ОСНОВНЫЕ ПОНЯТИЯ

1.1. КЛАССИФИКАЦИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Программирование - в широком смысле представляет собой все технические операции, необходимые для создания программы, включая анализ требований и все стадии разработки и реализации. В узком смысле - это кодирование и тестирование программы в рамках некоторого конкретного проекта.

Программное обеспечение (ПО) *(software)* - общий термин для обозначения «неосязаемых» (в отличие от физических) составных частей вычислительной системы. В большинстве случаев он относится к программам, выполняемым вычислительной системой, чтобы подчеркнуть их отличие от аппаратных средств той же системы. Этот термин охватывает как программы в символической записи, так и исполняемые формы этих программ. Все существующее ПО можно разделить на следующие классы:

системное: операционные системы; драйверы устройств; различные утилиты;

для разработчиков: среды программирования; трансляторы и интерпретаторы; CASE-средства; библиотеки программ;

для конечных пользователей: текстовые процессоры; электронные таблицы; графические редакторы; решатели математических задач; обучающие и контролирующие системы; компьютерные игры; прикладные программы.

Прикладная программа (application prograт) - любая программа, способствующая выполнению задачи, возложенной на ЭВМ в пределах данной организации, и вносящая прямой вклад в реализацию этой задачи. Например, там, где на ЭВМ возложена задача контроля финансовой деятельности какой-либо фирмы, прикладной программой будет программа подготовки платежных ведомостей. В противоположность этому операционная система не является прикладной программой, так как не вносит прямого вклада в удовлетворение конечных потребностей пользователя.

Программная система представляет собой набор решений множества различных, но связанных между собой задач (ОС, СУБД). Более узкоспециализированные программы не называют системами (редактор текстов, компилятор и т. п.)

1.2. ЦИКЛ ЖИЗНИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Жизненный цикл ПО (software life-cycle) - весь период времени существования системы программного обеспечения, начиная от выработки первоначальной концепции этой системы и кончая ее моральным устареванием.

Жизненный цикл (рис. 1) представляется в виде некоторого числа последовательных фаз, в определенных местах охватываемых обратными связями, когда может возникнуть необходимость повторения какого-либо одного или даже всех этапов разработки системы. Такая модель обеспечивает отражение итеративности* процессов существования ПО.

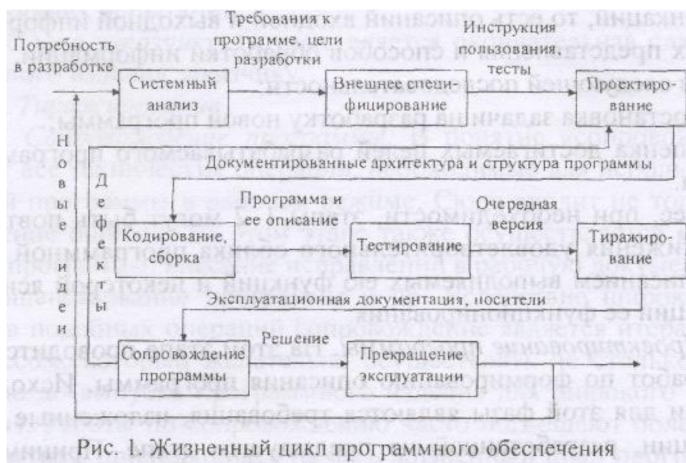

Рис. 1. Жизненный цикл программного обеспечения

Итерация - повторение численного или нечисленного процесса, когда результаты одного или нескольких шагов являются входной информацией для следующего начального шага этого цикла. Как правило, такая циклическая процедура заканчивается при достижении заданных результатов, или после того, как результаты перестают меняться.

Термин «жизненный цикл ПО» используется в том случае, когда предполагается, что программы будут иметь достаточно большой срок действия, в отличие от экспериментального программирования, при котором программы обычно прогоняются несколько раз, а затем аннулируются.

1.3. ЭТАПЫ СОЗДАНИЯ ПРОГРАММ

1. *Системный анализ.* в рамках этого этапа осуществляется анализ требований, предъявляемых к программной системе. Он проводится на основе первичного исследования всех потоков информации при традиционном проведении работ и осуществляется в следующей последовательности:

а) уточнение видов и последовательности всех работ;

б) определение целей, которые должны быть достигнуты разрабатываемой программой;

в) выявление аналогов, обеспечивающих достижение подобных целей, их достоинств и недостатков.

2. *Внешнее специфицирование.* Состоит в определении внешних спецификаций, то есть описаний входной и выходной информации, форм их представления и способов обработки информации. Реализуется в следующей последовательности:

а) постановка задачи на разработку новой программы;

б) оценка достигаемых целей разрабатываемого программного изделия.

Далее, при необходимости, этапы 1-2 могут быть повторены до достижения удовлетворительного облика программной системы с описанием выполняемых ею функций и некоторой ясностью реализации ее функционирования.

3. *Проектирование программы.* На этом этапе проводится комплекс работ по формированию описания программы. Исходными данными для этой фазы являются требования, изложенные в спецификации, разработанной на предыдущем этапе. Принимаются решения, касающиеся способов удовлетворения требований спецификации. Эта фаза разработки программы делится на два этапа:

а) архитектурное проектирование. Представляет собой разработку описания программы в самом общем виде. Это описание содержит сведения о возможных вариантах структурного построения программного изделия (либо в виде нескольких программ, либо в виде нескольких частей одной программы), а также об основных алгоритмах, и структурах данных. Результатом этой работы

являются окончательный вариант архитектуры программной системы, требования к структуре отдельных программных компонент и организации файлов для межпрограммного обмена данными;

б) рабочее проектирование. На этом этапе архитектурное описание программы детализируется до такого уровня, который делает возможными работы по ее реализации (кодированию и сборке). Для этого осуществляется составление и проверка спецификаций модулей, составление описаний логики модулей, составление окончательного плана реализации программы.

4. *Кодирование и тестирование.* Эти виды деятельности осуществляются для отдельных модулей и совокупности готовых модулей до получения готовой программы.

5. *Комплексное тестирование.*

6. *Разработка эксплуатационной документации.*

7. *Приемо-сдаточные и другие виды испытаний.*

8. *Корректировка программ.* Проводится по результатам предшествующих испытаний.

8. *Сдача заказчику.* Осуществляется окончательная сдача программного изделия заказчику.

10. *Тиражирование.*

11. *Сопровождение программы.* В понятие «сопровождение» входят все технические операции, необходимые для использования данной программы в рабочем режиме. Сюда входит не только исправление ошибок. На этом этапе также осуществляется модификация программы, внесение исправлений в рабочую документацию, усовершенствование программы и др. Вследствие широких масштабов подобных операций сопровождение является итеративным процессом, который желательно осуществлять не столько после, сколько до выпуска программного изделия для широкого использования. Работы по сопровождению часто поглощают более половины затрат, приходящихся на весь жизненный цикл программной системы в тоимостном выражении.

Современные технологии проектирования программного обеспечения направлены на частичную автоматизацию описанных выше этапов и на совмещение их во времени с целью сокращения сроков выполнения проектов.

1.4. ДОКУМЕНТИРОВАНИЕ ПРОГРАММ

Каждая стадия проектирования завершается составлением соответствующих документов, .поэтому важным элементом проектирования программных приложений является оформление программной документации. Исключение может составлять разработка бесхитростных программ с коротким жизненным циклом и небольшой трудоемкостью.

Программная спецификация (prograт specification) - точное описание того результата, которого нужно достичь с помощью программы. Это описание должно точно устанавливать, что должна делать программа, не указывая, как она должна это делать.

Для программ, заканчивающих свою работу каким-то результатом, обычно составляются спецификации ввода-вывода, в которых описывают желаемое отображение множества входных величин во множество выходных величин.

Для циклических программ (в которых нельзя указать точку завершения), разрабатывают спецификации, где основное внимание сосредоточивается на отдельных функциях, реализуемых программой в ходе циклических операций.

Существует большое число различных систем обозначений, используемых в программных спецификациях — от естественного языка с использованием математических уравнений и таблиц до формализованных описаний, основанных на исчислении предикатов *(Предикат* - функция, определяемая па некоторой предметной области переменных и принимающая значения в области истинностных значений.) первого порядка.

Разработку программных систем начинают с составления первичных спецификаций. В ходе выполнения проекта первичные спецификации последовательно претерпевают изменения до программных документов стадий и вплоть до документации, которая необходима для эксплуатации и сопровождения программы. Первичные спецификации обычно составляют в терминах решаемой задачи, а не программы. Первичная спецификация описывает:

объекты, участвующие в задаче (что делает программа и что делает человек, работающий с этой программой);

процессы и действия (проектные процедуры и действия человека, алгоритмы решения задачи в машине, порядок обработки информации, размер оперативной памяти, требуемый для работы программы);

входные и выходные данные, а также их организацию (например, сценарий диалога с экранными формами, организация файлов с указанием длин полей записей и предельного количества информации в файлах);

инструкции пользования будущей программой.

Различают внешнюю программную документацию, которая согласуется с заказчиком, и промежуточную внутреннюю документацию проекта. При составлении программной документации сначала разрабатываются внешние спецификации, а затем — внутренние.

Внешние спецификации включают спецификации входных и выходных данных, их организацию, реакции на исключительные ситуации, определение, что делает человек (по каким алгоритмам он работает и откуда берет информацию), а что машина. То есть все, что бы увидел пользователь, когда бы он получил готовую программу. Внешние спецификации зависят сильно от жизненного цикла программы.

Еще до разработки структуры и реализации программы к тестированию внешних спецификаций следует привлекать потенциальных пользователей. Пользователю можно показывать макеты экранов в порядке выполнения программы, а пользователь может готовить данные для тестирования всех функций программы и сможет апробировать методику работы с программой.

Внутренние спецификации включают описание внутренних данных программы (переменных, особенно структурированных) и описания алгоритмов всей программы и ее частей. Внутренние спецификации даются в единстве с описанием архитектуры программного комплекса и внутренней структурой построения отдельных программных компонент.

1.5. ОБЩЕСИСТЕМНЫЕ ПРИНЦИПЫ СОЗДАНИЯ ПРОГРАММ

При создании и развитии ПО рекомендуется применять следующие общесистемные принципы:

принцип включения, который предусматривает, что требования к созданию, функционированию и развитию ПО определяются со стороны более сложной, включающей его в себя системы;

принцип системного единства, который состоит в том, что на всех стадиях создания, функционирования и развития ПО его целостность будет обеспечиваться связями между подсистемами, а также функционированием подсистемы управления;

принцип развития, который предусматривает в ПО возможность его наращивания и совершенствования компонентов и связей между ними;

принцип комплексности, который заключается в том, что ПО обеспечивает связанность обработки информации, как отдельных элементов, так и для всего объема данных в целом на всех стадиях обработки;

принцип информационного единства, то есть во всех подсистемах, средствах обеспечения и компонентах ПО используются единые термины, символы, условные обозначения и способы представления;

принцип совместимости состоит в том, что язык, символы, коды и средства программного обеспечения согласованы, обеспечивают совместное функционирование всех подсистем и сохраняют открытой структуру системы в

целом;

принцип инвариантности определяет инвариантность подсистем и компонентов ПО к обрабатываемой информации, то есть являются универсальными или типовыми.

1.6. ТЕХНОЛОГИИ И ПАРАДИГМЫ ПРОГРАММИРОВАНИЯ

Технологии программирования - это апробированные стратегии создания программ, которые излагаются в виде методик с информационными фондами, описаниями проектных процедур и проектных операций. Существуют технология структурного программирования, технология проектирования программ с рациональной структурой данных, технология объектно-ориентированного программирования, технология визуального программирования.

В каждой из этих технологий используется одна или несколько парадигм программирования (концепций, систем взглядов). Последние представляют собой разные подходы к написанию программ. Для каждой из них необходимы: свой тип мышления, особая школа обучения, приемы и способы программирования, определяемые используемым языком.

Существуют четыре основные парадигмы, которые описывают большинство сегодняшних методов программирования: императивная, аппликативная, основанная на системе правил и объектно-ориентированная.

Императивная парадигма. Эта модель вытекает из особенностей аппаратной части стандартного компьютера, выполняющей инструкции (команды) последовательно. Основным видом абстракции, используемым в данной парадигме, являются алгоритмы. На основе ее разработано множество операторно-ориентиро-ванных языков программирования. Программа на таком языке состоит из последовательности операторов, выполнение каждого из которых влечет за собой изменение значения в одной или нескольких ячейках памяти. В целом синтаксис такого языка имеет вид:

```
Оператор 1:
Оператор 2:
...
```

Обычно при первом знакомстве с концепциями программирования люди сталкиваются именно с этой моделью, и многие широко распространенные языки поддерживают именно ее (например, С, С++, *FORTRAN, ALGOL, PL/I, Pascal, Ada, Smalltalk* и *COBOL).*

Аппликативная парадигма. Представляет собой другой взгляд на вычисления, производимые с помощью языка программирования. В основу этой парадигмы положено рассмотрение функции, которую выполняет программа. Здесь не рассматривается последовательность состояний, через которые должна пройти вычислительная машина. Вопрос ставится по-другому: какую функцию необходимо применить к начальному состоянию машины (путем выбора начального набора переменных и комбинирования их определенным образом), чтобы получить желаемый результат?

Языки, в которых акцентирован именно этот взгляд на вычисления, называются аппликативными, или функциональными. Синтаксис такого языка, как правило, выглядит следующим образом:

Функция_n (... функция_2 (функция_1 (данные))...)

Такую модель поддерживают такие языки как *ML* и *LISP.*

Парадигма, основанная на системе правил. Языки, основанные на этой парадигме, осуществляют проверку наличия необходимого разрешающего условия и *в* случае его обнаружения выполняют соответствующее действие. Наиболее известным языком, основанным на системе правил, является *Prolog.* Он называется также языком логического программирования.

Выполнение программы на подобном языке похоже на выполнение программы, написанной на императивном языке. Однако операторы выполняются не в той

последовательности, в которой они определены в программе. Порядок выполнения определяют разрешающие условия. Синтаксис таких языков выглядит следующим образом:

разрешающее условие_1 —> действие_1 разрешающее условие_2 —> действие__2

разрешающее условие_n —> действие _n

Иногда правила записываются в виде «действие if разрешающее условие», когда выполняемое действие записывается слева.

Объектно-ориентированная парадигма. В этой модели строятся сложные объекты данных. Для операций над ними определяется некоторый ограниченный набор методов. Создаваемые объекты могут наследовать свойства более простых объектов. Благодаря такой возможности объектно-ориентированные программы имеют высокую эффективность, свойственную программам, написанным на императивных языках. Возможность разработки различных классов, которые используют ограниченный набор объектов данных, обусловливает гибкость и надежность, которые свойственны аппликативному языку.

Существуют и другие парадигмы, что объясняется необходимостью решать разные задачи. Нельзя считать какую-либо парадигму наилучшей во всех областях практического применения. Например, для вычислительных задач более пригодна императивная парадигма, а для проектирования интеллектуальных систем применяется парадигма, ориентированная на правила. В последние годы особую популярность приобрела парадигма визуального программирования.

1.7. ТРАНСЛЯЦИЯ И ИНТЕРПРЕТАЦИЯ ПРОГРАММ

Компьютер представляет собой интегрированный набор алгоритмов и структур данных, способный хранить и выполнять программы. Компьютер может быть создан как реальное физическое устройство, которое называют просто компьютером. Но он также может быть построен с помощью программ, выполняемых на каком-то другом компьютере, тогда он называется *программно-моделируемым* или *виртуальным* компьютером, состоящим частично из аппаратуры и частично из программного обеспечения.

Подавляющее большинство программ пишется на языках программирования высокого уровня, существенно отличающихся от машинного языка реального компьютера. Теоретически можно создать аппаратный компьютер, использующий некий язык высокого уровня в качестве машинного, но это было бы очень сложно и дорого. К тому же он был бы крайне негибким, поскольку такой компьютер сложно (хотя и можно) использовать совместно с другими языками высокого уровня. Более практичное решение - аппаратно реализовать язык очень низкого уровня, что обеспечит выполнение наиболее распространенных элементарных операций и снабдить компьютер дополнительным программным обеспечением для взаимодействия с программами, написанными на языках высокого уровня.

Поэтому возникает вопрос: как организовать выполнение таких программ на конкретном компьютере? Для этой задачи существует два основных решения.

Трансляция (компиляция). Это метод перевода программ, написанных на языках высокого уровня, в эквивалентные программы на машинном языке используемого компьютера. После этого интерпретатор, встроенный в аппаратную часть микропроцессора, непосредственно выполняет оттранслированную в машинный код программу. Преимущество этого метода - очень быстрое выполнение программы после завершения процесса трансляции.

Транслятор - это языковой процессор, который воспринимает программы на некотором исходном языке в качестве входных данных, а на выходе выдает эквивалентные по своей функциональности программы, но уже на другом, так называемом объектном языке (который также может быть произвольного уровня).

Ассемблер - это транслятор, у которого исходным языком является

символическое представление машинного кода (ассемблер), а объектным языком является некая разновидность машинного языка какого-либо реального компьютера.

Компилятор — транслятор, для которого исходным является язык высокого уровня, а его объектный язык близок к машинному языку реального компьютера. Это либо язык ассемблера, либо какой-нибудь вариант машинного языка. Например, программы на языке С компилируются, как правило, в программы на языке ассемблера, которые затем транслируются ассемблером в машинный язык.

Компоновщик (редактор связей) - это транслятор, у которого исходный язык состоит из программ на машинном языке в перемещаемой форме и таблиц данных, указывающих те точки, в которых перемещаемый код должен быть модифицирован, чтобы стать выполняемым. Объектный язык состоит из готовых к выполнению машинных команд. Задачей компоновщика является создание единой выполняемой программы, в которой используются согласованные адреса, как показано в таблице.

Подпрограмма	Адреса	Адреса в выполняемой после компиляции программе
P	0-999	0-999
Q	0-1999	1000-2999
Библиотека	0-4999	3000-7999

Препроцессор (макропроцессор) - это транслятор, исходный язык которого является расширенной формой какого-либо языка высокого уровня (например, *Java* или C++), а объектный язык — стандартной версией этого языка. Объектная программа, созданная препроцессором, готова к трансляции и выполнению обычными процессорами исходного стандартного языка.

Главным недостатком трансляции является потеря информации о программе. Если в программе есть ошибка, то часто трудно определить, какой из операторов программы на исходном языке выполнялся и какие объекты данных использовались в нем. Кроме того, поскольку оператор на языке высокого уровня содержит гораздо больше информации, чем команда машинного языка, то исполняемая форма программы занимает в памяти гораздо больше места.

Интерпретация (программная имитация). Это метод, когда при помощи программы (интерпретатора), выполняемой на аппаратном компьютере, создается виртуальный компьютер с машинным языком высокого уровня. Интерпретатор декодирует и выполняет каждый оператор программы на языке высокого уровня в соответствующей последовательности и производит вывод результирующих данных, определяемый этой программой.

Достоинство этого подхода - легкость реализации многих операций отладки на уровне исходной программы, поскольку все сообщения об ошибках, возникающих в процессе выполнения, могут ссылаться на исходные модули программы.

Однако за это приходится расплачиваться необходимостью многократно декодировать один и тот же оператор, если он, например, встречается в цикле или часто вызываемой подпрограмме, что существенно снижает скорость выполнения интерпретируемых программ (в 10... 100 раз).

Смешанные системы реализации. Чаще всего для реализации языка высокого уровня на компьютере используется комбинированный подход. Сначала программа транслируется из своей исходной формы в форму, более удобную для выполнения. Обычно это делается путем создания нескольких независимых частей программы, называемых модулями. На этапе загрузки эти независимые части объединяются с набором программ поддержки выполнения, реализующих программно-моделируемые (интерпретируемые) операции. Это приводит к созданию выполняемой формы программы, операторы которой декодируются и выполняются посредством их интерпретации.

1.8. СРЕДЫ И РЕАЛИЗАЦИИ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

Среда программирования - это совокупность инструментов, используемых при разработке программного обеспечения. Этот набор обычно состоит из файловой системы, текстового редактора, редактора связей и компилятора. Дополнительно он может включать большое количество инструментальных комплексов с единообразным интерфейсом пользователя.

Старейшей средой программирования считается *UNIX* - машинно-независимая операционная система с разделением времени. Она предоставляет многочисленные мощные инструментальные средства для производства ПО и эксплуатации разнообразных языков. Работа с этой средой осуществляется с помощью графического интерфейса, устанавливаемого поверх нее. Во многих случаях этим интерфейсом является *Common Desktop Environment (CDE).*

Последнюю стадию развития сред разработки ПО представляют *Microsoft Visual C++, Visual BASIC, Delphi* и *Java Development Kit,* которые предлагают легкий способ создания графических интерфейсов для программ пользователя.

Ключевой вопрос реализации языка программирования заключается в том, какое представление имеет программа во время ее выполнения на реальном компьютере, является ли этот язык машинным языком данного компьютера или нет? В зависимости от ответа на этот вопрос языки (вернее, их реализации) делятся на *компилируемые* и *интерпретируемые.*

Компилируемые языки. Компилируемыми принято считать такие языки как *C, C++, FORTRAN, Pascal* и *Ada.* Это означает, что программы, написанные на этих языках, транслируются в машинный код данного компьютера перед началом выполнения. Программная интерпретация при этом ограничивается только интерпретацией набора программ поддержки выполнения, которые моделируют элементарные операции исходного языка, не имеющие близкого аналога в машинном языке.

Транслятор компилируемого языка является большой и сложной программой, и при трансляции основное значение имеет создание максимально эффективных (с точки зрения их выполнения) исполняемых программ.

Интерпретируемые языки. Реализуются с использованием программного интерпретатора. К таковым относятся языки *LISP, ML, Perl, Postscript, Prolog* и *Smalltalk.* При такой реализации транслятор выдает не машинный код используемого компьютера, а некую промежуточную форму программы. Эта форма легче для выполнения, чем исходная программа, но все же она отличается от машинного кода.

Использование необходимого для этого программного интерпретатора приводит к относительно медленному выполнению программы. Основная сложность здесь реализуется в программном обеспечении процесса интерпретации, поэтому трансляторы интерпретируемых языков обычно представляют собой довольно простые программы.

Развитие Всемирной паутины *WWW* и появление языка *Java* внесли изменения в описанную схему. Язык *Java* похож скорее на *Pascal* и C++, чем на *LISP,* но в большинстве случаев реализуется как интерпретируемый язык. Компилятор *Java* вырабатывает промежуточный набор байт-кодов для виртуальной машины *Java.* Передача байт-кодов на локальный компьютер (даже если он медленнее, чем web-сервер) выгоднее в отношении временных затрат, чем передача результатов выполнения программы на web-сервере. Однако web-сервер не *в* состоянии предугадать машинную архитектуру хост-компьютера. Поэтому браузер создает виртуальную машину Java, которая и выполняет стандартный набор байт-кодов Java.

СТРУКТУРЫ УПРАВЛЕНИЯ И ПОДПРОГРАММЫ

2.1. ТЕОРИЯ ПЕРВИЧНЫХ ПРОГРАММ

Теория первичных программ была предложена Маддуксом в качестве обобщения методологии структурного программирования для определения однозначной иерархической декомпозиции блок-схем. В этой теории предполагается, что графы программ могут содержать три класса узлов (рис. 2):

функциональные узлы — представляют вычисления, производимые программой, и изображаются прямоугольниками с одной входящей в этот узел дугой и одной выходящей. Функциональные узлы представляют операторы присваивания, выполнение которых вызывает изменение состояния виртуальной машины;

узлы принятия решения - изображаются в виде ромбов с одной входящей дутой и двумя выходящими (истина и ложь). Эти узлы представляют предикаты, и управление из узла принятия решения передается дальше либо по ветви истина, либо по ветви ложь;

узел соединения — представляется в виде точки, в которой сходятся две дуги графа, чтобы сформировать одну выходную дугу.

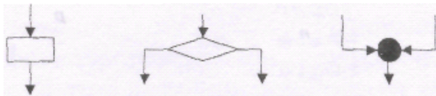

Рис. 2. Три класса узлов

Любая блок-схема состоит только из этих трех компонентов.

Правильная программа — блок-схема, являющаяся некоторой формальной моделью структуры управления, которая имеет: одну входящую дугу; одну выходящую дугу; путь от входящей дуги к любому узлу и из любого узла - к выходящей дуге.

Первичная программа является правильной программой, которую нельзя разделить на более мелкие правильные программы. Исключением из этого правила является последовательность функциональных узлов, которая считается одной первичной программой.

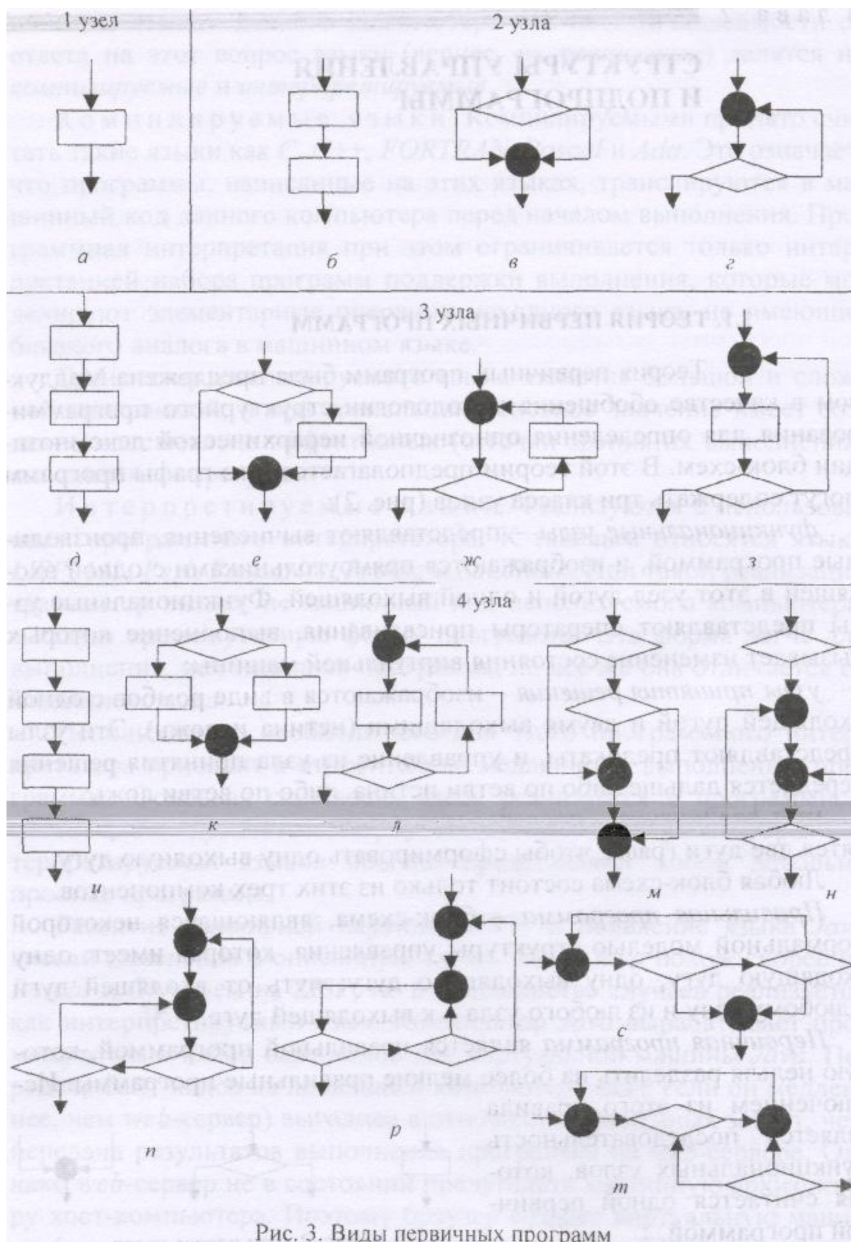

Рис. 3. Виды первичных программ

На рис. 3 изображены все первичные программы, которые включают в себя не более четырех узлов. Первичные программы *а, б, д, и* представляют собой последовательности функциональных узлов. Первичная программа *е* — конструкция *if-then, .ж — do-while, з - repeat-until, к - if-then-else, л - do-while-do.*

Первичные программы *в, г, м-т* состоят только из узлов принятия решения и соединения. В них нет функциональных узлов, поэтому они не изменяют пространство состояний виртуальной машины. Ни один из этих вариантов первичных программ не представляет эффективной структуры управления в программе.

Многие из описанных выше наборов управляющих структур были включены в существующие языки программирования. Эти структуры представляют собой первичные программы с небольшим количеством узлов и просты для понимания.

Перечислив все первичные программы, можно сделать очевидный вывод, что конструкция *do-while-do* является естественной структурой управления, хотя она была проигнорирована разработчиками языков.

2.2. АЛЬТЕРНАТИВЫ

Операторы выбора используются для выбора одного из нескольких возможных путей, по которому должно выполняться вычисление. Обобщенный оператор выбора называется *case*-оператором *(switch-оператор* в языке *С)*.

Условный оператор является частным случаем *case-* или *switch-оператора,* в котором выражение имеет булев тип. Так как булевы типы имеют только два допустимых значения, условный оператор делает выбор между двумя возможными путями. Конструкция для двух альтернатив на Паскале имеет следующий вид:

```
if  L
then begin
  {Действие при L-True} end; else begin
 {Действие при L=False}  end;
```
здесь L-логическое выражение.

Вариант конструкции для нескольких альтернатив имеет вид:

```
Switch : =      0;
L1  : = . ..
L2  : = . ..
L3  : = . ..
if L1 then Swich : = 1;
if L2 then Swich : = 2;
if L3 then Swich : = 3;
case Swich of 1: begin
 {Действие  при L1=True} end; 2 : begin
 {Действие при L2=True} end; 3:begin
 {Действие при L3=True} end; else begin
 {Вывод сообщения об ошибочном кодировании модуля} end; end;    {End of
Case}
```

2.3. ЦИКЛЫ

Оператор цикла имеет одну точку входа, последовательность операторов, которые составляют цикл, и одну или несколько точек выхода. Чтобы циклы завершались, с точкой выхода бывает связано условие, которое определяет, следует сделать выход или продолжить выполнение цикла. Циклы различаются числом, типом и расположением условий выхода. Универсальные циклы в Паскале имеют следующие конструкции.

Цикл while-do	**Цикл repeat-until**
{Подготовка}	*{Подготовка}*
While L do	repeat
begin	*{тело цикла}*
{тело цикла}	until L;
end;	

здесь L -логическое выражение, которое при значении *True* является условием продолжения выполнения *while-do* или условием окончания выполнения *repeat-until.* Подготовка и тело цикла являются цепочками функциональных узлов.

Тело цикла выполняется столько раз, сколько и весь цикл. При равноценности, из двух конструкций выбирают ту, запись которой короче. Операторы цикла наиболее трудны: в них легко сделать ошибку, особенно на границах цикла.

Очень часто количество итераций цикла известно заранее: это либо константа, известная при написании программы, либо значение, вычисляемое перед началом цикла. Цикл со счетчиком можно запрограммировать следующим образом:

```
for <параметр_цикла> := <нач_знач>
```

```
to <кон знач> do <оператор>;
```

здесь for, to, do — зарезервированные слова; <параметр _цикла> — переменная любого перечисляемого типа.
Цикл выполняется для каждого из значений от <нач знач>
и до <кон_знач>.

2.4. ОПЕРАТОРЫ ПЕРЕХОДА

Оператор безусловного перехода имеет следующий вид: goto, здесь goto — зарезервированное слово: <метка> — метка.

Метка - это произвольный идентификатор, позволяющий именовать некоторый оператор программы и таким образом ссылаться на него.

Можно теоретически показать, что достаточно *if-* и *while-операторов,* чтобы записать любую необходимую управляющую структуру. Однако есть несколько вполне определенных ситуаций, где лучше использовать оператор *goto.*

Первая состоит в том, что многие циклы не могут завершаться в точке входа, как этого требует цикл *while.*

Вторая ситуация, которую легко запрограммировать с помощью *goto,* - выход из глубоко вложенного вычисления. Например, если глубоко внутри вызовов процедур обнаружена ошибка, что делает неверным все вычисление. В этой ситуации естественно запрограммировать выдачу сообщения об ошибке и возвратить в исходное состояние все вычисление. Однако для этого требуется сделать возврат из многих процедур, которые должны знать, что произошла ошибка. Проще и понятнее выйти в основную программу с помощью *goto.*

В языке *С* нет никаких средств для обработки этой ситуации (не подходит даже *goto* по причине ограниченности рамками отдельной процедуры), поэтому для обработки серьезных ошибок нужно использовать средства операционной системы.

В современных языках *Object Pascal, Ada, C++* и *Eiffel* есть специальные языковые конструкции, так называемые исключения, которые непосредственно решают и эту проблему.

2.5. ПОДПРОГРАММЫ. ПРОЦЕДУРЫ И ФУНКЦИИ

Часто некоторую последовательность инструкций требуется повторить в нескольких местах программы. Чтобы программисту не приходилось тратить время и усилия на копирование этих инструкций, в большинстве языков программирования предусматриваются средства для организации подпрограмм. Таким образом, программист получает возможность присвоить последовательности инструкций произвольное имя и использовать это имя в качестве сокращенной записи в тех местах, где встречается соответствующая последовательность инструкций.

Подпрограмма -- некоторая последовательность инструкций, которая может повторяться в нескольких местах программы.

Процедурой называется особым образом оформленный фрагмент программы, имеющий собственное имя (идентификатор), который выполняет некоторую четко определенную операцию над данными, определяемыми параметрами.

Упоминание имени процедуры в тексте программы приводит к ее активизации и называется вызовом процедуры. Вызов может быть осуществлен из любой точки программы. При каждом таком вызове могут пересылаться различные параметры. Сразу после активизации процедуры начинают выполняться входящие в нее операторы, после выполнения последнего из них управление возвращается обратно в основную программу, и выполняются

операторы, стоящие непосредственно за оператором вызова процедуры.

Описание процедуры состоит из заголовка и тела. Заголовок процедуры имеет вид:

```
procedure <имя> [ (<сп.ф.п.>) ] ;
```

здесь <имя> имя процедуры (правильный идентификатор); <сп. ф. п. > - список формальных параметров.

Список формальных параметров необязателен и может отсутствовать. Если же он есть, то в нем должны быть перечислены имена формальных параметров и их типы, например procedure MyProc (a: Real; b: Integer; c: Char);

Функция отличается от процедуры тем, что результат ее работы возвращается в виде значения этой функции, и, следовательно, идентификатор функции может использоваться наряду с другими операндами в выражениях. Описание функции состоит из заголовка и тела. Заголовок функции имеет следующий вид:

function <имя> [(<сп.ф.п.>)]: <тип>;

здесь <тип> — тип возвращаемого функцией результата. Заголовок функции может иметь следующий вид:

function MyFunc (a, b: Real): Real;

Операторы тела подпрограммы рассматривают список формальных параметров как своеобразное расширение раздела описаний: все переменные из этого списка могут использоваться в любых выражениях внутри подпрограммы. Так осуществляется настройка алгоритма подпрограммы на конкретную задачу. Работа с процедурами и функциями отличаются в следующем:

1) в заголовке функции помимо описания формальных параметров обязательно указывается тип возвращаемого ею результата;
2) для возврата функцией значения в точку вызова среди ее операторов должен быть хотя бы один, в котором имени функции или переменной *Result* присваивается значение результата;
3) вызов процедуры выполняется отдельным оператором;
4) вызов функции может выполняться там, где допускается ставить выражение, например, в правой части оператора присваивания.

2.6. ПЕРЕДАЧА ПАРАМЕТРОВ

Для обмена информацией между основной программой и процедурой используется один или несколько параметров вызова. Они перечисляются за именем процедуры и вместе с ним образуют оператор ее вызова.

Механизм замены формальных параметров на фактические позволяет нужным образом настроить алгоритм, реализованный в подпрограмме. Компилятор обычно следит за тем, чтобы количество и тип формальных параметров строго соответствовали количеству и типам фактических параметров в момент обращения к подпрограмме.

Смысл используемых фактических параметров зависит от того, в каком порядке они перечислены при вызове подпрограммы. Поэтому программист должен сам следить за правильным порядком перечисления фактических параметров при обращении к подпрограмме. Формальные параметры подпрограммы могут быть трех видов:

параметры-значения;
параметры-переменные;
параметры-константы.

Например, procedure MyProc (A: Real; var B: Real; **const C: String);**

здесь А - параметр-значение, в - параметр-переменная, С - параметр-константа.

Способ определения формального параметра очень важен для вызывающей программы. Если формальный параметр объявлен как параметр-значение или параметр-константа, то при вызове ему может соответствовать произвольное

выражение. Если формальный параметр объявлен как параметр-переменная, то при вызове подпрограммы ему должен соответствовать фактический параметр в виде переменной нужного типа.

Чтобы понять, в каких случаях использовать тот или иной тип параметров, нужно знать, как осуществляется замена формальных параметров на фактические в момент обращения к подпрограмме.

Если параметр определен как *параметр-значение,* то перед вызовом подпрограммы это значение вычисляется, полученный результат копируется во временную память и передается подпрограмме. Любые возможные изменения в подпрограмме параметра-значения никак не воспринимаются вызывающей программой, так как в этом случае изменяется копия фактического параметра.

Если параметр определен как *параметр-переменная,* то при вызове подпрограммы передается сама переменная, а не ее копия (фактически в этом случае подпрограмме передается адрес переменной). Изменение параметра-переменной приводит к изменению самого фактического параметра в вызывающей программе.

В случае *параметра-константы* в подпрограмму также передается адрес области памяти, в которой располагается переменная или вычисленное значение. Однако компилятор блокирует любые присваивания параметру-константе нового значения в теле подпрограммы.

Параметры-переменные используются как средство связи алгоритма, реализованного в подпрограмме, с внешним миром. С помощью этих параметров подпрограмма может передавать результаты своей работы вызывающей программе.

Однако описание всех формальных параметров как параметров-переменных нежелательно по двум причинам. Во-первых, это исключает возможность вызова подпрограммы с фактическими параметрами в виде выражений, что делает программу менее компактной. Во-вторых, в подпрограмме возможно случайное использование формального параметра, например, для временного хранения промежуточного результата, т.е. всегда существует опасность непреднамеренно испортить фактическую переменную.

По той же причине не рекомендуется использовать параметры-переменные в заголовке функции. Если результатом работы функции не может быть единственное значение, то логичнее использовать процедуру или нужным образом декомпозировать алгоритм на несколько подпрограмм.

Существует еще одно обстоятельство, которое следует учитывать при выборе вида формальных параметров. Как уже говорилось, при объявлении параметра-значения осуществляется копирование фактического параметра во временную память. Если этим параметром будет массив большой размерности, то существенные затраты времени и памяти на копирование при многократных обращениях к подпрограмме можно минимизировать, объявив этот параметр параметром-константой. Параметр-константа не копируется во временную область памяти, что сокращает затраты времени на вызов подпрограммы, однако любые его изменения в теле подпрограммы невозможны - за этим строго следит компилятор.

Нетипизированные параметры. Одним из свойств языка *Object Pascal* является возможность использования *нетипизированных* параметров.

Параметр считается нетипизированным, если тип формального параметра-переменной в заголовке подпрограммы не указан, при этом соответствующий ему фактический параметр может быть переменной любого типа. Нетипизированными могут быть только параметры-переменные: `procedure MyProc(var aParametr);`

Нетипизированные параметры обычно используются в случае, когда тип данных несуществен. Такие ситуации чаще всего возникают при разного рода копированиях одной области памяти в другую, например, с помощью процедур *BlockRead, BlockWrite, Move-Memory* и т. п.

Процедурные типы. Основное назначение процедурных типов — дать

программисту гибкие средства передачи функций и процедур в качестве фактических параметров обращения к другим процедурам и функциям. Для объявления процедурного типа используется заголовок процедуры (функции), в котором опускается ее имя, например:

```
type
Prod = procedure (a, b, c: Real; var d: Real);
Proc3 == procedure;
Fund = function: String;
Func2 = function (var s: String): Real;
```

В программе могут быть объявлены переменные процедурных типов, например:

```
var
p1 :   Proc1;
fl,  f2:  Func2;
ap:  array  [1..N]  of  Prod;
```

Переменным процедурных типов допускается присваивать в качестве значений имена соответствующих подпрограмм. После такого присваивания имя переменной становится синонимом имени подпрограммы.

2.7. **РЕКУРСИЯ**

Содержание и мощность рекурсивного определения, а также его главное назначение, состоит в том, что оно позволяет с помощью конечного выражения определить бесконечное множество объектов. Аналогично, с помощью конечного рекурсивного алгоритма можно определить бесконечное вычисление, причем алгоритм не будет содержать повторений фрагментов текста.

Рекурсия - это такой способ организации вычислительного процесса, при котором подпрограмма в ходе выполнения составляющих ее операторов обращается сама к себе.

Программы, в которых используются рекурсивные процедуры, отличаются простотой, наглядностью и компактностью текста. Такие качества рекурсивных алгоритмов вытекают из того, что рекурсивная процедура указывает что нужно делать, а нерекурсивная больше акцентирует внимание на том, как нужно делать.

Однако за эту простоту приходится расплачиваться неэкономным использованием оперативной памяти, так как выполнение рекурсивных процедур требует значительно большего размера оперативной памяти во время выполнения, чем нерекурсивных. При каждом рекурсивном вызове для локальных переменных, а также для параметров процедуры, которые передаются по значению, выделяются новые ячейки памяти.

Таким образом, какой-либо локальной переменной *A* на разных уровнях рекурсии будут соответствовать различные ячейки памяти, которые могут иметь разные значения.

Глубиной рекурсии называется максимальное число рекурсивных вызовов процедуры без возвратов, которое происходит во время выполнения программы.

В общем случае любая рекурсивная процедура *Rec* включает в себя некоторое множество операторов *S* и один или несколько операторов рекурсивного вызова.

Безусловные рекурсивные процедуры приводят к бесконечным процессам, и на эту проблему нужно обратить особое внимание, так как практическое использование процедур с бесконечным самовызовом невозможно.

Следовательно, главное требование к рекурсивным процедурам заключается в том, что *вызов рекурсивной процедуры должен выполняться по условию,* которое на каком-то уровне рекурсии станет *ложным.*

Если условие истинно, то рекурсивный спуск продолжается. Когда оно становится ложным, то спуск заканчивается и начинается поочередный рекурсивный возврат из всех вызванных на данный момент копий рекурсивной

процедуры. Структура рекурсивной процедуры может принимать три разных формы:

1) форма с выполнением действий до рекурсивного вызова (на рекурсивном спуске);

```
procedure Rec; begin
S;
 if условие then Rec; end;
```

2) форма с выполнением действий после рекурсивного вызова (на рекурсивном возврате);

```
procedure Rec; begin
if условие then Rec;
  S; end;
```

3) форма с выполнением действий как до, так и после рекурсивного вызова (с выполнением действий как на рекурсивном спуске, так и на рекурсивном возврате).

```
procedure Rec; begin
S1;
if условие then Rec; S2;end;
```

Все формы рекурсивных процедур находят применение на практике. Многие задачи, в том числе вычисление факториала, безразличны к тому, какая используется форма рекурсивной процедуры. Однако есть классы задач, при решении которых программисту требуется сознательно управлять ходом работы рекурсивных процедур и функций. Такими, в частности, являются задачи, использующие списковые и древовидные структуры данных.

ТЕХНОЛОГИЯ СТРУКТУРНОГО ПРОГРАММИРОВАНИЯ

3.1. ПОНЯТИЕ СТРУКТУРНОГО ПРОГРАММИРОВАНИЯ

Исторически сложилось так, что императивные языки в настоящее время доминируют в программировании. Однако исследования, проведенные в 70-80-х годах XX века, показали, что аппликативная методика обеспечивает более эффективные способы верификации программ и доказательство их корректности. Это видно из блок-схем, представленных на рис. 4. На рис. 4, *а* изображена блок-схема, типичная для программ 60-х годов XX века. В ней нет никакой явной структуры. Такие программы называют про-граммами-спагетти. Из-за большого числа нерациональных передач управления назад и вперед трудно понять, каково состояние программы в каждый момент времени в процессе ее выполнения.

На рис. 4, б приведена более структурированная конструкция. Каждый сегмент данной блок-схемы можно заключить в пунктирный прямоугольник. Каждый из таких прямоугольников на этой схеме будет иметь одну точку входа и одну точку выхода. Эту программу можно рассматривать как композицию четырех функций-подпрограмм, и поведение программы можно определить как функцию, которая получает данное состояние на входе выделенного пунктиром прямоугольника и преобразует его в результирующее состояние на выходе из него. Писать сложные программы в тысячи и десятки тысяч строк без расчленения на самостоятельные фрагменты, т. е. без структурирования, просто невозможно.

Структурное программирование - подход, при котором для передачи управления в программе используются только три конструкции, допускающих последовательную, условную и итеративную передачи управления. При этом безусловная передача управления например, оператором *goto* запрещается.

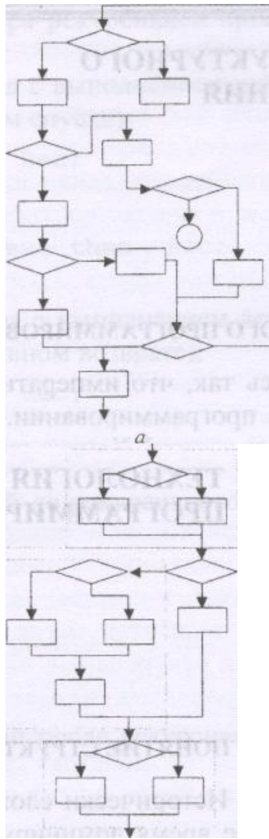

Рис. 4. Аппликативные методы в императивных языках

В результате каждая сложная команда в программе, являющаяся комбинацией последовательных, условных и циклических операторов, имеет только одну точку входа и одну точку выхода, что дает возможность разбиения программы на относительно самостоятельные фрагменты.

Структурное программирование является результатом применения аппликативных методов к императивным программам. Для этого используются процедурно-ориентированные языки, в которых имеется возможность описания программы как совокупности процедур. Процедуры могут вызывать друг друга, и каждая из них может быть вызвана основной программой, которую также можно рассматривать как процедуру.

Структурный подход к программированию представляет собой методологию создания программ. Его внедрение обеспечивает:

повышение производительности труда программистов при написании и контроле программ;

получение программ, которые более пригодны для сопровождения, так как состоят из отдельных модулей;

создание программ коллективом разработчиков;

окончание создания программ в заданный срок.

В структурированных программах обычно легко прослеживается основной алгоритм, они удобнее в отладке и менее чувствительны к ошибкам программирования. Эти свойства являются следствием важной особенности подпрограмм, каждая из которых представляет собой во многом самостоятельный фрагмент программы, связанный с основной программой лишь с помощью нескольких параметров. Такая самостоятельность подпрограмм позволяет локализовать в них все детали программной реализации того или иного алгоритмического действия, и поэтому изменение этих деталей, например в процессе отладки, обычно не приводит к изменениям основной программы.

3.2. ПРИНЦИП УТАИВАНИЯ ИНФОРМАЦИИ

Концепция структурного программирования предполагает разбиение программы на отдельные компоненты согласно принципу утаивания информации.

Принцип утаивания информации заключается в том, что идентификаторы локальных объектов (имена констант, типов, переменных, процедур, функций, меток, и полей в записях переменных), то есть тех, которые используются только внутри заданной последовательности инструкций, не должны иметь смысла за пределами этих инструкций.

Процедуры и функции выступают как естественные текстовые единицы, с помощью которых ограничивается область существования локальных идентификаторов.

Идентификатором называется строка символов, используемая для идентификации или именованная (последовательность букв, цифр и знаков подчеркивания, которая начинается с буквы или символа подчеркивания и не содержит пробелов).

Областью видимости (действия) идентификатора называется часть программы, где он может быть использован.

Область видимости идентификаторов определяется местом их объявления. Если идентификаторы допускается использовать только в рамках одной процедуры или функции, то такие идентификаторы называются *локальными.* Если действие идентификаторов распространяется на несколько вложенных (не менее одной) процедур и/или функций, то такие идентификаторы называются *глобальными.* Правила определения области видимости для идентификаторов состоят в следующем:

1) действуют все идентификаторы, определенные внутри процедуры/функции;

2)действуют все идентификаторы окружающего контекста, если их имена отличаются от имен, объявленных внутри процедуры/функции;

3)локальные идентификаторы процедуры/функции во внешнем окружении не действовуют никогда;

4)в случае совпадения имен глобального и локального идентификаторов действует только внутренний, локальный идентификатор, независимо от того совпадают они по типу, или нет.

3.3. МЕТОДЫ СТРУКТУРНОГО ПРОГРАММИРОВАНИЯ

При проектировании любого изделия, в том числе и алгоритма, на ранних стадиях основное внимание обращается на самые главные проблемы, и искусственно упускаются из виду многие частные детали. Поэтому наиболее общая тактика программирования состоит в разложении процесса на отдельные действия. На каждом таком шаге декомпозиции нужно удостовериться, что:
решения частных задач приводят к решению общей задачи;

данная последовательность отдельных действий наиболее рациональна;

осуществленная декомпозиция позволяет получить инструкции, по своему смыслу наиболее близкие к языку, на котором впоследствии будет написана программа.

Нисходящий подход к разработке программных систем. В соответствии с этим методом создание программы начинается сверху, т.е. с разработки самого главного, генерального алгоритма. Так как на верхнем уровне обычно еще не ясны детали реализации той или иной части программы, то эти части следует заменить временными заглушками .

Прогон незаконченной программы (перед заменой заглушек реально работающими процедурами) дает определенную уверенность перед разработкой и реализацией алгоритмов нижнего уровня. Если реализуемый в заглушке алгоритм достаточно сложен, его вновь структурируют, выделяя главный алгоритм и применяя новые заглушки и т.д. (*Заглушка* - заменяющая компонента, которая временно используется *в* программе с тем, чтобы можно

было продолжать ее разработку, т. е. компилирование или тестирование, до того времени, когда эта компонента будет сделана в надлежащем виде.). Процесс продолжается вниз до тех пор, пока не будет создан полностью работоспособный вариант программы.

На практике «чистую» нисходящую разработку осуществить невозможно. На одной из более поздних стадий часто обнаруживается, что некоторый выбор, сделанный ранее, был неадекватным и это приводит к необходимости итеративной разработки.

Восходящий подход к разработке программ. В этом случае осуществляется последовательное построение программы из уже имеющихся элементов, начиная с примитивов, предоставляемых выбранным языком программирования. Этот процесс заканчивается получением требуемой готовой программы. На каждом этапе из имеющихся элементов строятся более мощные элементы. Эти элементы будут использоваться на следующем этапе для построения еще более мощных элементов, и так далее до тех пор, пока не будут получены элементы, из которых можно непосредственно составить требуемую программу.

На практике восходящая разработка в чистом виде также как и нисходящая невозможна. Построение каждого нового элемента должно сопровождаться просмотром вперед с целью проверки, удовлетворяет ли он требованиям к разрабатываемой программе; но даже и при таком подходе на более позднем этапе часто обнаруживается, что использованная ранее последовательность построения была выбрана неправильно и требуется новая итерация.

При конструировании новых алгоритмов обычно доминирует нисходящий метод. При адаптации программ к несколько измененным требованиям предпочтение зачастую отдается восходящему методу. Оба этих метода позволяют разрабатывать *структурированные* программы.

3.4. СТРУКТУРНАЯ СХЕМА ПРОГРАММЫ И СРЕДСТВА ДЛЯ ЕЕ ИЗМЕНЕНИЯ

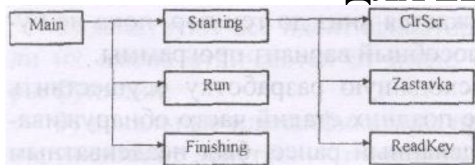

Рис. 5. Схема иерархии простой программы

В понятие *структуры программы (program structure)* включается состав и описание связей всех модулей, которые реализуют самостоятельные функции программы и описание носителей вводимых и выводимых данных, а также данных, участвующих в обмене между отдельными подпрограммами.

Для разработки больших и сложных программ программисту необходимо овладеть специальными приемами получения рациональной структуры программы, которая обеспечивает почти двукратное сокращение объема программирования и многократное сокращение

Подчиненность модулей программы отражается в схеме иерархии. Однако последняя не отражает порядок их вызова или функционирование программы. Схема иерархии может иметь вид, показанный на рис. 5. Она, обычно, дополняется расшифровкой функций, выполняемой модулями.

Перед составлением схемы иерархии целесообразно составить внешние спецификации программы и составить функциональные описания программы вместе с описанием переменных-носителей данных. Особое внимание следует уделять иерархии типов структурированных данных и их комментированию.

Расчленение программы на подпрограммы производится по принципу от общего к частному, более детальному. Процесс составления функционального описания и составления схемы иерархии является итерационным, а выбор наилучшего варианта является многокритериальным. Расчленение должно обеспечивать удобный порядок ввода частей в эксплуатацию.

Схеме иерархии можно придать любой топологический рисунок. *Фрагменты с вертикальными вызовами* могут быть преобразованы в вызовы одного уровня посредством введения дополнительного модуля, который может не выполнять никаких полезных функций с точки зрения алгоритма программы. Функция нового модуля может состоять лишь в мониторинге, то есть вызове других модулей в определенном порядке.

Фрагменты с горизонтальными вызовами на одном уровне могут быть преобразованы в вертикальные вызовы модулей разных уровней посредством введения дополнительных переменных, которые не могли быть получены декомпозицией функционального описания на подфункции. Эти дополнительные переменные обычно имеют тип целый или логический и называются флагами, семафорами, ключами событий. Их смысл обычно характеризуется фразой: в зависимости от следующей предыстории действий, выполнить такие-то действия.

В процессе проектирования нужно сделать несколько проектных итераций, каждый раз генерируя новую схему иерархии, и сравнить эти иерархии по данным критериям для отбора лучшего варианта.

Ключ - значение переменной, используемое для подтверждения полномочий на доступ к некоторой информации или подпрограмме.

Флаг — переменная, значение которой свидетельствует о том, что некоторый аппаратный или программный компонент находится в определенном состоянии или что для него выполняется определенное условие. Флаг используется для реализации условного ветвления и прочих процессов принятия решений.

Семафор - тип данных специального назначения, который является средством управления доступом к критическому ресурсу со стороны совместно идущих последовательных процессов.

Над семафором можно производить только две операции (не считая создания и аннулирования): *операцию ожидания* (занятия) и *операцию сигнализации* (освобождения). Семафор принимает целое значение, которое не может быть отрицательным. Операция ожидания уменьшает значение семафора на единицу, когда это можно сделать, не получая при этом отрицательного значения, и это означает, что свободный ресурс используется. Операция сигнализации увеличивает значение семафора на единицу, что означает освобождение ресурса.

Критический ресурс - ресурс, который в каждый момент времени используется не более чем одним процессом. Когда требуется, чтобы несколько асинхронных процессов координировали свой доступ к критическому ресурсу, используется управляемый доступ через семафор.

3.5. КРИТЕРИИ ОЦЕНКИ КАЧЕСТВА СТРУКТУРНОЙ СХЕМЫ ПРОГРАММЫ

Первый вариант структурной схемы, полученный путем простого членения функций программы на подфункции с указанием переменных, необходимых для размещения данных, чаще всего не является оптимальным и требуются проектные итерации для улучшения топологии схемы. Эти действия обычно выполняются методом «проб и ошибок». Каждый новый вариант сравнивается с предшествующим по описанным ниже критериям:

1) полнота выполнения специфицированных функций;
2) возможность быстрого и дешевого пополнения новыми, ранее не специфицированными функциями;
3) обозримость (понятность) для проектировщика составных частей программы;
4) максимальная независимость отдельных частей программы;
5) возможность связывания подпрограмм редактором связей;
6) достаточность оперативной памяти;

7) влияние топологии схемы иерархии на скорость выполнения программы при использовании динамической загрузки программы и механизма подкачки страниц;

8) отсутствие разных модулей со сходными функциями. Один и тот же модуль должен вызываться на разных уровнях схемы иерархии;

9) достижение такого графика работы коллектива программистов при реализации программы, который обеспечивает равномерную загрузку коллектива;

10) всемерное сокращение затрат на тестирование программы. Хорошая схема иерархии в 2-5 раз сокращает затраты на тестирование по сравнению с первоначальным вариантом;

11) использование в данном проекте как можно большего числа проработанных в предшествующих проектах модулей и библиотек при минимальном объеме изготавливаемых заново частей.

Генерация вариантов прекращается при невозможности дальнейших улучшений. Рациональная структура программы обеспечивает сокращение общего объема текстов в *2-3* раза, что соответственно удешевляет создание программы и ее тестирование, на которое обычно приходится не менее 60% от общих затрат. При этом облегчается и снижается стоимость сопровождения программы.

3.6. МОДУЛЬНОЕ ПРОГРАММИРОВАНИЕ

Реализация принципа структурного программирования осуществляется с использованием макрокоманд и механизмов вызова подпрограмм. Эти же механизмы подходят и для реализации модульного программирования, которое можно рассматривать как часть структурного подхода.

Необходимо различать использование слова *модуль,* когда имеется в виду единица дробления большой программы на отдельные блоки (которые могут быть реализованы в виде процедур и функций) и когда имеется ввиду синтаксическая конструкция языков программирования *(unit* в *Object Pascal).*

Модульное программирование — это организация программы как совокупности независимых блоков, называемых модулями, структура и поведение которых подчиняются определенным правилам.

Концепцию модульного программирования можно сформулировать в виде нескольких понятий и положений:

1) большие задачи разбиваются на ряд более мелких, функционально самостоятельных подзадач — модулей, которые связаны между собой только по входным и выходным данным;

2) модуль представляет собой «черный ящик» с одним входом и одним выходом. Это позволяет безболезненно производить модернизацию программы в процессе ее эксплуатации, облегчает ее сопровождение, а также позволяет разрабатывать части программодного проекта на разных языках программирования;

3) в каждом модуле должны осуществляться ясные задачи. Если назначение модуля непонятно, то это означает, что декомпозиция на модули была проведена недостаточно качественно. Процесс декомпозиции нужно продолжать до тех пор, пока не будет ясного понимания назначения всех модулей и их оптимального сочетания;

4) исходный текст модуля должен иметь заголовок и интерфейсную часть, где отражаются назначение модуля и все его внешние связи;

5) в ходе разработки модулей программы следует предусматривать специальные блоки операций, учитывающие реакцию на возможные ошибки в данных или в действиях пользователя.

Большое значение в концепции модульного программирования придается организации управляющих и информационных связей между модулями

программы, совместно решающими одну или несколько больших задач.

При работе с модулями нужно помнить их основное отличие от процедур и функций. Традиционные правила сферы действия глобальных и локальных переменных для модулей не работают. Эта языковая конструкция разработана так, чтобы исключить влияние глобальных переменных, объявленных в главной программе, на внутренние описания модуля. Поэтому, если возникает необходимость ввести доступные для всех блоков программы глобальные описания то следует создать модуль глобальных объявлений и включить его в список импорта всех модулей, где нужны его описания.

3.7. **СТРУКТУРА МОДУЛЯ В *OBJECT PASCAL***

Object Pascal имеет различные средства для структурирования программ. На нижнем уровне деления (для элементарных подзадач) чаще всего используются процедуры и функции, а на верхнем уровне (для больших задач) используются модули.

В среде *Delphi* каждой форме обязательно соответствует свой модуль, что позволяет локализовать все свойства окна в отдельной программной единице. Кроме этого, невизуальные алгоритмические действия также оформляются в виде отдельных модулей. Первая строка модуля начинается с ключевого слова:

```
unit <идентификатор_модуля>;
```

Для правильной работы среды программирования это имя должно совпадать с именем дискового файла, в который помещается исходный текст модуля. Далее следует

{Интерфейсный раздел} interface

где описывается взаимодействие данного модуля с другими пользовательскими и стандартными модулями, а также с главной программой.

Здесь содержатся объявления всех глобальных объектов модуля (типов, констант, переменных и подпрограмм), которые должны стать доступными основной программе и/или другим модулям. При объявлении глобальных подпрограмм в интерфейсной части указывается только их заголовок.

Связь модуля с другими модулями устанавливается специальным предложением:

{Список импорта интерфейсного раздела} uses
<список_модулей>

В этом списке через запятые перечисляются идентификаторы модулей, информация интерфейсных частей которых должна быть доступна в данном модуле.

{Список экспорта интерфейсного раздела} const type var
procedure function

Список экспорта состоит из подразделов описания констант, типов, переменных, заголовков процедур и функций, которые определены в данном модуле, но использовать которые разрешено во всех других модулях и программах, включающих имя данного модуля в своей строке uses. Для процедур и функций здесь описываются только заголовки, но с обязательным полным описанием формальных параметров.

{Раздел реализации) implementation

В этом разделе указывается реализационная (личная) часть описаний данного модуля, которая недоступна для других модулей и программ.

{Список импорта раздела реализации) uses

В этом списке через запятые перечисляются идентификаторы модулей, информация интерфейсных частей которых должна быть доступна в данном модуле. Здесь целесообразно описывать идентификаторы всех необходимых модулей, информация из которых не используется в описаниях раздела interface данного модуля.

{Подразделы внутренних для модуля описаний} label const type var procedure function

В этих подразделах описываются метки, константы, типы, переменные, процедуры и функции, которые описывают алгоритмические действия, выполняемые данным модулем, и которые являются «личной собственностью» исключительно только данного модуля. Эти описания недоступны ни одному другому модулю.

Исполняемая часть содержит описания подпрограмм, объявленных в интерфейсной части. Описанию подпрограммы должен предшествовать заголовок, в котором можно опускать список формальных параметров и тип результата для функции. Если заголовки указаны с параметрами, то их список должен быть идентичен такому же списку для соответствующей процедуры или функции в разделе interface.

{Раздел инициализации} initialization

В этом разделе между ключевыми словами initialization и finalization располагаются операторы начальных установок, необходимых для запуска корректной работы модуля. Эти операторы исполняются до передачи управления основной программе и обычно используются для подготовки ее работы. Операторы разделов инициализации модулей, используемых в программе, выполняются при начальном запуске программы в том же порядке, в каком идентификаторы модулей описаны в предложениях uses файла проекта. Если операторы инициализации не требуются, то зарезервированное слово initialization может быть опущено.

{Раздел завершения) finalization

Раздел завершения finalization является необязательным и может присутствовать только вместе с разделом инициализации initialization. В разделе завершения располагается список операторов, которые будут выполняться при завершении модуля, что обычно происходит при окончании работы приложения. Разделы finalization модулей приложения выполняются в порядке, противоположном выполнению разделов initialization этих модулей.

Раздел завершения используется, как правило, для освобождения ресурсов, которые выделяются приложению в разделе инициализации. Это гарантирует корректное завершение приложения, что особенно это важно, когда приложение заканчивается по возникновению исключительных ситуаций.

ТЕХНОЛОГИЯ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

4.1. ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ ПОДХОД

В основу структурного подхода положены структуризация и декомпозиция окружающего мира. Задача любой сложности разбивается на подзадачи, а те, в свою очередь, разбиваются далее, пока каждая подзадача не станет простой, соответствующей модулю (подпрограмме), выполняющему строго одно действие. Методы структурного проектирования используют модули в качестве строительных блоков программы, а структура программы представляется иерархией подчиненности модулей.

При объектно-ориентированном подходе в качестве строительных блоков используются объекты, содержащие свои собственные коды и данные. Структура программ при объектно-ориентированном подходе представляется графом взаимодействия объектов, а не деревом иерархии, как это имеет место в структурном проектировании.

Объектно-ориентированный анализ (object-oriented analysis) -это методология, при которой требования к системе воспринимаются с точки зрения классов и объектов, выявленных в предметной области. Объектно-ориентированный анализ направлен на создание моделей реальной

действительности на основе объектно-ориентированного подхода.

Объектно-ориентированное проектирование (object-oriented design) - это методология проектирования, соединяющая в себе процесс объектной декомпозиции и приемы представления моделей, отражающих логическую (классы и объекты) и физическую структуру системы (процессы и деление на компоненты, файлы или модули), а также ее статические и динамические аспекты.

Объектно-ориентированное программирование (object-oriented programming) - это технология реализации программ, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

На результатах объектно-ориентированного анализа формируются модели, на которых основывается объектно-ориентированное проектирование, а оно, в свою очередь, создает фундамент для окончательной реализации системы с использованием языковых средств объектно-ориентированного программирования.

4.2. ОСНОВНЫЕ ПОНЯТИЯ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

Понятие объектно-ориентированного программирования определяет три основные концепции, при соблюдении которых программа будет объектно-ориентированной:

объектно-ориентированное программирование использует в качестве базовых элементов *классы,* порождающие *объекты;*

в процессе выполнения программы может одновременно использоваться несколько объектов, порожденных от одного класса (экземпляров реализации класса);

классы организованы иерархически *(иерархия* означает «быть частью»).

Класс - представляет собой объединяющую концепцию набора объектов, имеющих общие характеристики. Класс также определяет интерфейс с окружающим миром, посредством которого осуществляется взаимодействие с отдельными объектами.

Класс является описанием того, как будет выглядеть и вести себя его представитель. Поэтому класс проектируют как образование, отвечающее за создание своих новых представителей (экземпляров или объектов). Создание объектов и их уничтожение осуществляется с помощью особых методов - называемых *конструктором* и *деструктором.*

Объект — это структурированная переменная типа *класс,* содержащая всю информацию о некотором физическом предмете или реализуемом в программе понятии. Все объекты - представители данного класса аналогичны друг другу в том смысле, что они имеют один и тот же набор операций - методов.

Объект, как логическая единица, содержит следующие данные и операции (методы с кодом алгоритма) в отдельном участке памяти:

поля объекта (или атрибуты исходных данных), значения которых определяют текущее состояние объекта;

методы объекта, которые реализуют действия (выполнение алгоритмов) в ответ на их вызов в виде преданного сообщения;

свойства — часть методов, которые определят поведение объекта, то есть его реакцию на внешние воздействия.

При объявлении классов определяются описанные выше три характеристики объектов: поля, методы и свойства, а также указывается предок данного класса.

Объекты в программах воспроизводят все оттенки явлений реального мира: «рождаются» и «умирают»; меняют свое состояние; запускают и останавливают процессы; «убивают» и «возрождают» другие объекты.

4.3. ПРИНЦИПЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ: ИНКАПСУЛЯЦИЯ, НАСЛЕДОВАНИЕ И ПОЛИМОРФИЗМ

В основе классов лежат три фундаментальных принципа - *инкапсуляция, наследование* и *полиморфизм.*

Инкапсуляция. Проектирование программных и технических систем базируется на том условии, что никакая подсистема данного уровня не должна зависеть от устройства любой другой подсистемы этого уровня. Такая независимость внутреннего устройства одного объекта от внутреннего устройства другого называется *инкапсуляцией.*

Принцип инкапсуляции использовался в технологии модульного программирования. В модуле в явной форме введена инкапсуляция путем разделения его на секции интерфейса и реализации.

В объектно-ориентированном программировании принцип инкапсуляции используется для изоляции класса от остальных частей программы, чтобы сделать его самодостаточным для решения конкретной задачи. Например, класс *TForm* в среде *Delphi* содержит (инкапсулирует в себе) все необходимое для создания *Windows-окна,* класс *TMemo* представляет собой полнофункциональный текстовый редактор, класс *TTimer* обеспечивает работу программы с таймером и т. д.

Инкапсуляция достигается путем совмещения в одной записи языка программирования структур данных с процедурами и функциями, которые манипулируют полями данных этой записи, для получения нового типа данных ~ класса. Инкапсуляция позволяет защитить по интерфейсу доступ к полям и методам. Доступ разрешается лишь к открытым методам и полям. Полная совокупность методов и тонкости их реализаций являются скрытыми.

```
type
TMyClass = class
IntField: Integer;
function MyFunc(a: Integer): Integer;
   procedure MyProc; end;
```

Путем использования принципа инкапсуляции появляется возможность осуществлять обмен готовыми к работе программными заготовками. Например, библиотека классов *Delphi* - это, фактически, набор кирпичиков для построения прикладных программ.

Наследование. Число абстракций в сложных программных системах намного превышает наши возможности их осознания. Инкапсуляция частично помогает устранить это препятствие, убирая из поля зрения внутреннее содержание абстракций.

Однако значительное упрощение понимания сложных задач достигается за счет усложнения иерархии. Под иерархией здесь понимается упорядочение абстракций, расположение их по уровням. Усложнение иерархии от уровня к уровню достигается за счет наследования.

Принцип наследования оперирует с понятиями «предок - потомок» и предусматривает расширение набора свойств наследника за счет принятия всех свойств предка.

Любой класс может быть порожден от другого класса. Для этого при его объявлении указывается имя класса-родителя:

```
TChildCIass = class (TParentClass )
```

Порожденный класс автоматически наследует поля, методы и свойства своего родителя и может добавлять их новыми. Таким образом, принцип наследования обеспечивает поэтапное создание сложных классов и разработку собственных библиотек классов.

Все классы в *Object Pascal* порождены от единственного родителя - класса

TObject. Этот класс не имеет полей и свойств, но включает в себя методы самого общего назначения, обеспечивающие весь жизненный цикл любых объектов - от их создания до уничтожения. Поэтому программист не может создать класс, который не был бы дочерним классом *TObject.* Следующие два объявления идентичны.

```
TaClass = class(TObject)  <==>  TaClass = class
```

Принцип наследования приводит к созданию ветвящегося дерева классов. Каждый потомок дополняет возможности своего родителя новыми и передает их своим потомкам. Например, класс *TPersistent* обогащает возможности своего родителя *TObject* тем, что он умеет сохранять данные в файле и получать их из него, в результате это умеют делать и все его потомки. Класс *TComponent,* в свою очередь, умеет взаимодействовать со средой разработчика и передает это умение своим потомкам. *TControl* не только способен работать с файлами и средой разработчика, но он еще умеет создавать и обслуживать видимые на экране изображения, а его потомок *TWinControl* может создавать ***Windows-окна*** **и т. д.**

В *Object Pascal* возможно только так называемое одиночное наследование, но в реальном мире у потомка два родителя, поэтому в ряде языков (например, в C++) предусмотрен механизм множественного наследования. Множественное наследование более логично с точки зрения моделирования реального мира, однако, оно усложняет реализацию языков программирования.

Полиморфизм. Одним из базовых понятий технологии объектно-ориентированного программирования является *полиморфизм.* Этот термин имеет греческое происхождение и приблизительно означает «много форм» *(poly* — много, *morphos* — форма).

Полиморфизм - это средство для придания различных значений одному и тому же событию в зависимости от типа обрабатываемых данных. Этот принцип определяет различные формы реализации одноименного действия.

Целью полиморфизма является использование одного имени для задания общих для класса действий, причем каждый объект или класс иерархии имеет возможность по-своему реализовать это действие своим собственным, подходящим для него, кодом. Таким образом, полиморфизм является свойством классов решать схожие по смыслу проблемы разными способами.

В рамках *Object Pascal* поведенческие свойства класса определяются набором входящих в него методов. Этот принцип используется, когда требуется расширить свойства класса не путем добавления новых методов, а путем достраивания одного из методов или набора методов. Изменяя алгоритм того или иного метода в потомках класса, программист может придавать этим потомкам отсутствующие у родителя специфические свойства.

Для изменения метода необходимо перекрыть его в потомке, т. е. объявить в потомке одноименный метод и реализовать в нем нужные действия. В результате в объекте-родителе и объекте-потомке будут действовать два одноименных метода, имеющие различную алгоритмическую основу и придающие объектам разные свойства. В этом сущность полиморфизма объектов.

Кроме этого, в *Object Pascal* полиморфизм достигается не только механизмом наследования и перекрытия методов родителя, но и их виртуализацией, позволяющей родительским методам обращаться к методам своих потомков.

4.4. ПОЛЯ

Класс представляет собой единство трех сущностей -*полей, методов* и *свойств.* Объединение этих сущностей в единое целое достигается за счет применения инкапсуляции.

Полями называются инкапсулированные в классе данные. По аналогии с описанием переменных, описание поля указывает идентификатор, именующий

поле и тип данного этого поля. Поля могут быть любого типа, в том числе классами, например:

```
type
TMyClass = class
aIntField: Integer; aStrField: String;
aObjField: TObject;

end;
```

Каждый объект получает уникальный набор полей, но общий для всех объектов данного класса набор методов и свойств. Фундаментальный принцип инкапсуляции требует обращаться к полям только с помощью методов и свойств класса. Однако в *Object Pascal* разрешается обращаться к полям и напрямую. Для этого используются составные имена полей, содержащие имя объекта в качестве префикса.

```
type
TMyClass = class
aIntField: Integer; aStrField: String;

     end; var
    aObject: TMyClass; begin

aObject.aIntField : = 0;
aObject.aStrField : = 'Строка символов';

end;
```

Класс-потомок получает все поля своих предков и может дополнять их своими, но он не может переопределять их или удалять. Таким образом, чем ниже в дереве иерархии располагается класс, тем больше данных получают в свое распоряжение его объекты.

4.5. МЕТОДЫ

Методы представляют собой процедуры и функции, принадлежащие заданному объекту. Поэтому методы определяют поведение объекта. Для класса можно самостоятельно создать произвольное количество любых методов, необходимых для решения конкретных задач. Создание метода осуществляется в два этапа. Сначала следует описать метод в объявлении типа, а затем создать текст его реализации. Вот пример описания и определения метода:

```
type
TMyClass = class Work: Boolean; procedure
DoWork; end;

procedure TMyClass.DoWork; begin
   Work := True; end;
```

При определении тела метода необходимо использовать его полное имя с указанием класса. Вторая важная деталь - к любому полю объекта его метод может обратиться непосредственно. Доступ к методам класса, как и к его полям, возможен с помощью составных имен.

```
var
   aObject: TMyClass; begin

aObject.DoWork;

end;
```

Статические методы. Как уже говорилось, методы класса могут

перекрываться в потомках. Например:

```
type
TParentClass = class
 procedure DoWork; end;

TChildClass = class(TparentClass)
 procedure DoWork; end;
```

Потомки обоих классов могут выполнять сходную по названию процедуру DoWork, но, в общем случае, будут это делать по-разному. Такое замещение методов называется статическим, так как реализуется компилятором.

Статический метод DoWork работает подобно обычной процедуре или функции. Этот тип методов принимается по умолчанию. Адрес такого метода известен уже на стадии компиляции, и компилятор в тексте программы оформляет все вызовы данного метода как статические. Такие методы работают быстрее других, однако не могут быть перегружены для поддержки полиморфизма объектов.

Динамические и виртуальные методы. В *Object Pascal* гораздо чаще используется динамическое замещение методов на этапе прогона программы. Для реализации этого, метод, замещаемый в родительском классе, должен объявляться как динамический (с директивой *dynamic)* или виртуальный *(virtual).* тив такое объявление, компилятор создаст две таблицы - *DMT (Dynamic Method Table)* и *VMT (Virtual Method Table)* и поместит в них адреса точек входа соответственно динамических и виртуальных методов.

При каждом обращении к замещаемому методу компилятор вставляет код, позволяющий извлечь адрес точки входа в подпрограмму из той или иной таблицы. В классе потомке замещающий метод объявляется с директивой override, которая вызывает замещение строки описания исходного метода в *VMT* строкой описания нового метода. Получив такое указание, компилятор создаст код, который на этапе прогона программы поместит в родительскую таблицу точку входа метода класса-потомка, что позволит родителю выполнить нужное действие с помощью нового метода.

Например, родительский класс с помощью методов Show и Hide соответственно показывает что-то на экране или прячет изображение. Для создания изображения он использует метод Draw с логическим параметром:

```
type
TVisualObject  = class(TWinControl)
procedure  Hide;
procedure  Show;
   procedure  Draw(IsShow:  Boolean);  virtual; end; TVisualChildObject  =
class(TVisualObject)
   procedure  Draw(IsShow:  Boolean);  override; end;
```

Реализация методов Show и Hide выглядит следующим образом:

```
procedure  TVisualObject.Show; begin
   Draw(True); end;

procedure TVisualObject.Hide begin
   Draw(False); end;
```

Методы Draw у родителя и потомка имеют разную реализацию и создают разные изображения. В результате родительские методы Show и Hide будут прятать или показывать те или иные изображения в зависимости от конкретной реализации метода Draw у любого из своих потомков. Таким образом, динамическое связывание в полной мере реализует полиморфизм классов.

Разница между динамическими и виртуальными методами состоит в том, что таблица динамических методов *DMT* содержит адреса только тех методов, которые объявлены как *dynamic* в данном классе, в то время как таблица *VMT* содержит адреса виртуальных методов не только данного класса, но и всех его родителей. Большая по размеру таблица *VMT* обеспечивает более быстрый поиск,

в то время как при обращении к динамическому методу программа сначала просматривает таблицу *DMT у* объекта, затем - у его родительского класса и так далее, пока не будет найдена нужная точка входа.

Конструкторы и деструкторы. В *Object Pascal* объекты создаются с помощью вызова одного из конструкторов этого объекта. Конструктор отвечает за создание объекта, а также за выделение памяти и необходимую инициализацию полей. Он распределяет объект в динамической памяти и помещает адрес этой памяти в переменную *Self,* которая автоматически объявляется в классе. Конструктор не только создает объект, но и приводит его в состояние, необходимое для его дальнейшего использования.

У класса *TObject* конструктор называется *Create(),* так же он называются в подавляющем большинстве его потомков. Каждый объект содержит, по крайней мере, один такой конструктор, который может иметь различное число параметров разного типа — в зависимости от типа объекта.

Функцией деструктора является удаление объекта из памяти. По своей форме конструкторы и деструкторы являются процедурами, но объявляются с помощью зарезервированных слов Constructor и Destructor:

```
type
TMyClass = class
IntField: Integer;
Constructor Create(Value: Integer); Destructor
Destroy; end;
```

В отличие от C++, в *Object Pascal* конструкторы автоматически не вызываются. Каждый объект создается с помощью вызова его конструктора. Обращение к конструктору должно предварять любое обращение к полям и некоторым методам объекта. Синтаксис вызова конструктора следующий:

```
var
  MyObject:  TMyClass; begin
MyObject.IntField  :=  0;   {Ошибка!  Объект не  создан
конструктором}
MyObject  :=  TMyClass.Create;  //Нужно создать  объект MyObject.IntField
:=  0;    //и обратиться к его полю

MyObject.Free;          //Уничтожаем ненужный объект
end;
```

Особенность вызова конструктора заключается в том, что он вызывается с помощью ссылки на класс, а не на объект. В этом заложен глубокий смысл, так как экземпляр класса в момент вызова конструктора еще не создан. Однако код конструктора класса *TMyClass* находится в памяти и поэтому такой вызов вполне корректен.

При создании объекта с помощью конструктора компилятор гарантирует, что все поля объекта будут инициализированы. Все числовые поля будут обнулены, указатели примут значение *Nill,* а строки будут пусты.

С помощью метода *Free()* освобождается выделенная для объекта память после его использования. Этот метод проверяет, не равен ли объект значению *Nill,* и затем вызывает деструктор объекта - метод *Destroy(),* который освобождает всю выделенную память и выполняет другие действия по освобождению захваченных конструктором объекта ресурсов. В отличие от вызова конструктора, вызов метода *Free()* выполняется с помощью ссылки на экземпляр, а не на класс.

Большинство конструкторов реализует некоторые действия, необходимые для правильной работы объекта. Поэтому в конструкторе класса потомка следует сначала вызвать конструктор своего родителя, а затем уже осуществлять дополнительные действия. Вызов любого метода родительского класса достигается с помощью ключевого слова Inherited (унаследованный). Например:

```
Constructor TmyClass.Create(Value: Integer);
//Возможная реализация конструктора
begin
Inherited Create; {Вызываем унаследованный
конструктор}
  IntField := Value; {Реализуем дополнительные действия} end;
```

В большинстве примеров, поставляемых вместе с *Delphi,* нет вызовов конструкторов и деструкторов. Это объясняется тем, что любой компонент, попавший при визуальном проектировании в приложение из палитры компонентов, включается в определенную иерархию, которая замыкается на форме *TForm,* а для всех ее составных частей конструкторы и деструкторы вызываются автоматически. Создает и уничтожает формы глобальный объект с именем *Application.* Для объектов, создаваемых динамически (во время выполнения приложения), обязательно нужен вызов конструктора.

4.6. СВОЙСТВА

Свойства объекта представляют собой специальный механизм классов, регулирующий доступ к полям объекта. По отношению к компонентам свойства являются теми элементами, сведения о которых отображаются в окне *Object Inspector.*

Свойства объявляются с помощью зарезервированных слов property, read и write. Слова read и write считаются зарезервированными только в контексте объявления свойства. Обычно свойство связано с некоторым полем и указывает те методы класса, которые должны использоваться при записи в это поле или при чтении из него. Например:

```
type
TaClass = class
IntField: Integer;
function GetField: Integer;
procedure SetField(Value: Integer);
property IntegerValue: Integer read GetField
                               write SetField; end;
```

В контексте программы свойство ведет себя как обычное поле. Поэтому возможен следующий оператор присваивания:

```
aClass.IntField := NewValue;
```

Разница между этим оператором и оператором aCIass.IntegerValue := NewValue;

заключается в том, что при обращении к свойству автоматически подключается метод SetField, в котором могут реализовываться специфические действия.

Когда нет необходимости в специальных действиях при чтении или записи свойства, вместо имени соответствующего метода можно указывать имя поля. Если нужно, чтобы поле было доступно только для чтения или только для записи, следует опустить соответственно часть write или read.

Вся эта технология имеет два основных преимущества. Во-первых, она позволяет представить конечному пользователю некий интерфейс, полностью скрывающий реализацию объекта и обеспечивающий контроль за доступом к объекту. Во-вторых, она дает программисту возможность замещать методы в классах-потомках с целью обеспечения полиморфного поведения объектов.

4.7. ОПРЕДЕЛЕНИЕ ОБЛАСТИ ВИДИМОСТИ КЛАССА

Object Pascal предоставляет дополнительный контроль над доступностью

членов классов (полей и методов) с помощью директив private, protected, public, published. Синтаксис использования этих директив следующий:

```
type
 TMyClass = class private
AprivateVariable: Integer;
AnotherPrivateVariable: Boolean; protected
procedure AProtectedProcedure; function
ProtectMe: Byte; public
constructor APublicContructor; destructor
APublicKiller; published property AProperty
read AprivateVariable write APrivateVariable;
end;
```

За каждой из директив может следовать любое необходимое количество объявлений полей или методов. Эти директивы имеют следующий смысл.

private - эта часть объекта доступна только для кода, находящегося в одном модуле с другим кодом данного объекта. Директива private скрывает особенности реализации объекта от пользователей и защищает члены этого объекта от непосредственного доступа и изменения извне.

protected - члены объекта, описанные в этой секции, доступны для производных объектов, что позволяет скрыть внутреннее устройство объекта от пользователя и в то же время обеспечить необходимую гибкость и эффективность доступа к полям и методам объекта для его потомков.

public - описанные подобным образом члены объекта доступны в любом месте данной программы. В этой секции всегда описываются конструкторы и деструкторы.

published - для этой части объекта при компиляции будет сгенерирована информация о типе времени исполнения. Это даст возможность другим частям приложения получать информацию о части объекта, описанной в этой секции. В частности, подобная информация используется утилитой *Object Inspector* при построении списков свойств объектов.

В *Object Pascal* разрешается многократно объявлять любую секцию, причем порядок следования секций не имеет значения. Любая секция может быть пустой.

4.8. ПРИНЦИПЫ РАБОТЫ ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ ПРОГРАММ

Технология объектно-ориентированного программирования предполагает, что любая процедура или функция в программе представляет собой метод объекта некоторого класса, причем класс должен формироваться в программе, как только возникает необходимость описания новых объектов программирования. Каждый новый шаг в разработке алгоритма также должен представлять собой разработку нового класса на основе уже существующих классов. Таким образом, формируется иерархия классов и, в конце концов, вся программа будет представлять собой объект некоторого класса с единственным методом *run* (выполнить).

В технологии объектно-ориентированного программирования схема взаимодействия методов и данных принципиально иная, чем при технологии структурного программирования: метод, вызываемый для одного объекта, как правило, не вызывает другой метод непосредственно. Для начала он должен иметь доступ к другому объекту (создать, получить указатель, использовать внутренний объект в текущем объекте и т. д.), после чего он уже может вызвать для него один из известных методов. Таким образом, структура программы определяется взаимодействием объектов различных классов между собой. Взаимосвязь между объектами осуществляется посредством сообщений.

Большинство объектно-ориентированных систем организованы так, что их объекты состоят из видимых и приватных частей. Это делается с целью того,

чтобы не создавать лишних копий не меняющихся частей объектов, приводящее к разбрасыванию ресурсов памяти. Общие для всех объектов одного типа части кода запоминаются на уровне класса. Чаще всего сюда попадают коды методов и переменные класса, составляющие разделяемую часть объектов. Класс содержит информацию о том, какие переменные создавать, но запоминаются они самим экземпляром.

К достоинствам объектно-ориентированного программирования следует отнести:

исключение избыточного кода;

возможность защиты объектов от кода других частей программы;

поддержка повторного использования отдельных составляющих программ;

создание более открытых систем;

экономия времени за счет построения программы из готовых, отлаженных частей;

К недостаткам объектно-ориентированного программирования относят:

ухудшение быстродействия системы, которое обусловлено посылкой сообщений от одного объекта к другому. Обращение к методу может занимать в 2-2,5 раза больше времени, чем к обычной подпрограмме;

необходимость создания методов для доступа к запрещенным переменным объекта, а многочисленность методов приводит к излишнему количеству вызовов.

Тем не менее, достоинства объектно-ориентированных систем, как правило, перевешивают перечисленные недостатки. Опыт также показывает, что размер исполнимых модулей таких систем обычно меньше.

ТЕСТИРОВАНИЕ, ОТЛАДКА И ОПТИМИЗАЦИЯ ПРОГРАММ

5.1. ПРОГРАММНЫЕ ОШИБКИ

Программ без ошибок не существует. Практика пока-зывдет, что виновниками ошибок в программах чаще всего бывают сами программисты. Один из общих законов практического программирования состоит в том, что ни одна программа не дает желаемых результатов при первой попытке трансляции и выполнения. Некоторое представление о действительных причинах появления ошибок в работе программы дает следующее процентное соотношение источников сбоев:

Входные данные	1%
Ошибки пользователя	5%
Аппаратура	1%
Системное программное обеспечение	3%
Разработка системы	15%
Программирование	75%

Программист должен не только писать эффективные программы, но и находить в них всевозможные ошибки. Современная практика обучения программированию ориентирована, в основном, только на выполнение программистом первой половины своей работы. Это все равно, как если обучать летчика только взлету, предполагая, что с посадкой машины он как-нибудь разберется сам, если будет выполнять все операции взлета в обратном порядке.

Существуют два типа программных ошибок:

синтаксические ошибки — возникают из-за нарушения правил языка программирования. Такие ошибки обычно выявляются во время компиляции. Могут быть исключены сравнительно легко. Даже если не просматривать текст программы можно быть уверенным, что компилятор на стадии трансляции найдет ошибки и выдаст соответствующие предупреждения. Фактически поиск ошибок осуществляет компилятор, а их исправление — программист;

семантические (логические) ошибки - те, что приводят к некорректным вычислениям или ошибкам во время выполнения *(run-time error).* Семантические, ошибки устраняют обычно посредством выполнения программы с тщательно

подобранными проверочными данными, для которых известен правильный ответ.

5.2. ТЕСТИРОВАНИЕ

Тестирование (testing) - любой вид деятельности, в рамках которого путем реального выполнения каких-либо задач проверяется соответствующая работа либо системы в целом, либо составной ее части.

Тестирование программы (program testing) - проверка, которая проводится в ходе прогона программы с целью убедиться, работает ли она так, как требуется. Это осуществляется при выполнении одного или нескольких тестовых прогонов, при которых в программную систему подаются входные (тестовые данные), а реакция системы фиксируется для последующего анализа. Может осуществляться как с ЭВМ, так и без ЭВМ.

Один из главных законов тестирования гласит: «Тестирование программы или ее отдельных модулей не должен осуществлять программист (группа программистов), создавший эту программу или модуль».

Для обеспечения достаточной степени надежности тестирования должен быть специальный отдел в фирме или привлечены программисты из сторонних организаций. Первоначально разработчик сам устраняет мелкие ошибки. Когда компиляция модуля завершается, и компилятор выдает соответствующее сообщение *«Compile successful",* программист обычно запускает откомпилированный фрагмент и производит несколько тестов.

После такого поверхностного тестирования разработчиком законченный модуль должен быть протестирован другим программистом. Дело в том, что разработчик изначально настраивается на какой-то один вид возможной ошибки и, не обнаружив ее при предварительном тестировании, не склонен проверять другие участки модуля. Сторонний программист рассматривает модуль, подключенный к программе как своего рода «черный ящик» и пытается запускать его на предельных нагрузках.

Статистика свидетельствует, что стоимость тестирования составляет не менее 50% всей стоимости начальной разработки.

Сколько бы сил, времени и денег не было потрачено на тестирование, один из главных законов программирования действует с неотвратимостью рока: «Тесты могут доказать наличие ошибок в программе, но они не могут доказать их отсутствия» (Э. Дейкстра «Заметки по структурному программированию»).

При тестировании могут возникать следующие вопросы:

1) Искать все ошибки или грубейшие?
2) Если не все, то как установить порог допустимости ошибки?
3) Когда завершать тестирование?
4) Что делать, если сроки поджимают или нет ресурсов на дальнейшее тестирование?
5) Где остановиться в документировании тестов?

Ответы на них во многом зависят от того, что считать качественной программой?

Качественная программа - это программа, выполняющая заранее объявленные действия известным способом и не выполняющая никаких необъявленных действий.

Под объявленными действиями понимаются, в том числе, и алгоритмы обработки данных. Таким образом, тестирование должно выполняться до тех пор, пока программа не избавится от всех необъявленных действий (частным случаем которых является неверная обработка, то есть, в сущности, просто порча данных).

5.3. ХОД ТЕСТИРОВАНИЯ

В процессе тестирования программного обеспечения осуществляются следующие виды деятельности:

Ручной прогон. На этом шаге программист с помощью карандаша и листа

бумаги моделирует прохождение данных через программу. При изучении текста программы от начала до конца, трудно проверить всевозможные комбинации данных. Самое большее, что можно сделать на практике, - проверить всевозможные типы или наиболее вероятные наборы комбинаций данных. Если они дают правильные результаты, предполагается, что непроверенные комбинации также дадут правильные результаты.

Проектирование тестов. Является наиболее ответственным процессом. Очень часто тест создается вручную. Иногда применяют генераторы тестовых данных - специальные программы, формирующие данные в соответствии со спецификациями, задаваемыми программистом. Тестовые данные могут систематически или случайно выбираться из другого заданного набора данных для уменьшения их общего количества.

Выполнение тестов. На этом этапе осуществляется проверка всех возможных алгоритмов специально подготовленными тестами, а также выявляется, насколько интерфейс программы выдержит реальную нагрузку. Проблема заключается в том, что тестирование происходит на очень ограниченных объемах данных. Когда база данных будет насчитывать десятки, а то и сотни тысяч записей, скорость выполнения запросов пользователей может стать неприемлемой.

Изучение результатов тестирования. Выявление и устранение ошибок часто имеет циклический характер. Устранение одной ошибки может порождать другие ошибки. Особенно это касается работы с глобальными переменными, которые коварны тем, что нельзя сказать с полной уверенностью, что где-то на нижнем уровне подпрограмм изменение состояния переменной не приведет к новой ошибке.

5.4. АВТОНОМНОЕ ТЕСТИРОВАНИЕ МОДУЛЕЙ ПРОГРАММЫ

Программа состоит из модулей, что сильно облегчает тестирование, так как каждый модуль может быть отдельно проверен для всех возможных комбинаций, поскольку их число меньше. Если программа хорошо структурирована, необходимо проверять только интерфейс между проверенными ранее модулями. Даже если проверяются все варианты интерфейса, это является посильной ношей как для программиста, так и для вычислительной системы.

К сожалению, объектно-ориентированное программирование ускоряет создание сложных программ, но зато увеличивает число ошибок, которые к тому же очень тяжело искать.

Автономное тестирование модуля должно как минимум обеспечивать прохождение всех путей вычислений. Проектная процедура тестирования модуля состоит из следующих действий:

1) по внешним спецификациям модуля готовятся тесты для каждой ситуации и каждого недопустимого условия;
2) просматривается текст модуля с целью убедиться, что все условные переходы будут выполняться в каждом направлении. При необходимости добавляются соответствующие тесты;
3) изучается текст модуля с целью убедиться, что тесты охватывают достаточно много путей. Для циклов должны быть тесты без повторения, с одним повторением и с несколькими повторениями;
4) проверяется текст модуля, чтобы определить его чувствительность к особым значениям данных. Наиболее опасные числа это ноль и единица. В случае необходимости нужно добавить тесты.

5.5. МЕТОДЫ ТЕСТИРОВАНИЯ

Существует два крайних подхода к проектированию тестов. Сторонники первого подхода создают тесты, исследуя внешние спецификации сопряжения программы или модуля, который необходимо протестировать. В этом случае программа является как бы «черным ящиком». Необходимо

проверить все возможные комбинации и значения на входе. Обычно их слишком много даже для простейших алгоритмов. Для программы расчета среднего арифметического четырех чисел надо готовить 10^7 тестовых данных.

Сторонники второго подхода связывают тесты только с логикой программы. При этом стремятся, чтобы каждая команда была бы выполнена хотя бы один раз. Цикл должен выполняться один раз, ни разу, максимальное число раз. Такое тестирование всех путей извне также недостижимо. В программе из двух последовательных циклов, внутри каждого из которых включено ветвление на десять путей имеется 10^{18} путей расчета.

Чтобы построить разумную стратегию тестирования надо разумно сочетать оба этих подхода и пользоваться математическими доказательствами.

Восходящее тестирование. Сначала автономно тестируются модули нижних уровней, которые не вызывают других модулей. При этом достигается такая же их высокая надежность, как и у встроенных в компилятор функций. Затем тестируются модули более высоких уровней вместе с уже проверенными модулями и так далее по схеме иерархии.

Нисходящее тестирование. При этом подходе изолированно тестируется головной модуль или группа модулей головного ядра. Программа собирается и тестируется сверху вниз. Недостающие модули заменяются заглушками. Достоинством этого подхода является то, что тестирование модуля совмещается с тестированием сопряжений.

Модифицированный нисходящий метод. Согласно этому методу каждый модуль автономно тестируется перед включением в программу, собираемую сверху вниз.

Метод большого скачка. Каждый модуль тестируется автономно. По окончании автономного тестирования все модули интегрируются в готовую программную систему. Этот метод применяется, если программа мала и хорошо спроектирована по сопряжениям.

Метод сэндвича. Представляет собой компромисс между нисходящим и восходящим подходами. По этому методу реализация и тестирование ведется одновременно сверху и снизу, и два этих процесса встречаются в заранее намеченной точке.

Модифицированный метод сэндвича. Нижние модули тестируются снизу вверх, а модули верхних модулей сначала тестируются автономно, а затем собираются нисходящим методом.

5.6. АКСИОМЫ ТЕСТИРОВАНИЯ

Тестирование программных систем в настоящее время остается в большей мере искусством, чем наукой. При проведении тестов рекомендуется придерживаться так называемых аксиом тестирования, представляющих собой эвристические приемы. Вот некоторые из них:

1)хорошим считается тот тест, в котором высока вероятность обнаружения ошибок;

2)тестирование собственной программы существенно осложняется психологическими причинами;

3)необходимой частью тестов считается описание выходных результатов;

4)нужно готовить тесты как для правильных, так, и для неправильных данных; ,

5)нельзя тестировать «с лету», наскоком;

6)необходимо тщательно изучать результаты каждого теста;

7)по мере обнаружения все большего числа ошибок в программе, возрастает вероятность обнаружения в ней еще большего числа ошибок;

8)тестирование программ должны осуществлять самые лучшие и опытные программисты;

9)одной из главных задач разработчиков программы является возможность осуществления ее тестирования;

10)нельзя изменять программу только с целью облегчить процесс ее тестирования;

11)чем раньше спроектирован тест, тем выше вероятность выявления ошибок. Лучше всего готовить тесты еще на этапе проектирования системы;
12)процесс тестирования должен быть документирован и полностью управляем, хаотичность недопустима;
13)необходимы повторяемость и завершенность тестов;
14)следует избегать добавления новых тестов уже в процессе тестирования.

5.7. КЛАССИФИКАЦИЯ ТЕСТОВ

По условиям их поведения тесты могут быть классифицированы следующим образом:

Доказательство (proof) - попытки найти ошибки в программе путем доказательств на основе математических теорем о правильности программы, безотносительно к внешней программной среде.

Верификация программы (prograт verification) - попытка найти ошибки, выполняя программу в тестовой или моделируемой среде.

Испытание (validation) - попытка найти ошибки, выполняя программу в заданной программной среде.

Приемо-сдаточные испытания - проверка пригодности программы для эксплуатации; такие испытания обычно проводят под контролем поставщика системы.

По назначению тесты классифицируются:

тестирование модуля (автономное тестирование) *(module testing)* — контроль, отдельного модуля в изолированной среде (например, с помощью ведущей программы), инспекция текста модуля на сессии программистов, которая иногда дополняется математическим доказательством правильности модуля;

тестирование сопряжений (integration testing) - контроль сопряжений между частями программной системы, как между компонентами в комплексе, так и между модулями отдельного компонента (например, у заглушки);

комплексное тестирование (system testing) — контроль и испытание системы по отношению к исходным целям. Осуществляется с целью проверки правильной совместной работы составных частей программы. При комплексном тестировании особое внимание обычно уделяется взаимодействию компонентов. Комплексное тестирование является процессом контроля, если оно выполняется в условиях моделируемой среды, и процессом испытания при выполнении в реальной среде;

системное тестирование - при системном тестировании вся система в целом обычно рассматривается как некоторый «черный ящик»; поведение этой системы исследуют, не вникая в подробности отдельных ее компонентов и взаимодействий между ними;

тестирование приемлемости (acceptance testing) - проверка соответствия программы требованиям пользователя.

5.8. ОТЛАДКА

Процесс тестирования нельзя путать с процессом отладки *(debugging)*. Первый служит лишь для обнаружения факта существования ошибок, а не для их локализации и устранения.

Отладка программ обычно осуществляется с использованием специальных программных средств. Последние используются для исследования внутреннего поведения программы. Типичный отладчик позволяет вводить в программу точки останова для оценки промежуточных результатов и производить проверку и модификацию значений переменных в этих точках.

Существуют несколько способов отладки программы.

Распечатка текущего состояния. Используется с целью фиксации фактических значений переменных для проверки хода вычислений. Для этого

во время отладки программы в места, которые программист считает критическими, помещают процедуры распечатки текущего состояния переменных. После окончания теста вызовы этих процедур удаляются, и программа снова перекомпилируется.

Точки останова. Используются в случае разного рода зацикливаний, когда программа в какой-то момент «зависает». В текст программы включают процедуры останова программы. Например, можно поместить процедуру вывода обычного сообщения вроде «Достигнута точка *#nnn»* и инициировать паузу до нажатия на любую клавишу. При таком подходе программист точно знает, до какой точки дошла программа перед зацикливанием.

Метод деления пополам. Этот метод используют связисты, когда ищут обрыв в кабеле, закопанном в землю. Например, если приблизительно известно до какого момента программа успешно выполняется, то в этом месте программы ставят точку останова. Затем ставят точку в конце «подозреваемой» процедуры и посредине - между первой и последней точками. Снова компилируют и выполняют программу. Если программа дошла до второй точки, то зацикливание произошло где-то между второй и третьей точками, если не дошла - между первой и второй. После этого вставляется новая точка останова в локализованный участок и программа снова компилируется. Таким образом, постоянно сжимая район поисков, можно найти ошибочный участок.

Трассировка. Является последним средством обнаружения ошибки. Она может оказаться очень эффективной, но значительно замедляет выполнение программы и, не будучи тщательно спланированной, приводит к колоссальным объемам выдаваемой информации. При трассировке происходит пошаговое выполнение программы с возможностью просмотра состояния всех переменных.

5.9. ОПТИМИЗАЦИЯ

Обычно программа создается в достаточно жестком временном режиме, что заставляет программиста искать, скорее, более правильные решения, чем более эффективные. Под эффективностью программы понимают, прежде всего, скорость ее выполнения, а также ее объем. Сегодня, когда уделяется особое внимание пользовательскому интерфейсу, в список влияющих на эффективность программы факторов можно, пожалуй, также занести и удобство интерфейса. Таким образом, под *оптимизацией* понимается процесс улучшения программы.

Оптимизация не является обязательным условием разработки программы. Однако существует целый класс программ, критичных как к скорости выполнения, так и к размеру. Таковыми являются программы графического вывода в силу большого объема вычислений, связанных с графическими преобразованиями.

При проектировании больших систем оптимизация производится в два этапа, Сначала оптимизируют текст программы на языке высокого уровня, а затем наиболее критичные ко времени выполнения процедуры переписывают на язык ассемблера. Существуют следующие способы оптимизации программных кодов.

Разгрузка участков повторяемости. Является способом оптимизации, который чаще всего подразумевает разгрузку циклов путем вынесения из них выражений, которые могут быть вычислены вне циклов.

К этому виду преобразований относятся также «чистки» тел рекурсивных процедур, когда выражения в соответствующем цикле (или теле многократно вызываемой процедуры) выносятся и размещаются перед входом в участок повторяемости — это так называемая чистка вверх.

Иногда применяют чистку вниз, когда соответствующие фрагменты кода помещаются после цикла. При этом нужно обратить внимание на то, что выносить можно только такие выражения, которые обязательно исполняются при каждом прохождении разгружаемого цикла.

Замена сложных операций на более простые. Очень часто одна

операция предпочтительнее другой на том основании, что выполняется быстрее, но знание таких нюансов приходит к программисту лишь с опытом.

Например, операция сложения выполняется быстрее операции умножения, а умножение быстрее операции деления. Поэтому один оператор умножения переменной на некоторое небольшое целое число (обычно не более трех) лучше заменить на эквивалентное количество сложений. Выражение:

Total := Summa + Summa + Summa;

эффективнее выражения: Total := 3 * Summa;

а операцию деления Summa := Summa/2; лучше заменить на более быстрое умножение, которое приведет к тому же самому результату Summa := Summa * 0.5;

Чистка программы. Данный способ повышает качество программы за счет удаления из нее ненужных объектов и конструкций. Набор преобразований этого типа включает в себя следующие варианты оптимизации:

удаление несущественных операторов, то есть операторов, не влияющих на результат программы;

удаление бесполезных операторов, вычисляющих вспомогательные переменные, используемые только для подстановки в другие выражения;

удаление объявленных, но неиспользуемых переменных и типов;

удаление идентичных операторов;

удаление процедур, к которым нет обращений.

Необходимость в такого рода чистках возникает потому, что очень часто программист «захлебывается» в общем количестве переменных и процедур одного слишком большого модуля. Подчас он объявляет переменные, которые потом нигде в программе не использует.

Например, программист может объявить целочисленную переменную для организации цикла, а затем, спустя какое-то время, для организации другого цикла объявляет еще одну переменную, забыв о существовании предыдущей и возможности ее повторного использования.

Экономия памяти. Одним из главных ресурсов после процессорного времени, который использует программа, является объем оперативной памяти. Объем памяти зависит как от размера кода самой программы, так и от количества статических и динамических переменных.

Программист должен учиться как можно экономнее использовать память. Каждую структуру следует тщательно продумывать и не требовать, скажем, для переменной, в которой будут храниться координаты текстового экрана, двухбайтового типа.

Это так называемая экономия на типе переменной. Существует и еще ряд способов более экономного расходования памяти:

1) Глобальная экономия памяти подразумевает совмещение по памяти не существующих одновременно статических переменных. Модульное программирование также подразумевает разнесение объявлений несвязанных переменных в различные модули.

2) Изменение области существования переменной.

3) Перемещение оператора объявления переменной (резервирования памяти) ближе к тому участку программы, в котором содержатся операторы, использующие эту переменную, то есть переменную следует объявлять в границах того блока, где она используется.

4) Экономия стека: при передаче массива в качестве параметра подпрограммы, следует использовать ссылку на массив. Помимо экономии памяти это приводит также и к экономии времени за счет того, что система не создает копии передаваемого массива в стеке.

1) АЛГОРИТМЫ И ИХ РАЗРАБОТКА

6.1. ПОНЯТИЕ АЛГОРИТМА И ЕГО СВОЙСТВА

Прежде чем компьютер сможет выполнить задачу, ему необходимо предоставить алгоритм ее решения, в точности описывающий, что и как надо делать. Поэтому изучение алгоритмов лежит в основе программирования.

Алгоритм - это точное, сформулированное на определенном языке, конечное описание того или иного способа действия, основанного на применении исполнимых элементарных однозначно трактуемых шагов.
Любой алгоритм имеет пять особенностей.

1) *Конечность алгоритма* (финитность). Означает, что алгоритм всегда должен заканчиваться после конечного числа шагов.

Это требование происходит из теории вычислений, которая пытается провести грань между правильными и неправильными алгоритмами. Алгоритм считают правильным, если на любом допустимом входе он заканчивает работу и выдает результат, удовлетворяющий требованиям задачи. Неправильный алгоритм для некоторого входа может вовсе не остановиться или дать неправильный результат.

Однако существуют примеры, использующие бесконечные процессы, например контроль показателей жизнедеятельности пациента в больнице или поддержание установленной высоты полета авиалайнера.

Поэтому на практике термин алгоритм часто неформально используется по отношению к последовательностям этапов, не обязательно определяющим конечные процессы. Примером может служить известный нам еще со школьной скамьи алгоритм деления в столбик, который не определяет конечный процесс в случае деления 1 на 3.

2) *Определенность алгоритма.* Каждый шаг алгоритма должен быть точно определен, то есть действия, которые необходимо произвести должны быть недвусмысленно определены в каждом возможном случае.

Чтобы исключить неоднозначность, разработаны определенные подходы для записи алгоритмов. Один из них - запись алгоритма в виде блок схемы, представляющей собой последовательность специальных пиктограмм, каждая из которых однозначно указывает на выполняемое действие.

Другой подход заключается в разработке формально определенных языков, в которых каждое утверждение имеет абсолютно точный смысл. При формулировке алгоритма для его выполнения на компьютере применяются языки программирования.

3) *Наличие входных данных.* Алгоритм имеет некоторое число входных величин, заданных ему до начала работы.

Иногда это число может равняться нулю, однако это означает, что входные величины должны быть описаны как часть алгоритма. Например, алгоритм сложения числа 777 с числом 333. В этом случае шаги 1 и 2 должны быть записаны так:

Шаг 1. Записать число 777.

Шаг 2. Под ним записать число 333

………

То есть, если даже входные величины отсутствуют, то какие-то величины будут описаны в самом алгоритме. В противном случае алгоритм не может быть выполнен.

4) *Наличие выходных данных.* Результатом выполнения любого алгоритма всегда будет обработанная информация, выдаваемая в том или ином виде и, следовательно, алгоритм обязательно имеет одну или несколько выходных величин, являющихся результатом обработки входных данных.

5) *Эффективность алгоритма.* Алгоритм, который выполняет действие за меньшее число шагов признается более эффективным.

Это связано с тем. что любая программа, в основе которой лежит тот или иной алгоритм, должна выполняться на реальном вычислительном устройстве. Поэтому большое значение имеет время выполнения программы, на которое влияет количество шагов алгоритма. Кроме того, программа занимает определенный объем в памяти компьютера. Поэтому количество шагов косвенно влияет на еще один показатель эффективности - размер программы.

6.2. ПРЕДСТАВЛЕНИЕ АЛГОРИТМА И ПСЕВДОКОД

Алгоритм является абстракцией и поэтому один и тот же алгоритм можно представить многими способами. Если с алгоритмом работает человек, то это может быть традиционный язык (русский, английский), язык картинок и пиктограмм, а также математические формулы.

В программировании эти проблемы решают путем создания четко определенного набора составных блоков, называемых примитивами, из которых могут конструироваться представления алгоритмов. Набор примитивов вместе с набором правил, устанавливающих, как эти примитивы могут комбинироваться для представления более сложных идей, образуют язык программирования.

Каждый примитив состоит из двух частей: *синтаксической* и *семантической.* Синтаксис относится к символьному представлению примитива, а семантика - к смысловому значению примитива.

Чтобы получить набор примитивов, пригодных для представления алгоритмов, выполняемых на вычислительной машине, можно использовать машинные команды. Однако описание алгоритма на таком уровне детализации весьма утомительно, поэтому обычно используется набор примитивов более высокого уровня, являющийся высокоуровневым языком программирования.

Псевдокод - это система обозначений, предназначенная для неформального представления идей в процессе разработки алгоритмов.

Один из путей создания псевдокода - ослабление правил того формального языка программирования, на котором требуется записать окончательную версию алгоритма. В подобной ситуации псевдокод может состоять из синтаксических и семантических структур, аналогичных структурам целевого языка программирования, но не столь формализованных.

Альтернатива выражается на псевдокоде следующей структурой: if *(условие)* then *(действие)* else *(действие)*

Сокращенный синтаксис этого конструкта когда не предусмотрено действие для варианта else выглядит так:

if *(условие)* then *(действие)*

Цикл-пока является алгоритмической структурой, которая заключается в необходимости продолжать выполнение последовательности действий до тех пор, пока некоторое условие остается верным.

Эта инструкция предписывает проверить условие и, если оно верно, выполнить действие, а затем вновь проверить условие. Если при очередной проверке условие оказывается неверным, следует перейти к инструкции, следующей за данной структурой.

while (условие) do *(действие)*

Цикл-do на псевдокоде имеет следующий вид: repeat *(действие)* until (условие)

Оператор присваивания. Часто желательно ссылаться на некоторые значения с помощью описательных имен. Для установки подобных связей будет использоваться следующая конструкция присваивания:

assign *имя* the value *выражение*

здесь параметр имя - это описательное имя, а параметр выражение описывает значение, связываемое с этим именем. Например: assign *Итог* the value *Цена* + *Налог*

При ее выполнении результат суммирования значений переменных *Цена* и *Налог* будет связан с именем *Итог* .

Процедуры. Используются для описания действий, которые могут выступать в роли вспомогательных программ в других приложениях. Такие программные элементы имеют несколько различных названий, а именно: подпрограммы, процедуры и функции.

В псевдокоде для обозначения заголовка, по которому можно распознать данный блок псевдокода используется термин procedure: procedure *имя*

здесь *имя* — это конкретное название, присвоенное данному блоку. Ниже следуют инструкции, определяющие выполняемые в этом блоке действия.

Процедуры должны разрабатываться так, чтобы быть как можно более общими. Например, процедура сортировки списков имен должна быть способна сортировать любой список, а не какой-то один определенный. Поэтому она должна быть написана таким способом, чтобы подлежащий сортировке список не определялся в самой процедуре, а передавался ей в качестве входных данных, представленных некоторым обобщенным именем.

Такие обобщенные имена в псевдокоде выделяют угловыми скобками и указывают их в круглых скобках а той же строке, в которой определяется имя данной процедуры. В частности, процедура *Сортировка,* предназначенная для сортировки произвольных списков имен, будет начинаться следующей инструкцией: procedure *Сортировка*(<список>)

Таким образом, назначение псевдокода состоит в предоставлении средств, позволяющих записывать схемы алгоритмов лишь в общих чертах, а не в написании законченных формальных программ. Поэтому в псевдокоде нет запрета на использование неформальных фраз, запрашивающих такие действия, детали которых не определены достаточно строго.

Поиск более удачных средств представления алгоритмов продолжается и поныне. Существующие тенденции заключаются в использовании графических методов, однако псевдокод остается достаточно эффективным при разработке процедурных компонентов небольшого размера, входящих в состав программных систем.

6.3. АЛГОРИТМ ПОСЛЕДОВАТЕЛЬНОГО ПОИСКА

Этот алгоритм решает задачу поиска в списке некоторого заданного значения. Он позволяет установить, есть ли заданное значение в списке. Если это значение в списке присутствует, поиск будет считаться успешным, в противном случае - неудачным.

При этом считается, что список отсортирован согласно некоторому правилу, позволяющему упорядочить его элементы. Например, если это список имен, то имена в нем расположены в алфавитном порядке. Если же это список числовых значений, то его элементы расположены в порядке возрастания. В результате получается процедура, текст которой приведен ниже.

procedure *Поиск* (<список>, <искомое значение>) if *(список пуст)*
then *(объявить поиск неудачным)*
else
(выбрать как <проверяемое значение> *первый элемент списка)*
while (<искомое значение> > <проверяемое значение> *и есть непроверенные элементы)* do *(выбрать следующий элемент списка* как <проверяемое значение>) if (<искомое значение> = <проверяемое значение>)
then (*объявить поиск успешным)* else *(объявить поиск неудачным)*

По окончании выполнения структуры while искомое значение либо будет найдено, либо выяснится, что его нет в списке. В любом случае успешность поиска можно установить, сравнивая искомое значение с проверяемым. Если они эквивалентны, поиск объявляется успешным. Для полной уверенности в правильности программы она помещается в предложение else инструкции if.

Такая процедура позволяет установить, является ли кто-то пассажиром некоторого рейса, или установить, входит ли сахар в перечень ингредиентов некоторого блюда.

Таким образом, алгоритм последовательно рассматривает все элементы

списка. В силу своей простоты он часто применяется к коротким спискам или когда это необходимо. Однако в случае длинных списков этот метод оказывается менее эффективным, чем другие.

6.4. АЛГОРИТМ ДВОИЧНОГО ПОИСКА

Также решает задачу поиска заданного элемента в отсортированном списке. Здесь используется процедура, которую человек использует при поиске имени в телефонном справочнике. Справочник открывается примерно в том месте, где может находиться нужное имя. Если повезет, оно окажется именно там, в противном случае поиск придется продолжить. Однако в этой точке область поиска сужается либо до начальной части справочника, предшествующей текущей позиции, либо до остальной части справочника, следующей за ней.

Реализовать эту стратегию можно с помощью следующего алгоритма, в котором учитывается возможность получения пустого списка.

procedure Search (<список>, <искомое значение>) if (<список> пуст)
then *(Объявить поиск неудачным)*
else
(Выбрать "средний" элемент в <список> в качестве
<проверяемый элемент>)
(Выполнить один из следующих трех блоков инструкций, в зависимости от того, является ли <искомое значение> равным, меньшим или большим, чем <проверяемый элемент>)
Case 1: <искомое значение> = <проверяемый элемент>
(*Объявить поиск успешным)*
Case 2: <искомое значение> < <проверяемый элемент>
(Применить процедуру Search, чтобы определить, есть ли в части списка, предшествующей элементу <проверяемый элемент>, элемент <искомое значение>, и if (*тот поиск успешен*
then *(Объявить этот поиск успешным)* **else** *(Объявить этот поиск неудачным)* **Case 3:** <искомое значение> >
<проверяемый элемент> (Применить процедуру Search, чтобы определить, есть ли в части списка, следующей за элементом <проверяемый элемент>, элемент
<искомое значение> и if (тот поиск успешен)
then (Объявить этот поиск успешным) **else** (Объявить этот поиск неудачным)

Если выбранный элемент не является искомым, то эта программа предлагает два варианта дальнейших действий - поиск в начальной или конечной половине списка. В каждом из них предусматривается выполнение вторичного поиска той же процедурой Search.

При выполнении этой процедуры и при достижении инструкции <Применить процедуру Search, чтобы ...>, будет применяться этот же метод поиска к меньшему списку, который является частью исходного списка. Если этот вторичный поиск завершится успешно, то осуществляется возврат в исходную процедуру, чтобы объявить выполняемый в ней поиск успешным. Если же вторичный поиск окончится неудачей, то объявляется неудачным и исходный поиск.

В этом процессе необходимо многократно разделять рассматриваемый список на две примерно равные части, после чего область дальнейшего поиска ограничивается лишь одной из этих частей. За это повторяющееся деление на два данный алгоритм был назван двоичным поиском.

Алгоритм двоичного поиска похож на алгоритм последовательного поиска, так как в обоих выполняется повторяющийся процесс. Однако реализация этого повторения в каждом случае существенно отличается. При последовательном поиске повторение организуется с помощью цикла, в случае двоичного поиска

каждая стадия повторения реализуется как подзадача предыдущей стадии. Этот метод повторения известен *как рекурсия.*

6.5. АЛГОРИТМ СОРТИРОВКИ МЕТОДОМ ВСТАВКИ

В этом алгоритме решается задача сортировки списка имен в алфавитном порядке. При этом необходимо отсортировать список путем перестановки его элементов, не перемещая весь список в другое место. Подобное ограничение типично для компьютерных приложений, где стремятся наиболее эффективно использовать весь доступный объем памяти.

Для их сортировки должны повторяться следующие действия. Поднять карточку с именем, первую в неотсортированной части списка, сдвинуть вниз карточки с именами, которые находятся ниже по алфавиту, чем имя на взятой карточке, а затем поместить эту карточку на освободившееся место в списке. Текст программы сортировки на языке псевдокода выглядит так:

```
procedure Сортировка (<список>) assign N the value 2;
while (значение N не превышает <длина списка>)
  do (Выбрать N-й элемент списка в качестве опорного; Переместить
    этот элемент во временное хранилище, оставив в списке пустое
    место;
    while (над пустым местом есть имя, которое по алфавиту
 размещается ниже, чем опорный элемент)
    do (переместить имя, находящееся над пустым местом вниз, оставив
  в прежней позиции пустое место); Поместить опорный элемент на
  пустое место в списке assign N the value N+1
```

здесь N - счетчик, параметр <длина списка> - количество элементов в списке.

Программа сортирует список, многократно повторяя следующие действия: «Элемент извлекается из списка, а затем вставляется на надлежащее ему место».

Каждое выполнение тела внешнего цикла приводит к тому, что внутренний цикл инициализируется и выполняется до тех пор, пока не будет выполнено условие его окончания. Условие окончания внешнего цикла выполняется, когда значение счетчика N превысит длину сортируемого списка.

Управление внутренним циклом инициализируется, когда опорный элемент извлекается из списка и в нем образуется пустое место. Операция модификации включает перемещение расположенных выше элементов на пустое место вниз, в результате чего свободное место перемещается вверх по списку. Условие окончания выполняется, когда пустое место или находится непосредственно под именем, которое по алфавиту размещается выше опорного значения или же достигает верхней позиции списка.

6.6. ЭФФЕКТИВНОСТЬ АЛГОРИТМОВ

Современные вычислительные машины способны выполнять миллионы операций в секунду, однако эффективность по-прежнему остается важнейшим аспектом разработки алгоритмов. Часто выбор между эффективным и неэффективным решением задачи может на самом деле означать выбор между реализуемым и нереализуемым способом ее решения.

Рассмотрим задачу, с которой сталкивается секретарь университета при поиске личных дел студентов и их заполнении. В университете на протяжении любого семестра числится около 30 тысяч студентов. Чтобы найти личное дело некоторого студента, секретарь должен выполнить поиск по его идентификационному номеру в общем списке. Почувствует ли секретарь разницу при поиске между алгоритмом последовательного и двоичного поиска?

Алгоритм последовательного поиска начинает работу с начала списка и последовательно сравнивает каждый выбираемый элемент с искомым числом. Средняя глубина поиска будет равна половине длины списка из 15-ти тысяч элементов. Если выборка каждого дела из памяти выполняется за 10 мс, то среднее время поиска будет составлять 150 секунд. Этот вариант совершенно

неприемлем.

Алгоритм двоичного поиска сравнивает искомое значение со средним элементом списка. Если это не искомый элемент, то область поиска сразу же сужается до половины исходного списка. После второго этапа область поиска в большинстве случаев сократится до 7 500 дел и т. д. Искомое значение будет найдено при выборе, максимум 15 элементов списка. Процесс поиска нужного личного дела потребует не более 0,15 секунды, то есть мгновенно.

В общем случае такой анализ осуществляется в более широком контексте. Хотя выше рассматривался список определенной длины, эти рассуждения можно обобщить на случай списка произвольной длины. При применении к списку из *n* элементов алгоритма последовательного поиска в среднем потребуется проверить $n/2$ элементов, тогда как алгоритму двоичного поиска, в самом худшем случае, потребуется проверить только $log_2 n$ элементов (логарифм показывает, сколько раз число *n* можно разделить на два).

Аналогичным образом можно проанализировать алгоритм сортировки методом вставки. Поскольку основным действием в реализации этого алгоритма является сравнение двух имен, то необходимо подсчитать количества таких сравнений, которые потребуется выполнить при сортировке списка длиной *n* элементов.

В наилучшем случае применение алгоритма сортировки методом вставки к списку из *n* элементов потребует выполнения n-1 сравнений. (Второй элемент сравнивается с одним элементом (первым), третий элемент — с одним элементом (вторым) и т. д.)

Наихудший сценарий имеет место в том случае, когда каждый опорный элемент потребуется сравнивать со всеми впереди стоящими элементами. Первый опорный элемент (второй элемент списка) сравнивается с одним элементом, второй опорный элемент (третий элемент списка) - с двумя элементами и т. д. Следовательно, общее количество сравнений при сортировке списка из *n* элементов составит $1 + 2 + 3 + ... + (n - 1)$, что эквивалентно $n(n-1)/2$ или $(1/2)(n^2 - n)$.

В среднем при сортировке методом вставки можно ожидать, что каждый опорный элемент потребуется сравнить с половиной предшествующих ему элементов. В этом случае общее количество выполненных сравнений будет вдвое меньше, т. е. $(1/4)(n2 — n)$.

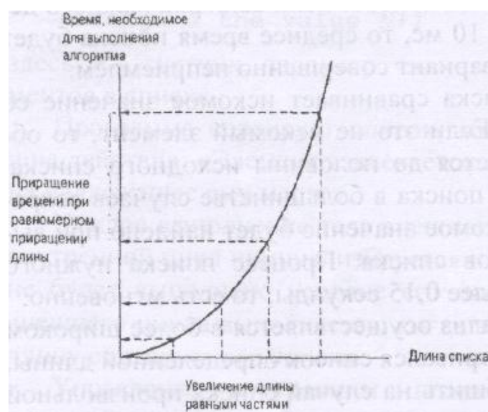

Рис. 6. Зависимость времени выполнения алгоритма сортировки от длины списка

Количество сравнений, выполненных алгоритмом сортировки методом вставки, позволяет оценить время, которое потребуется для выполнения сортировки. Эта оценка показана на графике (рис. 6), который построен по оценкам работы алгоритма в наихудшем случае, когда для списка длиной *n* требуется выполнить не менее $(1/2)(n^2-n)$ сравнений элементов. При увеличении длины списка на одно и то же количество элементов время, необходимое для сортировки списка, все больше и больше возрастает. Таким образом, с увеличением длины списка эффективность данного алгоритма уменьшается.

При использовании алгоритма двоичного поиска в наихудшем случае для

поиска в списке из *n* элементов потребуется проанализировать не более $log_2\ n$ элементов. На рис. 7 представлен график, построенный по результатам анализа времени, необходимого для выполнения алгоритма при различной длине сортируемого списка. Темпы роста времени выполнения алгоритма снижаются по мере увеличения длины списка, т. *е.* эффективность алгоритма двоичного поиска возрастает с увеличением длины списка.

Основным отличием между этими графиками является их общая форма, которая определяется типом отображаемого выражения. Все линейные выражения изображаются прямой линией, все квадратичные выражения - параболической кривой, а все логарифмические выражения порождают логарифмическую кривую.

Форма графика, представляющего зависимость времени выполнения алгоритма от объема входных данных, отражает общие характеристики эффективности алгоритма. Поэтому алгоритмы принято классифицировать согласно форме их графиков, построенных для самого неблагоприятного случая.

Способ обозначения, используемый для определения этих классов, иногда называют тета-классами. Алгоритмы, графики которых имеют параболическую форму (например, сортировка методом вставки), относятся к классу $\Theta\ (n^2)$, алгоритмы, графики которых имеют логарифмическую форму (например, двоичный поиск), - к классу $\Theta\ (\lg n)$. Таким образом, любой алгоритм из тета-класса $\Theta\ (lg\ n)$ по самой свой сути всегда более эффективен, чем алгоритм из тета-класса $\Theta\ (n^2)$.