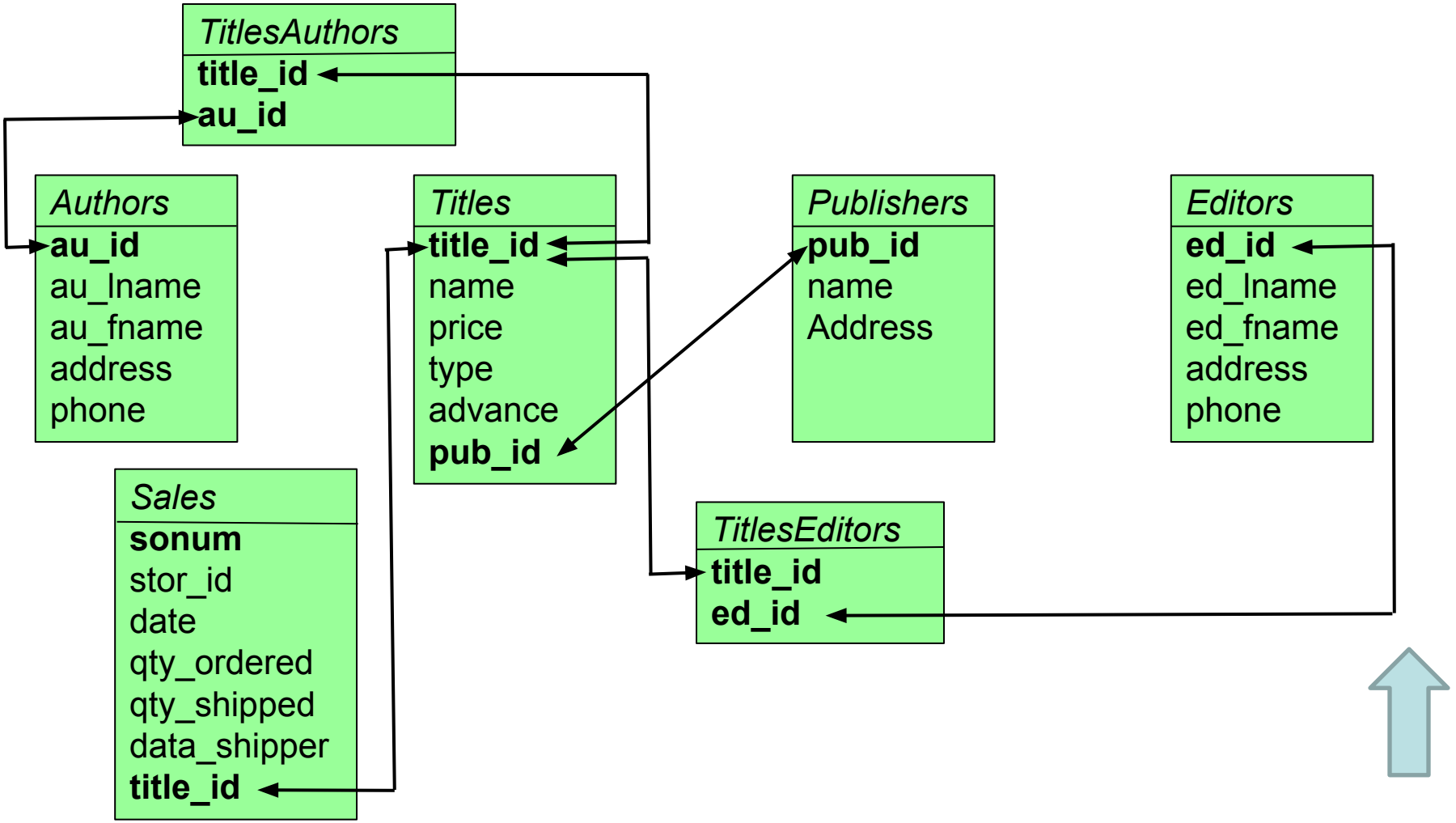
- свободны от необходимости угадывать будущие незапланированные запросы:
  Обнаружив такое отношение, можно
  - слить две таблицы в одну или
  - ничего не менять
- *Рекомендация*: выделять в отдельную таблицу то, что реже используется
  → это сократит время реакции для частых запросов

Проект БД «Книги, авторы, издатели» (8)

8. **Первый итог:**
   a) независимые объекты — строки таблиц;
   b) свойства — столбцы;
   c) у всех таблиц есть первичные ключи;
   d) надо проверить, есть ли еще отношения 1:N, и, если да то:
      – добавить внешние ключи к «многим»,
      – определить ограничения на целостность, связанные с отношениями.
   e) надо проверить, есть ли еще отношения N:M, и, если да то:
      – построить таблицы связи,
      – определить ограничения на целостность.
   f) что еще не учли? Если надо учесть, то доделать.

Проект БД «Книги, авторы, издатели» (9)

- Диаграммы «Сущность – связь» (ER-диаграммы)
  Их нужно уметь
  - Составлять
  - Читать



Лекция С. Нормализация

Понятие нормализации и первая нормальная форма

- ***Нормализация*** — набор стандартов проектирования БД, называемых *нормальными формами*, которые гарантирует качество с точки зрения выполнения различных критериев
- Существует пять (иногда говорят о семи!) нормальных форм $НФ_i \subseteq НФ_{i+1}$, (именно столько критериев качества)
  - уменьшение избыточности данных (за счет дробления таблиц и дублирования ключей) →
  - формальная непротиворечивость (для содержательной гарантий быть не может)

1. ***Первая нормальная форма 1НФ*** требует, чтобы на пересечении любых столбца и строки находилось *единственное атомарное значение*.

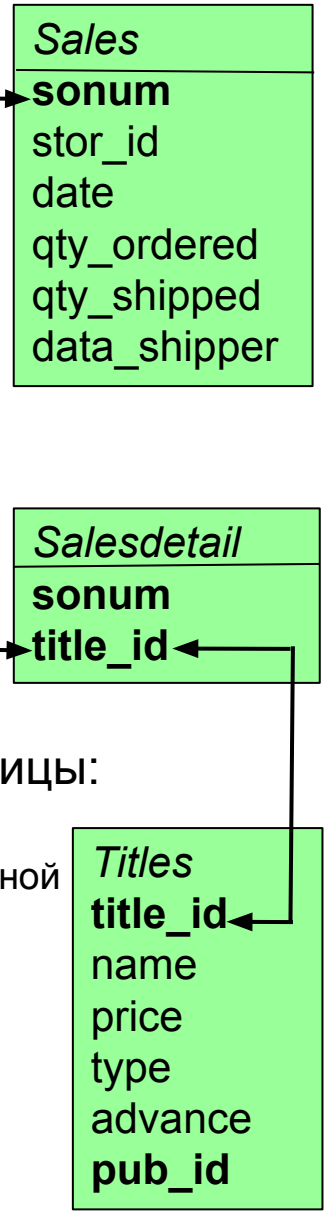
Содержательно: атрибут неделим (строковое поле для СУБД неделимо тоже)

- Что это дает?
  - Разграничение возможностей СУБД и приложений +
  - Корректное оперирование с атрибутами, которые изначально, казалось бы, требованиям 1НФ не удовлетворяют +
  - Технологичное решение: *главная* (*master*) и *детализирующая* (*detail*) таблицы.

Первая нормальная форма: пример

Таблица *Sales* не удовлетворяет пожеланию заказчика иметь в одном заказе несколько книг. Есть четыре варианта, как это преодолеть:

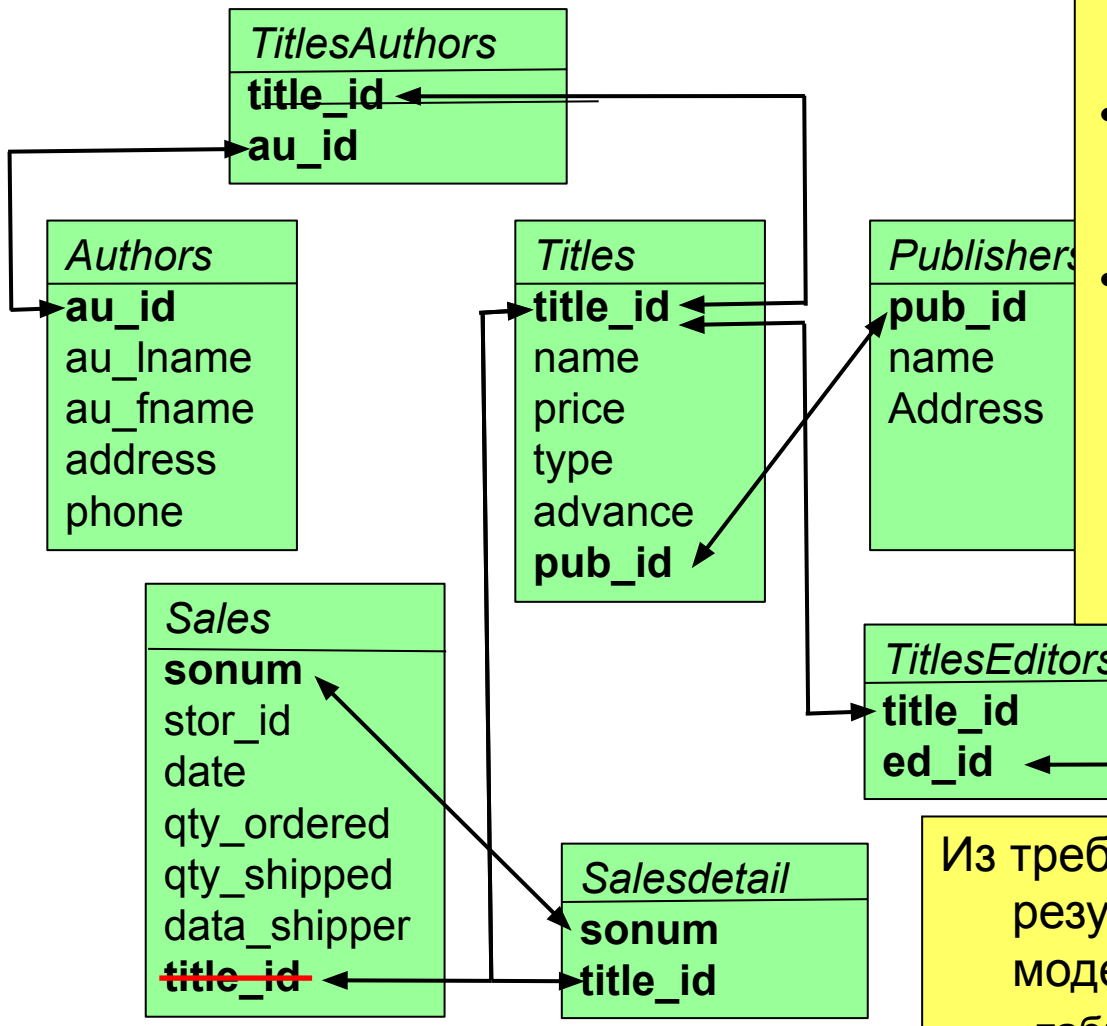
- на уровне приложения *синтезировать* общий заказ из нескольких «одно-книжных» — это не то: не появился объект «составной заказ», и для него не может быть ни~~каких действий~~, общих для всех приложений
- *нарушить* 1НФ: атрибут **title_id** сделать составным (множеством, коллекцией и др.) — плохо не только формально, но и по содержанию: трудно отслеживать ссылочную целостность, искать заказы по этому атрибуту, строить объединения таблиц по связи становится невозможным;
- вместо одного атрибута **title_id** построить несколько: **title_id1**, **title_id2**, … — не решает проблему при появлении заказа с числом *позиций* (книг) более чем их было до того, придется добавлять в *Sales* новый столбец (проблемы с преобразованием таблиц) или делать ее непрямоугольной (абсурдно)



Правильное решение: из первоначальной *Sales* сделать две таблицы:

- *главная*: *Sales* содержит сведения о заказе в целом и не содержит атрибута, указывающего на книги заказа (**title_id**), но связана отношением 1:N с дополнительной таблицей (механизм внешних ключей)
- *детализирующая*: *SalesDetail* описывает позиции всех заказов (составной атрибут в виде набора своих строк, каждая из которых должна давать доступ к названиям книг)
- *доступ к книгам* обеспечен через *SalesDetail* посредством связи с *Titles*

Первая нормальная форма: пример (результат)



Что здесь обяз
относится к
- **обязательн**
  - Связь гла
детализир
- **следствие**
  - пара *Title*
заказы n
  - не пришл
объект «

О «плоских» таблицах

Позиция заказа содержательно *подчинена* заказам, но реляционная модель не может выражать этого

В ООП объекты главной и подчиненной таблиц равнозначны

Это отрицательно сказывается на эффективности представления ООМодели в реляционном стиле

Из требований 1НФ, п
результат, что и пр
моделировании:
таблица связи межд
объектов, в отно

1НФ: искать столбцы с неатомарными значениями (требуется поиск тех, кто находится в одном штате → нужно выделить в address дополнительные атрибуты city и state).

Вторая нормальная форма

2НФ в добавление к 1НФ требует: *любой неключевой столбец должен зависеть* (= определяться функционально) *от* ***всего*** *первичного ключа* (требование для таблиц с составными первичными ключами).

Пример с полем `contract`

a) автор заключает индивидуальный контракт с заказчиком, не дожидаясь соавторов: `contract` есть функция от **`title_id`** и **`au_id`**.
Поле `contract` должно быть добавлено к таблице

TitlesAuthors
title_id
au_id

Ситуация зависимости решения от предметной области!

b) авторы заключает контракт все вместе: `contract` – функция только от **`title_id`**, (*не зависит от* **`au_id`**). Это атрибут книги, а не пары «автор, книга».
Поле `contract` должно быть добавлено к таблице

Titles
title_id
name
price
type
advance
pub_id

Что будет, если 2НФ нарушается?

- возможны странные запросы к БД (см. b);
- избыточность данных: в случае (b) надо хранить значение (общего) атрибута `contract` в разных строках таблицы *`TitlesAuthors`*

Третья нормальная форма

3НФ в добавление к к 1НФ и 2НФ требует: *ни один неключевой столбец не должен зависеть от каких бы то ни было других неключевых столбцов*. Он *зависит только* от всех столбцов первичного ключа и ни от чего больше

- Выплата гонорара зависит от номера в списке авторов.
  Попробуем вставить `au_royalper` и `au_ord` в *`Authors`*
  Это ошибка! `au_royalper` и `au_ord` зависят не только от **`au_id`**!
  То же про *`Titles`*. Это **атрибуты связи авторов и книг**:

TitlesAuthors
title_id
au_id
au_royalper
au_ord

Authors
au_id
au_lname
au_fname
address
phone
~~au_royalper~~
~~au_ord~~

- Где должен быть атрибут data_shipper? Это зависит от того, выполняется ли весь заказ сразу (наше решение), или он по позициям (тогда data_shipper надо перенести в таблицу связи).
- Здесь **зависимость решения от информационной среды**.

Sales
sonum
stor_id
date
qty_ordered
qty_shipped
data_shipper
title_id

Поддержка в СУБД выполнения требований 3НФ невозможна.

Причина для введения 3НФ та же, что и у 2НФ:
ликвидация дублей и логическая точность.

Другие нормальные формы

- ***Четвертая нормальная форма*** формализует требования, выполнение которых гарантирует от появления «дырявых» таблиц, т.е. от таких ситуаций, когда в таблицах возможны столбцы-атрибуты, которые не для всех объектов осмыслены
- ***Пятая нормальная форма*** требует, чтобы все таблицы были бы разбиты на минимально возможные части, тем самым полностью ликвидируется избыточность данных за счет дублирования ключей
  - Проблема управления целостностью упрощается до предела: изменение неключевых столбцов ничего не затрагивает
  - Однако изменение ключей становится довольно сложным: нужно проследить все вхождения ключей в другие таблицы. Но
    - это более редкое действие,
    - прослеживание изменений достаточно хорошо формализовано и не зависит от содержания базы данных, а потому обычно поддерживается на уровне СУБД.

Общие соображения о нормализации и реляционном подходе

- Правила нормализации удобны для того, чтобы проверять корректность конструируемых баз данных.
- В большинстве случаев они формализуют требования, которые понятны на уровне здравого смысла
  - Можно не знать их, но проектировать вполне приличные информационные системы
  - знание и даже применение этих правил *не гарантирует* от того, что конструируемая информационная система окажется хорошей с точки зрения области применения.
- Один из главных недостатков реляционного подхода, как уже упоминалось не раз, — принципиально равная значимость всех объектов с точки зрения манипулирования и хранения данных.
  - Это противоречит фактическому положению дел, когда объекты связаны, к примеру, иерархическими отношениями: наследование, вложенность и др.

Возможность построения сопряжения между объектной и реляционной моделями

- Эта задача не имеет однозначного решения