Д.В. Сикулер
В.В. Фомин

Технологии, методы и языки программирования

Допущено Учебно-методическим объединением по направлениям педагогического образования Министерства образования и науки Российской Федерации в качестве учебно-методического пособия для студентов высших учебных заведений, обучающихся по направлению 050200 «Физико-математическое образование»

Санкт-Петербург
Издательство политехнического университета
2012 г.

ББК 32.98
С 35
Д.В. Сикулер, В.В. Фомин

Технологии, методы и языки программирования. СПб.; Изд-во политехн. ун-та; 2012 – 166 с.

Данная публикация представляет собой выдержки лекционного материала читаемого авторами для студентов, учащихся как в бакалавариате, так и магистратуре. Отобраны и рассмотрены наиболее представительные языки программирования по своему классу технологических прерогатив и методологических отличий. Изложены теоретические положения различных подходов и методов применяемых в разработке программных систем.

Данное пособие предназначено студентам старших курсов специальностей, связанных с информационными технологиями, и может быть взято за основу дисциплин «Технологии программирования» или «Методы программирования».

Учебно-справочное издание

Фомин Владимир Владимирович – доктор технических наук, профессор кафедры «Информационных систем и программного обеспечения» Российского государственного педагогического университета им. А.И. Герцена

Сикулер Денис Валерьевич – кандидат технических наук, доцент кафедры «Информационных систем и программного обеспечения» Российского государственного педагогического университета им. А.И. Герцена

ISBN 978-5-7422-3360-2

ISBN 978-5-7422-3360-2

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В рамках данной публикации авторы не ставят перед собой задачи дать всеобъемлющий материал по технологиям и методам программирования. Мы заранее предупреждаем, что специально не включаем материал, который относится к таким отраслям разработки программного обеспечения как проектирование, Internet – программирование, мультимедийные технологии, проблемы человеко-машинного интерфейса, реинженеринг и рефакторинг и прочее. Материал лекций с 1 по 7 знакомит читателя с идеологией качественных подходов к разработке программ как структурированных алгоритмов и сложных систем. Лекции с 8 по16 больше напоминает онтологию языков программирования выбранных авторами в качестве практического примера именно различий в методах применения инструментария разработки программ. Начиная с 17-ой лекции представленный материал относиться больше к методологическим апориям, которые могут послужит основой формирования научно-практического мировоззрения применительно к разработкам программных систем и ориентированы на читателя который знает хотя бы несколько языков программирования.

Многие понятия и термины, используемые в сфере разработки программного обеспечения (ПО) и технологии программирования, не имеют однозначного, «абсолютного» толкования, что обусловлено, в частности, сложностью и многофакторностью объектов и процессов, которые они представляют. Кроме того, различные научные школы и авторы, как правило, предлагают и придерживаются своих вариантов определений для тех или иных понятий, обычно отражающих наиболее важные для них аспекты. Поэтому приведенные далее трактовки терминов не следует рассматривать как догмы или аксиомы: они носят в основном неформальный, описательный характер и лишь задают некоторые смысловые рамки для понятий, оставляя возможность для дальнейших рассуждений и уточнений.

Жизненный цикл программы — период разработки и эксплуатации программы, начиная от момента появления идеи о её создании и завершая моментом прекращения всех видов её использования. На концепцию жизненного цикла ориентировано большинство существующих технологий программирования.

Технология программирования — совокупность производственных процессов, обеспечивающих создание требуемой программы. Технология программирования изучает технологические процессы и порядок их прохождения (стадии и этапы) с использованием методов, средств и процедур. Методы обеспечивают решение конкретных задач разработки, связанных с планированием и оценкой проекта, анализом требований, проектированием, реализацией, тестированием и сопровождением программ. Средства (инструменты) предназначены для автоматизированной или автоматической поддержки методов. Различные инструментальные средства могут

объединяться в системы автоматизированной разработки программного обеспечения (ПО). Такие системы называются CASE-системами (Computer Aided Software Engineering systems). Процедуры описывают способы совместного использования методов и средств для обеспечения непрерывности технологической цепочки разработки. Они определяют порядок применения методов и инструментов, формирования отчетных документов в соответствии с требованиями, организации контроля, помогающего координировать изменения, обеспечивать качество и оценивать прогресс разработки.

Существуют технологии, используемые на конкретных этапах разработки или для решения отдельных задач этих этапов, и технологии, охватывающие несколько этапов, стадий или весь процесс разработки. В основу первых обычно положен ограниченно применимый метод, предназначенный для решения конкретной задачи. Вторые, как правило, основываются на использовании некоторой методологии.

Методология программирования — объединенная общим концептуальным (философским) подходом совокупность методов и способов их организации, используемых в ходе разработки программ. В настоящее время наиболее распространены две основные методологии — структурного и объектно-ориентированного программирования.

Технология характеризуется процессами, которые в ней задействованы, стадиями, на которых они протекают, и подходом, представляющем способ организации процессов и стадий. *Технологический процесс* — совокупность взаимосвязанных действий, преобразующих некоторые входные данные в выходные. Каждый процесс характеризуется определенными задачами, методами их решения и используемыми средствами, а также исходными данными и результатами. *Технологическая стадия* — часть действий по созданию программного обеспечения, ограниченная некоторыми временными рамками и заканчивающаяся выпуском конкретного продукта, определяемого заданными для данной стадии требованиями. Стадии состоят из этапов, которые обычно имеют итерационный характер. Иногда стадии объединяются в фазы. *Технологический подход* определяется спецификой комбинации процессов и стадий, ориентированной на разные классы программ и на особенности коллектива разработчиков.

Лекция № 1.

ТЕМА: Классические этапы разработки программного обеспечения.

Основные вопросы, рассматриваемые на лекции:

1. Понятие жизненного цикла программного обеспечения.
2. Классические этапы разработки программного обеспечения.

Жизненный цикл программного обеспечения (ПО) — период разработки и эксплуатации ПО, начиная от момента появления идеи о его создании и завершая моментом прекращения всех видов его использования. На концепцию жизненного цикла ориентировано большинство существующих методологий и технологий программирования.

Классические этапы разработки программного обеспечения.

- *Возникновение и исследование идеи*. Включает: первичное исследование идеи; детальное исследование идеи с целью выработки концепции и постановки задачи; анализ рисков и экспертизу идеи для принятия обоснованного решения о начале работы над проектом и выполнении планирования.
- *Управление*. Длится почти весь жизненный цикл ПО. Представляет собой деятельность, направленную на его реализацию с максимально возможной эффективностью при заданных ограничениях на материальные, временны*е*, трудовые и другие ресурсы, а также качество конечных результатов проекта. Одним из важнейших действий — планирование. Кроме того, управление подразумевает также: распределение работ, организацию и управление коллективом разработчиков, решение финансовых вопросов и управление бюджетом проекта, руководство проектом и контроль его выполнения, документирование различных аспектов развития проекта.
- *Анализ требований и проектирование*. Ориентированы на решение общей задачи, результатом которой должно стать четкое представление о системе, на основе которого будет создан программный код. *Анализ требований* — процесс уточнения, формализации и документирования требований заказчика к разрабатываемой программе. Основной вопрос, решаемый на данном этапе — «Что должна делать будущая система?». *Проектирование* —исследование и определение будущей структуры создаваемой системы и взаимосвязи её элементов. Основной вопрос, который решается на этом этапе — «Как система будет удовлетворять заданным требованиям?». Проектирование включает разработку архитектуры программной системы и детальное проектирование составляющих её модулей. Результатом анализа и проектирования должен стать проект, содержащий достаточные сведения для реализации системы на его основе.

• *Программирование (реализация)*. Написание программного кода создаваемого продукта в соответствии с планом и проектом разработки.

• *Тестирование и отладка*. Тесно взаимосвязанные этапы, предназначенные, соответственно, для выявления и устранения ошибок и неточностей в работе программного обеспечения с целью достижения требуемых показателей надежности и качества.

• *Ввод программы в эксплуатацию*. Осуществляемые при этом действия в значительной мере определяются спецификой и маркетингом ПО: специализированная разработка для конкретного заказчика или массовый продукт.

• *Сопровождение*. Представляет собой действия по повышению надежности программного продукта после ввода в эксплуатацию и разработку усовершенствованных версий. На этапе сопровождения решаются следующие основные задачи: *адаптация,* обычно заключающаяся в модификации функций программы; *усовершенствование*, как правило, состоящее в добавлении новых функций; *исправление* обнаруженных в ходе эксплуатации ошибок; *предупреждение* проблем, которые могут возникнуть в будущем. Для решения этих задач может быть выбран один из следующих типов сопровождения, под которым понимается степень вмешательства в программу: *незначительные* (*локальные*) изменения; *реструктурирование кода* — повторная разработка небольшой части программы при сохранении неизменным интерфейса с остальной частью; *реинжиниринг* — перестройка существующего программного продукта; *программирование заново*.

• *Завершение эксплуатации*. Несложный этап, но часто требующий организационной подготовки. Обычно он начинается с того, что пользователи заранее оповещаются о прекращении сопровождения программного продукта. Завершение эксплуатации часто является результатом того, что оказывается невозможным решить одну из задач сопровождения.

Лекция № 2.

ТЕМА: Понятие стиля программирования и его характеристики.

Основные вопросы, рассматриваемые на лекции:

1. Понятие стиля программирования.
2. Характеристики стиля программирования.
3. Основные правила повышения ясности кода программы.

Стиль программирования — это манера, в которой разработчик использует язык программирования при написании и оформлении текста

программы. Хорошим считается стиль, облегчающий восприятие и понимание программы людьми. Стиль программирования характеризуется двумя основными показателями: ясностью программы и способом использования средств языка.

Для повышения ясности при написании программы необходимо придерживаться следующих основных правил.

1. Каждый объект программы должен иметь осмысленное, содержательное имя, определяющее его назначение.
3. Объекты программы не должны иметь похожие имена. В соответствии с этим требованием нельзя также использовать в качестве имен ключевые слова языка и близкие к ним по написанию.
4. Следует избегать лишних промежуточных переменных.
5. Во избежание неоднозначности необходимо использовать скобки, поясняющие порядок выполнения, поскольку в разных языках программирования приоритет операций, в частности арифметических, различен.
6. Не следует размещать несколько операторов языка на одной строке.
7. Текст программы необходимо писать с использованием пропусков строк и отступов.
8. Текст программы должен быть прокомментирован во всех тех местах, где её понимание может быть затруднительным или смысл приведенного кода неочевиден. Комментарии должны содержать сведения, поясняющие код и отражающие цели выполнения тех или иных действий и последствия, к которым они приводят.
9. Следует придерживаться единого стиля при написании кода программы.
10. Если в практике применения языка программирования сложился более или менее определенный стиль оформления программ, то рекомендуется его использовать, если он не противоречит стандартам, принятым в организации или команде, которой выполняется разработка.

Вторая важная характеристика стиля программирования определяется тем, как разработчик использует средства и возможности, предоставляемые языком программирования. Основная рекомендация относительно применения средств языка заключается в том, что необходимо изучить и понять все возможности языка и избегать тех из них, которые могут уменьшить ясность программы, в частности связанных с плохо продуманными или реализованными особенностями и зависящих от выполнения различных трюков.

Лекция № 3.

ТЕМА: Представление информации в ЭВМ.

Основные вопросы, рассматриваемые на лекции:

1. Формы представления чисел в компьютерах.
2. Дополнительный код числа.
3. Форматы представления информации в компьютерах.

ЭВМ может работать только с информацией, представленной в числовой форме. Информация в иной форме перед обработкой в компьютере тем или иным способом преобразуется в числовой вид. Числовая информация внутри ПК кодируется в двоичной или двоично-десятичной системах счисления. Широкое распространение при программировании также получили восьмеричная и шестнадцатеричная системы счисления.

Формы представления чисел в компьютерах.

Для представления двоичных чисел в ЭВМ используются две формы:

- естественная форма, или *форма с фиксированной запятой* (точкой);
- нормальная форма, или *форма с плавающей запятой* (точкой).

В *естественной форме* представления числа записываются в виде последовательности цифр с постоянным для всех чисел положением запятой, отделяющей целую часть от дробной. Например, если для целой части числа выделено 5 разрядов, а для дробной — 2, то числа в такой разрядной сетке имеют следующий вид: +54321,10; –00372,96; +00000,04. Форма с фиксированной запятой наиболее проста и естественна для людей, но имеет небольшой диапазон представления чисел и поэтому обычно не применяется при вычислениях. Если в результате операции получается число, выходящее за диапазон допустимых значений, то происходит переполнение разрядной сетки, что приводит к искажению конечного результата. В связи с этим в современных компьютерах форма с фиксированной запятой является вспомогательной и применяется только для работы с целыми числами.

В *нормальной форме* для представления числа используются две группы разрядов, первая из которых называется *мантиссой*, а вторая — *порядком*. При этом мантисса по модулю меньше 1, а порядок является целым числом. В общем виде число N в форме с плавающей запятой представляется следующим образом:

$$N = \pm M \times B^{\pm P},$$

где M — мантисса ($|M| < 1$), P — порядок (целое число), B — основание используемой системы счисления. Мантиссу и порядок числа как правило представляют в заданной системе счисления с основанием B, а само осно-

вание B — в десятичной системе. Например, приведенные ранее десятичные числа могут быть записаны в нормальной форме так: $+0{,}543211 \times 10^5$; $-0{,}37296 \times 10^3$; $+0{,}4 \times 10^{-1}$. Форма представления с плавающей запятой имеет значительно больший, чем естественная, диапазон возможных значений чисел и поэтому является основной в современных ЭВМ. Поскольку одно и то же число в нормальной форме может быть представлено в нескольких видах (например, число +1,23 можно записать как $+0{,}123 \times 10^1$; $+0{,}0123 \times 10^2$; $+0{,}00123 \times 10^3$ и т.д.), в компьютерах используется *нормализованный вид* чисел с плавающей запятой, в котором старший разряд мантиссы отличен от нуля. Применительно к двоичной системе счисления это означает, что в старшем разряде мантиссы стоит 1 и, следовательно, $0{,}5 \le |M| < 1$.

Дополнительный код числа.

Для представления чисел с учетом их знака в ЭВМ чаще всего используется *дополнительный код*, который позволяет заменить операцию вычитания операцией сложения с отрицательным числом. Дополнительный код $N_{\text{доп}}$ числа N формируется следующим образом:

если $N \ge 0$, то $N_{\text{доп}} = 0 d_1 d_2 ... d_k$,

если $N < 0$, то $N_{\text{доп}} = 1 \bar{d}_1 \bar{d}_2 ... \bar{d}_k + 000...1$,

где старший разряд предназначен для хранения знака числа, d_i — i-я цифра числа, $\bar{d}_i$ — инвертированная i-я цифра числа. Для получения дополнительного кода отрицательного двоичного числа можно использовать следующее практическое правило: инвертировать все цифры числа, кроме последней, младшей единицы и тех нулей, которые за ней следуют, и к полученному результату слева приписать единицу. Например (знаковый разряд для наглядности отделен запятой):

N = 1101 $N_{\text{доп}}$ = 0,1101; N = –1101 $N_{\text{доп}}$ = 1,0011; N = –1010 $N_{\text{доп}}$ = 1,0110.

Подобным же образом можно получить дополнительный код отрицательных шестнадцатеричных чисел.

При выполнении арифметических операций в компьютере обычно применяется *модифицированный дополнительный код*. В модифицированном коде для представления знака числа используется 2 разряда. Второй знаковый разряд служит для автоматического обнаружения ситуации переполнения разрядной сетки. При отсутствии переполнения оба знаковых разряда имеют одинаковые цифры (нули для положительного и единицы для отрицательного числа), а при переполнении разрядной сетки цифры знаковых разрядов отличаются. Для определения конечного результата при переполнении биты полученного двоичного числа сдвигаются вправо на один разряд и слева к ним приписывается цифра, совпадающая с оставшимся после сдвига знаковым разрядом. Например (знаковые разряды для наглядности отделены запятой):

01,1101 соответствует число 00,11101 или +11101;

10,0010 соответствует число 11,00010 или –11110.

Форматы представления информации в компьютерах.

Последовательность нескольких битов или байтов, предназначенных для хранения некоторой информации, часто называют *полем данных*. Биты в поле данных нумеруются справа налево, начиная с 0-го разряда. Компьютеры могут работать с *полями постоянной и переменной длины*. Из полей постоянной длины обычно используются следующие форматы: полуслово — 1 байт, слово — 2 байта, двойное слово — 4 байта, расширенное слово — 8 байтов. Форматы полуслова и слова чаще всего используются для представления чисел с фиксированной запятой, а форматы двойного и расширенного слова — для чисел с плавающей запятой. Поля переменной длины могут иметь любой размер от 1 до 255 байт, равный целому числу байт.

При хранении чисел с фиксированной запятой в поле данных обычно старший разряд отводится для представления знака числа, а остальные — для самого числа. Например, число 34 в формате слова может храниться следующим образом:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0
Знак	Число														

При хранении чисел с плавающей запятой в поле данных обычно два старших разряда отводятся для представления соответственно знаков числа (мантиссы) и порядка. 0 в разряде знака означает плюс, а 1 — знак минус. Остальные разряды отводятся для хранения мантиссы и порядка числа. Чем больше разрядов отводится под запись мантиссы, тем выше точность представления числа. Чем больше разрядов выделено для записи порядка, тем больше диапазон чисел, представимых в данном формате. Например, если в поле формата двойное слово для хранения порядка выделено 6 разрядов, то число 34,6875, имеющее в двоичной системе в форме с плавающей запятой вид $0{,}1000101011 \times 2^{110}$, может быть представлено следующим образом:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	…	1	0
0	0	0	0	0	1	1	0	1	0	0	0	1	0	1	0	1	1	0	…	0	0
		Порядок						Мантисса													

Знак порядка

Знак числа (мантиссы)

Кодирование символьной информации обычно осуществляется в соответствии с принятым для данной ЭВМ стандартом. Наибольшее распространение в ПК получил однобайтовый код ASCII, который включает в себя стандартный набор символов и расширение. Стандартный набор, кодируемый в десятичных числах от 0 до 127, является международным и используется для представления в ЭВМ управляющих символов, букв латин-

ского алфавита, цифр, знаков пунктуации, некоторых математических и вспомогательных знаков. В расширенном наборе, кодируемом в десятичных числах от 128 до 255, представляются символы псевдографики и буквы некоторого национального алфавита. Наряду с кодом ASCII в компьютерах все чаще в последнее время используется универсальный двухбайтный код Unicode, совместимый со стандартным набором ASCII. 2 байта позволяют закодировать 65535 знаков, что достаточно для представления символов всех существующих алфавитов. Поэтому Unicode позволяет работать на ПК с данными, включающими в себя символы сразу из нескольких национальных алфавитов.

Лекция № 4.

ТЕМА: Базовые принципы организации программ ЭВМ.

Основные вопросы, рассматриваемые на лекции:

1. Принцип программного управления.
2. Понятие машинной программы и команды.
3. Структура машинных команд.
4. Типы машинных команд.
5. Адресация операндов команды.
6. Пример машинной программы.

Принцип программного управления.

В основе функционирования ЭВМ и решения задач с её помощью лежит принцип программного управления, который в общих чертах можно охарактеризовать следующим образом. В память ЭВМ вводится программа, представляющая собой последовательность команд, реализующих алгоритм решения задачи. После этого начинается автоматическое выполнение программы с первой команды. По завершению обработки очередной команды машина автоматически переходит к выполнению следующей команды, пока не будет достигнута команда, которая предписывает закончить вычисления.

Алгоритм решения задачи, заданный в виде последовательности команд на языке ЭВМ (т.е. в кодах машины), называется *машинной программой*. Команда машинной программы, по другому называемая *машинной командой*, представляет собой элементарную инструкцию машине, выполняемую ЭВМ автоматически без дополнительных указаний и пояснений. Любая программа, написанная на том или ином языке программирования, в конечном итоге преобразуется транслятором в машинную программу.

Структура машинной команды в простейшем случае включает в себя две части: операционную и адресную (рис. 1). *Операционная часть команды* — это группа разрядов в команде, предназначенная для представления кода операции машины. *Адресная часть команды* — это группа разрядов в команде, предназначенная для представления кодов адресов ячеек памяти машины, в которых хранятся операнды, используемые при выполнении заданной операции. По количеству адресов, указанных в команде, различают безадресные, одно-, двух- и трехадресные команды.

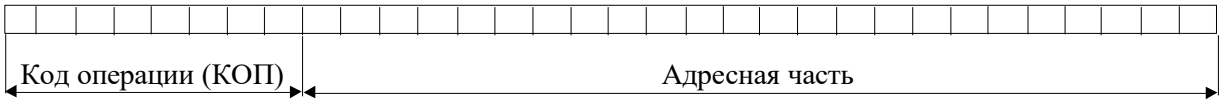

Рис. 1. Обобщенная структура машинной команды

Структура трехадресной команды имеет вид:

КОП A1 A2 A3

где A2 и A3 — адреса ячеек или регистров, где расположены, соответственно, первое и второе числа, участвующие в операции, A1 — адрес ячейки или регистра, куда следует поместить число, полученное в результате выполнения операции.

Структура двухадресной команды имеет вид:

КОП A1 A2

где A1 — это обычно адрес ячейки или регистра, где хранится первое из чисел, участвующих в операции, и куда после завершения операции должен быть записан её результат; A2 — обычно адрес ячейки или регистра, где хранится второе участвующее в операции число.

Структура одноадресной команды имеет вид:

КОП A1

где A1 в зависимости от модификации команды может обозначать либо адрес ячейки или регистра, в которой хранится одно из чисел, участвующих в операции, либо адрес ячейки или регистра, куда следует поместить число — результат операции.

Безадресная команда содержит только код операции, а информация для неё должна быть заранее помещена в определенные регистры машины.

Наибольшее применение в ПК нашли двухадресные команды.

Команда может храниться в памяти ЭВМ, как обычное слово длиной 2, 3, 4 байта и т.д. Длина команды в байтах обычно зависит от её структуры и числа разрядов, отведенных под адресную и операционную части. Восемь двоичных разрядов, отведенных под код операции, позволяют закодировать 2^8=256 различных команд, что обычно является достаточным. Длина адресной части команды зависит от числа адресов и числа разрядов, отведенных для одного адреса.

Типы машинных команд. Все машинные команды можно разделить на группы по видам выполняемых операций:

- операции пересылки информации внутри компьютера;
- арифметические операции;
- логические операции;
- операции над строками;
- операции обращения к внешним устройствам компьютера;
- операции передачи управления;
- обслуживающие и вспомогательные операции.

Для изменения естественного порядка выполнения команд служат операции передачи управления. Существуют операции безусловной и условной передачи управления. *Операции безусловной передачи управления* всегда приводят к выполнению после данной команды не следующей по порядку, а той, адрес которой в явном или неявном виде указан в адресной части команды. *Операции условной передачи управления* вызывают передачу управления по адресу, указанному в адресной части команды, только в том случае, если выполняется некоторое определенное для этой команды условие. Данное условие в явном или неявном виде указано в коде операции команды. Число команд условной передачи соответствует числу используемых условий. Команд безусловной передачи управления обычно 3:

- команда передачи управления, которая просто передает управление по заданному адресу и больше никаких действий не выполняет;
- команда передачи управления, часто называемая командой вызова процедуры или подпрограммы, которая, кроме передачи управления процедуре, еще и запоминает в специальной стековой памяти адрес следующей команды, т.е. адрес возврата из процедуры;
- безадресная команда передачи управления или команда возврата из процедуры, возвращающая управление по запомненному адресу возврата.

Адресация регистров и ячеек памяти в ПК. Адресация операндов в командах может быть:

- непосредственной — заключается в указании в команде самого значения операнда, а не его адреса;
- прямой — состоит в указании в команде непосредственно абсолютного или исполнительного адреса операнда;
- косвенной — подразумевает указание в команде регистра или ячейки памяти, в которых находится абсолютный, исполнительный адрес операнда или их составляющие;
- ассоциативной — служит для указания в команде не адреса, а идентифицирующего содержательного признака операнда, подлежащего выборке (применяется в ассоциативных запоминающих устройствах);

- неявной — адрес операнда в команде не указан, но он подразумевается кодом операции.

Пример машинной программы.

Пусть требуется вычислить выражение $y=(c(b+x))^2$. Алгоритм решения задачи включает в себя следующие операции:

- ввести в оперативную память значения исходных переменных c, x, b;
- вычислить b+x. Присвоить переменной y значение полученного результата;
- вычислить y*c. Присвоить переменной y значение полученного результата;
- вычислить y^2. Присвоить переменной y значение результата;
- вывести значение y.

Для составления программы решения данной задачи необходимо, чтобы система команд ЭВМ включала следующие команды: ввести данные, умножить, запомнить, сложить, вывести. Будем считать, что команды используемой ЭВМ двухадресные, коды операции представляются двухзначными числами и имеют следующие значения (табл. 1). Пусть в перечисленных командах адреса A1 и A2 могут меняться в пределах 00-99, а адресуемая ячейка оперативной памяти может хранить значение одной переменной или команду программы. В рассматриваемом примере вычисляемое выражение содержит четыре переменных c, b, x, y, для хранения значений которых отводится четыре ячейки с номерами 01, 02, 03, 04 соответственно. Команды программы размещаются в последующих ячейках.

Таблица 1. Таблица кодов операций

Код операции	Команда	Особенности выполнения каждой команды, и использования их адресной части
01	ВВЕСТИ	Вводятся с входного устройства A1 чисел и запоминаются в поле оперативной памяти, начинающемся с ячейки A2.
02	СЛОЖИТЬ	Складываются два операнда, один из которых храниться в ячейке оперативной памяти с адресом A1, второй — в ячейке с адресом A2.
03	УМНОЖИТЬ	Перемножаются два операнда, один из которых храниться в ячейке оперативной памяти с адресом A1, второй — в ячейке с адресом A2.
04	ЗАПОМНИТЬ	Записывается значение, хранящееся в сумматоре АЛУ, в ячейку с адресом A1. Адрес A2 в данной команде не используется.
05	ВЫВЕСТИ	Из поля оперативной памяти, начинающегося с адреса A2, на устройство вывода отправляются значения хранимых в памяти A1 чисел.
06	ОСТАНОВИТЬ	Прекращается выполнение программы. Значения адресов A1 и A2 не используются.

Программа для решения задачи включает следующие команды:

1) 01 03 01 — команда предписывает ввести три значения в поле оперативной памяти, начинающееся с ячейки 01. Предполагается, что переменные в соответствии с принятым распределением памяти вводятся в порядке: c, b, x.

2) 02 02 03 — команда предписывает сложить два числа, хранящиеся в ячейках с адресами 02 и 03.

3) 04 04 00 — команда предписывает запомнить в ячейке 04 значение результата предыдущей операции.

4) 03 01 04 — команда предписывает умножить число, хранящееся в ячейке с адресом 01, на число, хранящееся в ячейке с адресом 04. Результат равный значению c(b+x), сохраняется в АЛУ в сумматоре.

5) 04 04 00 — команда предписывает запомнить в ячейке с адресом 04 значение результата предыдущей операции.

6) 03 04 04 — команда предписывает умножить число, хранящееся в ячейке с адресом 04, на число, хранящееся в ячейке с адресом 04. Результат равный значению $(c(b+x))^2$, сохраняется в АЛУ в сумматоре.

7) 04 04 00 — команда предписывает запомнить в ячейке с адресом 04 значение результата предыдущей операции.

8) 05 01 04 — команда предписывает вывести одно число, хранящееся в ячейке с адресом 04.

9) 06 00 00 — команда предписывает прекратить выполнение программы.

Лекция № 5.

ТЕМА: Основы языка программирования Ассемблер.

Основные вопросы, рассматриваемые на лекции:

1. Основные особенности языка.
2. Классификация регистров процессора Intel 8088.
3. Основные элементы ассемблерной программы.
4. Основные команды.
5. Основные директивы.

Ассемблер — низкоуровневый машинно-ориентированный язык программирования. Реализация языка зависит от типа процессора и определяется архитектурой вычислительной системы. Ассемблер позволяет напрямую работать с аппаратурой компьютера. Программа на языке ассемблера включает в себя набор команд, которые после трансляции преобразуются в машинные команды.

Реализация большинства операций в языке Ассемблер основана на непосредственном использовании регистров процессора. Микропроцессорная память процессора Intel 8088 включает 14 двухбайтовых запоминающих регистров. По назначению они делятся на 4 группы.

- *Универсальные регистры AX, BX, CX, DX*. Каждый из универсальных регистров может использоваться для временного хранения любых данных. Разрешается работать как с регистром целиком, так и отдельно с каждой его половиной. Для обращения к старшим байтам используются регистры AH, BH, CH, DH, для обращения к младшим байтам — регистры AL, BL, CL, DL.
- *Сегментные регистры CS, DS, SS, ES* используются для хранения начальных адресов полей памяти (сегментов), отведенных в программах для команд (CS), данных (DS), стека (SS), данных при межсегментных пересылках (ES).
- *Регистры смещений IP, SP, BP, SI, DI* служат для хранения относительных адресов ячеек памяти внутри сегментов, т.е. смещений относительно начала сегментов.
- *Регистр флагов FL* содержит 9 одноразрядных флагов, управляющих прохождением программы в компьютере. Например, ZF — флаг нуля, устанавливаемый в 1, если результат операции равен нулю; SF — флаг знака, устанавливаемый в 1, если после арифметической операции получен отрицательный результат; OF — флаг переполнения, устанавливаемый в 1 при арифметическом переполнении; IF — флаг прерываний, разрешающий (значение 1) или запрещающий (значение 0) прерывания.

Ассемблерная программа может включать в себя следующие основные элементы: константы, команды, директивы и модификаторы. *Константы* языка ассемблер подразделяются на два типа: целые числа и строки. Целые числа могут быть представлены в двоичном, десятичном и шестнадцатеричном виде. Строки заключаются в одинарные или двойные кавычки.

Команды или операторы имеют следующий формат:

[<Метка>[:]] <Код оператора> [<Операнды>] [;<Комментарий>]

Основные типы команд.

- Выполнения арифметических операций. Например, команда `ADD <приемник>, <источник>` осуществляет сложение двоичных чисел, а именно содержимое источника складывается с содержимым приемника, при этом операнды должны иметь одинаковый формат. Команда `MUL <источник>` предназначена для перемножения чисел без учета знака. В этом случае операнд представляет собой 8- или 16-битовый множитель, множимое хранится соответственно в регистре AL или AX, а произведение будет сохранено в регистре AX или AX и DX. Команда `CMP <приемник>, <источник>` выполняет сравнение двоичных чисел.

• Выполнения логических операций. Например, команды OR <приемник>, <источник> и AND <приемник>, <источник> — выполняют соответственно поразрядную дизъюнкцию и конъюнкцию битов операндов.

• Пересылки данных. Например, MOV <приемник>, <источник> — пересылает в зависимости от формата операндов один байт или одно слово между регистрами или между регистром и памятью, т.е. заносит непосредственное значение в регистр или память.

• Передачи управления. Команда безусловного перехода: JMP <операнд> — выполняет передачу управления по заданному адресу. В качестве операнда указывается прямой или косвенный адрес. Команды условной передачи управления, называемые также иногда *триггерами*, имеют вид: J<условие> <метка>. Например, JG — переход, если больше, т.е., если флаги ZF=0 и SF=OF; JL — переход, если меньше, т.е., если флаг SF≠OF; JE/JZ — переход, если равно/нуль, т.е., если флаг ZF=1; JS — переход, если знак отрицательный, т.е., если флаг SF=1. К этой группе относятся также следующие команды: LOOP — управления циклом, CALL — вызова процедуры, RET — возврата из процедуры.

• Прерывания работы программы. *Прерывание* — это приостановка выполнения программы процессором с целью выполнения другой программы (*программы обработки прерывания*), после завершения которой, как правило, продолжается выполнение прерванной программы с момента её прерывания. Основной представителем этой группы является команда INT <номер прерывания>, прерывающая выполнение программы и передающая управление подпрограмме обработки, заданной номером прерывания.

• Управления процессором. Например, ST* и CL* — установки и сброса флагов, HLT — останова, WAIT — ожидания, NOP — холостого хода.

• Обработки строк символов. Например, MOVS — пересылки, CMPS — сравнения, LODS — загрузки, SCAS — сканирования.

Директивы или псевдооператоры ассемблера представляют собой инструкции транслятору. В отличие от команд, являющихся инструкциями машине, директивы обрабатываются только в ходе трансляции программы в машинный код, а не в ходе её выполнения на компьютере. Директивы имеют следующий формат:

[<Идентификатор>] <Код псевдооператора> [<Операнды>] [;<Комментарий>]

Для идентификации переменных и полей данных используются директивы определения данных. Они позволяют объявить переменную, задав её имя, присвоить ячейкам памяти начальные значения или зарезервиро-

вать область памяти для последующего сохранения в ней некоторых значений. Имеют следующий формат:

```
[<имя>] D<формат> <выражение>[, <выражение>][,…]
```

где имя — набор символов, начинающийся с буквы, используемый для ссылки на переменную или поле данных. Формат может иметь следующие значения: B — байт, W — слово (2 байта), D — двойное слово (4 байта), Q — 8 байтов, T — 10 байтов. Выражение показывает, какое количество элементов памяти необходимо выделить и какие данные там будут содержаться.

Также в программе используются директивы определения сегментов:

```
<имя сегмента> segment
<содержимое сегмента>
<имя сегмента> ends
```

В программе можно использовать 4 сегмента (по числу сегментных регистров) и для каждого из них в сегменте кода следует указывать соответствующий регистр сегмента директивой ASSUME. Например:

```
COMMANDS segment
assume CS:COMMANDS, DS:DATA, SS:STACK
…
COMMANDS ends
```

После директивы ASSUME необходимо загрузить адрес начала сегмента данных в регистр DS. Инициализация сегментных регистров CS и SS выполняется автоматически.

Модификаторы используются в командах и директивах ассемблера для выполнения некоторых операций над операндами, обрабатываемых в ходе трансляции программы.

Ниже приведен пример ассемблерной программы с комментариями, предназначенной для вычисления суммы целых чисел от 1 до N (здесь и далее знак ⇒ означает продолжение строки программы на следующей строке индекса).

```
TITLE sum1n.asm - программа вычисляет сумму целых чисел от 1 ⇒
до N (N < 1000)
STACK segment   ; Начало сегмента стека
dw 64 dup(?)
STACK ends   ; Конец сегмента стека
DATA segment   ; Начало сегмента данных
ZeroCode = 48   ; ASCII-код цифры 0
; Задаем строки с сообщениями, используемые в ходе работы ⇒
программы
; Символ "$" в конце строк - требование DOS
TitleMsg db "Программа вычисляет сумму целых чисел от 1 до N ⇒
(N < 1000)", 13, 10, "$"
InputMsg db "Введите число N: ", "$"
ResultMsg db 13, 10, "Сумма равна ", "$"
; Объявляем необходимые переменные
```

```
M10 dw 10   ; Множитель 10
N dw ?   ; Введенное число N в двоичном формате
P10 dw 1   ; Степень 10
StrN db 4, 5 dup(0)   ; Число N в формате строки ⇒
(первый байт - максимальное число вводимых символов)
StrSum db 6 dup(0), "$"   ; Сумма в формате строки
Sum dd 0   ; Вычисляемая сумма
DATA ends   ; Конец сегмента данных

CODE segment   ; Начало сегмента кода

assume CS:CODE, DS:DATA, SS:STACK   ; Связываем сегментные ⇒
регистры с сегментами
main proc far   ; Объявляем главную программную процедуру ⇒
(требование DOS)
; Записываем адрес префикса программного сегмента в стек ⇒
(требование DOS)
push ds
sub ax, ax
push ax
mov ax, DATA   ; Пересылаем адрес сегмента данных в регистр AX
mov ds, ax     ; Устанавливаем регистр DS на сегмент данных
; Выводим сообщение о назначении программы
mov AH, 09h   ; Определяем в качестве обработчика прерывания ⇒
DOS-функцию вывода строки на экран
mov DX, offset TitleMsg   ; Задаем смещение к началу строки
int 21h   ; Выводим строку
; Выводим сообщение с запросом на ввод числа N
mov AH, 09h
mov DX, offset InputMsg
int 21h
; Запрашиваем число и сохраняем его в переменной StrN
mov AH, 0Ah   ; Определяем в качестве обработчика прерывания ⇒
DOS-функцию ввода строки символов
mov DX, offset StrN   ; Задаем смещение к началу строки
int 21h   ; Запрашиваем строку
; Теперь в переменной StrN хранятся следующие значения ⇒ (бай-
ты перечисляются слева направо):
; 1-й байт - максимальное число вводимых символов ⇒ (для дан-
ного примера - 4)
; 2-й байт - действительное число введенных символов
; последующие байты - ASCII-коды введенных символов, ⇒ послед-
ний из которых всегда - 13 (код клавиши Enter)

; Переводим введенное число в двоичный формат
; (в данный момент число представлено двоичными кодами цифр, ⇒
т.е. ASCII-кодами введенных символов)
mov P10, 1   ; Заносим 1 в степень 10
```

```
mov DI, 0   ; Обнуляем регистр, в котором будем сохранять ⇒
результат
mov SI, offset StrN + 1   ; Заносим в регистр SI адрес ⇒ ячей-
ки, в которой хранится количество цифр числа ⇒
(хранится во втором байте переменной StrN)
mov BL, [SI]   ; Заносим в младший байт регистра BX ⇒ количе-
ство цифр числа
mov BH, 0   ; Обнуляем старший байт регистра BX
mov SI, offset StrN + 2   ; Заносим в регистр SI адрес ⇒ ячей-
ки первой цифры введенного числа
LS2B: mov AL, [SI + BX - 1]   ; Заносим в младший байт ⇒ реги-
стра AX ASCII-код текущей цифры числа (цифры перебираем справа
налево)
sub AL, ZeroCode   ; Вычитаем из кода цифры код цифры 0 для ⇒
получения соответствующего числа
mov AH, 0   ; Обнуляем старший байт регистра AX
mul P10   ; Умножаем значение регистра AX на число 10 в ⇒ сте-
пени, соответствующей позиции текущей цифры
add DI, AX   ; Добавляем полученное произведение к конечному ⇒
результату
mov AX, P10   ; Сохраняем степень 10 в регистре AX
mul M10   ; Увеличиваем степень 10 на порядок (содержимое ⇒
регистра AX умножаем на 10)
mov P10, AX   ; Сохраняем новое значение степени в ⇒ соответ-
ствующей переменной
dec BX   ; Определяем номер следующей цифры
jnz LS2B   ; Переходим к обработке следующей цифры, если ⇒ пе-
ребраны еще не все цифры числа

; Теперь в регистре DI находится введенное число в двоичном ⇒
формате
; Запоминаем его в переменной N
mov N, DI

; Вычисляем сумму от 1 до N
mov AX, 0   ; Обнуляем регистр AX
LSM: inc AX   ; Увеличиваем содержимое регистра AX на 1, ⇒ оп-
ределяя очередное слагаемое
add word ptr Sum[2], AX   ; Добавляем содержимое регистра AX ⇒
к младшему слову суммы
adc word ptr Sum[0], 0   ; Добавляем 0 к старшему слову ⇒ сум-
мы с учетом переноса от предыдущего сложения ⇒
(учитываем перенос, если он был)
cmp AX, N   ; Если содержимое регистра AX меньше числа N,
JB LSM      ; переходим к следующему слагаемому

; Переводим полученную сумму в форму строки
```

```
mov SI, offset StrSum + length StrSum - 1   ; Заносим в ⇒ ре-
гистр SI адрес ячейки для хранения последней цифры числа
mov DX, word ptr Sum[0]   ; Заносим в регистры DX:AX число, ⇒
которое будем переводить в строку
mov AX, word ptr Sum[2]
LB2S: div M10   ; Делим число, находящееся в регистрах DX:AX ⇒
на 10
add DL, ZeroCode   ; Добавляем к остатку от деления код ⇒ циф-
ры 0 для получения ASCII-кода цифры
mov [SI], DL   ; Сохраняем найденную цифру в соответствующей ⇒
позиции строки с результирующим числом
dec SI   ; "Переходим" к предыдущей цифре числа
mov DX, 0   ; Обнуляем регистр DX
cmp AX, 0   ; Если частное отлично от нуля (определены не ⇒
все цифры числа),
jne LB2S     ; то переходим к вычислению очередной цифры числа

; Выводим сообщение, предшествующее выводу результата
mov AH, 09h
mov DX, offset ResultMsg
int 21h
; Выводим полученное число
mov AH, 09h
mov DX, offset StrSum
int 21h

ret   ; Выходим из основной процедуры
main endp   ; Конец основной процедуры

CODE ends   ; Конец сегмента кода
END MAIN   ; Конец программы (требование DOS)
```

Лекция № 6.

ТЕМА: Методы структурного программирования.

Основные вопросы, рассматриваемые на лекции:

1. Понятие структурного программирования и его требования.
2. Основные конструкции структурных программ.
3. Понятие псевдокода.
4. Метод пошаговой детализации.
5. Сквозной структурный контроль.

Структурное программирование — программирование, ориентированное на обеспечение простоты написания и понимания программ людь-

ми, а не компьютерами. Преследуемые цели: писать программы минимальной сложности, заставить программиста мыслить ясно, облегчить восприятие программы. Для их достижения структурная программа должна удовлетворять следующим основным требованиям.

- Текст программы должен представлять собой композицию трех основных конструкций: последовательное соединение (следование), условное предложение (ветвление) и повторение (цикл).
- Употребление операторов перехода типа GOTO следует избегать всюду, где это возможно. Наихудшим применением GOTO считается переход на оператор, расположенный выше (раньше) в тексте программы.
- По возможности следует отказаться от использования оператора ELSE Конструкция ELSE нужна только в той редкой ситуации, когда конструкция THEN изменяет одну из переменных в условии.
- Программа должна быть написана в приемлемом стиле, облегчающем её понимание и модификацию.
- Каждый модуль должен иметь ровно один вход и один выход.
- Программа представляет собой простое и ясное решение задачи.

Структурные программы составляются из трех основных строительных конструкций (рис.1) управления вычислительным процессом:

- *следование* — обозначает последовательное выполнение действий;
- *ветвление* — соответствует выбору одного из двух вариантов действий в зависимости от условия (значения предиката);
- *цикл с предусловием* — определяет повторение действий до тех пор, пока не будет нарушено заданное условие, выполнение которого проверяется перед началом каждого повторения.

Помимо базовых могут использоваться еще три управляющие конструкции, которые легко реализуются на их основе:

- *выбор* — обозначает выбор одного варианта действий из нескольких в зависимости от значения некоторой величины или условия;
- *цикл с постусловием* — определяет повторение действий до тех пор, пока не будет выполнено некоторое условие, проверка которого осуществляется после каждого повторения;
- *цикл с заданным числом повторений* (*счетный цикл*) — определяет повторение действий указанное число раз.

Псевдокод. Псевдокод представляет собой текстовую нотацию, предназначенную для формализованного описания структуры программы с необходимой степенью детализации. Для этих целей в псевдокоде имеется возможность использовать все конструкции структурного программирования, представленные формализованно, скомпонованные вместе с фрагментами, определяющими выполняемые действия и проверяемые условия. Эти фрагменты, в зависимости от требуемого уровня детализации и этапа разработки, могут быть заданы как на естественном или формальном языке,

так и посредством соответствующих элементов выбранного языка программирования. Указанные особенности обуславливают удобство использования псевдокода при применении метода пошаговой детализации.

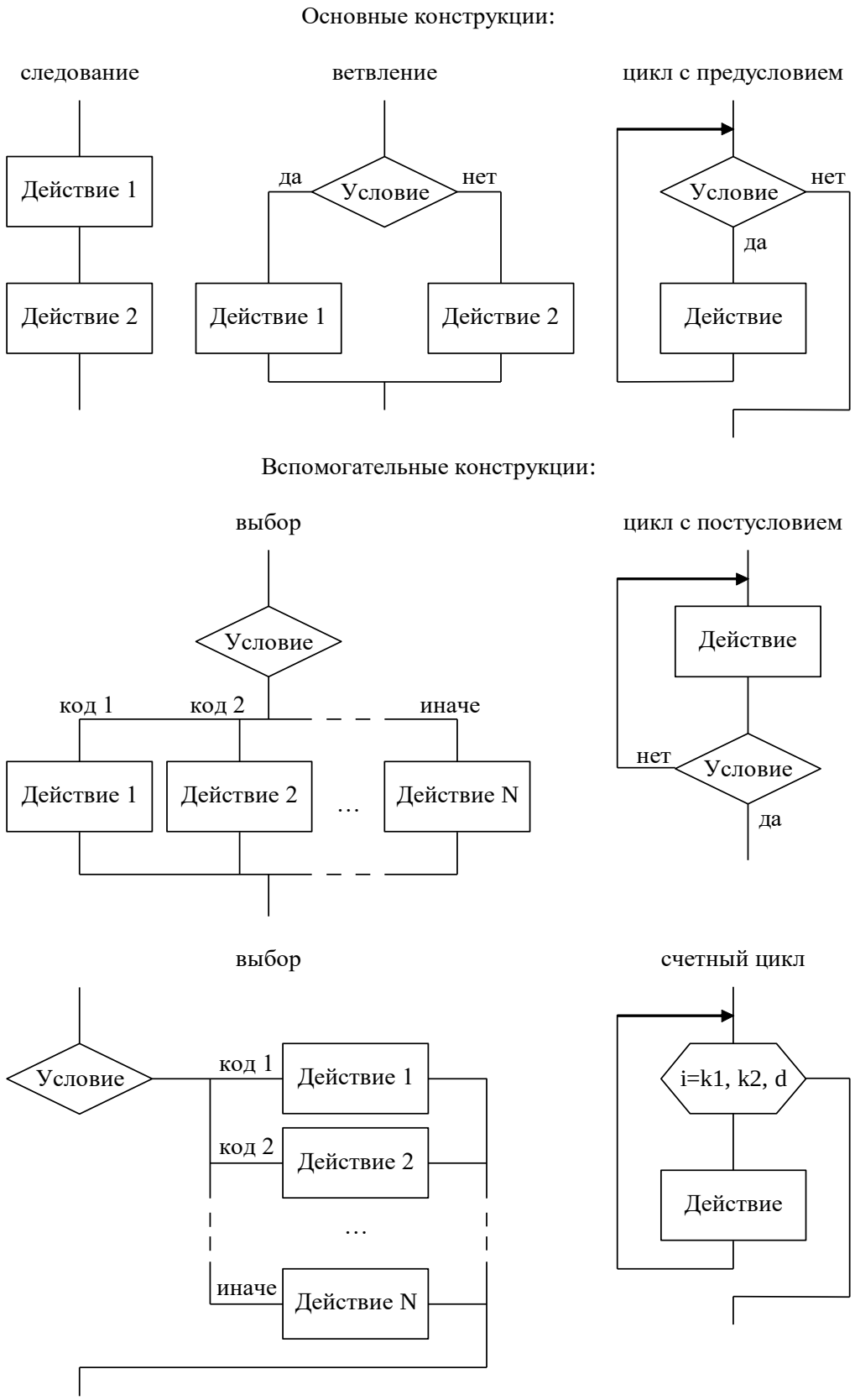

Рис. 1. Основные конструкции «структурного программирования».

Основные конструкции структурного программирования, представленные на псевдокоде

Конструкция	Псевдокод	Конструкция	Псевдокод
Следование	<действие 1> <действие 2>	Цикл с предусловием	ЦИКЛ-ПОКА <условие> <действие> ВСЁ-ЦИКЛ
Ветвление	ЕСЛИ <условие> ТО <действие 1> ИНАЧЕ <действие 2> ВСЁ-ЕСЛИ	Цикл с постусловием	ВЫПОЛНЯТЬ <действие> ДО <условие>
Выбор	ВЫБОР <условие> <код 1>: <действие 1> <код 2>: <действие 2> … ИНАЧЕ: <действие N> ВСЁ-ВЫБОР	Счетный цикл	ДЛЯ <индекс> = <k1>, <k2>, <d> <действие> ВСЁ-ЦИКЛ

Метод пошаговой детализации. Для получения текстов программ и составляющих их модулей, соответствующих принципам структурного программирования, рекомендуется использовать *пошаговую детализацию*. Сущность данного метода заключается в следующем: первоначально управляющая логика программы или модуля описывается в терминах гипотетического языка «очень высокого уровня», а затем каждое предложение постепенно детализируется в терминах языка более низкого уровня до тех пор, пока, наконец, не будет достигнут уровень используемого языка программирования. На протяжении всего процесса логика выражается основными конструкциями структурного программирования. Таким образом, метод пошаговой детализации предполагает разбиение процесса разработки программного текста на ряд шагов. На первом шаге описывается общая схема работы модуля в наглядной текстовой форме с использованием базовых («очень крупных») понятий, ориентированная на восприятие и понимание её человеком. На каждом следующем шаге выполняется уточнение и детализация одного из понятий, полученного на каком-либо предыдущем шаге. Уточняемое понятие выражается либо через новые, более «мелкие» понятия, либо в терминах (конструкциях) языка программирования, выбранного для разработки модуля. В любом случае, операции управления вычислительным процессом следует представлять с помощью конструкций структурного программирования, а полученное описание должно быть наглядным и достаточно простым для понимания. Этот процесс завершается, когда все уточняемые понятия будут выражены на используемом языке программирования. Последний шаг состоит в формировании готового текста модуля путем замены всех уточняемых понятий соответствующими программными фрагментами и выражения всех вхождений конструкций структурного программирования через средства языка.

Сквозной структурный контроль. Сквозной структурный контроль представляет собой совокупность технологических операций контроля,

используемых с целью как можно более раннего обнаружения ошибок, допущенных в процессе разработки программной системы. Термин «сквозной» в названии указывает на то, что контроль осуществляется на всех этапах разработки. Термин «структурный» означает, что для каждого этапа определены четкие рекомендации относительно набора и способов выполнения контролирующих операций. Помимо раннего обнаружения ошибок, метод позволяет обеспечить своевременную подготовку качественной документации по проекту.

Сквозной структурный контроль реализуется на специальных контрольных сессиях, в которых, помимо разработчиков, могут участвовать специально приглашенные эксперты, что, обычно, позволяет достичь лучших результатов. Одна из первых сессий организуется на этапе определения спецификаций разрабатываемой системы. На этой сессии проверяется полнота и точность спецификаций. При этом целесообразно присутствие заказчика или специалиста по предметной области, которые могут определить, насколько правильно выполнены спецификации и насколько они соответствуют поставленным целям и задачам. На сессиях, проводимых в процессе проектирования, вручную по частям проверяются модели и алгоритмы разрабатываемой программной системы на конкретных наборах данных и полученные результаты сверяются с существующими спецификациями. Основная задача проведения контрольных мероприятий в этом случае заключается в том, что необходимо убедиться в правильности понимания спецификаций и проанализировать достоинства и недостатки концептуальных решений, закладываемых в проект. На этапе реализации проверяется последовательность реализации модулей, их тексты, а также набор тестов для выявления ошибок.

Для всех этапов целесообразно иметь сведения о наиболее часто встречающихся ошибках, подготовленные по литературным источникам и на основании опыта предыдущих разработок. Это позволяет направить основные усилия на конкретные аспекты, а не проверять все подряд. Все найденные ошибки фиксируются в специальных документах, на основании которых в дальнейшем выполняются необходимые изменения.

Лекция № 7.

ТЕМА: Структурные преобразования блок-схем.

Основные вопросы, рассматриваемые на лекции:

1. Простые преобразования.
2. Метод введение дублирующих элементов.
3. Метод введение переменной состояния.

Простые преобразования. Связаны с представлением основных конструкций структурного программирования посредством единственного блока «Процесс». При необходимости упрощения схемы конструкция «сворачивается» в этот блок, а в случае детализованного описания — «разворачивается» из блока.

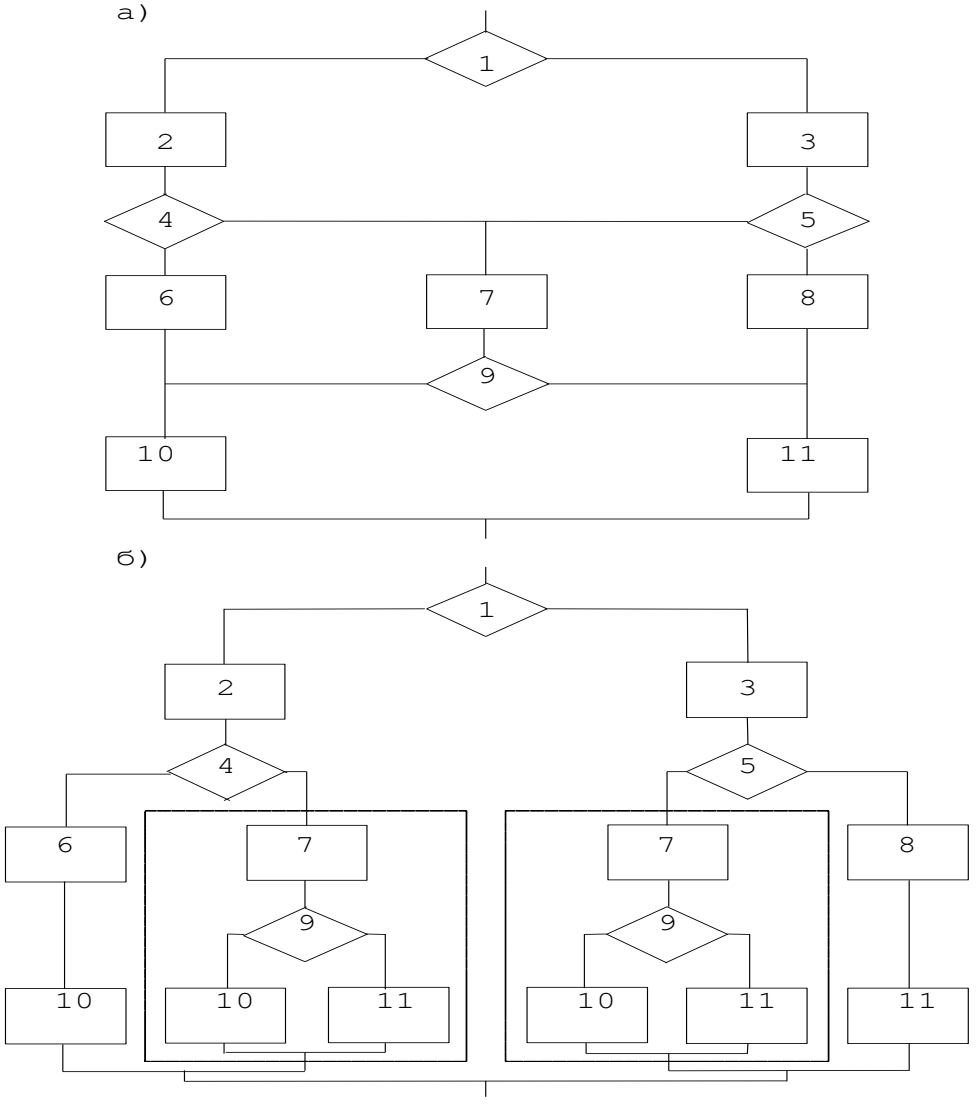

Рис. 1. Структурные преобразования: «дублирование блоков».

Метод введение дублирующих элементов. Данное преобразование позволяет исключить из блок-схемы варианты передачи управления, не согласующиеся с концепциями структурного программирования, путем введения в неё по определенным правилам дополнительных элементов, эквивалентных уже существующим. Пусть, например, имеется схема, представленная на рис.1.а. Стремление использовать блоки 7, 9, 10, 11 в ветвях, начинающихся в блоках 4 и 5, привело к запутанным связям по управлению. Дублируя блоки 7, 9, 10, 11 можно привести исходную схему к структурированному виду (рис.1.б). При дублировании, строя очередной фрагмент после разветвления, каждый раз вводят необходимые блоки, не обращая внимания на то, что они уже введены на альтернативных участках схемы.

Метод введение переменной состояния. Данный подход также позволяет устранить из схемы неструктурные способы передачи управления. Процесс преобразования включает в себя следующие операции:

1. Каждому блоку схемы приписывается номер от 1 до n, где n — исходное число блоков в схеме. Нуль резервируется для блока, представляющего последний исполняемый элемент, после которого следует выход из схемы.

11. Вводится новая переменная, принимающая значение в диапазоне от 0 до n, называемая *переменной состояния*.

12. Вводятся n операций присвоения значений переменной состояния. С каждым блоком связывается одна или несколько (для блока «Решение» или «Выбор» в соответствии с количеством его выходов) операций, в которых значения переменной становится равным номеру очередного исполняемого блока.

13. Вводятся n операций анализа переменной состояния в соответствии со следующим принципом: если значение переменной состояния равно m (0<m<=n), то управление передается блоку с номером m. Операция проверки значения переменной на равенство нулю резервируется для случая выхода из схемы.

14. Строится новая схема в виде цикла с предусловием с вложенными в него блоками «Решение» для анализа переменной состояния и блоками исходной схемы, к каждому из которых добавлены соответствующие элементы изменения значений переменной состояния. Перед циклом добавляется блок для присвоения единицы переменной состояния, а условием выхода из цикла является равенство значения переменной нулю.

На приведенных ниже рисунках приведен пример преобразования на основе введения переменной состояния.

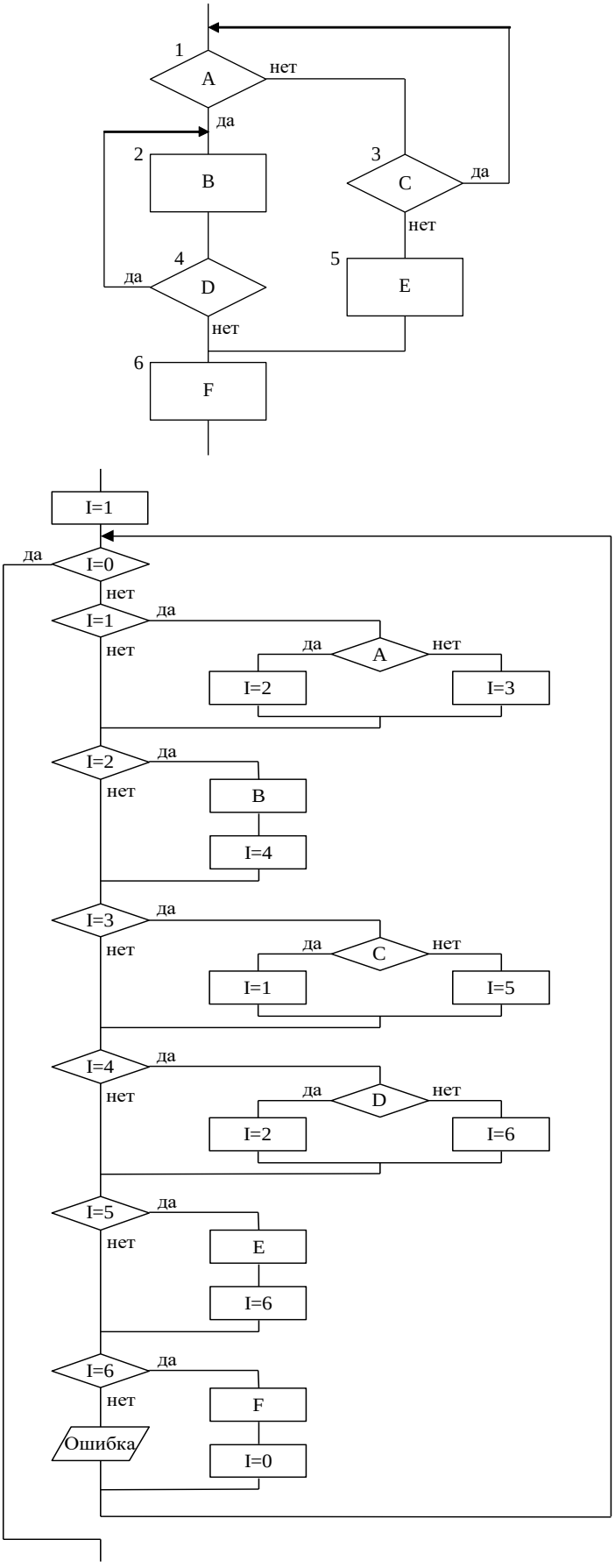

Рис. 2. Структурные преобразования. Введение переменной состояния.

Лекция № 8.

ТЕМА: Модульное программирование.

Основные вопросы, рассматриваемые на лекции:

1. Определение модульного программирования.
2. Понятие программного модуля.
3. Связность модуля и её виды.
4. Сцепление модулей и его виды.
5. Порядок разработки программного модуля.

Модульное программирование представляет собой метод программирования, в котором программная система разделяется на автономно разрабатываемые части, называемые программными модулями. Целью данного метода является упрощение процессов создания больших программ. Модульное программирование позволяет обеспечить независимость компонент программной системы и организовать их в иерархические структуры. Обеспечение независимости компонент означает разделение системы на такие части, между которыми должно остаться по возможности меньше связей.

Программный модуль — это самостоятельный программный продукт, предоставляющий средства для решения некоторого фиксированного набора задач и пригодный к использованию в составе разных программных систем при соблюдении определенных условий. Каждый модуль программируется, компилируется и отлаживается отдельно от других модулей, то есть физически отделен от них.

Для того чтобы набор модулей, на которые разделяется создаваемая программная система, способствовал уменьшению её сложности и упрощению процессов разработки, необходимо чтобы модули соответствовали некоторым критериям. Одним из основных среди них является требование независимости модулей друг от друга. Довольно высокая степень независимости модулей достигается путем усиления внутренних связей в каждом модуле и ослаблением взаимосвязей между ними.

Прочность или связность модуля — это мера связей между его компонентами. Чем выше прочность модуля, то есть чем больше взаимосвязаны его элементы, тем меньше число межмодульных связей и, соответственно, взаимовлияние модулей. Типы связности в порядке убывания их уровня:

1. *Функциональная связность*. Модуль предназначен для выполнения одной определенной функции, реализацию которой обеспечивают все входящие в него компоненты. Такой модуль имеет максимальную прочность и

не может быть разбит на другие модули, имеющие функциональную связность.

2. *Последовательная связность*. Модуль обеспечивает выполнение единственной функции, которая разделена на ряд последовательно осуществляемых независимых операций, реализуемых различными частями модуля. Результат одной операции представляет собой исходные данные для следующей. Такой модуль может быть разделен на несколько других модулей, как с последовательной, так и с функциональной связностью.

3. *Информационная или коммуникативная связность*. Модуль реализует несколько независимо используемых функций (операций), предназначенных для работы с одной и той же структурой данных, особенности организации и формат которой сокрыты в пределах модуля. В результате, при изменении структуры данных не приходится корректировать компоненты других модулей.

4. *Процедурная связность*. Модуль объединяет в своем составе компоненты, предназначенные для реализации одного процесса. Такие модули могут возникнуть при разбиении длинной программы на части в соответствии с операциями передачи управления, но без определения какого-либо функционального базиса при выборе разделительных точек. Модуль с процедурной связностью может также появиться в результате группировки в нем альтернативных частей программы.

5. *Временная связность*. Модуль содержит функционально несвязанные компоненты, используемые параллельно или в течение некоторого интервала времени. Функции, реализуемые модулем с временной связностью, обычно могут выполняться в любом порядке. Связность такого типа имеет место, например, в том случае, когда всё множество требуемых в момент начала работы программы функций объединяется в отдельный модуль инициализации.

6. *Логическая связность*. Модуль объединяет компоненты, выделенные по признаку функционального подобия (сходного назначения). Для выполнения некоторой операции в таком модуле всегда используется одна какая-либо его часть. Модуль с логической связностью часто реализует альтернативные варианты одной операции, например, сортировки массива чисел и массива строк.

7. *Случайная связность* (*по совпадению*). Модуль включает в себя компоненты, объединенные произвольным образом, связь между которыми отсутствует или незначительна.

При разработке программных систем рекомендуется использовать модули, имеющие функциональную, последовательную и информационную связность.

Сцепление модуля — это мера его зависимости от других модулей. Модуль независим, если он не содержит о других модулях никакой информации и может быть модифицирован без их переделки. Чем меньше

сцепление модуля, тем меньше сведений он включает о других модулях и тем больше степень его независимости от них. Типы сцепления в порядке увеличения их уровня:

1. *Независимое сцепление*. Модуль не использует другие модули для реализации своих функций и не обрабатывает одни и те же данные с другими модулями.

2. *Сцепление по данным* (*параметрическое*). Модуль обменивается с другими модулями данными, представленными в форме простых параметров со скалярными значениями. Вызывающий модуль «знает» только имя и назначение выполняемой операции, а также типы и значения её параметров. Изменения в способе реализации операции и структурах данных, используемых для её выполнения, не затрагивают другие модули.

3. *Сцепление по образцу*. Модуль обменивается с другими модулями данными, организованными в некоторые структуры. Изменение структуры передаваемых данных в одном модуле обычно приводит к необходимости модификации взаимодействующих с ним модулей.

4. *Сцепление по общей области*. Модуль работает с областью данных (памяти), которая является общей для нескольких модулей. В этом случае изменения, сделанные в общей области данных одним модулем, могут влиять на работу другого модуля и служить источником трудноуловимых ошибок в работе программы.

5. *Сцепление по управлению*. Один из модулей управляет логикой работы другого с помощью передачи специальных флагов или кодов. В этом случае управление работой подчиненного модуля частично или полностью выполняется извне, то есть вызывающий модуль «знает» особенности организации и внутренние функции вызываемого модуля.

6. *Сцепление по внешним ссылкам*. Модуль имеет доступ к компонентам другого модуля через внешнюю точку входа.

7. *Сцепление по содержимому* (*коду*). Один модуль содержит обращения (прямые ссылки) к внутренним компонентам другого, например, передает управление внутрь или обрабатывает внутренние данные. В этом случае содержимое каждого модуля должно учитываться в процессе разработки другого.

При разработке программных систем рекомендуется использовать модули, имеющие независимое сцепление, сцепление по данным или по образцу.

Порядок разработки программного модуля.

1. *Проектирование спецификации модуля и её проверка*. На этом этапе определяются внешние характеристики модуля, что выражается в виде его спецификации, которая содержит все сведения, необходимые для его использования, и ничего больше. Спецификация должна содержать следующие сведения:

- Имя модуля, используемое для его вызова.

- Функция (назначение). Определяются функции, выполняемые модулем.
- Список параметров. Указывается число и порядок параметров, передаваемых модулю.
- Входные параметры. Дается точное описание всех входных параметров, включающее определение их формата, размеров, атрибутов, единиц измерения и допустимых диапазонов значений.
- Выходные параметры. Дается точное описание всех данных, возвращаемых модулем, аналогичное описанию входных параметров. Определяется функциональная связь между входными и выходными данными, то есть указывается, какие выходные данные какими входными порождаются. Должны быть также приведены выходные данные, порождаемые модулями в случае, когда входные параметры имеют некорректные значения.
- Внешние эффекты. Приводится описание всех внешних для ПО событий, происходящих при работе модуля.

Правильность спецификаций каждого модуля должна быть проверена путем сравнения их с информацией о связях (способах взаимодействия) между модулями, полученной при проектировании структуры программной системы, и анализом их всеми программистами, разрабатывающими вызывающие модули.

2. *Выбор языка программирования*.

3. *Выбор алгоритма и структур данных*. Следует выяснить наличие готовых алгоритмов для решения соответствующих модулю задач и возможности по их адаптации, а также выбрать используемые структуры данных. При необходимости требуемые алгоритмы и структуры данных разрабатываются.

4. *Программирование* (*кодирование*) *модуля*. На выбранном языке программирования пишется текст модуля. Рекомендуется придерживаться следующей последовательности действий при написании модуля.

- Написать первое и последнее предложения, например заголовок модуля и оператор его окончания.
- Описать структуры данных и переменные, используемые при взаимодействии с другими модулями.
- Объявить внутренние структуры данных и переменные модуля, используемые для реализации выполняемых им функций.
- Написание функционального кода модуля с постепенной его детализацией. Этот шаг итеративный и предполагает последовательную детализацию логики модуля, начиная с достаточно высокого уровня абстракции и заканчивая готовым программным текстом. На этом шаге обычно используются методы пошаговой детализации и структурного программирования.

5. *Шлифовка текста модуля*. Текст модуля приводится к завершенному виду в соответствии с принятыми для разработки программной системы стандартами качества.

6. *Ручная проверка правильности работы модуля*.

7. *Компиляция модуля*.

Лекция № 9.

ТЕМА: Объектно-ориентированное программирование.

Основные вопросы, рассматриваемые на лекции:

1. Ключевые понятия ООП.
2. Основные принципы ООП: инкапсуляция, наследование, полиморфизм.
3. Средства ограничения доступа к компонентам класса.
4. Абстрактные методы и классы.

В основе объектно-ориентированного программирования (ООП) лежат понятия класса и объекта. *Класс* представляет собой некую абстракцию, объединяющую собой определенное множество элементов, характеризующихся некоторой общностью, например общей структурой, принципами действия и реакции. *Объект* (*экземпляр класса*) представляет собой элемент множества, определяющего класс. Объект реализует или воплощает класс в действительности. С практической точки зрения использования в некоем языке программирования класс можно рассматривать как своеобразный тип, определяющий набор данных, используемый для хранения информации, и набор операций, необходимый для обработки этих данных. Т.о., класс задает допустимые значения данных и действий над ними. Объект же представляет собой переменную соответствующего типа или класса, которая инициируется, как правило, посредством специальной функции или оператора.

Структура данных, поддерживаемых классом, задается с помощью переменных и констант, называемых *полями*. Операции по обработке этих данных определяются посредством подпрограмм, называемых *методами*. Вызовы методов называются *сообщениями*. Сообщение включает в себя следующие части: конкретный объект, которому оно адресуется, имя метода, определяющего необходимые действия над объектом, и необязательный набор параметров, конкретизирующих выполняемую операцию. Таким образом, вычисления в объектно-ориентированной программе осуществляются с помощью сообщений, передаваемых от одного объекта к другому.

При разработке с помощью ООП программ с графическим интерфейсом очень часто приходится определять обработчики событий и оперировать ими. *Обработчик события* представляет собой специальный метод, как правило с предопределенным именем, вызываемый автоматически при получении от операционной системы или другой программы того или иного стандартного сообщения и предназначенный для реализации некоторых действий в ответ на это сообщение. В данном контексте сообщение пред-

ставляет собой реакцию операционной системы или какой-либо программы в ответ на действия, происходящие в вычислительной системе, в которой она функционирует (например, нажатие кнопок клавиатуры или мыши, приход пакета данных по сети и т.д.). С точки зрения использования в программе обработчик события ничем не отличается от метода, в частности он объявляется в классе как обычный метод и может быть при необходимости выполнен вне зависимости от получения соответствующего сообщения.

Классы могут иметь два вида полей и методов. К первому типу относятся так называемые *поля и методы экземпляра/объекта класса*. Поля экземпляра класса представляют собой набор переменных, значения которых в каждый момент времени характеризуют особенности текущего состояния соответствующего объекта. Методы объекты предназначены для выполнения операций над его полями. Ко второму типу принадлежат *поля и методы класса*. Поля класса — это переменные, которые относятся непосредственно к классу, а не к конкретному объекту, поэтому они имеют одно общее значение для всех объектов. Методы класса предназначены для выполнения операций, присущих классу в целом и обычно реализующих действия, не связанные с конкретными объектами класса.

ООП характеризуется тремя основными свойствами: инкапсуляцией, наследованием и полиморфизмом. *Инкапсуляция* — это способ объединения в единое целое подпрограмм и данных, которые они обрабатывают, и определения интерфейса для оперирования ими. Инкапсуляция предполагает сокрытие внутренней прикладной структуры данных и подпрограмм класса и определение внешних интерфейсов доступа к ним, т.е. набора компонент, посредством которых можно оперировать объектом данного класса. Следствием этого обычно является изолирование внутри класса полей и, при необходимости, некоторых методов от остальной части программы.

Наследование представляет собой отношение между классами, при котором описание одного класса основано на описании другого (*одиночное наследование*) или нескольких других (*множественное наследование*) классов. Порождаемый таким образом класс называется *подклассом, дочерним классом* или *классом-потомком*. Классы, на основе которых создаётся подкласс, называются *суперклассами, родительскими классами* или *классами-предками*. Принцип наследования обеспечивает преемственность между родительскими и дочерними классами, т.е. «передачу и сохранение» структур данных и подпрограмм родительских классов в дочерних. Кроме того, подкласс может дополнять наследуемые структуры своими и модифицировать (замещать или перегружать) операции над данными. Метод *замещает* или переопределяет метод родительского класса, если он объявлен в дочернем классе с тем же самым протоколом. *Перегрузка* метода представляет собой определение в подклассе одноименного метода одного

из родительских классов с другим набором (или типами) параметров и/или типом возвращаемого значения.

Один и тот же класс может быть потомком для одних классов и предком для других. Теоретически у класса может быть любое количество предков, как непосредственно являющихся для него родительскими классами, так и тех, которые являются его «прародителями». При этом подкласс наследует структуру исходного родительского класса (или классов) и её модификации со стороны всех промежуточных предков.

С целью поддержки принципов инкапсуляции и наследования в большинстве объектно-ориентированных языков так или иначе реализованы средства ограничения доступа к компонентам класса. Для этого во многих языках используются спецификаторы или модификаторы доступа public–protected–private, возможно с некоторыми дополнениями или другими именами. Спецификатор `public` делает соответствующие элементы доступными извне класса. В частности, таким образом определяется внешний интерфейс его объектов. Модификатор `protected` ограничивает область использования компонентов описанием класса и его подклассов любого уровня. Таким образом обычно специфицируется большинство полей данных. Модификатор `private` используется для тех элементов, которые должны быть доступны только в описании этого класса и нигде больше. Такие компоненты обычно отражают специфические особенности класса, которые не должны быть «видны» извне класса, в том числе не должны передаваться по наследству.

Принцип *полиморфизма* связан с концепцией наследования. Он позволяет задавать для родственных объектов одинаковые наборы операций, которые могут быть реализованы по-разному в зависимости от класса объекта, но обращение к которым может быть унифицированным для всех экземпляров классов семейства. Полиморфизм поддерживается путем разрешения определения полиморфных переменных типа родительского класса, которые также могут ссылаться на объекты любых классов-потомков этого класса. Таким образом, если при объявлении переменной в качестве типа указан некоторый класс, то её значениями могут быть ссылки, как на объекты этого класса, так и на объекты любого дочернего класса. Вызов через такую переменную метода, определенного в родительском классе и замещенного в подклассе, приведет к динамическому связыванию с методом в соответствующем дочернем классе. В результате будет выполнен метод, описанный в классе объекта, на который ссылается переменная, а не в классе, который был задан при объявлении переменной.

С концепциями наследования и полиморфизма связаны понятия *абстрактного* (*виртуального*) *метода и класса*. Абстрактным называется метод, для которого в классе не определена реализация, т.е. не задан код, который должен выполняться при его вызове. Класс, в описание которого входят такие методы, в том числе и наследуемые от родительских классов,

называется абстрактным. Абстрактные методы используются при построении иерархий классов для обеспечения полиморфизма. В этом случае в классах верхних уровней вводятся абстрактные методы, которые затем реализуются в подклассах.

Лекция № 10.

ТЕМА: Функциональное программирование. Основы языков LISP и Scheme.

Основные вопросы, рассматриваемые на лекции:

1. Основные концепции функционального программирования.
2. Основы синтаксиса языка LISP.
3. Концептуальная схема работы LISP-интерпретатора.
4. Функции quote и eval.
5. Особенности языка Scheme.
6. Функции языка Scheme для работы со списками и предикатами.
7. Функции языка Scheme для определения новых функций.
8. Функции языка Scheme для организации разветвляющегося вычислительного процесса.

Функциональное программирование основывается на понятии математической функции. Чистый функциональный язык программирования не использует ни переменных, ни операторов присваивания. Отсутствие переменных делает невозможным использование итеративных конструкций, поскольку они управляются переменными. Повторение должно выполняться только с помощью рекурсии. Программы представляют собой определения функций и спецификаций применения функций, а выполнение заключается в вычислении применений функции. Выполнение функции при одних и тех же параметрах всегда приводит к одному и тому же результату. Функциональный язык содержит набор элементарных функций, набор функциональных форм для построения сложных функций из этих элементарных функций, операцию применения функции и некоторые структуры данных, используемые для представления параметров и значений, вычисленных функциями.

Первым функциональным языком был язык LISP. Синтаксическими элементами языка LISP являются *символьные выражения*, которые также называют *s-выражениями*. В виде s-выражений представляются и программы, и данные. S-выражение может быть атомом или списком. Атомы — это базовые синтаксические единицы языка, включающие числа и символы. Символьные атомы, называемые также просто символами, со-

стоят из букв, цифр и некоторых специальных символов. Список — это последовательность атомов или других списков, разделенных пробелами и заключенных в круглые скобки.

Вызовы функций в языке LISP записываются в виде списка в префиксной форме, называемой польской записью, а именно:

(<функция> <аргумент1> <аргумент2> … <аргумент*N*>)

Интерпретатор языка LISP работает в цикле, называемом *циклом чтения-оценки-печати*. Это означает, что интерпретатор выводит приглашение, считывает введенные пользователем данные, пытается их оценить, выводит полученный результат или сообщение об ошибке и снова переходит в режим ожидания ввода. Получив список, интерпретатор LISP пытается проанализировать его первый элемент как имя функции, а остальные элементы — как её аргументы. Выводимое значение является результатом применения этой функции к её аргументам. Оценивая функцию, LISP сначала оценивает её аргументы, а затем применяет функцию, задаваемую первым элементом выражения, к оценке аргументов. Если аргументы сами являются функциями, то LISP рекурсивно применяет это правило для их оценивания. Таким образом, LISP допускает вложенность вызовов функций произвольной глубины. По умолчанию оцениваются все объекты. При этом используются соглашение, что числа соответствуют сами себе. С символами могут быть связаны выражения. Связывание символов может осуществляться в результате вызова функции. При оценивании связанных символов возвращается результат связывания. Если символ не связан, то при его оценке выдается сообщение об ошибке.

Для предотвращения оценивания символа или списка используется специальная функция `quote`. Это функция зависит от одного аргумента и возвращает его без оценки. Функция `quote` используется для того, чтобы аргументы обрабатывались как данные, а не как оцениваемые выражения. Допускается сокращенное обозначение функции в виде символа `'` (одинарной кавычки) непосредственно перед аргументом. Например, выражения `(quote (a b c))` и `'(a b c)` эквивалентны и возвращают список `(a b c)`.

В LISP также существует функция `eval`, позволяющая оценить переданное в качестве аргумента s-выражение. Аргумент оценивается как обычный аргумент функции, однако полученный результат снова оценивается, и в качестве значения выражения `eval` возвращается окончательный результат. Например, в результате применения функции

```
(eval (quote (* 3 7)))
```

будет получено число 21, поскольку в данном случае `eval` отменяет результат выполнения функции `quote`.

Основы языка программирования Scheme.

Язык Scheme является одним из диалектов языка LISP и относительно невелик. В нем используется статический обзор данных, а функции об-

рабатываются как сущности первого класса. Последнее означает, что функции в языке Scheme могут быть значениями выражений и элементами списков, а также могут присваиваться переменным и передаваться как параметры. Параметры функций в языке Scheme передаются по значению, так что независимо от того, что именно функция делает со своими аргументами, фактические параметры не изменяют своего значения. Кроме того, передача по значению подразумевает, что фактические аргументы вычисляются перед применением функции.

Имена в языке Scheme могут состоять из букв, цифр и специальных символов, за исключением скобок. Имена не зависят от регистра символов и не должны начинаться с цифры. Язык Scheme содержит элементарные функции для выполнения основных арифметических операций, а именно +, -, * и / соответственно для сложения, вычитания, умножения и деления.

Поскольку основной структурой данных в языке LISP/Scheme являются списки, язык включает различные функции для работы с ними. В частности, имеются функции для создания списков, выбора их частей и манипулирования ими. Для выбора элементов из списка в языке LISP/Scheme могут использоваться функции `car` и `cdr` (произносится как «куд-ер»). Функция `car` возвращает первый элемент заданного списка (называемый «головой» списка), а функция `cdr` — список (называемый «хвостом»), получаемый из заданного списка после удаления его первого элемента. Например:

Выражение	Результат
`(car '(3 2))`	`3`
`(cdr '(3 2))`	`(2)`
`(car '((1 2) 3 4))`	`(1 2)`
`(cdr '((1 2) 3 4))`	`(3 4)`
`(car '(3))`	`3`
`(cdr '(3))`	`()`

С функциями `car` и `cdr` в языке LISP/Scheme связано понятие «пара» («точечная пара»). «Пара» представляет собой двухэлементную структуру, первый элемент которой (car-поле или «голова») возвращается функцией `car`, а второй (cdr-поле или «хвост») функцией `cdr`. Для представления «пары», «голова» которой есть `h`, а «хвост» есть `t`, обычно используется точечная нотация `(h . t)`.

Для конструирования списка может использоваться двухаргументная функция `cons`. Она создает «пару», «голова» которой представляет собой первый параметр, а «хвост» соответствует второму аргументу. Для формирования с помощью функции `cons` списка элементов необходимо, чтобы её вторым параметром был какой-либо список. Функция `list` создает список, элементами которого являются переданные ей параметры, число которых может быть произвольным. Функция `length` возвращает количество элементов списка. Например:

Выражение	Результат
`(cons 'a '(2 c))`	`(a 2 c)`
`(cons '(a 2) 'c)`	`((a 2) . c)`
`(cons '(a 2) '(c))`	`((a 2) c)`
`(cons '(a 2) '())`	`((a 2))`
`(list 'a '(2 c))`	`(a (2 c))`
`(list '(a 2) 'c)`	`((a 2) c)`
`(list '(a 2) '(c))`	`((a 2) (c))`
`(list '(a 2) '())`	`((a 2) ())`
`(length '((a 2) c (4 e (f))))`	3

В языке LISP/Scheme существуют специальные функции, называемые *предикатными* или *предикатами*. Предикат — это функция, возвращающая логическое значение («истина» или «ложь») в зависимости от того, удовлетворяют ли её параметры некоторому условию или свойству. В языке Scheme значение «истина» обозначается как `#t`, а «ложь» как `#f`. К предикатным функциям для числовых данных в языке Scheme относятся: = (равно), <> (не равно), > (больше), < (меньше), >= (больше или равно), <= (меньше или равно), even? (четное ли число), odd? (нечетное ли число), zero? (равно ли число нулю).

Функция `eq?` позволяет проверить, эквиваленты ли между собой два символьных параметра. Если неизвестно, являются ли атомы символьными или числовыми, то проверить их на равенство можно с помощью предиката `eqv?`. Определить, эквивалентны ли два параметра, каждый из которых может являться атомом или списком, позволяет предикат `equal?`. Например:

Выражение	Результат
`(eq? 'a 'a)`	`#t`
`(eq? 'a 'abc)`	`#f`
`(eqv? 'a 1)`	`#f`
`(eqv? 1 1)`	`#t`
`(equal? 'a '())`	`#f`
`(equal? '() '())`	`#t`
`(equal? '(a (2 c) (3)) '(a (2 c) (3)))`	`#t`
`(equal? '(a (2 c) (3)) '(a (2 c) 3))`	`#f`

Предикат `list?` позволяет проверить, является ли её параметр списком. Функция `null?` возвращает `#t`, если её параметр представляет собой пустой список. В противном случае она возвращает `#f`. Предикат `pair?` проверяет, является ли её аргумент «парой». Наконец, функции `number?` и `symbol?` позволяют определить, представляет ли их аргумент собой соответственно числовой или символьный атом. Например:

Выражение	Результат
`(list? '(a 2 c))`	`#t`
`(list? 'a)`	`#f`
`(list? '())`	`#t`
`(null? '(a 2 c))`	`#f`

(null? 'a)	#f
(null? '())	#t
(pair? '(a 2 c))	#t
(pair? 'a)	#f
(pair? '())	#f
(pair? (cons 'a 2))	#t
(number? 1.23)	#t
(number? 'a)	#f
(symbol? 'ab3)	#t
(symbol? 3)	#f

Функция let позволяет временно связать имена со значениями s-выражений. Эти имена затем могут использоваться при вычислении других выражений в контексте данной функции. Она имеет следующий общий вид:

```
(let (
       (<имя1> <выражение1>)
       (<имя2> <выражение2>)
       ...
       (<имяN> <выражениеN>)
      )
      <тело>
)
```

Каждое s-выражение связывается с указанным именем, после чего выполняются выражения, образующие тело функции. Значение последнего из них представляет собой результат функции let. Все связанные имена имеют областью действия только тело функции.

Для определения новых функций в языке LISP/Scheme используется система лямбда-обозначений в виде списка, который имеет следующий общий синтаксис:

```
(lambda <параметры> <тело>)
```

где <параметры> — имена одного или нескольких формальных параметров функции (лямбда-выражения), <тело> — набор s-выражений, представляющих тело функции, результат вычисления последнего из которых определяет результат применения функции. Формальные параметры могут быть заданы в виде одного символьного атома или их списка. В обоих случаях не следует предварять описание параметров кавычками, т.е. предотвращать их оценивание функцией quote. Если определение лямбда-выражения содержит один формальный параметр, то это означает, что соответствующая функция может иметь любое количество фактических параметров, однако при вызове функции они преобразуются в список, который связывается с указанным аргументом. Если определение функции включает список параметров, то при её вызове должно быть передано указанное число аргументов, которые связываются с соответствующими формальными параметрами. Связанные с формальными параметрами значение

не изменяются в процессе вычисления функции, что отличает их от параметров подпрограмм в императивных языках программирования.

Лямбда-выражение определяет безымянную функцию, которая может применяться так же, как именованная функция, т.е. путем размещения её в начале списка, включающего фактические параметры. Например:

Выражение	Результат
`((lambda (x y) (* (+ x y) 0.01)) 26 53)`	`0.79`
`((lambda (x y) (list (+ x y) (* x y))) 26 53)`	`(79 1378)`
`((lambda v (list (cdr v) (car v))) 'a 2 'c)`	`((2 c) a)`

Для определения именованной функции в языке Scheme используется функция `define`. В простейшей форме она позволяет связать s-выражение с именем и имеет следующий общий синтаксис:

```
(define <имя> <выражение>)
```

где <имя> — название новой функции, <выражение> — s-выражение, связываемое с данной функцией, результат вычисления которого определяет её значение. Например:

Выражение	Результат
`(define a 'alpha)` `a`	`alpha`
`(define tail cdr)` `(tail '(1 2 3 4))`	`(2 3 4)`
`(define mult10 (lambda (x) (* x 10)))` `(mult10 3)`	`30`

Определение именованной функции с параметрами задается с помощью функции `define` следующим образом:

```
(define (<имя> <параметры>) <тело>)
```

где <параметры> — набор разделенных пробелами имен формальных параметров, <тело> — набор s-выражений, представляющих тело функции, результат вычисления последнего из которых определяет результат применения функции. Семантически использование функции `define` в данном виде приводит к связыванию заданного имени с лямбда-выражением с указанными параметрами и телом. Например:

Выражение	Результат

`(define (parabola a b c x)`
`  (+ (* a x x) (* b x) c))`
`(parabola 1 2 3 -1)`
	`2`

`(define (swap12 lst)`
`  (let (`
`  (h1 (car lst))`
`  (h2 (car (cdr lst)))`
`  (t (cdr (cdr lst)))`
`  )`
` (cons h2 (cons h1 t))
	`(2 a c 4 e)`

```
    )
)
(swap12 '(a 2 c 4 e))
```

Для организации ветвления вычислений в языке LISP/Scheme можно использовать функции `if` и `cond`. Работа этих функций зависит от значений соответствующих условий. Следует учитывать, что в языке Scheme только значение `#f` рассматривается условными функциями как «ложь». Все остальные значения, включая символы, числа, списки (в том числе пустой) и пары, интерпретируются как «истина». Функция `if` имеет следующий вид:

```
(if <условие> <выражение> [<альтернатива>])
```

Если результат вычисления условия функции `if` соответствует значению «истина», то оценивается заданное выражение и возвращается результат этой оценки. В противном случае возвращается результат оценки третьего аргумента, если он задан, а когда альтернатива не указана, значение функции считается неопределенным. Общий синтаксис функции `cond` приведен ниже:

```
(cond
    (<условие1> <выражение> {<выражение>})
    (<условие2> <выражение> {<выражение>})
    ...
    (<условиеN> <выражение> {<выражение>})
    [(else <выражение> {<выражение>})]
)
```

Условия функции `cond` вычисляются по одному друг за другом до тех пор, пока какое-либо из них не примет значение «истина». В этом случае вычисляются соответствующие этому условию выражения и результат последнего из них возвращается в качестве значения функции. Если все условия оказались ложными, то при наличии условия `else` оцениваются его выражения, а если его нет, то значение функции считается неопределенным. Рассмотрим примеры использования условных функций.

Выражение	Результат

`(define (factorial n)`
`  (if (<= n 0)`
`  1`
`  (* n (factorial (- n 1))) )`
`)`
`(factorial 5)`
	120

`(define (sum v)`
`  (cond`
`  ((number? v) v)`
`  ((null? v) 0)`
`  ((list? v) (+ (sum (car v)) (sum (cdr v))))`
`  (else 0) )`
	14

```
)
(sum '(4 2 7 1))
```

Поскольку данные и программы имеют одинаковую структуру, пользовательские функции могут создавать программы «на лету» и немедленно выполнять их с помощью функции `eval`. Например, рассмотренную выше функцию `sum` можно определить следующим образом (при условии, что параметр-список не содержит вложенных списков):

```
(define (sum2 v)
  (cond
    ((number? v) v)
    ((null? v) 0)
    ((list? v) (eval (cons '+ v)))
    (else 0) )
)
```

Ниже приведены примеры других функций на языке Scheme с необходимыми комментариями.

```
; Складывает два списка как вектора, т.е. первый элемент
; результата есть сумма первых элементов аргументов,
; второй элемент - сумма вторых элементов аргументов и т.д.
; Если оба аргумента являются числами, возвращает их сумму
; как число.
(define (lists-sum list1 list2)
  (cond
    ; Если оба аргумента являются числами, возвращаем их
    ; сумму
    ((and (number? list1) (number? list2)) (+ list1 list2))
    ; Если первый аргумент является числом, а второй пустым
    ; списком, возвращаем первый аргумент
    ((and (number? list1) (null? list2)) list1)
    ; Если первый аргумент является числом, а второй списком,
    ; возвращаем этот список, добавив к первому элементу
    ; первый аргумент
    ((and (number? list1) (list? list2)) (cons (lists-sum ⇒
list1 (car list2)) (cdr list2)))
    ; Если первый аргумент является числом, а второй не
    ; является списком, возвращаем первый аргумент
    ((and (number? list1) (not (list? list2))) list1)
    ; Если второй аргумент является числом, то выполняем те
    ; же действия, что и для случая, когда первый аргумент
    ; является числом
    ((and (number? list2) (null? list1)) list2)
    ((and (number? list2) (list? list1)) (cons (lists-sum ⇒
list2 (car list1)) (cdr list1)))
    ((and (number? list2) (not (list? list1))) list2)
    ; Если оба аргумента являются пустыми списками,
    ; возвращаем пустой список
    ((and (null? list1) (null? list2)) '())
    ; Если один аргумент является пустым списком, а другой
```

```
    ; списком, возвращаем этот список
    ((and (null? list1) (list? list2)) list2)
    ((and (null? list2) (list? list1)) list1)
    ; Если оба аргумента являются списками, конструируем
    ; список, первый элемент которого есть сумма
    ; первых элементов аргументов, а "хвост" есть сумма
    ; "хвостов" аргументов.
    ((and (list? list1) (list? list2))
       (cons (lists-sum (car list1) (car list2))
             (lists-sum (cdr list1) (cdr list2))))
     ; Иначе возвращаем пустой список
     (else '())
  )
)

; Функциональная форма конструкция.
; Применяет каждую функцию из заданного списка аргументов
; к указанному аргументу и возвращает полученный список
; значений. Для применения функций используется функция eval.
; Параметры:
;      flst - список имен функций или атом, представляющий имя
;             функции;
;      v - значения, к которому применяется каждая функция.
; Возвращаемый результат:
;      Список, каждый элемент которого представляет собой
;      результат применения соответствующей по порядку
;      функции из списка flst к значению, заданному в
;      параметре v.
(define (construction flst v)
  (let (
        ; Выражение, отменяющее оценку параметра v
        (qv (list 'quote v))
       )
    ; Анализируем список имен функций
    (cond
      ; Если список имен функций пуст, результат – пустой
      ; список
      ((null? flst) '())
      ; Если список имен функций задан, в качестве результата
      ; конструируем список, "голова" которого есть результат
      ; применения первой функции из списка, а "хвост"
      ; результат рекурсивного вызова от "хвоста" списка
      ; функций
      ((list? flst) (cons (eval (list (car flst) qv)) ⇒ (con-
struction (cdr flst) v)))
      ; Если задан не список, а одно имя функции, в качестве
      ; результата конструируем список из одного элемента,
      ; соответствующего значению, полученному после
      ; применения данной функции
      ((symbol? flst) (list (eval (list flst qv))))
      ; Иначе результат - пустой список
```

```
      (else '())
      )
  )
)


; Преобразует исходный список таким образом,
; что положительные числа располагаются в начале
; результирующего списка, а остальные атомы - в конце.
(define (positive-head v)
  ; Анализируем аргумент
  (cond
    ; Если аргумент пустой список, возвращаем его
    ((null? v) '())
    ; Если аргумент (непустой) список, выделяем, анализируем
    ; и обрабатываем его "голову" и "хвост"
    ((list? v)
        (let (
              ; Объявляем необходимые локальные имена
              (h (car v))
              (t (cdr v))
              (tres (positive-head (cdr v)))
              )
          ; Анализ "головы"
          (cond
              ; Если "голова" - пустой список, возвращаем
              ; результат рекурсивного вызова функции
              ; от "хвоста"
              ((null? h) tres)
              ; Если "голова" - (непустой) список, а "хвост"
              ; - пустой список, возвращаем результат
              ; рекурсивного вызова функции от "головы"
              ((and (list? h) (null? t)) (positive-head h))
              ; Если "голова" - список и "хвост" - (непустой)
              ; список, возвращаем результат рекурсивного
              ; вызова функции от "разложенной на составляющие
              ; головы" и "хвоста"
              ((list? h) (positive-head (list (car h) ⇒
 (cdr h) t)))
              ; Если "голова" - положительное число,
              ; возвращаем список с данной "головой"
              ; и результатом рекурсивного вызова функции
              ; от "хвоста"
              ((and (number? h) (> h 0)) (cons h tres))
              ; Если "голова" - неположительное число или
              ; атом, анализируем результат рекурсивного
              ; вызова функции от "хвоста"
              (else
                  ; Если обработанный "хвост" пуст,
                  ; возвращаем список, включающий только
                  ; "голову".
```

```
                    ; Иначе, анализируем его "голову".
                    ; Если это положительное число,
                    ; "меняем головы местами" и вызываем
                    ; рекурсивно функцию от нового "хвоста",
                    ; иначе возвращаем список, включающий
                    ; исходную "голову" и обработанный "хвост".
                    (if (null? tres)
                        (list h)
                        (let (
                              (htr (car tres))
                              (ttr (cdr tres))
                              )
                          (if (and (number? htr) (> htr 0))
                              (cons htr (positive-head (cons h ⇒
ttr)))
                              (cons h tres))
                         )
                     )
                )
            )
          )
     )
    ; Если аргумент является атомом, возвращаем его в виде
    ; списка
    ((not (list? v)) (list v))
    ; Иначе возвращаем пустой список
    (else '())
  )
)


; Выделяет положительные числа из списка.
; Возвращает двухэлементный список, первый элемент которого –
; список найденных положительных чисел,
; а второй - список остальных атомов.
(define (extract-positives v)
  ; Анализируем аргумент
  (cond
    ; Если аргумент число, проверяем, является ли оно
    ; положительным и включаем его в соответствующий подсписок
    ((number? v) (if (> v 0) (list (list v) '()) (list '()⇒
(list v))))
    ; Если аргумент пустой список, возвращаем пустые
    ; подсписки
    ((null? v) (list '() '()))
    ; Если аргумент (непустой) список, выделяем, анализируем
    ; и обрабатываем его "голову" и "хвост"
    ((list? v)
        (let (
              ; Объявляем необходимые локальные имена
              (h (car v))
```

```
                (t (cdr v))
                (tres (extract-positives (cdr v)))
                )
            ; Анализ "головы"
            (cond
                ; Если "голова" - пустой список, возвращаем
                ; результат рекурсивного вызова функции от
                ; "хвоста"
                ((null? h) tres)
                ; Если "голова" - (непустой) список, а "хвост"
                ; - пустой список, возвращаем результат
                ; рекурсивного вызова функции от "головы"
                ((and (list? h) (null? t)) ⇒
 (extract-positives h))
                ; Если "голова" - список и "хвост" - (непустой)
                ; список, возвращаем результат рекурсивного
                ; вызова функции от "разложенной на составляющие
                ; головы" и "хвоста"
                ((list? h) (extract-positives (list (car h) ⇒
(cdr h) t)))
                ; Если "голова" - положительное число, включаем
                ; его в первый подсписок вместе с "головой"
                ; обработанного "хвоста", а во второй подсписок
                ; "хвост обработанного хвоста"
                ((and (number? h) (> h 0)) (list (cons h ⇒
 (car tres)) (car (cdr tres))))
                ; Если "голова" - отрицательное число или атом,
                ; включаем его во второй подсписок вместе
                ; с "хвостом обработанного хвоста",
                ; а в качестве первого подсписка берем
                ; "голову обработанного хвоста"
                (else (list (car tres) (cons h (car ⇒
 (cdr tres)))))
            )
        )
    )
    ; Если аргумент является нечисловым атомом, включаем его
    ; во второй подсписок
    ((not (list? v)) (list '() (list v)))
    ; Иначе возвращаем пустые подсписки
    (else (list '() '()))
  )
)


; Выделяет список всех атомов исходного списка.
(define (atom-list v)
  ; Анализируем аргумент
  (cond
    ; Если аргумент пустой список, возвращаем его
```

```
    ((null? v) '())
    ; Если аргумент (непустой) список, выделяем, анализируем
    ; и обрабатываем его "голову" и "хвост"
    ((list? v)
        (let (
              ; Объявляем необходимые локальные имена
              (h (car v))
              (t (cdr v))
              (tres (atom-list (cdr v)))
              )
          ; Анализ "головы"
          (cond
              ; Если "голова" - пустой список, возвращаем
              ; результат рекурсивного вызова функции
              ; от "хвоста"
              ((null? h) tres)
              ; Если "голова" - (непустой) список, а "хвост"
              ; - пустой список, возвращаем результат
              ; рекурсивного вызова функции от "головы"
              ((and (list? h) (null? t)) (atom-list h))
              ; Если "голова" - список и "хвост" - (непустой)
              ; список, возвращаем результат рекурсивного
              ; вызова функции от "разложенной на составляющие
              ; головы" и "хвоста"
              ((list? h) (atom-list (list (car h) (cdr h) t)))
              ; Если "голова" - атом, возвращаем список,
              ; включающий его и обработанный "хвост"
              (else (cons h tres))
          )
        )
     )
    ; Если аргумент является атомом, возвращаем его в виде
    ; списка
    ((not (list? v)) (list v))
    ; Иначе возвращаем пустой список
    (else '())
  )
)


; Преобразует исходный список таким образом,
; что положительные числа располагаются в начале
; результирующего списка, а остальные атомы - в конце.
; В данной версии используются функции extract-positives и
; atom-list.
(define (positive-head-v2 v)
  ; Анализируем аргумент
  (cond
    ; Если аргумент пустой список, возвращаем его
    ((null? v) '())
    ; Если аргумент (непустой) список, выделяем в нем
```

```
    ; положительные числа и остальные атомы
    ; и объединяем полученные списки в один
    ((list? v) (atom-list (extract-positives v)))
    ; Если аргумент является атомом, возвращаем его в виде
    ; списка
    ((not (list? v)) (list v))
    ; Иначе возвращаем пустой список
    (else '())
  )
)
```

Лекция № 11.

ТЕМА: Логическое программирование. Основы языка Prolog.

Основные вопросы, рассматриваемые на лекции:

1. Основные понятия логического программирования.
2. Резолюция. Понятия унификации, конкретизации, бэктрекинга.
3. Основные особенности логических языков программирования.
4. Синтаксис языка Prolog.
5. Структура программы на языке Prolog. Факты, правила и цели.
6. Особенности процедуры резолюции в языке Prolog.
7. Реализация арифметических вычислений.
8. Средства для работы со списками в языке Prolog.
9. Оператор отсечения.

В основе логического программирования лежит математический аппарат формальной логики, в частности исчисление (логика) предикатов. *Предикат* представляет отношение между некоторым (в том числе нулевым) числом объектов предметной области и их свойствами. Объекты предметной области и их свойства обозначаются с помощью *термов*. Терм исчисления предикатов может быть константой, переменной или функциональным выражением. Константа — это символ, представляющий некий объект или его свойство. Переменная — это символ, который может соответствовать разным объектам или свойствам в разное время. Функциональное выражение (*составной терм*) — это символ-идентификатор функции, за которым в круглых скобках перечислены разделенные запятыми термы, представляющие её аргументы или параметры. Идентификатор функции называют также *функтором*. Составные термы соответствуют *атомарным высказываниям или предложениям* в исчислении предикатов. Атомарные предложения с помощью логических операторов $\neg$ (отрицание), $\wedge$ (конъюнкция), $\vee$ (дизъюнкция), $\equiv$ (тождествен-

ность/эквивалентность) и → (импликация) могут комбинироваться в *предложения*. Переменная, встречающаяся в предложении, представляет неопределенный объект, т.е. может быть замещена любой константой. В исчислении предикатов переменные должны быть связаны одним из двух кванторов: универсальности/всеобщности (∀) и существования (∃), которые ограничивают значение предложения, содержащего переменную. Квантор универсальности означает, что предложение истинно для всех значений переменной. Квантор существования указывает, что предложение истинно по крайней мере для одного значения из области определения переменной.

Исчисление предикатов обеспечивает основу для логического вывода и доказательства теорем, т.е. возможность выводить новые правильные выражения из набора истинных утверждений. Логический вывод и доказательство теорем представляет собой основу логического программирования. Одним из используемых для этого методов является резолюция. *Резолюция* — это правило логического вывода, позволяющее получать выводимые высказывания по заданным высказываниям. Для использования принципа резолюции логические высказывания должны быть приведены к *дизъюнктивной форме*, имеющей следующий общий вид: $A_1 \wedge A_2 \wedge \ldots \wedge A_m \rightarrow B_1 \vee B_2 \vee \ldots \vee B_n$ или, сокращенно, $A \rightarrow B$
где A_i и B_j — термы. Левая часть формы называется предпосылкой или антецедентом (соответствующие термы предпосылками), а правая часть — следствием, заключением или консеквентом (соответствующие термы следствиями). Смысл логического высказывания, приведенного к дизъюнктивной форме, заключается в следующем: если истинны все предпосылки, то истинно по крайней мере одно из следствий. Ограниченный вид дизъюнктивных форм, называемый *хорновскими дизъюнктами* или *дизъюнктами Хорна*, позволяет упростить процесс логического вывода на основе резолюции. В хорновском дизъюнкте заключение либо отсутствует (хорновский дизъюнкт без головы), либо содержит единственный терм (хорновский дизъюнкт с головой).

Концепция резолюции заключается в следующем. Предположим, что заданы два следующих высказывания: $A \rightarrow B$ и $B \rightarrow C$. Логически из этого следует, что $A \rightarrow C$. Процесс вывода этого высказывания из двух исходных представляет собой резолюцию и в упрощенном виде выглядит следующим образом. Образуется новое высказывание, левая часть которого есть конъюнкция левых частей исходных высказываний, а правая часть — конъюнкция их правых частей, т.е. $(A \wedge B) \rightarrow (B \wedge C)$. Затем термы, появляющиеся в обеих частях нового высказывания, удаляются из него, в результате чего получается выводимое высказывание, т.е. $A \rightarrow C$. Наличие переменных в высказываниях приводит к необходимости выполнять в процессе резолюции поиск значений переменных, позволяющих установить соответствие между термами. Алгоритм определения необходимых подстановок значений переменных с целью приведения в соответствие двух выражений

исчисления предикатов называется *унификацией*. Временное присваивание значений переменным с целью унификации называется *конкретизацией*. Обычно во время резолюции переменная конкретизируется неким значением, не полностью удовлетворяющим требуемому соответствию. В этом случае следует отменить последнее действие, чтобы конкретизировать эту переменную новым значением. Этот процесс возврата к предыдущему состоянию называется *бэктрекингом*.

Языки логического программирования называются *декларативными языками*, поскольку написанные на них программы состоят из объявлений, соответствующих логическим высказываниям, а не из операторов присваивания и управляющих операторов. Программирование на таких языках является *непроцедурным* в том смысле, что программы не содержат указаний, как именно вычислить результат, а лишь описывают его форму (что нужно найти). В основном в языках логического программирования описание задачи представляется с помощью исчисления предикатов, а средством вывода является метод резолюции.

Еще одна особенность логических языков программирования обусловлена их семантикой, называемой *декларативной семантикой*. Основная её концепция состоит в том, что существует простой способ определения смысла каждого оператора, вне зависимости от того, как именно этот оператор используется для решения задачи. Другими словами, смысл заданного высказывания в языке логического программирования можно точно определить по самому оператору и для этого не требуется рассмотрения контекста или последовательности выполнения программы. Поэтому декларативная семантика проще, чем семантика императивных языков, в которых для определения смысла и результата обработки оператора (например, присваивания или цикла) обычно необходимо учитывать контекст, в котором он выполняется.

Основы языка программирования Prolog.

Язык Prolog является первым и наиболее известным на сегодняшний день языком логического программирования. Все операторы в языке Prolog образуются из *термов*. Как и в исчислении предикатов, терм может быть константой, переменной или структурой, составленной из других термов. Константа — это атом или целое число. Атом представляет собой символьную строку, состоящую из латинских букв, цифр и символов подчеркивания, начинающуюся со строчной буквы, либо строку любых символов, заключенных в апострофы. Переменная — это любая строка букв, цифр и символов подчеркивания, начинающаяся с прописной буквы. Переменные не связываются ни с какими типами с помощью объявлений. Связывание переменной со значением и типом называется *конкретизацией* и происходит только в процессе резолюции. Конкретизации осуществляются только для того, чтобы удовлетворить некую цель, представляющую собой доказательство или опровержение некоторого высказывания. *Структура* соот-

ветствует атомарному высказыванию исчисления предикатов и имеет ту же форму: функтор(терм$_1$, …, терм$_n$). Функтор может быть любым атомом и служит для идентификации структуры. В языке Prolog структуры используются для формулирования фактов и правил. Структура устанавливает некоторое отношение между соответствующими термами. В то же время, когда структура рассматривается как запрос, она представляет собой предикат.

Программа на языке Prolog состоит из набора операторов, образующих базу данных, на основе которой логически может быть выведена новая информация. Существуют две основные формы операторов, соответствующие хорновским дизъюнктам без головы и с головой. Признаком окончания оператора служит символ точки. Простейшая форма хорновского дизъюнкта без головы в языке Prolog представляет собой отдельную структуру, интерпретируемую как факт, то есть высказывание, которое предполагается истинным. Например:

```
male(john).
male(bill).
male(jake).
female(mary).
female(shelley).
father(john, mary).
father(bill, jake).
father(bill, shelley).
mother(mary, jake).
mother(mary, shelley).
```

Оператор языка Prolog, соответствующий хорновскому дизъюнкту с головой, имеет следующий общий вид:

<структура$_0$> :- <структура$_1$>, …, <структура$_n$>

где <структура$_0$> представляет собой заключение, а <структура$_1$>, …, <структура$_n$> — предпосылки. Разделяющая предпосылки запятая соответствует оператору конъюнкции. Высказывания могут содержать переменные. При этом подразумевается, что переменные неявно связаны с квантором всеобщности. Хорновские дизъюнкты с головой называются *правилами*, поскольку они определяют правила логического следствия между высказываниями. Данный оператор интерпретируется как правило «если-то», а именно: если предпосылка является истинной или она может быть сделана истинной путем некоторой конкретизации её переменных, то следствие считается истинным. Допускается описание правил, имеющих одно и то же следствие, но разные наборы предпосылок. В этом случае предполагается неявное использование оператора дизъюнкции, то есть следствие считается истинным, если истинен хотя бы один набор предпосылок. Ниже приведены примеры правил:

```
parent(X, Y) :- father(X, Y).
parent(X, Y) :- mother(X, Y).
grandparent(X, Y) :- parent(X, P), parent(P, Y).
```

```
sibling(X, Y) :- father(F, X), father(F, Y), mother(M, X),
mother(M, Y).
```

Факты и правила представляют сведения, необходимые для решения задачи, однако не указывают, что требуется получить. Для этого служат высказывания, называемые *целями* или *запросами*. В языке Prolog цель имеет синтаксическую форму, эквивалентную форме хорновского дизъюнкта без головы. Определение цели соответствует формулировке теоремы, которую система должна доказать или опровергнуть на основе заданных фактов и правил. На любой корректный запрос система реализации языка Prolog должна дать ответ “yes” («да») или “no” («нет»). Ответ «да» означает, что система доказала, что цель была истинной при заданных фактах и правилах. Ответ «нет» указывает, что либо было доказано, что цель ложна, либо система неспособна ни доказать, ни опровергнуть её на основе имеющихся сведений. В качестве целей могут выступать высказывания в форме конъюнкции и высказывания, включающие переменные. При наличии переменных система пытается найти их конкретизации, делающие цель истинной. Обычно в режиме интерактивной работы соответствующие значения переменных выводятся Prolog-системой на экран в диалоговом режиме. Примеры целей и полученных результатов для рассмотренных ранее фактов и правил:

Цель	Результат
`male(mary).`	`No`
`grandparent(john, shelley).`	`Yes`
`female(X).`	`X = mary ; X = shelley ; No`

`sibling(X, Y).`	`X = jake Y = jake ;`
	`X = jake Y = shelley ;`
	`X = shelley Y = jake ;`
	`X = shelley Y = shelley ; No`

`male(X), sibling(X, shelley).`	`X = jake ; No`

Если цель включает в себя конъюнкцию фактов или структур (как в последнем примере), то они называются *подцелями*. Для доказательства того, что цель/подцель истинна, в процессе логического вывода должна быть найдена цепочка правил логического вывода и/или факты, которые связывают цель/подцель с одним или несколькими фактами в базе данных. Например, если Q — цель, то она либо должна быть найдена как факт в базе данных, либо процесс логического вывода должен найти последовательность высказываний P_1, P_2, P_3, …, P_n, такую что: P_2 :- P_1, P_3 :- P_2, …, Q :- P_n. Эта задача усложняется тем, что правила могут иметь составные правые части или переменные. Процесс поиска фактов P_i, если они существуют, в основном сводится к сравнению или поиску соответствия между термами и обычно называется *сопоставлением* или *удовлетворением*.

Prolog-система всегда выполняет поиск в базе данных в направлении от первого оператора к последнему и удовлетворение подцелей слева направо. Наличие переменных в высказываниях приводит к необходимости использования унификации для установления соответствия между фактами.

При обработке цели с несколькими подцелями, может возникнуть ситуация, когда система не способна доказать истинность одной из подцелей. В этом случае выполняется *бэктрекинг*, то есть система восстанавливает предшествующее состояние цели, заново рассматривает предыдущую (ранее доказанную) подцель, если она есть, и пытается найти её альтернативное решение. Множественность решений для подцели обусловлена наличием различных конкретизаций её переменных. Новое решение находится в результате поиска, предпринятого с того места, где остановился предыдущий поиск для этой подцели. Бэктрекинг требует больших затрат времени и объема памяти, поскольку он может найти все возможные решения для каждой подцели.

Рассмотрим следующую цель: `male(X), sibling(X, shelley)`. Система языка Prolog сначала ищет в базе данных первый факт с функтором `male`. Затем она конкретизирует переменную `X` параметром найденного факта, например, параметром `john`. Далее она пытается доказать, что высказывание `sibling(john, shelley)` является истинным. Если это не удается сделать, то система возвращается к первой подцели и пробует снова её удовлетворить с помощью некоторой альтернативной конкретизации переменной `X`. Может оказаться, что выполняя резолюцию, система должна будет найти каждого мужчину в базе данных, прежде чем она найдет одного из них, являющегося братом Шелли. Кроме того, для того чтобы доказать, что цель не может быть удовлетворена, надо перебрать все возможные конкретизации переменной `X`. В данном примере поиск можно сделать более эффективным, если поменять местами подцели. Тогда только после того, как система с помощью резолюции найдет брата или сестру Шелли, она попытается доказать подцель `male(X)`.

Для выполнения арифметических вычислений в языке Prolog может использоваться оператор `is`, имеющий следующий синтаксис:

```
<Переменная> is <арифметическое выражение>
```

Этот дизъюнкт является истинным, если указанную переменную удалось унифицировать значением заданного выражения. Все переменные в выражении должны быть предварительно конкретизированы, а переменная в левой части оператора не должна конкретизироваться заранее. Поэтому оператор вида `A is A + B` не допустим. Рассмотрим пример правила, позволяющего вычислить стоимость указанного наименования товара, если известны его цена и количество. Данные о ценах и количестве представлены в форме фактов.

```
price(apples, 40).
```

```
price(bread, 12.51).
price(milk, 14.30).
price(rice, 27).
price(jam, 41.88).
qty(apples, 2.5).
qty(bread, 2).
qty(milk, 2).
qty(rice, 4).
qty(jam, 2).
cost(X, Y) :- price(X, P), qty(X, Q), Y is P * Q.
```

Язык Prolog поддерживает списки в качестве базовой структуры данных. Список представляет собой заключенную в квадратные скобки последовательность из любого количества элементов, разделенных запятыми. В качестве элементов списка могут выступать термы или другие списки. Пустой список обозначается следующим образом: `[]`. Обработка списков в Prolog основывается на унификации и рекурсии. Обращение к элементам списка осуществляется посредством специальной формы записи: `[H | T]`, где `H` — представляет «голову» (начальные элементы) списка, а `T` — его «хвост» (список остальных элементов, если они есть). Эта форма используется для реализации операций по работе со списками и может обозначать как создание списка, так и его разделение на части. Для представления в базе данных программы каких-либо сведений в виде списка может использоваться обычная структура, в которой один или несколько термов заменены списками. Например:

```
capitals([moscow, london, paris, berlin, prague]).
```

Эта структура связывает список констант `[moscow, london, paris, berlin, prague]` с именем отношения `capitals`, а не с именем соответствующей переменной. Поэтому допустимо формирование в программе еще одного высказывания с тем же именем, например: `capitals([rome, madrid, athens])`. Это приведет к тому, что запросы вида `capitals(X)` или `capitals([H|T])` будут удовлетворяться двумя значениями.

Реализации языка Prolog обычно включают встроенные или библиотечные предикаты (например, `member` и `append`), обеспечивающие выполнение основных операций обработки списков, аналогичных функциям языка LISP. Отличия описаний этих предикатов от LISP-функций заключается в том, что в Prolog-программе не нужно указывать, как следует создать новый список, а надо лишь уточнить его характеристики в терминах заданных списков. Для получения нового списка в процессе резолюции Prolog-системой используется некоторый вид рекурсии.

Предикат `member` позволяет определить, принадлежит ли указанный элемент заданному списку. Он описывается следующим образом:

```
member(Element, [Element | _]).
member(Element, [_ | List]) :- member(Element, List).
```

Символ подчеркивания обозначает анонимную переменную. Эта переменная служит для указания того, что её связывание не является частью процесса вычислений и нам безразлично, что её конкретизация может быть получена путем унификации. Поскольку система языка Prolog сопоставляет высказывания по порядку, начиная с первого, условие завершения размещается до рекурсивного высказывания.

Предикат `append` применяется для объединения двух списков в третий и имеет следующий вид:

```
append([], List, List).
append([H | List1], List2, [H | List3]) :- append(List1,
List2, List3).
```

Предикат `reverse` изменяет порядок следования элементов заданного списка на противоположный и унифицирует полученный результат с элементами второго списка. Он может быть определен следующим образом:

```
reverse([], []).
reverse([H | T], List) :- reverse(T, Result), append(Result,
[H], List).
```

В отличие от функций языка LISP, многие предикаты языка Prolog, в том числе приведенные выше, благодаря унификации могут быть использованы не только по своему прямому назначению, но и для выполнения связанных или производных операций. Например, предикат `append` позволяет найти список, который надо добавить к указанному, чтобы получить заданный список.

Цель	Результат
`append([a, b], Y, [a, b, c, d]).`	`Y = [c, d] ; No`

`append(X, Y, [a, b, c, d]).`	`X = [] Y = [a, b, c, d] ;`
	`X = [a] Y = [b, c, d] ;`
	`X = [a, b] Y = [c, d] ;`
	`X = [a, b, c] Y = [d] ;`
	`X = [a, b, c, d] Y = [] ; No`

`reverse(X, [a, b, c, d]).`	`X = [d, c, b, a] ; No`

Помимо того, что пользователь может управлять базой данных и порядком удовлетворения подцелей, язык Prolog позволяет использовать некоторые явные средства управления бэктрекингом. Это осуществляется с помощью *оператора отсечения*, обозначаемого знаком восклицания (!). В качестве подцели этот оператор всегда достигается немедленно, но он не может быть удовлетворен повторно с помощью бэктрекинга. Таким образом, побочный эффект оператора отсечения заключается в том, что подцели, расположенные левее его в составной цели, не могут быть удовлетворены снова с помощью бэктрекинга. Поэтому, если подцели, расположенные справа от оператора отсечения, не удалось удовлетворить, то вся цель считается ложной. Например, в случае цели `a, b, !, c, d`, если под-

цели a и b достигаются, а подцель c — нет, то вся цель не достигается. В данном случае оператор отсечения может использоваться для того, чтобы указать, что если подцель c не достигается в первый раз, то она никогда не достигается, а значит пытаться вновь удовлетворять подцели a или b — пустая трата времени. Таким образом, назначение оператора отсечения состоит в том, чтобы позволить пользователю сделать программу более эффективной, сообщив системе, когда не следует пытаться повторно удовлетворять подцели, которые предположительно не могут дать результата в завершенном доказательстве. В результате, если найденное множество значений не приводит к решению, то поиск решения не продолжается.

Ниже приведены примеры некоторых предикатов для работы со списками.

```
% Меняет местами первый и второй элементы списка.
swap12([], []).
swap12([X], [X]).
swap12([X, Y | T], [Y, X | T]).

% Меняет местами нечетные и четные элементы списка.
% Например, результатом обработки списка [a, b, c, d, e]
% будет список [b, a, d, c, e].
swap_odd_even([], []).
swap_odd_even([X], [X]).
swap_odd_even([X, Y | T], [Y, X | R]) :- swap_odd_even(T, R).

% Расщепляет заданный список L на список положительных чисел P
% и список остальных атомов A.
extract_positives([], [], []).
extract_positives([X | T], [X | P], A) :- number(X), X > 0, ⇒
extract_positives(T, P, A).
extract_positives([X | T], P, [X | A]) :- number(X), X =< 0, ⇒
extract_positives(T, P, A).
extract_positives([X | T], P, [X | A]) :- not(number(X)), ⇒
extract_positives(T, P, A).

% Преобразует исходный список таким образом, что положительные
% числа находятся в начале результирующего списка,
% а остальные атомы - в конце.
positive_head([], []).
positive_head([X], [X]).
positive_head([X, Y | T], R) :- ⇒
extract_positives([X, Y | T],  P, A), append(P, A, R).

% Возвращает список R, включающий не более чем N первых
% элементов исходного списка L.
list_beginning([], _, []).
list_beginning([_ | _], 0, []).
list_beginning([X | _], 1, [X]).
```

```
list_beginning([X | T], N, [X | R]) :- N > 1, M is N - 1, ⇒
list_beginning(T, M, R).

% Возвращает список R, включающий не более чем N последних
% элементов исходного списка L
% в следующем порядке: последний, предпоследний и т.д.
% Например, если L = [a, b, c, d, e], N = 3, то R = [e, d, c].
% Используемый предикат last позволяет определить последний
% элемент списка.
list_ending([], _, []).
list_ending([_ | _], 0, []).
list_ending([X], 1, [X]).
list_ending([_ | T], 1, [X]) :- last(T, X).
list_ending([X | T], N, R) :- N > 1, reverse([X | T], L), ⇒
list_beginning(L, N, R).

% Возвращает список с номерами позиций, на которых расположен
% элемент E в списке L.
% Нумерация элементов начинается с 1.
% Например, если E = c, L = [a, b, c, c, b, a, c],
% то R = [3, 4, 7].
elem_positions(E, L, R) :- elem_positions(E, L, 0, R).

% В elem_positions/4 3-й параметр используется в качестве
% счетчика просмотренных элементов исходного списка L
% и определяет номер текущего элемента. Таким образом, его
% начальное значение определяет способ нумерация элементов,
% т.е., например, для значения -1 элементы списка нумеруются
% 0, 1, 2, ..., а для значения 0 - 1, 2, 3, ...
elem_positions(_, [], _, []).
elem_positions(X, [X], N, [M]) :- M is N + 1, !.
elem_positions(_, [_], _, []) :- !.
elem_positions(X, [X | T], N, [M | R]) :- M is N + 1, ⇒ el-
em_positions(X, T, M, R), !.
elem_positions(X, [_ | T], N, R) :- M is N + 1, ⇒ el-
em_positions(X, T, M, R), !.

% Подсчитывает количество вхождений элемента E в список L.
elem_occur(_, [], 0).
elem_occur(X, [X | T], M) :- ⇒
elem_occur(X, T, N), M is N + 1, !.
elem_occur(X, [_ | T], N) :- elem_occur(X, T, N), !.
```

Лекция № 12.

ТЕМА: «Стековое» программирование. Основы языка Forth (Форт).

Основные вопросы, рассматриваемые на лекции:

1. Ключевые компоненты Форт-системы.
2. Особенности синтаксиса языка.
3. Слова для работы со стеком.
4. Определение новых слов.

Ключевыми компонентами Форт-системы, определяющими основные свойства и возможности языка, являются словарь и отдельный стек данных. *Словарь Форта* представляет собой совокупность слов, с каждым из которых связан некоторый фрагмент исполняемого кода и которые можно рассматривать как специфические подпрограммы. Словарь отражает текущее состояние языка, т.е. набор команд, из которых может быть составлена программа. Каждое слово реализует какую-либо операцию и может использовать для этих целей другие слова, определенные в словаре ранее. Любой фрагмент текста Форт-система пытается интерпретировать как слово из своего словаря. В процессе разработки программы словарь может быть дополнен специфическими и необходимыми для решения задачи словами. Новые слова обычно определяются на основе уже существующих и в дальнейшем, в свою очередь, могут использоваться при описании других слов. Это связано с тем, что все слова в Форт-системе совершенно равноправны. Обычно транслятор Форта не устанавливает различий между своими «родными» словами и словами, добавленными программистом. Поэтому существующие в базовом языке слова при необходимости могут быть переопределены программистом и получить иной смысл.

Стек Форт-системы представляет собой основную структуру данных, с которой работают слова языка. Стек предназначен для хранения чисел, характеризуется последовательным выполнением операций записи и чтения и организован по принципу «последним пришел, первым ушел» (LIFO). С помощью стека выполняется обмен данными между словами, в нем размещаются исходные данные и сохраняются результаты вычислений. При этом никакого перемещения информации не происходит, изменяется только содержимое указателя стека. В Форт существует возможность работы с именованными переменными, как в других языках, но для хранения чисел и для передачи их из одного слова в другое, в основном используется стек. Даже обращение с переменными во многом основано на оперировании со стеком.

Синтаксис языка Форт достаточно необычен по сравнению с большинством других языков. Исходный текст программы представляет собой последовательно обрабатываемый Форт-транслятором набор слов и чисел, разделенных пробелами или переводами строки. Все числа, встречающиеся в тексте программы, в конечном итоге попадают в стек. Слова обрабатываются в зависимости от текущего режима: либо ищутся в словаре и непосредственно выполняются, либо компилируются и «записываются» в словарь. Управление режимом работы транслятора Форта также осуществляется с помощью слов. Они могут рекурсивно вызывать транслятор, подключать другие словари для поиска слов, временно переключать входной поток на другой источник и выделять последующие слова как часть своих параметров. Благодаря этим возможностям в язык вводятся элементы привычных синтаксических конструкций, например ветвления и циклов, которые также представляют собой слова. Если в режиме интерпретации слово не найдено в словаре и не опознано как число, то Форт прерывает трансляцию, выполняет очистку стека и выдает сообщение об ошибке.

Одной из особенностей синтаксиса языка Форт является использование *постфиксной или обратной польской нотации*. Согласно ей символ операции (оператор) записывается после операндов. Например: `a b +`. Одним из важных достоинств постфиксной нотации является то, что она не нуждается в скобках, при этом порядок действий определяется порядком следования операторов. Постфиксная нотация тесно связана с использованием стека, куда записываются как исходные данные, так и результаты. Например, для выполнения операции перемножения двух чисел в языке Форт можно использовать следующий синтаксис:

```
5 2 *
```

В данном случае «*» представляет собой специальное слово. При выполнении этого примера в стек будут последовательно занесены числа 5 и 2, после чего выполнится слово «*», которое выберет их из стека, перемножит и занесет в стек полученный результат. Для того чтобы вывести полученный результат, необходимо использовать слово «.» (точка), которое выбирает (изымает) число, находящееся на вершине стека, и выводит его на экран. Таким образом, выполнение строки

```
5 2 * .
```

приведет к выводу на экран числа 10. При постфиксной нотации возможны операции типа `a b c * +`. При этом результат будет равен `a + (b * c)`.

Во многих случаях для корректного выполнения операции требуется изменить порядок следования или состав элементов стека. Для этих целей используются специальные слова, позволяющие осуществить необходимые преобразования с содержимым стека. Далее представлены некоторые из них.

- SWAP — меняет местами два верхних элемента стека.

- DUP — дублирует верхний элемент стека.
- OVER — копирует 2-й элемент стека и заносит его наверх.
- ROT — переносит последний элемент стека наверх.
- DROP — удаляет верхний элемент стека.
- DEPTH — помещает в вершину стека количество элементов, находившихся в нем до выполнения этого слова.

Например, для того чтобы вычислить `(a - b) * c`, когда в исходном состоянии стек содержит `a`, `b`, `c`, необходимо изменить порядок элементов на `c`, `a`, `b`. Для получения требуемого порядка элементов в стеке можно выполнить набор слов `SWAP ROT SWAP`. В результате, чтобы вычислить указанное выражение для исходного состава элементов стека потребуется выполнить следующую последовательность слов:

```
SWAP ROT SWAP - *
```

Таким образом, при работе со стеком необходимо учитывать его текущее состояние. При попытке записать в стек слишком большое количество чисел или извлечь что-то из «пустого» стека возникнет сообщение об ошибке.

Для работы со словарем, его модификации и дополнения в Форте предусмотрены специальные слова. Добавление новых и переопределение существующих слов в словаре осуществляется с помощью команды «:» (двоеточие) и имеет следующий синтаксис:

```
:  <ОПРЕДЕЛЯЕМОЕ_СЛОВО>  [<СУЩЕСТВУЮЩИЕ СЛОВА>] ;
```

Действия, связанные с определяемым словом, записываются через пробел за его именем и представляют собой разделенные пробелами слова, которые уже имеются в словаре. Определение завершается точкой с запятой, которая также является специальным словом. Поэтому она должна отделяться от предшествующих слов пробелом. Определенное слово немедленно становится равноправным членом словаря и может быть использовано в создании новых определений. Если переопределяется уже существующее слово, то все последующие обращения к нему будут использовать именно это последнее определение. Однако ранее созданные слова будут по-прежнему использовать старую версию, поскольку они были скомпилированы на её основе.

Рассмотрим, например, определение слова, позволяющего возвести в квадрат число, находящееся на вершине стека:

```
: SQUARE DUP * ;
```

В соответствии с ним будет продублировано число, находящееся на вершине стека. В результате в начале стека будут находится два одинаковых числа, соответствующих исходной вершине стека. После этого они будут выбраны из стека словом «*», перемножены и результат будет размещен на вершине стека. Сразу после определения слово `SQUARE` можно использовать в программе. Например,

```
7 SQUARE .
```

выведет на экран число 49. А следующее определение на базе только что описанного слова SQUARE позволит вычислять уже куб числа:

```
: CUBE DUP SQUARE * ;
```

Таким образом, программирование решения задачи на языке Форт представляет собой составления описаний новых слов на базе ранее заданных, пока не будет определено главное слово, т.е. то слово, которое нужно ввести, чтобы выполнить программу и получить конечные результаты.

Еще одна особенность языка Форт заключается в том, что он дает возможность программисту определить некоторые слова на языке ассемблера, когда требуется максимальное быстродействие. Базовая Форт-система содержит множество слов, которые определены не с помощью других слов, а непосредственно на машинном языке. К числу таких слов относятся, например, +, *, DUP, SWAP.

Примеры программ на Forth

```
Программа перевода английских единиц измерения в дюймы (1 ярд=
36 дюймам и 1 фут = 12 дюймам).

: ярдов 36 * ;
: футов 12 * ;
: ярд  ярдов ;
: фут  футов ;
: дюйм 1 *
1 ярд 5 футов + 1 дюйм + .       (результат 97)
6 ярдов 1 фут + .                (результат 228)

Другой пример. Проверить, не превышает ли температура лабора-
торного бойлера допустимого значения. Значение температуры
нужно получить в стеке.

: ?жарко  220 > if ."Опасный перегрев" ELSE ."Норма" THEN ;

290 ?жарко          (результат – сообщение "Опасный перегрев")
130 ?жарко          (результат – сообщение "Норма")

Примеры циклов
: ДЕКАДА 10 0 DO I . LOOP         (арифметический цикл)
ДЕКАДА              (результат – цифры: 0 1 2 3 4 5 6 7 8 9)

: ТЕСТ 0 Begin 1 + ."привет" 3 > UNTIL (цикл с постусловием)
ТЕСТ                (результат – четыре сообщения "привет" )
```

Лекция № 13.

ТЕМА: Программирование обработки баз данных. Основы языка SQL.

Основные вопросы, рассматриваемые на лекции:

1. Введение в языки программирования баз данных.
2. Основы синтаксиса языка SQL. Синтаксис оператора SELECT.
3. Объединение данных из нескольких таблиц. Ключевое слово JOIN.
4. Предложение WHERE. Предикаты для отбора записей.
5. Агрегирующие функции и группировка данных. Отбор групп.
6. Подзапросы и предикаты для работы с ними.
7. Объединение результатов запросов с помощью конструкции UNION.
8. Изменение содержимого таблицы. Операторы INSERT, UPDATE, DELETE.

Язык программирования баз данных (ЯПБД) обычно включает в себя два подъязыка: *язык определения данных*, используемый для определения схемы (структуры) базы данных (БД), и *язык манипулирования данными*, предназначенный для работы с данными, хранимыми в базе. Как правило, в этих языках отсутствуют конструкции управления вычислительным процессом, такие как условные операторы или операторы цикла. Для устранения этого ограничения многие системы управления базами данных (СУБД) предоставляют возможность внедрения операторов ЯПБД в тот или иной язык программирования высокого уровня, например C++, Pascal, Java, FoxPro, называемый базовым языком. Перед компиляцией программы на базовом языке, включающей операторы ЯПБД, они транслируются в конструкции базового языка, обеспечивающие вызов соответствующих подпрограмм СУБД. Кроме того, большинство СУБД и сред разработки приложений для работы с БД предоставляют инструментальные средства для интерактивного выполнения операторов ЯПБД.

SQL представляет собой стандартизированный и наиболее распространенный на данный момент ЯПБД для реляционных СУБД. Он включает в себя как язык определения данных, так и язык манипулирования данными. Оператор языка состоит из зарезервированных (ключевых) слов, слов, определяемых пользователем, литералов и знаков операций, скомпонованных в соответствии с установленными синтаксическими правилами. Зарезервированные слова являются частью языка SQL и позволяют описать выполняемую операцию. Слова, определяемые пользователем, представляют собой имена различных объектов БД, над которыми надо выполнить операцию. Литералы задают используемые в операторе константы. Все нечисловые константы должны заключаться в одинарные кавычки.

Однако, в некоторых диалектах языка для представления литералов того или иного типа, например дат и времени, могут применяться другие символы. Во многих реализациях SQL каждый оператор должен заканчиваться специальным символом, как правило точкой с запятой.

Язык манипулирования данными SQL включает следующие основные операторы:

- SELECT — для извлечения данных из базы;
- INSERT — для добавления данных в таблицу;
- UPDATE — для изменения данных в таблице;
- DELETE — для удаления данных из таблицы.

Общий формат оператора SELECT имеет следующий вид, где в угловых скобках заданы определяемые пользователем слова-параметры:

```
SELECT [ALL | DISTINCT] [{<табли-
ца>|<псевдоним>}.]{* | <выражение>
        [AS <другое имя столбца>]} [,…]
  FROM <таблица> [<псевдоним>] [,…]
  [WHERE <условие отбора записей>]
  [GROUP BY <группируемый столбец>, [,…]]
  [HAVING <условие отбора групп>]
  [ORDER BY <сортируемый столбец> [ASC | DESC] [,…]]
```

Обязательными в операторе являются только конструкции SELECT и FROM. Ключевое слово ALL указывает на необходимость включения в результирующую выборку всех записей, удовлетворяющих запросу, в том числе повторяющихся, если они есть. Ключевое слово DISTINCT используется для удаления дублирующих строк, то есть в результирующую выборку не будут включаться записи, совпадающие по значениям всех полей с одной из ранее отобранных. Параметр <таблица> представляет собой имя таблицы БД, из которой осуществляется выборка. <выражение> задает имя столбца таблицы или выражение из нескольких имен, определяющее вычисляемое поле, содержимое которого включается в результирующую выборку. Выражение помимо имен столбцов, арифметических операций сложения, вычитания, умножения и деления, а также круглых скобок, используемых в сложных выражениях, может содержать в зависимости от диалекта языка те или иные функции от значений полей. Звездочка (*) вместо имени столбца указывает на необходимость включения всех полей. Название любого столбца в результирующей таблице может быть изменено с помощью параметра <другое имя столбца>, что обычно используется для именования вычисляемых полей. Если данные извлекаются из нескольких таблиц, которые имеют совпадающие по названию столбцы, то имя каждого поля должно предваряться именем или псевдонимом таблицы. Псевдоним задает сокращенное имя для таблицы, используемое в пределах данного оператора. Параметр <условие отбора записей> описывает фильтр, определяющий какие строки

должны быть включены в результат. <группируемый столбец> задает имя поля, по значениям которого производится группировка записей. Параметр <условие отбора групп> представляет фильтр, накладываемый на сформированные группы. Наконец, <сортируемый столбец> указывает название поля, в соответствии со значениями которого должна быть упорядочена формируемая выборка. Результат выполнения оператора представляет собой таблицу, в которой находятся извлеченные из БД сведения.

Далее на примерах приведены различные варианты и особенности применения оператора SELECT.

Пример 1. Вывести список товаров с указанием помимо существующих данных общей стоимости товара, имеющегося на складе. Список отсортировать по убыванию цены, а при равной цене — по названию.

```
SELECT *, price*quantity AS cost
FROM Product
ORDER BY price DESC, name ASC;
```

Результирующая таблица включает в себя все поля из таблицы Product и дополнительное вычисляемое поле, значения которого представляют собой стоимость всех единиц товара, имеющихся на складе. Для этого поля задано имя, которое может быть использовано для обращения к нему в других частях оператора. Конструкция ORDER BY выполняет упорядочивание отобранных данных сначала по убыванию значений поля price, что определяется ключевым словом DESC, а при равных величинах в этом поле — по возрастанию (алфавитному порядку) значений столбца name. Ключевое слово ASC при этом можно опустить, поскольку по умолчанию осуществляется сортировка значений столбца по возрастанию.

Пример 2. Вывести список товаров с указанием наименования типа товара, его названия, наименования поставщика и цены. Список отсортировать по наименованиям типов, а при одинаковых типах — по названиям товаров.

```
SELECT T.name AS type_name, P.name, S.name AS supplier_name,
P.price
FROM Product P JOIN Supplier S ON P.supplier_id = S.id
     JOIN Type T ON P.type_id = T.id
ORDER BY T.name, P.name
```

Тот же результат может быть получен с помощью следующего оператора:

```
SELECT T.name AS type_name, P.name, S.name AS supplier_name,
P.price
FROM Product P, Supplier S, Type T
WHERE P.supplier_id = S.id AND P.type_id = T.id
ORDER BY T.name, P.name
```

В обоих вариантах решения выполняется объединение данных из нескольких таблиц, называемое *внутренним соединением*. Оно позволяет включить в результат только те записи из объединяемых таблиц, которые

удовлетворяют заданным условиям. В первом случае соединение формируется с помощью конструкции JOIN, а во втором — посредством фильтра, заданного в конструкции WHERE. Для устранения неоднозначности относительно того, к какой таблице относится тот или иной столбец, название каждого поля предваряется ссылкой на таблицу. Для сокращения записи применяются псевдонимы, указанные после имени каждой таблицы. Для столбцов, содержащих наименования типов товаров и поставщиков заданы другие названия, чтобы избежать конфликта имен в результирующей выборке.

Для выполнения объединения данных из нескольких таблиц достаточно перечислить их через запятую в предложении FROM. В результате будет сформирована выборка, представляющая собой декартовое произведение таблиц, то есть каждая запись первой таблицы будет объединена с каждой записью второй, затем каждая строка полученного набора будет объединена с каждой строкой третьей таблицы и т.д. Фильтр в конструкции WHERE позволяет сократить получаемое после объединения множество записей, указывая какие из них требуется включить в результирующую таблицу, а поля, перечисляемые в предложении SELECT, определяют её столбцы.

Конструкция JOIN задает порядок и условия объединения данных из нескольких таблиц. Во многих диалектах SQL для выполнения внутреннего соединения вместо ключевого слова JOIN используется связка INNER JOIN. Объединения выполняются последовательно друг за другом, но в каждый промежуточный результат включается не декартовое произведение двух таблиц, а «пересечение» множеств строк, формируемое на основании заданного условия. Другими словами, сначала в двух первых таблицах находятся пары записей (первая запись из одной таблицы, вторая из другой), удовлетворяющих условию соединения. Каждая такая пара объединяется в одну строку, которая включается в промежуточный результат. Затем полученный набор записей тем же способом соединяется с третьей таблицей и т.д. В итоге будет сформирована выборка, содержащая только необходимые строки из всех объединяемых таблиц. Следующий простой пример демонстрирует концепцию работы предложения JOIN:

table1	
f11	**f12**
1	3
2	2
4	5
7	8

table1 t1 JOIN table2 t2 ON t1.f11=t2.f21

table2	
f21	**f22**
1	8
1	4
2	3
3	6

=>

join			
f11	**f12**	**f21**	**f22**
1	3	1	8
1	3	1	4
2	2	2	3

Помимо внутренних с помощью конструкция JOIN можно организовать так называемые *внешние соединения*. При выполнении внешнего соединения в результирующую таблицу включаются также записи, которые не удовлетворяют условиям объединения. Существует три типа внешнего соединения: левое, правое и полное. Общий формат конструкции JOIN имеет следующий вид:

```
<таблица 1> [<псевдоним 1>] [LEFT|RIGHT|FULL] JOIN
  <таблица 2> [<псевдоним 2>] ON <условие объединения 1>
[[LEFT|RIGHT|FULL] JOIN <таблица 3> [<псевдоним 3>]
  ON <условие объединения 2> …]
```

При этом в зависимости от диалекта языка может требоваться, чтобы каждая «внутренняя» связка JOIN … ON … была заключена в круглые скобки, то есть чтобы вся конструкция имела вид:

```
((((t1 JOIN t2 ON c1) JOIN t3 ON c2) JOIN t4 ON c3) … ) JOIN
tn ON cm
```

В ходе выполнения *левого внешнего соединения* двух таблиц каждая запись первой таблицы (расположенной слева от предложения LEFT JOIN) объединяется с одной или несколькими записями второй таблицы (расположенной справа от предложения LEFT JOIN). Если для записи первой таблицы существуют записи второй таблицы, удовлетворяющие заданному условию, то запись первой таблицы объединяется с каждой такой записью второй таблицы и полученные строки включаются в результат. Если для записи из первой таблицы нет подходящей по условию записи из второй таблицы, то она объединяется с «пустой» записью второй таблицы, у которой все поля имеют неопределенное значение NULL, и сформированная строка также включается в результирующую выборку. Например (здесь и далее знаком ? обозначены NULL-значения):

table1	
f11	**f12**
1	3
2	2
4	5
7	8

table1 t1
LEFT JOIN
table2 t2
ON
t1.f11=t2.f21

table2	
f21	**f22**
1	8
1	4
2	3
3	6

=>

join			
f11	**f12**	**f21**	**f22**
1	3	1	8
1	3	1	4
2	2	2	3
4	5	?	?
7	8	?	?

Правое внешнее соединение двух таблиц формируется противоположным образом. Каждая запись второй таблицы (расположенной справа от предложения RIGHT JOIN) объединяется с одной или несколькими записями первой таблицы (расположенной слева от предложения RIGHT JOIN). Если для записи второй таблицы существуют записи первой таблицы, удовлетворяющие заданному условию, то запись второй таблицы объединяется с каждой такой записью первой таблицы и полученные строки включаются в результат. Если для записи из второй таблицы нет подходящей по условию записи из первой таблицы, то она объединяется с «пустой» записью первой таблицы, у которой все поля имеют неопределенное

значение NULL, и сформированная строка также включается в результирующую выборку.

Например:

Table1	
f11	**F12**
1	3
2	2
4	5
7	8

table1 t1
RIGHT
JOIN
table2 t2
ON
t1.f11=t2.f21

table2	
f21	**f22**
1	8
1	4
2	3
3	6

=>

join			
f11	**f12**	**f21**	**f22**
1	3	1	8
1	3	1	4
2	2	2	3
?	?	3	6

При выполнении *полного внешнего соединения* двух таблиц в результат включаются объединения записей, удовлетворяющих условию, а также те записи из обеих таблиц, для которых нет подходящих записей из другой таблицы. Другими словами, если для записи первой таблицы существуют записи второй таблицы, удовлетворяющие заданному условию, то запись первой таблицы объединяется с каждой такой записью второй таблицы и полученные строки включаются в результат. Если запись из первой таблицы не имеет подходящей по условию записи из второй таблицы, то она объединяется с «пустой» записью второй таблицы и добавляется в результирующую выборку. Наконец, записи второй таблицы, для которых отсутствуют удовлетворяющие условию записи первой таблицы, объединяются с «пустыми» записями первой таблицы и также включаются в результат.

Например:

Table1	
f11	**F12**
1	3
2	2
4	5
7	8

table1 t1
FULL JOIN
table2 t2
ON
t1.f11=t2.f21

table2	
f21	**f22**
1	8
1	4
2	3
3	6

=>

Join			
f11	**f12**	**f21**	**f22**
1	3	1	8
1	3	1	4
2	2	2	3
4	5	?	?
7	8	?	?
?	?	3	6

Пример 3. Вывести отсортированный по времени список выполненных за май 2004 г. покупок с указанием имени кассира, времени покупки, номера чека, номера дисконтной карты, а также типа и номера пластиковой карты в тех случаях, когда она была использована.

```
SELECT C.surname + ' ' + C.first_name + ' ' + C.patronymic
       AS cashier_name, P.time, P.check,
       P.discount_card_number AS discount_card, PC.name AS
       plst_card_type,
       P.plastic_card_number AS plst_card
FROM Purchase P LEFT JOIN Cashier C ON P.cashier_id = C.id
     LEFT JOIN Plastic_Card PC ON P.plastic_card_id = PC.id
WHERE '5/1/2004' <= P.time AND P.time <= '5/31/2004'
ORDER BY P.time
```

Для получения требуемого набора столбцов выполняется левое внешнее соединение таблиц Purchase, Cashier и Plastic_Card. Для отбора строк, содержащих сведения о покупках, выполненных за указанный период, в предложении WHERE задано соответствующее условие. Фильтр в предложении WHERE может строиться на основе следующих базовых условий отбора (предикатов):

- Сравнение (операции =, < > или !=, <, >, <=, >=). Результат вычисления одного выражения сравнивается с результатом вычисления другого выражения.
- Неопределенное/пустое значение (IS NULL/IS NOT NULL). Поле проверяется на наличие какого-либо значения.
- Диапазон (BETWEEN/NOT BETWEEN). Результат вычисления выражения проверяется на вхождение в заданный диапазон значений.
- Принадлежность множеству (IN/NOT IN). Результат вычисления выражения проверяется на принадлежность заданному множеству значений.
- Соответствие шаблону (LIKE/NOT LIKE). Результат вычисления строкового выражения проверяется на соответствие заданному шаблону.

Сложные условия отбора могут быть построены из базовых с помощью логических операций AND, OR и NOT, а также скобок, определяющих порядок вычисления выражений.

Пример 4. Вывести список выполненных за 2004 г. покупок, при оплате которых использовались дисконтные или пластиковые карты, и которые были проведены кассирами, родившимися в 1974 или 1975 году и чья фамилия заканчивается на «-нова».

```
SELECT C.surname + ' ' + C.first_name + ' ' + C.patronymic
       AS cashier_name, P.time, P.check,
       P.discount_card_number AS discount_card, PC.name AS
plst_card_type,
       P.plastic_card_number AS plst_card
FROM Purchase P LEFT JOIN Cashier C ON P.cashier_id = C.id
     LEFT JOIN Plastic_Card PC ON P.plastic_card_id = PC.id
WHERE (P.time BETWEEN '1/1/2004' AND '12/31/2004')
      AND (P.discount_card_number IS NOT NULL
           OR P.plastic_card_number IS NOT NULL)
      AND YEAR(C.dob) IN (1974, 1975)
      AND C.surname LIKE '%нова'
ORDER BY P.time
```

Для получения требуемого результата используется сложный фильтр, включающий несколько условий. Предикат BETWEEN задает границы диапазона, в которые должно попадать значение времени покупки (поля time таблицы Purchase), чтобы запись была включена в результат. Существует также противоположная версия предиката (NOT BETWEEN), требующая, чтобы проверяемое значение лежало вне границ заданного диапазона. Предикат IS NOT NULL позволяет отобрать записи, в которых задан номер дисконтной (поле discount_card_number таблицы Purchase) или

пластиковой карты (поле plastic_card_number). Если поле имеет неопределенное (NULL-) значение, то соответствующая запись не будет включена в результат. Проверка на совпадения значения с NULL может быть выполнена с помощью предиката IS NULL. Предикат IN задает множество значений, в которое должен попадать год рождения кассира, чтобы запись была включена в результирующую выборку. Для определения года применяется встроенная во многие СУБД функция YEAR, возвращающая число, представляющее год в дате, переданной ей в качестве аргумента. В данном случае в качестве аргумента функции используется день рождения кассира (поле dob таблицы Cashier). Множество в предикате IN задается в виде заключенного в круглые скобки списка значений через запятую. Если результат вычисления выражения совпадает с одним из значений, представленным в списке, то соответствующая запись включается в выборку. Противоположная версия проверки (NOT IN) применяется с целью отбора строк, для которых результат выражения не входит в заданный список. Предикат LIKE проверяет, оканчивается ли фамилия кассира (поле surname таблицы Cashier) на «-нова». Он задает шаблон, которому должен соответствовать результат вычисления символьного выражения, чтобы запись была включена в выборку. Шаблон может включать специальные символы, определяющие вариации значений. Символ процента (%) обозначает последовательность из нуля или более любых символов. Символ подчеркивания (_) обозначает один любой символ. Все остальные символы в шаблоне представляют сами себя. Если шаблон должен включать в себя символ, интерпретируемый как специальный, то с помощью конструкции ESCAPE следует определить «маскирующий» символ, который указывает, что следующий за ним символ не имеет специального значения, и поместить его в шаблоне перед подстановочным символом. Например, для проверки строки на наличие в ней подстроки ‘10%’, можно использовать следующее условие:

```
LIKE '%10#%%' ESCAPE '#'
```

Пример 5. Определить минимальную, максимальную и среднюю стоимость товара на складе.

```
SELECT MIN(price*quantity) AS min_cost, MAX(price*quantity) AS
max_cost, AVG(price*quantity) AS average_cost
FROM Product
```

Для выполнения этого запроса использованы *агрегирующие функции* языка SQL, чей результат зависит от набора записей, для обработки которого они используются. Существует пять стандартных агрегирующих функций:

- COUNT — возвращает количество записей в обрабатываемом наборе;
- SUM — возвращает сумму значений выражения, вычисленного на обрабатываемом наборе записей;

• MIN — возвращает минимальное из значений выражения, вычисленного на обрабатываемом наборе записей;

• MAX — возвращает максимальное из значений выражения, вычисленного на обрабатываемом наборе записей;

• AVG — возвращает среднее от значений выражения, вычисленного на обрабатываемом наборе записей.

Каждая агрегирующая функция, кроме COUNT, вычисляет выражение, переданное в качестве аргумента, для каждой записи обрабатываемого набора, и на основании полученного множества значений формирует результат. Функции COUNT, MIN и MAX могут использоваться как с числовыми, так и с нечисловыми выражениями, а функции SUM и AVG применимы только для работы с числовыми данными. За исключением варианта COUNT(*), при вычислении результата любой функции из полученного набора значений сначала исключаются пустые (NULL-) значения, после чего требуемая операция выполняется над оставшимися непустыми значениями. Функция COUNT(*) предназначена для подсчета всех строк в обрабатываемом наборе записей, независимо от значений входящих в них полей.

Агрегирующие функции могут использоваться только в списке выражений предложения SELECT и в конструкции HAVING. Наиболее часто эти функции применяются вместе с группировкой записей, выполняемой с помощью предложения GROUP BY. При использовании агрегирующих функций предложение SELECT не может включать ссылки на столбцы вне выражений функций, за исключением случая использования конструкции GROUP BY. Например, следующий запрос является некорректным:

```
SELECT name, MAX(price)
FROM Product
```

Пример 6. Определить количество единиц хлебобулочных изделий, а также молочных, мясных и рыбных продуктов, проданных за октябрь 2004 г.

```
SELECT SUM(I.quantity) AS total_quantity
FROM (Item I JOIN Purchase P ON I.purchase_id = P.id)
     JOIN Product PD ON I.product_id = PD.id
WHERE (PD.type_id IN (2, 1, 6, 7))
      AND (P.time BETWEEN '10/1/2004' AND '10/31/2004')
```

Пример 7. Для каждого кассира, обслуживавшего покупки, подсчитать их количество.

```
SELECT cashier_id, COUNT(*) AS purchase_quantity
FROM Purchase
GROUP BY cashier_id
```

Для получения требуемых результатов необходимо выполнить так называемый *группирующий запрос*, в котором формируются группы строк, совпадающих по значениям некоторых выражений, а затем для каждой полученной группы в соответствии с заданным в конструкции SELECT набо-

ром столбцов создается «итоговая» запись, включаемая в результат. Выражения, определяющие способ группировки, перечисляются через запятую в предложении GROUP BY и называются *группируемыми столбцами*. Любое поле, входящее в список выборки конструкции SELECT, должно иметь единственное значение для каждой группы и может быть задано только константой, именем столбца, агрегирующей функцией или выражением, составленным из этих элементов. Кроме того, если в предложении SELECT указано имя какого-либо столбца, то оно должно также присутствовать и в конструкции GROUP BY (то есть столбец должен быть группируемым), за исключением случаев, когда поле используется только в агрегирующей функции. При выполнении группировки все отсутствующие (NULL-) значения считаются равными. Если две записи таблицы содержат значение NULL в одном из группируемых столбцов и совпадающие значения во всех остальных непустых группируемых столбцах, то такие строки включаются в одну и ту же группу.

Пример 8. Для каждого кассира вывести количество единиц товара каждого типа для покупок, которые он обслужил за первое полугодие 2004 г.

```
SELECT P.cashier_id, PR.type_id, SUM(I.quantity) AS prod-
uct_quantity
FROM Item I JOIN Purchase P ON I.purchase_id = P.id
     JOIN Product PR ON I.product_id = PR.id
WHERE P.time BETWEEN '1/1/2004' AND '30/6/2004'
GROUP BY P.cashier_id, PR.type_id
```

Представленный выше оператор реализует указанные действия. Группировка осуществляется в начале по идентификатору кассира (поле cashier_id таблицы Purchase), а затем по идентификатору типа товара (поле type_id таблицы Product). Следующий пример демонстрирует принцип, в соответствии с которым осуществляется процедура формирования и обработки групп в данном случае:

C	T	Q
2	2	1
1	1	2
1	3	4
2	3	4
2	4	3
2	4	2
1	1	2
1	1	3
2	3	1
2	2	2
1	3	1
2	4	3

Группировка по полю C =>

C	T	Q
1	1	2
1	3	4
1	1	2
1	1	3
1	3	1
2	2	1
2	3	4
2	4	3
2	4	2
2	3	1
2	2	2
2	4	3

Группировка по полю T =>

C	T	Q
1	1	2
1	1	2
1	1	3
1	3	4
1	3	1
2	2	1
2	2	2
2	3	4
2	3	1
2	4	3
2	4	2
2	4	3

Суммирование по полю Q =>

C	T	SUM(Q)
1	1	7
1	3	5
2	2	3
2	3	5
2	4	8

Пример 9. Для каждого кассира вывести те типы продуктов, которые встречались более одного раза в покупках, обслуженных им за первое полугодие 2004 г.

```
SELECT C.surname + ' ' + C.first_name + ' ' + C.patronymic
       AS cashier_name, T.name AS type_name,
       COUNT(I.quantity) AS quantity
FROM Item I JOIN Purchase P ON I.purchase_id = P.id
     JOIN Cashier C ON P.cashier_id = C.id
     JOIN Product PR ON I.product_id = PR.id
     JOIN Type T ON PR.type_id = T.id
WHERE P.time BETWEEN '1/1/2004' AND '30/6/2004'
GROUP BY C.surname + ' ' + C.first_name + ' ' + C.patronymic,
T.name
HAVING COUNT(I.quantity) > 1
```

Данный запрос во многом похож на предыдущий. Для каждого типа подсчитывается сколько раз какие-либо продукты данной категории встречались в покупках и в результат с помощью условия в предложении HAVING включаются только те строки, для которых полученное число больше единицы. Конструкция HAVING предназначена для использования совместно с предложением GROUP BY и позволяет задать ограничения с целью отбора групп, «итоговые» строки для которых включаются в результирующую выборку. Условие отбора в конструкции HAVING формируется в соответствии с синтаксическими правилами, которые во многом похожи на используемые для предложения WHERE. Однако, в конструкции HAVING, в отличии от WHERE, могут использоваться агрегирующие функции. Еще одно отличие состоит в том, что поле, используемое в HAVING, должно либо входить в список столбцов, по которым осуществляется группировка, то есть находиться в списке элементов GROUP BY, либо применяться в агрегирующей функции. На практике условие отбора групп в предложении HAVING всегда включает, как минимум, одну агрегирующую функцию.

Пример 10. Вывести список продуктов с указанием их наименования, цены и поставщика, имеющих минимальную цену среди всех товаров.

```
SELECT P.name AS product, S.name AS supplier, P.price
FROM Product P JOIN Supplier S ON P.supplier_id = S.id
WHERE P.price = (SELECT MIN(price) FROM Product)
```

Для реализации этого запроса используются два оператора SELECT — внешний и внутренний. Внутренний оператор SELECT представляет собой полноценную команду выборки данных, включаемую в тело другого оператора SELECT, INSERT, UPDATE или DELETE, называемого внешним. При использовании SELECT в качестве внешнего оператора, внутренний оператор SELECT называется *подзапросом* или *вложенным запросом*. Подзапросы могут находиться в конструкциях WHERE и HAVING и обычно служат для отбора строк, включаемых в результат. Подзапросы могут использоваться совместно с операторами сравнения, такими как =, < >, <, >, <=, >=, а также со специальными предикатами ALL,

ANY/SOME, IN/NOT IN, EXISTS/NOT EXISTS. В любом случае, вложенный запрос должен быть указан после оператора или предиката. Существует три варианта применения подзапроса:

• <выраже ие> <оператор сравнения> [ALL|ANY|SOME] (<подзапрос>) — для сравнения результата вычисления выражения с одним или несколькими значениями, возвращаемым подзапросом; предикат ALL позволяет отобрать только те записи, для которых результат выражения удовлетворяет сравнению со всеми значениями, полученными подзапросом; предикаты ANY/SOME предназначены для отбора записей, для которых результат выражения удовлетворяет сравнению хотя бы с одним значением, возвращенным подзапросом; если в результате подзапроса будет получено пустое (NULL-) значение, то для предиката ALL условие сравнения будет считаться выполненным, а для предикатов ANY/SOME — невыполненным;

• <выражение> [NOT] IN (<подзапрос>) — для проверки вхождения или отсутствия (NOT) результата вычисления выражения в список значений, возвращаемый подзапросом;

• [NOT] EXISTS (<подзапрос>) — для определения того, возвращает или нет (NOT) подзапрос какие-либо записи.

При работе с подзапросами следует учитывать следующие особенности.

• Во вложенных запросах нельзя использовать конструкцию ORDER BY.

• Список выборки в предложении SELECT подзапроса должен состоять из имен отдельных полей или составленных из них выражений, кроме случая, когда он применяется совместно с предикатом EXISTS/NOT EXISTS.

• В подзапросе можно использовать ссылки на столбцы таблиц, которые перечисляются в конструкции FROM внешнего запроса, для чего применяются уточненные имена, состоящие из псевдонима таблицы и имени поля.

• Если подзапрос возвращает единственное значение, то он может быть включен в список отбираемых полей предложения SELECT внешнего запроса.

• Подзапрос, в свою очередь, может содержать вложенные запросы, влияющие на возвращаемые им результаты.

В данном примере подзапрос возвращает единственное значение, представляющее собой минимальную среди всех товаров цену, которое позволяет во внешнем запросе отобрать требуемые продукты с помощью операции сравнения на равенство. Для того чтобы продемонстрировать возможность использования подзапросов в конструкции SELECT внешнего запроса, модифицируем приведенный оператор с целью вывода списка

товаров, имеющих цену выше средней относительно всех продуктов, с указанием средней цены и разницы между ней и ценой товара. В этом случае оператор примет следующий вид:

```
SELECT P.name AS product, S.name AS supplier, P.price,
       (SELECT AVG(price) FROM Product) AS avg_price,
       P.price - (SELECT AVG(price) FROM Product) AS differ-
ence
FROM Product P JOIN Supplier S ON P.supplier_id = S.id
WHERE P.price > (SELECT AVG(price) FROM Product)
```

Пример 11. Вывести список продуктов с указанием их названия, типа и стоимости на складе, чья стоимость меньше любой средней стоимости товаров других категорий.

```
SELECT P.name AS product, T.name AS type, P.price*P.quantity
AS cost
FROM Product P JOIN Type T ON P.type_id = T.id
WHERE P.price*P.quantity < ALL (SELECT AVG(price*quantity)
                                FROM Product
                                WHERE type_id <> P.type_id
                                GROUP BY type_id)
```

Пример 12. Вывести список продуктов с указанием их названия, типа и стоимости на складе, чья стоимость больше хотя бы одной средней стоимости товаров других категорий.

```
SELECT P.name AS product, T.name AS type, P.price*P.quantity
AS cost
FROM Product P JOIN Type T ON P.type_id = T.id
WHERE P.price*P.quantity > SOME (SELECT AVG(price*quantity)
                                 FROM Product
                                 WHERE type_id <> P.type_id
                                 GROUP BY type_id)
```

Пример 13. Вывести список всех товаров для каждого поставщика, один из продуктов которого имеют минимальную цену.

```
SELECT P.name AS product, S.name AS supplier, P.price
FROM Product P JOIN Supplier S ON P.supplier_id = S.id
WHERE P.supplier_id IN (SELECT supplier_id
                        FROM Product
                        WHERE price = (SELECT MIN(price) FROM
Product))
```

Пример 14. Вывести список продуктов с указанием наименования, поставщика, типа и цены, для каждого из которых есть хотя бы один другой продукт, совпадающий по цене.

```
SELECT P.name, S.name AS supplier_name, T.name AS type_name,
P.price
FROM (Product P JOIN Supplier S ON P.supplier_id = S.id)
     JOIN Type T ON P.type_id = T.id
WHERE EXISTS (SELECT * FROM Product WHERE price = P.price AND
id <> P.id)
```

Пример 15. Вывести список продуктов с указанием их наименования и цены, которые имеют минимальную, максимальную или цену, отличающуюся от средней не более чем на 5%.

```
SELECT name, price
FROM Product
WHERE price = (SELECT MIN(price) FROM Product)
UNION
SELECT name, price
FROM Product
WHERE price = (SELECT MAX(price) FROM Product)
UNION
SELECT name, price
FROM Product
WHERE price >= (SELECT AVG(price)*0.95 FROM Product)
      AND price <= (SELECT AVG(price)*1.05 FROM Product)
```

В данном случае для получения требуемых результатов с помощью предложения UNION выполняется объединение результатов трех запросов. Соединяемые посредством UNION результирующие таблицы запросов должны иметь одинаковую структуру, то есть одинаковое число столбцов, совместимых по типу и длине с соответствующими полями других таблиц. По умолчанию, при использовании конструкции UNION из получаемого в результате объединения набора записей исключаются дублирующиеся строки. Если необходимо оставить все записи, то после предложения UNION через пробел следует указать ключевое слово ALL.

Операторы языка SQL для изменения содержимого базы данных.

Существует две формы оператора INSERT. Первая предназначена для добавления в указанную таблицу единственной записи и имеет следующий формат:

```
INSERT INTO <имя таблицы> [(<список полей>)]
VALUES (<значения полей >)
```

Параметр <имя таблицы> задает таблицу, в которую необходимо добавить данные. Необязательный параметр <список полей> представляет собой список имен одного или более столбцов, разделенных запятыми. Если он опущен, то предполагается, что используется список из имен всех полей таблицы, указанных в том порядке, в котором они были заданы при её создании. Если в операторе INSERT определен конкретный список столбцов, то любые пропущенные поля должны быть объявлены при создании таблицы как допускающие значение NULL или имеющие некоторое значение по умолчанию, которое будет использовано при добавлении записи. Параметр <значения полей> должен соответствовать параметру <список полей> по количеству и типам.

Пример 16. Добавить сведения о новом кассире в базу данных.

```
INSERT INTO Cashier (surname, first_name, patronymic,
                     dob, hire_date, phone)
VALUES ('Семенова', 'Марина', 'Алексеевна',
```

```
    '7/26/1972', '10/30/2004', '528 1293')
```

Вторая форма оператора INSERT позволяет добавить в таблицу множество записей, полученных как результат запроса. В этом случае оператор имеет следующий вид:

```
INSERT INTO <имя таблицы> [(<список полей>)]
<запрос на выборку данных>
```

Параметры <имя таблицы> и <список полей> имеют тот же смысл и формат, что и для первой формы оператора. <запрос на выборку данных> может представлять собой любой допустимый оператор SELECT. Возвращаемый им набор строк добавляется в указанную таблицу. При этом структура результирующей таблицы вложенного запроса должна соответствовать структуре таблице, в которую осуществляется вставка данных.

Пример 17. Предположим, что создана дополнительная таблица Controller для хранения сведений о кассирах-котроллерах, по структуре совпадающая с таблицей Cashier. Требуется заполнить таблицу Controller, добавив в неё сведения о тех кассирах, которые не провели ни одной покупки.

```
INSERT INTO Controller
SELECT *
FROM Cashier
WHERE id NOT IN (SELECT DISTINCT cashier_id FROM Purchase)
```

Оператор UPDATE позволяет изменить содержимое некоторых записей указанной таблицы. Он имеет следующий формат:

```
UPDATE <имя таблицы>
SET <имя поля 1> = <значение поля 1>
    [, <имя поля 2> = <значение поля 2> …]
[WHERE <условие отбора записей>]
```

Параметр <имя таблицы> определяет таблицу, в которой требуется изменить данные. В конструкции SET задаются имена модифицируемых полей и их новые значения, которые должны быть совместимы с соответствующими столбцами по типу. Необязательная конструкция WHERE позволяет ограничить множество записей таблицы, в которых будут изменены значения указанных полей. Другими словами, модифицируются только те строки, которые удовлетворяют заданному условию отбора записей. Соответствующий параметр аналогичен по формату выражению фильтра в предложении WHERE оператора SELECT. Если конструкция WHERE отсутствует, то оператор UPDATE выполняет изменения во всех строках таблицы.

Пример 18. Повысить цены на все молочные продукты на 1%.

```
UPDATE Product
SET price = price*1.01
WHERE type_id = 1
```

Оператор DELETE позволяет удалить некоторые записи из указанной таблицы. Он имеет следующий формат:

```
DELETE FROM <имя таблицы>
```

```
[WHERE <условие отбора записей>]
```
Параметр <имя таблицы> определяет таблицу, в которой выполняется удаление строк. С помощью необязательной конструкции WHERE задается множество записей, которые подлежат удалению. Другими словами, строка будет удалена из таблицы только в том случае, если она удовлетворяет указанному условию отбора. Соответствующий параметр аналогичен по формату выражению фильтра в предложении WHERE оператора SELECT. Если конструкция WHERE отсутствует, то оператор DELETE удаляет все существующие в таблице записи, но не саму таблицу.

Пример 19. Удалить сведения о кассирах, которые не обслужили ни одной покупки.

```
DELETE FROM Cashier
WHERE id NOT IN (SELECT DISTINCT cashier_id FROM Purchase)
```

Лекция №.14

ТЕМА: Основы языка программирования Fortran.

Основные вопросы, рассматриваемые на лекции:

1. Основные особенности языка Fortran.
2. Структура программы.
3. Базовые типы данных.
4. Описание переменных.
5. Массивы и их сечения.
6. Операторы для работы с массивами.
7. Производные типы.
8. Основные операторы ввода-вывода.
9. Операторы управления ходом вычислительного процесса.

Fortran — один из первых языков программирования высокого уровня. Он был разработан в 1950-х годах в фирме IBM и предназначался для решения вычислительных задач. Fortran получил широкое распространение в научной и инженерной среде как язык, позволяющий создавать эффективные программы для математических расчетов, обработки больших объемов информации и моделирования различных процессов. Это было достигнуто благодаря следующим особенностям языка:

- Высокая эффективность исполняемого кода. Компиляторы языка генерируют быстрый и компактный код. Это связано с хорошей проработанностью алгоритмов компиляторов и относительной простотой конструкций языка.
- Межплатформенная переносимость. Компиляторы языка существуют для большого количества платформ, что позволяет переносить про-

граммы между ними. Переносимость обеспечивается наличием достаточно жестких стандартов языка, которых должны придерживаться разработчики компиляторов.

- Большой набор средств для математических вычислений. Язык содержит множество встроенных математических функций и арифметических операций. На Fortran создано большое количество разнообразных библиотек, в первую очередь для математических и научно-технических расчетов.

Существует несколько основных стандартов языка: Fortran 77 — устаревшая версия, Fortran 90 и его обновление Fortran 95 — версии языка, поддерживаемые большинством существующих компиляторов, Fortran 2003 и его обновление Fortran 2008 — современные версии языка, включающие много нововведений. Представленное далее описание языка соответствует версиям Fortran 90/95.

Исходный код на языке Fortran может быть записан в фиксированном и свободном формате. Фиксированный формат используется в Fortran 77 и более старых версиях языка, а также поддерживается в Fortran 90/95 для обеспечения совместимости. В этом формате строка имеет длину 72 символа, первые пять из которых отведены для меток, шестой служит для ввода признака продолжения строки, а последующие предназначены для ввода оператора. В свободном формате, введенном в Fortran 90, все позиции строки равноправны и её длина составляет 132 символа. Для разделения программной конструкции на несколько строк следует использовать символ “&” (амперсанд), который ставится в конце строки в месте переноса. В этом случае следующая строка считается продолжением данной. В одной строке можно записать несколько операторов, разделенных символом “;” (точкой с запятой). Комментарии начинаются символом “!”, который может находиться в любом месте строки. Идентификаторы могут иметь длину до 31 символа. Fortran является регистронезависимым языком. По соглашению ключевые слова языка записываются в верхнем регистре, а все остальные имена — в нижнем регистре.

Программа на Fortran состоит из главной программы и набора подпрограмм, компилируемых отдельно друг от друга. Подпрограммы могут быть процедурами или функциями. Главная подпрограмма имеет следующую структуру (в квадратных скобках приведены необязательные элементы):

```
[PROGRAM <имя_программы>]
[<раздел описаний>]
[<исполняемые операторы>]
[CONTAINS
    <внутренние подпрограммы>]
END [PROGRAM [<имя_программы>]]
```

Раздел описаний включает операторы описания переменных, констант, меток, подпрограмм и других объектов, используемых в программе. Оператор CONTAINS отделяет исполняемые операторы от описания внутренних подпрограмм. Внутренние подпрограммы могут использоваться только тем программным компонентом, в котором они описаны.

Fortran поддерживает следующие базовые типы данных:

- `INTEGER` — для работы с целыми числами.
- `REAL` — для представления вещественных чисел.
- `COMPLEX` — для работы с комплексными числами.
- `LOGICAL` — для представления логических значений «истина» и «ложь», которые задаются с помощью логических констант `.TRUE.` и `.FALSE.` соответственно.
- `CHARACTER` — для работы со строками.

Диапазон допустимых значений для числовых типов в Fortran не стандартизирован, и программист может определить его самостоятельно. Числовые константы могут быть заданы в двоичной (например, `B'10110'`), восьмеричной (`O'7346'`) или шестнадцатеричной (`Z'def'`) системе счисления. Вещественная константа может быть представлена в формате с фиксированной точкой, включающим необязательный знак, целую часть, десятичную точку и дробную часть. Целая часть или дробная часть с точкой может быть опущена. В формате с плавающей запятой дополнительно используется символ `E` или `D`, обозначающий основание 10, после которого указывается необязательный знак порядка и порядок. Символ `E` соответствует одинарной точности, а `D` — двойной. Например, константа `-75E-2` соответствует числу `-0.75`. Константы комплексных чисел записываются в круглых скобках, например: `(2, 5.8)`. Строковые константы заключаются в одинарные или двойные кавычки. Длина строк не ограничена. Она задается с помощью параметра `LEN` при описании соответствующей переменной. Каждому символу строки соответствует номер. Нумерация символов начинается с 1. Fortran позволяет оперировать частями строк с помощью так называемых сечений. Выражение `str(i:j)` позволяет выделить из строки `str` подстроку, включающую символы с номерами от `i` до `j` включительно. Если одна из границ диапазона не задана, то предполагается, что она совпадает с началом или концом строки соответственно. Сечения позволяют не только выделять части строк, но и изменять их. Следующий пример демонстрирует это:

```
str = "Hello, world!"
str(8:12) = "Sam"
```

В результате переменная `str` будет иметь значение `"Hello, Sam !"` (с двумя пробелами перед восклицательным знаком). Это связано с тем, что длина выражения при необходимости дополняется справа пробелами

до длины переменной, стоящей в левой части присваивания. Если же длина выражения превышает длину переменной, то оно будет урезано до нужной длины. Поэтому присваивание

```
str(8:12) = "Samuel"
```

приведет к тому, что строка `str` будет иметь значение `"Hello, Samue!"`.

В Fortran задавать тип переменной можно двумя способами. Первый заключается в определении типа по имени. Если первым символом имени переменной или функции является буква I, J, K, L, M или N, то по умолчанию предполагается, что переменная имеет (функция возвращает) тип INTEGER, в противном случае предполагается тип REAL. Данный способ описания типа рекомендуется не использовать. Предложение IMPLICIT позволяет изменить правила неявного определения типов. В частности, IMPLICIT NONE отменяет действие правила задания типа по первой букве имени.

Второй способ определения типа переменной связан с использованием предложения описания типов. В этом случае переменные описываются следующим образом:

<тип>[, <список атрибутов>] :: <список переменных>

где атрибуты являются необязательными. Атрибуты и переменные в списках разделяются запятыми. Переменной можно присвоить начальное значение, которое указывается после символа “=”. В описании можно использовать объекты, описанные ранее. В Fortran 90 поддерживаются следующие атрибуты объектов:

- `PARAMETER` — объект является именованной константой;
- `PUBLIC` — объект является доступным за пределами содержащего его модуля;
- `PRIVATE` — объект недоступен за пределами содержащего его модуля;
- `POINTER` — объект является указателем;
- `TARGET` — объект можно использовать в качестве адресата в операторах назначения указателей;
- `DIMENSION` — объект является массивом;
- `ALLOCATABLE` — объект является динамическим массивом;
- `INTENT` —для параметра подпрограммы позволяет указать, является ли он входным (`IN`), выходным (`OUT`) или входным и выходным (`INOUT`);
- `OPTIONAL` — для параметра подпрограммы указывает, что он является необязательным;
- `SAVE` — предписывает сохранять значения локальной переменной подпрограммы в промежутке между её вызовами;
- `EXTERNAL` — для внешней подпрограммы;

- INTRINSIC — для внутренней подпрограммы.

Следующий пример демонстрирует объявление переменных с помощью предложения описания типа:

```
REAL :: v, w = 4.7, h = 2.1
```

Массивы в Fortran характеризуются типом элементов, размерностью (рангом) и формой, определяющей количество элементов (экстент) в каждом измерении массива. Массивы одинаковой формы называются совместимыми. Ранг массива не может быть больше 7. Количество элементов массива и его форму можно определить с помощью встроенных функций SIZE и SHAPE соответственно. Функция SHAPE возвращает одномерный массив, длина которого равна рангу массива-параметра, а элементы представляют собой количество элементов массива-параметра в соответствующем измерении. По умолчанию элементы в любом измерении нумеруются с 1 до количества элементов в этом измерении, однако можно задать другие границы для индексов при объявлении массива следующим образом:

```
[<нижняя_граница>:]<верхняя_граница>
```

Если нижняя граница не задана, то она считается равной 1. Если оказывается, что нижняя граница массива превосходит верхнюю, то длина массива становится нулевой. Форма массива нулевой длины сохраняется, т.е. два массива нулевой длины разных форм несовместимы. Если количество элементов массива неизвестно в момент описания, то границы для каждого измерения задаются в следующей форме:

```
[<нижняя_граница>]:
```

При описании массива можно задать значения для его элементов. Если все элементы должны иметь одно и то же начальное значение, то достаточно указать его за именем массива после символа “=”. Кроме того, задать значения элементам можно с помощью *конструктора массива*, имеющего следующий вид:

```
(/<список значений конструктора>/)
```

Каждое значение из списка может быть либо выражением, либо неявным циклом, представленным в форме

```
(<значение>, <индекс> = <начало>, <конец>[, <шаг>])
```

где <начало>, <конец> и необязательный <шаг> — это выражения целого типа, определяющие нижнюю и верхнюю границы переменной-индекса и шаг его изменения, по умолчанию равный 1. <значение> может быть выражением или неявным циклом, т.е. неявные циклы можно вкладывать друг в друга. Конструктор позволяет создать только одномерный массив. Однако с помощью встроенной функции RESHAPE его можно преобразовать в массив любой формы. Ниже приведены примеры объявления и инициализации массивов.

```
INTEGER, DIMENSION(10) :: list = 1
INTEGER, DIMENSION(-2:2) :: mark = (/ (i, i = -2, 2) /)
```

```
REAL, DIMENSION(3, 5) :: matrix = &
    RESHAPE(SOURCE = (/ ((i*j*0.1, i = 1, 3), j = 1, 5) /), &
            SHAPE = (/3, 5/))
list = (/ (i**2, i = 1, SIZE(list)) /)
```

Для обращения к отдельному элементу массива необходимо использовать его индексы, например:

```
matrix(3, 2) = -0.5
list = (/ (list(SIZE(list) - i + 1), i = 1, SIZE(list)) /)
```

Последнее выражение позволяет изменить порядок следования элементов массива на противоположный.

Следует учитывать, что некоторые операторы и функции обрабатывают элементы массива по порядку, который заключается в том, что индексы перебираются слева направо. Сначала пробегает все свои значения самый левый индекс, затем второй слева и так далее. Например, именно в таком порядке по умолчанию присваиваются значения элементам массива, создаваемого функцией RESHAPE, и выводятся элементы массива оператором PRINT.

Массивы могут выступать в качестве операндов в выражениях числового типа и операциях присваивания. При присваивании массива совместимому массиву присваиваются значения элементов одного массива соответствующим элементам другого массива. При присваивании массиву некоторого скалярного значения оно присваивается каждому элементу массива. Над совместимыми массивами можно выполнять все арифметические операции и логические операции отношения. Результатом арифметических операций с массивами является массив, совместимый исходным, элементы которого получены применением соответствующей операции к элементам массивов-операндов. Например, при перемножении совместимых массивов результатом будет массив, элементы которого представляют собой произведения соответствующих элементов исходных массивов. Результатом логических операций отношения является массив логического типа, элементы которого принимают значения .TRUE. или .FALSE. в зависимости от результата применения данной операции к паре элементов массивов-операндов с одинаковыми индексами. Например, результатом сравнения двух массивов будет массив, элементы которого принимают значение .TRUE., если соответствующие элементы исходных массивов совпадают. Почти все встроенные функции, имеющие формальные аргументы-скаляры, пригодны для обработки массивов, если тип элементов массива совпадает с типом аргумента функции. Например, если a — вещественный массив, то массив логарифмов его элементов можно вычислить следующим образом: LOG(a).

Fortran допускает обращение к частям массива и операции с ними. В качестве индекса можно указать диапазон значений в рамках границ массива. Получаемый в результате объект называется *сечением массива* и его

можно использовать почти так же, как исходный массив. Диапазон отбираемых значений в каждом измерении задается в следующем виде:

```
[<начало>]:[<конец>[:<шаг>]]
```

где <начало>, <конец> и <шаг> — это необязательные выражения целого типа, задающие соответственно индекс нижней и верхней границы сечения, а также шаг, с которым отбираются элементы. Если верхняя или нижняя граница сечения не задана, то предполагается, что эта граница совпадает с соответствующей границей массива в этом измерении. Шаг по умолчанию предполагается равным единице. Нижняя граница может быть больше верхней, а шаг может быть отрицательным. Это позволяет отбирать элементы массива в обратном порядке. Например, выражение `list(9:3:-2)` позволяет выбрать из массива `list` в обратном порядке элементы с нечетными индексами в диапазоне с третьего по девятый включительно. Для обращения к произвольному подмножеству элементов массива можно использовать *векторный индекс* — массив целого типа ранга 1, задающий индексы отбираемых элементов. Например:

```
INTEGER, DIMENSION(4) :: index = (/2, 5, 6, 8/)
list(index) = (/0, 0, 1, 2/)
```

приведет к изменению значений элементов массива `list` с индексами 2, 5, 6 и 8. То же самое можно записать следующим образом, используя конструктор массива для описания сечения:

```
list( (/2, 5, 6, 8/) ) = (/0, 0, 1, 2/)
```

В Fortran существуют операторы WHERE и FORALL, позволяющие выполнить некоторые операции над элементами массива, удовлетворяющими заданным условиям. Оператор WHERE имеет следующий вид:

```
WHERE (<условие>) <переменная> = <выражение>
```

где <условие> представляет собой выражение, результатом которого является массив логических значений (маска), определяющий набор элементов, над которыми должна быть выполнена операция, <переменная> указывает обрабатываемый массив, а <выражение> задает способ изменения его элементов. Изменяются только те элементы массива, которым соответствуют истинные значения маски. Формы маски и обрабатываемого массива должны совпадать. Например, приведенный ниже оператор позволяет заменить значениями экспоненциальной функции те элементы массива, которые лежат в диапазоне от 0.3 до 0.7 включительно:

```
WHERE (0.3 <= matrix .AND. matrix <= 0.7) matrix =
EXP(matrix)
```

Для выполнения нескольких присваиваний применяется конструкция `WHERE…END WHERE`, имеющая вид:

```
WHERE (<условие 1>)
    <операторы присваивания 1>
[ELSEWHERE (<условие 2>)
    <операторы присваивания 2>
```

```
...
ELSEWHERE (<условие N>)
    <операторы присваивания N>
ELSEWHERE
    <операторы присваивания ELSE>]
END WHERE
```

Ветки ELSEWHERE являются необязательными и позволяют организовать альтернативные присвоения. Ниже приведен пример использования данной конструкции (массив p должен иметь ту же форму, что и массив matrix):

```
WHERE (matrix < 0.3)
    matrix = EXP(matrix)*2
    p = matrix
ELSEWHERE (0.3 <= matrix .AND. matrix <= 0.7)
    matrix = EXP(matrix)
ELSEWHERE (0.7 < matrix .AND. matrix <= 1)
    matrix = LOG(matrix)
ELSEWHERE
    matrix = matrix + 10
END WHERE
```

Оператор FORALL имеет следующий вид:

```
FORALL(<индексное выражение>[, <условие>]) <присваивание>
```

где <индексное выражение> задает диапазоны индексов просматриваемых элементов, <условие> представляет собой логическое выражение (маску), результатом которого является скалярная величина, определяющая должен ли быть изменен элемент массива с соответствующим индексом, а <присваивание> описывает способ изменения этого элемента. Если операторов присваивания несколько, то их следует заключить в конструкцию FORALL…END FORALL. Оператор FORALL выполняется следующим образом. Сначала вычисляется индексное выражение, а затем маска для всех индексов. Для всех элементов из индексного выражения, для которых маска равна .TRUE., вычисляются правые части выражения присваивания, а потом выполняется присваивание. Ниже приведены примеры использования оператора FORALL:

```
FORALL(i=2:SIZE(list):2, list(i) < 50) list(i) = list(i)**2
FORALL(i=1:3, j=1:5:2, matrix(i, j) > 0.4)
    matrix(i, j) = matrix(i, j) + 0.7
    p(i, j) = matrix(i, j)
END FORALL
```

В первом случае возводятся в квадрат элементы массива с четными индексами, которые имели значение меньше 50. Во втором случае обрабатываются элементы нечетных столбцов, значения которых больше 0.4.

Fortran предоставляет средства для создания новых, производных, типов данных. *Производный тип* представляет собой структуру, которая состоит из нескольких компонентов встроенных или производных типов. Новый тип определяется с помощью оператора TYPE:

```
TYPE <имя_типа>
    <описание компонентов типа>
END TYPE <имя_типа>
```

Переменные производного типа называются структурами и объявляются следующим образом:

```
TYPE(<имя_типа>) <список переменных>
```

Для назначения значений компонентам структур можно использовать конструктор структур:

```
<имя_типа>(<список значений компонентов>)
```

Для обращения к отдельной компоненте используется спецификатор `%`. Для объектов производных типов можно определить операцию присваивания и арифметические операции. Ниже приведен пример описания и использования типа.

```
TYPE item
    CHARACTER(LEN = 50) :: name, category
    REAL :: price
    INTEGER :: qty
END TYPE item
TYPE(item) pen, table
pen = item("pen", "office", 7.85, 16)
table%name = "table"
table%category = "furniture"
table%price = 3784
table%qty = 7
```

Основными операторами ввода-вывода в Fortran являются `READ` (для ввода), `PRINT` и `WRITE` (для вывода). Они поддерживают форматирование, описывающее форму передачи данных и способ их преобразования. Для ввода набора данных в соответствии с некоторым форматом используется оператор `READ`. Оператор `PRINT` позволяет вывести на экран список значений в указанном формате. Оператор `WRITE` используется для вывода форматированных данных на экран или в файл. Формат задается перед вводимыми/выводимыми значениями в виде символьной переменной или выражения и может содержать спецификаторы преобразования данных и управляющие спецификаторы. Спецификаторы в формате разделяются запятой. Спецификаторы преобразования заключаются в круглые скобки. Перед открывающей скобкой может быть задано число повторений. Спецификаторы преобразования должны соответствовать элементам списка ввода-вывода. Спецификаторы и элементы списка интерпретируются слева направо. Каждый спецификатор преобразования применяется к соответствующему значению в списке ввода-вывода. Если в качестве формата задан символ `*` (звездочки), то форматирование определяется передаваемым оператору списком. Поддерживаются следующие спецификаторы (в квадратных скобках заданы необязательные элементы): `A[w]` — символьное значение, `Bw[.m]` — двоичное значение, `Dw.d` — вещественное значение двойной точности, `Ew.d[Ee]` — вещественное значение в экспоненци-

альной форме, Fw.d — вещественное значение с фиксированной запятой, Gw.d[Ee] — значение встроенного типа, Iw[.m] — целое значение, Lw — логическое значение, Ow[.m] — восьмеричное значение, Zw[.m] — шестнадцатеричное значение, nX — пропуск n позиций вправо от текущей позиции, Tn — табуляция до указанной позиции, TLn — табуляция влево на указанное число позиций, TRn — табуляция вправо на указанное число позиций, / — перевод строки, 'str' или "str" — вывести указанную в кавычках строку. Символы w и d задают соответственно общую длину поля и количество знаков в десятичной части, e — число знаков для показателя степени экспоненты, m — минимальное количество цифр для целого значения, n — число позиций. Ниже приведен пример использования операторов READ и PRINT.

```
TYPE(item) article
PRINT *, "Enter data about item:"
READ "(A20,/,A20,/,F10.2,/,I4)", &
     article%name,   article%category,   article%price,   arti-
cle%qty
PRINT "(/,'item data:',/,2X,'name - ',T15,A,/, &
      & 2X,'category - ',T15,A,/, &
      & 2X,'price - ',T15,F10.2,',', &
      & T30,'quantity - ',I4,/)", &
      article%name,   article%category,   article%price,   arti-
cle%qty
```

Условный оператор в языке Fortran имеет следующий вид:

```
IF (<логическое выражение 1>) THEN
    <операторы 1>
[ELSE IF (<логическое выражение 2>) THEN
    <операторы 2>
...
ELSE IF (<логическое выражение N>) THEN
    <операторы N>]
[ELSE
    <операторы E>]
END IF
```

Конструкции ELSE IF и ELSE необязательны. Оператор выбора задается в следующей форме:

```
SELECT CASE (<выражение>)
    CASE (<список значений 1>)
        <операторы 1>
    CASE (<список значений 2>)
        <операторы 1>
...
    CASE (<список значений N>)
        <операторы N>
    CASE DEFAULT
        <операторы D>
END SELECT
```

Значения в списках разделяются запятыми. При выполнении оператора SELECT вычисляется значение выражения, которое должно быть целым, символьным или логическим, и выполняются операторы той ветки, в список для которой входит полученная величина. Ветка DEFAULT используется, если значение выражения не было найдено ни в одном из списков. Помимо отдельных значений в список могут входить диапазоны значений, задаваемые в форме <начало>:<конец>, где одна из границ может быть опущена. Приведенный ниже пример демонстрирует использование оператора SELECT.

```
REAL :: a, b, res
CHARACTER(LEN = 3) :: oper
PRINT *, "Enter data: a operation b"
READ *, a, oper, b
SELECT CASE (oper)
    CASE ('+', 'a':'c')
        res = a + b
    CASE ('-', 'sub')
        res = a - b
    CASE ('*', 'mul')
        res = a * b
    CASE ('/', 'div')
        res = a / b
    CASE DEFAULT
        res = 0
END SELECT
PRINT "(A,f20.7)", "Result - ", res
```

Цикл с предусловием организуется с помощью следующего оператора:

```
DO WHILE (<логическое выражение>)
    <операторы тела цикла>
END DO
```

Цикл с постусловием в языке Fortran отсутствует. Арифметический цикл организуется с помощью оператора DO:

```
DO <счетчик> = <начало>, <конец>[, <шаг>]
    <операторы тела цикла>
END DO
```

где <счетчик> — переменная целого типа, <начало>, <конец> и необязательный <шаг> — это выражения, определяющие начальное и конечное значение счетчика и шаг его изменения, по умолчанию равный 1. В теле цикла нельзя изменять значение переменной-счетчика. По завершению цикла значение этой переменной становится неопределенным. Ниже представлен пример реализации арифметического цикла.

```
INTEGER :: i, m
m = 1
DO i = 1, 10, 2
    m = m * i
    PRINT *, i, " - ", m
END DO
```

Лекция №. 15.

ТЕМА: Основы языка программирования С.

Основные вопросы, рассматриваемые на лекции:

1. Структура программы на языке С.
2. Базовые типы данных.
3. Описание констант и строк.
4. Указатели и операции с ними.
5. Массивы и структуры.
6. Базовые функции ввода-вывода.
7. Операторы управления ходом вычислительного процесса.
8. Функция main.
9. Директивы препроцессора.

Программа на языке С представляет собой совокупность одного или нескольких модулей — самостоятельно компилируемых файлов. Такой файл обычно содержит директивы препроцессора, а также одну или несколько функций, состоящих из операторов языка С. Функция, с которой начинается выполнение программы, всегда имеет имя `main`. Модуль может содержать любую целостную комбинацию директив препроцессора, указаний компилятору, объявлений и определений. Под целостностью подразумевается, что такие объекты, как определения функций, структуры данных, либо связанные между собой директивы условной компиляции, должны целиком располагаться в одном файле, т.е. не могут начинаться в одном файле, а продолжаться в другом. Директивы препроцессора описывают действия препроцессора по преобразованию текста программы перед компиляцией. Указания компилятору — это специальные инструкции, которым компилятор языка С следует во время компиляции. Объявление переменной задает её имя и атрибуты, а объявление функции — её имя, тип возвращаемого значения и атрибуты её формальных параметров. Определение переменной приводит к выделению для неё памяти и явно или неявно задает её начальное значение. Определение функции специфицирует тело функции, которое представляет собой составной оператор (блок), содержащий определения переменных и операторы. Объявление типа позволяет программисту создавать собственный тип данных, точнее присвоить своё имя некоторому базовому или составному типу языка С. Для типа понятие объявления и определения совпадают.

В программах на языке С используются два типа файлов: файлы реализации, имеющие расширение .c, и заголовочные файлы (файлы интерфейса), имеющие расширение .h. В заголовочных файлах обычно располагаются именованные константы, макроопределения и объявления, а в фай-

лах реализации — определения переменных и функций. Заголовочные файлы подключаются посредством директивы препроцессора `#include`.

Отличительной чертой языка С является то, что в нем нет ни одного встроенного оператора для выполнения ввода/вывода, динамического распределения памяти, управления процессами и т.д. В его окружение входит библиотека стандартных функций, в которых реализованы подобные и многие другие действия. Вынос этих функций в библиотеку позволяет обеспечить независимость реализации языка и созданных на нем программ от особенностей архитектуры конкретного компьютера и операционной системы.

Комментарии можно задавать одним из двух способов. В первом случае символы `/*` открывают комментарий, а символы `*/` — закрывают его. Такой тип обычно используется для написания многострочных комментариев. Такие комментарии не могут быть вложенными. Второй тип комментариев задается с помощью символов `//`. Они открывают комментарий, заканчивающийся в конце той же строки. Комментарии второго типа не входят в стандарт языка С, но распознаются многими компиляторами.

К базовым в языке С относятся следующие типы данных:

- Целые со знаком: `int`, `long`, `short`.
- Целые без знака: `unsigned`, `unsigned long`, `unsigned short`. Имеют только нулевые и положительные значения.
- Символы: `char`.
- Числа с плавающей точкой: `float` (обычной точности), `double` или `long float` (двойной точности).

Объем памяти в битах, необходимый для каждого типа данных, зависит от реализации языка. Исключение составляет тип `char`, который требует одного байта памяти. Конкретные значения можно узнать непосредственно из программы, для чего в языке предусмотрен специальный оператор `sizeof`, который возвращает число байт, необходимых для размещения конкретного типа данных.

В языке С любое значение, отличное от нуля, соответствует значению истина. До версии С99 логический тип данных не являлся частью стандарта языка С. При подключении к программе файла `stdbool.h`, введенного в С99, можно использовать макросы `bool`, `true` и `false` для ссылки соответственно на логический тип данных и значения истина и ложь.

Язык С содержит средства явного преобразования типов. Если нужно временно изменить формат некоторой переменной или значения выражения, то следует перед переменной или выражением указать тип, к которому необходимо выполнить преобразование. Например:

```
double k = 10.4, n = 3.57;
```

```
int v = (int) (k / n);
```

Наряду с явным преобразованием в языке реализованы следующие правила автоматического преобразования типов, выполняемого, если выражение является параметром функции или операндом.

1. Значения типов `char` и `short` преобразуются к типу `int`, а значения типа `float` — к типу `double`.

2. Если один из операндов имеет тип `double`, то и другие преобразуются к этому типу.

3. Если один из операндов имеет тип `long`, то и другие преобразуются к этому типу.

4. Если один из операндов имеет тип `unsigned`, то и другие преобразуются к этому типу.

5. Иначе оба операнда имеют тип `int`.

Проверка типов параметров при вызове функций не выполняется. Тем не менее, при этом всегда выполняется первое правило.

В языке С имеется 4 типа констант: целые, вещественные, символьные и перечисляемого типа. Целые числа можно выражать в одной из трех систем счисления: десятичной, шестнадцатеричной (начинаются с 0x или 0X, например 0x4b, 0XA2C) и восьмеричной (начинаются с 0, например 0537). Любая константа, которая лежит вне диапазона целых чисел одинарной точности, или константа, заканчивающаяся на l или L, рассматривается как длинное целое число. Вещественные константы представляют собой числа по основанию 10. Они всегда содержат десятичную точку и/или обозначение показателя степени E (или e) в научной нотации. Константы перечисляемого типа представляют собой идентификаторы, определяемые с помощью объявления типа `enum`, которые служат в качестве имен, повышающих читабельность программы.

Символьная константа — это один символ, заключенный в одинарные кавычки (апострофы `'`). Символы делятся на две группы: печатаемые и непечатаемые. Печатаемые символы включают в себя буквы, цифры и другие знаки, которые можно набрать на клавиатуре. Непечатаемым символам соответствуют специальные управляющие последовательности, называемые также *esc-последовательностями*, которые служат для управления внешними устройствами или для других видов управления. Для введения управляющих последовательностей, позволяющих получить визуальное представление некоторых символов, не имеющих графического аналога, используется символ косой черты — обратного слэша (\), за которым следует специальный символ, восьмеричное или шестнадцатеричное число. Допустимы следующие управляющие последовательности: `'\a'` — звуковой сигнал, `'\b'` — возврат на шаг, `'\f'` — перевод формата, `'\n'` — перевод строки, `'\r'` — возврат каретки, `'\t'` — горизонтальная табуляция, `'\v'` — вертикальная табуляция, `'\\'` — обратная косая

черта, `'\''` — апостроф, `'\"'` — двойная кавычка, `'\?'` — вопросительный знак, `'\восьмеричная константа'` и `'\шестнадцатеричная константа'` — позволяют задавать любое целое значение в качестве кода символьной константы. Очень важную роль играет константа `'\0'`, которая представляет байт с числовым значением 0. Её называют *пустым символом, нулевым символом* или *null-символом*.

Строка в языке С представляет собой последовательность символов, заключенную в кавычки. Для хранения строк компилятор использует по одному байту на каждый символ и автоматически добавляет к ней признак конца строки, которым служит нулевой символ. Таким образом, для хранения строки из N символов всегда выделяется N+1 байт памяти. Чтобы ввести в строку какую-либо символьную константу, перед ней необходимо указать символ \. Для улучшения читаемости текста длинную строку можно разбить на несколько строк. Для этого перед переходом на новую строку надо вставить обратный слэш. Стандарт ограничивает длину строки 509 символами. Однако многие компиляторы поддерживают строки большей длины.

При программировании на языке С широко используются указатели. *Указатель* — это переменная, предназначенная для хранения адреса в памяти. Различают две категории указателей: указатели объектов и указатели функций. Указатели функций используются для доступа к функциям и для передачи одних функций другим в качестве аргументов. Указатель объектов — это самостоятельная переменная, которой можно присваивать значение адреса некоторой переменной, но нельзя записывать значение самой переменной. Определение указателя объектов всегда должно устанавливать его на некоторый конкретный тип, даже если это тип `void` (что означает указатель на любой объект). Указатели описываются следующим образом:

```
<имя_типа> *<имя_указателя>
```

Для корректной работы перед использованием указатель должен быть инициализирован. Если не предполагается никакого начального значения, то можно инициализировать указатель значением 0, которое гарантирует отличие этого адреса от любого другого допустимого адреса, используемого в программе. В файлах заголовков стандартной библиотеки для этого значения определен специальный идентификатор `NULL`. При работе с указателями используются две основные операции:

- & — операция получения адреса. Выдает адрес переменной, имя которой стоит за обозначением операции.
- * — косвенная адресация. Выдает значение, хранящееся по адресу, на который ссылается указатель.

Следующий пример демонстрирует использование этих операций. После выполнения данного кода переменная k будет иметь значение 10, а переменная n — 5.

```
int k, n, *ptr;
ptr = &k;
k = 5;
n = *ptr;
*ptr = 10;
```

Над указателями можно выполнять некоторые операции целочисленной арифметики: складывать, вычитать и сравнивать между собой. Сложение и вычитание указателей всегда выполняется в единицах памяти, т.е. в единицах того типа, к которому относится указатель. Язык С допускает создание указателей на указатели. Например, объявление `int **p;` определяет указатель, который содержит указатель на тип `int`.

Массивы в языке С описываются следующим образом:

```
<тип_данных> <имя_массива>[<размер_массива>];
```

Массив занимает непрерывную область памяти, по размеру позволяющую разместить все его элементы. Каждый из элементов массива нумеруется от нуля до величины, на единицу меньшую, чем размер массива. В языке С отсутствует проверка выхода за границу массива. Имя массива фактически является указателем-константой, ссылающимся на начальный адрес данных. Начальный адрес массива определяется компилятором в момент его описания и никогда не может быть переопределен. Доступ к элементам массива осуществляется по его имени и индексу элемента. Кроме того, обратиться к элементу массива можно с помощью указателя. При описании глобального или статического массива его элементам можно присвоить начальные значения, заключив их в фигурные скобки:

```
<тип_данных> <имя_массива>[<размер_массива>] = {<значение 1>, {<значение 2>, ..., {<значение N>};
```

В этом случае размер массива можно не указывать, т.к. компилятор определит его по числу заданных значений. Нижеприведенный пример демонстрирует описание и использование массива.

```
int list[] = {1, 2, 3};
list[0] = list[1] * list[2];
*(list + 1) = 5;
*(list + 2) = list[1];
```

Строки представляют собой одномерные массивы типа `char` и их можно инициализировать следующим образом:

```
char first[] = "first";
char *second = "second";
```

Многомерные массивы создаются путем объявления массивов из элементов типа «массив» и хранятся по строкам. То есть, многомерные массивы располагаются в памяти так, что быстрее всего меняется последний индекс. Например, двумерный массив вещественных чисел объявляет-

ся следующим образом: `float matrix[5][3];`. Имя двумерного массива является указателем на первую строку массива. Поэтому справедливо следующее равенство:

```
array[row][col] == *(*(data + row) + col)
```

При инициализации двумерных массивов значения элементов каждой строки также следует заключать в фигурные скобки.

Язык С поддерживает специальный тип данных, называемый структурой. *Структура* — это набор переменных различных типов, образующих единый объект. Описываются структуры следующим образом:

```
struct <имя_структуры> {
    <тип1> <элемент1>;
    <тип2> <элемент2>;
    ...
    <типN> <элементN>;
};
```

Соответствующие переменные задаются с помощью следующей записи:

```
    struct <имя_структуры> <имя_переменной>;
```

Допускается создавать указатели на структуры. Структуры можно инициализировать в момент определения. Для этого в фигурных скобках через запятую следует перечислить необходимые значения элементов структуры. Для обращения к элементу структуры следует задать имя структурной переменной и имя требуемого элемента, разделив их операцией . (точка) или операцией -> при использовании указателя. Ниже приведен пример, демонстрирующий использование структур.

```
struct item {
    char *name;
    char *category;
    float price;
    int qty;
};
struct item pen = {
    "pen",
    "office",
    7.85,
    16
};
struct item *itemPtr = &pen;
float cost = pen.price * pen.qty;
itemPtr->price = 10;
```

Базовыми функциями ввода-вывода в языке С являются функции `getchar()` и `putchar()`. Функция `getchar()` обеспечивает считывание одного символа из «стандартного ввода», который обычно связан с клавиатурой, и возвращает его в качестве результата своей работы. Функ-

ция putchar() выводит символ, переданный ей в качестве параметра, на «стандартный вывод», который обычно связан с дисплеем. Функция printf позволяет выполнить так называемый форматированный вывод. Общий вид обращения к функции printf выглядит следующим образом (в квадратных скобках представлены необязательные элементы):

printf(<управляющая строка>[, <список аргументов>])

Управляющая строка может содержать символы, которые следует напечатать, управляющие последовательности, перед которыми стоит обратная косая черта, и спецификации преобразования. Каждой спецификации преобразования должен ставиться в соответствие некий аргумент из списка параметров. Спецификация имеет следующий общий вид: %[-][w][.p][l]t, где

- - — означает, что выдаваемое значение должно быть в поле выровнено влево (по умолчанию оно выравнивается вправо);
- w — число, задающее минимальный размер поля, т.е. общее число позиций, занимаемых печатаемым материалом. В избыточных позициях поля по умолчанию «печатается» пробел. Если первая цифра размера есть ноль, то поле должно дополняться нулями. Если величина слишком велика для заданного спецификацией поля, то она будет напечатана полностью. Если вместо строки цифр стоит * (звездочка), то это означает, что размер поля задается значением, приведенным в соответствующей позиции списка аргументов.
- p — число, указывающее для величины типа float или double, сколько будет напечатано цифр после десятичной точки. Если речь идет о строках, то задает число печатаемых из строки символов.
- l — указывает размер данных, переданных функции. Например, значение l определяет, что соответствующий аргумент относится к типу long.
- t — символ, определяющий тип и представление выводимой величины. Допустимы следующие значения: c — вывод символа с кодом, соответствующим переданному аргументу, d, i — десятичное знаковое число, u — десятичное беззнаковое число, o — восьмеричное беззнаковое число, x — шестнадцатеричное число, f — число с плавающей запятой, e — число с плавающей запятой в экспоненциальной форме записи, g — наиболее короткое представление из %f и %e, a — число с плавающей запятой в шестнадцатеричном виде, s — вывод строки, p — вывод указателя, % —вывод знака процента.

Для ввода данных можно использовать функцию scanf, обращение к которой имеет следующий общий вид:

scanf(<управляющая строка>[, <список аргументов>])

Управляющая строка обычно содержит спецификации преобразования, которые используются для интерпретации входных данных. Пробелы, символы табуляции и переходы на новую строку внутри управляющей строки игнорируются. Если в управляющей строке появляется какой-либо символ, кроме тех, которые относятся к спецификации преобразования, то считается, что он должен совпадать с первым непустым символом из входного потока. Спецификация преобразования имеет следующий общий вид: `%[*][w]t`, где

- `*` — символ подавления присваивания (ввод пропускается и присваивание не производится);
- `w` — число, задающее максимальную ширину поля;
- `t` — символ преобразования, определяющий интерпретацию вводимых данных. Допустимы следующие значения: `c` — символ, `h` — число типа `short`, `d` — число типа `int`, `ld` — число типа `long`, `o` или `lo` — восьмеричное целое типа `int` или `long`, `x` или `lx` — шестнадцатеричное целое типа `int` или `long`, `e` или `f` — число типа `float`, `le` или `lf` — число типа `double`, `s` — строка символов.

Ниже приведен простой пример использования функций `printf` и `scanf`:

```
int size;
printf("Enter size: ");
scanf("%d", &size);
printf("You entered: %d\n", size);
```

Все рассмотренные выше функции ввода-вывода объявлены в файле `stdio.h`.

Для организации разветвляющегося вычислительного процесса в языке С предусмотрены операторы `if` и `switch`. В общем виде оператор `if` имеет следующую структуру:

```
if (<выражение1>)
    <оператор1>
[else if (<выражение2>)
    <оператор2>
...
else if (<выражениеN>)
    <операторN>]
[else
    <операторE>]
```

Операторы могут быть простыми или составными. Конструкции `else if` и `else` являются необязательными. При сопоставлении конструкции `else` определенному оператору `if` действует следующее правило: `else` соответствует ближайшему `if`. Как уже было сказано ранее, следует учитывать, что в языке С любое значение, отличное от нуля, считается истинным.

В самом общем случае оператор выбора `switch` имеет вид:

```
switch (<выражение>)
{
    case <метка1>: <оператор1>;
    case <метка2>: <оператор2>;
...
    case <меткаN>: <операторN>;
    default: <операторD>;
}
```

Управление передается оператору, метка которого совпадает со значением выражения. После него выполняются операторы, соответствующие всем последующим меткам и конструкции `default`, если она есть. Чтобы избежать этого, следует использовать оператор `break`, который позволяет прервать выполнение оператора `switch` и передает управление следующему за ним оператору. Как выражение, так и метки должны иметь значения целого типа или типа `char`. Метки должны быть константами или константными выражениями. Если значению выражения не соответствует никакая метка, управление передается оператору с меткой `default`, в случае отсутствия которого выполняется оператор, следующий за оператором `switch`. Операторы могут быть простыми или составными, причем их необязательно заключать в фигурные скобки. Рассмотрим следующий пример.

```
switch (op)
{
    case '+': res = a + b;
    case '-': res = a - b;
    case '*': res = a * b;
    case '/': res = a / b;
    default: res = 0;
}
```

В данном случае вне зависимости от значения переменной `op` переменная `res` всегда будет принимать значение 0. Чтобы переменная `res` получала значение, соответствующее определенной операции, необходимо преобразовать оператор `switch` к следующему виду:

```
switch (op)
{
    case '+': res = a + b; break;
    case '-': res = a - b; break;
    case '*': res = a * b; break;
    case '/': res = a / b; break;
    default: res = 0;
}
```

В языке С существует три оператора для организации циклов, соответствующие циклу с предусловием, циклу с постусловием и арифметическому циклу. Цикл с предусловием организуется с помощью оператора `while`, который в общей форме имеет следующий вид:

```
while (<выражение>)
```

```
    <простой или составной оператор>
```

Данный цикл повторяется до тех пор, пока проверяемое выражение не станет ложным, или нулем. Для организация цикла с постусловием используется оператор `do while`, который задается следующим образом:

```
do
    <простой или составной оператор>
while (<выражение>);
```

Повторения выполняются до тех пор, пока выражение является истинным.

Оператор `for` позволяет организовать арифметический цикл. Он имеет следующую общую форму:

```
for (<инициализация>; <проверка_условия>; <управление_циклом>)
    <простой или составной оператор>
```

В каждом разделе может быть указано несколько выражений, разделенных запятой. Выражения, приведенные в разделе инициализации вычисляются один раз до начала выполнения операторов тела цикла. Выражения во втором разделе служат для проверки условия, которая производится перед каждым возможным выполнением тела цикла. Когда выражение, которое приведено последним, становится ложным (равным нулю), цикл завершается. Выражения в разделе «управление циклом» вычисляются после каждого выполнения тела цикла. Ниже приведен пример цикла `for`, в котором выводятся значения факториалов от 1 до n.

```
for (i = 1, f = 1; f = f * i, i <= n; i++)
    printf("factorial of %d - %d\n", i, f);
```

Функция, с которой начинается выполнение программы, имеет имя `main` и должна быть единственной в программе. В общем случае она выглядит следующим образом:

```
main(int argc, char **argv, char **envp)
```

Аргументы являются необязательными и служат для передачи в программу информации из командой строки:

- `argc` — число, определяющее количество аргументов, передаваемых в программу из командной строки. Поскольку в качестве первого аргумента всегда передается имя программы, это значение не бывает меньше 1.
- `argv` — массив строк, содержащий аргументы, переданные из командой строки, который можно объявлять как массив указателей на `char` (`char *argv[]`) или как указатель на указатель `char` (`char **argv`). `argv[0]` всегда содержит имя программы, `argv[argc]` равен `NULL`.
- `envp` — указатель на массив строк, определяющих среду выполнения программы, для которого правила объявления такие же, как для `argv`. Конец массива задается приравниванием последнего указателя `NULL`.

Препроцессор C обрабатывает исходный текст программы, прежде чем он поступит на вход компилятора. Он может также изменить условия компиляции. Для обозначения директив препроцессора используется символ #. Директивы могут быть записаны в любом месте исходного файла. Их действие распространяется от точки, в которой они записаны, до конца этого файла. Ниже приведены некоторые директивы препроцессора (в квадратных скобках представлены необязательные элементы).

- `#define идентификатор[(список_параметров)][ текст]` — используется для замены часто используемых в программе констант, ключевых слов, операторов и выражений содержательными идентификаторами. Идентификатор не должен содержать пробелов.
- `#undef идентификатор` — отменяет текущее определение идентификатора.
- `#include <файл>` или `#include "файл"` — включает содержимое указанного файла. Имя файла должно соответствовать соглашениям операционной системы и может состоять либо только из имени файла, либо из имени файла с предшествующим ему путем. Если имя файла указано в кавычках, то поиск файла осуществляется в соответствии с заданным путем, а при его отсутствии в текущем каталоге. Если имя файла задано в угловых скобках, то поиск файла производится в стандартных каталогах операционной системы, задаваемых командой PATH. Данная директива широко используется для включения в программу так называемых заголовочных файлов, содержащих прототипы библиотечных функций.
- `#if константное_выражение` — условная подстановка фрагмента текста в зависимости от значения константного выражения: фрагмент подставляется, если оно истинно.
- `#endif` — Обозначает конец условно подставляемого фрагмента текста, начатого директивой `#if`.
- `#ifdef идентификатор`, `#ifndef идентификатор` — условная подстановка фрагмента текста в зависимости от того, определен ли идентификатор.
- `#else` — позволяет организовать альтернативную ветвь условной подстановки внутри директивы `#if`.
- `#elif константное_выражение` — используется для организации альтернатив внутри директивы `#if`.
- `#pragma последовательность_символов` — определяет зависящие от реализации указания компилятору (прагмы).

Лекция № 16.

ТЕМА: Основы языка Java и особенности реализации в нем принципов объектно-ориентированного программирования.

Основные вопросы, рассматриваемые на лекции:

1. Основные особенности языка Java.
2. Базовые инструментальные средства разработки и выполнения Java-программ: JDK и JRE.
3. Принципы и особенности организации Java-программ.
4. Документирование программ. Аннотации.
5. Особенности реализации принципов объектно-ориентированного программирования.
6. Пакеты.
7. Интерфейсы.

Java представляет собой набор инструментальных средств и технологий, предназначенных для разработки программных приложений различного назначения. К числу этих средств и технологий, в частности, относятся Java SE, Java EE, Java ME, JavaFX, каждая из которых ориентирована на создание того или иного типа программного обеспечения. Однако всех их объединяет лежащий в основе объектно-ориентированный язык программирования Java.

Ключевая особенность Java состоит в том, что программа компилируется в команды так называемой виртуальной машины Java (Java Virtual Machine, JVM). Виртуальная машина Java — это совокупность команд, называемых байт-кодами, вместе с системой их выполнения. Компиляция Java-программы в байт-коды не зависит от типа какого-либо конкретного процессора и архитектуры компьютера. Поэтому программу не надо перекомпилировать под разные платформы. Байт-коды записываются в одном или нескольких файлах и могут храниться во внешней памяти или передаваться по сети. Все стандартные функции, вызываемые в программе, подключаются к ней только на этапе выполнения, а не включаются в байт-коды, т.е. происходит динамическая компоновка. Полученные в результате компиляции байт-коды можно выполнять на любом компьютере, имеющем систему, реализующую JVM.

Кроме реализации JVM для выполнения байт-кодов на компьютере еще нужно иметь набор функций, вызываемых из байт-кодов и динамически компонующихся с ними. Этот набор оформляется в виде библиотеки классов Java, состоящей из одного или нескольких пакетов. Каждая функция может быть записана в системе команд конкретного компьютера, на котором она будет храниться. Такие функции, реализованные обычно на

языке C/C++ и скомпилированные под определенную платформу, называют «родными» методами (native methods). Их применение ускоряет выполнение программы.

Инструменты JDK и JRE. JDK (Java Development Kit) представляет собой набор необходимых программных инструментов для полного цикла работы с языком Java: компиляции, интерпретации, отладки, включающий и богатую библиотеку классов. JDK содержит: компилятор из исходного текста в байт-коды javac; интерпретатор java, содержащий реализацию JVM; программу просмотра апплетов appletviewer, заменяющую браузер; отладчик jdb; дизассемблер javap; программу архивации и сжатия jar; программу сбора и генерирования документации javadoc; программу генерации заголовочных файлов языка C для создания «родных» методов javah; программу добавления электронных подписей javakey; программу native2ascii, преобразующую бинарные файлы в текстовые; программы rmic и rmiregistry для работы с удаленными объектами; программу serualver, определяющую номер версии класса; библиотеки и заголовочные файлы «родных» методов; библиотеку классов Java API.

Кроме JDK компания Sun отдельно распространяет еще и набор JRE (Java Runtime Environment). Набор программ и пакетов классов JRE содержит всё необходимое для выполнения байт-кодов, в том числе интерпретатор java и библиотеку классов. Это часть JDK, не содержащая компиляторы, отладчики и другие средства разработки.

Набор JDK ориентирован на выполнение из командой строки. После создания файла программы из командной строки вызывается компилятор `javac` и ему передается исходный файл как параметр: `javac Program.java`. Компилятор создает в том же каталоге по одному файлу на каждый класс, описанный в программе, называя каждый файл именем класса с расширением class. Если компилятор заметит ошибки, то он выведет на экран сообщения о них. Далее из командной строки вызывается интерпретатор байт-кодов java, которому передается файл с байт-кодами, причем его имя записывается без расширения: `java Program`. На экране появится вывод результатов работы программы или сообщения об ошибках времени выполнения.

Объектно-ориентированная структура организации Java-программ. Рассмотрим простую Java-программу.

```
class Example1
{
    public static void main(String[] args)
    {
        System.out.println("Hello world!");
    }
}
```

Любая программа, написанная на языке Java, представляет собой один или несколько классов. Имя файла программы должно в точности

совпадать с именем класса, который описан в этом файле. Все, что содержится в классе, записывается в фигурных скобках и составляет тело класса. Все действия в программе производятся с помощью методов. Методы различаются по именам и параметрам. Один из методов одного из классов обязательно должен называться `main`, с него начинается выполнение программы. Метод всегда возвращает в результате только одно значение, тип которого обязательно указывается перед именем метода. Метод может и не возвращать никакого значения, играя роль процедуры. Тогда вместо типа возвращаемого значения записывается слово `void`. После имени метода в скобках через запятую перечисляются параметры метода. Для каждого параметра указывается его тип и, через пробел, имя. У метода `main()` только один параметр, его тип — массив, состоящий из строк символов. Строка символов — это встроенный в Java API тип `String`, а квадратные скобки — признак массива. Имя параметра может быть произвольным. Перед типом возвращаемого методом значения могут быть записаны модификаторы. В примере их два: слово `public` означает, что этот метод доступен отовсюду; слово `static` обеспечивает возможность вызова метода `main()` в самом начале выполнения программы. Модификаторы необязательны, но для метода `main` они необходимы. Все, что содержит метод называется телом метода и записывается в фигурных скобках.

Единственное действие, которое выполняет метод `main()` в рассмотренном примере, заключается в вызове другого метода со сложным именем `System.out.println` и передаче ему на обработку одного аргумента — текстовой константы “Hello world!”. Текстовые константы записываются в кавычках, которые являются ограничителями и не входят в текст. Составное имя `System.out.println` означает, что в классе `System`, входящем в Java API, определяется переменная с именем `out`, содержащая экземпляр одного из классов Java API, а именно класса `PrintStream`, в котором есть метод `println()`. Действие этого метода заключается в выводе заданного ему аргумента в выходной поток, связанный обычно с выводом на экран текстового терминала.

В состав JDK включена программа javadoc, извлекающая комментарии вида `/** текст */` в отдельные файлы формата HTML и создающая гиперссылки между ними. Обычно такой комментарий содержит описательную часть, за которой располагаются указания программе javadoc, начинающиеся с символа @ и называемые тэгами. Существует два типа тэгов: блочные, которые записываются как @tag, и внутристрочные, которые располагаются внутри фигурных скобок и выглядят как {@tag}. Блочные тэги должны записываться с начала строки и могут предваряться пробелами или символами *. В противном случае они рассматриваются как обычный текст. По соглашению, тэги с одним и тем же именем группируются вместе. Каждый блочный тэг имеет связанный с ним текст, который может

быть разбит на несколько строк и продолжается до начала следующего блочного тэга или конца комментария. Внутристрочный тэг может быть записан внутри любого текстового фрагмента комментария. Текст описательной и тэговой части комментария должен быть написан в формате HTML. Утилита javadoc распознает, в частности, следующие тэги.

- @author <имя>. Добавляет сведения об авторе описываемого элемента в документацию.
- @deprecated <текст>. Добавляет комментарий, указывающий, что описываемый элемент не должен более использоваться при написании программ.
- {@link <пакет>.<класс>#<элемент> <текст>}. Вставляет ссылку с заданным текстом, которая указывает на документацию некоторого пакета, класса или элемента класса. Текст является необязательным и если он не указан, то используется имя элемента, на который указывает ссылка.
- @param <имя параметра> <описание>. Добавляет описание параметра метода, конструктора или класса.
- @return <описание>. Описывает возвращаемое методом значение.
- @see <ссылка>. Добавляет ссылку в раздел “Смотри также”. Ссылка может быть задана как обычная строка (например, “The Java Programming Language”), как HTML-ссылка (например, Java Spec) или в форме <пакет>.<класс>#<элемент> <текст>, указывающей на некоторый пакет, класс или элемент класса. Текст в последнем случае является необязательным и если он не указан, то используется имя элемента, на который указывает ссылка.
- @throws <имя класса> <описание>. Добавляет раздел, описывающий исключение, которое может быть сгенерировано методом или конструктором. Имя класса задает имя генерируемого исключения. Синонимом данного тэга является тэг @exception.
- @version <текст>. Указывает версию описываемого пакета или класса.

Начиная с пятой версии Java SE, появилась возможность описывать и использовать в тексте программы тэги, которые получили название аннотаций. Аннотации задают способ обработки программы различными утилитами и библиотеками. Они могут быть считаны из исходных или откомпилированных файлов, а также во время выполнения. Аннотации дополняют javadoc-тэги и записываются в следующей форме:

@annotation <необязательный список пар элемент=значение через запятую>

Они представляют собой специальный вид модификаторов и записываются не внутри комментариев вида `/**...*/`, а там, где допустимы другие модификаторы (`public`, `static` и т.д.). По соглашению, аннотации

предваряют остальные модификаторы. Например, если записать перед заголовком какого-либо метода аннотацию `@Deprecated`, компилятор будет выводить на консоль предупреждение о том, что этот метод устарел и его не следует больше использовать в программе. Некоторые аннотации поддерживаются непосредственно компилятором и их можно использовать без дополнительных усилий. Кроме них разработчик может объявить и использовать в своем приложении собственные аннотации.

Особенности реализации принципов объектно-ориентированного программирования в Java. Описание класса в языке Java начинается со слова `class`, после которого записывается имя класса. Перед словом `class` можно записать *модификаторы* класса, к которым относятся `public`, `abstract`, `final`, `strictfp`. Перед именем вложенного класса можно поставить также модификаторы `protected`, `private`, `static`. В теле класса, заключенном в фигурные скобки, в любом порядке перечисляются поля, методы, конструкторы, вложенные классы и интерфейсы. При описании поля указывается его тип, затем, через пробел, имя и, может быть, начальное значение после знака равенства, задаваемое константным выражением. Описание поля может начинаться с одного или нескольких необязательных модификаторов `public`, `protected`, `private`, `static`, `final`, `transient`, `volatile`. При описании метода указывается тип возвращаемого им значения или слово `void`, затем, через пробел, имя метода, потом, в скобках, список параметров. После этого в фигурных скобках записывается код метода. Описание метода может начинаться с модификаторов `public`, `protected`, `private`, `abstract`, `static`, `final`, `synchronized`, `native`, `strictfp`. В списке параметров через запятую перечисляются тип и имя каждого параметра. Перед типом параметра может стоять модификатор `final`, означающий, что параметр нельзя менять внутри метода. Список параметров может отсутствовать, но скобки сохраняются. Кроме полей и методов язык Java позволяет в классе описать *вложенные классы* и *вложенные интерфейсы*, в которые, в свою очередь, можно вложить классы и интерфейсы. Поля, методы и вложенные классы первого уровня называются *членами класса*. Ниже приведен пример описания класса.

```
public class Automobile
{

    // Поле, содержащее наибольшую скорость автомоби-
ля
    protected int maxVelocity;
    // Поле, содержащее текущую скорость автомобиля
    protected int speed;
    // Поле, содержащее вес автомобиля
```

```
    protected int weight;
    // Другие поля...

    // Метод, моделирующий перемещение автомобиля в
точку (x, y).
    // Параметры метода x и y - не поля, а локальные
переменные.
    public void moveTo(int x, int y)
   {
        // Локальная переменная, используемая в теле
метода
        int a = 1;
        // Тело метода, задающее его реализацию...
    }
    // Другие методы класса...
}
```

После того как описание класса закончено, можно создавать конкретные объекты, называемые *экземплярами* данного класса. Создание экземпляров производится в три этапа. Сначала объявляются ссылки на объекты: записывается имя класса и после пробела через запятую перечисляются экземпляры класса, точнее ссылки на них. Например:

```
Automobile lada2110, fordFocus, nissanXTrail;
```

Затем операцией `new` определяются сами объекты, под них выделяется оперативная память, ссылка получает адрес этого участка в качестве своего значения. Например:

```
lada2110 = new Automobile();
fordFocus = new Automobile();
nissanXTrail = new Automobile();
```

На третьем этапе происходит инициализация объектов, задаются начальные значения. Этот этап, как правило, совмещается со вторым, когда выполняется так называемый *конструктор* класса. Обращение к полю, методу или вложенному классу конкретного объекта осуществляется с помощью оператора точка: задается ссылка на объект, после которой через точку следует имя требуемого члена класса. Например:

```
lada2110.maxVelocity = 170;
fordFocus.maxVelocity = 200;
nissanXTrail.maxVelocity = 180;
fordFocus.moveTo(74, 150);
```

Действия, выполняемые методом, зависят от объекта и от его текущего состояния (значений его полей). Обращение к методу происходит только на этапе выполнения программы. Это называется «поздним связыванием» в противовес «раннему связыванию», при котором процедура присоединяется к программе на этапе компоновки. В языке Java аргументы

передаются в метод по значению. Если метод имеет параметр ссылочного типа, то в этом случае в локальную переменную-параметр копируется значение аргумента-ссылки. После этого метод будет работать непосредственно с объектом, на который ссылается аргумент, при помощи локальной копии аргумента-ссылки.

В языке Java изолирование (сокрытие) членов класса достигается добавление модификатора `private` к его описанию. Такие члены класса становятся закрытыми и к ним можно обращаться только внутри данного класса. Модификатор `public` делает соответствующий член класса открытым, т.е. доступным извне класса. К нему может обратиться любой объект любого класса. Модификатор `protected` ограничивает область использования компонентов описанием класса и его подклассов любого уровня. На доступ к члену класса влияет еще и пакет, в котором находится класс.

Если необходимо обратиться к закрытому полю класса, то рекомендуется включить в класс специальные *методы доступа*, отдельно для чтения этого поля и отдельно для записи в это поле. Имена методов доступа рекомендуется начинать со слов `get` и `set`, добавляя к ним имя поля. Кроме методов доступа рекомендуется создавать проверочные *is-методы,* возвращающие логическое значение `true` или `false` в зависимости от значения некоторого логического поля или выполнения некоторого условия.

Наследование в языке Java задается с помощью слова `extends`, записываемого после имени класса, за которым следует имя родительского класса. Таким образом, язык Java поддерживает только одиночное наследование. Принцип наследования обеспечивает «передачу и сохранение» структур данных и подпрограмм родительских классов в дочерних. При этом включать в описание класса-потомка компоненты класса-предка не надо, а использовать их можно. Кроме того, подкласс может дополнять наследуемые структуры своими и модифицировать (замещать или перегружать) операции над данными.

Имя метода, число и типы параметров образуют *сигнатуру* метода. Компилятор различает методы не по их именам, а по сигнатурам. Это позволяет записывать разные методы с одинаковыми именами, отличающиеся числом и/или типами параметров. Такое дублирование методов называется их *перегрузкой*. Если записать в подклассе метод с тем же именем и параметрами, что и в суперклассе, то он перекроет метод суперкласса, т.е. произойдет *переопределение* метода. В этом случае метод будет выполняться так, как он реализован в дочернем, а не родительском классе. Проверку соответствия сигнатуры переопределяемого метода можно возложить на компилятор, записав перед методом подкласса аннотацию `@Override`. В этом случае компилятор пошлет на консоль сообщение об

ошибке, если сигнатура помеченного метода не будет соответствовать сигнатуре ни одного метода с тем же именем, описанного в одном из родительских классов. При переопределении метода права доступа к нему можно только расширить, но не сузить. Чтобы внутри переопределяемого метода обратиться к методу суперкласса, необходимо перед именем метода использовать слово `super` с точкой, например `super.moveTo(10, 20)`. Чтобы обратиться к какому-либо члену данного класса, можно воспользоваться словом `this` с точкой, например `this.name = newName`.

Полиморфизм поддерживается путем разрешения определения полиморфных переменных типа родительского класса, которые также могут ссылаться на объекты любых классов-потомков этого класса. Таким образом, если при объявлении переменной в качестве типа указан некоторый класс, то её значениями могут быть ссылки, как на объекты этого класса, так и на объекты любого дочернего класса. Вызов через такую переменную метода, определенного в родительском классе и замещенного в подклассе, приведет к динамическому связыванию с методом в соответствующем дочернем классе. В результате будет выполнен метод, описанный в классе объекта, на который ссылается переменная, а не в классе, который был задан при объявлении переменной. Рассмотрим следующий пример.

```
abstract class Pet
{
    abstract void voice();
}
class Dog extends Pet
{
    int k = 10;
    @Override
    void voice()
    {
        System.out.println("Gav-gav!");
    }
}
class Cat extends Pet
{
    @Override
    void  voice()
    {
        System.out.println("Miaou!");
    }
}
class Cow extends Pet
{
```

```
    @Override
    void  voice()
    {
        System.out.println("Mu-u-u!");
    }
}
public class Chorus
{
    public static void main(String[] args)
    {
        Pet[] singer = new Pet[3];
        singer[0] = new Dog();
        singer[1] = new Cat();
        singer[2] = new Cow();
        for (Pet p : singer)
        {
            p.voice();
        }
    }
}
```

Хотя массив ссылок `singer[]` имеет тип `Pet`, каждый его элемент ссылается на объект своего типа: `Dog`, `Cat`, `Cow`. При выполнении программы вызывается метод конкретного объекта, а не метод класса, которым определялось имя ссылки.

Абстрактные методы в языке Java объявляются с помощью модификатора `abstract`. Если класс содержит хоть один абстрактный метод, то создать его экземпляры нельзя, и он становится абстрактным (класс `Pet` в вышерассмотренном примере), что обязательно надо указать модификатором `abstract`. От абстрактных классов порождаются подклассы, в которых абстрактные методы переопределяются.

Пометив метод модификатором `final`, можно запретить его переопределение в подклассах. Если же пометить модификатором `final` весь класс, то его вообще нельзя будет расширить. Использование модификатора `final` в описании переменной указывает, что её значение нельзя изменить ни в подклассах, ни в самом классе. Так в языке Java определяются константы, например:

```
public final int MIN_VALUE = -1, MAX_VALUE = 100;
```

На самой вершине иерархии классов Java находится класс `Object`. Если при описании класса не указано никакое расширение, т.е. не записано слово `extends` и имя класса за ним, то Java считает этот класс расширением класса `Object`. В класс `Object` включено лишь несколько самых общих методов, например, метод `equals()`, сравнивающий данный объ-

ект на равенство с объектом, заданным в аргументе, и возвращающий логическое значение. Например:

```
Object obj1 = new Dog(), obj2 = new Cat();
if (obj1.equals(obj2)) …
```

Ссылки можно сравнивать на равенство и неравенство. В этом случае сопоставляются адреса объектов, т.е. можно узнать, не указывают ли две ссылки на один и тот же объект. Метод `equals()` же сравнивает содержимое объектов в их текущем состоянии. Фактически он реализован в классе `Object` как тождество: объект равен только самому себе. Поэтому его обычно переопределяют в подклассах.

Второй метод класса `Object`, часто требующий переопределения, — метод `hashCode()` — возвращает целое число, уникальное для каждого объекта данного класса, его идентификатор. Это число позволяет однозначно определить объект. Оно используется многими стандартными классами Java. Третий метод класса `Object`, который следует переопределять в подклассах, — метод `toString()`. Этот метод выражает содержимое объекта строкой символов и возвращает объект класса `String`. Метод `toString()` важен потому, что исполняющая система Java обращается к нему каждый раз, когда требуется представить объект в виде строки, например в методе `println()`.

При создании экземпляра класса с помощью операции `new` вызывается *конструктор класса*. Его задача состоит в инициализации состояния объекта, в том числе задании начальных значений его полей. Конструктор описывается как специальный метод, имя которого совпадает с именем класса. В классе может быть несколько конструкторов, отличающихся типом и/или количеством параметров.

Переменные класса образуются в Java модификатором `static`. К статическим переменным можно обращаться не только с именем экземпляра, но и с именем класса. Делать это можно, даже если не создан ни один экземпляр класса, т.к. поля класса определяются при загрузке файла с классом в оперативную память, еще до создания экземпляров класса. Для работы со статическими полями обычно создаются *статические методы*, отмеченные модификатором `static`. Такие методы работают напрямую только со статическими полями и методами класса. Статические методы могут выполняться, даже если не создан ни один экземпляр класса. Достаточно уточнить имя метода именем класса, чтобы выполнить его.

Пакеты. Все классы Java распределяются по *пакетам*. Кроме классов пакеты могут содержать интерфейсы и вложенные *подпакеты*. Таким образом образуется древовидная структура пакетов и подпакетов. Эта структура в точности отображается на структуру файловой системы. Все файлы с расширением class (содержащие байт-коды), образующие один пакет, хранятся в одном каталоге файловой системы. Подпакеты образуют

подкаталоги этого каталога. Каждый пакет создает своё *пространство имен*, т.е. все имена классов, интерфейсов и подпакетов в пакете должны быть уникальны. Имена в разных пакетах могут совпадать, но это будут разные программные единицы. Таким образом, ни один класс, интерфейс или подпакет не может оказаться сразу в двух пакетах. Если в одном месте программы надо использовать два класса с одинаковыми именами из разных пакетов, то имя класса уточняется именем пакета: `пакет.Класс`. Такое уточненное имя называется *полным именем класса*. Подпакет не является частью пакета, и классы, находящиеся в нем, не относятся к пакету, а только к подпакету. Поэтому, для того чтобы создать подпакет, не надо предварительно создавать пакет. С другой стороны, включение в программу пакета не означает включение его подпакетов.

Чтобы создать пакет, надо просто в первой строке java-файла с исходным кодом записать строку

`package <имя пакета>;`

например: `package ui;`

Тем самым создается пакет с указанным именем `ui` и все классы, записанные в этом файле, попадут в пакет `ui`. Повторив эту строку в начале каждого файла с исходным кодом, можно включить в пакет все необходимые классы. Имя подпакета уточняется именем пакета. Чтобы создать подпакет с именем, например, `window`, следует в первой строке исходного файла написать: `package ui.window;` и все классы этого файла и всех файлов с такой же первой строкой попадут в подпакет `window` пакета `ui`. Поскольку строка `package <имя пакета>;` только одна и это обязательно первая строка файла, каждый класс попадает только в один пакет или подпакет. Если для класса не указан пакет, то компилятор создает для него *безымянный пакет*, которому соответствует текущий каталог файловой системы. Однако рекомендуется все классы помещать в пакеты.

По умолчанию компилятор ищет классы только в двух пакетах: в том, что указан в первой строке файла, и в пакете стандартных классов `java.lang`. Для классов из других пакетов надо указывать полные имена, например `p1.Base`, `java.util.Date` и т.д. Кроме того, можно использовать оператор `import`, указывающий компилятору полные имена классов. Пишется слово `import` и через пробел полное имя класса, завершенное точкой с запятой. Импортировать разрешается только открытые классы, помеченные модификатором `public`. Используется также вторая форма оператора `import` — указывается имя пакета или подпакета, а вместо короткого имени класса ставится звездочка *. Этой записью компилятору предписывается просмотреть весь пакет. Например:

```
import p1.*;
import java.awt.geom.*;
```

Таким образом, исходный файл с текстом программы на языке Java имеет следующую структуру.

- В первой строке файла может быть необязательный оператор package.
- В следующих строках могут быть необязательные операторы import.
- Далее идут описания классов и интерфейсов.
- Среди классов может быть только один открытый public-класс.
- Имя файла должно совпадать с именем открытого класса, если последний существует.

Отсюда следует, что если в проекте есть несколько открытых классов, то они должны находиться в разных файлах. Рекомендуется открытый класс, если он имеется в файле, описывать первым. Кроме того, обычно исходный текст каждого класса записывается в отдельном файле.

Пакетами пользуются еще и для того, чтобы добавить к уже имеющимся правам доступа к членам класса private, protected и public еще один, «пакетный» уровень доступа. Если член класса не отмечен ни одним из модификаторов private, protected, public, то по умолчанию к нему осуществляется *пакетный доступ*, т.е. к такому члену может обратиться любой метод любого класса из того же пакета. Пакеты ограничивают доступ к классу целиком — если класс не помечен модификатором public, то все его члены, даже открытые (public), не будут видны из других пакетов. Члены с пакетным доступом не видны в подпакетах данного пакета.

Интерфейсы. Поскольку множественное наследование в Java запрещено, интерфейсы используются, чтобы обеспечить наследование свойств сразу от нескольких суперклассов. *Интерфейс*, в отличие от класса, содержит только константы и заголовки методов, без их реализации. Интерфейсы тоже размещаются в пакетах и подпакетах, часто в тех же самых, что и классы, и тоже компилируются в class-файлы. Описание интерфейса начинается со слова interface, перед которым может стоять модификатор public, означающий, как и для класса, что интерфейс доступен всюду. Если же этого модификатора нет, интерфейс будет виден только в своем пакете. После слова interface записывается имя интерфейса, потом может стоять слово extends и список интерфейсов-предков через запятую. Таким образом, одни интерфейсы могут порождаться от других интерфейсов, образуя свою, независимую от классов, иерархию, причем в ней допускается множественное наследование интерфейсов. В этой иерархии нет общего предка. Затем в фигурных скобках записываются в любом порядке константы и заголовки методов. Константы всегда статические, но слова static и final указывать не нужно. Все константы и методы в интерфейсах всегда открыты, поэтому модификатор public указывать не обязательно.

Реализация интерфейса — это класс, в котором расписываются методы одного или нескольких интерфейсов. В заголовке класса после его имени или после имени его суперкласса, если он есть, записывается слово

implements и, через запятую, перечисляются имена интерфейсов. Например:

```
interface Automobile {...}
interface Car extends Automobile {...}
interface Truck extends Automobile {...}
class Pickup implements Car, Truck {...}
```

Реализация интерфейса может быть неполной, т.е. некоторые методы могут быть расписаны, а другие — нет. Такая реализация — абстрактный класс, его обязательно надо пометить модификатором abstract.

На интерфейсы можно создавать ссылки. Указывать такая ссылка может только на какую-нибудь реализацию интерфейса. Благодаря этому возможен еще один способ организации полиморфизма, что демонстрирует следующий пример.

```
interface Voice
{
    void voice();
}
class Dog implements Voice
{
    public void voice()
    {
        System.out.println("Gav-gav!");
    }
}
class Cat implements Voice
{
    public void voice()
    {
        System.out.println("Miaou!");
    }
}
class Cow implements Voice
{
    public void voice()
    {
        System.out.println("Mu-u-u!");
    }
}
```

```
public class Chorus
{
    public static void main(String[] args)
    {
        Voice[] singer = new Voice[3];
        singer[0] = new Dog();
        singer[1] = new Cat();
        singer[2] = new Cow();
        for (Voice v : singer)
        {
            v.voice();
        }
    }
}
```

Лекция № 17.

ТЕМА: Методы защитного программирования.

Основные вопросы, рассматриваемые на лекции:

1. Понятие корректной и надежной программы.
2. Понятие защитного программирования.
3. Основные способы предупреждения ошибок.
4. Виды ошибок, влияющих на возникновение неверных данных и способы их нейтрализации.
5. Изоляция ошибок.
6. Способы обработки ошибок.
7. Исключения.
8. Утверждения.

Программа является *корректной*, если она удовлетворяет внешним спецификациям, т.е. выдает ожидаемые ответы на определенные комбинации значений входных данных. Программа является *надежной*, если она корректна, приемлемо реагирует на неточные входные данные и удовлетворительно функционирует в необычных условиях.

В связи с тем, что в процессе разработки программы обычно невозможно избежать всех ошибок, целесообразно включить в создаваемое программное обеспечение средства их обнаружения. Это позволяет минимизировать как влияние ошибки на работу программы, так и последующие затруднения для человека, которому придется извлекать информацию об

этой ошибке, находить её место и исправлять. После обнаружения ошибки либо она сама, либо её последствия должны быть исправлены программой, для чего могут использоваться различные методы. Таким образом, использование соответствующих методов и средств позволяет обеспечить функционирование программной системы при наличии в ней ошибок.

Программирование, при котором применяются специальные приемы предупреждения, раннего обнаружения и нейтрализации ошибок, называется защитным или программированием с защитой от ошибок. Эти приемы концентрируются на защите программ и их составляющих от неправильных входных данных, а также способах выявления, изоляции и обработки ошибок.

Основные концепции предупреждения возникновения ошибок в работающей программе включают в себя:

- проверку правильности входных данных и операций ввода-вывода;
- проверку допустимости промежуточных результатов;
- предотвращение накопления погрешностей.

Эти идеи направлены на контроль правильности исходных или промежуточных данных программы. Неверные данные могут появиться как в результате внутренней ошибки, например ошибки устройств ввода-вывода или ПО, так и в результате внешней ошибки, например ошибки пользователя или взаимодействующей программы.

Выделяют следующие виды ошибок, связанных с исходными данными:

- *Ошибки передачи* — вызываются аппаратными средствами, которые, например вследствие неисправности, могут искажать данные. Такие ошибки обычно контролируются аппаратно.
- *Ошибки преобразования* — возникают из-за того, что программа неверно преобразует данные из входного формата во внутренний рабочий. Для защиты от таких ошибок полученные в результате ввода данные обычно сразу демонстрируются пользователю (так называемый «эхо-вывод»). При этом сначала осуществляется преобразование во внутренний формат, а затем обратно. Однако предотвратить все ошибки преобразования, как правило, довольно сложно. Поэтому следует использовать другие методы защитного программирования, а также тщательно прорабатывать и тестировать соответствующие фрагменты программы.
- *Ошибки перезаписи* — обусловлены тем, что пользователь ошибается при вводе данных, например, вводит лишний или другой символ. Подобные ошибки можно обнаружить и устранить при использовании избыточных данных, например контрольных сумм, и специальных ограничений на значения и формат входных данных, например интервалов допустимых значений.
- *Ошибки данных* — вызваны тем, что пользователь вводит неверные данные. Такие ошибки обычно может обнаружить только пользова-

тель, поэтому имеет смысл в процессе ввода демонстрировать ему введенные данные и запрашивать подтверждение на выполнение дальнейших операций.

Один из методов защитного программирования состоит в изоляции ошибок, которые могут возникнуть в связи с неверными входными данными. С этой целью может быть разработан специализированный программный интерфейс (например, набор подпрограмм, обрабатывающих входные данные), используемый в качестве своеобразной «защитной» оболочки для остальной части кода. Проверка корректности данных и обработка соответствующих ошибок осуществляется на уровне этого интерфейса и пропущенные через него данные в дальнейшем считаются безопасными. Тот же подход применим и на уровне отдельного модуля или класса.

Проверка промежуточных результатов позволяет снизить вероятность позднего проявления не только ошибок неверно определенных данных, но и некоторых ошибок, допущенных на этапе проектирования и кодирования. Для организации такой проверки необходимо в программе использовать переменные, для которых существуют ограничения любого вида, например, связанные с особенностями предметной области. При этом следует учитывать то обстоятельство, что любые дополнительные операции требуют использования дополнительных ресурсов и могут также содержать ошибки. Поэтому проверку промежуточных результатов целесообразно выполнять только в тех случаях, когда это не сложно и действительно позволяет обнаружить ошибки.

Чтобы снизить погрешности результатов вычислений рекомендуется:

- избегать вычитания близких чисел (машинный ноль);
- избегать деления больших чисел на малые;
- начинать сложение длинной последовательности чисел с тех, которые имеют меньшее по модулю значение;
- не использовать условие равенства вещественных чисел;
- стремиться по возможности уменьшать количество операций;
- применять методы с известными оценками погрешностей;
- вычисление производить с двойной точностью, а результат выдавать с одинарной.

В зависимости от вида ошибки и обстоятельств может использоваться один или сразу несколько из следующих приемов обработки ошибок.

- *Использовать нейтральное значение*.
- *Заменить следующим корректным блоком данных*.
- *Вернуть тот же результат, что и в предыдущий раз*.
- *Использовать ближайшее допустимое значение*.
- *Записать предупреждающее сообщение в специальный файл*.
- *Вернуть код ошибки*.

- *Вызывать специальную подпрограмму или объект, предназначенный для обработки ошибок.*
- *Обработать ошибку в месте возникновения наиболее подходящим способом.*
- *Показать сообщение об ошибке, где бы она ни случилась.*
- *Прекратить выполнение.*

Выбор подходящих методов обработки ошибок зависит от типа программной системы и требований к ней. В одних случаях предпочтительнее продолжение работы, даже если это может привести к частично неверным результатам, а в других приоритетным является получение корректных данных. Однако, вне зависимости от выбранных методов, необходимо стараться придерживаться единых принципов обработки ошибок для всех частей программы.

Исключение или исключительная ситуация (exception) — это любое необычное событие, в том числе ошибочное, обнаруживаемое аппаратным или программным обеспечением, которое может потребовать особой обработки. Эта обработка выполняется программным фрагментом, называемым обработчиком исключительной ситуации (exception handler). Исключительная ситуация возбуждается (raised), когда происходит связанное с ней событие.

Исключения представляют собой специальное средство, позволяющее передать в вызывающий код сведения о возникших ошибках или исключительных ситуациях. Исключения генерируются в ответ на нештатные ситуации, которые не могут быть обработаны локально. В результате управление передается другой части системы, которая, возможно, способна интерпретировать ошибку и выполнить соответствующие ей осмысленные действия. Принцип действия исключений заключается в следующем. В программном фрагменте, в котором возникло нештатное событие, неявно или явно с помощью специального оператора (например, `raise` или `throw`) генерируется исключение. Это исключение перехватывается другим программным фрагментом, выполняющим его обработку. Во многих языках для организации обработки исключений используется конструкция `try-catch`. В этом случае конструкция `try` включает программный фрагмент, который может возбуждать исключения, а конструкция `catch` содержит операторы их обработки.

Чрезмерное и неверное применение исключений приводит к усложнению программ. Для того чтобы избежать этого и повысить эффективность использования исключений, следует придерживаться следующих правил:

- Используйте исключения для оповещения других частей программы об ошибках, которые нельзя игнорировать.

- Генерируйте исключения только для действительно нештатных ситуаций.
- Не используйте исключения для тех ситуаций, которые могут быть обработаны локально.
- Избегайте генерировать исключения в конструкторах и деструкторах классов.
- Вносите в описание исключения всю информацию о его причинах.
- Избегайте пустых обработчиков исключений.
- Выясните, какие исключения генерируют используемые подпрограммы сторонних библиотек.
- Рассмотрите вопрос о централизованном выводе информации об исключениях.
- Выработайте стандарты применительно к реализуемой системе по использование исключений в тех или иных ситуациях и частях программы.

Средства выявления ошибок на стадии разработки.

К ним относятся утверждения и другие отладочные средства, позволяющие ускорить нахождение ошибок в программе.

Утверждение (assertion) — это используемый в процессе разработки код (обычно специальная подпрограмма или макрос), посредством которого программа проверяет правильность своего выполнения. Если утверждение истинно, то программа работает так, как ожидается. В противном случае это означает, что в коде обнаружена ошибка. Обычно утверждение имеет два параметра: логическое выражение, описывающее предположение, которое должно быть истинным, и сообщение или оператор, выполняемый в противном случае. В ходе разработки программы утверждения позволяют выявить противоречивые допущения, непредвиденные условия, некорректные значения параметров и т.д. При подготовке версии системы, поставляемой заказчику, утверждения, как и другие отладочные средства, обычно удаляются из кода. Рекомендуется придерживаться следующих рекомендаций по применению утверждений.

- Используйте процедуры обработки ошибок для ожидаемых событий и утверждения для событий, которые происходить не должны.
- Старайтесь не размещать выполняемый код в утверждении.
- Используйте утверждения совместно с обработчиками ошибок.

Использование отладочных средств обычно предполагает расходование дополнительных ресурсов и может нарушать некоторые требования к ПО. В связи с этим в ходе разработки должно быть предусмотрено полное или частичное удаление или отключение соответствующих средств при сборке выходной версии системы.

Лекция № 18.

ТЕМА: Основные принципы и методы тестирования программ.

Основные вопросы, рассматриваемые на лекции:

1. Основные понятия и определения.
2. Базовые правила тестирования.

Тестирование — процесс выполнения программы с намерением найти ошибки. Если Ваша цель — показать отсутствие ошибок, Вы их найдете не слишком много. Если же Ваша цель — показать наличие ошибок, Вы найдете значительную их часть.

Доказательство — попытка найти ошибки в программе безотносительно к внешней для программы среде. Большинство методов доказательства предполагает формулировку утверждений о поведении программы и затем вывод и доказательство математических теорем о правильности программы. Доказательства могут рассматриваться как форма тестирования, хотя они и не предполагают прямого выполнения программы.

Контроль — попытка найти ошибки, выполняя программу в тестовой, или моделируемой, среде.

Испытание — попытка найти ошибки, выполняя программу в заданной реальной среде.

Аттестация — авторитетное подтверждение правильности программы. При тестировании с целью аттестации выполняется сравнение с некоторым заранее определенным стандартом.

Тестирование модуля, или автономное тестирование — контроль отдельного программного модуля, обычно в изолированной среде. Тестирование модуля иногда включает также математическое доказательство.

Тестирование сопряжений — контроль сопряжений между частями системы (модулями, компонентами, подсистемами).

Тестирование внешних функций — контроль внешнего поведения системы, определенного внешними спецификациями.

Комплексное тестирование — контроль и испытание системы по отношению к исходным целям. Комплексное тестирование является процессом испытания, если выполняется в реальной, жизненной среде.

Тестирование приемлемости — проверка соответствия программы требованиям пользователя.

Тестирование настройки — проверка соответствия каждого конкретного варианта установки системы с целью выявить любые ошибки, возникшие в процессе настройки системы.

Базовые правила тестирования.

1. Хорош тот тест, для которого высока вероятность обнаружить ошибку, а не тот, который демонстрирует правильную работу программы. Поскольку невозможно показать, что программа не имеет ошибок, процесс тестирования должен представлять собой попытки обнаружить в программе прежде не найденные ошибки.

Одна из самых сложных проблем при тестировании — решить, когда нужно закончить. Поскольку исчерпывающее тестирование (т.е. испытание всех входных значений) невозможно, то при тестировании возникает экономическая проблема: как выбрать конечное число тестов, которое дает максимальную отдачу (вероятность обнаружения ошибок) для данных затрат.

2. Недопустимо тестировать свою собственную программу. Тестирование должно быть в высшей степени разрушительным процессом, но имеются глубокие психологические причины, по которым программист не может относиться к своей программе как разрушитель.

3. Необходимая часть всякого теста — описание ожидаемых выходных данных или результатов. Одна из самых распространенных ошибок при тестировании состоит в том, что результаты каждого теста не прогнозируются до его выполнения. Ожидаемые результаты нужно определять заранее, чтобы не возникла ситуация, когда «глаз видит то, что хочет увидеть». Чтобы совсем исключить такую возможность, лучше разрабатывать самопроверяющиеся тесты, либо пользоваться инструментами тестирования, способными автоматически сверять ожидаемые и фактические результаты.

4. Избегайте невоспроизводимых тестов, не тестируйте «с лету». В условиях диалога программист слишком часто выполняет тестирование «с лету», т.е., сидя за терминалом, задает конкретные значения и выполняет программу, чтобы посмотреть, что получится. Это — неряшливая и нежелательная форма тестирования. Основной ее недостаток в том, что такие тесты мимолетны; они исчезают по окончании их выполнения. Никогда не используйте тестов, которые тут же выбрасываются.

5. Готовьте тесты как для правильных, так и для неправильных входных данных. Многие программисты ориентируются в своих тестах на «разумные» условия на входе, забывая о последствиях появления непредусмотренных или ошибочных входных данных. Однако многие ошибки, которые потом неожиданно обнаруживаются в работающих программах, проявляются вследствие никак не предусмотренных действий пользователя программы. Тесты, представляющие неожиданные или неправильные входные данные, часто лучше обнаруживают ошибки, чем «правильные» тесты.

6. Детально изучите результаты каждого теста. Самые изощренные тесты ничего не стоят, если их результаты удостаиваются лишь беглого взгляда. Тестирование программы означает большее, нежели выполнение

достаточного количества тестов; оно также предполагает изучение результатов каждого теста.

По мере того как число ошибок, обнаруженных в некоторой компоненте программного обеспечения увеличивается, растет также относительная вероятность существования в ней необнаруженных ошибок. Этот феномен наблюдался во многих системах. Его понимание способно повысить качество тестирования, обеспечивая обратную связь между результатами прогона тестов и их проектированием. Если конкретная часть системы окажется при тестировании полной ошибок, для нее следует подготовить дополнительные тесты.

7. Поручайте тестирование специалистам в этой области или, при невозможности их привлечения, самым способным программистам. Тестирование, и в особенности проектирование тестов, — этап в разработке программного обеспечения, требующий особенно творческого подхода. Во многих организациях тестирование часто считают рутинной, нетворческой работой. Однако практика показывает, что проектирование тестов требует даже больше творчества, чем разработка архитектуры и проектирование программного обеспечения.

8. Считайте тестируемость одной из ключевых задач разработки. Сложность тестирования программы напрямую зависит от её структуры и качества проектирования. Несмотря на то, что эта связь осознана еще недостаточно глубоко, можно утверждать, что многие характеристики хорошего проекта (например, небольшие, в значительной степени независимые модули и независимые подсистемы), улучшают и тестируемость программы. Учет этого фактора, а также планирование и проектирование тестов начиная с ранних этапов разработки, обычно позволяет улучшить конечные результаты.

9. Никогда не изменяйте программу, чтобы облегчить её тестируемость. Часто возникает соблазн изменить программу, чтобы было легче ее тестировать. Например, программист, тестируя модуль, содержащий цикл, который должен повторяться 100 раз, меняет его так, чтобы цикл повторялся только 10 раз.

Лекция № 19.

ТЕМА: Основные принципы и методы отладки программ.

Основные вопросы, рассматриваемые на лекции:

1. Определение отладки.
2. Рекомендуемый подход к процессу отладки.
3. Основные принципы локализации и исправления ошибок.
4. Изучение результатов процесса отладки.

Отладка не является разновидностью тестирования. Хотя слова «отладка» и «тестирование» часто используются как синонимы, под ними подразумеваются разные виды деятельности. Тестирование — деятельность, направленная на обнаружение ошибок; отладка направлена на установление точной природы известной ошибки, а затем на исправление этой ошибки. Эти два вида деятельности связаны: результаты тестирования являются исходными данными для отладки.

Рекомендуемый подход к методам отладки включает в себя следующие этапы:

1. Поймите задачу. Многие программисты начинают процесс отладки бессистемно, пропуская жизненно важный этап детального анализа имеющихся данных. Первым делом нужно тщательно исследовать, что в программе выполнено правильно, а что неправильно, чтобы выработать одну или несколько гипотез о природе ошибки. Одна из самых распространенных причин затруднений при отладке — не учтен какой-нибудь существенный фактор в выходных данных программы. Важно исследовать данные в поисках противоречий гипотезе (например, ошибка возникает только в каждой второй записи), потому что это поведет к уточнению гипотезы или, возможно, покажет, что имеется не одна причина ошибки.

2. Разработайте план. Следующий шаг — построить одну или несколько гипотез об ошибке и разработать план проверки этих гипотез.

3. Выполните план. Следуя своему плану, пытайтесь доказать гипотезу. Если план включает несколько шагов, нужно проверить каждый.

4. Проверьте решение. Если кажется, что точное местоположение ошибки обнаружено, необходимо выполнить еще несколько проверок, прежде чем пытаться исправить ошибку. Проанализируйте, может ли предполагаемая ошибка давать в точности известные симптомы. Убедитесь, что найденная причина полностью объясняет все симптомы, а не только их часть. Проверьте, не вызовет ли ее исправление новой ошибки.

Главная причина затруднений при отладке — такая психологическая установка, когда разум видит то, что он ожидает увидеть, а совсем не то,

что имеет место в действительности. Один из способов преодоления такой установки — скептицизм в отношении всего, что Вы изучаете, в особенности комментариев и документации. Опытные специалисты по отладке, изучая модуль, часто закрывают комментарии, поскольку комментарии нередко описывают, что программа делает, по мнению её создателя. Обратный просмотр, т.е. чтение программы в обратном направлении — еще один полезный тактический прием, поскольку он помогает по-новому взглянуть на алгоритм.

Еще одна трудность при отладке — такое состояние, когда все идеи зашли в тупик и найти местоположение ошибки кажется просто невозможно. Это означает, что Вы либо смотрите не туда, куда нужно, и следует еще раз изучить симптомы и построить новую гипотезу, либо подозрения правильные, но разум уже не способен заметить ошибку. Если кажется, что именно так и есть, то лучший принцип — «утро вечера мудренее». Переключите внимание на другую деятельность, и пусть над задачей работает Ваше подсознание.

Когда Вы найдете и проверите ошибку и убедитесь в том, что нашли ее правильно, не забудьте о том, что вероятность других ошибок в этой части программы теперь выше. Изучите программу в окрестности найденной ошибки в поисках новых неприятностей. Проверьте, не была ли сделана такая же ошибка в других местах программы.

Исследования методов отладки вначале концентрировались на сравнении отладки в пакетном и диалоговом режимах, причем большинство исследований приходило к выводу, что диалоговый режим предпочтительнее. Однако более поздняя работа показала, что, вероятно, наилучший способ отладки — просто читать программу и изо всех сил стараться вникнуть в алгоритм, хотя это требует усердия и собранности.

Важно подчеркнуть, что многие из методов разработки помогают и в процессе отладки. Такие методы, как структурное программирование и хороший стиль программирования не только уменьшают исходное количество ошибок, но и облегчают отладку, делая программу более простой для понимания.

После того, как точно установлено, где находится ошибка, надо ее исправить. Самая большая трудность на этом шаге — суметь охватить проблему целиком; самая распространенная неприятность — устранить только некоторые симптомы ошибки. Избегайте «экспериментальных» исправлений; они показывают, что Вы еще недостаточно подготовлены к отладке этой программы, поскольку не понимаете её.

В деле исправления ошибок очень важно понимать, что оно возвращает нас назад, к стадии проектирования. Обидно, если после завершения хорошо организованного проектирования весь его строгий порядок нарушается, когда вносятся поправки. Исправления должны выполняться, по крайней мере, так же строго, как первоначальное выполнение программы.

Если необходимо, следует обновить документацию, поправки должны проходить сквозной структурный контроль или другие формы контрольного чтения программы. Ни одна поправка не «мала» настолько, чтобы не нуждаться в тестировании.

По самой своей природе исправления всегда имеют некоторое отрицательное влияние на структуру программы и легкость ее чтения. Тот факт, что они делаются в условиях жесткого давления, усиливает это влияние. Опыт показывает, что при исправлении довольно высока вероятность внесения в программу новой ошибки (обычно от 20 до 50%). Из этого следует, что отладка должна выполняться лучшими программистами проекта.

Изучение результатов процесса отладки.

Один из лучших способов повысить надежность программного обеспечения в нынешних или в будущих проектах — очевидный, но часто упускаемый из виду процесс обучения на сделанных ошибках. Каждую ошибку следует внимательно изучить, чтобы понять, почему она возникла и что должно было бы сделано, чтобы ее предотвратить или обнаружить раньше. Редко можно встретить программиста или организацию, которые выполняли бы такой полезный анализ, а когда он проводится, то обычно имеет поверхностный характер и сводится, например, к классификации ошибок: ошибки проектирования, логические ошибки, ошибки сопряжения или другие, не имеющие особого смысла категории.

Нужно уделять время изучению природы каждой обнаруженной ошибки. Необходимо подчеркнуть, что анализ ошибок должен быть в значительной мере качественным и не сводиться просто к упражнению в количественном подсчете. Чтобы понять причины, лежащие в основе ошибок, и усовершенствовать процессы проектирования и тестирования, нужно ответить на следующие вопросы:

1. Почему возникла именно такая ошибка? В ответе должны быть указаны как первоисточник, так и непосредственный источник ошибки. Например, ошибка могла быть сделана при программировании конкретного модуля, но в её основе могла лежать неоднозначность в спецификациях или исправление другой ошибки.

2. Как и когда ошибка была обнаружена? Поскольку мы только что добились значительного успеха, почему бы нам не воспользоваться приобретенным опытом?

3. Почему эта ошибка не была обнаружена при проектировании, контроле или на предыдущей фазе тестирования?

4. Что следовало сделать в процессе разработки, чтобы предупредить появление этой ошибки или обнаружить ее раньше?

Собирать эту информацию нужно не только для того, чтобы учиться на ошибках. Официально отчетность об ошибках и об их исправлении необходима и для того, чтобы гарантировать, что обнаруженные ошибки в

работающих или тестируемых системах не упущены и что исправления выполнены в соответствии с принятыми нормами.

Лекция № 20.
ТЕМА: Классические технологические процессы

Основные вопросы, рассматриваемые на лекции:

1. Последовательность технологических процессов.
2. Технологические процессы по стандарту ISO/IEC 12207.

Классический набор, сложившийся исторически в результате практического опыта разработки программ, включает в себя следующие основные процессы.

- *Возникновение и исследование идеи*. Этот процесс подразумевает следующие действия: собственно возникновение и первичное исследование идеи, носящее максимально творческий и неформальный характер; детальное исследование идеи, в результате которого происходит выработка концепции и постановка задачи; анализ рисков и экспертизу идеи, по результатам которой принимается решение о начале работы над проектом и выполнении планирования.
- *Управление*. Этот процесс длится почти весь жизненный цикл программного обеспечения. Управление проектом представляет собой деятельность, направленную на его реализацию с максимально возможной эффективностью при заданных ограничениях на материальные, временны*е*, трудовые и другие ресурсы, а также качество конечных результатов проекта. Одним из важнейших действий управления является планирование, которое начинается после принятия решения о разработке. Кроме того, управление подразумевает следующие, не менее важные действия: распределение работ, организацию и управление коллективом разработчиков, решение финансовых вопросов и управление бюджетом проекта, руководство проектом и контроль его выполнения, документирование различных аспектов развития проекта.
- *Анализ требований и проектирование*. Анализ и проектирование являются близкими и взаимосвязанными процессами. Они ориентированы на решение общей задачи, результатом которой должно стать четкое представление о системе, на основе которого будет создан программный код. *Анализ требований* — процесс жизненного цикла программы, во время которого требования заказчика уточняются, формализуются и документируются. Основной вопрос, решаемый на данном этапе — «Что должна делать будущая система?». *Проектирование* — процесс жизненного цикла программы, во время которого исследуются и определяются её будущие структура

и взаимосвязи элементов. Основной вопрос, который решается на этом этапе — «Как система будет удовлетворять заданным требованиям?». Проектирование, как правило, подразумевает под собой разработку архитектуры программной системы и детальное проектирование составляющих её модулей. Результатом анализа и проектирования должен стать проект, содержащий достаточные сведения для реализации системы на его основе.

- *Программирование (реализация)*. Данный процесс предназначен для получения программного кода создаваемого продукта в соответствии с планом и проектом разработки.
- *Тестирование и отладка*. Эти процессы взаимосвязаны и служат, соответственно, для выявления и устранения ошибок и неточностей в работе программного обеспечения с целью достижения требуемых показателей надежности и качества.
- *Ввод программы в эксплуатацию*. Данный процесс выполняется после принятия решения о передаче результатов разработки заказчику или потребителю. Осуществляемые при этом действия в значительной мере определяются тем, была ли разработка рассчитана на конкретного заказчика или же созданный программный продукт ориентирован на широкого потребителя. В первом случае обычно требуется выполнить установку ПО на оборудовании заказчика и подготовку персонала. Выполняемые во втором случае действия существенно зависят от класса и особенностей программного продукта, его маркетинга и специфики разработчика.
- *Сопровождение*. Представляет собой действия по повышению надежности программного продукта после ввода в эксплуатацию и разработку усовершенствованных версий. На этапе сопровождения решаются следующие основные классы взаимосвязанных задач: *адаптация*, обычно заключающаяся в модификации функций программы; *усовершенствование*, как правило, состоящее в добавлении новых функций; *исправление* обнаруженных в ходе эксплуатации ошибок; *предупреждение* проблем, которые могут возникнуть в будущем. Для решения этих задач может быть выбран один из следующих типов сопровождения, под которым понимается степень вмешательства в программу: *незначительные* (*локальные*) изменения; *реструктурирование кода* — повторная разработка небольшой части программы при сохранении неизменным интерфейса с остальной частью; *реинжиниринг* — перестройка существующего программного продукта; *программирование заново*.
- *Завершение эксплуатации*. Данный процесс несложен, но часто требует организационной подготовки. Обычно он начинается с того, что пользователи заранее оповещаются о прекращении сопровождения программного продукта. Завершение эксплуатации часто является результатом того, что оказывается невозможным решить одну из задач сопровождения.

Технологические процессы по стандарту ISO/IEC 12207:1995

Международный стандарт ISO/IEC 12207:1995 “Information Technology — Software Life Cycle Processes” («Информационные технологии — Процессы жизненного цикла программного обеспечения») описывает структуру жизненного цикла программного обеспечения и его процессы. Процессы делятся на три группы — основные, организационные и вспомогательные (рис.1). Классические технологические процессы фактически являются подмножеством стандартного набора, выступая там как процессы или действия процессов.

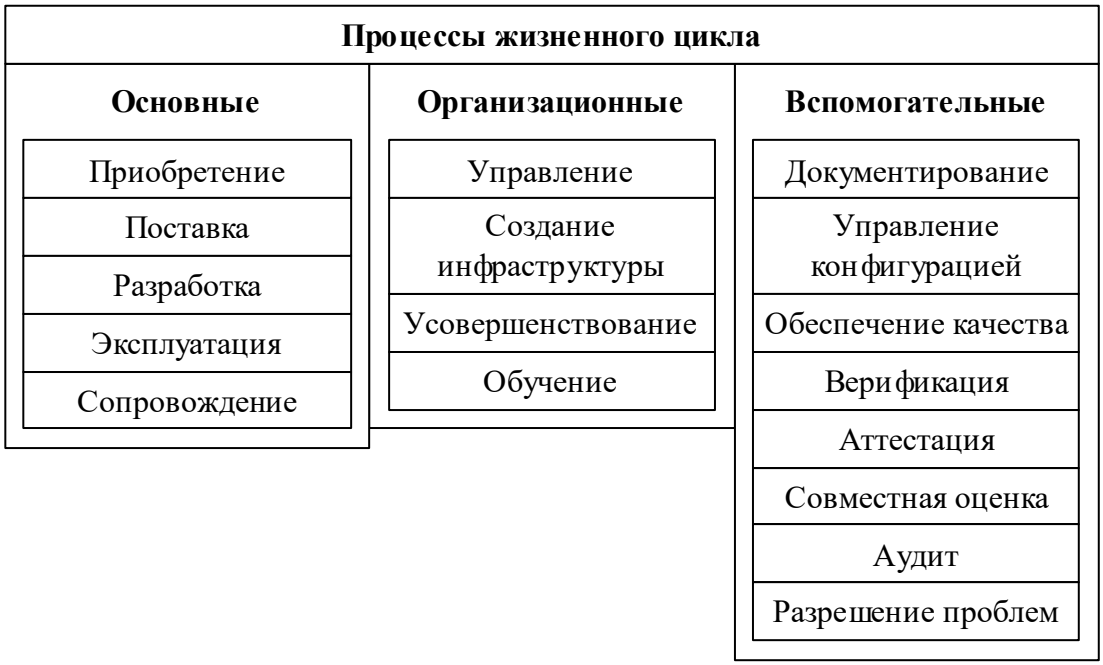

Рис.1. Структура процессов жизненного цикла ПО.

Приобретение состоит из действий заказчика и включает в себя инициирование приобретения, подготовку заявочных предложений, подготовку и корректировку договора, надзор за деятельностью поставщика, приемку и завершение работ.

Поставка охватывает действия, выполняемые поставщиком программного продукта или услуги, а именно: инициирование поставки, подготовку ответа на заявочные действия, подготовку договора, планирование, выполнение и контроль, проверку и оценку, поставку и завершение работ.

Разработка предусматривает работы по созданию ПО и его компонентов в соответствии с заданными требованиями, выполняемые разработчиком. Процесс включает следующие действия: подготовительная работа, анализ требований к системе, проектирование архитектуры системы, анализ требований к ПО, проектирование архитектуры ПО, детальное проектирование ПО, кодирование и тестирование ПО, интеграция ПО, квалифи-

кационное тестирование ПО, интеграция системы, квалификационное тестирование системы, установка ПО, приемка ПО.

Эксплуатация охватывает действия эксплуатирующей программный продукт организации, в число которых входят подготовительная работа, эксплуатационное тестирование и поддержка пользователей.

Сопровождение включает действия, выполняемые службой сопровождения, к которым относятся подготовительная работа, анализ проблем и запросов на модификацию ПО, модификация ПО, проверка и приемка, перенос ПО в другую операционную среду, снятие ПО с эксплуатации.

Управление состоит из действий, которые могут выполняться любой стороной, управляющей своими процессами. Эта сторона отвечает за управление проектом, а также действиями и задачами стандартных процессов. Процесс включает следующие действия: инициирование и определение области управления, планирование, выполнение и контроль, проверку и оценку, завершение.

Создание инфраструктуры охватывает выбор и поддержку технологий, стандартов и инструментальных средств, используемых в ходе разработки, эксплуатации и сопровождения ПО. Инфраструктуру следует модифицировать и сопровождать в соответствии с изменениями требований к соответствующим процессам. Данный процесс предполагает выполнение трех действий: подготовительной работы, создания и сопровождения инфраструктуры.

Усовершенствование определяет оценку, измерение, контроль и усовершенствование процессов жизненного цикла и включает действия по созданию, оценке и усовершенствованию процесса.

Обучение предполагает первоначальное обучение и последующее постоянное повышение квалификации персонала. Процесс предусматривает осуществление подготовительной работы, разработку учебных материалов и реализацию плана обучения.

Документирование предусматривает формализованное описание информации, получаемой в течение жизненного цикла ПО. Данный процесс состоит из набора действий, с помощью которых планируют, проектируют, разрабатывают, выпускают, редактируют, распространяют и сопровождают документы, необходимые для всех заинтересованных лиц, таких как руководство, технические специалисты, пользователи системы и т.д. Процесс включает подготовительную работу, проектирование и разработку, выпуск и сопровождение документации.

Управление конфигурацией предполагает применение административных и технических процедур на всем протяжении жизненного цикла ПО и служит для:

- определения состояния компонентов ПО в системе;
- управления модификациями ПО;

- описания и подготовки отчетов о состоянии компонентов ПО и запросов на модификацию, обеспечение полноты, совместимости и корректности компонентов;
- управления хранением и поставкой ПО.

Процесс подразумевает выполнение следующих действий: подготовительная работа, идентификация конфигурации, контроль конфигурации, учет состояния конфигурации, оценка конфигурации, управление выпуском и поставкой.

Процесс *обеспечения качества* предназначен для обеспечения гарантий того, что ПО и процессы его жизненного цикла соответствуют заданным требованиям и утвержденным планам, и включает в себя подготовительную работу, обеспечение качества продукта, процесса и прочих показателей качества системы.

Верификация представляет собой подтверждение того, что программные продукты, являющиеся результатом некоторого действия, полностью удовлетворяют требованиям или условиям, определяемым предыдущими действиями. Процесс включает два действия — подготовительную работу и собственно верификацию.

Аттестация служит для определения полноты соответствия заданных требований и созданной системы или программного продукта их конкретному функциональному назначению. Процесс подразумевает выполнение подготовительной работы и непосредственно аттестации.

Совместная оценка предназначена для определения состояния работ по проекту и ПО, создаваемого в ходе их выполнения, и включает в себя подготовительную работу, оценку управления проектом и техническую оценку.

Аудит представляет собой независимую проверку соответствия программного продукта требованиям, планам и условиям договора. Процесс подразумевает под собой подготовительную работу и собственно аудиторскую проверку.

Разрешение проблем предусматривает анализ и решение проблем, которые обнаружены в ходе основных процессов, независимо от их происхождения или источника. Любую обнаруженную проблему следует идентифицировать, описать, проанализировать и разрешить.

Лекция № 21.
ТЕМА: Основные технологические стадии

Основные вопросы, рассматриваемые на лекции:

1. Технологические стадии.
2. Технологические этапы.

3. Понятие версии программного продукта.

Технологические стадии выделяются исходя из соображений разумного и рационального планирования и организации работ. Как правило, названия стадий отражают наименования классических процессов (или их подмножества или надмножества), большая часть времени которых приходится на данную стадию. Практически во всех технологических подходах присутствуют следующие стадии:

- возникновение и исследование идеи;
- планирование;
- анализ требований;
- проектирование;
- программирование (реализация);
- тестирование и отладка;
- ввод в эксплуатацию;
- сопровождение;
- завершение эксплуатации.

Один из вариантов более детального разделения на стадии и этапы может иметь следующий вид.

1. Стадия возникновения и исследования идеи.
   1.1. Исследование идеи.
   1.2. Выработка концепции и постановка задачи.
   1.3. Принятие решения о начале работы над проектом.
2. Стадия планирования проекта.
   2.1. Подготовка плана проекта.
   2.2. Техническое рецензирование проекта.
   2.3. Начало формирования команды разработчиков.
   2.4. Обзор планов инженеров.
   2.5. Исследование и поиск необходимых ресурсов.
   2.6. Достижение соглашения о выделении ресурсов.
   2.7. Утверждение проекта.
3. Стадия анализа и проектирования.
   3.1. Анализ и проектирование продукта.
   3.2. Планирование разработки тестов и документации.
4. Стадия реализации.
   4.1. Кодирование, разработка тестов и документации.
   4.2. Интеграция кодов, тестов и документации.
5. Стадия версии разработчика.
   5.1. Объединение компонентов, составляющих проект, в единую систему.
   5.2. Стабилизация версии.
   5.3. Тестирование.

5.4. Распространение версии.
6. Стадия оценки жизнеспособности продукта.
7. Стадия альфа-версии.
   7.1. Объединение компонентов, составляющих проект.
   7.2. Стабилизация версии.
   7.3. Формальное начало выпуска версии.
   7.4. Распространение версии.
   7.5. Продолжение разработки.
   7.6. Формальное тестирование.
   7.7. Начало сопротивления изменениям.
8. Стадия бета-версии.
   8.1. Объединение компонентов, составляющих проект.
   8.2. Стабилизация версии.
   8.3. Создание электронного образа версии поставки (дистрибутива).
   8.4. Формальное тестирование версии.
   8.5. Начало процесса распространения версии.
9. Стадия версии первой поставки пользователю (выходной версии).
   9.1. Принятие решения о выпуске выходной версии.
   9.2. Объединение компонентов, составляющих проект.
   9.3. Стабилизация версии.
   9.4. Создание дистрибутива.
   9.5. Формальное тестирование версии.
   9.6. Начало процесса распространения версии.
   9.7. Завершающий анализ работы над версией.

Каждый из этапов, в свою очередь, может быть детализирован и разделен на более мелкие временны*е* промежутки. При планировании процесса разработки определяются *контрольные точки* (*отметки*), символизирующие окончание определенного этапа или промежутка работ. Для каждой контрольной точки создается отчет, на основании которого руководство проекта может выполнить текущую оценку и корректировку процесса и планов его развития. Кроме того, обычно по завершению основных стадий и этапов заказчику ПО предоставляются результаты их выполнения, так называемые *контрольные проектные элементы*. Это может быть документация, отдельные законченные подсистемы программного продукта, прототип всего продукта и т.д.

Версии программного продукта

Версией программного продукта (системы) называется экземпляр, имеющий определенные отличия от других экземпляров этого же продукта. Новые версии могут отличаться функциональными возможностями, эффективностью или исправлением ошибок. Некоторые версии могут иметь одинаковую функциональность, но быть разработанными под различные конфигурации аппаратного или программного обеспечения. Если

отличия между версиями незначительны, то они называются *вариантами* одной системы.

- *Версия разработчика* является ранней версией для внутреннего использования и тестирования.
- На этапе *альфа-версии* продукт должен быть функционально полным, все фрагменты должны быть правильно интегрированы в систему. Обычно на этом этапе формулируется список специальных критериев, которым должен удовлетворять продукт. Например, должны быть исправлены все высокоприоритетные ошибки. Запрещается внесение новых свойств и изменение интерфейса. Должны быть готовы материалы для проведения генерального тестирования и документация. Как правило, ПО на этом этапе не передается для внешнего тестирования.
- На этапе *бета-версии* продукт также должен удовлетворять специальным критериям. Например, должны быть исправлены все известные ошибки, имеющие высокий и средний приоритет. Продукт должен быть готов к внешнему тестированию и содержать инсталляционную программу.
- *Выходная версия* — эта версия, поставляемая заказчику. Она должна удовлетворять установленным критериям качества (например, полное отсутствие существенных ошибок) и быть готовой к поставке.

Лекция № 22.
ТЕМА: Классификация технологических подходов

Основные вопросы, рассматриваемые на лекции:

1. Технологические подходы.
2. Технологические этапы.
3. Понятие версии программного продукта.

Технологические подходы к созданию ПО можно разделить на три основные группы: подходы со слабой формализацией, строгие и гибкие подходы (рис.1). В подходах со слабой формализацией не используются явные технологии и их можно применять только для очень маленьких проектов, как правило, завершающихся созданием демонстрационного прототипа. К таким подходам относятся так называемые ранние технологические подходы, в частности подход «кодирование и исправление». Строгие (классические, жесткие, предсказуемые) подходы рекомендуется применять для средних, крупномасштабных и гигантских проектов с фиксированными требованиями и объемом работ, а также достаточно большим коллективом разработчиков разной квалификации. В подходах данной группы весь объем предстоящих работ прогнозируется и упорядочивается,

чтобы обеспечить предсказуемость, требуемую проектом. Гибкие (адаптивные, легкие) подходы рекомендуется применять для небольших или средних проектов в случае неясных или изменяющихся требований к системе и малочисленном коллективе разработчиков. Для успешного применения этих подходов команда разработчиков должна быть ответственной и высококвалифицированной, а заказчики должны быть достаточно грамотны и согласны принимать участие в процессе разработки.

Рис.1. Классификация технологических подходов к созданию ПО.

Ранние технологические подходы

Одним из представителей данной группы является подход *«кодирование и исправление»* (code and fix). Его суть состоит в следующем. Разработчик начинает кодирование системы с самого первого дня, не занимаясь

сколь-либо серьезным проектированием. Большинство ошибок обнаруживается, как правило, к концу кодирования и требует исправления через повторное кодирование. При использовании данного подхода затрачивается время лишь на кодирование и отладку, поэтому заказчику легко демонстрировать прогресс в разработке в объеме кода. Этот подход можно рекомендовать к использованию для очень маленьких проектов, которые должны завершиться разработкой демонстрационного прототипа, и для доказательства жизнеспособности некоторой программной концепции.

Каскадные технологические подходы

Подходы этой категории задают некоторую последовательность выполнения процессов, которую можно представить в виде каскада или водопада. Поэтому эти подходы также иногда называются подходами на основе водопадной модели.

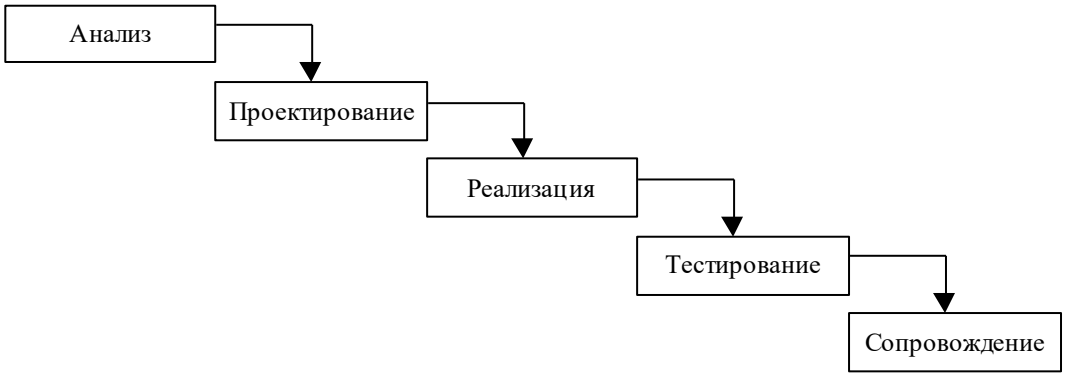

Рис.2. Каскадный технологический подход

Каскадный подход (pure waterfall) является одним из старейших подходов к разработке ПО. Он сформировался в 1970-80-х годах и послужил отправной точкой для развития большого числа других подходов. Специфика «чистого» каскадного подхода состоит в том, что переход к следующему процессу осуществляется только после того, как завершена работа с текущим процессом. Возвраты к уже пройденным процессам не предусмотрены (рис.2). Данный подход может быть успешно использован в тех проектах, где в самом начале могут быть точно и полно сформулированы все требования, например, в задачах вычислительного и исследовательского характера. Главное достоинство подхода заключается в том, что достаточно просто вести планирование работ и формирование бюджета. Основной недостаток каскадного подхода состоит в отсутствии гибкости, поскольку процесс создания ПО разделяется на ряд фиксированных этапов и не предполагает отклонений от стандартной последовательности шагов. В целом необходимость возвратов к предыдущим стадиям обусловлена следующими причинами:

- неточные или изменяющиеся в процессе разработки требования и спецификации, модификация которых может привести к необходимости пересмотра уже принятых решений;

- быстрое моральное устаревание используемых технических и программных средств;
- отсутствие удовлетворительных средств поддержки разработки на стадиях постановки задачи, анализа и проектирования.

Игнорирование указанных факторов часто приводит к тому, что созданный программный продукт не удовлетворяет требованиям заказчика или морально устарел, а также к увеличению времени и стоимости разработки.

Каскадно-возвратный подход позволяет преодолеть основной недостаток каскадного подхода, поскольку в нем предусмотрены возвраты к предыдущим стадиям и возможность пересмотра и уточнения ранее принятых решений (рис.3). Данный подход отражает итерационный характер создания ПО и подразумевает контроль, выполняемый после завершения каждого этапа и позволяющий при необходимости вернуться к любому предыдущему процессу и внести требуемые изменения. Каскадно-возвратный подход учитывает важные особенности реального процесса разработки, в том числе и существенные задержки с достижением результата, к которым могут привести корректировки при возвратах. Основная опасность использования данного подхода связана с тем, что процесс разработки может существенно затянуться или вообще никогда не будет завершен, постоянно находясь в состоянии уточнения и усовершенствования.

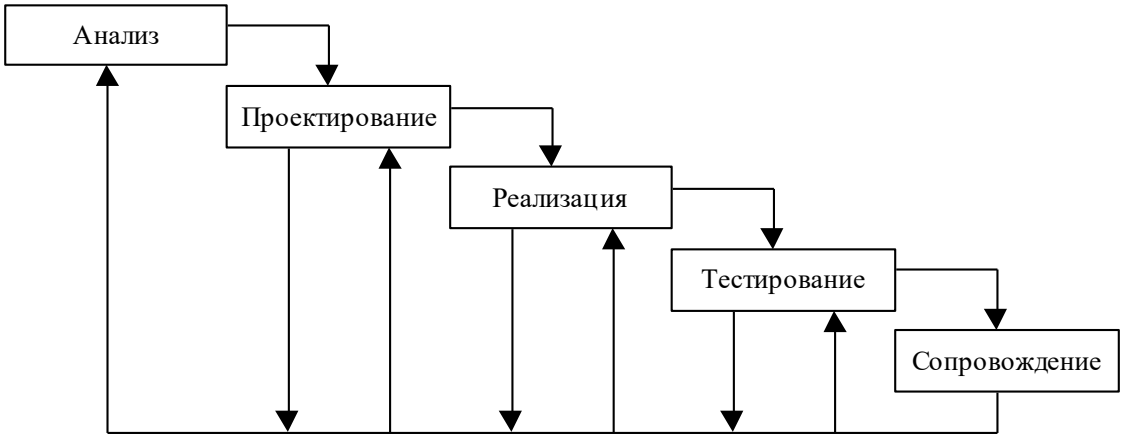

Рис.3. Каскадно-возвратный технологический подход

Каскадно-итерационный подход предусматривает последовательное итерации каждого процесса до тех пор, пока не будет достигнут желаемый результат (рис.4.). Каждая итерация является завершенным этапом и её итогом должен служить некоторый конкретный результат, который, однако, может быть промежуточным, не реализующим все необходимые требования.

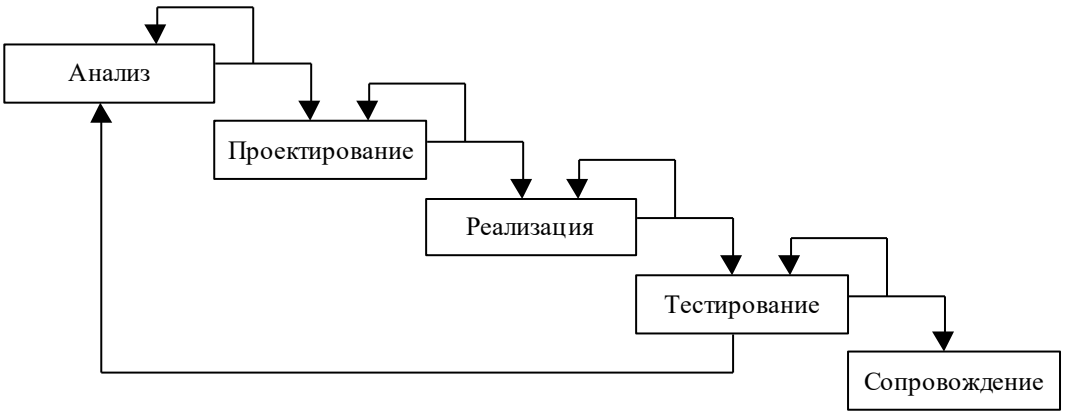

Рис.4. Каскадно-итерационный технологический подход

Каскадный подход с перекрывающимися процессами (waterfall with overlapping) подразумевает, что следующий процесс начинается до завершения текущего (рис.5.). Кроме того, несколько процессов могут выполняться параллельно. Это обеспечивает большую гибкость и позволяет начинать и выполнять отдельные работы, не дожидаясь, пока будут завершены предыдущие этапы. Как следствие может упроститься внесение изменений, поскольку, когда обнаруживается их необходимость, ранние процессы могут еще находиться в работе. Также данный подход позволяет раньше получать видимые результаты и демонстрировать их заказчику. Кроме того, в ряде случаев можно уменьшить количество рабочей документации по сравнению с классическим каскадным подходом, если отдельные разработчики задействованы сразу на нескольких стадиях. Вместе с тем усложняется контроль за процессом разработки, могут возникнуть проблемы из-за непредвиденных взаимозависимостей этапов и сложности во взаимодействии между разработчиками.

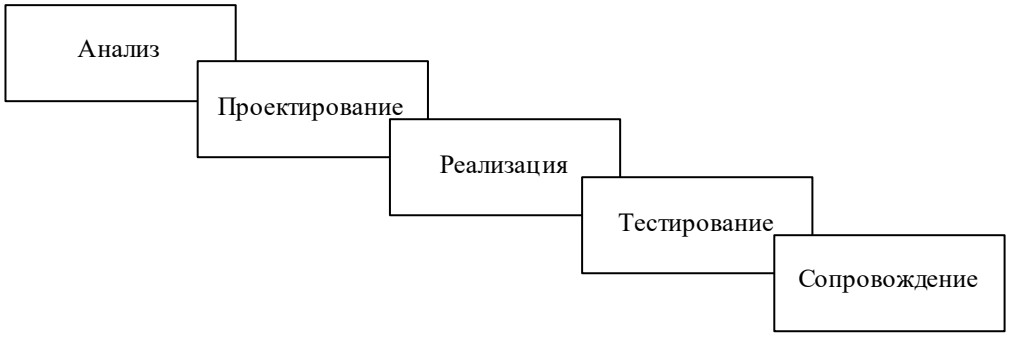

Рис.5. Каскадный подход с перекрывающимися процессами

Каскадный подход с подпроцессами (waterfall with subprocesses) имеет много общего с подходом с перекрывающимися процессами (рис.6.). Его особенность заключается в том, что проект разделяется на подпроекты, соответствующие подсистемам, которые могут разрабатываться отдельно и

параллельно. В связи с этим в данном подходе требуется дополнительный этап тестирования подсистем до объединения их в общую систему.

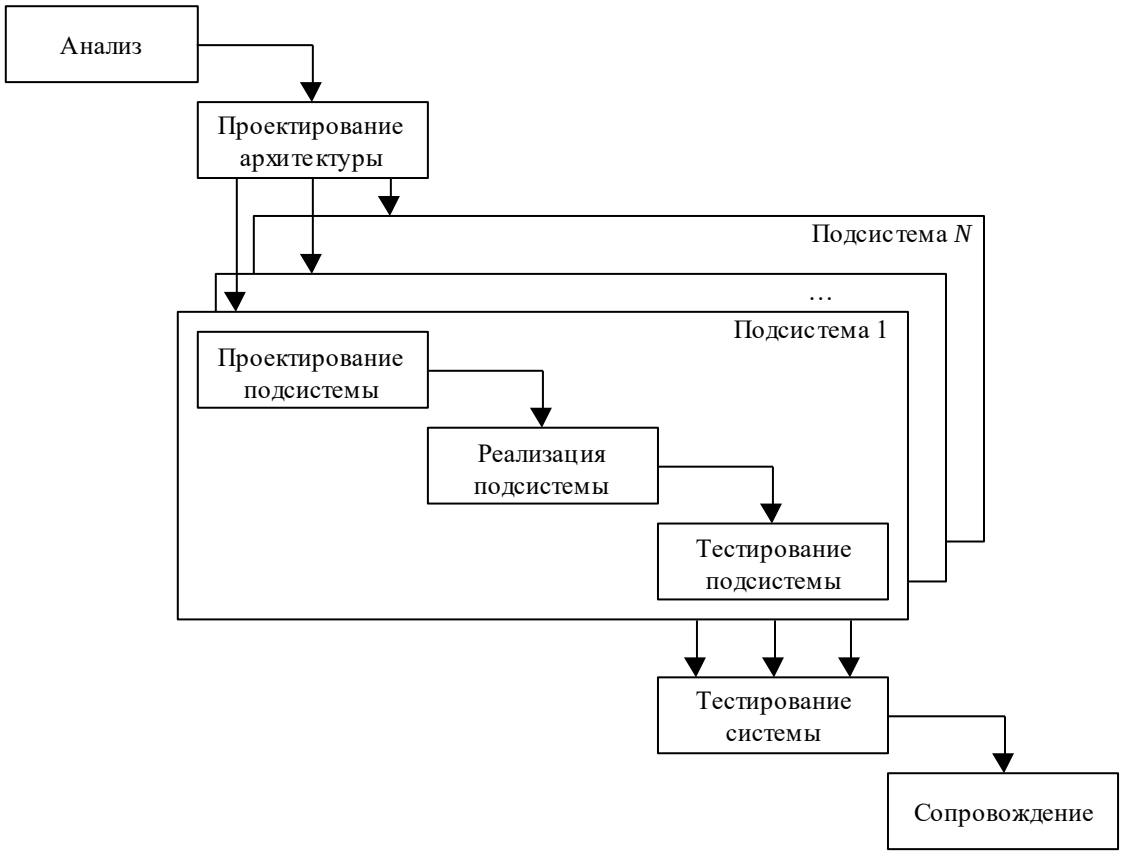

Рис.6. Каскадный подход с подпроцессами

Спиральная модель (spiral model) была предложена с целью сократить возможный риск разработки и фактически являлась реакцией на устаревание каскадной модели. В соответствии с данным подходом ПО создается не сразу, а итерационно путем разработки ряда прототипов. Под прототипом в данном случае подразумевается действующий программный продукт, реализующий отдельные функции и внешние интерфейсы создаваемой системы. Создание прототипов осуществляется за несколько витков спирали, каждый из которых включает в себя следующие действия (рис.):

- Планирование — определение целей, вариантов и ограничений.
- Анализ риска — анализ вариантов и определение/выбор риска.
- Конструирование — разработка и тестирование прототипа продукта следующего уровня.
- Оценивание — оценка заказчиком текущих результатов и внесение необходимых изменений.

Основное отличие спиральной модели от других каскадных подходов заключается в определении и оценивании рисков. На первом витке спирали определяются начальные цели, варианты и ограничения, выполня-

ется анализ и определение риска. С учетом полученных результатов выбирается тот или иной подход к разработке продукта, по завершению которой осуществляется оценка системы и корректировка требований. На каждом витке спирали принимается решение о том, продолжать или нет работу над проектом. Достоинством спиральной модели является то, что она наиболее адекватно отражает эволюционный характер процесса разработки ПО и позволяет явно учитывать риск на каждом витке развития продукта. Недостатки подхода заключаются в трудностях контроля и управления временем разработки, а также в повышении требований к заказчику.

Рис.8. Спиральная модель.

Каркасные технологические подходы

Каркасные подходы представляют собой каркас для процессов. Одним из наиболее известных подходов данной группы является *рациональный унифицированный процесс* (rational unified process, RUP). С точки зрения управления проектом RUP предлагает упорядоченный подход к тому, как должны распределяться работа и ответственность в организации или команде, занимающейся производством ПО. Данный подход может быть достаточно легко адаптирован для самых разных проектов и организаций. RUP характеризуется следующими особенностями.

- RUP является итеративным и управляемым процессом. Он предполагает постепенное проникновение в суть проблемы путем последовательных уточнений и построение все более емкого решения на протяжении нескольких циклов. Подходу присуща внутренняя гибкость, позволяющая в ходе разработки учитывать новые требования или тактические изменения в целях. RUP предоставляет возможность выявлять и устранять связанные с проектом риски на возможно более ранних этапах разработки.
- Суть работы в рамках RUP заключается в создании и сопровождении моделей, выраженных на языке UML и всесторонне представляющих разрабатываемое ПО. Применяемые модели основаны на понятиях классов и объектов и отношений между ними.

• В центре разработки в рамках RUP лежит архитектура ПО. Главное внимание уделяется раннему определению архитектуры и формулированию основных ее особенностей. Наличие «устойчивой» архитектуры программы облегчает параллельную разработку, минимизирует переделки, увеличивает вероятность того, что компоненты можно будет использовать повторно, и делает систему более удобной для последующего сопровождения. Эта архитектура служит для планирования использования и управления развитием программных компонентов.

• Разработка в рамках RUP сосредоточена на прецедентах. Прецеденты (Use Cases) описывают процесс взаимодействия программы с пользователями и другими системами, определяя её назначение и функциональные особенности. Каждый прецедент представляет характерную процедуру применения разрабатываемой системы конкретными действующими лицами, в качестве которых могут выступать не только люди, но и другие системы или устройства. Концепции прецедентов и сценариев взаимодействия с пользователями используются на всех стадиях процесса, от формулирования требований до тестирования.

• RUP поддается конфигурированию и содержит рекомендации по его выполнению для нужд конкретной организации или команды разработчиков. Хотя ни один отдельно взятый процесс не способен удовлетворить требованиям всех разработчиков, RUP может быть настроен и масштабирован для использования как в маленьких коллективах, так и в больших компаниях. Он основан на простой и ясной архитектуре, которая обеспечивает концептуальное единство для множества процессов разработки, но при этом позволяет выполнить адаптацию к различным проектам.

• RUP поощряет объективный контроль качества и управление рисками на всех стадиях реализации проекта. Контроль качества является неотъемлемой частью процесса, охватывает все виды работ и всех участников. Управление рисками также встроено в процесс таким образом, чтобы выявлять и устранять возможные препятствия на пути успешного завершения проекта на ранних этапах разработки.

• RUP поддерживается инструментальными средствами, позволяющими автоматизировать различные действия, выполняемые в ходе процесса. Они используются для создания и совершенствования промежуточных продуктов на разных этапах процесса создания ПО, например, при визуальном моделировании, программировании, тестировании и т.д.

Работы по созданию ПО в RUP разделены на четыре фазы.

• Начало (inception) — определение бизнес-целей проекта. На этой стадии определяются цели системы и устанавливаются рамки проекта. Анализ целей включает выработку критерия успешности, оценку рисков, необходимых ресурсов и составление плана, в котором отражены основные опорные точки. Нередко создается исполняемый прототип, демонст-

рирующий реалистичность концепции. В конце начальной фазы еще раз подвергается внимательному изучению весь жизненный цикл проекта и принимается решение, стоит ли начинать полномасштабную разработку.

- Исследование (elaboration) — разработка плана и архитектуры проекта. На данном этапе выполняется анализ предметной области, разрабатывается архитектурная основа, составляется план проекта и устраняются наиболее опасные риски. Архитектурные решения принимаются тогда, когда стала ясна структура системы в целом и больш*а*я часть требований уже сформулирована. Для подтверждения правильности выбора архитектуры создается система, демонстрирующая предложенные принципы в действии и реализующая некоторые наиболее важные прецеденты. В конце фазы исследования изучаются детально расписанные цели проекта, его рамки, выбор архитектуры и методы управления основными рисками, а затем принимается решение о том, надо ли приступать к построению.
- Построение (construction) — постепенное создание системы. В данной фазе итеративно разрабатывается продукт, готовый к внедрению. На этапе построения описываются оставшиеся требования и критерии, завершается разработка и тестирование ПО. В конце фазы принимается решение о готовности программ, эксплуатационных площадок и пользователей к внедрению.
- Внедрение/переход (transition) — поставка системы конечным пользователям. В фазе внедрения ПО передается пользователям. После этого часто возникают требующие дополнительной проработки вопросы по настройке системы, исправлению ранее незамеченных ошибок и окончательному оформлению ряда функций, реализация которых была отложена. Обычно эта стадия воплощения проекта начинается с выпуска бета-версии системы, которая затем замещается коммерческой версией. В конце фазы внедрения делается заключение о том, достигнуты ли цели проекта и надо ли начинать новый цикл разработки.

Фазы начала и исследования охватывают проектные стадии жизненного цикла процесса разработки. Фазы построения и внедрения относятся к производству. Внутри каждой фазы происходит несколько итераций. Итерация представляет полный цикл разработки — от выработки требований во время анализа до реализации и тестирования. Итерация — это завершенный этап, в результате которого выпускается версия продукта для внутреннего или внешнего использования, реализующая часть запланированных функций. Затем эта версии от итерации к итерации дорабатывается до получения готовой системы. Во время каждой итерации выполняется несколько рабочих процессов, хотя в разных фазах основной упор делается на разных работах. В начальной фазе главной задачей является выработка требований, в фазе исследования — анализ и проектирование, в фазе построения — реализация, а в фазе внедрения — развертывание. Все фазы и итерации подразумевают определенные действия, направленные на сни-

жение рисков. В конце каждой фазы находится четко определенная опорная точка, где оценивается степень достижения установленных целей и принимаются решения о внесении в процесс необходимых изменений.

Прохождение через четыре основные фазы называется циклом разработки. Каждый цикл завершается генерацией версии системы. Первый проход через все четыре фазы называют начальным циклом разработки. Если после этого работа над проектом не прекращается, то полученный продукт продолжает развиваться и снова минует те же фазы.

RUP включает девять рабочих процессов:

- моделирование бизнес-процессов — описывается структура и динамика организации;
- разработка требований — определяется назначение системы и в форме прецедентов документируются требования к ней;
- анализ и проектирование — разрабатываются различные виды архитектуры системы и модели для её последующей реализации;
- реализация — выполняется разработка программ, их автономное тестирование и интеграция;
- тестирование — описываются и реализуются тестовые сценарии, процедуры и метрики для измерения числа ошибок и их исправления;
- развертывание — охватывает выпуск версий ПО, конфигурирование и поставку пользователям;
- управление конфигурацией — предназначается для управления изменениями и поддержания целостности получаемых в ходе работы над проектом результатов и продуктов;
- управление проектом — определяются стратегии работы с итеративным процессом, ориентированные на эффективное распределение и решение задач, снижение рисков и преодоление возникающих трудностей;
- анализ среды — рассматриваются вопросы инфраструктуры, необходимой для разработки ПО.

Генетические технологические подходы

Синтезирующее программирование предполагает построение (синтез) программы по её спецификации. Напомним, что спецификацией называется точное, однозначное описание каких-либо аспектов программной системы, например её функций и ограничений. В отличие от готовой программы документ на языке спецификаций является лишь базисом для последующей реализации. Для её осуществления необходимо решить следующие основные задачи:

- Доопределить детали, которые нельзя выразить при помощи языка спецификаций, но необходимые для получения исполняемого кода.
- Выбрать язык и аппаратно-программную платформу для реализации.

• Зафиксировать отображение понятий языка спецификаций на язык реализации и аппаратно-программную платформу.

• Осуществить трансформацию представления из спецификации в исполняемую программу на языке реализации.

• Протестировать и отладить полученную программу.

Существует большое число разных языков спецификаций, позволяющих выполнить автоматическую генерацию программ по спецификациям, в том числе UML, ASN.1, SDL.

Сборочное (*расширяемое*) *программирование* предполагает, что программа создается на базе уже существующих компонентов. Обобщенная модель такого подхода к разработке ПО приведена на рис.9.

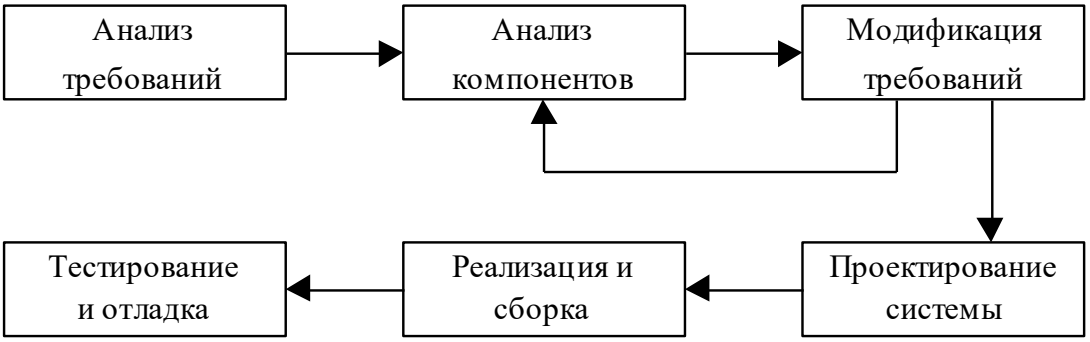

Рис. 9. Обобщенная модель сборочного программирования.

Специфическими в данном подходе являются промежуточные процессы. Анализ компонентов подразумевает поиск программных компонентов, которые могли бы удовлетворить сформулированным требованиям. Поскольку обычно невозможно точно сопоставить функции, реализуемые готовыми компонентами, и функции, определенные в полученных спецификациях требований, то выполняется процесс модификации требований. Требования изменяются таким образом, чтобы максимально использовать возможности отобранных компонентов. Если изменение требований недопустимо по каким-либо причинам, повторно осуществляется анализ компонентов для того, чтобы найти альтернативный вариант решения. При проектирование системы следует учитывать выбранные программные компоненты и строить её структуру в соответствии с их функциональными возможностями. Недоступные компоненты проектируются и разрабатываются как новые в соответствии с заданными требованиями. Сборка готовой системы может выполняться вручную или быть автоматизирована с помощью специальных инструментальных средств.

Основное достоинство сборочного программирования состоит в том, что сокращается количество непосредственно разрабатываемых компонентов и уменьшается общая стоимость и время создания системы. Однако использование данного подхода, как правило, предполагает модификацию и адаптацию исходных требований. Это может привести к тому, что закон-

ченная система не будет удовлетворять всем требованиям заказчика. Кроме того, в ходе сопровождения и модернизации ПО отсутствует возможность влиять на появление и изменение новых версий компонентов, используемых в системе, что усложняет сам процесс модернизации.

Конкретизирующее программирование предполагает, что частные, специальные программы «извлекаются» из универсальной. Наиболее известным представителем данной группы является подход с применением паттернов проектирования. *Паттерн* (*шаблон*) *проектирования* (design pattern) — абстракция, которая представляет успешные проектные решения. Паттерн описывает проблему и метод её решения таким образом, чтобы можно было использовать это решение в различных условиях и для разных частных задач. Разработчик, использующий паттерны, может конкретизировать их и применить к решению новой задачи.

Паттерн включает четыре основных элемента:

- Имя — содержательное имя, однозначно определяющее проблему проектирования.
- Задача — описание проблемной области с перечислением условий, в которых можно использовать паттерн.
- Решение — описание решения, его элементов и взаимоотношений между ними, заданное в обобщенной форме, которая должна конкретизироваться при применении. Представляет собой шаблон проектных решений, которые можно использовать в разных ситуациях и различными способами.
- Результаты — описание результатов применения паттерна и связанных с этим компромиссов. Обычно используется для того, чтобы помочь разработчикам оценить конкретную ситуацию и выбрать для неё наиболее подходящий паттерн.

Часто в описание паттерна вводятся также разделы мотивации, описывающий обоснование полезности паттерна, и применимости, представляющий ситуации, в которых его можно использовать.

Паттерны проектирования упрощают повторное использование удачных проектных и архитектурных решений. Вместе с тем паттерны могут быть довольно сложны и требовать значительных затрат для освоения. Поэтому их эффективное использование предполагает наличие достаточного опыта у разработчиков, позволяющего распознавать общие ситуации, в которых можно применить тот или иной паттерн.

Кроме паттернов в конкретизирующем программировании используются так называемые каркасы (frameworks). Это наборы взаимосвязанных программных компонентов, представляющих повторно используемую структуру и настраиваемую основу для некоторого типа программ. Каркас устанавливает определенную архитектуру для разрабатываемой системы и предлагает проектные решения, общие для предметной области. Например,

существуют каркасы для разработки компиляторов и трансляторов, создания web-приложений и приложений БД и т.д.

Подходы на основе формальных преобразований

Подходы данной группы имеют много общего с каскадными подходами, но построены на основе формальных математических преобразований системной спецификации в исполняемую программу. Обобщенная схема такого подхода приведена на рис.

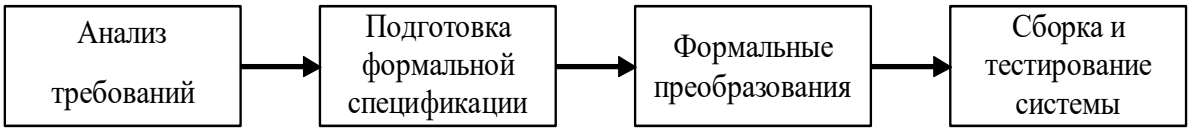

Рис.10. Обобщенная модель подхода на основе формальных преобразований

Кардинальные отличия данных подходов от каскадных подходов заключается в следующем. Системные спецификации в данном случае представляются в детализированной форме и записываются с помощью математической нотации. Процессы проектирования, реализации и тестирования программных модулей заменяются процессом, в котором формальное математическое представление системы путем последовательных формальных преобразований трансформируется в программный код, постепенно все более детализированный. Эти преобразования выполняются до тех пор, пока все позиции формальной спецификации не будут переведены в эквивалентную программу. Поскольку преобразования осуществляются математически корректно, исчезает проблема проверки соответствия спецификации и программы.

Основное преимущество подходов данной группы заключается в точном соответствии готовой программы исходной спецификации. Поэтому они обычно применяются для разработки ПО, которое должно удовлетворять строгим требованиям надежности, безотказности и безопасности. Использование подходов на основе формальных преобразований требует специальных знаний и опыта, поскольку выбор для применения соответствующих формальных преобразований обычно сложен и неочевиден. Кроме того, для большинства систем такие подходы не дают существенного выигрыша в стоимости или качестве по сравнению с другими подходами к разработке ПО. Главным образом это связано с тем, что функционирование большинства систем с трудом может быть описано с помощью формальных спецификаций.

Наиболее известным представителем подходов на основе формальных преобразований является *технология стерильного цеха* (cleanroom process model), ориентированная на предотвращение дефектов в разрабатываемом ПО, а не на их устранение. Обобщенная модель технологии приведена на рис.11.

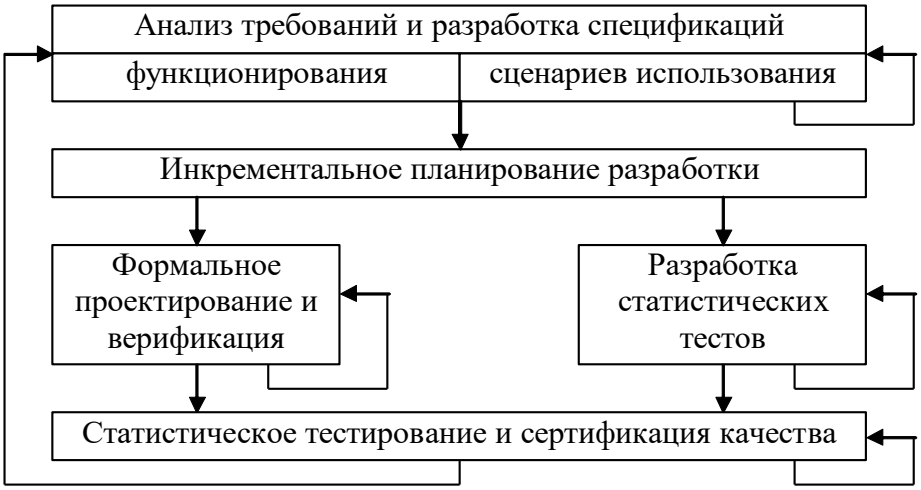

Рис. 11. Обобщенная модель технологии стерильного цеха.

Данный подход характеризуется следующими основными особенностями.

- *Формальная спецификация*. Программа рассматривается как совокупность математических функций и отношений между ними. Проектирование и реализация системы осуществляется путем поэтапной декомпозиции функциональных спецификаций.
- *Пошаговая* (*инкрементальная*) *разработка*. Разработка ПО разделяется на несколько этапов. Каждый этап представляет собой завершенный цикл процесса разработки, в результате которого получается вполне работоспособная система, но с ограниченными возможностями. Заказчик оценивает представленную версию системы и при необходимости вносит изменения. На первых этапах реализуются наиболее важные для заказчика системные функции, а остальные дорабатываются на последующих.
- *Формальная верификация правильности программы*. На каждом этапе проверяется соответствие ПО спецификациям посредством просмотра и проверки кода. Программа считается правильной, если она удовлетворяет формальным условиям корректности.
- *Статистическое тестирование и сертификация*. В конце каждого этапа разработки выполняется тестирование системы статистическими методами, позволяющими оценить её надежность и готовность к эксплуатации. Статистические тесты основываются на спецификациях сценариев использования. В результате сертификации определяются научно-обоснованные показатели качества и готовности программы, которые учитываются на следующем этапе.

Технология стерильного цеха предполагает разделение разработчиков на несколько групп:

- Группа управления (менеджер проекта) — осуществляет планирование и управление процессом разработки.

• Группа спецификации — отвечает за разработку и поддержку программных спецификаций.

• Группа разработки — занимается разработкой ПО на основе спецификаций и его проверкой.

• Группа сертификации — осуществляет разработку и реализацию статистических тестов, а также оценивает готовность ПО для использования.

Группы обычно включают от 5 до 8 человек. В маленьких проектах отдельные разработчики могут участвовать в нескольких группах. В больших проектах группы могут разделяться на подгруппы в зависимости от структуры системы и развития процесса разработки и включать сотни людей.

Технология стерильного цеха одинаково эффективно может использоваться для создания ПО разного объема и сложности. Данный подход может применяться как для разработки систем «с нуля», так и для поддержки и реинжиниринга уже существующих продуктов.

Формальные генетические подходы основываются на генетических подходах, в которых используются формальные спецификации.

Ранние подходы быстрой разработки

Подходы данной группы объединяют следующие основные черты.

• Этапы разработки технических требований, проектирования и реализации перекрываются. Детальная системная спецификация обычно отсутствует. Исходные пользовательские требования определяют только наиболее важные характеристики системы и уточняются в процессе разработки. Проектная документация, как правило, зависит от используемых для реализации ПО инструментальных средств.

• Система разрабатывается пошагово (итерационно). Для её реализации часто используются методы и средства быстрой разработки, в том числе CASE-средства, динамические языки высокого уровня, повторно используемые компоненты. Пользовательский интерфейс системы обычно создается с помощью интерактивных систем разработки.

• ПО создается в тесном взаимодействии с заказчиком, который участвует в процессе на каждой итерации и оценивает полученные результаты. На их основе заказчик может предлагать изменения и вносить новые требования, которые должны быть реализованы в следующей версии системы.

Подходы на основе (*эволюционного*) *прототипирования* или *макетирования* (evolutionary prototyping) относятся к технологиям быстрой разработки приложений (Rapid Application Development, RAD). В основе эволюционного прототипирования лежит идея разработки первоначальной версии системы, демонстрации её заказчику и последующей модификации вплоть до получения продукта, отвечающего всем требованиям (рис.12.).

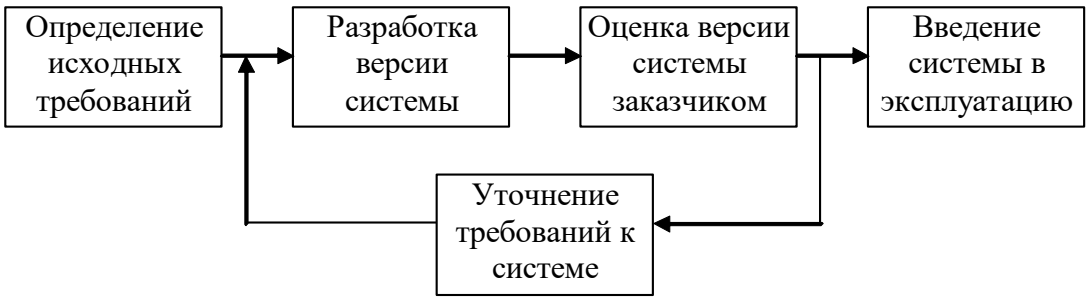

Рис.12. Модель разработки на основе эволюционного прототипирования.

Данный подход имеет ряд недостатков.

- Заказчик может принять один из промежуточных прототипов за конечный вариант системы.
- Конечный продукт может быть реализован не самыми эффективными средствами. Поскольку основная цель заключается в быстром создании действующего прототипа и предоставлении его заказчику, для реализации версии продукта могут применяться методы и средства, ориентированные на скорость разработки, а не на эффективность программ. При создании следующих прототипов разработчик может осознанно или неосознанно «забыть» о необходимости переделки данных частей с целью повышения эффективности их работы.
- Усложняется управление проектом, поскольку быстрое создание прототипов приводит к трудностям в выделении отдельных этапов процесса разработки и оценке получаемых результатов.
- Могут возникнуть проблемы с сопровождением системы. Так как непрерывные изменения в прототипах приводят к модификации структуры системы, то готовый продукт будет труден для понимания всем, кроме, возможно, первоначальных разработчиков. Помимо этого, использовавшиеся для создания ПО средства быстрой разработки могут устареть.

Постадийная разработка (staged delivery) предполагает, что процесс создания ПО разделяется на ряд стадий, после завершения каждой из которых заказчику передается работоспособная версия продукта для эксплуатации и оценки (рис.13.). Перед началом разработки определяются требования к программе, выполняется планирование работ и общее проектирование архитектуры системы. На каждой стадии осуществляется детальное проектирование, реализация и тестирование ПО. На первой стадии разрабатывается базовая версия системы, обеспечивающая основные требования. На последующих стадиях эта версия модифицируется для реализации дополнительной и доработки существующей функциональности. Получаемый в итоге продукт улучшается от стадии к стадии и учитывает изменяющиеся требования заказчика. Основное отличие данного подхода от прототипирования заключается в том, что после каждой стадии заказчику предоставляется работающее ПО. В результате система может быть введе-

на в эксплуатацию на ранних этапах разработки. Вместе с тем успешность использования постадийной разработка зависит от тщательности планирования работ и точности архитектуры системы.

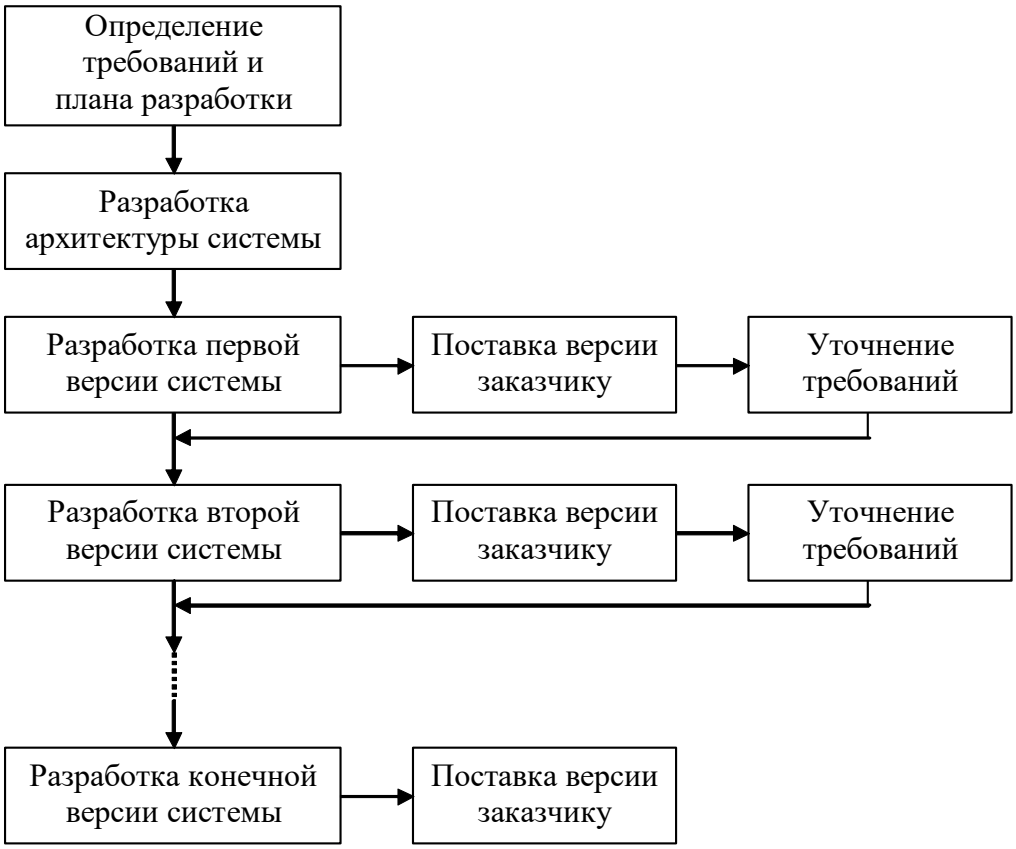

Рис. 13. Модель постадийной разработки.

Итеративная разработка (iterative delivery) имеет много общего с прототипированием и постадийной разработкой. В процессе итеративной разработки в начале создается архитектура и ядро системы, а оставшиеся компоненты программы затем дорабатываются на следующих этапах (рис.14.). Функциональные требования к создаваемой системе определяются перед началом разработки в общих чертах. При этом устанавливается, какие функции являются для заказчика наиболее важными и должны быть реализованы в первую очередь. Перед началом создания ПО осуществляется разделение процесса разработки на шаги и в соответствии с полученными приоритетами определяется, какие функции на каком этапе должны быть реализованы. На начальных итерациях создается ядро системы, а также анализируются и детализируются требования к остальным компонентам, которые должны разрабатываться на более поздних шагах. По завершении каждой итерации полученные компоненты интегрируются с уже существующей системой, в результате чего продукт приобретает все большую функциональность. Заказчик оценивает готовые подсистемы и компоненты и, при необходимости, уточняет требования к следующим

версиям разрабатываемых подсистем или к компонентам, создаваемым на последующих шагах. Изменения требований для готовых частей продукта не допускаются. На каждой итерации для разработки подсистем может использоваться наиболее пригодный подход, например каскадный, генетический или прототипирование.

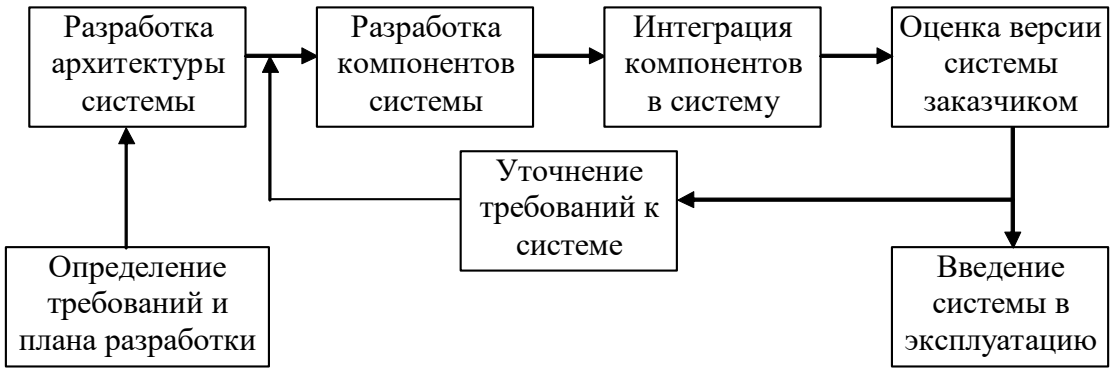

Рис.14. Модель итеративной разработки.

Основное достоинство данного подхода заключается в том, что уменьшается риск появления ошибок в особо важных частях программы. Поскольку подсистемы с высоким приоритетом разрабатываются первыми, а все последующие компоненты интегрируются с ними, то наиболее значимые части системы подвергаются более тщательному всестороннему тестированию и проверке. Проблемы, которые могут возникнуть при использовании итеративной разработки, связаны с трудностями ранжирования значимости функций и распределения их реализации по шагам и компонентам. Многие важные общесистемные функции обычно определяются взаимодействием различных подсистем и поэтому не могут быть реализованы, пока не созданы соответствующие компоненты.

Адаптивные технологические подходы

Подходы данной группы характеризуются теми же основными чертами, что и другие подходы быстрой разработки. Главное их отличие от остальных технологических подходов заключается в том, что они ориентированы на человека (учитывают человеческий фактор), а не на процесс разработки, а также на результаты работы и минимизацию самого процесса при максимальном увеличении взаимодействия между людьми, участвующими в разработке. Адаптивные подходы поддерживают изменения, касающиеся не только требований к ПО, но и непосредственно осуществляемых при его создании технологических процессов.

Экстремальное программирование (extreme programming, XP) является наиболее известным на сегодняшний день представителем адаптивных подходов. Данный подход ориентирован на группы малого и среднего размера, разрабатывающие ПО в условиях неопределенных или быстро изменяющихся требований. Основная цель XP состоит в минимизации высокой стоимости изменений, характерной для большинства бизнес-

приложений, которая достигается за счет гибкости процесса разработки. Гибкость обеспечивается четырьмя ключевыми характеристика подхода:

- непрерывное взаимодействие с заказчиком, в том числе и в пределах команды разработчиков, а также между самими разработчиками;
- простота выбираемых для реализации решений;
- обратная связь, позволяющая на основе тестирования оценить полученные результаты и внести необходимые изменения;
- смелость разработчиков при принятии решений относительно проведения профилактики и устранения возможных проблем, а также внесения изменений.

Происходящие отсюда принципы, лежащие в основе подхода, включают в себя минимальность, простоту, итерационный цикл разработки, малую длительность итераций, участие пользователей в разработке, использование стандартов кодирования и др. Данные принципы применяются и в других подходах, однако в XP они рационально объединены с целью учета динамики изменений и достигают «экстремальных» значений.

Благодаря этим особенностям изменения требований в XP возможны на всем протяжении итерационного цикла разработки. Каждая итерация включает в себя выслушивание заказчика, проектирование, кодирование и тестирование. Для реализации этих действий подход предлагает следующие так называемые практики:

- Игра планирования (planning game) — быстрое определение области действия следующей реализации путем объединения приоритетов заказчика и технических оценок. Заказчик определяет требуемую функциональность, приоритетность и сроки реализации, а разработчики оценивают и отслеживают продвижение реализации системы.
- Частая смена версий (small releases) — быстрый запуск в эксплуатацию каждой версии системы, начиная от самой простой. Новые версии реализуются в очень коротком цикле, продолжительностью от одной до трех недель. Каждая версия должна обеспечивать ограниченный, но законченный набор функций.
- Метафора (metaphor) — вся разработка проводится на основе простой, общедоступной истории о том, как работает вся система. Метафора представляет высокоуровневую архитектуру создаваемой системы и отражает происходящие в ней в процессе разработки изменения.
- Простое проектирование (simple design) — проектирование и реализация выполняется настолько просто, насколько это возможно в данный момент. Поскольку в условиях частого изменения требований программный код может быстро устаревать, то не имеет смысла реализовывать решения и возможности или применять инструменты, которые потребуют большего времени и усилий. Проектируется и реализуется только та функ-

циональность, которая относится к текущей итерации, а любые будущие потребности не учитываются.

- Тестирование (testing) — непрерывное написание тестов, которые должны выполняться безупречно. Тесты модулей создаются до начала кодирования самими программистами. Тесты для проверки полноты реализации требуемых функций разрабатываются при непосредственном участии заказчика. Роль, которая тестированию отводится в подходе, отражает фраза «тестируй, а затем кодируй». Она отражает принцип, согласно которому тестирование планируется в начале разработки, а тестовые варианты создаются параллельно анализу требований.
- Реорганизация (refactoring) — система реструктурируется, т.е. перерабатываются отдельные модули, но общее поведение программы не изменяется. Цель реорганизации заключается в устранении дублирования, улучшении взаимодействия модулей, упрощении реализации или повышении гибкости системы.
- Парное программирование (pair programming) — код каждого модуля пишется двумя программистами, работающими за одним компьютером. Один разработчик пишет код, а другой параллельно его проверяет и корректирует. Это позволяет улучшить качество исходного кода, что в дальнейшем снижает затраты сопровождения. Кроме того, эти мини-группы не должны быть фиксированными, т.е. у каждого программиста напарник периодически меняется, причем группы работают над разными частями системы. Постоянная смена области приложения повышает квалификацию разработчиков в результате обмена опытом и мотивацию к совершенствованию собственного мастерства.
- Коллективное владение кодом (collective ownership) — любой разработчик может улучшать любой программный фрагмент системы в любое время и принимать участие в обсуждении планов её развития. Непрерывная интеграция, тестирование и парное программирование позволяют защититься от возникающих при этом проблем.
- Непрерывная интеграция (continuous integration) — система интегрируется и строится несколько раз в день, по мере завершения каждой задачи. Непрерывное регрессионное тестирование, то есть повторение предыдущих тестов, гарантирует, что изменения требований не приведут к потере функциональности.
- 40-часовая неделя (40-hour week) — как правило, команда работает над проектом не более 40 часов в неделю. Следует избегать сверхурочных работ.
- Локальный заказчик (on-site customer) — в команде всегда должен находиться представитель заказчика, действительно готовый отвечать на вопросы разработчиков.

• Стандарты кодирования (coding standards) — должны выдерживаться общие для всех разработчиков правила, обеспечивающие одинаковое представление программного кода во всех частях системы.

Разработка в XP начинается с того, что выполняется анализ назначения системы и определяется первоочередная функциональность. В результате составляется список так называемых историй, определяющих варианты применения системы. История — это компактный документ, составленный пользователем и описывающий одну отдельную операцию, выполняемую при взаимодействии с программой. Каждая история должна быть ориентирована на определенные задачи бизнеса, которые можно оценить с помощью количественных показателей. Ценность истории определяется материальными и временными затратами на её разработку командой разработчиков. Реализация истории должна быть ограничена по срокам временем разработки очередной версии системы. В противном случае такая история должна быть разбита на более мелкие.

Заказчик выбирает истории для очередной итерации, основываясь на их значимости и ценности. Для первой версии системы определяется небольшое количество логически связанных наиболее важных историй. Для каждой следующей версии выбираются наиболее важные истории из числа оставшихся. В любое время к существующим могут быть добавлены новые истории, благодаря чему обеспечивается учет быстро изменяющихся требований. Цель каждой итерации заключается в том, чтобы включить в версию несколько новых историй. На собрании по планированию итерации определяется, какие именно истории будут реализованы и каким образом это будет сделано командой разработчиков.

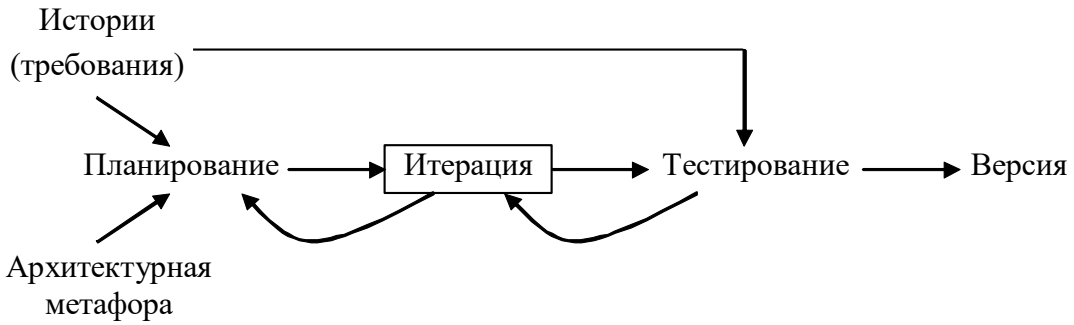

Рис. 15. Упрощенная модель подхода на основе экстремального программирования.

В данном подходе уделяется значительное внимание организации группы разработчиков и рабочего пространства. Основная идея заключается в том, что разработчики должны быть постоянно доступны для взаимодействия. Желательно располагать команду в одной большой комнате. Рекомендуется одноранговая структура группы, в которой существует только профессиональный авторитет, но не административное деление.

В основе подхода *адаптивной разработки* (adaptive software development, ASD) лежит концепции теории сложных адаптивных систем. Подход рассчитан на использование в проектах, для которых характерен быстрый темп разработок, непредсказуемость и частые изменения требований. В адаптивной разработке жизненный цикл включает три нелинейные перекрывающиеся фазы: обдумывание, взаимодействие и обучение. Обдумывание предназначено для исследования проблемы и способов её решения, а также определения и обсуждения организационных, технических, ресурсных и других аспектов проекта. При этом не исключается планирование, однако планы должны быть «гибкими» и допускать любые необходимые изменения. Взаимодействие подразумевает постоянное сотрудничество всех вовлеченных в проект сторон с целью успешной разработки системы в условиях неопределенности. Непрерывная совместная работа позволяет разработчикам эффективнее решать возникающие проблемы, повысить качество получаемых результатов и оперативнее реагировать на возникающие изменения. Обучение предполагает анализ и оценку качества проделанной работы с целью пересмотра, как разработчиками, так и заказчиками существующих обязательств, планов и решений, а также учета обнаруженных проблем и недостатков и определения того, какие изменения необходимо произвести. Таким образом, итоги каждого цикла разработки должны использоваться для подготовки следующего и служить для повышения качества работы.

Адаптивная разработка характеризуется следующими основными особенностями.

- Целенаправленность. Цель определяет границы проекта и направление его развития, а не конечные результаты. Основные положения цели задают исходные рамки для первоначального исследования, которые затем в процессе работы над проектом начинают сужаться.
- Компонентный подход. Жизненный цикл проекта строится исходя из результатов, а не задач. В качестве результатов выступают компоненты системы. Под компонентом понимается некий набор свойств программы или некоторых элементов, входящих в поставку системы, который должен быть разработан в течение итерации.
- Итеративность. Разработка ведется на основе итераций и предполагает «переделку» продукта. Компоненты программы изменяются на протяжении нескольких итераций.
- Временные рамки. Для каждой итерации определяются фиксированные сроки поставки, ограничивающие не столько время, сколько принятие решений. Существование временных рамок заставляет доводить начатое до конца и постоянно пересматривать основные показатели проекта: его границы, расписание работ и поставок очередных версий, ресурсы, дефекты и т.д.

• Учет рисков. Как и в спиральной модели, разработка ведется на основе анализа и оценки рисков.

• Допустимость изменений. Адаптивная разработка приемлет все возникающие изменения. В данном подходе отклонения от планов и исходных требований не являются ошибкой, а позволяют получать правильные и улучшать существующие решения.

Адаптивная разработка допускает возможность использования методов и практик, применяемых в других подходах. Так, например, в фазе взаимодействия для реализации компонентов могут использоваться парное программирование и совместное владение кодом, характерные для экстремального программирования.

Помимо экстремального программирования и адаптивной разработки существует несколько других адаптивных подходов, в том числе Crystal, SCRUM, Feature Driven Development.

Подходы исследовательского программирования

В основе исследовательского программирования в большей степени, чем в других подходах, лежит искусство. Исследовательское программирование характеризуется следующими особенностями:

• разработчик ясно представляет направление поиска, но не знает заранее, как далеко он может продвинуться к цели;

• нет возможности предвидеть объем ресурсов для достижения того или иного результата;

• разработка не поддается детальному планированию и ведется методом проб и ошибок;

• работа связана с конкретными исполнителями и отражают их личностные качества.

К представителям данной группы подходов относится так называемый *компьютерный дарвинизм*. Этот подход основан на принципе восходящей разработки, когда система строится вокруг ключевых компонентов и программ, которые создаются на ранних стадиях проекта, а затем постоянно модифицируются. Все более крупные блоки собираются из ранее созданных мелких блоков. Разработка в компьютерном дарвинизме построена на методе проб и ошибок, основанном на интенсивном тестировании. На любом этапе система должна работать, даже если это минимальная версия того, к чему стремятся разработчики. Естественный отбор оставляет только самые жизнеспособные решения. Подход включает три основных процесса: прототипирование, тестирование и отладку. Одна из особенностей подхода заключается в распараллеливании процессов тестирования и отладки.

Лекция № 23.

ТЕМА: Классификация языков программирования.

Основные вопросы, рассматриваемые на лекции:

1. Основные характеристики языков программирования.
2. Классические этапы разработки программного обеспечения.

Основные характеристики языков программирования. Существует большое количество различных характеристик, отражающих те или иные особенности языков программирования. К основным можно отнести следующие характеристики.

- *Уровень* — определяет сложность задач, которые могут быть запрограммированы с помощью языка, и соответствующие трудозатраты на разработку программ. Чем сложнее задачи, для решения которых может быть использован язык программирования, и чем меньше трудозатраты для написания реализующих эти задачи программ, тем более высокоуровневым считается язык.
- *Мощность* — характеризует совокупность типов задач, для решения которых могут быть разработаны программы на данном языке. Чем более разнообразно множество классов задач, тем более мощным является язык программирования.
- *Простота* — определяет, насколько сложен язык для изучения, использования и понимания. Язык будет более сложным, если он включает большое количества базовых конструкций, предлагает различные способы реализации одних и тех же действий и позволяет модифицировать способы использования конструкций языка.
- *Ортогональность* — означает, что управляющие операторы и структуры данных языка могут быть выражены с помощью относительно небольшого количества элементарных конструкций и с использованием ограниченного числа способов, причем любая возможная комбинация элементарных конструкций разрешена и имеет смысл. Отсутствие ортогональности приводит к появлению исключений из правил в языке программирования, которые обычно способствуют усложнению языка.
- *Выразительность* — определяет степень лаконичности языка для осуществления тех или иных действий. Более выразительный язык упрощает написание программ и способствует снижению их размера. С другой стороны, значительная выразительность языка в некоторых случаях может затруднить понимание программ, написанных на нем.
- *Мобильность/переносимость* — характеризует степень независимости языка от аппаратной и операционной среды, определяя возможность

переносимости создаваемых с помощью языка программ и связанные с этим трудозатраты.

• *Эффективность* — означает эффективность работы средств реализации языка (компилятора/интерпретатора) и эффективность генерируемого ими машинного кода. Более эффективными, в частности, считаются языки, позволяющие получить быстрый и компактный машинный код при относительно минимальных затратах времени.

• *Расширяемость* — определяет возможность создания и использования новых языковых конструкций или модификации существующих, с целью предоставления возможностей, изначально неподдерживаемые языком.

• *Управляющие операторы, типы и структуры данных* — характеризует наличие и степень многообразия в языке соответствующих встроенных элементов и конструкций языка. Развитые структуры управления и данных в языке в общем случае способствуют ускорению разработки программ с его помощью.

• *Поддержка абстракции* — характеризует возможность описывать и использовать сложные структуры данных или операции, игнорируя при этом второстепенные в контексте применения детали. Языки программирования могут поддерживать как абстракцию процессов, примером которой служит подпрограмма, так и абстракцию данных, примером которой является класс.

• *Проверка типов* — означает проверку совместимости типов в программе, осуществляемую в ходе её компиляции или выполнения. Проверка типов, особенно выполняемая в ходе компиляции, является одним из важных факторов обеспечения надежности разрабатываемых с помощью языка программ.

• *Обработка исключительных ситуаций* — определяет наличие в языке средств, позволяющих организовать распознавание и обработку возникающих в программе ошибок и других нестандартных ситуаций, и, тем самым, повысить надежность разрабатываемых программ.

Классификация языков программирования. Классификация языков программирования может быть выполнена в соответствии с различными критериями. Рассмотрим наиболее важные из них и приведем соответствующие критериям основные классы языков.

Классификация по поддерживаемой методологии. Каждый язык в большей или меньшей степени ориентирован на применения в рамках той или иной методологии программирования. В соответствии с существующими методологиями языки программирования делятся на следующие классы.

• *Языки структурного* (*императивного*) *программирования или процедурные языки*. К их числу относятся такие языки, как FORTRAN, C,

Pascal, BASIC, Ada. Ключевой особенностью представителей данного класса является то, что они в наибольшей степени подходят для реализации и использования на базе классической неймановской архитектуре, в соответствии с которой функционирует основная масса существующих на сегодняшний день компьютеров. Процедурные (императивные) языки программирования предоставляют средства разработки программ, манипулирующих обрабатываемыми данными в пошаговом режиме и описывающих изменения состояний вычислительной системы, согласующейся с неймановской архитектурой. Таким образом, подобные языки позволяют четко описать способ получения требуемого результата. Программы формируются из простых (атомарных) и структурных операторов. Простой оператор представляет собой единый функциональный (исполняемый) блок, который не может быть разделен на отдельно используемые части (операторы). Результатом выполнения такого оператора является соответствующее единичное изменение состояния вычислительной системы. К числу атомарных принадлежат, в частности, оператор присваивания, безусловного перехода, вызова процедуры. Структурный оператор позволяет объединить несколько операторов в новый, более крупный функциональный блок. Порядок исполнения входящих в него «дочерних» операторов зависит от соответствующего структурного оператора. Примерами структурного оператора являются составной оператор, операторы выбора, цикла. Помимо этого, структурирование программы также осуществляется с помощью подпрограмм, к числу которых относятся процедуры и функции. Подпрограмма может иметь параметры, формирующие вариации результата её работы, и определять используемые только внутри неё элементы, такие как, константы, переменные, типы и подпрограммы. Еще одной особенностью подпрограммы является то, что она может вызывать сама себя, т.е. рекурсивно. Ключевое отличие функции от процедуры состоит в том, что функция всегда возвращает некоторые значения как результат своей работы, позволяющее использовать её для вычисления выражений.

- *Языки объектно-ориентированные программирования*. Представителей данного класса можно разделить на три подгруппы. «Чистые» объектно-ориентированные языки изначально разрабатывались для использования в рамках объектно-ориентированной методологии. Такие языки, как правило, имеют небольшую языковую часть, обеспечивающую реализацию концепций объектно-ориентированного программирования, и расширяемую библиотеку классов поддержки, предоставляющую возможности создания различных типов программ. В данную подгруппу входят, в частности, Simula, Smalltalk, Self. Вторую подгруппу составляют «гибридные» языки, являющиеся развитием процедурных языков, в которые добавлена поддержка объектно-ориентированного программирования. К числу таких языков относятся Object Pascal и C++. Последнюю подгруппу образуют «адаптированные» языки, представляющие варианты развития

«гибридных» языков по пути исключения средств, не согласующихся с концепциями объектно-ориентированного программирования. Примерами таких языков являются Java и C#. Общей чертой всех объектно-ориентированных языков является то, что они позволяют представить разрабатываемую программу в форме множества объектов, являющихся абстракциями некоторых сторон действительности, в процессе взаимодействия которых осуществляется решение поставленной задачи. В этом случае состояние вычислительной системы изменяется посредством обращения к объекту, реализующему необходимые действия. Обращение представляет собой посылку специального сообщения, характеризующего одну из допустимых над объектом операций. Структура объекта состоит из совокупности полей и методов. Поля предоставляют возможность хранения данных, соответствующих конкретному объекту, в том числе ссылок на другие объекты. Методы предназначены для обработки сообщений, отправляемых объекту, и выполнения поддерживаемых им функций. Каждый объект является экземпляром некоторого класса, представляющего описание структуры объектов. Совокупность всех используемых классов определяет структуру программы.

- *Языки функционального программирования*. В данный класс входят такие представители, как LISP, Scheme, ML, Haskell, Caml. Функциональные языки обычно применяются для программирования тех задач, для которых трудно составить четкий алгоритм решения. К ним относятся, в частности, различные задачи из области разработки искусственного интеллекта. Базовым элементом программы, в этом случае, является функция, в её исходном математическом понимании. Функции можно использовать как значения переменных, включать в структуры данных, передавать в качестве аргументов в другие функции и возвращать как результат вычислений. В «чистых» функциональных языках обмен данными между функциями выполняется без использования промежуточных переменных и присваиваний. Переменные получают своё значение один раз и в дальнейшем оно не меняется. Следствием отсутствия присваивания является, в частности, то, что циклические процессы реализуются с помощью рекурсивных функций. Кроме того, исключается так называемый побочный эффект, присущий другим типам языков и связанный с тем, что результаты промежуточных вычислений размещаются в памяти, а затем извлекаются оттуда, который может приводить к получению неверных результатов вычислений, например при использовании глобальных переменных. Таким образом, в функциональных языках обращение к функции приводит лишь к вычислению её результата и не оказывает побочного влияния на внешний контекст, в котором она применяется. Поэтому при обработке выражения независимые друг от друга функции, являющиеся его частями, могут вычисляться в любом порядке или параллельно, поскольку это никак не влияет на конечный результат. Другая особенность функциональных языков

заключается в поддержке «ленивых» или отложенных вычислений, связанных с тем, что аргумент функции или часть выражения вычисляется только в случае необходимости, т.е. когда это требуется для получения значения функции или выражения. Это отличает их, в частности, от императивных языков, в которых, например, при вызове функции определяются значения всех её аргументов без учета того, понадобятся ли они для вычисления её значения. Функциональные языки позволяют определять и использовать так называемые функционалы или функции высших порядков, результат работы которых зависит от того, какие функции переданы им в качестве аргументов. Примером функционала является функция, которая вычисляет значения для множества переданных в качестве аргументов функций от другого аргумента и формирует из полученных величин результат своей работы.

- *Языки логического программирования*. Большинство из существующих языков логического программирования представляют собой одну из разновидностей языка Prolog. Область использования подобных языков во многом сходна со сферой применения функциональных языков. Программа на языке Prolog формируется из набора предложений, определяющих свойства и отношения объектов предметной области. Эти свойства и отношения называются предикатами. В общем случае предикат состоит из совокупности фактов и правил. Факты служат для описания положений, которые всегда истинны, т.е. аксиом. Правила позволяют определить логические связи между предикатами в форме «если–то», т.е. описывают способ вывода одних положений из других. Факты и правила формируются из термов, являющихся основным синтаксическим элементом программы. Терм может быть константой, переменной или выражением, составленным из других термов. Все термы рассматриваются как утверждения, истинность которых либо задана, либо требуется доказать.

- *Языки программирования в ограничениях*. К данному классу относятся, например, языки УТОПИСТ, OPS5, OPL, Prolog III. Представители этого класса во многом связаны с языками логического программирования или являются их специализированной разновидностью. Обычно они предназначены для решения определенных типов задач, формулируемых в терминах некоторых ограничений. К таким задачам относятся многие задачи исследования операций и искусственного интеллекта. Программа представляет собой описание задачи, состоящее из набора переменных, соответствующих им конечных (перечислимых) множеств допустимых значений и совокупности ограничений, заданных в форме утверждений, в которые в качестве параметров входят некоторые переменные. Ограничение может, например, иметь вид уравнения, неравенства или логического выражения. Решение задачи определяется в процессе выполнения программы и заключается в нахождении допустимых значений переменных, удовлетворяющих всем имеющимся ограничениям.

Существует также вариант классификации, в которой все языки делятся на две группы: императивные и декларативные. *Императивные языки* предназначенные для разработки программ, в которых описывается процесс получения требуемого результата, т.е. как решить задачу. К этому классу относятся процедурные и объектно-ориентированные языки. *Декларативные языки* позволяют создавать программы, представляющие собой описание задачи, т.е. что требуется найти, абстрагируясь от того, как это следует делать. Процесс решения задачи выполняется системой реализации и поддержки языка. К этой подгруппе относятся языки функционального, логического программирования и программирования в ограничениях.

Классификация по прикладной области. В соответствии с прикладными областями использования выделяют следующие основные классы языков.

- *Универсальные языки*. Позволяют создавать достаточно эффективные программы для широкого круга предметных областей. Как правило, обладают большим количеством языковых конструкций разного назначения. К числу таких языков относятся Ada, ALGOL, C++, Java, Pascal, PL/I, Python.
- *Языки для научных приложений*. Используются для программирования задач из различных научных областей, например математики, физики, химии. Обычно содержат эффективные средства для выполнения необходимых вычислений. Наиболее известным представителем этого класса является язык FORTRAN.
- *Языки для экономических приложений*. Предназначены для реализации коммерческих задач, например разработки систем бронирования и бухгалтерских программ. Отличаются развитыми средствами описания форматов обрабатываемых данных и генерации отчетов. Основной представитель данного семейства язык COBOL.
- *Языки системного программирования*. Применяются для разработки операционных систем и других программ, непосредственно взаимодействующих с аппаратным обеспечением. Включают средства низкоуровневого программирования, обеспечивающие возможности работы с аппаратурой. К числу таких языков относятся Ассемблер, C, Rebol.
- *Языки программирования и обработки баз данных*. Используются для создания программ, предназначенных для работы с различными базами данных. Включают эффективные средства манипулирования данными больших объемов и разных форматов. К этому семейству принадлежат языки Clipper, dBase, FoxPro, Progress 4GL, SQL.
- *Языки разработки интернет-приложений*. Предназначены для реализации программ, обеспечивающих функционирование различных сервисов сети Интернет, например интернет-порталов и электронных магазинов. Как правило, содержат развитые средства обработки и генерации

текстовых данных в разных форматах. Примерами подобных языков являются ASP, JSP, PHP.

- *Языки разработки скриптов*. Служат для написания относительно небольших программ, называемых сценариями или скриптами, выполняемых различными интерпретаторами, например командными оболочками операционных или других программных систем. Кроме того, с развитием сети Интернет многие скриптовые языки стали использоваться для создания интернет-приложений. Обычно содержат ограниченный набор программных конструкций и специализированы для решение узкого класса задач. К таким языкам относятся awk, JavaScript, Tcl, VBScript. Некоторые языки, изначально ориентированные на разработку скриптов, со временем превратились в универсальные языки программирования. Примером такого языка является Perl. С другой стороны, некоторые скриптовые языки были созданы на базе одного из универсальных языков, в целом сохранив их синтаксис, но исключив ненужные и добавив специфичные для области использования возможности. Например, к ним относятся языки JavaScript и VBScript.
- *Языки программирования графики*. Применяются для разработки программ, автоматизирующих различные процессы компьютерной графики или предназначенных для построения графических компьютерных интерфейсов. Ключевой особенностью является наличие развитых средств для работы с различными графическими объектами. В данную группу входят, в частности, языки AppleScript, LOGO, PostScript, Tk.
- *Языки разработки приложений искусственного интеллекта*. Предназначены для создания интеллектуальных программ различного назначения, таких как экспертные и консультирующие системы, системы анализа и распознавания данных, системы извлечения знаний. Отличаются ориентацией на выполнение символьных вычислений. К данному классу относятся такие языка, как Haskell, LISP, ML, Prolog.

Классификация по методу реализации. С точки зрения метода реализации все языки делятся на три класса.

- *Компилируемые языки*. Программы, написанные на таких языках, с помощью специальных средств транслируются в машинный код, который может непосредственно выполняться компьютером. Основным достоинством компилируемых языков является высокая скорость выполнения транслированных программ. Главные недостатки связаны с трудностями переноса на другие операционные платформы и со сложностями реализации процессов отладки. К данному классу, например, относятся языки Ada, C/C++, COBOL, Pascal.
- *Интерпретируемые языки*. В этом случае программа интерпретируется и непосредственно выполняется другой программой, называемой интерпретатором. Преимущества интерпретируемых языков заключаются в лучшей переносимости (достаточно наличия интерпретатора для нужной

платформы) и относительной простоте реализации операций отладки. Основным недостатком является значительно более медленная, чем у компилируемых языков, скорость исполнения программ. К интерпретируемым относятся многие скриптовые языки, языки разработки интернет-приложений и приложений искусственного интеллекта, в частности PHP, Prolog, Tcl, VBScript.

- *Языки со смешанной реализации*. Представляют подход, объединяющий два указанных выше метода реализации. Программа сначала транслируется в некий промежуточный код на языке, который достаточно просто интерпретируется, а затем этот код используется каждый раз для выполнения программы. Достоинствами этого метода является более быстрая, чем у интерпретируемых языков, скорость выполнения и хорошая переносимость. Данный метод реализации используется, в частности, в таких языках, как Java, Perl, Progress 4GL, Python.

Для некоторых языков система реализации включает как компилятор, так и интерпретатор. В этом случае разработка и отладка программы осуществляется с использованием интерпретатора, а после её создания выполняется компиляция в машинный код. К числу таких языков принадлежит, например, FoxPro.

Кроме того, следует отметить тенденцию к исчезновению четких границ между рассмотренными классами, обусловленную тем, что для языков со смешанным типом реализации создаются компиляторы непосредственно в машинный код, а для «чистых» интерпретируемых языков разрабатываются системы смешанной реализации.

Классификация по степени абстракции от аппаратного обеспечения. В соответствии с этим критерием языки можно разделить на три класса.

- *Языки низкого уровня*. Обычно это машинно-ориентированные языки, позволяющие писать программы, непосредственно работающие с памятью и другими аппаратными ресурсами. Одним из представителей этого класса языков является Ассемблер.
- *Языки высокого уровня*. Позволяют в значительной мере абстрагироваться от особенностей аппаратного обеспечения и предоставляют средства манипулирования сложными структурами данных, скрывая детали их реализации. Вместе с тем, во многих случаях есть возможность работы непосредственно с аппаратными ресурсами. К таким языкам относятся, в частности, языки Ada, C/C++, Java, Pascal.
- *Языки сверхвысокого уровня*. В языках данного класса полностью скрывается прямой доступ к аппаратным ресурсам и работа с ними осуществляется «прозрачно» для пользователя, без необходимости учитывать специфику реализации тех или иных операций обработки данных. Примером подобных языков служат Haskell, Prolog, Python, Ruby.

ЗАКЛЮЧЕНИЕ

Ограниченный объем издания не позволяет в рамках одной книги рассмотреть всё интересные языки программирования. К числу языков, не рассмотренных в данной публикации, но, тем не менее, заслуживающих, на наш взгляд, внимания, в частности относятся:

- C# (http://msdn.microsoft.com/en-us/vcsharp) — флагманский язык для платформы Microsoft .NET, позволяющий создавать приложения различного назначения и предоставляющий ряд возможностей (например, свойства, итераторы, лямбда-выражения, делегаты, поддержка LINQ), упрощающих и ускоряющих разработку программ.
- Erlang (http://www.erlang.org) — язык, разработанный корпорацией Ericsson и сочетающий в себе концепции функционального и параллельного программирования. Возможности параллельного программирования обеспечиваются за счет использования «легковесных» процессов, которые не имеют разделяемой памяти и взаимодействуют между собой посредством обмена асинхронными сообщениями. Программа на языке Erlang может работать в распределенной среде, т.е. процессы на одной (виртуальной) машине могут создавать процессы, выполняющиеся на другой машине и взаимодействовать с ними обычным образом. Кроме того, в случае сбоя или отказа машины процессы могут «мигрировать» для выполнения на другие машины. Еще одной особенностью языка является то, что он позволяет выполнять обновление кода программы (например, установку обновлений или исправлений для ошибок) во время её выполнения. Erlang широко используется для создания телекоммуникационных и сетевых приложений.
- Haskell (http://haskell.org/) — чистый функциональный язык, поддерживающий ленивые вычисления. Существуют как компиляторы, так и интерпретаторы языка. Характеризуется наличием большого количества модулей и библиотек. Язык используется для создания различных приложений, в том числе для символьных вычислений.
- JavaScript (https://developer.mozilla.org/en/JavaScript), ECMAScript (http://www.ecmascript.org) — язык, сочетающий в себе возможности объектно-ориентированного (на основе прототипов) и функционального программирования. Поддерживается большинством современных web-браузеров и используется, главным образом, для создания динамичных web-сайтов и интерактивных пользовательских интерфейсов, функционирующих в рамках браузера. JavaScript является одним из ключевых компонентов технологии Ajax, обеспечивающей, в том числе, возможности разработки интерактивных интерфейсов на стороне браузера. ECMAScript представляет собой стандартизированную версию языка, одним из диалектов которой является JavaScript. К диалектам ECMAScript также относится ActionScript — язык программирования для платформы Adobe Flash.
- Lua (http://www.lua.org) — встраиваемый скриптовый язык, сочетающий возможности императивного, функционального и рефлективного программирования. Lua поддерживает средства (мета-механизмы), позволяющие модифицировать и расширять его семантику и реализовывать возможности, исходно отсутствующие в языке (например, таким образом в Lua реализованы классы и наследование). Язык достаточно просто встраивается в сторонние приложения и поддерживает интерфейс для взаимодействия с другими языками. К основным областям использования Lua относятся встраиваемые системы и игры, где он применяется в качестве скриптового языка.
- PHP (http://www.php.net) — относительно простой скриптовый язык, широко используемый для разработки web-приложений. PHP изначально предназначался для динамической генерации web-страниц, поэтому содержит ряд средств, позволяющих ус-

корить и упростить написание web-приложений, в том числе возможность включения кода в HTML, встроенная поддержка большого количества СУБД, развитые средства текстовой обработки, в частности XML-документов, а также функции для взаимодействия с другими сетевыми приложениями по различным протоколам. Язык включает большое количество встроенных функций разного назначения. Кроме того, для PHP существует множество сторонних библиотек и компонентов. Начиная с пятой версии, язык полноценно поддерживает объектно-ориентированное программирование.

- Python (http://www.python.org/) — объектно-ориентированный интерпретируемый язык, ориентированный на быструю разработку приложений и интеграцию различных программ в единую систему. В качестве характерных отличий выделим простой, легкочитаемый синтаксис, высокоуровневые структуры данных, развитые возможности интроспекции, а также большой набор стандартных библиотек и сторонних модулей. Python широко используется, в частности, при разработке web-приложений, а также научных и вычислительных приложений (например, в биоинформатике). Кроме того, Python применяется в качестве встроенного скриптового языка в различных программах, в том числе для работы с трехмерной и двухмерной графикой.
- Ruby (http://www.ruby-lang.org) — язык, сочетающий в себе концепции объектно-ориентированного, императивного, функционального и рефлективного программирования. Среди отличительных особенностей Ruby можно выделить возможность переопределять и дополнять встроенные конструкции языка, интерпретацию данных любых типов как объектов, поддержку «примесей» (mixins) на основе модулей в качестве альтернативы множественному наследованию, возможность изменять и дополнять методы классов и объектов во время выполнения программы, интерпретацию блоков кода как выражений, которые можно присоединять к любым методам. Язык применяется в различных областях, однако одной из основных является разработка web-приложений, для чего существует, в частности, очень популярный фреймворк Ruby on Rails.
- Smalltalk (http://www.smalltalk.org) — один из первых объектно-ориентированных языков программирования, получивший широкое распространение и оказавший большое влияние на многие последующие языки, на современные графические пользовательские интерфейсы (GUI), а также на используемые в настоящее время интегрированные среды разработки (IDE). Все элементы программы в Smalltalk, в том числе значения базовых типов (числа, строки и т.д.), классы и блоки кода, представляют собой объекты, а выполнение программы осуществляется за счет обмена сообщениями между ними (вызова соответствующих методов). Управляющие структуры также реализованы за счет отправки сообщений объектам. Еще одна особенность языка состоит в том, что он допускает модификацию (структуры) существующих объектов во время выполнения программы и эти изменения сразу же вступают в силу. Существует несколько реализаций языка, например Dolphin Smalltalk, Squeak, VisualWorks.

Этот краткий перечень далеко не исчерпывает обширное множество используемых для разработки программ языков. Каждый год появляются всё новые языки, представляющие новые концепции и/или предназначенные для эффективного решения задач из определенных прикладных областей. В качестве примера можно привести выпущенный в свет в ноябре 2009 года корпорацией Google язык Go (http://golang.org), предназначенный для системного программирования и реализующий параллельное программирование на основе взаимодействующих процессов.

Список литературы

1. Артемов И. Л. Fortran. Основы программирования. - М.: Диалог-МИФИ, 2007. - 304 с.
2. Горелик А. М. Программирование на современном Фортране. - М.: Финансы и статистика, 2006. - 352 с.
3. Дейтел Х. М., Дейтел П. Дж. Как программировать на С. - М.: Бином-Пресс, 2009. - 910 с.
4. Жоголев Е.А. Технология программирования. — М.: Научный мир, 2004. — 216 с.
5. Иванова Г.С. Технология программирования: Учебник для вузов. — М.: Изд-во МГТУ им. Н.Э. Баумана, 2003. — 320 с.
6. Коннолли Т., Бегг К. Базы данных. Проектирование, реализация и сопровождение. Теория и практика: Пер. с англ. — М.: Издательский дом “Вильямс”, 2003. — 1440 с.
7. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ / Пер. с англ. — М.: МЦНМО, 2002. — 960 с.
8. Макконнелл С. Совершенный код. Мастер-класс / Пер. с англ. — М.: Издательско-торговый дом «Русская редакция»; СПб.: Питер, 2005. — 896 с.
9. Методы программирования. Структурное программирование на алгоритмическом языке СИ: методические указания по выполнению практических работ / В.В. Фомин, Д.В. Сикулер, И.К. Фомина. — СПб.: СПГУВК, 2005. — 110 с.
10. Монахов В. Язык программирования Java и среда NetBeans. - СПб.: БХВ-Петербург, 2008. - 640 с.
11. Непейвода Н.Н. Стили и методы программирования. — М.: Интернет-университет информационных технологий, 2005. — 320 с.
12. Ноутон П., Шилдт Г. Java 2. - СПб.: БХВ-Петербург, 2008. - 1072 с.
13. Одинцов И.О. Профессиональное программирование. Системный подход. — СПб.: БХВ-Петербург, 2002. — 512 с.
14. Орлов С.А. Технологии разработки программного обеспечения. Учебное пособие. — СПб.: Питер, 2003. — 480 с.
15. Себеста Р.У. Основные концепции языков программирования: Пер. с англ. — М.: Издательский дом “Вильямс”, 2001. — 672 с.
16. Соммервилл И. Инженерия программного обеспечения: Пер. с англ. — М.: Издательский дом “Вильямс”, 2002. — 624 с.
17. Фомин В.В. Методы проектирования программных систем. — СПб.: СПГУВК, 1996. — 70 с.