Д.А. Чернев

ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Д. А. ЧЕРНЕВ

ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Допущено Министерством высшего и среднего специального образования Республики Узбекистан в качестве учебного пособия для высших учебных заведений

ТАШКЕНТ — «МЕНҲАТ» — 2004

32.973.2я
Ч49

В пособии описываются вопросы технологии разработки программного обеспечения: этапы его разработки (жизненный цикл), проблемы проектирования сложных программных средств для различных типов ЭВМ, средства автоматизации разработки программного обеспечения, методология программирования, включая визуальное и объектно-ориентированное программирование, а также правила документирования проекта и программного обеспечения.

Пособие предназначено для студентов технических вузов, обучающихся по направлениям, «Программирование». Может быть полезно студентам родственных специальностей, а также разработчикам программного обеспечения.

Научный редактор: доктор технических наук, профессор
Р.Х. ХАМДАМОВ

Рецензенты: кандидат технических наук, доцент **С.С. ГУЛЯМОВ**,
кандидат технических наук, доцент **Е.М. ХАЧАТУРОВА**

Ч $\frac{2404020000-40}{\text{М } 359(04)-2004}$ без объяв. 2004

ISBN 5—8244—1596—X

Введение

Ошибки замысла и проектирования программ, некачественная документация приводят к весьма ощутимым временным и экономическим потерям. По этому поводу красноречива американская статистика: «если размеру большой программы поставить в соответствие время, необходимое для ее разработки, и число занятых в ней программистов, то окажется, что все происходит так, словно каждый программист создает 5 команд в день, а остальное время пишет команды, которые потом признаются бесполезными или неправильными, и разыскивает сделанные ошибки».

Существуют, однако, методы, позволяющие нарушить печальный цикл: «ошибка — исправление — новая ошибка». В настоящее время накоплен значительный опыт применения ряда прогрессивных отечественных и зарубежных технологий разработки программного обеспечения, благодаря которым производительность труда разработчиков программ можно значительно повысить. Производительность здесь не означает «эффективность» в традиционном смысле (т.е. искусство и приемы, позволяющие сэкономить байт или полсекунды). Речь идет о понятии более общем — о производительности, при которой создаются программы, отвечающие следующим требованиям:

— «правильные», т.е. они решают именно ту задачу, для которой были задуманы;

— надежные;

— легко читаемые и предполагающие их передачу другому программисту;

— в полной мере использующие возможности машин, которые их исполняют.

«Производительность разработки программного обеспечения», таким образом, означает здесь оптимальное использование и машин, и людей.

Применяя прогрессивные современные технологии и

методологию разработки программного обеспечения, можно добиться следующих улучшений в типичном цикле его разработки:

— заметно сократить сроки создания комплексов программ и повысить экономический эффект их разработки;

— быстро обнаружить и устранить в процессе разработки разногласия и отсутствие связи между членами команды разработчиков и между программистами и конечными пользователями;

— упростить и облегчить для новых специалистов подключение к проекту на любой стадии его разработки;

— свести к минимуму затраты на дальнейшую доработку, модификацию и сопровождение готового продукта благодаря фундаментальному анализу задачи, на котором базируется конечное приложение;

— создать в процессе разработки мощный пакет документации, позволяющий в дальнейшем упростить неизбежное сопровождение и дополнения в программный продукт;

— сэкономить много времени при реализации последующих проектов, благодаря оптимально выстроенному циклу разработки приложения.

Software Engineering Institute (SEI) в Университете Карнеги-Меллон в Питсбурге установил некоторые градационные рамки, которые позволяют каждому пользователю и производителю отнести себя к одной из пяти категорий по отношению к проектированию и разработке программного обеспечения. При обследовании ряда частных фирм и государственных учреждений получены следующие результаты.

Уровень 1. **Хаотичный.** Плохое управление порядком. Отсутствие управления операциями. Высокая себестоимость проектов и проблемы с планированием. Отсутствие управления технической стороной реализации проектов, неиспользование новых средств и технологий. Под данную категорию подпадают от 74 % до 86 % всех разработчиков программного обеспечения.

Уровень 2. **Повторяющийся.** Переоценка стоимости, планирование, изменение требований, обзор состояния дел и прочее повторяются от проекта к проекту. Используются стандартные методы разработки программного обеспечения. Стоимость и планирование проектов под контролем. В данной категории от 22 % до 23 % всех разработчиков.

Уровень 3. **Определенный.** Процесс разработки определен в терминах технического стандарта разработки программного

обеспечения, включая проектирование, лицензирование кода и обучение. Только от 1 % до 4 % разработчиков достигли данного уровня.

Уровень 4. **Управляемый.** Процесс определен, оценен и хорошо управляем. Используются специальные средства для контроля и управления процессом разработки и для поддержки сбора и анализа данных. Ведется обширный анализ данных о проекте, собранных при помощи обзоров и тестирования. Практически 0 % разработчиков достигли данного уровня компетенции.

Уровень 5. **Оптимизированный.** Достигнута высокая степень управления процессом. Организация концентрирует усилия на оптимизации отдельных операций. Исчерпывающий анализ допущенных и предотвращение возможных ошибок постоянно ведет к совершенствованию процесса. Данного уровня достигли 0 %.

В настоящем учебном пособии:

— рассматриваются и подробно анализируются этапы разработки программного обеспечения, начиная от этапа его предварительного проектирования и заканчивая сдачей готового испытанного программного продукта заказчику с последующим его сопровождением;

— с позиции современной системотехники рассматриваются проблемы проектирования сложных программных средств для различных типов ЭВМ;

— излагаются технологии и основные методы разработки программного обеспечения, включая объектно-ориентированные;

— описываются методы традиционного и визуального программирования;

— представляются средства автоматизации разработки, отладки, испытаний и сопровождения программного обеспечения;

— определяется оценка качества готового программного продукта;

— описываются правила документирования программного обеспечения.

методологию разработки программного обеспечения, можно добиться следующих улучшений в типичном цикле его разработки:

— заметно сократить сроки создания комплексов программ и повысить экономический эффект их разработки;

— быстро обнаружить и устранить в процессе разработки разногласия и отсутствие связи между членами команды разработчиков и между программистами и конечными пользователями;

— упростить и облегчить для новых специалистов подключение к проекту на любой стадии его разработки;

— свести к минимуму затраты на дальнейшую доработку, модификацию и сопровождение готового продукта благодаря фундаментальному анализу задачи, на котором базируется конечное приложение;

— создать в процессе разработки мощный пакет документации, позволяющий в дальнейшем упростить неизбежное сопровождение и дополнения в программный продукт;

— сэкономить много времени при реализации последующих проектов, благодаря оптимально выстроенному циклу разработки приложения.

Software Engineering Institute (SEI) в Университете Карнеги-Меллон в Питсбурге установил некоторые градационные рамки, которые позволяют каждому пользователю и производителю отнести себя к одной из пяти категорий по отношению к проектированию и разработке программного обеспечения. При обследовании ряда частных фирм и государственных учреждений получены следующие результаты.

Уровень 1. **Хаотичный.** Плохое управление порядком. Отсутствие управления операциями. Высокая себестоимость проектов и проблемы с планированием. Отсутствие управления технической стороной реализации проектов, неиспользование новых средств и технологий. Под данную категорию подпадают от 74 % до 86 % всех разработчиков программного обеспечения.

Уровень 2. **Повторяющийся.** Переоценка стоимости, планирование, изменение требований, обзор состояния дел и прочее повторяются от проекта к проекту. Используются стандартные методы разработки программного обеспечения. Стоимость и планирование проектов под контролем. В данной категории от 22 % до 23 % всех разработчиков.

Уровень 3. **Определенный.** Процесс разработки определен в терминах технического стандарта разработки программного

обеспечения, включая проектирование, лицензирование кода и обучение. Только от 1 % до 4 % разработчиков достигли данного уровня.

Уровень 4. **Управляемый.** Процесс определен, оценен и хорошо управляем. Используются специальные средства для контроля и управления процессом разработки и для поддержки сбора и анализа данных. Ведется обширный анализ данных о проекте, собранных при помощи обзоров и тестирования. Практически 0 % разработчиков достигли данного уровня компетенции.

Уровень 5. **Оптимизированный.** Достигнута высокая степень управления процессом. Организация концентрирует усилия на оптимизации отдельных операций. Исчерпывающий анализ допущенных и предотвращение возможных ошибок постоянно ведет к совершенствованию процесса. Данного уровня достигли 0 %.

В настоящем учебном пособии:

— рассматриваются и подробно анализируются этапы разработки программного обеспечения, начиная от этапа его предварительного проектирования и заканчивая сдачей готового испытанного программного продукта заказчику с последующим его сопровождением;

— с позиции современной системотехники рассматриваются проблемы проектирования сложных программных средств для различных типов ЭВМ;

— излагаются технологии и основные методы разработки программного обеспечения, включая объектно-ориентированные;

— описываются методы традиционного и визуального программирования;

— представляются средства автоматизации разработки, отладки, испытаний и сопровождения программного обеспечения;

— определяется оценка качества готового программного продукта;

— описываются правила документирования программного обеспечения.

Глава 1. Начальные сведения

В течение нескольких последних лет информатика — исследование автоматической обработки информации формируется в самостоятельную науку. Еще не все применяемые в информатике понятия сформулированы точно, и тем не менее, особых разногласий в их применении нет.

В рамках дисциплины «Технология разработки программного обеспечения» некоторые понятия мы будем использовать в нижеописанном узком смысле.

Программа — это тексты любых программ на языке программирования или в объектном коде, пригодные для исполнения на электронно-вычислительных машинах.

Комплекс программ (КП) — это совокупность взаимосвязанных программ для электронно-вычислительных машин, в основном как объект разработки конечного программного продукта на различных этапах его создания, однако еще не достигший завершенного состояния, пригодного для тиражирования и эксплуатации с определенными качественными показателями.

Программное обеспечение (ПО), или программные средства (ПС), или программное изделие (ПИ), или программный продукт (ПП) – это совокупность программ и связанных с ними данных определенного назначения, пригодных для исполнения на электронно-вычислительных машинах, прошедших испытания с зафиксированными показателями качества и снабженных комплектом документации, достаточной для квалифицированной эксплуатации по назначению и использования как продукции производственно-технического назначения.

В процессе разработки программного обеспечения преимущественно будем использовать термин «Комплекс программ» (или просто «Программа», если он не вызовет недоразумений). И только после успешного его завершения, испытания и внедрения — термин «Программное обеспечение» («Программные средства», «Программное изделие», «Программный продукт»).

Следует заметить, что понятие «Программа» используется

программистами в самом широком диапазоне значений: от примитивной, простой программы (быть может еще не до конца написанной) до готового программного продукта. И, если это не будет усложнять понимание смысла, мы для простоты изложения также будем употреблять его в различных ситуациях по-разному – в традиционном для этого понятия смысле. Но в основном будем придерживаться приведенных выше формулировок.

С появлением новых технологий в разработку программного обеспечения (визуальные и объектно-ориентированные), появились и новые понятия: «Проект» и «Приложение» («Приложение для пользователя») как объект разработки. Разработка приложения стала его «проектированием».

Программное обеспечение как изделие

Современные программы решают самые различные задачи по содержанию и отраслевому значению.

В НИИ и в ВУЗах во многих случаях программы создаются в единственном экземпляре для решения частных исследовательских задач, для ускорения вычислений, моделирования процессов, обработки экспериментального материала и т.д. Такие программы не имеют массового применения и доступны для использования только тем, кто их разработал. Они становятся объектами научно-технического творчества и редко становятся промышленными изделиями.

Совершенно иным классом программ являются индустриальные программные средства, которые можно квалифицировать как продукцию производственно-технического назначения.

Они представляют собой программы на носителях данных с технической (эксплуатационной и технологической) документацией, разработанные в соответствии с действующими стандартами и прошедшие государственные, межведомственные или ведомственные испытания.

Программные средства, принятые в производство, изготавливаются по утвержденной в установленном порядке технологии. Они должны соответствовать утвержденным техническим условиям и действующей нормативно-технической документации, обеспечиваться гарантиями поставщика.

В этом случае речь идет о программном изделии (о программном продукте для ЭВМ).

Под ***программным изделием*** (ПИ) понимается универсальное программное обеспечение, которое предназначается для ши-

рокого круга пользователей, быть может, даже не известных заранее, и должно рекламироваться, поддерживаться в работоспособном состоянии, расширяться на протяжении длительного периода времени.

Программное изделие — это собственно программы плюс документация, гарантия качества, рекламные материалы, обучение, распространение и сопровождение. Отдельная машинная программа или совокупность программ и программное изделие — далеко не одно и то же.

Программное изделие есть продукт тщательного планирования и целенаправленной разработки, сопровождаемый четкой документацией, прошедшей все необходимые испытания; описанный в соответствующих технических публикациях, размноженный в требуемом количестве экземпляров, обслуживаемый и контролируемый поставщиком по заранее продуманному плану, и может рассматриваться как товар.

По вопросам разработки систем программного обеспечения (но не программных изделий) можно сделать следующие предположения:

— разработчик создает программное обеспечение для себя или, по крайней мере, организационно связан с пользователями разрабатываемого программного обеспечения;

— пользователь формулирует свои требования непосредственно разработчику, если последний сам не является одновременно пользователем;

— пользователь активно участвует в разработке или в обслуживании программного обеспечения;

— программное обеспечение должно работать только на определенной конфигурации комплекса технических и программных средств в ограниченном диапазоне изменений его состава и структуры данных;

— разработчик сам вводит в действие программное обеспечение у пользователя;

— проблемы, возникшие при использовании программного обеспечения, решаются пользователем совместно с разработчиком или с персоналом, осуществляющим его техническое обслуживание (сопровождение);

— программы не имеют массового применения и доступны для использования только тем, кто их разработал;

— использование программы прекращается после получения результата.

При разработке программного изделия (за исключением особого случая разработки программного обеспечения по контракту для единственного пользователя) можно сделать следующие предположения:

— разработчик не знаком с пользователем;

— требования пользователя формируются либо разработчиком, либо передаются ему посреднической организацией (например, поставляющей программное обеспечение);

— пользователи не участвуют в рассмотрении и согласовании проектных решений, если не считать редких случаев, когда их интересы представлены посредниками;

— программное обеспечение должно сохранять работоспособность в широком диапазоне конфигураций вычислительных комплексов и при самых различных системных программных средствах;

— пользователи вводят программное обеспечение в действие либо сами, либо с посторонней помощью, но эта помощь исходит не от разработчика;

— проблемы, возникшие при использовании программного обеспечения, разрешаются путем переписки, а иногда через посредника;

— программное изделие предназначено для широкого круга пользователей, быть может неизвестных;

— программное обеспечение используется многократно и длительное время.

Технология разработки программного обеспечения

Технология (с греческого: ремесло + наука) — совокупность знаний о способах и средствах проведения производственных процессов.

В одном крайнем случае один человек осуществляет поэтапную разработку программы со своего терминала в непринужденной обстановке. Естественно, он создает сравнительно небольшую программу, не требующую особой оценки.

В другом обычном случае разрабатывается очень сложное программное обеспечение, предназначенное для функционирования в реальном масштабе времени и требующее трудозатрат объемом в тысячи человеко-часов.

Эти две взаимно-противоположные ситуации характеризуются различной степенью формализации и проведения процесса разработки программных средств.

Степень формализованности и проведения процесса разработки программного обеспечения напрямую зависит от целей его создания, его величины, численности группы разработчиков и других факторов. От того, насколько правильно и удачно с точки зрения технологии разработки программного обеспечения построено приложение, зависит качество и жизнеспособность конечного продукта.

Под *технологией разработки программного обеспечения* (ТРПО) понимается совокупность обобщенных и систематизированных знаний, или наука об оптимальных способах (приемах) проведения процесса разработки программного обеспечения, обеспечивающего в заданных условиях получение программной продукции с заданными свойствами.

Технология разработки программного обеспечения представляет собой инженерный подход к разработке программных средств ЭВМ, охватывающий методологию программирования, проблемы обеспечения надежности программ, оценки рабочих характеристик и качества проектов.

Технология разработки программного обеспечения рассматривает вопросы управления проектированием систем программного обеспечения, а также средства и стандарты разработки программ.

Технология разработки программного обеспечения определяет некоторую профессиональную культуру работы специалистов (не только программистов), обеспечивающую заданный уровень производительности труда и качества получаемой в результате программной продукции.

Технология разработки программного обеспечения охватывает процесс разработки программного обеспечения от появления потребности в нем до его изготовления, передачи пользователю, модификации в процессе эксплуатации и прекращения его использования вследствие морального старения.

В идеале технология разработки программного обеспечения должна удовлетворять основным нижеперечисленным требованиям.

1. Необходима стандартизация языков проектирования программ, оформления и испытания программных модулей, а также гарантии их качества. Это позволит значительно сократить дублирующие разработки, внедрить сборочное программирование и вести накопление на предприятиях и в стране высококачественного программного продукта для его многократного использования в качестве типовых комплектующих изделий.

2. Вести постоянный контроль и обеспечение качества программ.

3. Программы не должны содержать непроверенных путей и ситуаций функционирования, которые приводят к неожиданным результатам.

4. Пользователю или покупателю программ необходимо дать четкое представление о возможностях данной программы и технологических условиях эксплуатации, при которых гарантируются определенные функции и качества.

5. Технология разработки программного обеспечения должна обеспечивать отторжимость программного изделия от его разработчика, т.е. человеческий фактор в программировании должен быть сведен к минимуму.

6. Технология разработки программного обеспечения и средства ее поддержки (автоматизации) должны обеспечивать целенаправленную работу прежде всего коллектива программистов, а не отдельных личностей. Она должна побуждать коллектив работать только правильно и слаженно; должна автоматически блокировать любые не санкционированные технологией действия.

7. Необходимо вести аккуратное документирование всех этапов разработки. Документация должна также заноситься и храниться на магнитных носителях. Доступ к этой информации должен быть открытым, простым и автоматизированным.

8. Работа пользователя должна обеспечиваться развитой информационно-справочной системой.

9. Средства автоматизации технологии должны охватывать все этапы работы коллектива программистов.

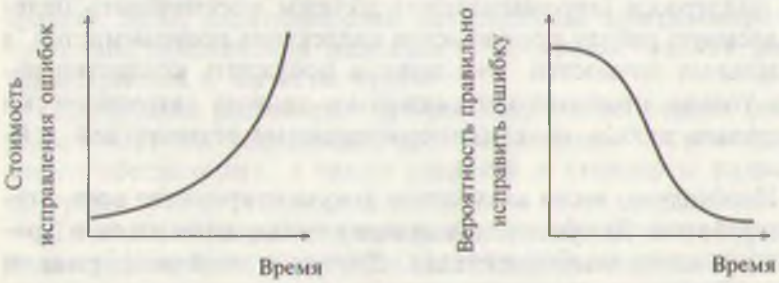
10. Технология разработки программного обеспечения должна быть простой в освоении, с автоматически включаемыми средствами подсказки.

11. Технология разработки программного обеспечения должна иметь средства автоматической фиксации в хронологическом порядке всех действий, выполняемых в процессе коллективного изготовления программного изделия — должны вестись и храниться в системе журналы (протоколы, дневники) разработки. Эти средства должны позволять восстанавливать любое состояние процесса разработки на любом интервале изготовления программного изделия, а также использоваться в процессе его эксплуатации.

Надежность программного обеспечения

В программном обеспечении имеется ошибка, если оно не выполняет того, что пользователю разумно от него ожидать. Отказ программного обеспечения — это проявление ошибки в нем. Слово «разумно» употреблено в определении для того, чтобы исключить ситуации, когда, например, к терминалу информационно-поисковой системы публичной библиотеки подходит человек и просит определить объем своего вклада в местном банке.

Ошибки в программном обеспечении не являются внутренним его свойством. Это значит, что, как бы долго и пристально мы не разглядывали (или тестировали, или доказывали) программу, мы никогда не сможем найти в ней все ошибки. Мы можем обнаружить лишь некоторые ошибки.



Зависимости стоимости и вероятности обнаружения и исправления ошибок от времени проектирования программного обеспечения

Надежность программного обеспечения есть вероятность его работы без отказов в течение определенного периода времени, рассчитанная с учетом стоимости для пользователя каждого отказа. Слово «вероятность» в определении по существу означает вероятность того, что пользователь не введет в систему некоторый конкретный набор данных, выводящий систему из строя.

Надежность также не является внутренним свойством программы. Она во многом связана с тем, как программа используется.

Надежность программного обеспечения существенно отличается от надежности аппаратуры. Программы не изнашиваются, поломка программы невозможна. Таким образом, надежность программного обеспечения — есть следствие исключения ошибок проектирования, т.е. ошибок, внесенных в процессе разработки программного обеспечения.

Надежность является составной частью более общего понятия — качества. Качественная программа, например, не только надежна, но и компактна, совместима с другими программами, эффективна, удобна в сопровождении, вполне понятна. Можно добавить: программа должна быть разработана в срок и в пределах бюджетной стоимости.

Среди прочих характеристик качества программ надежность стоит на первом месте, и поэтому дальнейшие вопросы разработки программного обеспечения рассматриваются через призму надежности.

Объектно-ориентированная разработка программ

Разработанный программный продукт решает задачи реального мира, предметы и понятия которого при объектно-ориентированной разработке программ заменяются их моделями, т.е. определенными формальными конструкциями, представляющими их в программной системе.

Модель содержит не все признаки и свойства представляемого ею предмета (понятия), а только те, которые существенны для разрабатываемой программной системы. Это значит, что модель «беднее», а, следовательно, проще отображаемого ею предмета (понятия). Но главное даже не в этом, а в том, что модель есть формальная конструкция. Формальный характер моделей позволяет определить формальные зависимости и формальные операции над ними, что значительно упрощает разработку программного продукта.

Объектно-ориентированная разработка программ помогает справиться с такими сложными проблемами, как:

— уменьшение сложности программного обеспечения;

— повышение надежности программного обеспечения;

— обеспечение возможности модификации отдельных компонентов программного обеспечения без изменения остальных его компонентов;

— обеспечение возможности повторного использования отдельных компонентов программного обеспечения.

Систематическое применение объектно-ориентированного подхода к разработке программ позволяет разрабатывать хорошо структурированные, надежные в эксплуатации, достаточно просто модифицируемые программные системы. В настоящее время объектно-ориентированная разработка программ является одним

из наиболее интенсивно развивающихся направлений в теоретическом и прикладном программировании.

Объектно-ориентированная разработка программного обеспечения использует инструментальные средства, поддерживающие объектно-ориентированные методологии (технологии) и связана с применением объектно-ориентированных моделей при разработке программных систем и их компонентов.

Можно назвать разные объектно-ориентированные методологии, например, SA/SD (Structured Analysis/Structured Design), JSD (Jackson Structured Development), OSA (Object-Oriented System Analysis).

Одной из наиболее продвинутых и популярных объектно-ориентированных методологий является OMT (Object Modeling Technique). Ее графический язык (система обозначений для диаграмм) получил достаточно широкое распространение и используется в некоторых других объектно-ориентированных методологиях, а также в большинстве публикаций по объектно-ориентированным методологиям. Методология OMT полностью поддерживается системой Paradium+, одной из наиболее известных инструментальных систем объектно-ориентированной разработки.

Жизненный цикл программного обеспечения

Понятие «жизненный цикл» предполагает нечто рождающееся, развивающееся и умирающее. Подобно живому организму программные изделия создаются, эксплуатируются и развиваются во времени.

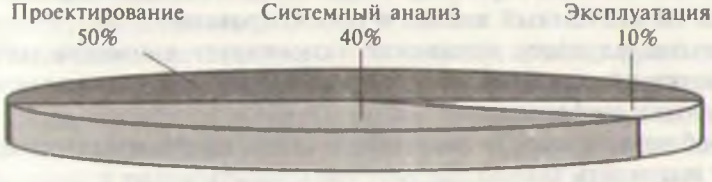
Жизненный цикл программного обеспечения включает в себя все этапы его развития: от возникновения потребности в нем до полного прекращения его использования вследствие морального старения или потери необходимости решения соответствующих задач.

Можно выделить несколько фаз существования программного изделия в течение его жизненного цикла. Общепринятых названий для этих фаз и их числа пока еще нет. Но и особых разногласий по этому вопросу нет. Поэтому существует несколько вариантов разбиения жизненного цикла программного обеспечения на этапы. Вопрос о том, лучше ли данное конкретное разбиение, чем другие, не является основным. Главное, необходимо правильно организовать разработку программного обеспечения с их учетом.

По длительности жизненного цикла программные изделия можно разделить на два класса: с *малым* и *большим временем жизни*. Этим классам программ соответствуют гибкий (мягкий) подход к их созданию и использованию и жесткий промышленный подход регламентированного проектирования и эксплуатации программных изделий. В научных организациях и вузах, например, преобладают разработки программ первого класса, а в проектных и промышленных организациях — второго.

Программные изделия с малой длительностью эксплуатации создаются в основном для решения научных и инженерных задач, для получения конкретных результатов вычислений. Такие программы обычно относительно невелики. Они разрабатываются одним специалистом или маленькой группой. Главная идея программы обсуждается одним программистом и конечным пользователем. Некоторые детали заносятся на бумагу, и проект реализуется в течение нескольких дней или недель. Они не предназначены для тиражирования и передачи для последующего использования в другие коллективы. По существу, такие программы являются частью научно-исследовательской работы и не могут рассматриваться как отчуждаемые программные изделия.

Их жизненный цикл состоит из длительного интервала системного анализа и формализации проблемы, значительного этапа проектирования программ и относительно небольшого времени эксплуатации и получения результатов. Требования, предъявляемые к функциональным и конструктивным характеристикам, как правило, не формализуются, отсутствуют оформленные испытания программ. Показатели их качества контролируются только разработчиками в соответствии с их неформальными представлениями.

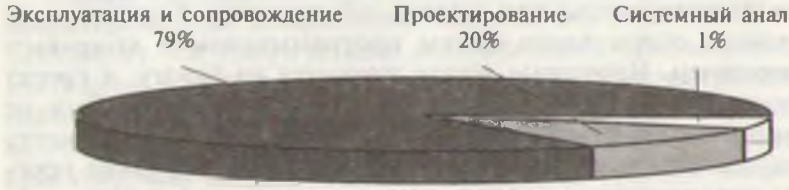


Программные изделия с малой длительностью эксплуатации

Сопровождение и модификация таких программ не обязательны, и их жизненный цикл завершается после получения результатов вычислений. Основные затраты в жизненном цикле таких программ приходятся на этапы системного анализа и проектирования, которые продолжаются от месяца до 1...2 лет, в результате

чего жизненный цикл программного изделия редко превышает 3 года.

Программные изделия с большой длительностью эксплуатации создаются для регулярной обработки информации и управления. Структура таких программ сложная. Их размеры могут изменяться в широких пределах (1...1000 тыс. команд), однако все они обладают свойствами познаваемости и возможности модификации в процессе длительного сопровождения и использования различными специалистами. Программные изделия этого класса допускают тиражирование, они сопровождаются документацией как промышленные изделия и представляют собой отчуждаемые от разработчика программные продукты.



Программные изделия с большой длительностью эксплуатации

Их проектированием и эксплуатацией занимаются большие коллективы специалистов, для чего необходима формализация программной системы, а также формализованные испытания и определение достигнутых показателей качества конечного продукта. Их жизненный цикл составляет 10...20 лет. До 70...90 % этого времени приходится на эксплуатацию и сопровождение. Вследствие массового тиражирования и длительного сопровождения совокупные затраты в процессе эксплуатации и сопровождения таких программных изделий значительно превышают затраты на системный анализ и проектирование.

Все последующее изложение акцентирует внимание на теме разработки крупных (сложных) программных средств управления и обработки информации.

Обобщенная модель *жизненного цикла* программного изделия может выглядеть так:

1. *Системный анализ:*
   а) исследования;
   б) анализ осуществимости:
      — эксплуатационной;
      — экономической;
      — коммерческой.

II. *Проектирование программного обеспечения:*

а) конструирование:

— функциональная декомпозиция системы, ее архитектура;

— внешнее проектирование программного обеспечения;

— проектирование базы данных;

— архитектура программного обеспечения;

б) программирование:

— внутреннее проектирование программного обеспечения;

— внешнее проектирование программных модулей;

— внутреннее проектирование программных модулей;

— кодирование;

— отладка программ;

— компоновка программ;

в) отладка программного обеспечения.

III. *Оценка (испытания) программного обеспечения.*

IV. *Использование программного обеспечения:*

а) эксплуатация;

б) сопровождение.

I. **Системный анализ.** В начале разработки программного обеспечения проводят системный анализ (предварительное его проектирование), в ходе которого определяются потребность в нем, его назначение и основные функциональные характеристики. Оцениваются затраты и возможная эффективность применения будущего программного изделия.

На этом этапе составляется перечень требований, то есть четкое определение того, что пользователь ожидает от готового продукта. Здесь же осуществляется постановка целей и задач, ради реализации которых и разрабатывается сам проект. В фазе системного анализа можно выделить два направления: исследование и анализ осуществимости.

Исследования начинаются с того момента, когда руководитель разработки осознает потребность в программном обеспечении.

Работа состоит в планировании и координации действий, необходимых для подготовки формального рукописного перечня требований к разрабатываемому программному изделию.

Исследования заканчиваются тогда, когда требования сформированы в таком виде, что становятся обозримыми и при необходимости могут быть модифицированы и одобрены ответственным руководителем.

Анализ осуществимости есть техническая часть исследований и начинается тогда, когда намерение руководства окрепнет нас-

только, что назначается руководитель проекта, организующий проектирование и распределение ресурсов (рабочей силы).

Работа заключается в исследовании предполагаемого программного изделия с целью получения практической оценки возможности реализации проекта, в частности определяются:

— *осуществимость эксплуатационная*, будет ли изделие достаточно удобно для практического использования?

— *осуществимость экономическая*, приемлема ли стоимость разрабатываемого изделия? Какова эта стоимость? Будет ли изделие экономически эффективным инструментом в руках пользователя?

— *осуществимость коммерческая*, будет ли изделие привлекательным, пользоваться спросом, легко устанавливаемым, приспособленным к обслуживанию, простым в освоении?

Эти и другие вопросы необходимо решать главным образом при рассмотрении указанных выше требований.

Анализ осуществимости заканчивается, когда все требования собраны и одобрены.

Прежде чем продолжить дальнейшую работу над проектом необходимо удостовериться, что вся необходимая информация получена. Эта информация должна быть точной, понятной и осуществимой. Она должна представлять собой полный комплекс требований удовлетворяющих пользователя к разрабатываемому программному продукту, оформляемый в виде спецификации.

При несоблюдении данного требования можно значительно замедлить реализацию проекта в будущем вследствие многократного повторного обращения к пользователю за уточнением неверно трактованных деталей, необговоренных условий и, как следствие, потребуется переделка уже разработанных его частей.

Часто в период системного анализа принимается решение о прекращении дальнейшей разработки программного обеспечения.

II. Проектирование программного обеспечения. Проектирование является основной и решающей фазой жизненного цикла программного обеспечения, во время которого создается и на 90% приобретает свою окончательную форму программное изделие.

Эта фаза жизни охватывает различные виды деятельности проекта и может быть разделена на три основных этапа: конструирование, программирование и отладку программного изделия.

Конструирование программного обеспечения обычно начинается ещё в фазе анализа осуществимости, как только оказываются зафиксированными на бумаге некоторые предварительные цели и требования к нему.

К моменту утверждения требований работа в фазе конструирования будет в самом разгаре.

На этом отрезке жизни программного обеспечения осуществляют:

— функциональную декомпозицию решаемой задачи, на основе которой определяется архитектура системы этой задачи;

— внешнее проектирование программного обеспечения, выражающееся в форме внешнего взаимодействия его с пользователем;

— проектирование базы данных, если это необходимо;

— проектирование архитектуры программного обеспечения — определение объектов, модулей и их сопряжения.

Программирование начинается уже в фазе конструирования, как только станут доступными основные спецификации на отдельные компоненты программного изделия, но не ранее утверждения соглашения о требованиях. Перекрытие фаз программирования и конструирования приводит к экономии общего времени разработки, а также к обеспечению проверки правильности проектных решений, и в некоторых случаях влияет на решение ключевых вопросов.

На этом этапе выполняется работа, связанная со сборкой программного изделия. Она состоит в подробном внутреннем конструировании программного продукта, в разработке внутренней логики каждого модуля системы, которая затем выражается текстом конкретной программы.

Фаза программирования завершается, когда разработчики закончат документирование, отладку и компоновку отдельных частей программного изделия в одно целое.

Отладка программного обеспечения осуществляется после того, когда все его компоненты будут отлажены по отдельности и собраны в единый программный продукт.

III. **Оценка (испытания) программного обеспечения.** В этой фазе программное изделие подвергается строгому системному испытанию со стороны группы лиц, не являющихся разработчиками.

Это делается для того, чтобы гарантировать, что готовое программное изделие удовлетворяет всем требованиям и спецификациям, может быть использовано в среде пользователя,

свободно от каких-либо дефектов и содержит необходимую документацию, которая точно и полно описывает программное изделие.

Фаза оценки начинается, как только все компоненты (модули) собраны вместе и испытаны, т.е. после полной отладки готового программного продукта. Она заканчивается после получения подтверждения, что программное изделие прошло все испытания и готово к эксплуатации.

Она продолжается так же долго, как и программирование.

IV. **Использование программного обеспечения.** Если системный анализ — сигнал к бою, проектирование — атака и возвращение с победой, то использование программного изделия — это ежедневная оборона, жизненно необходимая, но обычно не почетная для разработчиков.

Такое сравнение уместно ввиду того, что во время использования программного изделия исправляются ошибки, вкравшиеся в процессе его проектирования.

Фаза использования программного изделия начинается тогда, когда изделие передается в систему распределения.

Это то время, в течение которого изделие находится в действии и используется эффективно.

В это время выполняются обучение персонала, внедрение, настройка, сопровождение и, возможно, расширение программного изделия — так называемое продолжающееся проектирование.

Фаза использования заканчивается, когда изделие изымается из употребления и упомянутые выше действия прекращаются. Отметим, однако, что программное изделие может долго применяться кем-либо еще и после того, как фаза использования в том виде, как она определена здесь, завершится. Потому что этот некто может плодотворно использовать программное изделие у себя даже без помощи разработчика.

Использование программного продукта определяется его эксплуатацией и сопровождением.

Эксплуатация программного изделия заключается в исполнении, функционировании его на ЭВМ для обработки информации и в получении результатов, являющихся целью его создания, а также, в обеспечении достоверности и надежности выдаваемых данных.

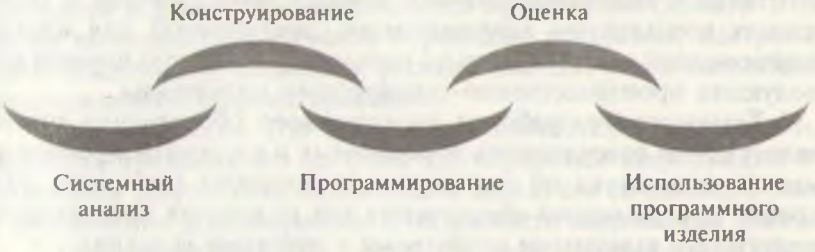
Сопровождение программного обеспечения состоит в эксплуатационном обслуживании, развитии функциональных возмож-

ностей и повышении эксплуатационных характеристик программного изделия, в тиражировании и переносе программного изделия на различные типы вычислительных средств.

Сопровождение играет роль необходимой обратной связи от этапа эксплуатации.

В процессе функционирования программного обеспечения возможно обнаружение ошибок в программах, и появляется необходимость их модификации и расширения функций.

Эти доработки, как правило, ведутся одновременно с эксплуатацией текущей версии программного изделия. После проверки подготовленных корректировок на одном из экземпляров программ очередная версия программного изделия заменяет ранее эксплуатировавшиеся или некоторые из них. При этом процесс эксплуатации программного изделия может быть практически непрерывным, так как замена версии программного изделия является кратковременной. Эти обстоятельства приводят к тому, что процесс эксплуатации версии программного изделия обычно идет параллельно и независимо от этапа сопровождения.



Перекрытие между фазами жизненного цикла программного изделия

Возможны и обычно желательны перекрытия между разными фазами жизненного цикла программного изделия. Однако не должно быть никакого перекрытия между несмежными процессами.

Возможна обратная связь между фазами. Например, во время одного из шагов внешнего проектирования могут быть обнаружены погрешности в формулировке целей, тогда нужно немедленно вернуться и исправить их.

Рассмотренная модель жизненного цикла программного изделия с некоторыми изменениями, может служить моделью и для малых проектов.

Например, когда проектируется единственная программа, то часто обходятся без проектирования архитектуры системы и

проектирования базы данных; процессы исходного и детального внешнего проектирования зачастую сливаются воедино и т.п.

РЕЗЮМЕ

* Программа – это тексты любых программ на языке программирования или в объектном коде, пригодные для исполнения на ЭВМ.

* Комплекс программ – это совокупность взаимосвязанных программ для ЭВМ, в основном как объект разработки конечного программного продукта на различных этапах его создания, однако еще не достигший завершенного состояния, пригодного для тиражирования и эксплуатации с определенными качественными показателями.

* Программное обеспечение, или программные средства, или программное изделие, или программный продукт – это совокупность программ и связанных с ними данных определенного назначения, пригодных для исполнения на ЭВМ, прошедших испытания с зафиксированными показателями качества и снабженных комплектом документации, достаточной для квалифицированной эксплуатации по назначению и использования как продукции производственно-технического назначения.

* Технология разработки программного обеспечения представляет собой совокупность обобщенных и систематизированных знаний, или наука об оптимальных способах (приемах) разработки программного обеспечения для получения программной продукции с заданными свойствами в заданных условиях.

В программном обеспечении имеется ошибка, если оно не выполняет того, что пользователю «разумно» от него ожидать.

* Надежность программного обеспечения есть «вероятность» его работы без отказов в течение определенного периода времени.

* Предметы и понятия реального мира при объектно-ориентированной разработке программ заменяются их моделями, т.е. определенными формальными конструкциями, представляющими их в программной системе.

* Жизненный цикл ПО включает в себя все этапы его развития: от возникновения потребности в нем до полного прекращения его использования.

* Необходимо организовать свое мышление и календарный план разработки программного обеспечения таким образом, чтобы учитывать его жизненный цикл.

* В ходе системного анализа (предварительного проекти-

рования) определяются потребность в ПО, его назначение и основные функциональные характеристики. Оцениваются затраты и возможная эффективность применения будущего программного продукта. Составляется перечень требований к нему.

* Проектирование является основной и решающей фазой жизненного цикла программного обеспечения, во время которой создается и на 90% приобретает свою окончательную форму программное изделие.

* На этапе программирования выполняется работа, связанная со сборкой ПИ. Она состоит в подробном внутреннем конструировании программного продукта, в разработке внутренней логики каждого модуля системы, которая затем выражается текстом конкретной программы.

* На этапе оценки (испытания) программного обеспечения ПИ подвергается строгому системному испытанию со стороны группы лиц, не являющихся разработчиками. Оно продолжается так же долго, как и программирование.

* Использование ПИ — это то время, в течение которого оно находится в действии и используется эффективно (живет). В это время выполняются обучение персонала, внедрение, настройка, сопровождение и, возможно, расширение ПИ – так называемое продолжающееся проектирование.

* Эксплуатация ПИ изделия заключается в исполнении, функционировании его на ЭВМ для обработки информации и в получении результатов, являющихся целью его создания, а также в обеспечении достоверности и надежности выдаваемых данных.

* Сопровождение ПО состоит в эксплуатационном обслуживании, развитии функциональных возможностей и повышении эксплуатационных характеристик ПИ, в тиражировании и переносе ПИ на различные типы ВС.

* Сопровождение играет роль необходимой обратной связи от этапа эксплуатации.

* Возможны и обычно желательны перекрытия между разными фазами жизненного цикла ПИ. Однако не должно быть никакого перекрытия между не смежными процессами. Возможна обратная связь между фазами.

Темы для повторения

1. Программа, комплекс программ, программное обеспечение, программные средства, программное изделие, программный продукт, проект и приложение. Что их объединяет и что — различает?

2. Программное обеспечение как изделие.
3. Технология разработки программного обеспечения.
4. Требования, предъявляемые к «идеальной» технологии разработки программного обеспечения.
5. Надежность программного обеспечения.
6. Объектно-ориентированные технологии разработки программного обеспечения.
7. Жизненный цикл программного обеспечения.
8. Программные изделия с малой и большой длительностью эксплуатации.
9. Обобщенная модель жизненного цикла программного изделия.
10. Системный анализ (предварительное проектирование).
11. Проектирование программного обеспечения.
12. Конструирование программного обеспечения.
13. Программирование программного обеспечения.
14. Оценка (испытание) программного обеспечения.
15. Использование программного обеспечения.
16. Эксплуатация программного обеспечения.
17. Сопровождение программного обеспечения.

Глава 2. Системный анализ

Разработка программного обеспечения начинается с системного анализа, после которого окончательно определяется стоимость и время исполнения проекта. Иначе этот процесс называют предварительным проектированием. Он может продлиться от двух часов до нескольких недель, в зависимости от сложности поставленной задачи.

На этом этапе проводятся анализ существующих систем, исследования проекта на осуществимость и оценка достоинств будущего разрабатываемого продукта. Этот этап жизненного цикла представляет собой итеративный процесс, основной задачей которого является определение полного комплекса требований и целей к разрабатываемому программному продукту, определяющих критерии его удовлетворенности. Работа ведется в основном в форме бесед с пользователями.

Анализ требований и пожеланий заказчика начинается с получения заказа на новую разработку (или на модификацию существующей) и заканчивается составлением документа, в деталях описывающего данную разработку.

На первом этапе проектирования программного обеспечения важно найти и понять, что же на самом деле хочет пользователь. Иной раз это не так просто сделать, поскольку пользователь не всегда представляет, что он действительно хочет получить. Банальным примером могут служить пользователи, заказывающие, например, одновременно несколько больших задач типа «Учет заработной платы», «Ведение складского учета», «Составление табеля» и т.п., называя все это «Бухгалтерией».

Первые ошибки в программном изделии прокрадываются тогда, когда определяются требования и цели к нему. Причина большинства ошибок — неправильное понимание потребности пользователя. В дальнейшем возникают и другие ошибки, когда требования и цели транслируются во внешние спецификации (в документ).

Некачественное определение требований приводит к соз-

данию программного продукта, который будет правильно решать неверно сформулированную задачу и, следовательно, он не будет соответствовать истинным потребностям заказчика. Поэтому при определении требований необходима максимально возможная аккуратность и точность, чтобы организация-разработчик программного изделия могла транслировать эти требования в проект с минимальным числом ошибок.

Требования к программному изделию традиционно задаются на естественном языке и должны быть очень точно сформулированы. Они обычно являются сводом законов, на котором базируются все последующие решения по разработке программного изделия.

О методах проверки правильности требований можно сказать, что пользователь несет ответственность за проверку требований на полноту и точность, а разработчик — за проверку осуществимости и понятности.

Результатом системного анализа должно быть ясное понимание того, что требует пользователь, и что он хочет. Тонкое различие между этими двумя понятиями немаловажно. Обычно в процессе дискуссии пользователь свои требования «озвучивает» четко и ясно, однако много существенного «остается за кадром», и не потому что пользователь намеренно молчит, а потому что он подсознательно считает некоторые требования естественными и потому не требующими специального выделения. Следует заметить, что пользователи, предъявившие минимальные требования на стадии системного анализа и оставившие разработку проекта на рассмотрение производителя, впоследствии возмущаются, что конечный продукт не удовлетворяет их требованиям, работает некорректно и поэтому требуют его переделки.

Требования к средней или крупной системе должны разрабатываться небольшой группой.

Один участник группы должен быть основным представителем организации-пользователя, облеченным достаточными полномочиями, чтобы принимать решения. Он обязательно должен быть настоящим пользователем. Вторым членом группы должен быть человек, который действительно будет пользоваться программной системой. Например, при разработке требований к системе резервирования авиабилетов, им должен быть опытный кассир.

Организация-разработчик также должна быть представлена в этой группе. Одним из ее представителей должен быть человек, который в конечном счете будет играть главную роль в процессе

внешнего проектирования, другой — в одном из процессов внутреннего проектирования.

Программные проекты можно разбить на три группы:

— управляемые пользователем;

— контролируемые пользователем;

— независимые от пользователя.

В случае управляемого пользователем проекта требования к программному изделию разрабатываются непосредственно организацией-пользователем. Разработчик программного обеспечения является ее субподрядчиком, а требования представляют собой контракт или его часть.

В проекте, контролируемом пользователем, требования к ПИ формулируются либо его разработчиком, либо совместными усилиями разработчика и организацией-пользователя. В проектах такого типа организация-пользователь имеет право утверждать требования и, как правило, спецификации следующих уровней, в частности — внешние спецификации.

В независимом от пользователя проекте вся ответственность за определение требований ложится на разработчика ПИ. Большинство проектов, связанных с разработкой общедоступных, поставляемых на рынок программ, относятся к этой группе. Однако и здесь по-прежнему крайне важно в какой-либо форме привлечь пользователя к работе над проектом.

И управляемые пользователем, и независимые от пользователя проекты должны планироваться таким образом, чтобы обеспечить участие обеих сторон.

Постановка целей

Цели — это конкретные ориентиры для программного продукта. Процесс их постановки — это прежде всего процесс принятия компромиссных решений.

Отличие цели от требования заключается в том, что *требование должно быть обязательно выполнено*, а цель допускает ее достижение с некоторым приближением.

В целом цели программного обеспечения можно разбить на 9 больших групп.

Надежность — мера работы программного обеспечения без отказов в течение определенного периода времени.

Общность — характеризуется числом, мощностью и областью действия представляемых функций программным обеспечением.

Не должно быть формулировки типа «Необходимо добиться

максимальной общности»: — должны быть просто перечислены необходимые пользователю функции. Каждая функция программного изделия должна быть оценена с точки зрения реальной ее выгоды для пользователя и ее влияния на надежность, так как «обобщенные» системы обычно больше и сложнее.

Общность конфликтует с надежностью. Конфликт между надежностью и общностью можно сгладить, избегая обобщений в тех аспектах, которые не очень или вовсе не важны для пользователя. Например, некоторые компиляторы предлагают пользователю столько дополнительных возможностей, что простое их перечисление занимает несколько страниц. Не исключено, что некоторые из них никогда не будут использованы.

Психологические факторы готового программного изделия — это мера легкости его понимания и удобства использования. Мера защищенности программного изделия от неправильного употребления и от частоты ошибок пользователя.

Хотя «гуманизация» взаимодействия с пользователем может увеличить сложность программного изделия и, таким образом, отрицательно повлияет на его надежность, психологические факторы и надежность в принципе не находятся в конфликте.

Адаптируемость — это мера легкости расширения программного изделия, например, добавление еще одной потребовавшейся пользователю функции.

Требования адаптируемости и надежности согласуются между собой. Рассматриваемые методы проектирования позволяют создавать программные изделия, которые не только более надежны, но и легче расширяются.

Удобство сопровождения — это мера затрат времени и средств на исправление ошибки в работающем программном изделии.

Удобство сопровождения согласуется с требованиями надежности, так как оно тесно связано с адаптируемостью. Методы обеспечения надежности типа обнаружения и изоляции ошибок положительно влияют на удобство сопровождения системы.

Безопасность — это мера вероятности того, что один пользователь системы может случайно или намеренно обратиться к данным, являющимся собственностью другого пользователя, разрушить их или помешать работе системы.

Средства защиты (обеспечение безопасности) включает тщательную изоляцию данных и программ разных пользователей друг от друга и от операционной системы. Безопасность обычно согласуется со стремлением к надежности.

Документация — это вопрос качества и количества публикаций для пользователя.

Цели здесь аналогичны тем, которые касаются психологических факторов, — они тоже связаны с легкостью понимания и использования продукта. Поэтому цели документирования не противоречат стремлению к надежности.

Стоимость программного изделия включает затраты на первоначальную разработку плюс сопровождение продукта.

Рост стоимости во многом вызывается возрастанием числа ошибок. Поэтому стремление к высокой надежности и желание минимизировать стоимость разработки и сопровождения не противоречат друг другу.

Календарный план — определение срока получения результата.

Одна из главных причин срывов графика — невысокая надежность создаваемого продукта. Например, если на отладку ПО будет отведено недостаточно времени, то после завершения проектирования в программном обеспечении может остаться много ошибок. И время на их исправление можно сократить, уменьшая число ошибок, допущенных в процессе проектирования.

Две тенденции: обеспечить надежность программного обеспечения и сократить календарное время — вполне согласуются между собой при условии, конечно, что календарные сроки не сокращены до такой крайности, когда на надлежащее проектирование просто не остается времени.

При разработке программного обеспечения следует рассматривать два набора целей: цели продукта, т.е. окончательного результата с точки зрения пользователя и цели проекта, такие, как график, стоимость, степень тестирования и т.д.

Цели продукта:

1. *Резюме*. Вначале следует коротко сформулировать общее назначение ПО.

2. *Определение пользователя*. Если разрабатывается большое ПО с разными группами пользователей, они должны быть определены.

3. *Перечисление функций*, обеспечивающихся программным изделием с точки зрения пользователя.

4. *Публикации* — цели для документации, поставляемой пользователю, в том числе типы документации и предполагаемый круг читателей каждого типа.

5. *Эффективность* — цели производительности, такие, как временные характеристики, пропускная способность, использование ресурсов. Также необходимые средства измерения производительности и средства настройки.

6. *Совместимость* программного изделия с другим. Указываются также относящиеся к делу международные и государственные стандарты.

7. *Конфигурация* аппаратуры и ПО, в которых система может работать и другие программные продукты, от которых она зависит.

8. *Безопасность* данных от несанкционированного доступа.

9. *Обслуживание.*

10. *Установка* — методы и средства настройки программного изделия на конкретные условия эксплуатации.

11. *Надежность.*

Цели проекта:

1. Ориентировочная стоимость каждого проекта.

2. Календарный план проекта.

3. Цели для каждого процесса тестирования.

4. Цели в области адаптируемости, указывающие степень расширяемости программного изделия, которая должна быть достигнута.

5. Вопросы сопровождения создаваемого программного изделия, которые необходимо учитывать при разработке.

6. Уровни надежности на каждом этапе разработки для достижения заданной надежности продукта.

7. Внутренняя документация при работе над проектом.

8. Критерии для готовности готового продукта к использованию.

При постановке целей распространены следующие ошибки:

— цели не формулируются явно;

— составляется беглый набросок списка целей, причем жизненно важные цели в него не включаются;

— цели конфликтуют друг с другом. Поэтому каждый программист разрешает эти конфликты сам, по-своему, что дает совершенно непредсказуемый результат. Эта ошибка, когда не определяются необходимые компромиссные решения на уровне проекта в целом, — ошибка серьезная и распространенная.

— цели формулируются только для продукта. Забывается формулировка целей для проекта.

Документ «Соглашение о требованиях»

Основной целью системного анализа является определение требований к разрабатываемому программному обеспечению, оформляемых в виде документа, название которого может варьироваться:

— соглашение о требованиях;
— техническое задание;
— технические требования;
— постановка задачи.

Как бы ни назывался документ, в нем должно содержаться письменное изложение того, что будет сделано и что не будет делаться при выпуске программного обеспечения.

Документ «Соглашение о требованиях» является основным средством управления разработкой программного обеспечения или генеральным планом его разработки.

Все участники разработки программного обеспечения должны выполнять то, что установлено в документе «Соглашение о требованиях» или запрашивать и получать разрешение на его изменение.

Предполагается, что все утверждения, включенные в соглашение о требованиях, являются требованиями, если они не определены как цели.

Каждый документ «Соглашение о требованиях» должен точно соответствовать некоторой установленной форме. Тогда каждый раздел можно будет найти в одном и том же месте аналогичного документа любой разработки программного обеспечения. В документ целесообразно включить заголовки всех предусмотренных разделов, если только специально не оговариваются условия, при которых какой-либо раздел может быть опущен. Тогда при рассмотрении документа будет решаться вопрос, действительно ли такие разделы нужны.

Соглашения о требованиях пишутся на естественном языке в терминах понятных и пользователю и разработчику программного обеспечения. Стороны должны четко представлять каждое требование.

Следует напомнить, что пользователь несет ответственность за проверку требований на полноту и точность, а разработчик - за проверку их на осуществимость и понятность.

Документ «Постановка задачи»

В организациях, специализирующихся на разработке программного обеспечения, в результате системного анализа формируется документ «Соглашение о требованиях» («Техническое задание»).

В учебном заведении такой документ традиционно называют «Постановка задачи», который является упрощенным вариантом документа «Соглашение о требованиях» и подчиняется всем требованиям последнего.

Предложения в постановке задачи пишутся на естественном языке в терминах понятных и пользователю и разработчику программного обеспечения и должны выражать однозначный смысл. Неправильное толкование предложения приводит к созданию программного продукта, который правильно решает неверно сформулированную задачу.

Документ «Постановка задачи» может содержать разделы:

1. Заголовок к программе.

2. Условие задачи. Формулируется условие задачи, краткое описание разрабатываемой программы и ее назначение.

3. Начало / окончание работы. Указывается месяц и год начала/окончания разработки программного продукта.

4. Основание для разработки. Основанием для разработки программного обеспечения может быть заказ пользователя, задание администрации учебного заведения, контракт учебного заведения с другой организацией и пр.

5. Краткая характеристика объекта. Описывается объект разработки. Как решается задача без компьютера. Какая часть ручной работы будет заменена программой и т.д.

6. Пользователь (заказчик). Указываются заказчики программного продукта, и поясняется, почему он им необходим.

7. Цель и назначение программы.

8. Основные требования. Перечисляются требования пользователя к разрабатываемому программному продукту. Здесь же с точки зрения пользователя следует перечислить функции программного продукта.

9. Входная информация. Описываются все входные данные программного обеспечения с точки зрения их содержания и назначения — отчеты, файлы, записи, поля данных, таблицы. Их возможные носители и средства отображения информации и т.д.

10. Выходная информация. Описываются выходные данные так же, как в пункте 9.

11. Требования к аппаратному и программному обеспечению. Конфигурация аппаратуры и программного обеспечения, в которых разрабатываемая система может работать и другие программные продукты, от которых она зависит.

12. Внешние ограничения.

13. Эффективность. Цели производительности, такие, как временные характеристики, пропускная способность, использование ресурсов, а также необходимые средства измерения производительности и средства настройки.

14. Безопасность данных от несанкционированного доступа.

15. Эргономические характеристики. Эргономическими характеристиками изделия являются такие свойства, которые обеспечивают надежность, комфорт и продуктивность работы пользователей и операторов.Эргономика (греч.) — труд + закон — отрасль знания, изучающая трудовые процессы с целью создания наилучших условий труда.

16. Мобильность. Описываются требования и цели обеспечения переноса программного продукта из одних рабочих условий в другие.

17. Окупаемость капиталовложений. Определяется прибыль, которую даст создание программного продукта в понятиях, соответствующих целевому назначению организации.

18. Другие соглашения сторон.

19. Терминология. Четко определяется вся терминология, которая может оказаться специфической для данной разработки.

РЕЗЮМЕ

* Системный анализ представляет собой итеративный процесс, основной задачей которого является установление полного комплекса требований и целей к разрабатываемому программному продукту, определяющих критерии его удовлетворенности. Работа ведется в основном в форме бесед с пользователями.

* На первом этапе проектирования программного обеспечения важно определить и понять, что же на самом деле хочет пользователь.

* Некачественное определение требований приводит к созданию программного продукта, который будет правильно решать неверно сформулированную задачу.

* Требования к программному изделию традиционно задаются на естественном языке и должны быть очень точно сформулированы. Они обычно являются сводом законов, на котором

базируются все последующие решения по разработке программного изделия.

* О методах проверки правильности требований можно сказать, что пользователь несет ответственность за проверку требований на полноту и точность, а разработчик — за проверку осуществимости и понятности.

* Требования к средней или крупной системе должны разрабатываться небольшой группой.

* Цель — это конкретные ориентиры для программного продукта. Процесс их постановки — это прежде всего процесс принятия компромиссных решений.

* Отличие цели от требования заключается в том, что требование должно быть обязательно выполнено, а цель допускает ее достижение с некоторым приближением.

* Документ «Соглашение о требованиях» является основным средством управления разработкой программного обеспечения или генеральным планом его разработки.

* Предполагается, что все утверждения, включенные в соглашение о требованиях, являются требованиями, если они не определены как цели.

Темы для повторения

1. Системный анализ.
2. Определение требований к программному изделию.
3. Постановка целей.
4. Документ «Соглашение о требованиях».
5. Состав группы разработки требований к средней или крупной системе.
6. Разбиение программных проектов на группы с точки зрения разработки требований к ним.
7. Цели программного продукта и проекта.
8. Документ «Соглашение о требованиях».
9. Документ «Постановка задачи».

Глава 3. Проектирование программного обеспечения

Слово «Проектирование» определяется в словаре как «Придание формы в соответствии с планом». Термин «Проектирование программного обеспечения» укоренился, или, по крайней мере, сделался популярным после двух конференций, организованных НАТО в 1968, 1969 гг.

Проектирование программного обеспечения представляет собой иерархическую декомпозицию, т.е. разбиение сложного проекта (проблемы) на ряд проще решаемых небольших проблем, которые, в свою очередь разделяются на подпроблемы до тех пор, пока каждая необходимая деталь в ней не будет определена достаточно ясно.

Концепция иерархической декомпозиции настолько естественна, что мы не всегда в состоянии осознать, как часто нам приходится использовать ее на практике. Она вытекает из человеческой потребности иметь дело с поддающимся управлению вполне определенным числом дискретных источников информации и производить «отсечение» информации до тех пор, пока число дискретных источников не станет приблизительно равно семи.

Строгая иерархическая декомпозиция подчиняется правилам:

1. На каждом уровне иерархии план или проект должен иметь законченный вид на данном уровне детализации;

2. На любом уровне иерархии каждое разбиение полностью охватывает отдельную функцию или проблему, соответствующую данному уровню детализации.

Некоторые формы иерархической декомпозиции программных средств называются так:

— нисходящее проектирование программного обеспечения;

— структурное проектирование программного обеспечения;

— структурированные планы выпуска и спецификаций изделия;

— поэтапное проектирование программного обеспечения;

— нисходящее программирование;

— модульное программирование;
— структурное программирование;

Проектирование охватывает различные виды деятельности по созданию программного обеспечения — от функциональной декомпозиции системы, определения архитектуры программного изделия до получения готового программного продукта и может быть разделено на этапы: конструирование, программирование и оценку программного продукта.

На этом этапе создается и на 90 % приобретает свою окончательную форму разрабатываемое программное обеспечение.

Проблемы проектирования больших программных средств

Большие программные средства, как правило, обладают всеми свойствами сложных систем. Они содержат большое количество (сотни и тысячи) компонент-модулей, тесно взаимосвязанных в процессе решения общей целевой задачи.

Для обеспечения взаимодействия компонент в едином комплексе широко используются иерархические структуры с несколькими уровнями группирования и подчиненности модулей.

Каждый модуль имеет свою целевую задачу и специфический частный критерий качества, как правило, не совпадающий с целевой задачей и критерием качества программного изделия в целом.

Особенно сложно в программном изделии, содержащем сотни модулей, обеспечить наилучшее использование ресурсов ЭВМ с точки зрения основного критерия эффективности при сохранении ряда частных показателей качества в допустимых пределах.

Многочисленность и сложность путей исполнения программ требует их высокой устойчивости как по отношению к ошибкам во входной информации, так и по отношению к внутренним сбоям ЭВМ, выполняющей программу, что определяется понятием «Надежность программного изделия».

Создание больших программных изделий с заданными характеристиками при ограниченных ресурсах требует проведения определенного комплекса мероприятий для достижения поставленной цели, который получил название «проект».

Целенаправленно управление проектом предназначено для пропорционального распределения ресурсов между работами по созданию программных изделий на протяжении всего цикла

проектирования вплоть до внедрения программных изделий в серийное производство.

Для значительного повышения производительности труда при разработке сложных программных изделий требуется стандартизация и комплексная автоматизация всего технологического процесса создания программного обеспечения.

На каждую программу в программном изделии должны задаваться технические условия, обеспечивающие детальную расшифровку ее функций и возможность полной проверки ее функционирования при массовом тиражировании.

Эти мероприятия позволяют обеспечить возможность широкого применения отдельных программ в различных системах без участия их разработчиков, замену устаревших компонент новыми без разрушения программного изделия.

В идеале создание больших программных изделий желательно сводить к сопряжению комплектующих изделий (групп программ или модулей).

Необходимо стандартизировать структуру и формы представления документов на каждую разработанную и испытанную программу, на программное изделие. В настоящее время такой стандарт существует и носит название «Единая система программной документации».

Другой задачей является стандартизация структуры и правил сопряжения программ при передаче управления и при обменной информации. Должны быть унифицированы правила описания и использования переменных, правила распределения памяти, требования к обмену информацией между отдельными программами, комплексами программ и автономными системами управления.

Необходима также стандартизация методов и требований к обеспечению и измерению качества сложных программных изделий. Эти методы должны позволять контролировать надежность функционирования созданных программных изделий в реальных условиях, рассчитывать и прогнозировать возможную достоверность результатов в зависимости от затрат на отладку и принятых мер для автоматического выявления искажений и для исправления его результатов.

Многолетние попытки создать универсальный алгоритмический язык высокого уровня, обеспечивающий удобную разработку различных программ при высоком их качестве по занимаемой памяти ЭВМ и использованию ее производительности, до настоящего времени не увенчались успехом.

В пределах каждого класса систем преимущественно используются 2—3 языка программирования, причем практически всегда некоторая часть программ разрабатывается на автокодах.

Стандартизацию языков программирования и структурного построения программ целесообразно рассматривать в пределах некоторых классов систем с сохранением возможности создания программ в автокодах.

Перечисленные задачи стандартизации должны объединяться единой технологической схемой и методологией создания больших программных изделий.

Проектирование больших высококачественных программных изделий — сложный и длительный процесс труда коллектива специалистов различной квалификации. Этот труд необходимо организовать, упорядочить и автоматизировать, используя современные эффективные методы и средства поддержки процесса проектирования больших программных изделий.

Все виды поддержки процесса проектирования программных изделий имеют между собой глубокую связь и в совокупности прежде всего зависят от класса и объема объекта разработки — программного изделия.

Методическая поддержка процесса проектирования программного обеспечения

Методическая поддержка процесса проектирования программного обеспечения включает в себя комплекс стандартов, инструкций и методик, определяющих правила проектирования программ.

Документы регламентируют построение объекта разработки и процесса его создания. В методиках и инструкциях конкретизируются языки проектирования программного изделия, правила использования символов и обозначений, правила структурного построения программных компонент и их взаимодействия и другие важнейшие методические принципы организации комплекса программ. Кроме того, в документах содержатся методические основы процесса создания программных изделий: правила программирования и отладки компонент, правила их испытания и оценки качества и т.д. На базе государственных и отраслевых стандартов, содержащих методические основы проектирования программных изделий, для разработки конкретного программного изделия или группы программных изделий одного класса

создаются стандарты конкретного предприятия и руководящие указания по их проектированию.

В совокупности эти документы отражают различные аспекты методологии создания конкретных программных средств.

Технологическая поддержка процесса проектирования программного обеспечения

Технологическая поддержка процесса проектирования программного обеспечения является детализацией документов методической поддержки,регламентирующих конкретную технологию обеспечения жизненного цикла программных изделий.

Документы технологической поддержки разработки тесно связаны с технологией сопровождения и эксплуатации программных изделий. Они определяют этапы проектирования, их результаты и методы контроля соблюдения предписанной технологии. Технология форм реализует методы и критерии оценки количества и качества программного продукта на различных этапах его создания. Для каждого этапа создания компонент программных изделий регламентируется допустимая трудоемкость и длительность его выполнения с учетом характеристик объекта разработки.

В технологии создания конкретных программных изделий определяется использование инструментальных средств автоматизации разработки программных изделий.

В результате технологический процесс представляется методами, документами и инструментальными средствами автоматизации, в совокупности обеспечивающими необходимое качество программных изделий при допустимых затратах различных ресурсов на их создание.

Инструментальная поддержка процесса проектирования программного обеспечения

Инструментальная поддержка процесса проектирования программного обеспечения состоит из программных средств вычислительной техники, обеспечивающих автоматизацию процесса создания программного изделия.

Уровень автоматизации и инструментальной поддержки процесса проектирования программного обеспечения зависит от оснащенности процесса разработки и сопровождения программных изделий, которая включает в себя программные средст-

ва автоматизации технологических процессов разработки, изготовления и сопровождения программных изделий, а также аппаратные средства вычислительной техники, связи и тиражирования, используемые в типовой технологии.

Программная оснащенность разработчиков программного обеспечения определяется функциональными возможностями программных систем для автоматизации разработки программного обеспечения (САРПО) или иначе систем автоматизированного проектирования программного обеспечения (САПР ПО).

Для каждого этапа разработки программного обеспечения могут применяться методы и средства, различающиеся эффективностью, которые, в частности, зависят от особенностей проекта.

В первом приближении степень программной оснащенности можно характеризовать объемом программ, активно используемых в типовой технологии.

Высокая степень программной оснащенности разработчиков достигается при автоматизации всех этапов разработки, изготовления и сопровождения программного изделия.

При этом используются следующие программные средства:

1) трансляторы программных спецификаций и текстов программ с языков высокого уровня;

2) средства планирования и контроля статического и динамического тестирования программ;

3) средства программного моделирования объектов внешней среды для программного продукта;

4) средства автоматизированного управления разработкой и конфигурационного контроля программных изделий;

5) средства современных методов автоматизации создания больших программных изделий.

Такие САПР ПО, объединяющие все эти методы и средства, имеют объемы до 1 млн команд, и стоимость их разработки составляет несколько миллионов сумов. Однако средняя программная оснащенность разработчиков программ, как правило, значительно ниже.

Требования к системе автоматизированного проектирования ПО зависят от сложности объектов разработки, имеющихся ресурсов для создания программного обеспечения, и ряда других конструктивных и организационных факторов.

Требования к подобным системам для наиболее сложного

случая создания крупных программных средств состоят в следующем:

— снижение общей трудоемкости, длительности создания программного обеспечения;

— повышение производительности труда разработчиков;

— обеспечение высокого качества и надежности функционирования создаваемых и сопровождаемых программных продуктов;

— комплексная автоматизация коллективной разработки программного обеспечения большого объема и высокой сложности;

— обеспечение унифицированной технологии разработки и сопровождения программных изделий для реализующих их ЭВМ широкого класса;

— обеспечение эффективного использования ресурсов памяти и производительности реализующих ЭВМ.

Аппаратурная оснащенность разработчиков программного обеспечения определяется мощностью используемых профессиональных персональных ЭВМ и возможностью доступа к ним разработчиков программ.

Организационная поддержка процесса проектирования программного обеспечения

Организационную поддержку процесса проектирования программного обеспечения составляют документы, регламентирующие взаимодействие специалистов внутри коллектива разработчиков и с соисполнителями, а также с заказчиками и пользователями разрабатываемых программных изделий. Они определяют права, обязанности и меру ответственности за программы конкретных специалистов и руководителей с учетом их должности и квалификации.

На организационную поддержку процесса проектирования программного обеспечения влияют методологические и технологические принципы проектирования программных изделий, а также характеристики объекта и этапов разработки.

Особенно важна четкая организационная поддержка во всем жизненном цикле больших (сложных) программных изделий. В этом случае регламентирование коллективного труда большого числа специалистов и взаимодействия руководителей проекта с заказчиком и пользователями может практически полностью определить успех всего жизненного цикла программного изделия. От того, насколько удачно построено программное обеспечение системы, зависит в итоге ее жизнеспособность.

Восходящее и нисходящее проектирование программного обеспечения

Иерархическое многоуровневое функциональное и программное построение программного обеспечения значительно облегчает организацию их проектирования и эксплуатации, сокращает длительность и стоимость их разработки.

По количеству функциональных и программных компонент на разных уровнях с учетом сложности связей между ними можно оценивать объем выполненной работы, прогнозировать ее перспективы по срокам и трудоемкости, вследствие чего повышается достоверность контроля состояния и процесса проектирования программного обеспечения.

Многоуровневый иерархический подход к проектированию позволяет проектировать сложные программные изделия по принципу *сверху — вниз* с позиции назначения и наилучшего решения основной целевой задачи всей системы. Декомпозиция проекта, разделение на иерархические уровни требует некоторых затрат. В целом ресурсы используются более эффективно, чем при отсутствии четкой иерархии, за счет экономного построения и упрощения компонент проекта на каждом уровне.

Иногда основному проектированию сверху — вниз сопутствует разработка компонент проекта *снизу — вверх*. Разработка начинается от компонента нижнего уровня, далее переходят к разработке компонента следующего уровня иерархии и т.д. Достоинством этого принципа является то, что при переходе к разработке компонентов более высокого уровня иерархии компоненты проекта нижних уровней можно считать готовыми и подключать их к проектируемым компонентам верхнего уровня.

Однако на практике при таком подходе отсутствие целостного взгляда на весь проект с позиций верхнего уровня, определяющего цели проекта, не позволяет в ряде случаев принимать верные решения, что приводит к повторной разработке или значительной корректировке компонент проекта. При разработке компонент нижнего уровня необходимо все время помнить об общих целях проекта.

Разработка программного обеспечения полностью по принципу снизу-вверх возможна лишь для сравнительно небольших групп программ, ограниченных по количеству несколькими модулями, когда разработчики способны оценивать в любое время структуру комплекса программ в целом и структуру и

функции отдельных модулей на всех уровнях иерархии.

Поэтому при разработке больших программных изделий, содержащих сотни модулей, наиболее рациональным принципом является проектирование сверху — вниз.

Часто оба метода применяются одновременно:

- сверху вниз — при объединении в единое целое;
- снизу вверх — при разработке общих хорошо отлаженных блоков.

Объектно-ориентированные технологии проектирования прикладных программных средств

Объектно-ориентированное проектирование программного обеспечения сводится прежде всего к разработке модели реальной системы, для решения задачи которой она создается.

Моделью системы (или какого-либо другого предмета или явления) называют формальное описание системы, в котором выделены основные объекты, составляющие систему, и отношения между этими объектами. В модели системы опускают многочисленные детали, усложняющие ее понимание.

Моделирование систем широко распространено в науке и технике для изучения и упрощения сложных систем.

Модели систем помогают:

- проверить работоспособность разрабатываемой системы на ранних этапах ее разработки;
- общаться с заказчиком системы, уточняя его требования к системе;
- вносить в случае необходимости изменения в проект системы как в начале ее проектирования, так и на других фазах ее жизненного цикла.

В объектно-ориентированной технологии проектируемое программное обеспечение представляется в виде трех взаимосвязанных моделей:

1) *объектной модели*, которая представляет собой статические и структурные аспекты системы, в основном связанные с данными;

2) *динамической модели*, которая описывает работу отдельных частей системы;

3) *функциональной модели*, в которой рассматривается функциональное взаимодействие отдельных частей системы (как по данным, так и по управлению) в процессе ее работы.

Эти три вида моделей позволяют получить три взаимно-ортогональных представления системы в одной системе обозначений.

Совокупность моделей системы может быть проинтерпретирована на компьютере (с помощью инструментального программного обеспечения), что позволяет продемонстрировать заказчику характер работы с будущей системой и существенно упрощает согласование предварительного проекта системы.

Модели, разработанные и отлаженные на первом этапе проектирования программного обеспечения, продолжают использоваться на всех последующих этапах его проектирования, тем самым облегчая программирование, отладку, тестирование, сопровождение и дальнейшую модификацию программного продукта. Они никак не связаны с языком программирования, на котором будет реализована система.

Объектная модель системы

Объектная модель системы описывает структуру объектов, составляющих систему, их свойства, операции и взаимосвязи с другими объектами.

В объектной модели должны быть отражены те понятия и объекты реального мира, которые важны для разрабатываемой системы. В ней отражается прежде всего прагматика разрабатываемой системы. Прагматика выражается в использовании терминологии прикладной области, связанной с использованием разрабатываемой системы.

Объектом называется понятие, абстракция или любая другая вещь с четко очерченными границами, имеющая смысл в контексте рассматриваемой прикладной проблемы. Примеры объектов: форточка, центральный банк, школа № 42, Петр Сидоров, дело № 7461, сберкнижка и т.д.

Введение объектов преследует две цели:

— понимание прикладной задачи (проблемы);

— введение основы для реализации ее на компьютере.

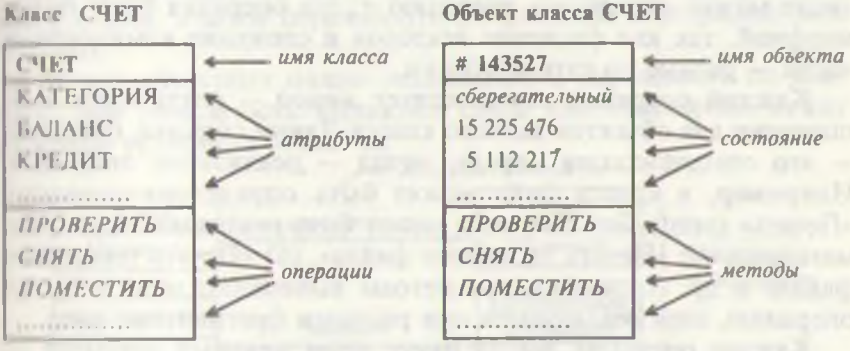
Целью разработки объектной модели является выделение и описание объектов, составляющих в совокупности проектируемую систему, а также выявление и указание различных зависимостей между объектами. Этот процесс представляет собой декомпозицию системы (проблемы) на объекты — процесс творческий и плохо формализуемый.

Все объекты системы могут иметь свои, характеризующие их

свойства, которые отличают один объект от другого. Например, объект «яблоко» может быть охарактеризован цветом, формой, весом, вкусом и пр. Между объектами можно установить отношение тождества. Тогда два объекта, удовлетворяющие этому отношению, считаются одинаковыми (тождественными) и принадлежат к одному классу.

Все объекты одного и того же класса характеризуются одинаковыми наборами свойств (атрибутов). Однако объединение объектов в классы определяется не наборами свойств, а семантикой. Так, например, объекты «Конюшня» и «Лошадь» могут иметь одинаковые атрибуты: «Цена» и «Возраст». При этом они могут относиться к одному классу, если рассматриваются в задаче просто как товар, либо к разным классам, что более естественно.

Объединение объектов в классы позволяет ввести в задачу абстракцию и рассмотреть ее в более общей постановке. Класс имеет имя (например, лошадь), которое относится ко всем объектам этого класса. Кроме того, в классе вводятся имена атрибутов (свойств), которые определены для объектов. В этом смысле описание класса аналогично описанию типа структуры (записи). При этом каждый объект имеет тот же смысл, что и экземпляр структуры (переменная или константа соответствующего типа).



Пример класса СЧЕТ

Следует напомнить, что данный этап проектирования системы не подразумевает использование какого-либо объектно-ориентированного языка программирования. Это, в частности, выражается в том, что на этом этапе следует рассматривать только такие свойства объектов, которые имеют смысл в реальности.

Свойства объектов связаны с особенностями общей реали-

зации проекта. Например, если известно, что будет использоваться база данных, в которой каждый объект имеет уникальный идентификатор, то включать этот идентификатор в число атрибутов объекта на данном этапе не следует. Дело в том, что, вводя такие свойства, мы ограничиваем возможности реализации системы. Так, вводя в качестве атрибута уникальный идентификатор объекта в базе данных, мы уже в самом начале проектирования отказываемся от использования СУБД, которые такой идентификатор не поддерживают.

Над объектом можно выполнить некоторые операции. Например, «Проверить», «Снять», «Поместить» для объектов класса «Счет» или «Открыть», «Читать», «Закрыть» для объектов класса «Файл» и т. п.

Операция — это функция (или преобразование), которую можно применить к объекту. Все объекты одного класса используют один и тот же экземпляр каждой операции (т.е. увеличение количества объектов некоторого класса не приводит к увеличению количества загруженного программного кода).

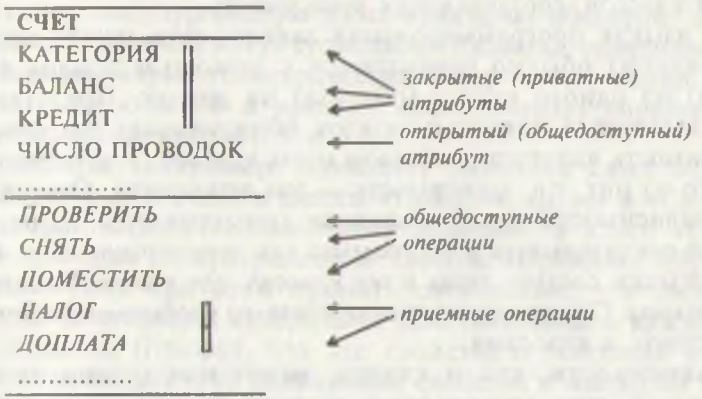
Одна и та же операция может, вообще говоря, применяться к объектам разных классов. Такая операция называется полиморфной, так как она может иметь разные формы для разных классов. Например, для объектов классов вектор и комплексное число можно определить операцию +; эта операция будет полиморфной, так как сложение векторов и сложение комплексных чисел — разные по сути операции.

Каждой операции соответствует *метод* — реализация этой операции для объектов данного класса. Таким образом, операция — это спецификация метода, метод — реализация операции. Например, в классе файл может быть определена операция «Печать» (print). Эта операция может быть реализована разными методами: (а) «Печать двоичного файла», (б) «Печать текстового файла» и др. Логически эти методы выполняют одну и ту же операцию, хотя реализуются они разными фрагментами кода.

Каждая операция всегда имеет один неявный аргумент — объект, к которому она применяется. Кроме того, операция может иметь и другие аргументы — свойства (параметры). Эти дополнительные аргументы характеризуют операцию, и никак не связаны с выбором метода.

Метод связан только с классом и объектом. (Некоторые объектно-ориентированные языки, например C++, допускают одну и ту же операцию с разным числом аргументов, причем,

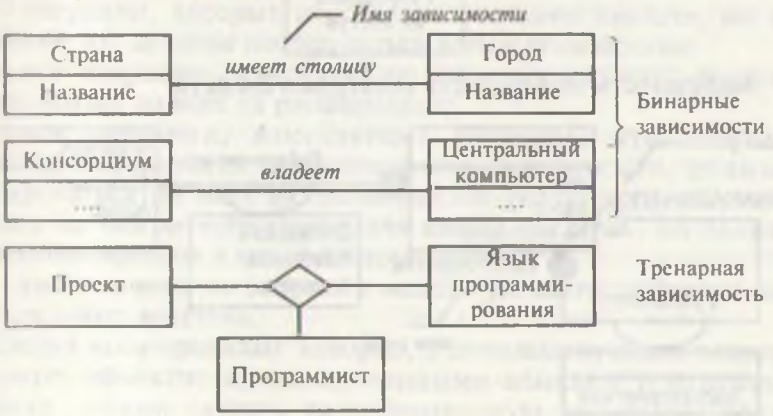
используя то или иное число аргументов, мы практически выбираем один из методов, связанных с такой операцией. На этапе предварительного проектирования системы удобнее считать эти операции различными, давая им разные имена, чтобы не усложнять проектирование).



Значения некоторых свойств объекта могут быть доступны только операциям этого объекта. Такие свойства называются закрытыми.

Итак, для задания класса объектов необходимо указать имя этого класса, а затем перечислить его свойства и операции (или методы).

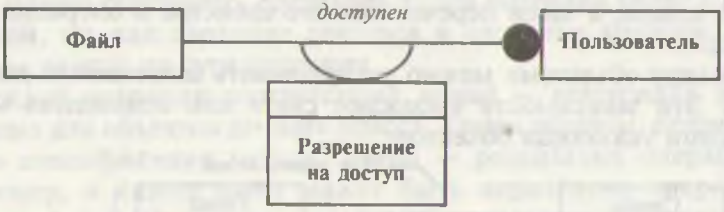
Между объектами можно устанавливать зависимости по данным. Эти зависимости выражают связи или отношения между классами указанных объектов.



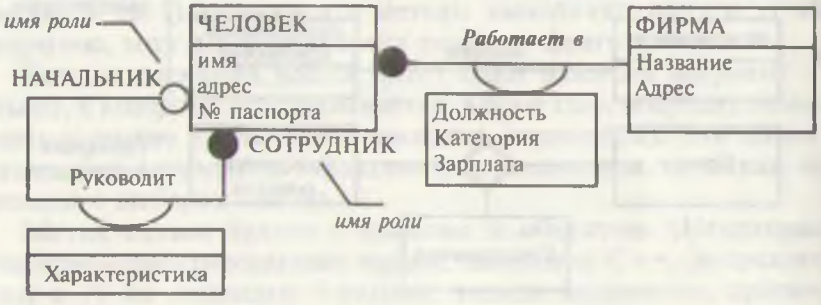
Зависимости между классами являются двусторонними: все классы в зависимости равноправны. Это так даже в тех случаях, когда имя зависимости как бы вносит направление в эту зависимость. Подобных недоразумений можно избежать, если идентифицировать зависимости не по именам, а по наименованиям ролей классов, составляющих зависимость.

В языках программирования зависимости между классами (объектами) обычно реализуются с помощью ссылок (указателей) из одного класса (объекта) на другой. Представление зависимостей с помощью ссылок обнаруживает тот факт, что зависимость является свойством пары классов, а не какого-либо одного из них, т.е. зависимость — это отношение. Отметим, что хотя зависимости между объектами двунаправлены, их не обязательно реализовывать в программах как двунаправленные. Оставлять ссылки следует лишь в тех классах, где это необходимо для программы. При проектировании системы удобнее оперировать не объектами, а классами.

Зависимости, как и классы, могут иметь свои свойства. Например, при организации доступа пользователя к файлу «Разрешение на доступ» является свойством зависимости «Доступен». Отметим, что разрешение на доступ связано как с пользователем, так и с файлом, и не может быть атрибутом ни пользователя, ни файла в отдельности.



Зависимость может иметь несколько свойств.



Иногда зависимости, имеющие много свойств, представляют с помощью классов. В базах данных такие зависимости представляются временными таблицами, организуемыми в процессе работы с базой данных.

Обобщение и *наследование* позволяют выявить аналогии, определяющие многоуровневую классификацию объектов. Так, в графических системах могут существовать классы, определяющие обрисовку различных геометрических фигур: точек, линий (прямых, дуг окружностей и кривых, определяемых сплайнами), многоугольников, кругов и т. п.

Обобщение, например, позволяет выделить класс «Одномерные фигуры» и считать классы «Прямая», «Дуга» и «Сплайн» подклассами класса «Одномерные фигуры», а класс «Одномерные фигуры» — суперклассом классов «Прямая», «Дуга» и «Сплайн». Если при этом принять соглашение, что свойства (атрибуты) и операции суперкласса действительны в каждом из его подклассов (говорят, что эти свойства и операции наследуются подклассами), то одинаковые свойства и операции классов «Прямая», «Дуга» и «Сплайн» (подклассов) могут быть вынесены в класс «Одномерные фигуры» (суперкласс).

Легко заметить, что отношения «Подкласс — суперкласс» (обобщение) и «Суперкласс — подкласс» (наследование) транзитивны. При этом свойства и операции каждого суперкласса наследуются его подклассами всех уровней (осуществляется как бы вынос за скобки одинаковых операций). Это значительно облегчает и сокращает описание классов.

Целесообразно придерживаться следующих правил наследования:

— операции, которые используют значения свойств, но не изменяют их, должны наследоваться всеми подклассами;

— все операции, изменяющие значения свойств, должны наследоваться во всех их расширениях;

— все операции, изменяющие значения ограниченных свойств, или свойств, определяющих зависимости, должны блокироваться во всех их расширениях (например, операция «размер по оси x» естественна для класса «эллипс», но должна быть заблокирована в его подклассе «круг»);

— унаследованные операции можно уточнять, добавляя дополнительные действия.

Следуя этим правилам, которые, к сожалению, редко поддерживаются объектно-ориентированными языками программирования, можно сделать разрабатываемую программу более

понятной, легче модифицируемой, менее подверженной влиянию различных ошибок и недосмотров.

Множественное наследование позволяет классу иметь более одного суперкласса, наследуя свойства (атрибуты и операции) всех своих суперклассов. Класс, имеющий несколько суперклассов, называется объединенным классом.

В объектно-ориентированном проектировании каждый объект может рассматриваться как переменная или константа структурного типа. Следовательно, множество объектов можно рассматривать как множество взаимосвязанных данных, т.е. нечто очень похожее на базу данных. Поэтому применение понятий баз данных часто оказывается полезным при объектно-ориентированном анализе и объектно-ориентированном проектировании прикладных программных систем.

РЕЗЮМЕ

* Проектирование программного обеспечения представляет собой иерархическую декомпозицию, т.е. разбиение сложного проекта (проблемы) на ряд проще решаемых небольших проблем, которые в свою очередь разделяются на подпроблемы до тех пор, пока каждая необходимая деталь в ней не будет определена достаточно ясно.

*Проектирование охватывает различные виды деятельности по созданию программного обеспечения — от функциональной декомпозиции системы, определения архитектуры программного изделия до получения готового программного продукта и может быть разделено на этапы: конструирование, программирование и оценку программного продукта.

* На этапе проектирования создается и на 90 % приобретает свою окончательную форму разрабатываемое программное обеспечение.

* Большие программные средства обладают всеми свойствами сложных систем.

* Для значительного повышения производительности труда при разработке сложных программных изделий требуется стандартизация и комплексная автоматизация всего технологического процесса создания ПО.

* Методическая поддержка процесса проектирования программного обеспечения включает в себя комплекс стандартов, инструкций и методик, определяющих правила проектирования программ.

* Технологическая поддержка процесса проектирования программного обеспечения является детализацией документов методической поддержки, регламентирующих конкретную технологию обеспечения жизненного цикла программных изделий.

* Инструментальная поддержка процесса проектирования программного обеспечения состоит из программных средств вычислительной техники, обеспечивающих автоматизацию процесса создания программного изделия.

* Организационную поддержку процесса проектирования программного обеспечения составляют документы, регламентирующие взаимодействие специалистов внутри коллектива разработчиков и с соисполнителями, а также с заказчиками и пользователями разрабатываемых ПИ.

* На организационную поддержку процесса проектирования программного обеспечения влияют методологические и технологические принципы проектирования программных изделий, а также характеристики объекта и этапов разработки.

* Иерархическое многоуровневое функциональное и программное построение программного обеспечения значительно облегчает организацию их проектирования и эксплуатации, сокращает длительность и стоимость их разработки.

* Многоуровневый иерархический подход к проектированию позволяет проектировать сложные программные изделия по принципу сверху - вниз с позиции назначения и наилучшего решения основной целевой задачи всей системы.

* Разработка программного обеспечения полностью по принципу снизу-вверх возможна лишь для сравнительно небольших групп программ, ограниченных по количеству несколькими модулями, когда разработчики способны оценивать в любое время структуру комплекса программ в целом и структуру и функции отдельных модулей на всех уровнях иерархии.

* Объектно-ориентированное проектирование программного обеспечения сводится прежде всего к разработке модели реальной системы, для решения задачи которой она создается.

* Моделью системы (или какого-либо другого предмета или явления)называют формальное описание системы, в котором выделены основные объекты, составляющие систему, и отношения между этими объектами. В модели системы опускают многочисленные детали, усложняющие ее понимание.

* Модель системы никак не связана с языком программирования, на котором будет реализована система.

* Объектная модель системы описывает структуру объектов, составляющих систему, их свойства, операции и взаимосвязи с другими объектами.

* В объектной модели должны быть отражены понятия и объекты реального мира, которые важны для разрабатываемой системы.

* Объектом называется понятие, абстракция или любая другая вещь с четко очерченными границами, имеющая смысл в контексте рассматриваемой прикладной проблемы. Объекты могут иметь свои, характеризующие их, свойства; могут объединяться в классы.

* Операция — это функция (или преобразование), которую можно применить к объекту.Ее реализация для объектов данного класса представляется методом. Метод связан только с классом и объектом.

* Для задания класса объектов необходимо указать имя этого класса, а затем перечислить его свойства и операции (или методы).

* Обобщение и наследование позволяют выявить аналогии, определяющие многоуровневую классификацию объектов.

Темы для повторения

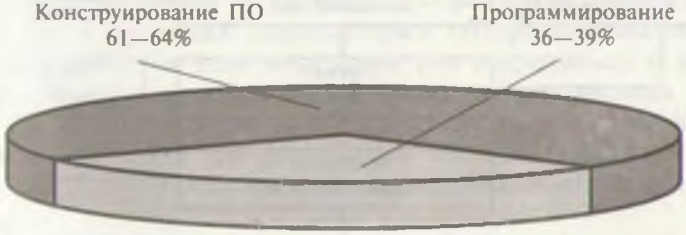
1. Проектирование программного обеспечения.
2. Проблемы проектирования больших программных средств.
3. Методическая, технологическая, инструментальная и организационная поддержки процесса проектирования программного обеспечения.
4. Система автоматизированного проектирования программного обеспечения (САПР ПО).
5. Восходящее и нисходящее проектирование программного обеспечения.
6. Объектно-ориентированное проектирование программного обеспечения.
7. Моделирование системы. Модель системы.
8. Объектная модель. Объект, класс, свойства объектов, операции, методы.
9. Обобщение и наследование.

Глава 4. Конструирование программного обеспечения

Проектирование программного обеспечения начинается, собственно, с его конструирования, которое определяет стратегию для его внутреннего проектирования - для этапа программирования. Заметим, что этот этап выполняется без использования языка программирования, но с ориентацией на определенный программный инструмент разработки ПО.

В процессе конструирования программного изделия осуществляют:

- *функциональную декомпозицию* решаемой задачи, на основе которой определяется архитектура системы, представляющей задачу;
- *внешнее проектирование* программного обеспечения, выражающееся в форме его внешнего взаимодействия с пользователем;
- *проектирование базы данных*, если это необходимо;
- *проектирование архитектуры* программного обеспечения, т.е. определение множества объектов или модулей, функционально связанных с решаемой задачей, включая сопряжения между ними и требования к ним.



Распределение допущенных ошибок в предположении, что требования сформулированы верно

Одной из наиболее опасных болезней жизненного цикла разработки программного изделия является синдром ползущего

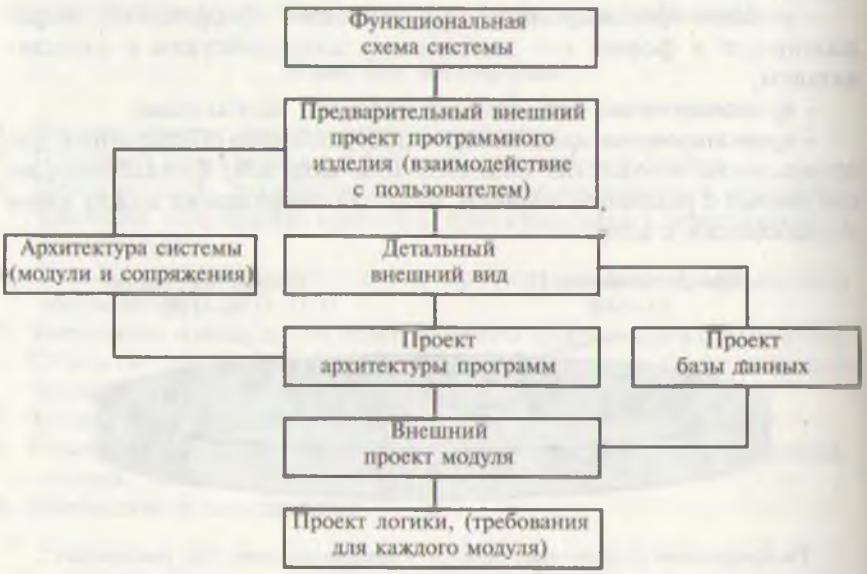
проекта или оползня. Он проявляется, когда конструирование программного изделия проведено неполно и недостаточно; неверно сконструированы отдельные аспекты проекта.

В этом случае, по мере создания программного обеспечения, пользователи, рассматривая работу отдельных готовых ветвей программы, будут просить внести некоторые усовершенствования, ссылаясь на неясные описания этого участка проекта во внешней спецификации.

Постепенно программная система будет приобретать вид огромного динозавра в заплатках. Поскольку глобальные изменения уже разработанных частей программы производить будет нельзя, а изменения и усовершенствования в нее внести надо.

Данная ситуация в первую очередь приведет к перерасходу временного лимита на создание отдельных частей проекта и нестабильности работы программного обеспечения из-за искажения или выпадения отдельных функциональных конструкций из общей строгой схемы всего проекта.

Основные принципы проектирования программного обеспечения можно представить в виде следующей схемы:



Предварительный внешний проект высокого уровня предполагает определение взаимодействия будущего программного продукта с внешним миром (обычно с пользователем), но не

рассматривает многие его мельчащие детали, такие как форматы ввода-вывода. Последнее уточняется в детальном внешнем проектировании.

Внешнее проектирование программного обеспечения

Внешнее проектирование программного обеспечения — это процесс описания внешних функций проекта и ожидаемого поведения разрабатываемого продукта с точки зрения внешнего по отношению к нему наблюдателя-пользователя.

Цель этого процесса — «конструирование» внешних взаимодействий будущего программного продукта с внешней средой (обычно с пользователем) без конкретизации его внутреннего устройства.

Внешнее проектирование программного изделия выражается в форме внешних спецификаций, предназначенных для широкой аудитории, включая пользователя (для проверки и одобрения), авторов документации для пользователя, всех участвующих в проекте программистов, а также всех тех, кто будет заниматься тестированием продукта.

Документ должен быть читаемым и хорошо логически организованным. Он должен учитывать все требования пользователя и отвечать на все вопросы пользователей и разработчиков в области функциональной разработки. Если требование пользователя не может быть удовлетворено, необходимо объяснить, почему, а не просто исключить его из спецификации.

После завершения генерации документа, необходимо отправить его пользователю для внесения поправок и комментариев. Это их первый взгляд на будущий программный продукт.

После завершения пользователем обзора документа, разработчику придется еще несколько раз встретиться с ним для обсуждения его поправок. Изменения и модификации должны быть немедленно включены в последнюю версию спецификации, чтобы техническая группа — люди, составляющие внутреннюю спецификацию, имели как можно больше неискаженной информации.

Окончательный вариант внешней спецификации в дальнейшем практически не должен изменяться. Любое его изменение на последующих стадиях вызовет цепную реакцию изменения всех последующих стадий, на которых будет значительно сложнее вносить изменения, нежели на стадии внешнего проектирования.

Спецификация внешнего проекта — это документ, объяс-

няющий в бизнес-терминах, что и в каком виде должен делать программный продукт. Все в нем должно представлять интерес для пользователя.

Он не должен быть перегружен техническими подробностями, структурами файлов и прочими технологическими деталями.

Пользователю интересно будет знать, как будет устроен интерфейс приложения: состав меню, внешний вид экрана, подсказки и помощь пользователю и т.д; какие отчеты будут представлены программой и как она будет осуществлять переход из одной точки в другую, интерактивный режим работы программного обеспечения.

Хотя методологии внешнего проектирования не существует, важно соблюдать принцип концептуальной целостности, гармонии (или стремление к ней) между внешними функциями проекта.

Концептуальная целостность представляет собой меру единообразия способа взаимодействия программного обеспечения с пользователями. Если нет единообразия, такой проект характеризуется слишком сложным взаимодействием с пользователем и излишне сложной структурой.

В зависимости от масштабов проекта ответственность за конструирование программного обеспечения должны нести один — два человека. В случае крупного проекта этим людям потребуется помощь исследователей, ассистентов, чертежников, секретарей и т.д. Помощники занимаются сбором и обработкой информации, но не проектированием, т.е. принятием решений или собственно составлением внешних спецификаций.

Внешнее проектирование программного обеспечения мало чем связано (если связано вообще) с программированием. Более непосредственно оно касается понимания обстановки, проблем и нужд пользователя, психологии общения человека с ЭВМ. Более того, эта сторона внешнего проектирования становится все более значительной по мере того, как применение ЭВМ все больше начинает затрагивать пользователей, незнакомых с программированием. Для них приходится специально разрабатывать сценарий в форме диалога программного изделия с пользователем.

Программистов можно привлекать для внешнего проектирования продукта, предназначенного для программистов, например, языков программирования или инструментов отладки, но неразумно ожидать, чтобы программист выполнил внешнее проектирование операционной системы или системы диспетчеризации грузовиков.

Из-за сложности внешнего проектирования и его возрас-

тающей важности для разработки современного программного обеспечения оно требует специалистов особого рода. Такой специалист должен разбираться в упоминавшихся выше областях, быть знакомым со всеми фазами проектирования и тестирования системы, чтобы понимать влияние на них внешнего проектирования.

При проектировании внешних сопряжений системы разработчик обычно интересуется следующими аспектами, имеющими отношение к надежности программного изделия:

— минимизацией ошибок пользователя;

— обнаружением ошибок пользователя, когда они все же возникают;

— минимизацией сложности программных изделий;

— хорошим внешним экранным оформлением;

— простотой в использовании программного изделия.

Основные правила организации диалога программного изделия с пользователем

1. Согласовывайте способ взаимодействия программного изделия с пользователем, с его подготовкой и уровнем, а также с ограничениями, в условиях которых пользователь работает.

Нельзя, например, чтобы дата рождения выводилась в форме: 2.5E1 -0.5E1 -1.001E3.

2. Сообщения, вводимые пользователем, должны быть как можно короче, но не настолько, чтобы исчезла их осмысленность. Будет лучше, если ему будет дана возможность выбора ввода сообщения, например, из списка, что исключит ошибки ввода пользователем.

3. Старайтесь, чтобы пользователь вводил как можно меньше данных с клавиатуры. Величины, которые можно вычислить в программе, вводить не надо, их надо определять в программе.

В качестве отрицательного примера можно привести случай, когда программа просит ввести код сотрудника фирмы. Разве можно запомнить все коды сотрудников в большой организации? Код сотрудника не только не надо вводить, его даже не надо выводить. Инспектор кадров не должен знать о существовании кода сотрудника. Ведение кода сотрудника — внутреннее дело программы.

4. Обеспечьте концептуальную целостность для разных типов вводимых или выводимых сообщений. Например, все сообщения

выдачи на экран дисплея, отчеты должны иметь одинаковые форматы, стиль и сокращения.

5. Обеспечьте средства «Помощи» — специальный набор функций (подсказки) по оказанию пользователю помощи, если он запутается или забудет какое-либо правило взаимодействия с программным изделием.

Средства помощи не должны содержать избыточную информацию: само-собой разумеющиеся действия в подсказку не выносят (Например, «Отказ — Esc»). Но если описание какого-либо действия в подсказку попало, то это действие в данный момент времени обязательно должно работать.

6. Помните о дизайне экрана. С эстетично оформленным экраном приятнее работать. В художественном оформлении экрана не принято использовать более трех цветов.

7. Общайтесь с пользователем на его языке, а не на «тарабарском» жаргоне программистов.

8. Старайтесь, чтобы программное изделие не рассердило пользователя, ибо это может привести к некоторым неожиданным ситуациям на входе.

Избегайте оскорбительных сообщений. Сообщение-просьба лучше, чем сообщение-приказ. Хорошо, если сообщение содержит «волшебные» слова типа «пожалуйста».

9. Обеспечьте концептуальную целостность интерактивого режима работы программного изделия. Во всех ветвях программы одинаковые действия должны инициализироваться одинаковыми клавишами. Например, переход в предыдущее состояние или отказ от предложения во всех ветвях программы всегда осуществляется нажатием клавиши Esc, выход из программы — F10 и т. д.

10. Программное изделие должно принимать любые вводимые пользователем данные.

Если данные не являются тем, что система считает допустимым, то она все равно должна их принять, информировать об этом пользователя и обработать их соответствующим образом.

11. Старайтесь на каждое входное сообщение пользователя выдавать какое-либо уведомление. Без этого пользователь может засомневаться, правильно ли сообщение было введено, и попытаться повторить ввод, вследствие чего может возникнуть чреватая ошибками ситуация.

12. Спроектируйте программное изделие так, чтобы пользователь в любой момент работы с системой мог закончить эту

работу или перейти в предыдущее состояние. Предполагается, что в первом случае система успешно завершит свою работу (закроет открытые файлы, очистит переменные памяти и т.д.).

13. Ошибки пользователя должны обнаруживаться немедленно.

14. Не стремитесь исправлять входное сообщение пользователя.

Например, в медицинской информационной системе пользователь случайно нажимает на лишнюю клавишу, вследствие чего входное сообщение принимает вид «Рэтиловый спирт» вместо входного сообщения «Этиловый спирт». Система исправляет: «Метиловый спирт».Известно, что этиловый спирт опьяняет, а метиловый спирт убивает.

15. Выходные данные должны выдаваться в требуемой форме, и обязательно прокомментированными. Нельзя, например, выдавать их в виде числа, или в виде набора цифр. Анкетные данные сотрудника фирмы, например, лучше вывести в привычной для инспектора отдела кадров виде — анкетной карточки и пр.

Разработка пользовательских интерфейсов

На ранних этапах развития вычислительной техники пользовательский интерфейс рассматривался как средство общения человека с операционной системой и был достаточно примитивным в виде командной строки. Он в основном позволял запустить программу — задание на выполнение, связать с ней конкретные данные и выполнить некоторые процедуры обслуживания вычислительной установки.

Со временем по мере совершенствования аппаратных средств появилась возможность создания интерактивного программного обеспечения, использующие специальные пользовательские интерфейсы и рассчитанного на непрофессиональных пользователей.

Пользовательский интерфейс представляет собой совокупность программных и аппаратных средств, обеспечивающих взаимодействие пользователя программы с компьютером. Основу такого взаимодействия составляют диалоги.

Под *диалогом* понимают регламентированный обмен информацией между человеком и компьютером, (программным обеспечением) осуществляемый в реальном масштабе времени и направленный на совместное решение конкретных задач: обмен

информацией между пользователем и компьютером и координация их действий.

Каждый диалог состоит из отдельных процессов ввода-вывода, которые физически обеспечивают связь между пользователем программного обеспечения и компьютером.

Обмен информацией между пользователем и компьютером осуществляется с помощью передачи сообщений и управляющих сигналов.

Сообщение — это определенная порция информации в диалоговом обмене человека и компьютера.

Различают:

— *входные сообщения*, которые генерируются человеком с помощью средств ввода (клавиатуры, манипуляторов, например «мыши» и т. п.);

— *выходные сообщения*, которые генерируются компьютером в виде текстов, звуковых сигналов и/или изображений и выводятся пользователю на экран монитора или другие устройства вывода информации.

Пользователь в основном генерирует сообщения следующих типов: запрос информации, запрос помощи, запрос операции или функции, ввод или изменение информации, выбор поля кадра и т. д.

В ответ он получает: подсказки или справки; информационные сообщения, не требующие ответа; приказы, требующие действий; сообщения об ошибках, нуждающиеся в ответных действиях; изменение формата кадра и т. д.

Схема организации взаимодействия компьютера и пользователя

Различают процедурно-ориентированный и объектно-ориентированный подходы к разработке интерфейсов.

Процедурно-ориентированные интерфейсы используют тради-

ционную модель взаимодействия программного обеспечения с пользователем, основанную на понятиях «процедура» и «операция».

В рамках этой модели программное обеспечение предоставляет пользователю возможность выполнения некоторых действий, для которых пользователь определяет соответствующие данные и следствием выполнения которых является получение желаемых результатов.

Объектно-ориентированные интерфейсы используют несколько иную модель взаимодействия программного обеспечения с его пользователем, ориентированную на манипулирование *объектами* в некоторой предметной области.

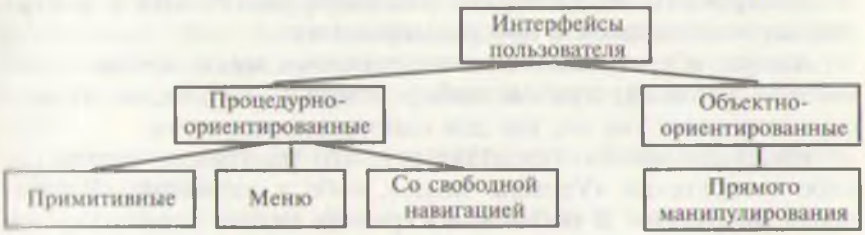
В рамках этой модели пользователю предоставляется возможность напрямую взаимодействовать с каждым объектом системы и инициировать выполнение некоторых операций программным обеспечением, в процессе которых взаимодействуют между собой несколько объектов.

Задача пользователя в этом случае формулируется как целенаправленное изменение некоторого объекта, который имеет свою внутреннюю структуру, определенное содержание и внешнее символьное или графическое представление. Объект при этом понимается в широком смысле слова, например, модель реальной системы или процесса, база данных, текст и т. п.

Пользователю программного обеспечения предоставляется возможность создавать объекты, изменять их параметры и связи с другими объектами, а также инициировать, взаимодействие этих объектов.

Элементы интерфейсов данного типа как правило уже включены в пользовательский интерфейс Windows, что значительно облегчает построение последнего.

Например, пользователь может «взять» файл в одной папке и «переместить» его в другую папку. Таким образом он инициирует выполнение операции перемещения или копирования файла.



Типы интерфейсов

Различают процедурно-ориентированные интерфейсы трех типов:

— «примитивные»;

— меню;

— со свободной навигацией.

Примитивным называют интерфейс, который организует взаимодействие с пользователем в консольном режиме (чаще всего через командную строку).

Обычно такой интерфейс реализует конкретный сценарий работы программного обеспечения, например:

— ввод данных;

— решение задачи;

— вывод результата.

Подобные интерфейсы в настоящее время используют только в процессе обучения программированию или в тех случаях, когда вся программа реализует одну функцию, например, в некоторых системных утилитах.

Интерфейс-меню в отличие от примитивного интерфейса позволяет пользователю выбирать необходимые операции из специального списка, выводимого ему программой. Эти интерфейсы предполагают реализацию множества сценариев работы, последовательность действий в которых определяется пользователем.

Различают одноуровневые, иерархические и контекстные меню.

Одноуровневое меню используется для сравнительно простого управления вычислительным процессом, когда вариантов немного (не более 5—7), и оно включает, как правило, операции одного типа, например, «Создать», «Открыть», «Закрыть» и т. п.

Иерархическое меню используется при большом количестве вариантов или при их очевидных различиях, например, операции с файлами и операции с данными, хранящимися в этих файлах.

Интерфейсы данного типа несложно реализовать в рамках структурного подхода к программированию.

Алгоритм программы с многоуровневым меню обычно строится по уровням, причем выбор команды на каждом уровне осуществляется так же, как для одноуровневого меню.

Интерфейс-меню предполагает, что программа находится либо в состоянии «Уровень меню», либо в состоянии «Выполнение операции». В состоянии «Уровень меню» осуществляется вывод меню соответствующего уровня и выбор нужного пункта

меню, а в состоянии «Выполнение операции» реализуется сценарий выбранной операции.

Древовидная организация меню предполагает строго ограниченную навигацию: либо переходы «*вверх*» к корню дерева, либо — «*вниз*» по выбранной ветви.

В условиях ограниченной навигации независимо от варианта реализации поиск требуемого пункта более чем двух уровневого меню может оказаться непростой задачей.

В порядке исключения иногда пользователю предоставляется возможность завершения операции независимо от стадии выполнения сценария и/или программы, например, по нажатию клавиши Esc.

Контекстное меню включает операции, вероятность обращения к которым из данной зоны окна приложения с точки зрения разработчика максимальна. В процессе тестирования «удобства использования» содержание контекстного меню может уточняться. Причем, чтобы облегчить пользователю поиск нужной операции целесообразно операции контекстного меню делить на группы горизонтальными линиями.

Интерфейсы со свободной навигацией также называют графическими пользовательскими интерфейсами (GUI — Graphic User Interface) или интерфейсами WYSIWYG (What You See Is What You Get — «что видишь, то и получишь»). Эти названия подчеркивают, что интерфейсы данного типа ориентированы на использование экрана в графическом режиме с высокой разрешающей способностью.

Графические интерфейсы поддерживают концепцию интерактивного взаимодействия с программным обеспечением, осуществляя визуальную обратную связь с ним и возможность прямого манипулирования объектами и информацией на экране. Кроме того, интерфейсы данного типа поддерживают концепцию совместимости программ, позволяя перемещать между ними информацию (технология OLE).

В отличие от интерфейса-меню интерфейс со свободной навигацией обеспечивает возможность осуществления любых допустимых в конкретном состоянии операций и событий, доступ к которым возможен через различные интерфейсные компоненты.

Например, окна программ, реализующих интерфейс Windows, обычно содержат:

— меню различных типов: ниспадающее, кнопочное, контекстное;

— разного рода компоненты ввода данных.

Существенной особенностью интерфейсов данного типа является способность их изменяться в процессе взаимодействия программного обеспечения с пользователем, предлагая выбор только тех операций, которые имеют смысл в конкретной ситуации.

Интерфейсы со свободной навигацией реализуют, используя событийное программирование и объектно-ориентированные библиотеки, что предполагает применение визуальных сред разработки программного обеспечения. Причем выбор следующей операции в меню может быть осуществлен как «мышью», так и с помощью клавиатуры.

Объектно-ориентированные интерфейсы пока представлены только интерфейсами прямого манипулирования. Этот тип интерфейса предполагает, что взаимодействие пользователя с программным обеспечением осуществляется посредством выбора и перемещения пиктограмм, соответствующих объектам предметной области. Для реализации таких интерфейсов также используют событийное программирование и объектно-ориентированные библиотеки.

Различают также однодокументные (SDI — Single Document Interface) и многодокументные (MDI — Multiple Document Interface) интерфейсы.

Однодокументные (или однооконные) интерфейсы организуют работу, как следует из названия, только с одним документом, например, текстом или рисунком. Чтобы посмотреть другой текст, необходимо запустить еще одну копию приложения, что допустимо только в мультипрограммной операционной системе. Такие интерфейсы используют в случаях, когда одновременная работа с несколькими документами маловероятна.

Многодокументные (или многооконные) интерфейсы соответственно организуют в тех случаях, когда велика вероятность, что пользователю понадобится одновременно работать с несколькими документами. Реализация этих интерфейсов существенно сложнее, а меню должно предусматривать специальные операции управления окнами.

Критерии оценки интерфейса пользователем. Многочисленные опросы и обследования, проводимые ведущими фирмами по разработке программного обеспечения, показали, что основными критериями оценки интерфейсов пользователем являются:

— простота освоения и запоминания операций системы —

конкретно определяется время освоения и продолжительность сохранения информации в памяти;

— скорость достижения результатов при использовании системы — определяется количеством вводимых или выбираемых мышью команд и настроек;

— субъективная удовлетворенность при эксплуатации системы (удобство работы, утомляемость и т. д.).

Причем для пользователей-профессионалов, постоянно работающих с одним и тем же пакетом, на первое место достаточно быстро выходят второй и третий критерии, а для пользователей-непрофессионалов, работающих с программным обеспечением периодически и выполняющих сравнительно несложные задачи — первый и третий.

С этой точки зрения на сегодняшний день наилучшими характеристиками для пользователей-профессионалов обладают интерфейсы со свободной навигацией, а для пользователей-непрофессионалов — интерфейсы прямого манипулирования. Давно замечено, что при выполнении операции копирования файлов при прочих равных условиях большинство профессионалов используют оболочки типа Far, а непрофессионалы — «перетаскивание объектов» Windows.

Психофизические особенности человека, связанные с восприятием, запоминанием и обработкой информации

При проектировании пользовательских интерфейсов необходимо учитывать психофизические особенности человека, связанные с восприятием, запоминанием и обработкой информации.

Исследованием принципов работы мозга человека занимается **когнитивная психология**. Специалисты в этой области предлагают упрощенную информационно-процессуальную модель мозга, представленную на рисунке.

Информация о внешнем мире поступает в наш мозг в огромных количествах. Часть мозга, которую условно можно назвать «процессором восприятия», постоянно без участия сознания перерабатывает информацию, сравнивая с прошлым опытом, и помещает в хранилище уже в виде зрительных, звуковых и прочих образов.

Любые внезапные или просто значимые для нас изменения в окружении привлекают наше внимание, и тогда интересующая нас информация поступает в кратковременную память. Если же

наше внимание не было привлечено, то информация в хранилище пропадает, замещаясь следующими порциями.

Упрощенная информационно-процессуальная модель мозга

В каждый момент времени фокус внимания человека может фиксироваться в одной точке. Поэтому, если возникает необходимость «одновременно» отслеживать несколько ситуаций, то обычно фокус перемещается с одного отслеживаемого элемента на другой. При этом внимание «рассредоточивается», и какие-то детали могут быть упущены. Например, при «прокрутке» текста или рисунка с использованием линейки прокрутки окна Windows приходится одновременно смотреть на текст, чтобы определить, где остановиться, и на ползунок. Поскольку текст важнее, фокус внимания перестает перемещаться на мышь, и она «соскакивает» с ползунка линейки.

Следует иметь в виду, что обработка процессором восприятия требует некоторого времени и, если сигнал выдается в течение времени, меньшем времени обработки, то наш мозг его не воспринимает.

Существенно и то, что восприятие во многом основано на мотивации. Например, если человек голоден, то он в первую очередь будет замечать все съедобное, а если устал — то, войдя в комнату, он в первую очередь увидит диван или кровать.

Необходимо также учитывать, что в процессе переработки информации мозг сравнивает поступающие данные с предыдущими. Так, если показать человеку последовательность символов: A, B, C, то он может принять 13 за B.

При смене кадра мозг на некоторое время блокируется: он «осваивает» новую картинку, выделяя наиболее существенные детали. А значит, если необходима быстрая реакция пользователя, то резко менять картинку не стоит.

Краткосрочная память — самое «узкое» место «системы обработки информации» человека. Ее емкость приблизительно равна 7 ± 2 несвязанных объектов. Краткосрочная память является своего рода оперативной памятью мозга, именно с ней работает процессор познания, но невостребованная информация хранится в ней не более 30 с.

Чтобы не забыть какую-нибудь важную для нас информацию, мы обычно повторяем ее «про себя», «обновляя» информацию в краткосрочной памяти. Таким образом, при проектировании интерфейсов следует иметь в виду, что подавляющему большинству людей сложно, например, запомнить и ввести на другом экране число, содержащее более 5 цифр или некоторое сочетание букв.

Люди вносят в каждую деятельность свое понимание того, как она должна выполняться. Это понимание — *модель деятельности* — базируется на прошлом опыте человека. Множество таких моделей хранится в долговременной памяти человека.

В *долговременную память* записываются постоянно повторяемые сведения или информация, связанная с сильными эмоциями.

Долговременная память человека это хранилище информации с неограниченной емкостью и временем хранения. Однако доступ к этой информации весьма непрост: по всей вероятности, механизмы извлечения информации из памяти имеют ассоциативный характер.

Специальная методика запоминания информации (мнемоника) использует именно это свойство памяти: для запоминания информации ее «привязывают» к тем данным, которые память уже хранит и позволяет их легко получить. Поскольку доступ к долговременной памяти затруднен, целесообразно рассчитывать не на то, что пользователь вспомнит нужную информацию, а на то, что он ее узнает. Именно поэтому интерфейс типа меню так широко используется.

Особенности восприятия цвета. Цвет в сознании человека ассоциируется с эмоциональным фоном. Известно, что теплые цвета: красный, оранжевый, желтый человека возбуждают, а холодные: синий, фиолетовый, серый — успокаивают. Причем цвет для человека является очень сильным раздражителем, поэтому применять цвета в интерфейсе необходимо и крайне осторожно.

Следует иметь в виду, что обилие оттенков привлекает внимание, но быстро утомляет. Поэтому не стоит ярко раскрашивать окна, с которыми пользователь будет долго работать. Необходимо учитывать и индивидуальные особенности восприятия цветов человеком (примерно каждый десятый человек плохо различает какие-либо цвета), поэтому в ответственных случаях необходимо предоставить пользователю возможность настройки цветов.

Особенности восприятия звука. В интерфейсах звук обычно используют с разными целями: для привлечения внимания, как фон, обеспечивающий некоторое состояние пользователя, как источник дополнительной информации и т. п. Применяя звук, следует учитывать, что большинство людей очень чувствительны к звуковым сигналам, особенно, если последние указывают на наличие ошибки. Поэтому при создании звукового сопровождения целесообразно предусматривать возможность его отключения.

Субъективное восприятие времени. Человеку свойственно субъективное восприятие времени. Считают, что внутреннее время связано со скоростью и количеством воспринимаемой и обрабатываемой информации. Занятый человек обычно времени не замечает. Зато в состоянии ожидания время тянется бесконечно, что связано с тем, что в это время мозг оказывается в состоянии информационного вакуума. (К аналогичному состоянию приводит и усталость: информация поступает, но больше обрабатывается, а потому и ход времени замедляется).

Доказано, что при ожидании более 1—2 секунд пользователь может отвлечься, «потерять мысль», что неблагоприятно сказывается на результатах работы и увеличивает усталость, так как каждый раз после ожидания много сил тратится на включение в работу.

Сократить время ожидания можно, заняв пользователя, но не отвлекая его от работы. Например, можно предоставить ему какую-либо информацию для обдумывания. По возможности целесообразно выводить пользователю промежуточные резуль-

таты: во-первых, он будет занят их обдумыванием, во-вторых, по ним он сможет оценить будущие результаты и отменит операцию, если они его не удовлетворяют.

Известны попытки использования для «развлечения» пользователя анимации, например, в Windows при копировании файлов демонстрируется «ролик» с летающими листочками. Однако следует иметь в виду, что при первом просмотре анимации она интересна, а когда в течение получаса наблюдаешь, как «летают» листочки при получении информации из Интернета, то это начинает раздражать.

Чтобы уменьшить раздражение, возникающее при ожидании, необходимо соблюдать основное правило: информировать пользователя, что заказанные им операции потребуют некоторого времени выполнения. Обычно для этого используют индикаторы оставшегося времени, анимированные объекты, как в Интернете, и изменение формы курсора «мыши» на песочные часы.

Очень важно точно обозначить момент, когда система готова продолжать работу. Обычно для этого используют значительные изменения внешнего вида экрана.

В итоге взаимодействие пользователя с интерфейсом будет определяться не только физическими возможностями и особенностями человека по восприятию, обработке и запоминанию информации, представленной в различных формах, а также по выполнению им разнообразных действий, но и пользовательской моделью интерфейса.

Архитектура программного обеспечения

Процесс проектирования архитектуры программного обеспечения состоит в проектировании структуры всех его компонент, функционально связанных с решаемой задачей, включая сопряжения между ними и требования к ним.

Архитектура программного обеспечения в традиционном смысле включает определение всех модулей программ, их иерархии и сопряжения между ними и данными.

Если разрабатывается отдельная программа, исходными данными для этого процесса будут детальные внешние спецификации.

Если разрабатывается система программного изделия, исходными данными для этого процесса будут детальные внешние спецификации и функциональная архитектура системы.

Традиционный метод борьбы со сложностью — принцип «Разделяй и властвуй» — часто называют «Модуляризацией».

Во время разработки архитектуры программного обеспечения выполняется его модульно-иерархическое построение.

Модуль — это замкнутая программа, которую можно вызвать из любого другого модуля в программе и можно отдельно компилировать.

Стремление к созданию модульных программ объясняется следующими факторами:

— программные модули, как правило, решают небольшую функциональную задачу, используют на входе и на выходе немного данных. Внутренние переменные модуля не связаны с внутренними переменными других модулей. Поэтому отдельные модули могут создаваться и отлаживаться различными разработчиками независимо друг от друга;

— модульные программы легко читать, сопровождать и модифицировать. Исправление отдельного модуля вызывает минимальные изменения в других модулях, связанных с ним по управлению и информации;

— модульные программы обладают повышенной надежностью, так как при их разработке существует возможность распределения работ по созданию модулей различной сложности и важности между программистами различной квалификации;

— можно создавать библиотеки наиболее употребительных подпрограмм, которые затем можно использовать в качестве комплектующих частей при разработке других приложений;

— процедура загрузки всей программы в оперативную память упрощается при использовании метода оверлейности;

— возникает много естественных контрольных точек для наблюдения за продвижением проекта по управлению и по информации.

Иерархия представляет собой свойство упорядоченного множества компонентов, между которыми установлено отношение приоритета. Компоненты, между которыми отсутствует предпочтительность, образуют один иерархический уровень.

Минимальными компонентами, из которых строятся модули, являются операторы языка программирования. Разнообразие операторов сравнительно невелико (50 — 100 типов), и каждый оператор реализуется алгоритмом на базе в среднем 1 — 10 машинных команд ЭВМ. С повышением уровня языка программирования возрастает функциональная сложность операторов.

Нижнему иерархическому уровню архитектуры программного изделия соответствуют программные и информационные в высокой степени независимые модули.

Программные модули решают небольшую функциональную задачу и реализуются 10 — 100 операторами языка программирования высокого уровня или 100 — 1000 операторами ассемблера. В результате программа модуля имеет 100 — 1000 машинных команд. Каждый модуль может использовать на входе около десятка типов переменных. Количество выходных данных несколько меньше. Если для решения небольшой функциональной задачи требуется 500 операторов или более, то целесообразно провести декомпозицию задачи на несколько более простых, для реализации каждой из которых потребуется модуль, реализуемый 10 — 500 операторами.

Модули (10 — 100 шт.) объединяются в группы программ определенного функционального назначения с автономной целевой задачей.

Несколько (5 — 20) групп программ образуют комплекс программ. В особо сложных случаях возможно создание системы программ из нескольких взаимодействующих комплексов.

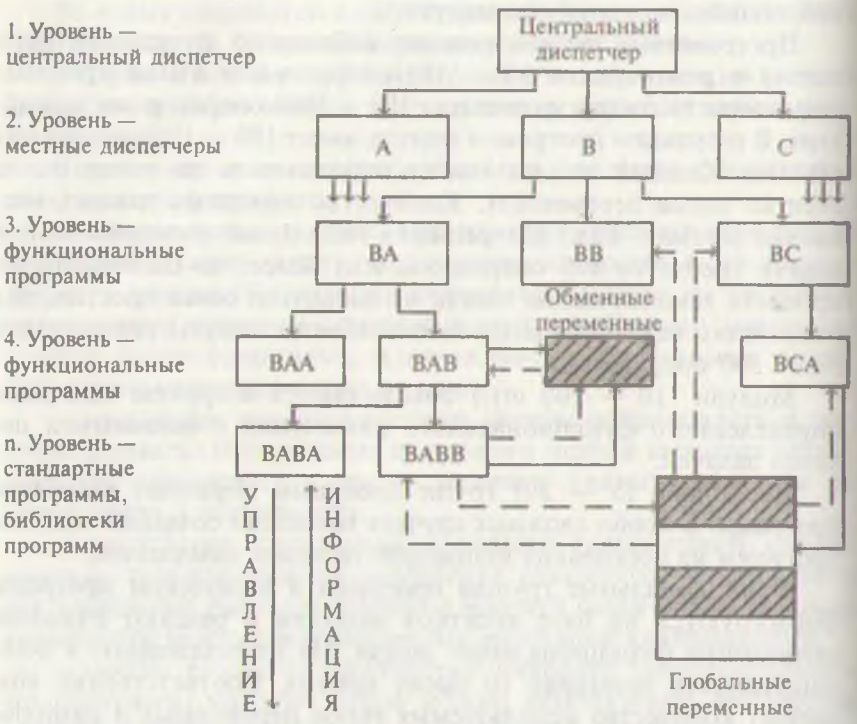
Функциональные группы программ и комплексы программ формируются на базе десятков модулей и решают сложные автономные функциональные задачи. На их реализацию в ЭВМ используется примерно 10 тысяч команд. Соответственно возрастает количество используемых типов переменных и разнообразие выходных данных. При этом значительно быстрее растет количество типов переменных, обрабатываемых модулем и локализующихся в пределах одного или нескольких модулей.

В итоге все комплексы программ объединяются в одно целое большое программное обеспечение.

Большое программное обеспечение создается для решения особо сложных задач управления и обработки информации или вычислительных задач в науке и технике. В большое программное обеспечение объединяются несколько или десятки комплексов программ для решения общей целевой задачи.

Размеры больших программных средств исчисляются сотнями модулей, десятками и сотнями тысяч машинных команд. Встречаются программные средства, содержащие до двух — трех десятков структурных иерархических уровней, построенных из модулей.

Типовая архитектура программного обеспечения может иметь вид:



Архитектуре программного обеспечения как иерархической системе присущ ряд свойств, важнейшими из которых являются:

— вертикальная соподчиненность, заключающаяся в последовательном упорядоченном расположении взаимодействующих компонент, составляющих данный комплекс программ;

— компоненты одного уровня обеспечивают реализацию функций компонент следующего уровня;

— каждый уровень иерархии реализуется через функции компонент более нижних уровней;

— каждый компонент знает о компонентах более низких уровней и ничего не знает о компонентах более высоких уровней;

— право вмешательства и приоритетного воздействия на компоненты любых уровней со стороны компонент более высоких иерархических уровней;

— взаимозависимость действий компонент верхних уровней от

реакций на воздействия и от функционирования компонент нижних уровней, информация о которых передается верхним уровням.

В результате в иерархических структурах комплексов программ образуются два потока взаимодействий между компонентами разных уровней:

— *сверху вниз* — координирующие и управляющие воздействия верхних уровней иерархии на нижние.

— *снизу вверх* — передача информации о состоянии и реализации предписанных функций компонентами нижнихуровней иерархии верхнему уровню.

Взаимодействие компонент системы предполагает выбор способа координации и реализации координирующих алгоритмов с выработкой соответствующих воздействий на них.

Координируемые компоненты имеют некоторую автономность поведения и подготовки локальных решений. Степень автономности компонентов и интенсивность координирующих воздействий устанавливается компромиссом при выделении иерархических уровней.

Взаимодействие компонентов в пределах одного уровня иерархии целесообразно максимально ограничивать, что позволяет упростить общее координирование компонентов и проводить его только по вертикали.

Компоненты нижних уровней иерархии влияют на вышестоящие непосредственно, поставляя информацию о своем состоянии и о результатах функционирования, а также косвенно, подготавливая возможные решения для их выбора на более высоких уровнях.

Информация о результатах фукционирования верхних уровней иерархии может учитываться компонентами нижних уровней для улучшения собственных решений и для косвенной координации.

Таким образом, нижние уровни иерархии являются основными компонентами, обрабатывающими информацию и подготавливающими данные для выдачи их за пределы программной среды.

Обычно лишь компоненты самого верхнего и самого нижнего уровней иерархии могут быть выведены из постановки задачи: самый верхний уровень — это требуемая система разработки, а самый нижний уровень — это доступные ресурсы: аппаратура, операционная система, инструментальные средства разработки, имеющиеся библиотеки и пр. Промежуточные уровни иерархии

определяются разработчиком системы. Чтобы перенести программную систему с уровневой архитектурой на другую платформу, достаточно переделать только компоненты самого нижнего уровня иерархии.

Уровневые архитектуры бывают двух видов: замкнутые и открытые.

В *замкнутой* архитектуре каждый уровень иерархии строится на базе непосредственно следующего за ним уровня. Это сокращает зависимости между уровнями и упрощает внесение изменений в систему.

В *открытой* архитектуре каждый уровень иерархии строится на базе всех следующих за ним уровней. Это уменьшает потребность в переопределении операций на каждом уровне и приводит к более эффективному и компактному коду. Однако она не удовлетворяет принципу упрятывания информации, поскольку изменения в какой-либо подсистеме могут потребовать соответствующих изменений в подсистемах более высоких уровней.

Общие правила структурного построения программного обеспечения

1. Программные модули, как правило, решают небольшую функциональную задачу, используют на входе и на выходе немного данных. Внутренние переменные модуля не связаны с внутренними переменными других модулей. Поэтому отдельные модули могут создаваться и отлаживаться различными разработчиками независимо друг от друга.

2. Каждый модуль характеризуется функциональной законченностью, автономностью и независимостью в оформлении от модулей, которые его используют и которые он вызывает.

Высокую степень независимости модулей можно достичь с помощью двух методов оптимизации:

— усилением внутренних связей в каждом модуле;

— ослаблением взаимосвязи между модулями.

Если программу рассматривать как набор предложений, связанных между собой некоторыми отношениями, то основное, что требуется, — это догадаться, как распределить эти предложения по отдельным модулям так, чтобы предложения внутри каждого модуля были тесно связаны, а связь между любой парой предложений в разных модулях была минимальной.

Для этого нужно стремиться:

— реализовать отдельные функции отдельными модулями (высокая прочность модуля).

— ослаблять связь между модулями по данным, применяя формальный механизм передачи параметров (слабое сцепление модулей).

3. Применяются стандартные правила организации связей по управлению и информации с другими модулями (смотри ниже).

4. Комплексы программ разрабатываются в виде совокупности небольших по количеству операторов программных модулей, связанных иерархическим образом, что дает возможность полностью и относительно просто уяснить функцию и правила работы отдельных частей и комплекса программ в целом.

5. Определена структура принятия решений. Где можно, желательно, чтобы те модули, на которые прямо влияет принятое решение, были подчиненными (вызываемыми) по отношению к принимающему решение модулю.

6. Как правило, модуль содержит от 10 до 500 выполняемых операторов языка высокого уровня. Размеры модуля влияют на степень независимости элементов программы, легкость ее чтения и тестирования.

7. Модуль прочный, прочность модуля измеряется его внутренними связями.

Модуль — это замкнутая программа, которая выполняя одну или несколько функций, обладает некоторой логикой.

Функция — это внешнее описание модуля, т.е. что делает модуль (но не как он это делает).

Логика описывает внутренний алгоритм модуля, т.е. как он выполняет функцию.

8. Модуль предсказуем, т.е. работа модуля не зависит от предыстории его использования. Все модули должны быть предсказуемыми, т.е. они не должны сохранять никаких «воспоминаний» о предыдущем вызове.

Хитрые, неуловимые, зависящие от времени ошибки возникают в тех программах, которые пытаются многократно вызывать непредсказуемый модуль.

9. Минимизация доступа к данным.

Объем данных, на которые модуль может ссылаться, должен быть сведен к минимуму. Проблема глобальных данных не должна решаться передачей одного огромного списка параметров всем модулям.

10. Внутренняя процедура (или подпрограмма) — эта замкну-

тая подпрограмма, физически расположенная в вызывающем ее модуле. Ее следует избегать, так как ее трудно изолировать для тестирования (автономного тестирования), и она не может быть вызвана из модулей, отличных от тех, которые ее физически содержат. Правда, можно включить копии внутренней процедуры во все модули, которым она нужна.

Когда возникает потребность во внутренней процедуре, проектировщик должен рассмотреть возможность оформления ее в виде отдельного модуля.

11. Обмен данными с процедурами.

Параметры процедур — это «окна во внешний мир», через которые процедура получает исходные данные и отдает свои результаты.

В параметры следует включать только те переменные, через которые идет обмен информацией с другими программными единицами. Другие переменные — это внутреннее дело процедуры и больше эти переменные никого не интересуют.

12. Процедуры, которые выдают в качестве результата только одно значение, оформляются как функции.

Функция удобнее в использовании, так как ее результат непосредственно можно использовать в арифметическом и логическом выражениях.

Правила связи программных модулей по управлению

1. Передача управления вызываемому модулю всегда осуществляется через его начало, т.е. через первый оператор или команду.

2. Если необходимо исполнить модуль с некоторой внутренней точки, то вызов осуществляется стандартным образом (через первый оператор), а точка начала задается в виде параметра, при этом в начале вызываемого модуля должен стоять переключатель, который обеспечивает передачу управления к внутренним точкам входов по параметру, указанному при обращении.

3. Выход из вызываемого модуля всегда происходит через его естественное окончание, т.е. после нормального его завершения.

4. По окончании исполнения вызываемого модуля управление передается в вызывающий модуль на оператор, следующий непосредственно за оператором вызова.

5. Модули низших уровней или одного уровня иерархии могут вызываться для исполнения только модулями высших уровней,

т.е. модули низших уровней не могут вызывать модули высших уровней, а модули одного уровня — вызывать друг друга.

6. В каждом модуле должна быть предусмотрена возможность подключения контрольных и отладочных средств; операторы, реализующие эти средства, обычно сосредоточиваются в конце модуля.

Правила связи программных модулей по информации

1. Информация зон глобальных переменных доступна для использования любым модулем, входящим в комплекс программ или в группу программ в соответствии с областью действия зоны глобальных переменных, т.е. глобальные переменные могут быть доступны не для всего комплекса программ, а лишь для указанной в описании группы модулей.

2. Локальные переменные доступны лишь в пределах того модуля, в котором они определены или объявлены.

3. Для взаимодействия вызываемых и вызывающих модулей создаются зоны обменных переменных, информация из которых доступна лишь модулям, непосредственно связанным по управлению.

4. После окончания работы вызываемого модуля считается, что регистры не содержат информации, являющейся результатом его работы. Запрещается их использовать в вызывающем модуле.

5. Информация, находящаяся в регистрах вызывающего модуля, при вызове должна быть сохранена на период выполнения вызываемого модуля и восстановлена при возврате управления в вызывающий модуль. Сохранение регистров может осуществлять как вызывающий, так и вызываемый модуль.

Конструирование объектной модели

Прикладная система представляет собой множество взаимозависимых объектов. Каждый объект характеризуется набором свойств, значения которых определяют состояние объекта, и набором операций, которые можно применять к этому объекту.

При разработке прикладных систем удобно считать, что все свойства объектов являются закрытыми (т.е. они не доступны вне объекта). Операции объектов могут быть как открытыми, так и закрытыми.

Таким образом, каждый объект имеет строго определенный интерфейс, т.е. набор открытых операций, которые можно приме-

нять к этому объекту. Все объекты одного класса имеют одинаковый интерфейс.

Объектная модель представляет статическую структуру проектируемой системы (подсистемы). Однако знания статической структуры недостаточно, чтобы понять и оценить работу подсистемы. Необходимо иметь средства для описания изменений, которые происходят с объектами и их связями во время работы подсистемы.

Одним из таких средств является *динамическая модель* подсистемы. Она строится после того, как объектная модель подсистемы построена и предварительно согласована и отлажена. Динамическая модель подсистемы состоит из диаграмм состояний ее объектов и подсистем.

Текущее состояние объекта характеризуется совокупностью текущих значений его атрибутов и связей. Во время работы системы составляющие ее объекты взаимодействуют друг с другом, в результате чего изменяются их состояния. Единицей влияния является *событие*: каждое событие приводит к смене состояния одного или нескольких объектов в системе, либо к возникновению новых событий. Работа системы характеризуется последовательностью происходящих в ней событий.

Событие происходит в некоторый момент времени. Примеры событий: старт ракеты, старт забега на 100 м, начало проводки, выдача денег и т.п. Событие не имеет продолжительности (точнее, оно занимает пренебрежимо малое время).

Если события не имеют причинной связи (т.е. они логически независимы), они называются *независимыми* (concurrent). Такие события не влияют друг на друга. Независимые события не имеет смысла упорядочивать, так как они могут происходить в произвольном порядке. Модель распределенной системы обязательно должна содержать независимые события и активности.

Последовательность событий называется сценарием. После разработки и анализа сценариев определяются объекты, генерирующие и принимающие каждое событие сценария.

При возникновении события происходит не только переход объекта в новое состояние, но и выполняется действие, связанное с этим событием. *Действием* называется мгновенная операция, связанная с событием.

Процесс построения объектной модели включает в себя следующие этапы:

— определение объектов и классов;

— подготовка словаря данных;
— определение зависимостей между объектами;
— определение свойств объектов и связей;
— организация и упрощение классов при использовании наследования;
— дальнейшее исследование и усовершенствование модели.

Определение классов

Анализ внешних требований к проектируемой прикладной системе позволяет определить объекты и классы объектов, связанные с прикладной задачей, которую должна решать эта система. Все классы должны быть осмыслены в рассматриваемой прикладной области; классов, связанных с компьютерной реализацией, как, например, список, стэк и т.п. на этом этапе вводить не следует.

Начать нужно с выделения возможных классов из письменной постановки прикладной задачи (соглашения о требованиях). Следует иметь в виду, что это очень сложный и ответственный этап разработки, так как от него во многом зависит дальнейшая судьба проекта.

При определении возможных классов нужно постараться выделить как можно больше классов, выписывая имя каждого класса, который приходит на ум. В частности, каждому существительному, встречающемуся в постановке задачи, может соответствовать класс. Поэтому при выделении возможных классов необходимо каждому существительному сопоставить класс.

Список возможных классов должен быть проанализирован с целью исключения из него «ненужных классов». Такими классами являются:

— *избыточные классы:* если два или несколько классов выражают одинаковую информацию, следует сохранить только один из них;

— *нерелевантные классы* — не имеющие прямого отношения к проблеме, они исключаются;

— *нечетко определенные классы*;

— *свойства* (атрибуты), некоторым существительным больше соответствуют не классы, а свойства (например, имя, возраст, вес, адрес и т.п.);

— *операции*, некоторым существительным больше соответствуют не классы, а имена операций, например, телефонный звонок;

— *роли:* некоторые существительные определяют имена ролей в объектной модели, например, владелец, водитель, начальник, служащий (для класса «человек»);

— *реализационные конструкции:* конструкции, больше связанные с программированием и компьютерами, не следует определять как классы.

Подготовка словаря данных, содержащего четкие и недвусмысленные определения всех объектов, классов, свойств (атрибутов) и пр. Без такого словаря обсуждение проекта с коллегами по разработке и заказчиками системы не имеет смысла, так как каждый может по-своему интерпретировать обсуждаемые термины.

Определение зависимостей. Прежде всего из классов исключаются свойства, являющиеся явными ссылками на другие классы; такие свойства заменяются зависимостями.

Аналогично тому, как имена возможных классов получались из существительных, встречающихся в предварительной постановке прикладной задачи, имена возможных зависимостей могут быть получены из глаголов или глагольных оборотов, встречающихся в указанном документе. Так обычно описываются: физическое положение («следует за», «является частью», «содержится в»), направленное действие («приводит в движение»), общение («разговаривает с»), принадлежность («имеет», «является частью») и т.п.

Затем следует убрать ненужные или неправильные зависимости, используя следующие критерии:

— *зависимости между исключенными классами* должны быть исключены, либо переформулированы в терминах оставшихся классов;

— *нерелевантные зависимости* и зависимости, связанные с реализацией, должны быть исключены;

— *тренарные зависимости*, большую часть зависимостей между тремя или большим числом классов можно разложить на несколько бинарных зависимостей;

— *производные зависимости* — зависимости, которые можно выразить через другие зависимости, так как они избыточны.

Удалив избыточные зависимости, нужно уточнить семантику оставшихся зависимостей следующим образом:

— *неверно названные зависимости* следует переименовать, чтобы смысл их стал понятен;

— *имена ролей*, их нужно добавить там, где это необходимо;

ими роли описывает роль, которую играет соответствующий класс в данной зависимости с точки зрения другого класса, участвующего в этой зависимости; если имя роли ясно из имени класса, его можно не указывать;

— *квалификаторы*, добавляя их там, где это необходимо, мы вносим элементы контекста, что позволяет добиться однозначной идентификации объектов; квалификаторы позволяют также упростить некоторые зависимости, понизив их кратность;

— *кратность зависимостей*, для них необходимо добавить обозначения; при этом следует помнить, что кратность зависимостей может меняться;

— *неучтенные зависимости* должны быть выявлены и добавлены в модель.

Уточнение свойств (атрибутов). Корректируются свойства классов, вводятся, в случае необходимости, новые.

Свойства, как правило, слабо влияют на структуру объектной модели. Не следует стремиться определить как можно больше свойств: большое количество свойств усложняет модель, затрудняет понимание проблемы.

Необходимо вводить только те свойства, которые имеют отношение к проектируемой прикладной системе, опуская случайные, малосущественные и производные свойства.

Наряду с атрибутами объектов необходимо ввести и свойства зависимостей между классами (связей между объектами).

При уточнении свойств руководствуются нижеперечисленными критериями.

Замена свойств на объекты. Если наличие некоторой сущности важнее, чем ее значение, то это объект, если же важнее значение, то это свойство.

Примеры. Начальник — это объект. Зарплата — это свойство (ее значение весьма существенно). Город — всегда объект, хотя в некоторых случаях может показаться, что это свойство (например, город как часть адреса фирмы).

В тех случаях, когда нужно, чтобы город был атрибутом, следует определить зависимость (скажем, находится) между классами фирма и город.

Квалификаторы. Если значение свойства зависит от конкретного контекста, его следует сделать квалификатором.

Имена. Именам обычно лучше соответствуют квалификаторы, чем свойства объектов; во всех случаях, когда имя позволяет

сделать выбор из объектов некоторого множества, его следует сделать квалификатором.

Идентификаторы. Идентификаторы объектов связаны с их реализацией. На ранних стадиях проектирования их не следует рассматривать в качестве свойств.

Атрибуты связей. Если некоторое свойство характеризует не объект сам по себе, а его связь с другим объектом (объектами), то это атрибут связи, а не атрибут объекта.

Внутренние значения. Свойства, определяющие лишь внутреннее состояние объекта, незаметное вне объекта, следует исключить из рассмотрения.

Несущественные детали. Свойства, не влияющие на выполнение большей части операций, рекомендуется опустить.

Организация системы классов с использованием наследования. Необходимо постараться найти суперклассы для введенных классов. Это полезно, так как проясняет структуру модели и облегчает последующую реализацию.

Дальнейшее исследование и усовершенствование модели. Лишь в очень редких случаях построенная объектная модель сразу же оказывается корректной. Модель должна быть исследована и отлажена.

Некоторые ошибки могут быть найдены при исследовании модели без компьютера, другие — при ее интерпретации совместно с динамической и функциональной моделями на компьютере (эти модели строятся после того, как объектная модель уже построена).

В основе бескомпьютерного поиска и исправления ошибок в объектной модели лежат внешние признаки, по которым можно находить ошибки в модели; эти признаки могут быть объединены в следующие группы.

Признаки пропущенного объекта (класса):

— несимметричности связей и обобщений (наследований); для исправления ошибки необходимо добавить пропущенные классы;

— несоответствие атрибутов и операций у класса; для исправления ошибки необходимо расщепить класс на несколько других классов так, чтобы атрибуты и операции новых классов соответствовали друг другу;

— обнаружена операция, не имеющая удовлетворительного целевого класса; необходимо добавить пропущенный целевой класс;

— обнаружено несколько зависимостей с одинаковыми именами и назначением; для исправления ошибки необходимо сделать обобщение и добавить пропущенный суперкласс.

Признаки ненужного (лишнего) класса:

— нехватка атрибутов, операций и зависимостей у некоторого класса; для исправления ошибки необходимо подумать, не следует ли исключить такой класс.

Признаки пропущенных зависимостей:

— отсутствуют пути доступа к операциям; для исправления ошибки необходимо добавить новые зависимости, обеспечивающие возможности обслуживания соответствующих запросов.

Признаки ненужных (лишних) зависимостей:

— избыточная информация в зависимостях; для исправления ошибки необходимо исключить зависимости, не добавляющие новой информации, или пометить их как производные зависимости;

— не хватает операций, пересекающих зависимость; для исправления ошибки необходимо подумать, не следует ли исключить такую зависимость.

Признаки неправильного размещения зависимостей:

— имена ролей слишком широки или слишком узки для их классов; для исправления ошибки необходимо переместить зависимость вверх или вниз по иерархии классов.

Признаки неправильного размещения атрибутов:

— нет необходимости доступа к объекту по значениям одного из его атрибутов; для исправления ошибки следует рассмотреть нужно ли ввести квалифицированную зависимость.

Документ «Внешняя спецификация»

Документ «Внешняя спецификация» является дальнейшим развитием документа «Соглашение о требованиях» и отражает результаты внешнего проектирования программного обеспечения. У нас принято называть этот документ «Техническим проектом». В последнее время этот документ стали чаще называть «Эскизным проектом».

Документ «Внешняя спецификация» содержит описания внешних функций проекта и ожидаемого поведения разрабатываемого продукта с точки зрения внешнего по отношению к нему наблюдателя — пользователя.

Внешняя спецификация содержит документацию описания

лишь внешних аспектов программного изделия (что представляет собой изделие) и не связано с его внутренней структурой (как программное изделие организовано).И может содержать:

— название и краткое описание программного изделия;

— функциональную схему системы;

— организацию диалога программного изделия с пользователем;

— описание меню, подменю, действий функциональных клавиш;

— все экранные формы или протокольные экранные сообщения;

— сообщения, выдаваемые пользователю во время проведения сеанса работы программного изделия и ответы на них;

— сообщения об ошибках;

— подсказки пользователю, организация помощи;

— структуру и организацию баз данных;

— описание и подготовку входных данных;

— выходные печатные формы;

— другие внешние сопряжения программного изделия.

Внешняя спецификация предназначена для широкой аудитории, включая пользователя (для проверки и одобрения), авторов документации для пользователя, всех участвующих в проекте программистов, а также всех тех, кто будет заниматься тестированием продукта.

Документ должен быть читаемым и хорошо логически организованным. Он должен учитывать все требования пользователя и отвечать на все вопросы пользователей и разработчиков в области функциональной разработки. Если требование пользователя не может быть удовлетворено, необходимо объяснить, почему, а не просто исключить его из спецификации.

Спецификация внешнего проекта — это документ, объясняющий в бизнес-терминах, что и в каком виде должен делать программный продукт. Все в нем должно представлять интерес для пользователя.

Документ пишется на естественном языке в терминах понятных и пользователю, и разработчику. Он не должен быть перегружен техническими подробностями, структурами файлов и прочими технологическими деталями.

Пользователю интересно будет знать, как будет устроен интерфейс приложения: состав меню, внешний вид экрана,

подсказки и помощь пользователю и т.д; какие отчеты будут представлены программой и как она будет осуществлять переход из одной точки в другую, интерактивный режим работы.

Желательно, чтобы спецификация была удобочитаемой, краткой, точной и исчерпывающей. Двусмысленность в спецификации недопустима.

В работе Parnas DL Technigue for software module specification with examples. Д.А. Парнас предложил следующий метод составления спецификации на программный модуль.

1. Информация о структуре вызывающего модуля не должна содержаться во внешней спецификации на вызываемый модуль.

2. Внешняя спецификация должна быть написана на понятном пользователю и производителю языке для уменьшения вероятности возможных недоразумений.

3. Модули должны быть построены таким образом, чтобы при необходимости внешних изменений в одном из них не приходилось бы изменять связанные с ним модули и процедуры.

4. Рецензент внешней спецификации должен считать, что будут реализованы только те свойства, которые определены в спецификации.

Например, если во внешней спецификации без дополнительных оговорок написано, что «параметр А может принимать любое значение в пределах от 3 до 14», то он вправе предположить, что дробные числа, такие, как 5,71 допустимы, а граничные значения 3 и 14 недопустимы. Недопустимыми являются также числа 123, 0, 2.9999, 3.001, 115.

5. Ограничения должны быть полностью и точно определены, но причину возникновения этих ограничений указывать необязательно.

Причины ограничений можно включить во внутреннюю спецификацию, если эта информация в дальнейшем будет помогать специалистам, устраняющим дефекты или расширяющим возможности модулей, избегать нарушения установленных ограничений.

6. Проверка корректности и полноты внешней спецификации должна проводиться еще до начала программирования.

Всегда целесообразно перед программированием организовывать рассмотрение этого документа пользователями или их представителями для утверждения и одобрения.

РЕЗЮМЕ

* Конструирование программного обеспечения выполняется без использования языка программирования, но с ориентацией на определенный программный инструмент разработки ПО.

* В процессе конструирования программного изделия осуществляют: функциональную декомпозицию решаемой задачи, внешнее проектирование ПО, проектирование базы данных, проектирование архитектуры ПО.

* Внешнее проектирование программного обеспечения — это процесс описания внешних функций проекта и ожидаемого поведения разрабатываемого продукта с точки зрения внешнего по отношению к нему наблюдателя-пользователя.

* Цель внешнего проектирования — «конструирование» внешних взаимодействий будущего программного продукта с внешней средой (обычно с пользователем) без конкретизации его внутреннего устройства.

* Внешнее проектирование программного изделия выражается в форме внешних спецификаций, предназначенных для широкой аудитории, включая пользователя, авторов документации для пользователя, всех участвующих в проекте программистов, а также всех тех, кто будет заниматься тестированием продукта.

* В зависимости от масштабов проекта ответственность за конструирование программного обеспечения должны нести один — два человека.

* Внешнее проектирование программного обеспечения мало чем связано (если связано вообще) с программированием. Более непосредственно оно касается понимания обстановки, проблем и нужд пользователя, психологии общения человека с ЭВМ.

* Процесс проектирования архитектуры программного обеспечения состоит в проектировании структуры всех его компонент, функционально связанных с решаемой задачей, включая сопряжения между ними и требования к ним.

* Во время традиционной разработки архитектуры программного обеспечения выполняется его модульно-иерархическое построение.

* Иерархия представляет собой свойство упорядоченного множества компонент, между которыми установлено отношение приоритета. Компоненты, между которыми отсутствует предпочтительность, образуют один иерархический уровень.

* Нижнему иерархическому уровню архитектуры программного изделия соответствуют программные и информационные в высокой степени независимые модули. Верхнему – требуемая система разработки.

* Нижние уровни иерархии являются основными компонентами, обрабатывающими информацию и подготавливающими данные для выдачи их за пределы программной среды.

* Обычно лишь компоненты самого верхнего и самого нижнего уровней иерархии могут быть выведены из постановки задачи: самый верхний уровень — это требуемая система разработки, а самый нижний уровень — это доступные ресурсы: аппаратура, операционная система, инструментальные средства разработки, имеющиеся библиотеки и пр.

* Промежуточные уровни иерархии определяются разработчиком системы.

* Чтобы перенести программную систему с уровневой архитектурой на другую платформу, достаточно переделать только компоненты самого нижнего уровня иерархии.

* В иерархических структурах комплексов программ образуются два потока взаимодействий между компонентами разных уровней:

— сверху вниз — координирующие и управляющие воздействия верхних уровней иерархии на нижние.

— снизу вверх — передача информации о состоянии и реализации предписанных функций компонентами нижних уровней иерархии верхнему уровню.

* В замкнутой архитектуре каждый уровень иерархии строится на базе непосредственно следующего за ним уровня. В открытой — каждый уровень иерархии строится на базе всех следующих за ним уровней.

* Прикладную систему можно рассматривать как множество взаимозависимых объектов. Каждый объект характеризуется набором свойств, значения которых определяют состояние объекта, и набором операций, которые можно применять к этому объекту.

* Событие происходит в некоторый момент времени и приводит к смене состояния одного или нескольких объектов в системе, либо к возникновению новых событий. Работа системы характеризуется последовательностью происходящих в ней событий.

Темы для повторения

1. Конструирование программного обеспечения.
2. Основные принципы проектирования программного обеспечения.
3. Внешнее проектирование программного обеспечения.
4. Использование концептуальной целостности при проведении внешнего проектирования.
5. Основные правила организации диалога программного обеспечения с пользователем.
6. Архитектура программного обеспечения.
7. Стремление к созданию модульных программ.
8. Архитектура программного обеспечения как иерархическая система.
9. Типовая архитектура программного обеспечения.
10. Общие правила структурного построения программного обеспечения.
11. Правила связи программных модулей по управлению.
12. Правила связи программных модулей по информации.
13. Конструирование объектной модели.
14. Процесс построения объектной модели: определение объектов и классов, подготовка словаря данных, определение зависимостей между объектами, определение свойств объектов и связей, организация и упрощение классов при использовании наследования, усовершенствование модели.
15. Документ «Внешняя спецификация».

Глава 5. Программирование

С момента появления компьютеров и компьютерных технологий, а также с возникновением такого принципиально нового и поэтому редкого вида деятельности, как программирование, разработка программного обеспечения на первых порах оставалась монополией сравнительно небольшого числа опытных программистов. Поэтому каждая готовая программа была штучным, индивидуальным продуктом высококвалифицированного специалиста.

Сам процесс программирования в недавнем прошлом был недостаточно хорошо изучен. Во всех его тонкостях и «хитростях» учиться разбираться приходилось почти «на ощупь», методом проб и ошибок. Овладение необходимыми знаниями и приемами для составления программ давалось только фанатикам-энтузиастам этого дела. В настоящее время программирование стало предметом многочисленных систематических исследований и достигло определенной степени формализации.

Современное программирование ставит своей целью разработку программного обеспечения на поточной линии. Уже имеется некоторый опыт автоматизации производства программирования с привлечением малоквалифицированных специалистов. Конечно это касается составления только таких программ, которые пишутся для не очень сложных по достижимости целей, когда речь не идет о множестве операций, принятии нестандартных решений. Безусловно, со временем будут автоматизированы в той или иной мере все этапы программирования, однако беспрерывно усложняются и решаемые задачи, поэтому при всех успехах формализации программирования и упрощения процесса овладения данной специальностью, потребность в высококвалифицированных специалистах-программистах будет всегда высокой.

Программирование начинается уже в фазе конструирования, как только становятся доступными основные спецификации на

отдельные компоненты программного обеспечения, но не ранее утверждения соглашения о требованиях. Перекрытие фаз программирования и конструирования приводит к экономии общего времени разработки, а также к обеспечению проверки правильности проектных решений, и в некоторых случаях влияет на решение ключевых вопросов.

В фазе программирования рабочая нагрузка разработки программного обеспечения достигает наибольшей величины. На этом этапе выполняется работа, связанная со сборкой программного изделия. Она состоит в подробном внутреннем конструировании программного продукта:

— внешнее проектирование каждого модуля (разработка сопряжений каждого модуля);

— проектирование логики каждого модуля (определение данных, выбор алгоритма, разработка логики, кодирование, тестирование).

Внешнее проектирование модуля может быть переходным от этапа конструирования программного обеспечения в этап программирования. Вначале времени последнего окончательно определяются все внешние характеристики каждого модуля, т.е. определяются сведения, необходимые вызывающим его модулям, и ничего больше. Хорошо эту информацию включить в виде комментария в начале текста программного модуля.

Иначе эту фазу жизненного цикла программного изделия называют «Внутренним проектированием». Эта фаза завершается, когда разработчики закончат документирование, отладку и компоновку отдельных частей программного изделия в одно целое.

Алгоритм

Слово «Алгоритм» произошло от слова «Algorithmi», являющегося латинской транслитерацией арабского имени узбекского математика Мохаммеда ибн Муса аль-Хорезми, который еще в IX веке (825 г.) описал правила выполнения четырех арифметических действий в десятичной системе счисления.

Алгоритм — точное предписание, которое задает процесс (называющийся алгоритмическим), начинающийся с произвольных исходных данных и направленный на получение полностью определяемого этими исходными данными результата.

Алгоритмический процесс представляет собой этапы (шаги) последовательного преобразования исходных данных в результирующие.

Представленный алгоритм пишется на естественном (разговорном) языке, с использованием терминов отрасли, в которой решается задача, и не должен содержать программистские термины; тем более алгоритм не может содержать операторы языка программирования. Считается, что по представленному алгоритму можно написать программу практически на любом языке программирования.

Текст алгоритма должен быть читаемым без дополнительных пояснений автора. В алгоритме не показывают описания, т.к. описания — это особенности языка программирования, а не алгоритма. Алгоритм не может быть «привязан» к конкретному языку программирования. В нем описываются конкретные действия — что делается, а не как это делается.

Основные требования к блок-схеме:

— схема выполняется в обозначениях ГОСТа;

— после названия алгоритма необходимо описать целевые функции алгоритма в целом;

— в блоках схемы обычно пишут «скупые» тексты. Поэтому ее дополняют пояснениями в разделе «Пояснения к алгоритму»;

— в пояснении к схеме описывают действия отдельных частей схемы и в случае крайней необходимости — действия отдельного блока. Описание каждого блока схемы бессмысленно. Пояснения должны содержать некоторую дополнительную информацию;

— управление по схеме должно в основном идти вниз (вправо), возвращаясь назад только в циклах;

— альтернативно выполняемые ветви должны размещаться параллельно;

— переменные должны быть инициализированы в каком-либо блоке с описанием. Если комментарии к переменной не помещаются в блоке, то ее описание необходимо поместить в раздел «Пояснения к алгоритму»;

— входные и выходные блоки процедур должны содержать, соответственно, входные и выходные (формальные) параметры;

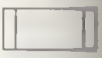
— блоки можно объединять в более крупные пунктирными линиями, которые нужно комментировать — описывать их назначение.

Виды блоков в схемах алгоритмов

1. Процесс 

2. Решение, условие 

3. Модификация 

4. Предварительный процесс, подалгоритм 

5. Ручная операция 

6. Вспомогательная операция 

7. Слияние 

8. Выделение 

9. Группировка 

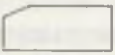
10. Сортировка 

11. Ручной ввод 

12. Ввод / вывод 

13. Неавтономная память 

14. Документ 

15. Перфокарта 

16. Колода перфокарт 

17. Перфолента 

18. Магнитная лента 

19. Магнитный барабан 

20. Магнитный диск 

21. Оперативная память 

22. Дисплей 

23. Линия потоков

24. Параллельные действия

25. Канал связи 

26. Соединитель 

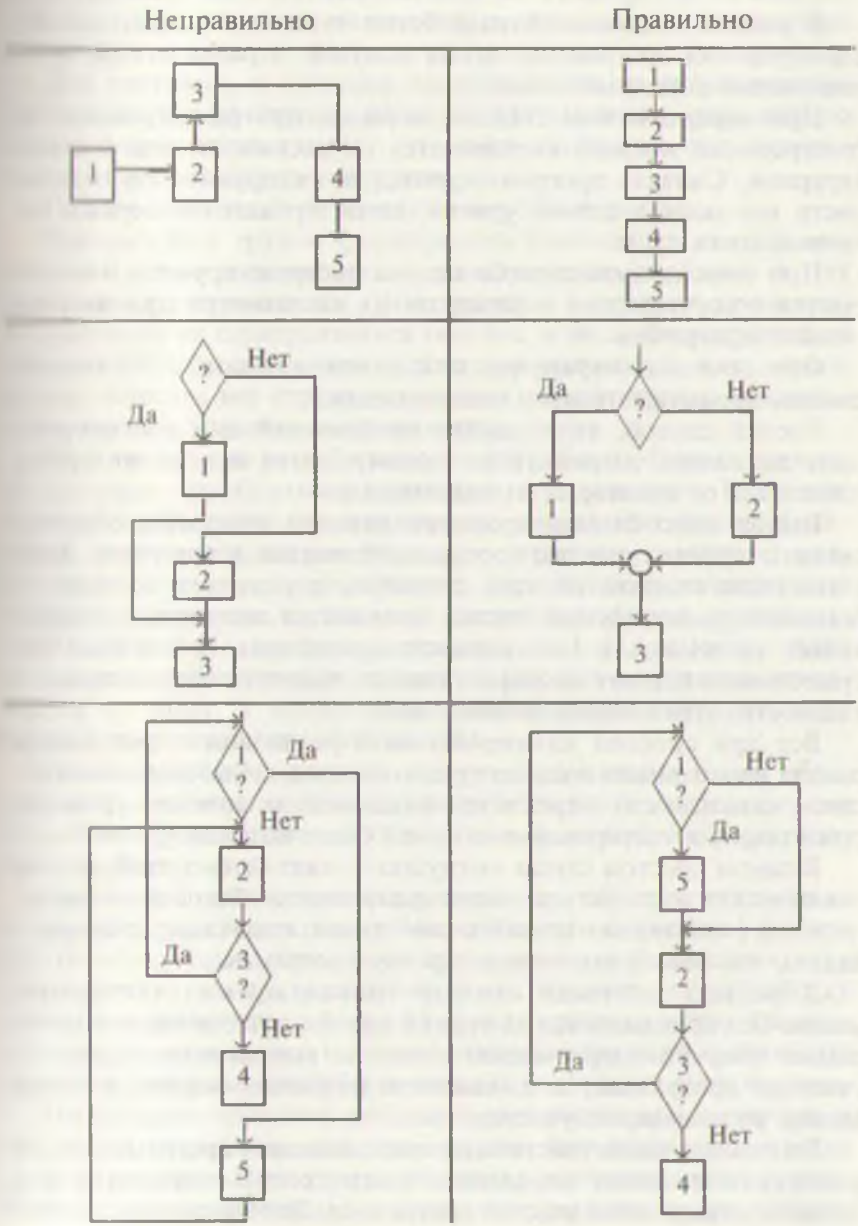
27. Пуск / останов

28. Комментарий 

29. Магистральный соединитель 

30. Источник Приемник данных 

Примеры неправильно и правильно составленных блок-схем



Планирование программирования

В рамках нисходящей разработки существуют три способа планирования программирования модулей: иерархический, операционный и их комбинация.

При *иерархическом* способе порядок программирования и тестирования модулей определяется их расположением в схеме иерархии. Сначала программируются и тестируются по отдельности все модули одного уровня, затем происходит переход на уровень ниже и т. д.

При *операционном* способе модули программируются и тестируются в соответствии с порядком их выполнения при запуске готовой программы.

Оба способа, иерархический и операционный, обладают своими преимуществами и недостатками.

Третий способ, являющийся *комбинацией* двух рассмотренных, позволяет сохранить их преимущества и в то же время избавиться от некоторых их недостатков.

Выбор способа планирования непосредственным образом связан с эффективностью процесса оптимизации программ. Дело в том, что каждый из этих способов определяет последовательность расположения текстов модулей на листинге, т.е. определяет читаемость и понимаемость программы. А эти факторы существенно влияют на эффективность процесса оптимизации, в частности, управляемой оптимизации.

Все три способа планирования и реализации программ в рамках нисходящего подхода предполагают использование «заглушек», заменяющих нереализованные модули нижних уровней, при отладке и тестировании модулей более высоких уровней.

В самом простом случае «заглушка» может быть пустой, но она также может дополняться операторами печати. Такое дополнение позволяет «заглушке» печатать сообщения, отмечающие ее активацию, что обеспечивает трассировку программы.

Дополнив заглушки самыми тривиальными счётчиками, можно без дополнительных усилий уже на этапе создания программы получать информацию о частоте выполнения отдельных участков программы, т. е. выявлять наиболее «узкие», с точки зрения оптимизации, участки.

Выделение таких участков на этапе создания программы до их кодирования может определять язык программирования для каждого конкретного участка программы. Действительно, наибо-

все «узкие», с точки зрения оптимизации, участки, можно запрограммировать на языках, для которых имеются эффективные трансляторы, в частности, можно использовать язык ассемблера.

Для программ, к которым предъявляются не очень жесткие требования по эффективности, это может исключить дальнейшую специальную оптимизацию, что даст экономию как человеческих, так и машинных ресурсов.

Планировать также необходимо основные данные.

Чрезвычайно трудно формировать глобальные данные «на лету» во время кодирования и одновременно сообщать о них остальным программистам, участвующим в программировании, и принуждая их подстраиваться под вас, а вас — подстраиваться под них. Необходимо, насколько это возможно, как можно раньше описать все структуры данных, задействуемые несколькими членами команды одновременно.

Если программа разрабатывается для работы с базами данных, то последнее необходимо тщательно спланировать. В первую очередь необходимо спроектировать основные таблицы (например, для отдела кадров фирмы — анкетные данные ее сотрудников). Затем — сопутствующие, справочные и установочные таблицы (например, постоянные сведения о фирме, номенклатура должностей, тарифные разряды и т.д.). И, конечно, связи между этими таблицами (например, должность сотрудника заполняется по коду, а конкретное наименование согласно коду берется из справочника должностей).

Среда разработки системы должна быть создана таким образом, чтобы все члены команды разработчиков с минимальными затратами времени могли обратиться к любой информации, относящейся к проекту.

Временной успех создания программного обеспечения зависит от удачного распределения (по способностям и умению) фронта работы между членами команды программистов. В зависимости от сложности и размера отдельной ветви проекта, ее разработку поручают одному или небольшой группе программистов. Распределение (планирование) и координирование всей работы по созданию программного обеспечения осуществляет один человек.

Необходимо составить детальное расписание сроков начала и окончания разработки каждого модуля, частей проекта и всего проекта в целом. При этом необходимо учитывать время, затрачиваемое на дополнительные контакты с заказчиком, разработку

специфических инструментальных средств, а также возможные проблемы, связанные с непредвиденными обстоятельствами (например, болезнь сотрудника или частичная потеря данных вследствие сбоев аппаратного обеспечения).

В дальнейшем вся команда разработчиков должна регулярно встречаться для обсуждения текущей работы. На этих встречах будет производиться как обзор всего проекта, так и обзор каждого модуля низкого уровня, а в отдельных случаях — обзор псевдокода, разработанного каждым членом группы.

Проектирование логики модуля

Внутреннее проектирование — один из последних этапов в длинной цепи процесса проектирования программного обеспечения. Оно представляет собой подробное внутреннее конструирование программного продукта, разработку внутренней логики каждого модуля системы, которая затем выражается текстом конкретной программы.

Следующие предложения составляют набросок дисциплинированного подхода к проектированию модуля:

1. Выбор языка программирования (обычно диктуется требованиями контракта или принятыми в организации стандарта).

2. Проектирование внешних спецификаций модуля.

3. Выбор алгоритма и структуры данных. Сегодня лишь немногие алгоритмы создаются впервые, огромное их число уже изобретено.

4. Запись первого и последнего предложения модуля.

В языке Паскаль: PROGRAM ... BEGIN ... END;

5. Объявить все данные и сопряжения.

6. Детализирование текста программы — шаг итеративный.

Он предполагает последовательную детализацию логики модуля, начиная с достаточно высокого уровня абстракции и заканчивая готовым текстом программы.

Используются методы пошаговой детализации и структурного программирования (будут позже).

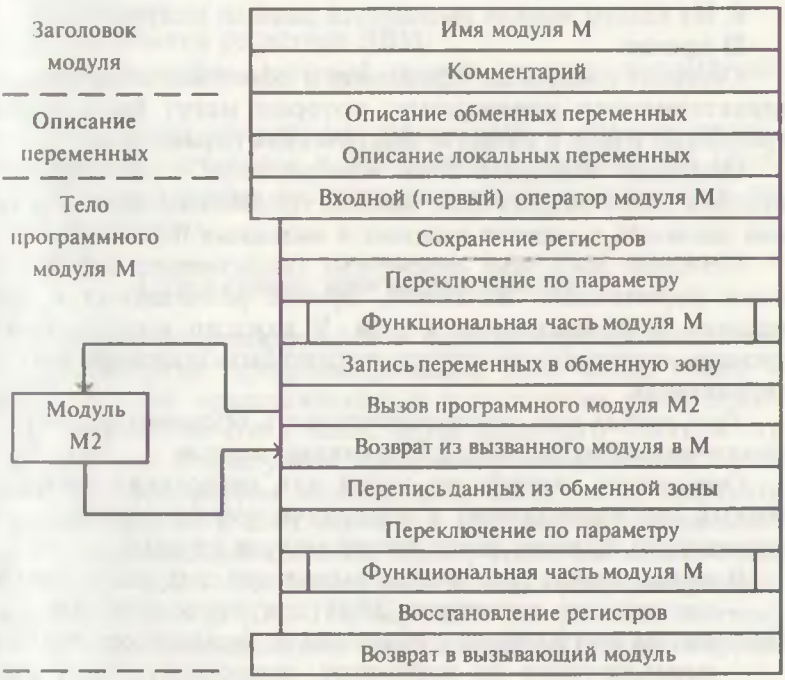
7. Объявить остальные данные. Поскольку трудно предсказать все переменные, которые понадобятся. Это предложение часто перекрывается с предыдущим.

8. Шлифование текста программы — для достижения оптимальности и ясности, со снабжением его дополнительными

комментариями, отвечающими на вопросы, которые могут возникнуть при чтении программы.

9. Проверка правильности программы (ручная проверка).
10. Тестирование.

Типовая структура программного модуля может выглядеть так:



Под *структурой программного модуля* понимается совокупность смысловых частей, образующих модуль и используемых для различных целей при его разработке и исполнении.

Типовая структура модуля в общем случае включает:
- заголовок модуля;
- описание переменных;
- тело модуля.

Заголовок модуля содержит имя модуля, блок комментариев и совокупность формальных параметров, если таковая имеется.

Блок комментариев обычно содержит общую информацию описания программного модуля:

1. Имя модуля.
2. Функция модуля.
3. Список параметров, передаваемых вызывающим модулем.
4. Входные, выходные данные.
5. Внешние эффекты.
6. Из какого модуля вызывается данный модуль.

И прочее.

Описание глобальных переменных и обменной зоны определяет характеристики переменных, которые могут быть переданы программе извне в качестве фактических параметров.

Описание обменной зоны «связывается» с некоторым множеством ячеек оперативной памяти, предназначенных для хранения значений и адресов входных и выходных параметров.

Описание локальных переменных представляет собой список имен переменных, их типов, правил размещения в памяти машины и комментарии к ним. У каждого модуля, транслируемого независимо от других, должно быть описание локальных переменных.

Тело модуля есть последовательность операторов программы, обеспечивающих выполнение функции модуля.

Оно может состоять из одной или нескольких частей, связанных (по управлению) в иерархической последовательности посредством вызовов одних частей модуля другими.

В общем случае тело модуля выполняет следующие функции:

— сохранение регистров ЭВМ для последующего восстановления их при возврате управления от вызываемого его модуля;

— переключение по параметру, задающему точку входа в модуль, если его исполнение может начинаться с некоторого внутреннего оператора;

— выполнение операторов, реализующих функциональную задачу модуля;

— запись переменных в обменную зону вызываемого модуля, если модуль вызывает другой с передачей ему параметров;

— передачу управления на начало вызываемого модуля с запоминанием возврата в вызывающий модуль в точку, следующую за вызовом;

— --- выполнение вызванного модуля;

— --- возврат управления в данный модуль из вызванного модуля;

— перепись результатов исполнения вызванного модуля из обменной зоны в локальную зону рассматриваемого модуля или в глобальную зону;

— переключение по параметру, задающему точку возврата в вызывающий модуль, если возврат должен осуществляться к оператору, непосредственно не следующему за вызовом;

— выполнение операторов, реализующих функциональную задачу программы;

— восстановление регистров ЭВМ;

— возврат в модуль, который вызвал рассматриваемый модуль.

При разработке конкретных программных средств приведенные правила уточняются и оформляются специальной инструкцией по структурному построению комплекса программ и его основных компонент.

Структурное программирование

Структурное программирование — одно из крупнейших достижений в технологии программирования. Хотя самое общее и довольно смутное представление о структурном программировании имеется почти у всех, общепринятого четкого его определения нет. Структурное программирование ставит своей целью писать программы минимальной сложности, заставить программиста мыслить ясно, облегчить восприятие программы.

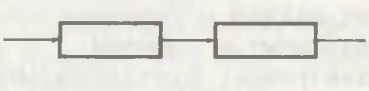
Текст программы должен быть таким, чтобы его можно было читать «сверху - вниз». Неограниченное использование операторов безусловного перехода (GO TO) нарушает это условие, поэтому структурное программирование часто называют программированием без GO TO.

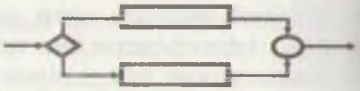
Можно привести примеры программ, которые не содержат GO TO, и аккуратно расположены лесенкой в соответствии с уровнем вложенности операторов, но совершенно непонятны и бывают другие программы, содержащие GO TO и все же понятны. Так что наличие или отсутствие GO TO — плохой показатель качества программы (иллюстрации в Д. Кнут). И все же: наиболее трудно контролируемым и потенциально неустойчивым является оператор безусловного перехода — GO TO.

Структура тела модулей и используемые базовые конструкции программирования должны быть потенциально устойчивыми к аппаратурным сбоям, искажениям исходных данных и к ошибкам в программах.

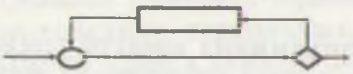
Любую программу можно синтезировать на основе элементарных базовых конструкций трех типов:

1. Простая последовательность. 2. Условия (альтернативы).

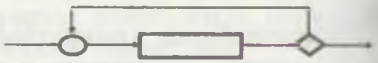




3. Повторение (циклы, итерации). Возможны один из двух или оба вида:



Делать «пока»



Делать «пока не»

4. Можно добавить четвертую конструкцию — выбор (переключатель).

Блоки определяются рекурсивно: прямоугольники изображают вложенные строительные блоки, используемые вместо прямоугольника.

Перечисленные элементарные конструкции, например, в языке Паскаль реализованы так:

Условный оператор IF *лог. выр.* THEN *оператор1* ELSE *оператор2*;
IF *лог. выражение* THEN *оператор*;

Делать «пока» WHILE *условие продолжения* DO *оператор*;

Делать «пока не» REPEAT *оператор* UNTIL *условие завершения*;

Цикл с перебором FOR *K:=B1* TO *B2* DO *оператор*;
FOR *K:=B2* DOWNTO *B1* DO *оператор*;

Выбор (переключатель)

```
CASE условие OF
        N1, ... Nk: оператор1;
        .....
        .....
        Ni, ... Nm: оператор n;
END;
```

В языке FoxBASE эти конструкции реализованы в виде:

Условный оператор

```
IF лог. выражение
    операторы1
[ ELSE
    операторы2 ]
ENDIF
```

Цикл

```
DO WHILE выражение
    операторы
ENDDO
```

Выбор (переключатель)

```
DO CASE
    CASE лог.выражение1
        операторы
    CASE лог. выражение2
        операторы
    CASE лог.выражение n
        операторы
    [OTHERWISE
        операторы]
ENDCASE
```

Каждая структура характеризуется единственной точкой передачи управления в структуру (единственный вход) и единственной точкой выхода из структуры.

Эти конструкции имеют систематизирующее и дисциплинирующее значение. Простота исходных конструкций структурного программирования предотвращает появление сложных информационных связей и запутанных передач управления.

При повышении структурированности модулей снижается сложность программ, возрастает их наглядность, что способствует сокращению числа ошибок. Однако за повышение качества программ приходится расплачиваться дополнительной памятью и временем их реализации на ЭВМ.

Структурность программы зависит от используемого языка программирования. Современные программные средства разработки программ являются «наилучшими» языками структурного программирования. Из распространенных языков программирования самыми подходящими считаются Паскаль, Basic, FoxBASE. Структурное программирование, например, на языке Ассемблер почти невозможно. Сам факт использования языка

Ассемблера указывает на то, что программа написана в основном в терминах машинного языка.

Структурное программирование ориентировано на общение с людьми, а не с машиной, способствует написанию программ, представляющих собой простое и ясное решение задачи.

Важно, чтобы программист, еще только приступая к программированию логики, мыслил в терминах основных базовых структурных конструкций.

Позиция в отношении оператора GO TO должна быть следующей: избегать использование GO TO всюду, где это возможно, но не ценой ясности программ. Часто оказывается целесообразным использовать GO TO для выхода из цикла или модуля, для перехода в конструкции ON (например, в языке Basic) или во избежание слишком большой глубины вложенности развилок, тем более что переход осуществляется на последующие (расположенные ниже) операторы программы, и структурная программа продолжает оставаться легко читаемой сверху вниз. Наихудшим применением оператора GO TO считается передача управления на оператор, расположенный выше (раньше) в тексте программы.

Кроме того, чтобы облегчить чтение программы, ее текст необходимо разбить физически на части, добавляя пустые строки между разделами, подразделениями. Текст программы должен быть написан с правильными сдвигами, так что разрывы в последовательности выполнения легко прослеживаются. Одноцелевые выполняемые предложения каждого модуля должны умещаться на одной странице печатающего устройства.

Пошаговая детализация (программирование сверху вниз или нисходящая разработка)

Пошаговая детализация представляет собой простой процесс, предполагающий первоначальное выражение логики модуля в терминах гипотетического (условного) языка очень высокого уровня с последующей детализацией каждого предложения в терминах языка более низкого уровня, до тех пор, пока, наконец, не будет достигнут уровень используемого языка программирования. Здесь уместно напомнить: чем меньше язык содержит деталей, тем более он высокого уровня. Можно считать языком самого высокого уровня обычную человеческую речь, а языком низкого уровня — машинный язык.

На протяжении всего процесса пошаговой детализации логика процесса выражается основными конструкциями структурного программирования.

Достоинство пошаговой детализации состоит в том, что она позволяет проектировщику упорядочить свои рассуждения. На каждом шаге мы имеем дело с элементарной задачей.

Рассмотрим этот метод на конкретном примере.

ЗАДАЧА. Дана матрица размером 10*10 элементов. Для каждого столбца среди элементов, лежащих выше первого нулевого, и значения которых лежат в интервале [c, d], найти наименьший и наибольший элементы и их номера в строке. Если нулевого элемента в столбце нет, то обрабатывается весь столбец.

План решения задачи

1. Вход — выход.
2. Основной алгоритм (цикл по столбцам).
3. Обработка столбца (внутренний цикл).
4. Обработка элементов матрицы.
5. Поиск наибольшего и наименьшего элементов в столбце.
6. Обработка начальных и конечных операторов циклов.
7. Оптимизация и шлифовка программы.

Ввод - вывод

A(10, 10)— исходная матрица.

C и D — границы интервала.

max(10) и min(10) — массивы, содержащие наибольшие и наименьшие значения каждого столбца исходной матрицы.

Imax(10) и Imin(10) — массивы номеров строк, в которых встречаются найденные, соответственно, наибольшие и наименьшие значения в столбце.

а) *Первый шаг.* *Детализация ввода-вывода.*

```
PROGRAMM PRIMER;
VAR
   A: ARRAY [1..10,1..10] OF REAL;        (* Исходная матрица *)
   C,D: REAL;                             (* Границы интервала *)
   I,J: INTEGER;                          (* Номера строк и столбцов *)
   MAX, MIN: ARRAY [1..10] OF REAL;       (* Значения наибольших и *)
                                          (* наименьших элементов *)
   IMAX, IMIN: ARRAY [1..10] OF INTEGER;  (* и их номера строк *)
   . . . . . . . . . . . . . . . . . . .
   : Другие переменные :
   . . . . . . . . . . . . . . . . . . .
BEGIN
   WRITELN ('Введите элементы матрицы');
   FOR I:=1 TO 10 DO BEGIN
         FOR J:=1 TO 10 DO READ (A[I,J]); WRITELN;
   END;
   WRITE ('Введите границы интервала'); READ (C, D);
   . . . . . . . . . . . . . . . . . . .
   : Основной алгоритм :
   . . . . . . . . . . . . . . . . . . .
   FOR I:=1 TO 10 DO WRITELN ('MIN=', MIN[I], ',его номер', IMIN[I],
                              '; MAX=', MAX[I], '; его номер', IMAX[I]);
END.
```

б) *Второй шаг.* *Детализация основного алгоритма.*

Необходимо выполнить одно и то же для каждого столбца.

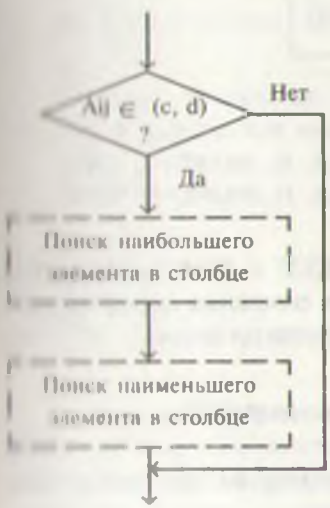

```
. . . . . . . . . . . . . . . . . . .
: Начальные операторы :
. . . . . . . . . . . . . . . . . . .
FOR J:=1 TO 10 DO
BEGIN
   . . . . . . . . . . . . . . . . . . .
   : Обработка столбца J :
   . . . . . . . . . . . . . . . . . . .
END;
. . . . . . . . . . . . . . . . . . .
: Конечные операторы :
. . . . . . . . . . . . . . . . . . .
```

в) *Третий шаг. Обработка столбца.*

В этом столбце необходимо обработать элементы, лежащие выше первого нулевого элемента матрицы A.

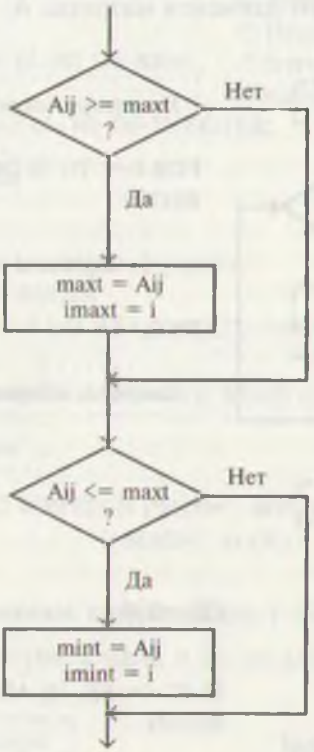

```
. . . . . .
: Начальные операторы :
. . . . . .

FOR I:=1 TO 10 DO WHILE A[I,J] # 0 DO
BEGIN
    . . . . . .
    : Обработка элементов матрицы :
    . . . . . .
END;
. . . . . .
: Конечные операторы :
. . . . . .
```

г) *Четвертый шаг.* *Обработка элементов матрицы A.*




```
IF (C <= A[I, J]) AND (A[I, J] <= D) THEN
BEGIN
    . . . . . .
    : Поиск наибольшего :
    : элемента в столбце :
    . . . . . .
    . . . . . .
    : Поиск наименьшего :
    : элемента в столбце :
    . . . . . .
END;
```

д) *Пятый шаг. Поиск наибольшего и наименьшего элементов в столбце матрицы A и их номера в строке.*




```
IF A[I, J]>= MAXT THEN BEGIN  MAXT:= A[I, J]; IMAXT:= I; END;
IF A[I, J]<= MINT THEN BEGIN  MINT:= A[I, J]; IMINT:= I; END;
```

Появились новые переменные: MAXT и MINT, IMAXT и IMINT, которые необходимо включить в описания переменных с присвоением типа REAL и INTTGTR соответственно.

е) *Шестой шаг. Начальные и конечные операторы.*

Движемся изнутри циклов наружу и смотрим, что необходимо для работы циклов.

Для срабатывания внутреннего цикла надо, чтобы на первом шаге MAXT и MINT имели какие-то значения и что делать в случае отсутствия в столбце элементов из интервала (c, d)?

Примем, что индексы в этом случае равны нулю, а значения наибольших и наименьших элементов, попадающие в выходные массивы, несущественны.

Итак, вначале MAXT: = C; MINT:= D;

IMAXT: = 0; IMINT: = 0;

Результаты этого цикла надо заслать в J-е элементы результирующих массивов, поэтому конечные операторы будут такими:

MAX[J]:= MAXT; MIN[J]:= MINT;

IMAX[J]:= IMAXT; IMIN[J]:= IMINT;

Для цикла по столбцам никаких начальных и конечных операторов не требуется.

■) *Последний шаг. Шлифование и оптимизация программы.*

Получив программу, можно заняться ее улучшением, чтобы она стала короче или выполнялась быстрее.

Например, если

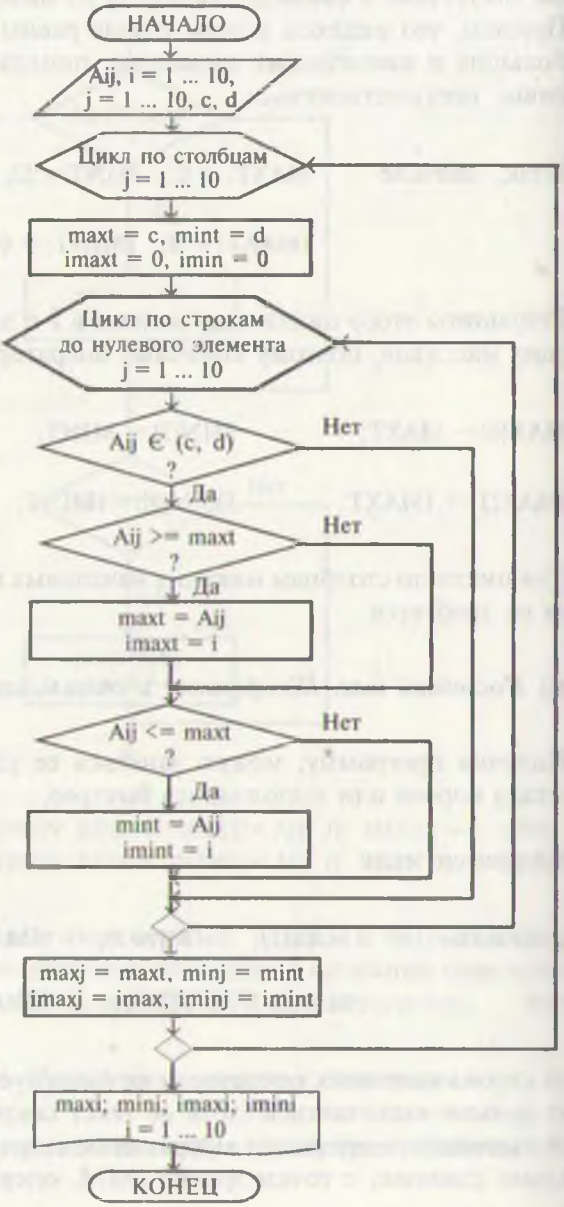
использовать MAX(J), MIN(J), IMAX(J), IMIN(J)

вместо MAXT, MINT, IMAXT, IMINT,

то строка конечных операторов не потребуется, но программа будет дольше выполняться (хотя ее текст сократится), т.к. участится выполнение операции «Обращение к элементу массива» — довольно длинная, с точки зрения ЭВМ, операция.

Можно выполнить другие оптимизирующие действия.

Результирующий алгоритм



Результирующая программа

```
PROGRAMM PRIMER;
VAR
    A: ARRAY [1..10, 1..10] OF REAL;          (* Исходная матрица *)
    C, D: REAL;                               (* Границы интервала *)
    I, J: INTEGER;          (* Номера строк и столбцов матрицы A *)
    MAX, MIN: ARRAY[1..10] OF REAL; (* Значения наибольших и *)
                                              (* наименьших элементов *)
    IMAX, IMIN: ARRAY [1..10] OF INTEGER; (* и их номера строк *)
    MAXT, MINT: REAL;  (* Временные переменные, наибольшее и *)
                              (* наименьшее значения элементов *)
    IMAXT, IMINT: INTEGER;                    (* и их номера в столбце *)
BEGIN
    WRITELN ('Введите элементы матрицы:');
    FOR I:=1 TO 10 DO
    BEGIN
        FOR J:= 1 TO 10 DO READ (A[I, J]);          WRITELN;
    END;
    WRITE ('Введите границы интервала');       READLN (C, D);
    FOR J:=1 TO 10 DO          (* Обработка столбцов матрицы A *)
    BEGIN IMAXT:= 0; IMINT:= 0; MINT:= C; MINT:= D;
        (* Обработка элементов столбца матрицы A *)
        FOR I:=1 TO 10 DO WHILE A[I,J] # 0 DO
        BEGIN
            IF (C<=A[I, J]) AND (A[I, J]< = D) THEN
            (* Элемент матрицы принадлежит отрезку CD? *)
            BEGIN
                IF A[I, J]>= MAXT  (* Элемент матрицы наибольший? *)
                    THEN BEGIN MAXT:=A[I, J]; IMAXT:= I; END;
                IF [I, J]<= MINT    (* Элемент матрицы наименьший? *)
                    THEN BEGIN MINT:=A[I, J]; IMINT:= I; END;
            END;
        END;
        MAX[J]:= MAXT;          MIN[J]:= MINT;
        IMAX[J]:= IMAXT;        IMIN[J]:= IMINT;
    END;
    FOR I:=1 TO 10 DO WRITELN ('MIN=', MIN[I],' ,его номер',IMIN[I],
                              '; MAX=', MAX[I], ', его номер', IMAX[I]);
END.
```

Система автоматизации программирования

Система автоматизации программирования (САП) представляет собой совокупность программных и аппаратных средств, предназначенных для автоматизации одного из наиболее важных этапов разработки — этапа программирования, т.е. перевода исходных алгоритмов автоматизированного управления на машинный язык, используемый в конкретной ЭВМ. Она существенно сокращает время изготовления программы.

В общем случае эта система состоит из одного или нескольких входных языков, систем трансляции программ с этих языков и компоновки программ, баз данных проектирования и системы выпуска технической документации на программные средства.

Система автоматизации программирования выполняет следующие функции:

— синтаксический и семантический контроль правильности записи программ на входных языках и выдачу информации о наличии, месте и характере ошибок;

— формирование структуры общего распределения памяти ЭВМ и описание глобальных переменных;

— трансляцию отдельных модулей комплекса программ, записанных на одном из входных языков, в объектные коды команд машины;

— компоновка оттранслированных программ по передаче управления, по глобальным переменным, а также по использованию общих зон памяти ЭВМ в единую исполняемую программу;

— накопление в базе данных проектирования результатов трансляции модулей для их последующей комплексной отладки и загрузки в память управляющей ЭВМ;

— автоматизированный выпуск технической документации на программные средства и ее корректировку.

Требования к САП зависят от объема и сложности разрабатываемого программного обеспечения, имеющихся ресурсов для его создания, и ряда других конструктивных и организационных факторов.

Требования к САП состоят в следующем:

— снижение общей трудоемкости и длительности создания программ;

— повышение производительности труда программистов;

— обеспечение высокого качества и надежности функционирования создаваемых программ;

— обеспечение унифицированной технологии разработки программ для реализующих их ЭВМ широкого класса;

— использование новых технологий программирования;

— обеспечение эффективного использования ресурсов памяти и производительности реализующих ЭВМ.

Документ «Внутренняя спецификация»

Документ «Внутренняя спецификация» — предполагает наличие документов «Соглашение о требованиях» и «Внешняя спецификация». При этом предполагается также, что вся информация о том, что представляет собой программное изделие содержится в вышеназванных спецификациях.

Внутренняя спецификация должна объяснять, как изделие устроено и как достигаются установленные для него цели и требования. От внутренней спецификации не требуется, чтобы она обеспечивала полное понимание пользователем работы программного обеспечения без обращения к тексту программ. Однако совместно с текстами программ внутренняя спецификации должна представлять пользователю информацию в полном объеме.

Внутренние спецификации являются своего рода записной книжкой проекта, заполняющейся по мере разработки программного изделия, которое она описывает. У нас этот документ называется «Рабочим проектом». Внутренняя спецификация описывается и утверждается еще до начала фазы программирования, так что руководство группы разработки имеет гарантию того, что проект тщательно продуман. Во внутренние спецификации всегда включаются описания всех используемых алгоритмов.

Программирование начинается лишь тогда, когда имеется достаточно полное описание данных и алгоритма их обработки, так что частичное тестирование программного обеспечения приобретает смысл, а блок-схема будущего изделия отработана настолько, что можно гарантировать корректную обработку нетривиальных исходных данных и получение нетривиальных результатов. С этого момента внутренняя спецификация и машинные программы разрабатываются совместно и могут снабжаться ссылками друг на друга для обеспечения полноты описания внутренней структуры программного обеспечения.

В процессе создания программ желательно включать в ли листинги как можно больше информации, относящейся к документации внутреннего проектирования, чтобы внутренняя спецификация не содержала избыточных сведений. При этом ин вынесенные в листинги сведения в документе «Внутренняя спецификация» заменяются соответствующими ссылками.

Внутренняя спецификация имеет много назначений.

Начинается она как рабочий журнал проекта, который показывает, как создается программное изделие, каковы его функции и что представляют собой алгоритмы, используемые для реализации этих функций. Без такого рабочего журнала мало надежды на сохранение информации о сложной внутренней структуре программного обеспечения при уходе основного персонала, занимающегося проектом, или при отсрочке работы над проектом даже на месяц.

По мере развития проекта и его внутренней спецификации последняя может использоваться как субъективный индикатор текущего состояния проекта.

Позже внутренняя спецификация включается в спецификацию сопровождения, которая представляет собой сочетание внутренней спецификации и листингов программ.

Наконец, внутренняя спецификация является широко используемым справочным документом, содержащим описания идей проекта, алгоритмов и даже модулей, которые могут быть полезны в последующих разработках.

РЕЗЮМЕ

* Современное программирование ставит своей целью разработку программного обеспечения на поточной линии. Уже имеется некоторый опыт автоматизации производства программирования с привлечением малоквалифицированных специалистов.

* Искусство программирования пока остается настолько сложным, что оно еще нуждается в высококвалифицированных специалистах-программистах.

* На этапе программирования выполняется работа, связанная со сборкой программного изделия. Она состоит в подробном внутреннем конструировании программного продукта.

* Система автоматизации программирования (САП) представляет собой совокупность программных и аппаратных средств, предназначенных для автоматизации. Она существенно сокращает время изготовления программы.

* Алгоритм — точное предписание, которое задает процесс (называющийся алгоритмическим), начинающийся с произвольных исходных данных и направленный на получение полностью определяемого этими исходными данными результата.

* Алгоритмический процесс представляет собой этапы (шаги) последовательного преобразования исходных данных в результирующие.

* Представленный алгоритм пишется на естественном (разговорном) языке, с использованием терминов отрасли, в которой решается задача и не должен содержать программистские термины; тем более алгоритм не может содержать операторы языка программирования.

* Считается, что по представленному алгоритму можно написать программу практически на любом языке программирования.

* Текст алгоритма должен быть читаемым без дополнительных пояснений автора.

* В алгоритме не показывают описания, т.к. описания — это особенности языка программирования, а не алгоритма.

* Алгоритм не может быть «привязан» к конкретному языку программирования. В нем описываются конкретные действия — *что* делается, а не *как* это делается.

* В рамках нисходящей разработки существуют три способа планирования программирования модулей: иерархический, операционный и их комбинация.

* При иерархическом способе порядок программирования и тестирования модулей определяется их расположением в схеме иерархии. Сначала программируются и тестируются по отдельности все модули одного уровня, затем происходит переход на уровень ниже и т. д.

* При операционном способе модули программируются и тестируются в соответствии с порядком их выполнения при запуске готовой программы.

* Комбинированный способ позволяет сохранить преимущества и в то же время избавиться от некоторых недостатков первых двух.

* Планировать необходимо также основные данные. Необходимо как можно раньше описать все структуры данных, задействуемые несколькими членами команды одновременно.

* В зависимости от сложности и размера отдельной ветви проекта, ее разработку поручают одному или небольшой группе программистов. Распределение (планирование) и координиро-

вание всей работы по созданию программного обеспечения осуществляет один человек.

* Необходимо составить детальное расписание сроков начала и окончания разработки каждого модуля, частей проекта и всего проекта в целом.

* Типовая структура модуля в общем случае включает: заголовок модуля,описание переменных, тело модуля.

* Заголовок модуля содержит имя модуля, блок комментариев и совокупность формальных параметров, если таковая имеется.

* Описание переменных представляет собой список имен переменных, их типов, правил размещения в памяти машины и комментарии к ним. У каждого модуля, транслируемого независимо от других, должно быть описание локальных переменных.

* Тело модуля есть последовательность операторов программы, обеспечивающих выполнение функции модуля. Оно может состоять из одной или нескольких частей, связанных (по управлению) в иерархической последовательности посредством вызовов одних частей модуля другими.

* Структурное программирование ставит своей целью писать программы минимальной сложности на основе элементарных базовых конструкций: последовательности, условия, повторения (циклы) и выбора. Базовые конструкции обеспечивают получение программы, текст которой читается «сверху вниз», что значительно облегчает ее чтение и понимание.

* Пошаговая детализация представляет собой процесс, предполагающий первоначальное выражение логики модуля в терминах условного языка очень высокого уровня с последующей детализацией каждого предложения в терминах языка более низкого уровня, до тех пор, пока, наконец, не будет достигнут уровень используемого языка программирования.

Темы для повторения

1. Внутреннее проектирование ПО — программирование.
2. Система автоматизации программирования (САП).
3. Алгоритм.
4. Основные требования к блок-схеме. Виды блоков в схемах алгоритмов.
5. Планирование программирования.
6. Проектирование логики программного модуля.
7. Типовая структура программного модуля.
8. Структурное программирование.
9. Пошаговая разработка программ (пошаговая детализация).
10. Документ «Внутренняя спецификация».

Глава 6. Стиль программирования

Под *стилем программирования* понимается набор приемов или методов программирования, которые используют опытные программисты, чтобы получить правильные, надежные, эффективные, удобные для применения и легко читаемые программы.

Правила хорошего стиля программирования — это результат соглашения между опытными программистами (маленький стандарт). Если бы все программисты придерживались своего индивидуального стиля, то результатом было бы Вавилонское столпотворение.

Программы должны составляться таким образом, чтобы их могли прочитать в первую очередь люди, а не машины. Программа — это документ для последующего использования, учебный материал по кодированию алгоритмов и средство для дальнейшей разработки более совершенных программ.

Слишком часто, стремясь побыстрее получить работающую программу, забывают о ее удобочитаемости. Трудночитаемые программы обычно сложно модифицировать. Особенно, если это приходится делать не автору программы. Как правило, к разработке программы приступают со скромными целями, а в дальнейшем расширяют ее возможности. Легко читаемая программа свидетельствует, что ее автор хорошо знал свое дело.

Программа должна передавать логику и структуру алгоритма настолько просто, насколько это возможно. Следует избегать всевозможных программистских трюков, т.к. чем их больше, тем труднее будет разобраться в логике программы самому автору, а кто-либо другой это вообще не сможет сделать.

Программу кодируют просто и рационально. На ранних этапах разработки сложной программы лучше без колебания переписать заново ее громоздкие блоки, если это ведет к упрощению.

Если программы составляют для какой-либо организации, то применение согласованного стиля поможет сделать их достоянием этой организации, а не личной собственностью отдельного программиста.

Малый программистский стандарт

Даже элементарную задачу по программированию можно решить настолько различными способами, что с первого взгляда трудно понять, что обе программы делают одно и то же.

Каждый программист обычно использует свои любимые приемы, и когда он им изменяет, то часто именно в этом месте он делает ошибку. Поэтому каждому программисту можно порекомендовать выработать для себя некий «*малый программистский стандарт*» и не отклоняться от него без весьма веских оснований.

Еще лучше, если этот стандарт обязателен к применению в данной организации: это облегчит взаимопонимание программистов. То же замечание относится к системе идентификации. «Малый стандарт» очень помогает, когда надо вспомнить старую программу.

Общая организация программы и ее запись

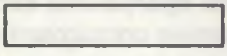
Так же как разделение большого произведения на главы и параграфы облегчает его чтение, так и разбиение большой программы на параграфы, разделы (подпрограммы и модули), путем выделения логических единиц улучшает ее восприятие, помогает избежать однообразия и хорошо организовать материал. Название раздела отражает его цель.

Структура программы хорошо реализуется с некоторым смещением. Для ЭВМ форма записи программы безразлична: *смещение строк предназначено для удобства чтения программы человеком.* Смещение строк не должно быть большим: двух-трех позиций вполне достаточно, чтобы выделить структуру программы.

Перечислим основные правила записи структурных конструкций на примере языка Pascal.

1. *Короткий составной оператор*: BEGIN A1; A2; ...; An END;

2. *Длинный составной оператор*: BEGIN



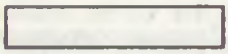
END;

4. *Короткое ветвление и обход*: IF e THEN A1 ELSE A2;
IF e THEN A3;

5. *Более длинное ветвление*: IF e THEN A1
ELSE A2;

6. *Очень длинное ветвление*: IF e
THEN BEGIN

END
ELSE BEGIN



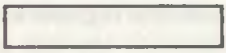
END;

7. *Циклы короткие*: WHILE e DO a;
REPEAT a UNTIL e;
FOR k:= b1 TO b2 DO a;
FOR k:= b2 DOWN TO b1 DO a;

8. *Циклы длинные*: а) WHILE e DO
BEGIN

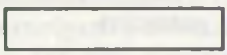
END;

б) REPEAT



UNTIL e;

в) FOR k:= b1 TO b2 DO
BEGIN



END;

8. *Циклы с общим началом или концом*:

```
FOR k:= b1 TO b2 DO WHILE e DO
BEGIN
  [          ]
END;
```

9. *Выбор*:

```
CASE e OF
    N1, N2, ..., Nk: A1;
    .....
    Ne, e, ..., Nv: An1;
END;
```

10. *Длинные операторы.*

Если оператор не помещается в строку, то его продолжение в следующей строке следует записывать со смещением, причем небезразлично, в каком месте сделан перенос. Например:

```
WRITELN ('первый корень уравнения равен'
,X1,'второй корень уравнения равен', X2);
```

читается хуже, чем

```
WRITELN ('первый корень уравнения равен', X1,
         'второй корень уравнения равен', X2);
```

11. *Объединение операторов в строку.*

Если язык программирования позволяет размещать несколько операторов в одной строке, то в строку следует группировать логически связанные операторы, и так, чтобы их можно было кратко откомментировать.

Вместо

```
I:= 0;
X:= RO*Cos(FI);
У:= RO*Sin(FI);
```

можно написать

```
I:= 0;
X:= RO*Cos(FI);   У:= RO*Sin(FI);   (* Координаты вектора *)
```

Или вместо

```
WRITE ('Введите коэффициенты:');
READLN (A, B, C);
```

можно написать

```
WRITE ('Введите коэффициенты:');  READLN (A,B,C);
```

Увеличив интервалы между некоторыми операторами (добавив не меньше четырех пробелов), можно дополнительно улучшить восприятие программы.

12. Все метки оператора должны быть вынесены в начало строки и должны выступать из общего текста программы влево и в порядке возрастания, чтобы их можно было легко найти.

13. К началу строк выносятся заголовки блоков, начала крупных циклов и очень крупных составных операторов.

14. Операторы END должны размещаться либо в той же строке, что и соответствующий ему DO или BEGIN, либо под парным ему DO или BEGIN.

15. Использование достаточного количества пробелов, отступов, пустых строк в тексте программы является мощным средством, позволяющим сделать программу более выразительной.

16. Одноцелевые выполняемые предложения необходимо умещать на одной странице печатающего устройства.

Комментарии

Для удобства чтения программа должна быть хорошо откомментирована. Если операторы языка программирования позволяют понять как работает программа, то комментарии должны пояснить что и зачем делают эти операторы.

Иногда утверждают, что комментарии будут вставлены позже. Но через удивительно короткое время авторы программы обнаруживают, что забыли ее многие детали. Программы с пояснительными комментариями значительно легче отлаживать.

Комментарии нужны как на стадиях проектирования и отладки программы, так и позже. Они должны содержать некоторую дополнительную информацию, а не перефразировать программу.

Для структурных программ обычно требуется меньше комментариев, чем для неструктурированных, так как программы первого вида понятнее и в них меньше переходов.

Каждая программа, подпрограмма или процедура (модуль)

должна начинаться с блока комментариев (вводных комментариев), поясняющих, что она делает, и может включать ниже перечисленные сведения.

1. Имя программы.
2. Назначение программы. Функции, которые она выполняет.
3. Указания по вызову программы и ее использованию.
4. Список и назначение основных переменных.
5. Ожидаемые входные и выходные данные.
6. Список вызываемых подпрограмм и модулей.
7. Необычные условия или аварийные ситуации, которые могут возникнуть при выполнении программы.
8. Сведения о времени и месте выполнения программы.
9. Требуемый объем памяти, другие ограничения и условия использования.
10. Специальные указания оператору, если это необходимо.
11. Сведения об авторе.

Следующие рекомендации способствуют эффективному документированию внутри программы.

1. Комментарии следует писать в таких местах программы, которые необходимо кому-то пояснить.

2. Комментарии должны быть правильными с самого начала и изменяться в соответствии с изменениями программы. Неправильные комментарии хуже их отсутствия, так как они вводят в заблуждение читателя программы.

3. Комментарии должны содержать некоторую дополнительную информацию, а не перефразировать программу.

4. Комментариев должно быть ровно столько, сколько необходимо. Лишние комментарии — это мусор в программе.

5. В каждом отдельном операторе языка редко проявляются какие-либо действия, и поэтому нет необходимости описывать каждый оператор отдельным комментарием.

6. Комментарии к переменным должны нести информацию об их физических свойствах или функциональном назначении.

7. Комментарии удобно использовать для разделения программы на отдельные подразделы. Такие подразделы содержат обычно от 5 до 10 операторов и имеют одноцелевое назначение.

8. Идентификаторы должны быть мнемоничны, что существенно дополняет комментарии. Необходимо выбирать осмысленные имена. X=X+A дает гораздо меньше информации, чем SUMMA=SUMMA+A.

9. В качестве имен переменных должны употребляться тер-

имена, используемые в данной области. Необходимо избегать имен по виду имен, их неестественных написаний и подобных по написанию символов (X10 и X10). Числа лучше писать в конце имени.

10. При составлении имени переменной можно следовать определенным соглашениям. Например, переменные целого типа могут начинаться с букв i, j, k, l, m, n (обычно и общепринято).

11. Для сокращения идентификаторов можно использовать правила, разработанные Микаэлом Джексоном:

а) каждое значащее слово в имени может быть сокращено, но не более трех;

б) в аббревиатуру всегда должны включаться начальные буквы слов;

в) согласные важнее гласных;

г) начало слова важнее его конца;

д) аббревиатура должна включать в себя от 6 до 15 букв.

12. Все выдаваемые программой печати должны быть откомментированы и иметь естественный формат (матрица не печатается в строку и т.д.).

13. Необходимо придерживаться очевидного порядка действия. Например, если надо вычислить

$$X1,2 = \frac{-в \pm \sqrt{(в^2 - 4ac)}}{2a}$$, то

```
DISKRM:= SQR(B)-4*A*C ;
IF DISKRM > 0 THEN
BEGIN
  X[1]:= (-B+SQRT(DISKRM)) / (2*A) ;
  X[2]:= (-B-SQRT(DISKRM)) / (2*A) ;
END;
```

Намного понятней, чем

```
  DISKRM: = B*B-4*A*C
  IF DISKRM > 0 THEN J: =1 ELSE GO TO 2 ;
1: I:= 2-(J+1)/2 ;
  X[I]:= (-B+J*EXP(0.5*LN(X))) / (2*A) ;
  J:= --J
  IF J < 0 THEN GO TO 1 ;
2: - - - - - - -
```

Корректность программ

Понятие «корректность» или «правильность» подразумевает соответствие проверяемого объекта некоторому эталонному объекту или совокупности формализованных эталонных характеристик и правил.

Корректность или правильность программы наиболее полно определяется степенью ее соответствия предъявляемым к ней формальным требованиям программной спецификации.

При отсутствии полностью формализованной спецификации требований в качестве эталона, которому должны соответствовать программа и результаты ее функционирования, иногда используются неформализованные представления разработчика, пользователя или заказчика программ.

Однако понятие корректности программ по отношению к запросам пользователя или заказчика сопряжено с неопределенностью самого эталона, которому должна соответствовать программа. Вследствие этого понятие корректности программ становится субъективным и его невозможно определить количественно.

Неопределенность программных спецификаций уменьшается в процессе разработки программ путем уточнения требований по согласованию между разработчиком и заказчиком.

Для сложных комплексов программ всегда существует риск обнаружить их некорректность по мнению пользователя или заказчика относительно имеющихся спецификаций вследствие неточности самих спецификаций.

Формальные правила проектирования программ устанавливаются стандартами и инструкциями подготовки текстов программ и их структурного построения. Эталоны этого вида включают описание языка программирования, правила оформления текстов программ и описания данных, они являются наиболее универсальными и в ряде случаев формализуются на уровне ГОСТов на языки программирования и базу данных.

Будем считать программное изделие правильным, если оно:

— решает действительно ту задачу, для которой оно было разработано;

— не «зависает» и не заканчивает свою работу аварийно;

— удовлетворяет всем требованиям из документа «Соглашение о требованиях» («Техническое задание») с учетом их уточнений в процессе проектирования программного изделия;

— разработано в соответствии с формальными правилами проектирования программного обеспечения.

Эффективность программы

Основной задачей программирования является создание правильных (не обязательно при этом эффективных) программ. Эффективная программа не нужна, если она не обеспечивает правильных результатов.

Правильность программы не является дополнительной характеристикой программы в отличие от эффективности.

Правильную, но не эффективную программу можно оптимизировать и сделать ее более эффективной. Поэтому оптимизацию программы можно назвать вторым этапом в программировании, а первый этап — это получение правильной программы. Причем, рассмотреть возможность оптимизации программы следует лишь в том случае, если готовая программа пригодна и нужна для работы, если ее будут выполнять многократно и если статус проекта и фирмы это позволяет.

Требования к эффективности программы обычно определяют на стадии проектирования.

Эффективность или удобочитаемость?

В одном эксперименте провели анализ двух различных подходов к обеспечению высокого качества программы. В одном случае программа разрабатывалась «напористым» программистом, побуждаемым к написанию максимально эффективного ее варианта, а в другом — «осмотрительным» программистом, нацеленным на обеспечение простоты и удобочитаемости программы.

Были проведены 1000 тестовых прогонов и в том и в другом случае. В результате проведенного исследования оказалось, что в «эффективной» программе было обнаружено в 10 раз больше ошибок, чем в простой. Иными словами, число показателей качества «простой» программы оказалось значительно выше.

В принципе многие методы, делающие программу эффективной, не наносят ущерба ее простоте и удобочитаемости. Эти методы следует использовать всегда.

Удобочитаемость программы более существенна, чем ее эффективность. Удобочитаемую программу легче отлаживать, моди-

фицировать и использовать, что делает обычно не тот человек, который ее писал.

Лишь в особых случаях программу следует делать более эффективной: программа либо не помещается в памяти, либо слишком долго выполняется, или она должна быть включена в библиотеку и часто использоваться. В этом случае эффективности отдают предпочтение в ущерб удобочитаемости.

Оптимизация программы

Оптимизацией программы называют такие преобразования, которые позволяют сделать ее более эффективной, т.е. сделать ее более экономной по памяти и/или более быстрой по выполнению тех же функций, что и до оптимизационного преобразования.

Два частных критерия оптимизации — время выполнения программы и объем используемой ею памяти, в общем случае, противоречат друг другу, так же как и эффективное написание программ сопровождается увеличением работы программиста. Известно, что сокращение времени выполнения программы, как правило, можно добиться за счет увеличения объема используемой памяти и наоборот. В этом случае при выборе нужного критерия вступают в силу эвристические соображения программиста, отдающего предпочтение одному из них. Обычно конкретные обстоятельства диктуют важность оптимизации программы по времени или по памяти. Например, если программа работает в реальном масштабе времени, она должна вовремя выдать результат.

Частично оптимизацию программы может выполнить оптимизирующий транслятор (компилятор). Но в основном этот процесс творческий зависит от квалификации программиста и невозможно дать алгоритм, оптимизирующий любую программу. Можно лишь обратить внимание на те аспекты, где скрыты резервы оптимизации и проиллюстрировать их на примерах.

Под областью оптимизации понимается часть программы, т.е. множество операторов, над которыми выполняются оптимизирующие преобразования независимо от других операторов программы. В частности, областью оптимизации может быть вся программа.

Существуют два подхода к оптимизации программ: «*чистка*» и *перепрограммирование*. Оба подхода имеют как достоинства, так и недостатки.

Первый подход заключается в исправлении очевидных небрежностей в исходной программе. Его достоинство — данный метод требует мало времени. Однако повышение эффективности при этом обычно незначительно.

Второй подход состоит в переделке исходной программы. Можно переделать часть программы, которая, например, расходует наибольшую часть времени. Этот подход обеспечивает обычно наилучший результат, но и самый дорогой. Он приемлем, если оптимизируемая программа подвергалась значительным изменениям.

В каждом конкретном случае повышение эффективности программы зависит от ряда факторов, таких, как стоимость улучшения программы, частота ее использования, относительная скорость выполнения различных операций в машине, от способа компилирования различных операций и т.п.

Желательно выработать в себе комплекс правил, которые будут способствовать составлению более эффективных программ, оптимизация которых в дальнейшем не потребуется.

Рассмотрим основные машинонезависимые приемы оптимизации программ.

1) *Заголовки сообщений.* Если большая часть сообщений содержит пробелы (или другие повторяющиеся символы), то для экономии памяти ЭВМ следует использовать команды, выполняющие заполнение этой части сообщения пробелами (символами).

2) *Инициирование переменных.* Если начальные значения присваиваются переменным (или константам) одновременно с их объявлением, то экономится и память, и время выполнения программы, т.к. в этом случае переменные получают начальные значения во время компилирования программы, а не во время ее выполнения.

Кроме облегчается внутреннее документирование программы и обходится ошибки в случае, когда переменной не было присвоено начальное значение.

3) *Уменьшение числа переменных.* Отработанная «временная» переменная на участке программы (например, управляющая переменная цикла) продолжает занимать область памяти и дальше во все время выполнения оставшейся части программы. Ее надо удалить (очистить).

Но ее можно использовать в программе и дальше вместо другой «временной» переменной (например, в качестве уп-

равляющей переменной в другом цикле) в другом качестве, чтобы не создавать новую. При этом тип переменной остается неизменной.

Так как идентификатор переменной, как правило, обозначает определенную ячейку памяти, то результатом выполнения рассматриваемой процедуры является определение минимального числа ячеек, необходимых для хранения промежуточных результатов.

Встречаются случаи, когда переменная, которой присвоено значение, в дальнейшем нигде не используется, или же между двумя определениями одной переменной нет ни одного ее использования. Такие определения переменных назовем неиспользованными. Оптимизационная процедура в этом случае заключается в удалении из программы лишних присваиваний значений неиспользованным переменным.

Рост числа временных и неиспользованных переменных обычно происходит у начинающих и неаккуратных программистов. Их типичная ошибка – выбор первой пришедшей в голову структуры данных. Считается хорошим стилем программирования, если в программе нет лишних переменных.

4) *Выбор типов данных.* Переменные разных типов данных в ЭВМ обрабатываются с разной затратой времени и памяти. В данном случае требуется минимальное понимание особенностей программы.

Например, если элемент массива A(I) – целочисленная, то в операторе FOR I:=1 TO 1000 DO A(I):= 0.0; произойдут 1000 преобразований типов из вещественного в целый. Здесь надо написать: A(I):= 0;.

5) *Удаление излишних операторов присваивания.* Рассматриваемая процедура состоит в удалении из программы некоторых операторов присваивания и замене в некоторых других операторах (присваивания или условного перехода) переменных, являющихся левой частью удаляемых операторов, выражениями, являющимися правой частью удаляемых операторов, то есть выполняется объединение операторов. Например, следующие четыре оператора присваивания

```
C1:= 5. + B1*D**2;
C2:= B+ SIN (R);
C3:= 5.+ A(1)*A(2);
C:= (C1 + C2)*C3;
```

можно заменить одним

C = (5 + B1*D**2 + B + SIN(R))*(5 + A(1)*A (2));

При замене только в одном операторе присваивания переменных, являющихся левой частью удаляемых операторов, выражениями, являющимися правой частью удаляемых операторов, всегда возникает положительный оптимизационный эффект, так как в этом случае просто происходит сокращение числа операций пересылок информации.

При этом предполагается, что переменные, являющиеся левой частью удаляемых операторов, в других операторах не используются.

6) *Отождествление переменных.* Если в программе имеется оператор вида A:=B, то во всех последующих операторах программы можно заменить переменную A переменной B или числовым значением переменной B, а оператор A: = B удалить.

Если переменная A является индексируемой переменной, то оператор A: = B остается в программе, а переменная A в некоторых операторах заменяется переменной B.

7) *Удаление тождественных операторов.* Суть этой процедуры состоит в удалении из программы тождественных операторов, т.е. операторов вида A: = A. Такие операторы могут появляться в программе в результате выполнения оптимизационных и других преобразований.

8) *Устранение невыполняемых операторов.* В реальных программах могут встретиться участки, содержащие невыполняемые операторы, т. е. такие операторы, которые не выполняются при любых наборах начальных данных.

Появление таких операторов можно объяснить двумя причинами:

1) в процессе отладки программы или ее модификации программист либо забывает о таких операторах, либо не хочет их менять, либо просто не в состоянии проследить за всеми последствиями, вызванными исправлениями некоторых участков программы;

2) большая логическая или информационная сложность программы не позволяет программисту «увидеть» такие операторы или даже целые невыполняемые участки программы.

Смысл рассматриваемой процедуры состоит в удалении из программы невыполняемых операторов.

9) *Использования ввода-вывода.* Операции ввода-вывода занимают много времени и должны быть сокращены до минимума. Данные, которые можно вычислить внутри программы, вводить не надо!

Две последовательные команды ввода-вывода для одного и того же устройства можно объединить в одну команду. Это сокращает количество обращений к системным подпрограммам ввода-вывода, что, естественно, сокращает время выполнения программы.

Не забудьте после отладки программы изъять все лишние (обычно отладочные) операторы ввода-вывода.

Неформатный ввод-вывод обычно быстрее форматного. Его можно использовать, если данные вводятся во внешнюю память, а затем считываются опять той же или другой программой. В этом случае не требуется преобразования данных внутренней формы представления во внешнюю и обратно. Этот способ ввода-вывода более точен, т.к. во время форматного преобразования данных могут быть утеряны значащие цифры.

10) *Выделение процедур (функций).* Если в программе имеются последовательности одинаковых операторов, которые отличаются только разными идентификаторами и значениями констант, то может быть полезно выделить их в процедуру (или функцию). Это дает значительное сокращение текста программы, но замедляет ее выполнение, т.к. вход-выход процедуры — трудоемкая операция, и работа с параметрами идет медленнее, чем с локальными переменными.

Лучше все же использовать не процедуры, а отдельные модули.

11) *Сокращение числа процедур (функций).* Связь процедур (функций) с основной программой обеспечивается операторами вызова в основной программе, которые замедляют работу программы.

Когда критерием оптимизации является время счета программы, то нужно хорошо подумать прежде, чем выделить участок программы в отдельную процедуру (функцию).

Если много операторов выполняются много раз и в разных местах программы, то, конечно, их надо выделить в отдельную процедуру. Но неразумно вызывать небольшую процедуру два (три ...?) раза.

Сокращение числа процедур (функций) выполняется заменой операторов их вызова телами этих процедур (функций) и заменой

формальных параметров соответствующими им фактическими параметрами. Зато увеличивается текст программмы на соответствующее число команд.

Описанная процедура является примером того, как два критерия оптимизации — время счета программы и объем занимаемой ею памяти — противоречат друг другу.

12) *Альтернативы.* Когда одну переменную нужно сравнить с несколькими значениями, некоторые программисты пишут так (очень плохой стиль программирования):

```
IF A = 1 THEN... ;
IF A = 2 THEN... ;
IF A = 3 THEN... ;
```

В этом случае, даже если A = 1, будут выполнены все условные операторы IF.

При использовании конструкции ELSE сравнение будет прекращено, как только будет найдено истинное условие:

```
IF A = 1    THEN...
            ELSE IF A = 2    THEN...
                             ELSE IF A = 3 THEN... ;
```

Дальнейшее улучшение эффективности по времени (оно может привести к ухудшению удобочитаемости программы) состоит в том, чтобы расположить на первом месте условие, которое по всей вероятности, является истинным в первую очередь (имеет наибольшую вероятность быть истинным), затем остальные условия — в порядке убывания их вероятности быть истинными.

То же замечание касается конструкции выбора (CASE).

13) *Арифметические операции.* Арифметические операции выполняются с разной скоростью.

Перечислим их в порядке возрастания времени их выполнения:

1) сложение и вычитание;
2) умножение;
3) деление;
4) возведение в степень.

Иногда бывает целесообразно заменить одну операцию другой. Например, 3*I может быть заменено I+I+I или X**2 можно заменить на X*X. Тем более, что для возведения в степень обычно требуется библиотечная программа.

Замена возведения в степень несколькими операциями умножения экономит и память, и время, еслипоказатель степени является небольшим целым числом.

Умножение выполняется почти в два раза быстрее деления.

Вместо A/5 можно написать 0.2*A;

Вместо Al = B/D/E + C можно написать Al = B/(D*E) + C;

SQRT(A); выполняется быстрее и точнее, чем A**0.5;

14) *Преобразование выражений.* Предварительное преобразование (упрощение) выражений может привести к исключению многих арифметических операций.

Например, X: = 2*У + 2*T;

можно заменить на X: = 2*(У + T);

Последнее преобразование исключило одну операцию умножения.

15) *Оптимизация выражений.* Суть процедуры состоит в замене всех сложных семантически эквивалентных подвыражений на одно простое.

Подвыражения считаются семантически эквивалентными, если они алгебраически эквивалентны (т. е. их можно преобразовать друг в друга, применив алгебраические преобразования), и их соответствующие операнды имеют одинаковые числовые значения.

Оптимизационный эффект в этом случае заключается в сокращении времени выполнения программы и уменьшении объема занимаемой ею памяти.

16) *Предварительное вычисление арифметических подвыражений.* Вместо, например:

```
A: = (M*B*C)/(D−E);
K: = D−E−B*C*M;
```

можно написать:

```
T: = N*B*C;
U: = D−E;
A: = T/U;
K: = U−T;
```

Сокращается текст программы и время ее выполнения за счет сокращения количества выполняемых операций. Зато быть может чуть-чуть увеличивается память для T и U.

17) *Устранение лишних скобок.* Суть этой процедуры заклю-

ются в устранении пар скобок в арифметических или логических выражениях, являющихся содержательно излишними.

18) *Устранение лишних меток.* По ряду причин в программах встречаются метки перед операторами, на которые нет передач управления.

Такие метки являются лишними и их следует удалить из программы.

19) *Устранение лишних операторов перехода.* После многих оптимизационных переделок программы, в ней могут появиться операторы безусловного перехода (GO TO), которые передают управление оператору с несуществующей меткой. Их надо удалить.

20) *Неявные операции.* Операция T:= M(I) требует в 1,5—2 раза больше времени и памяти, чем T: = A;, а если M — параметр или функция, то в 2,5—3 раза больше.

Там, где часто используется M(I), можно использовать простую переменную MI: = M(I) и т. д.

Но если индекс I — константа, то индексация выполняться не будет: при трансляции вычисляется сдвиг элемента относительно начала массива, и в процессе выполнения программы он обрабатывается как простая переменная.

21) *Чистка циклов.* Немного о памяти. Обычно программисты не заботятся о памяти до тех пор, пока не превысят ее размеры. Тогда становится очевидным, что память не безразмерна. Имеются прогнозы, что скоро будем располагать достаточной памятью для решения любой задачи. Но размеры решаемых задач также растут по мере увеличения размера памяти ЭВМ.

Если программист тратит много времени, пытаясь повысить эффективность своей программы, улучшая оператор, который выполняется только один раз, это подобно попытке уменьшить свой вес, остригая ногти.

Время выполнения программы в значительной мере зависит от времени выполнения многократно повторяемых участков —от циклов. Поэтому основное внимание необходимо уделять именно циклам.

Процедура чистки цикла уменьшает время выполнения цикла (следовательно, и программы) путем удаления из его тела арифметических выражений или их частей, не зависящих от управляющей переменной цикла.

Выражения, которые не меняют своих значений в теле цикла,

должны вычисляться перед циклом. Это не меняет длины программы, но дает выигрыш в скорости ее выполнения.

```
FOR I: = 1 TO 100 DO FOR J: = 1 TO 10 DO X: = У*Z + C[I, J].
```

Здесь У*Z будет вычисляться тысяча раз, а в следующем примере — только один раз:

```
УZ: = У*Z;
FOR I: = 1 TO 100 DO FOR J: = 1 TO 10 DO X: = УZ + C[I, J].
```

За исключением инвариантных выражений из циклов извлекается небольшая плата в виде части памяти, необходимой для запоминания и неоднократного использования результатов инвариантных выражений.

Интересной разновидностью чистки циклов является изменение порядка циклов:

Цикл

```
FOR I: = 1 TO N DO FOR J: = 1 TO M DO...
```

при N < M выполняется быстрее, чем цикл

```
FOR I: = 1 TO M DO FOR I: = 1 TO N DO...
```

т.к. в первом случае выполняется M*N + N операций организации циклирования, а во втором — M*N + M.

22) *Использование циклов.* Циклы требуют некоторого дополнительного количества памяти на инициирование, проверку, изменение управляющей переменной и установку всех констант.

Иногда бывает полезным отказаться от использования циклов.

Немногократно повторяющиеся последовательности операторов, требующие сложной организации циклов, часто можно писать в программе последовательно (линейно), а не итеративно.

23) *Объединение циклов.* При трансляции любого цикла с языка высокого уровня в машинные коды в объектной программе появляются операции, реализующие заголовок цикла, т. е. увеличение управляющей переменной цикла на значение его шага, сравнение управляющей переменной с граничным значением, передачу управления.

Процедура объединения циклов производит соединение нескольких циклов с одинаковыми заголовками в один. Например:

```
FOR I: = 1 TO 500 DO X[I]: = 0;
FOR J: = 1 TO 500 DO У[J]: = 0;
```

можно объединить в один цикл, что сокращает как время выполнения программы, так и память:

```
FOR I: = 1 TO 500 DO BEGIN X[I]: = 0;   У[I]: = 0 END;
```

14) *Вынос ветвлений из цикла.* Пусть исходный цикл состоит из двух частей, и в зависимости от выполнения определенного условия выполняется одна из этих частей.

```
FOR I = 1 TO 200 DO
    IF K = 2  THEN A(I) = B(I)*C(I)  ELSE A (I) = B (I) + 2;
```

Тогда исходный цикл можно заменить двумя циклами, соответствующими двум частям исходного цикла:

```
IF K = 2   THEN   FOR I = 1   TO   200   DO   A(I) = B(I)*C(I);
           ELSE   FOR I = 1   TO   200   DO   A (I) = B (I) + 2...
```

В результате выноса ветвления из цикла уменьшается время счета программы благодаря сокращению числа проверок выполнения условий.

15) *Удаление пустых циклов.* Суть этой процедуры состоит в удалении из программы пустых циклов.

Такие циклы могут появиться в программе в результате небрежности программиста, а чаще всего в результате выполнения оптимизационных и других преобразований.

Следует, однако, отметить, что в некоторых случаях пустые циклы используются как специальный прием программирования, например, в программах реального времени для создания задержек.

В общем случае выполнение процедуры устранения пустых циклов экономит время выполнения программы и объем занимаемой ею памяти.

16) *Сжатие циклов.* В ряде случаев в теле цикла встречаются условия, которые могут ограничивать область изменения управляющей переменной цикла.

```
FOR I = A TO B DO IF I <= C THEN R(I) = P(I);
```

Процедура сжатия циклов осуществляет перенос таких ограничений в заголовок цикла:

```
DO I = A TO C DO R(I) = P(I);
```

Здесь предполагается, что A <: C < B.

В результате выполнения процедуры сжатия циклов из тела

цикла удаляется логический оператор перехода и сокращается число выполнениий тела цикла.

27) *Управление по выбору.* В операторе выбора выполняется первая группа команд, у которой условие выбора истинно.

Поэтому для увеличения скорости выполнения программы необходимо располагать группы команд в операторе выбора в порядке убывания частоты вероятности их использования.

Иными словами, в операторе выбора необходимо первой расположить группу команд, которая наиболее часто должна быть выполнена. Второй – следующую по частоте выполнения группу команд. И так далее. Группа команд, которая выполняется крайне редко, в операторе должна располагаться последней.

Оптимизация эффективности эксплуатации программного обеспечения

Оптимизация эффективности эксплуатации ПО – это оптимизация качества разработки, которая касается непосредственно разработчика (автора) и которая за счет некоторой потери в эффективности позволяет получить более надежную, удобную в эксплуатации и легкую в модификациях программу.

Необходимо заранее определить *запас на развитие программы*. Программа, рассчитанная на эксплуатацию в течение длительного времени, должна быть написана с учетом того, что со временем в ее функции или исходные данные могут быть внесены изменения.

Важным частным случаем запаса на развитие программы является ее *универсальность* и *настройка по параметрам*. Например, вряд ли можно найти много алгоритмов, для которых существенно то, что матрица имеет размер 20×20, а для матрицы размером 21x21 они уже непригодны. При этом, если в алгоритме нет ничего «существенного квадратного», то лучше считать размер матрицы с некоторыми параметрами – M и N.

Задание значений всем таким параметрам сводится в одно место (лучше в начало программы). В результате получится блок настройки программного обеспечения по параметрам, и все, относящиеся к параметрам изменения в нем, затрагивают только этот блок.

Минимизация сложности программного обеспечения имеет непосредственное отношение к надежности, так как она противостоит его универсальности: представление пользователю слиш-

ром большого числа дополнительных возможностей и вариантов усложняет программное изделие.

Ошибки пользователя увеличивают вероятность перехода программного обеспечения в непредсказуемое состояние.

Минимизация ошибок пользователя не уменьшает число ошибок в программном обеспечении, но увеличивает его надежность за счет вероятности обнаружения ошибок, которые могут быть допущены пользователем.

Помимо минимизации ошибок пользователя, программный продукт также должен надлежащим образом обращаться с ошибками, если они все же возникают, а возникать они будут независимо от того, насколько хорошо были спроектированы правила взаимодействия программного изделия с пользователем.

Увеличить надежность программного продукта можно за счет *блокировки ошибок*, которую скорее можно отнести к неэффективности программы, так как она снижает и качество программы (на нее расходуется и время и память), и качество разработки (требуется дополнительное программирование и отладка; усложняется логика программы). Но программа без блокировки ошибок может столь «эффективно» обработать, например, файл, что затем его нельзя будет восстановить даже эффективными программами.

Конкретные методы блокировки ошибок и степень контроля данных выбираются в зависимости от конкретных условий и обстоятельств.

Источник исходных данных. Если от человека, то контроль нужен, так как человеку свойственно ошибаться; если от физических датчиков или подготовленные другой программой, то контроль не нужен, так как современные программы и датчики надежны.

Объем исходных данных. Программы принято делить на научные, коммерческие и системные.

Научные программы в основном имеют небольшой объем исходных данных, представляющих, как правило, однородные числа. Их легко проверить вручную. Кроме того, замену одного числа другим ЭВМ часто обнаружить не может. Поэтому программа может их не контролировать.

Коммерческие программы в основном вводят и выводят большой объем разнородной информации, который невозможно качественно проверить вручную. Контроль данных необходим.

Системные программы, обращенные к пользователю (напри-

мер, трансляторы), должны контролировать данные как коммерческие. Прочие системные программы получают данные от других программ и контроль данных почти не ведут.

Язык исходных данных. Чем больше избыточность языка, тем более он устойчив к ошибкам. Неправильную цифру в числе часто обнаружить логическим контролем невозможно, а неправильную букву в слове обнаружить можно. Ведь искаженное слово не входит в нужный словарь. Там, где нужен контроль данных, должен быть соответствующий язык.

Последствия ошибки. Контроль данных можно свести к минимуму или вообще пренебречь им, если он сложен, а ошибка не влечет катастрофических последствий.

Защита файлов. Помимо тщательного контроля данных, нужно предусмотреть еще организационные и/или программные меры защиты самих файлов их содержащих.

К организационным мерам можно отнести периодическое копирование файлов данных и хранение их в надежном месте (например, в сейфе).

К программным мерам защиты данных можно отнести, например, запрос из программы у пользователя подтверждения на выполнение опасной операции.

Программные средства могут сами формировать копию файла с данными на другой носитель, перепись удаленных (замещенных) записей во вспомогательный файл, откуда их в случае необходимости можно вернуть назад.

Модификация программы

В процессе функционирования отлаженного программного обеспечения возможно обнаружение в нем ошибок (так называемые подводные камни) и появление необходимости модификации и расширения его функций.

Под модификацией понимается внесение изменений в уже отлаженную и правильно функционирующую программу. Такие изменения связаны в основном с изменением алгоритма решения задачи, а также с созданием любого достаточно сложного комплекса программ.

Внесение даже простых изменений в программу большого объема и сложной структуры — это далеко не тривиальная задача, так как они затрагивают большое число управляющих и информационных связей, что может вызвать появление нежелательных эффектов.

Как правило, программист не в состоянии предвидеть все подобные эффекты, и для того, чтобы быть уверенным в работоспособности модифицированной программы, повторно осуществляет процесс отладки и верификации модифицированной программы.

Однако задача модификации программ существенно упрощается, если программисту будут автоматически сообщаться возможные эффекты необходимого изменения и связанные с данным изменением потенциальные места появления ошибок.

Для такой автоматизации можно использовать технику, применяемую при оптимизации программ, а именно, методы и алгоритмы, осуществляющие глобальный анализ управляющих и информационных связей программы.

Методы, используемые при оптимизации программ, позволяют более целенаправленно модифицировать программу за счет явного проявления изменений, вносимых в управляющий и информационный графы программы.

Автоматизация процесса модификации программ (помимо своего прямого назначения) облегчит выполнение отладки и верификации программ, так как эти процессы тесно связаны между собой.

Опыт построения систем, являющихся прообразом систем модификации программ, позволяет говорить о некотором стандартном наборе модификационных процедур, аналогичных оптимизационным процедурам. Выделение набора модификационных процедур позволяет строить системы модификации программ так же, как системы оптимизации программ.

Здесь также следует подчеркнуть, что в связи со сложностью управляющих и информационных связей программы, управляемая модификация, в частности, диалог человек—ЭВМ, имеет особое значение в системах модификации программ (так же, как и в системах оптимизации программ и в других рассмотренных системах).

РЕЗЮМЕ

* Под стилем программирования понимается набор приемов или методов программирования, которые используют опытные программисты, чтобы получить правильные, надежные, эффективные, удобные для применения и легко читаемые программы.

* Правила хорошего стиля программирования — это результат

соглашения между опытными программистами (маленький стандарт).

* Программы должны составляться таким образом, чтобы их могли прочитать в первую очередь люди, а не машины. Программа — это документ для последующего использования, учебный материал по кодированию алгоритмов и средство для дальнейшей разработки более совершенных программ.

* Программа должна передавать логику и структуру алгоритма настолько просто, насколько это возможно. Следует избегать всевозможных программистских трюков, т.к. чем их больше, тем труднее будет разобраться в логике программы самому автору, а кто-либо другой это не сможет сделать.

* Так же как разделение большого произведения на главы и параграфы облегчает его чтение, так и разбиение большой программы на параграфы, разделы (подпрограммы и модули), путем выделения логических единиц улучшает ее восприятие, помогает избежать однообразия и хорошо организовать материал. Название раздела отражает его цель.

* Структура программы хорошо реализуется с некоторым смещением.

* Для удобства чтения программа должна быть хорошо откомментирована. Если операторы позволяют понять как работает программа, то комментарии к ним должны пояснить что и зачем делают эти операторы.

* Программы с пояснительными комментариями значительно легче отлаживать и модифицировать. Комментарии должны содержать некоторую дополнительную информацию, а не перефразировать программу. Они нужны как на стадиях проектирования и отладки программы, так и позже.

* Каждая программа, подпрограмма или процедура (модуль) должна начинаться с блока комментариев (вводных комментариев).

* Структурные программы обычно требуют меньше комментариев, чем неструктурированные, так как программы первого вида понятнее и в них меньше переходов.

* Корректность или правильность программы наиболее полно определяется степенью ее соответствия предъявляемым к ней формальным требованиям, поэтому она является субъективным и ее невозможно определить количественно. Неопределенность программных спецификаций уменьшается в процессе разработки программ путем уточнения требований по согласованию между разработчиком и заказчиком.

* Формальные правила проектирования программ устанавливаются стандартами и инструкциями подготовки текстов программ и их структурного построения. Они формируются на уровне тестов на языки программирования и на базу данных.

* Будем считать программное изделие правильным, если оно:

— решает действительно ту задачу, для которой оно было разработано;

— не «зависает» и не заканчивает свою работу аварийно;

— удовлетворяет всем требованиям из документа «Соглашение о требованиях» («Техническое задание») с учетом их уточнений в процессе проектирования программного изделия;

— разработано в соответствии с формальными правилами проектирования программного обеспечения.

* Основной задачей программирования является создание правильных, а не эффективных программ. Эффективная программа бесполезна, если она не обеспечивает правильных результатов.

* Только тогда следует рассмотреть возможность оптимизации программы, если готовая программа пригодна и нужна для работы, если ее будут выполнять многократно, и если статус проекта и фирмы это позволяет.

* Удобочитаемость программы более существенна, чем ее эффективность.

* Оптимизацией программы называют такие преобразования, которые позволяют сделать ее более эффективной, т.е. сделать ее более экономной по памяти и/или более быстрой по выполнению тех же функций, что и до оптимизационного преобразования.

* Частично оптимизацию программы может выполнить оптимизирующий транслятор (компилятор). Но в основном этот процесс творческий, зависит от квалификации программиста и невозможно дать алгоритм, оптимизирующий любую программу.

* Оптимизация эффективности эксплуатации — это оптимизация качества разработки, которая касается непосредственно разработчика (автора), и которая за счет некоторой потери в эффективности позволяет получить более надежную, удобную в эксплуатации и легкую в модификациях программу.

Темы для повторения

1. Стиль программирования.
2. Малый программистский стандарт.
3. Общая организация программы и ее запись.

4. Правила записи структурных конструкций.
5. Комментарии.
6. Блок комментариев.
7. Рекомендации, способствующие эффективному документированию внутри программы.
8. Корректность программ.
9. Эффективность программ. Эффективность или удобочитаемость?
10. Оптимизация программ.
11. Оптимизация эффективности эксплуатации ПО.
12. Модификация программ.

Глава 7. Объектно-ориентированное программирование

В объектно-ориентированном программировании (ООП) различают три вида модели проблемной области: объектную, динамическую и функциональную.

Объектная модель показывает статическую структуру проблемной области, для которой разрабатывается система. Сначала определяются классы объектов, затем зависимости между объектами, включая агрегацию. Для упрощения структуры классов используется наследование. Объектная модель должна содержать краткие комментарии на естественном языке.

Динамическая модель показывает поведение системы, в особенности последовательность взаимодействий ее компонентов (объектов). Сначала готовятся сценарии типичных сеансов взаимодействия с системой, затем определяются внешние события, управляющие взаимодействие системы с внешним миром; после этого строится диаграмма состояний для каждого активного объекта, на которой представлены образцы событий, получаемых и порождаемых системой, а также действий, выполняемых ею. Построенные диаграммы состояний сравниваются между собой, чтобы убедиться в их непротиворечивости.

Функциональная модель показывает функциональный вывод значений безотносительно к тому, когда они вычисляются. Сначала определяются входные и выходные значения системы как параметры внешних событий. Затем строятся диаграммы потоков данных, показывающие как вычисляется каждое выходное значение по входным и промежуточным значениям. Диаграммы потоков данных выявляют взаимодействие с внутренними объектами системы, которые служат хранилищами данных в периоды между сеансами работы системы. В заключение определяются ограничения и критерии оптимизации.

Каждая модель хорошо подходит к одному из перечисленных систем:

— *системы пакетной обработки* — обработка данных производится один раз для каждого набора входных данных;

— *системы непрерывной обработки* — обработка данных производится непрерывно над сменяющимися входными данными,

— *системы с интерактивным интерфейсом* — системы, управляемые внешними воздействиями (обычно пользователем);

— *системы динамического моделирования* — системы, моделирующие поведение объектов внешнего мира;

— *системы реального времени* — системы, в которых преобладают строгие временные ограничения;

— *системы управления транзакциями* — системы, обеспечивающие сортировку и обновление данных; они имеют коллективный доступ. Типичной системой управления транзакциями является СУБД коллективного пользования.

Проектируя систему одного из вышеперечисленных типов, небходимо использовать соответствующую архитектуру. Во время разработки архитектуры программного обеспечения необходимо предусмотреть поведение каждого объекта, компонента и всей системы в пограничных ситуациях: инициализации, терминации и обвале.

Инициализация. Перед тем, как начать работать, система (компонент, объект) должна быть приведена в фиксированное начальное состояние: должны быть проинициализированы все константы, начальные значения глобальных переменных и параметров, задачи и, возможно, сама иерархия компонентов. Во время инициализации, как правило, бывает доступна лишь часть возможностей системы.

Терминация состоит в освобождении всех внешних ресурсов, занятых задачами системы.

Обвал — это незапланированная терминация системы. Обвал может возникнуть в результате ошибок пользователя, нехватки ресурсов, или внешней аварии. Причиной обвала могут быть и ошибки в программном обеспечении системы.

На этапе программирования объектная модель уточняется и пополняется, в нее вводятся внутренние (вспомогательные) классы. Новые внутренние классы не имеют соответствий в реальном мире. Они связаны с программированием и могут существенно его упростить.

Во многих случаях бывает полезным внести некоторые изменения в структуру объектной модели. Эти изменения сводятся к введению дополнительных классов и к перераспределению операций между классами. При распределении операций по классам руководствуются соображениями:

— если операция выполняется только над одним объектом, то она определяется в классе, экземпляром которого является этот объект;

— если аргументами операции являются объекты разных классов, то ее следует поместить в класс, к которому принадлежит результат операции;

— если аргументами операции являются объекты разных классов, причем изменяется значение только одного объекта, а значения других объектов только читаются, то ее следует поместить в класс, к которому принадлежит изменяемый объект;

— если классы вместе с их зависимостями образуют звезду с центром в одном из классов, то операцию, аргументами которой являются объекты этих классов, следует поместить в центральный класс.

Реализация управления системы связана с реализацией ее динамической модели. Известны три подхода к реализации динамической модели:

— *процедурное управление* — традиционный способ реализации динамической модели; ему соответствует определенный фрагмент программы;

— *управление через события* — это явная реализация конечного автомата; хорошо реализуется визуальным пргограммированием;

— *использование параллельных независимых задач.*

Уточняя определения классов и операций, надо постараться увеличить степень *наследуемости*: чем больше классов находятся в отношении наследования, тем меньше функций, реализующих операции этих классов, необходимо будет запрограммировать.

Для увеличения степени наследуемости требуется:

— перестроить классы и операции;

— выявить одинаковые (или взаимно-однозначно соответствующие) операции и атрибуты классов и определить для этих классов абстрактный суперкласс;

— использовать делегирование операций, когда наследование семантически некорректно.

Иногда одна и та же операция бывает определена в нескольких классах, что позволяет ввести общий суперкласс для этих классов, в котором и реализуется эта операция.

Но чаще операции в разных классах бывают похожими, но не одинаковыми. В таких случаях нужно попытаться внести несущественные изменения в определения этих операций, чтобы они

стали одинаковыми, т.е. имели одинаковый интерфейс и семантику.

При этом можно использовать следующие приемы:

— если операции имеют одинаковую семантику, но разное число формальных параметров, то можно добавить отсутствующие параметры, но игнорировать их при выполнении операции; например, операция обрисовки изображения на монохромный монитор не требует параметра «цвет», но его можно добавить и не принимать во внимание при выполнении операции;

— некоторые операции имеют меньше параметров потому, что они являются частными случаями более общих операций, такие операции можно не реализовывать, сведя их к более общим операциям с соответствующими значениями параметров; например, добавление элемента в конец списка есть частный случай вставки элемента в список;

— одинаковые по смыслу свойства или операции разных классов могут иметь разные имена; их можно переименовать и перенести в класс, являющийся общим предком рассматриваемых классов;

— операция может быть определена не во всех классах некоторой группы классов; можно тем не менее вынести ее в их общий суперкласс, переопределив ее в подклассах как пустую там, где она не нужна.

В основе объектно-ориентированного программирования лежит идея объединения в одно целое структуры данных и действий (называемых методами), которые производятся над этими данными. При таком подходе организация данных и программная реализация методов оказываются гораздо сильнее связаны, чем при традиционном программировании.

Объектно-ориентированное программирование базируется на трех основных понятиях: инкапсуляции, наследовании, полиморфизме.

Инкапсуляция — это комбинирование данных с процедурами и функциями, которые манипулируют этими данными. В результате получается новый тип данных — объект.

Наследование — это возможность использования уже определенных объектов для построения иерархии объектов, производных от них. Каждый из «наследников» наследует описание данных «прародителя» и доступ к методам их обработки.

Полиморфизм — это возможность определения единого по имени действия (процедуры или функции), применяемого от-

применимо ко всем объектам иерархии наследования, причем каждый объект может «заказывать» особенность реализации этого действия над «самим собой».

Объектно-ориентированный стиль программирования может заметно упростить написание сложных программ, придать им гибкость.

Программы в этом случае получают такие дополнительные свойства, как:

- повторная используемость;
- расширяемость;
- устойчивость к неправильным данным;
- системность.

Программа обладает свойством системности, если она применима в качестве обобщенного оператора при «крупноблочном программировании».

Крупноблочное программирование — это систематическое использование ранее разработанных крупных программных единиц (таких, как классы, подсистемы или модули) при разработке новых программных систем.

Следующие рекомендации могут помочь разрабатывать классы, обладающие свойством системности:

- методы должны быть хорошо продуманы;
- методы должны быть понятны всем, кто их прочитает;
- методы должны быть легко читаемы;
- имена методов (компонентов) должны быть такими же, как и в модели;
- методы должны быть хорошо документированы;
- спецификации методов должны быть доступны.

В объектно-ориентированном программировании объектом можно назвать все, что может иметь имя и может быть рассмотрено как одно целое.

В качестве примера объекта можно привести: комбинацию вида и данных, которую можно рассматривать как одно целое, элемент управления, графический элемент, рисунок, метку, форму и пр.

Каждый объект определен классом (class).

Класс используется для создания объектов и определяет их характеристики. Например, объект, известный как элемент управления, не существует, пока он не нарисован на форме.

Когда создается объект, создается копия, или экземпляр (instance) класса, который он представляет. Затем уточняются

другие его характеристики. Все объекты создаются как идентичные копии своих классов. Например, каждый объект «Кнопка управления» является экземпляром класса «Command Button», а «Элемент управления списком» представляет собой экземпляр класса «List Box». Объекты одного класса используют общий набор характеристик и способностей (свойств, методов и событий).

Свойства индивидуальных объектов можно изменять. Каждому объекту присваивается свое имя. Объекты можно по отдельности заблокировать и разблокировать, поместить в разные места на форме и т.д.

Визуальные инструментальные средства разработки программ предоставляют инструменты, позволяющие строить свои объекты или комбинировать их из различных источников, включая другие приложения, поддерживающие OLE технологию. Последнее сохранит время разработчика приложения, так как ему не придется писать код, воспроизводящий все функциональные возможности объектов другого приложения.

Объекты обладают свойствами, событиями и методами.

Свойства отображают некоторые атрибуты объекта, методы – его действия, а события – это реакции объекта. Индивидуальные свойства объекта можно установить (изменить) во время выполнения программы или во время разработки приложения. В последнем случае лучше использовать окно Properties (Свойств), что позволяет вообще не писать никакого кода.

Методы могут воздействовать на значения свойств. В программе метод вызывается по имени, например, object.method (имя объекта + название метода).

События инициируются, когда изменяются некоторые свойства объекта. Событие можно вызвать шелчком или движением «мыши», нажатием клавиши, из программы и пр. События вызывают такие действия, как открытие / закрытие формы, получение объектом фокуса и пр.

Некоторые объекты могут содержать другие объекты. Удобство использования объектов в качестве контейнеров (containers) для других объектов заключается в том, что можно ссылаться на контейнер в коде для уточнения используемого объекта. В частности, контейнером объектов может быть форма, которая сама является объектом.

Наиболее часто формы используются для создания интерфейса приложения, но они также являются объектами, которые можно вызывать из других модулей приложения. Формы тесно

...ются с модулями классов. Главное различие между ними заключается в том, что формы могут быть видимыми объектами, тогда как модули не имеют видимого интерфейса.

Объектно-ориентированные языки программирования

Реализация программного обеспечения связана с использованием одного из языков программирования. Показано, что наиболее удобными для реализации программных систем, разрабатываемых в рамках объектно-ориентированного подхода, являются объектно-ориентированные языки программирования, хотя возможна реализация программ и на обычных (не объектно-ориентированных) языках (любой алгоритмический язык программирования ранней версии).

Объектно-ориентированные языки программирования пользуются в последнее время большой популярностью среди программистов, так как они позволяют использовать преимущества объектно-ориентированного подхода не только на этапах проектирования (конструирования и программирования) программного обеспечения, но и на этапах его реализации, тестирования и сопровождения.

Первый объектно-ориентированный язык программирования Simula 67 был разработан в конце 60-х годов в Норвегии. Авторы этого языка очень точно угадали перспективы развития программирования: их язык намного опередил свое время. Однако программисты того времени оказались не готовы воспринять новизну языка Simula 67, и он не выдержал конкуренции с другими языками программирования (прежде всего, с языком Fortran).

Прохладному отношению к языку Simula 67 способствовало и то обстоятельство, что он был реализован как интерпретируемый (а не компилируемый) язык программирования, что было совершенно неприемлемым в 60-е годы, так как интерпретация связана со снижением эффективности (скорости выполнения) программ.

Но достоинства языка программирования Simula 67 были замечены некоторыми программистами, и в 70-е годы было разработано большое число экспериментальных объектно-ориентированных языков программирования: например, языки CLU, Alphard, Concurrent Pascal и др. Эти языки так и остались экспериментальными, но в результате их исследования были разработаны современные объектно-ориентированные языки программирования: C++, Smalltalk, Eiffel и др.

Наиболее распространенным объектно-ориентированным языком программирования, безусловно является C++. Свободно распространяемые коммерческие системы программирования C++ существуют практически на любой платформе.

Разработка новых объектно-ориентированных языков программирования продолжается. С 1995 года стал широко распространяться новый объектно-ориентированный язык программирования Java, ориентированный на сети компьютеров и, прежде всего, на Internet. Его синтаксис напоминает синтаксис языка C++, однако эти языки имеют мало общего.

В настоящее время практически любой классический язык программирования (Basic, Pascal и пр.) дорос до объектно-ориентированного с элементами визуального программирования.

Визуальное программирование

Разрабатывая программное обеспечение (приложение) для пользователя, разработчик тратит много времени на программирование пользовательского интерфейса.

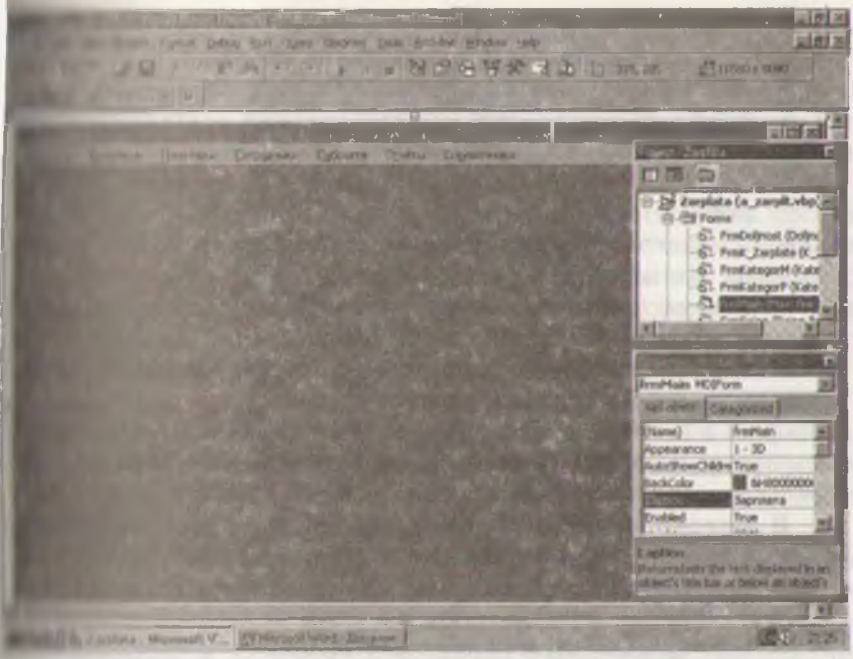
С появлением операционной системы Windows появились и новые инструментальные средства типа Delphi, Vizual Basic, Access и др. для создания приложений по методу GUI (Graphical User Interface — графический интерфейс пользователя), с помощью которого основа приложения «визуально рисуется» на экране.

Теперь можно в короткие сроки построить пользовательский интерфейс разрабатываемого программного продукта на основе простого механизма создания нестандартных панелей инструментов, командных кнопок и пользовательских меню.

При этом совсем не требуется писать программы в классическом понимании этого слова. Они будут автоматически составлены на встроенном языке программирования во время разработки приложения.

Выполнение программы, полностью написанной на алгоритмическом языке высокого уровня в традиционном смысле, начинается с первой строчки и протекает через различные ее части согласно заданному алгоритму. Управление приложением пользователь, как правило, осуществляет через пункты меню, «горячие» клавиши, а также — через сообщения, вводимые им на запрос программы.

Приложение, разработанное по Windows GUI-технологии, работает совершенно по-другому.



Интегрированная среда разработки в Vizual Basic

Его ядром является набор независимых частей кода (процедур), активизирующихся в ответ на события, происходящие вокруг. Это является принципиальным.

Теперь вместо создания программы, отражающей мысли программиста, получается приложение, управляемое пользователем.

Визуальное программирование выполняется в так называемой интегрированной рабочей среде (Integrated Development Environment) и предполагает следующие действия:

— *создание интерфейса* пользователя в виде формы, на которой размещают («рисуют») элементы управления приложением, например, текстовые поля и кнопки управления;

— *определение свойств* для формы и элементов управления, чтобы задать, например, надписи, цвет и размер;

— *запись* соответствующего программного *кода* (текста программы) в ответ на события, чтобы оживить приложение (чтобы оно работало).

Интегрированная среда разработки для Windows (IDE for Windows) обычно состоит из следующих частей:

— *заголовок* — верхняя строка, содержащая также три кнопки Windows: свертывания, развертывания и закрытия окна;

— *строка меню* — позволяет управлять приложением;

— *панель инструментов* (кнопок) — для ускорения выполнения многих обычных действий, часто служащая альтернативой команд меню;

— *рабочая область* — область окна, в которой создаются и обрабатываются все объекты приложения, она практически занимает все окно;

— *контекстное меню* — содержит команды для часто выполняемых действий.

Контекстное окно открывается после нажатия на правую кнопку мыши, когда ее указатель находится на объекте, с которым в данный момент происходит работа. Список доступных команд контекстного меню зависит от того, на какой области среды разработки был выполнен щелчок.

В зависимости от инструментальной среды разработки приложения добавляются другие части. Например:

— строка статуса и подсказок — нижняя строка. В ней выводятся сообщения о работе системы;

— панель элементов управления (Toolbox), которая содержит набор инструментов, необходимых во время разработки приложения для размещения элементов управления на форме;

— окно свойств (Properties), в котором устанавливаются свойства для выбранной формы или элемента управления;

— окно редактора кода, в котором редактируется программный код;

— дизайнер форм – окно, в котором разрабатывается форма;

— другие окна.

Отметим, что все окна представляют собой обычные окна Windows, получающие по умолчанию все стандартные характеристики окна текущей версии Windows. В частности, они наследуют такие элементы, как границы с изменением размера, оконное меню управления (в левом верхнем углу), а также кнопки Maximize, Minimize и Exit в правом верхнем углу.

На экране одновременно могут находиться несколько окон, представляющих разные приложения или объекты. Каждый объект имеет полный доступ к буферу обмена (clipboard) и к информации большинства приложений Windows, работающих в одно и то же время. Но только с одним окном может работать пользователь в данный момент времени. Это окно будет иметь «Активный

заголовок окна» (active title bar), выделенный повышенной яркостью. Про него говорят, что он имеет фокус.

Современные инструментальные средства разработки приложений для пользователя предлагают разработчику набор мастеров, которые помогут ему в короткий срок спроектировать нужное программное обеспечение.

Мастер разработки автоматизирует процесс создания проекта или его части. Разработчику проекта остается только отвечать на вопросы, задаваемые мастером в ходе его создания. В процессе разработки приложения можно в любое время вернуться назад или остановить (закончить) работу мастера. В итоге будет автоматически сгенерировано большое количество полезного кода.

Конечно, все требования пользователя мастер выполнить не в состоянии. Поэтому, чтобы полностью и окончательно «оживить» все автоматически созданные мастером возможности, проектировщику приложения потребуется добавить некоторое количество дополнительного кода.

Мастер разработки приложения представляет собой мощное, но всего лишь вспомогательное средство программирования. Он не может полностью заменить программиста.

Фундаментом, основой создания интерфейса приложения являются формы (forms).

Форма — это объект, который обладает свойствами, определяющими внешний вид формы, методами, определяющими ее поведение, и событиями, которые определяют ее взаимодействие с пользователем.Режим формы и ее удобный дизайн представляет собой дружественную среду общения приложения с пользователем.

Форма содержит элементы управления приложением: поля ввода (text boxes), меню, командные кнопки (command buttons), переключатели (option buttons), флажки (check boxes), списки (list boxes), линейки прокрутки (scroll bars), а также диалоговые окна для набора файла или каталога. Она может содержать объекты OLE (Object Linking and Embedding — связь и внедрение объектов), например, картинки, музыку и диаграммы.

Элементы управления — это объекты, содержащиеся внутри формы и обеспечивающие интерактивный режим работы пользовательского приложения.

Каждый тип элемента управления относится к определенному классу и имеет свой собственный набор свойств, методов и событий.

Одни элементы управления лучше всего подходят для ввода или отображения текста, другие – для управления приложением.

Для отображения или ввода текста в форме применяются элементы управления метка (Label) и тестовое поле (Text Box).

Метка используется для отображения на форме текста, который пользователь не может изменять.

Она обычно используется для оформления формы. Например, для вывода заголовков и подсказок пользователю, для описания других элементов управления и пр.

Отображаемый в метке текст управляется свойством Caption, которое может быть установлено во время разработки приложения в окне Properties (Свойств) или оператором присваивания во время выполнения программы.

Текстовое поле является гибким элементом управления и применяется как для получения вводимых пользователем данных, так и для отображения текста.

Его не следует использовать для отображения неизменяемого пользователем текста.

Его значение можно установить или изменить тремя способами: разработчиком приложения во время проектирования в окне Properties (Свойств), оператором присваивания во время выполнения программного кода или пользователем во время выполнения приложения.

Современные инструментальные средства разработки приложений обычно содержат несколько стандартных элементов управления, полезных для организации выбора: флажок (Check boxes), переключатели (Option buttons), список (List box) и комбинированное окно. (Combo box)

Флажок показывает, включено или нет определенное условие и может принимать только два значения типа «да/нет», «правда/ложь», «1/0» и т.п.

Так как флажки работают независимо друг от друга, пользователь может установить любое их число одновременно.

При выполнении щелчка на флажке для него происходит событие *Click* (Щелчок), которое может быть обработано в программном коде.

Переключатели предоставляют пользователю выбор из двух или более возможностей.

Обычно они работают в группе. При выборе одного, немедленно сбрасываются все другие переключатели в группе.

Объединение переключателей в группу означает для пользо-

ется, что он может выбрать одну и только одну возможность из данного набора переключателей.

Список представляет пользователю перечень возможных вариантов выбора, отображаемых обычно вертикально в виде списка.

Если число элементов списка превышает возможности отображения их в окне, то появляется полоса прокрутки, с помощью которой пользователь может просматривать список элементов, прокручивая его вверхи вниз.

Комбинированное окно совмещает возможности списка и текстового поля.

В этом элементе управления пользователь может производить выбор значения либо вводом текста в поле ввода комбинированного окна, либо выбором элемента из его списка.

Одним из основных способов обеспечения взаимодействия пользователя с приложением является *кнопка управления* (Command Button), нажатием которой пользователь может выполнять разнообразные действия.

Существует много способов «нажать» кнопку управления:

— щелкнуть «мышью» на кнопке;

— переместить фокус на кнопку, нажимая клавишу <Tab>, и затем выбрать эту кнопку, нажав клавишу <Spacebar> или <Enter>;

— нажать клавиши доступа (<Alt> + <подчеркнутая буква>) для выбранной кнопки управления;

— установить значение для кнопки в программе равным True.

Все эти действия заставляют вызывать процедуру обработки события *Click* (Нажатие на кнопку).

Между прочим, кнопка, получающая фокус, становится трехмерной. Это достигается рисованием тонкого пунктирного прямоугольника вокруг текста на кнопке, и тонкого прямоугольника вокруг самой кнопки.

Для улучшения дизайна формы применяются так называемые «*Графические элементы управления*», которые облегчают работу с графикой. Это могут быть изображения (картинки), линии, прямоугольники и т.п.

При запуске проекта формы становятся окнами, которые и видит пользователь.

Создание интерфейса приложения заключается в размещении элементов управления в форме, в задании для них значений свойств в окне Properties (Свойства) и в написании программ,

обрабатывающих события типа щелчка «мыши»; нажатия на клавишу, открытия формы и пр. Последнее действие представляет собой традиционное программирование.

Программные коды размещаются в процедурах обработки событий (event procedures) или в отдельных модулях и могут быть вызваны на изменение в окне редактора кода.

Созданное приложение работает по следующему алгоритму.

1. Отслеживаются все окна и элементы управления для каждого окна на предмет определения всех событий, относящихся к ним (движения или щелчки мыши, нажатие клавиш, получение или потеря фокуса и т.п.).

2. Когда событие определено и для него не находится встроенная процедура ее обработки, то ищется процедура, написанная программистом для обработки данного события.

3. Если такая процедура существует, она выполняется, иначе ожидает следующее событие.

4. Выполнение пункта 1.

Перечисленные действия повторяются циклически до тех пор, пока приложение не завершит работу.

Создав интерфейс приложения с помощью форм и элементов управления, необходимо написать программный код, который определяет поведение приложения.

Современные инструментальные средства разработки приложений делают написание программ приятным и легким процессом, позволяя автоматически вносить операторы, свойства и параметры в текст программы, при наборе которого редактор отображает список подходящих операторов, прототипов функций или значений.

При вводе имени элемента управления появляется раскрывающийся список его свойств. Такая возможность полезна, если программист не знает точно, какие свойства доступны данному элементу управления. Если программист забыл формат оператора или функции, ему будет предоставлена синтаксическая помощь — при вводе правильного ключевого слова или правильного имени функции немедленно под текущей строкой отображается синтаксис оператора или функции.

Можно отмечать строки кода с помощью закладок (bookmarks), чтобы позже, при необходимости вернуться к этим строкам.

Осталось заметить, что каким бы ни был мощным редактор кода, сколько бы он ни выполнял функций за программиста, он

никогда не заменит последнего. Только опытный программист, использующий правила хорошего стиля программирования, может написать правильные, надежные, эффективные, удобные для применения и легко читаемые программы.

РЕЗЮМЕ

* Объектно-ориентированное программирование базируется на трех основных понятиях: инкапсуляции, наследовании, полиморфизме.

* Инкапсуляция — это комбинирование данных с процедурами и функциями, которые манипулируют этими данными.

* Наследование — это возможность использования уже определенных объектов для построения иерархии объектов, производных от них. Каждый из «наследников» наследует описание данных «прародителя» и доступ к методам их обработки.

* Полиморфизм — это возможность определения единого по имени действия (процедуры или функции), применимого одновременно ко всем объектам иерархии наследования, причем каждый объект может «заказывать» особенность реализации этого действия над «самим собой».

* Крупноблочное программирование — это систематическое использование ранее разработанных крупных программных единиц (таких, как классы, подсистемы, или модули) при разработке новых программных систем.

* В объектно-ориентированном программировании объектом называют все, что может иметь имя и может быть рассмотрено как одно целое.

* Каждый объект определен классом (class). Класс используется для создания объектов и определяет их характеристики.

* Объекты обладают свойствами, событиями и методами. Свойства отображают некоторые атрибуты объекта, методы — его действия, а события — это реакции объекта.

* Визуальное программирование предполагает действия: создание интерфейса пользователя, определение свойств для формы и элементов управления, запись соответствующего программного кода (текста программы) в ответ на события, чтобы оживить приложение (чтобы оно работало).

* Современные инструментальные средства разработки приложений для пользователя предлагают разработчику набор мастеров, которые помогут ему в короткий срок спроектировать

нужное программное обеспечение. Мастер разработки автоматизирует процесс создания проекта или его части.

* Мастер разработки приложения представляет собой мощное, но всего лишь вспомогательное средство программирования. Он не может полностью заменить программиста.

* Фундаментом, основой создания интерфейса приложения являются формы.

* Форма — это объект, который обладает свойствами, определяющими внешний вид формы, методами, определяющими ее поведение, и событиями, которые определяют ее взаимодействие с пользователем.Режим формы и ее удобный дизайн представляют собой дружественную среду общения приложения с пользователем.

* Форма содержит элементы управления приложением: поля ввода, меню, командные кнопки, переключатели, флажки, списки, линейки прокрутки, а также диалоговые окна для выбора файла или каталога. Она может содержать объекты OLE.

* При запуске проекта формы становятся окнами, которые и видит пользователь.

* Элементы управления — это объекты, содержащиеся внутри формы и обеспечивающие интерактивный режим работы пользовательского приложения.

* Метки используются для отображения текста на форме, который пользователь не может изменять.

* Текстовые поля являются гибкими элементами управления и применяются как для получения вводимых пользователем данных, так и для отображения текста.

* Флажок показывает, включено или нет определенное условие, и может принимать только два значения типа «да/нет», «правда/ложь», «1/0» и т.п.

* Переключатели предоставляют пользователю выбор из двух или более возможностей. Обычно они работают в группе. При выборе одного, немедленно сбрасываются все другие переключатели в группе.

* Список представляет пользователю перечень возможных вариантов выбора, отображаемых обычно вертикально в виде списка.

* Комбинированное окно совмещает возможности списка и текстового поля. В этом элементе управления пользователь может производить выбор значения либо вводом текста в поле ввода комбинированного окна, либо выбором элемента из его списка.

• Одним из основных способов обеспечения взаимодействия пользователя с приложением является кнопка управления, нажатием которой пользователь может выполнить разнообразные действия.

• Для улучшения дизайна формы применяются так называемые «Графические элементы управления», которые облегчают работу с графикой. Это могут быть изображения (картинки), линии, прямоугольники и т.п.

• Создание интерфейса приложения заключается в размещении элементов управления в форме, в задании для них значений свойств в окне и в написании программ, обрабатывающих события.

Темы для повторения

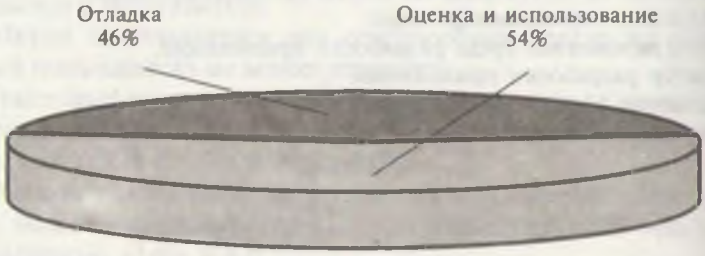
1. Объектно-ориентированное программирование.
2. Понятия объектно-ориентированного программирования: объект, класс, событие, метод, инкапсуляция, наследование, полиморфизм, форма.
3. Типы проблемных систем.
4. Распределение операций по классам.
5. Объектно-ориентированные языки программирования.
6. Визуальное программирование.
7. Интегрированная среда разработки приложения.
8. Мастер разработки приложения.
9. Элементы управления приложением.
10. Создание интерфейса приложения.

Глава 8. **Ошибка**

Под ошибкой в широком смысле слова понимается неправильность, погрешность или неумышленное, невольное искажение объекта или процесса. При этом подразумевается, что известно правильное или неискаженное эталонное состояние объекта, к которому относится ошибка.

Считается, что если программа не выполняет того, что пользователь от нее ожидает, то в ней имеется ошибка.

Важной особенностью процесса выявления ошибок в сложных программах является отсутствие полностью определенной правильной программы-эталона, которому должен соответствовать проверяемый текст. Поэтому нельзя гарантированно утверждать, что возможно написать программу без ошибок.



Распределение выявленных ошибок

Искажения в тексте программ (первичные ошибки) являются элементами, подлежащими корректировке. Однако непосредственно наличие ошибки обнаруживаются по ее вторичным проявлениям. Искажение выходных результатов исполнения программ (вторичная ошибка) вызывает необходимость выполнения ряда операций по локализациии устранению первичной ошибки (отладка программ).

Одним из критериев профессионального мастерства программистов является их способность обнаруживать и исправлять собственные ошибки. Начинающие программисты не умеют этого

Опыт у опытных программистов это не вызывает затруднений. Ошибки в программах делают все. Только программисты разной квалификации делают разные по сложности и количеству ошибки.

На этапе отладки программ выявляются и исправляются много ошибок, но не все... Опыт показывает, что всякая последняя найденная ошибка на самом деле является предпоследней, т.к. программисты не продумывают до конца последствия исправления найденной ошибки.

Убывание ошибок в программном обеспечении и интенсивность их обнаружения не беспредельно. После отладки в течение некоторого времени интенсивность обнаружения ошибок при самом активном тестировании снижается настолько, что разработчики попадают в зону нечувствительности к программным ошибкам и отказам. При такой интенсивности отказов программ трудно прогнозировать затраты времени, необходимые для обнаружения очередной ошибки. Создается представление о полном отсутствии ошибок в программе, о невозможности и нецельности их поиска. Поэтому усилия на отладку сокращаются, и интенсивность обнаружения ошибок еще больше снижается. Этой предельно низкой интенсивности обнаружения ошибок соответствует наработка на обнаруженную ошибку, при которой прекращается улучшение характеристик программного обеспечения на этапах его отладки и испытаний у заказчика.

Ошибку можно отнести к одному из нижеперечисленных классов:

- системные ошибки;
- ошибки в выборе алгоритма;
- алгоритмические ошибки;
- технологические ошибки;
- программные ошибки.

Системные ошибки в большом (сложном) программном обеспечении определяются прежде всего неполной информацией о реальных процессах, происходящих в источниках и потребителях информации.

На начальных стадиях проектирования ПО не всегда удается точно сформулировать целевую задачу всей системы и требования к ней. В процессе проектирования ПО целевая функция системы уточняется и выявляются отклонения от уточненных требований, которые могут квалифицироваться как системные ошибки.

Некачественное определение требований к программе приводит к созданию программы, которая будет правильно решать

неверно сформулированную задачу. В таких случаях, как правило, требуется полное перепрограммирование.

Признаком того, что создаваемая для заказчика программа может оказаться не соответствующей его истинным потребностям, служит ощущение неясности задачи. Письменная регистрация требований к программе заставляет заказчика собраться с мыслями и дать достаточно точное определение требований. Всякие устные указания являются заведомо неполными и часто приводят к взаимному недопониманию.

При автономной и в начале комплексной отладки ПО доля найденных системных ошибок в нем невелика (примерно 10%), но она существенно возрастает (до 35—40%) на завершающих этапах комплексной отладки. В процессе эксплуатации преобладающими являются системные ошибки (примерно 80% всех ошибок). Следует отметить также большое количество команд и групп программ, которые корректируются при исправлении каждой системной ошибки.

Ошибки в выборе алгоритма. В настоящее время накоплен значительный фонд алгоритмов для решения типовых задач.

К сожалению, часто плохой выбор алгоритма становится очевидным лишь после его опробования. Поэтому все же следует уделять внимание и время выбору алгоритма, с тем, чтобы впоследствии не приходилось переделывать каждую программу.

Во избежание выбора некорректных алгоритмов, необходимо хорошо ознакомиться с литературой по своей специальности.

К *алгоритмическим ошибкам* следует отнести прежде всего ошибки, обусловленные некорректной постановкой функциональных задач, когда в спецификациях не полностью оговорены все условия, необходимые для получения правильного результата. Эти условия формируются и уточняются в значительной части в процессе тестирования и выявления ошибок в результатах функционирования программ.

К алгоритмическим ошибкам следует отнести также ошибки связей модулей и функциональных групп программ. Их можно квалифицировать как ошибки некорректной постановки задачи.

Алгоритмические ошибки проявляются в неполном учете диапазонов изменения переменных, в неправильной оценке точности используемых и получаемых величин, в неправильном учете связи между различными переменными, в неадекватном представлении формализованных условий решения задачи в спецификациях или схемах, подлежащих программированию и т. д.

Эти обстоятельства являются причиной того, что для исправления каждой алгоритмической ошибки приходится изменять иногда целые ветви программного обеспечения, т.е. пока еще существенно больше операторов, чем при исправлении программных ошибок.

Алгоритмические ошибки значительно труднее поддаются обнаружению методами формализованного автоматического контроля. Вот почему необходимо тщательным образом продумывать алгоритм прежде, чем транслировать его в программу.

Некоторые программисты проверяют алгоритм следующим образом. Через несколько дней после составления алгоритма они повторно обращаются к описанию задачи и составляют алгоритм заново. Затем сличают оба варианта. Такой шаг на первый взгляд может показаться пустой тратой времени, однако всякая ошибка на уровне алгоритма может в дальнейшем обернуться катастрофой и повлечь основательный пересмотр программы.

Технологические ошибки — это ошибки документации и фиксирования программ в памяти ЭВМ. Они составляют 5—10 % от общего числа ошибок, обнаруживаемых при отладке. Большинство технологических ошибок выявляются автоматически формализованными методами (например, транслятором).

Программные ошибки. Языки программирования — это искусственные языки, созданные человеком для описания алгоритмов. Все предложения таких языков строятся по строгим синтаксическим правилам, обеспечивающим однозначное их понимание, что позволяет поручать расшифровку алгоритма ЭВМ, построенного по правилам семантики.

Синтаксис — это набор правил построения из символов алфавита специальных конструкций, с помощью которых можно составлять различные алгоритмы (программы). Эти правила требуют их неукоснительного соблюдения. В противном случае будет нарушен основной принцип — четкая и строгая однозначность в понимании алгоритма.

Семантика языка — это система правил истолкования построений конструкций. Например, если фрагмент конструкции имеет вид A*(B + C), то правила семантики предписывают, что сначала выполняется сложение величин B и C, а затем результат умножается на величину A. Правила семантики (смысла) конструкций обычно вполне естественны и понятны, но в некоторых случаях их надо специально оговаривать, комментировать.

Таким образом, программы, позволяющие однозначно про-

изводить процесс переработки данных, составляются с помощью соединения символов из алфавита в предложения в соответствии с синтаксическими правилами, определяющими язык, с учетом правил семантики.

Итак, выделяют синтаксические и семантические ошибки.

Под *синтаксическими ошибками* понимается нарушение правил записи программ на данном языке программирования. Они выявляются самой машиной, точнее транслятором, во время перевода записи алгоритма на язык машины. Исправление их осуществляется просто — достаточно сравнить формат исправляемой конструкции с синтаксисом в справочнике и исправить его.

Семантические (смысловые) ошибки — это применение операторов, которые не дают нужного эффекта (например, а—в вместо а+в), ошибка в структуре алгоритма, в логической взаимосвязи его частей, в применении алгоритма к тем данным, к которым он неприменим и т.д. Правила семантики не формализуемы. Поэтому поиск и устранение семантической ошибки и составляет основу отладки.

Программные ошибки по количеству и типам в первую очередь определяются степенью автоматизации программирования и глубиной формализованного контроля текстов программ.

Число программных ошибок зависит также от квалификации программистов, от общего объема комплекса программ, от глубины логического и информационного взаимодействия модулей и от ряда других факторов.

Каждая программная ошибка влечет за собой необходимость изменения команд существенно меньше, чем при алгоритмических и системных ошибках. На этапах комплексной отладки ПО и эксплуатации удельный вес программных ошибок падает и составляет примерно 15 и 30 % соответственно от общего количества ошибок, выявляемых в единицу времени.

Защитное программирование

Защитное программирование — основано на важной предпосылке: худшее, что может сделать модуль, это принять неправильные входные данные и затем вернуть неверный, но правдоподобный результат.

Чтобы разрешить эту проблему, в начале каждого модуля помещаются проверки входных данных на соответствие их свойств атрибутам и диапазонам изменения, на полноту и осмысленность.

Программа может проверять также вводимые данные (сооб-

...ния) пользователем с клавиатуры во время работы приложения. При этом, она должна принимать любое вводное значение (сообщение) и обработать ее соответствующим образом (в данном случае – выдать свое сообщение и попросить пользователя ввести новое, правильное по смыслу значение (сообщение).

При этом требуется разумный подход. Если над входными данными выполнять все мыслимые проверки, то защищающая часть программы может стать настолько сложной, что ее влияние на надежность и эффективность будет не позитивным, а негативным.

Аварийные ситуации внутри программы такие, как переполнение, деление на ноль, извлечение квадратного корня или вычисление логарифма из отрицательного числа и т.д., должны быть обработаны программой. После наступления аварийной ситуации должно быть выдано соответствующее сообщение и выполнена заданная команда. Выполнять же вышеперечисленные операции лучше после проверки аргументов заданным условиям. Это займет немного времени на дополнительное программирование, но значительно повысит надежность программного обеспечения в целом.

Стоимость устранения ошибок

Программирование без ошибок

Если программист верит, что он может писать программы правильно, так оно и будет. Если же он убежден, что написанная им программа непременно содержит ошибки, — они неизбежны.

В первом случае программиста не будет подводить небре
ность, он сможет сосредоточится на исключении всех ошибо
ещё до первого цикла компилирования программы.

Прежде всего, после написания части программы, необ
ходимо сразу ее проверить. Иначе, после написания закончено
блока программы, можно обнаружить целый ряд опечаток ил
логических ошибок, и придётся подвергать программу мно
гократным испытаниям, пока в ней не исчезнут все явны
ошибки.

Итак, если программист ожидает, что программа будет содер
жать ошибки, он даже и не станет пытаться сделать её правильно
с самого начала.

В этом отношении полезны следующие рекомендации:

1. Добивайтесь правильности работы логических узлов прог
раммы посредством последовательного продвижения по блок
схеме алгоритма сверху вниз, пошагового уточнения и попутно
проверки.

2. Старайтесь с самого начала избегать синтаксических оши
бок. Разумеется, вы знакомы с подавляющим большинством
синтаксических конструкций используемого языка програм
мирования, однако, если возникает хоть какое-то сомнение,
обратитесь к синтаксическим правилам вновь, так же как вы
проверяете написанные слова по словарю. Путаница синтак
сических конструкций возникает, если программист знаком со
многими языками программирования.

3. Не допускайте опечаток. Проверяйте текст сразу после
введения в ЭВМ нескольких строк. Не останавливайтесь перед
проведением ещё одной проверки. Многие виды опечаток не
обнаруживаются компилятором и поэтому превращаются впос
ледствии в ошибку.

Наилучший способ организации отладки — это сведение к
минимуму необходимости в ней.

О квалификации программиста судят по числу ошибок,
обнаруживаемых в его программном обеспечении после того, как
оно передано для широкого использования. Поэтому лучше иметь
репутацию медлительного программиста, но создающего хоро
шие и тщательно выверенные программные продукты, чем
репутацию быстро работающего программиста, но разраба
тывающего программные изделия, испещренные ошибками.

Если в нашем распоряжении находится программа, в которой
при тестировании обнаружено и исправлено 10 ошибок, и

программа, в которой не выявлено ни одной ошибки, то мы должны в большей степени полагаться на последнюю программу. Поэтому лучший путь к сохранению уверенности в хорошем качестве программы — недопущение в ней ошибок. Ошибку найти и исправить труднее, чем ее не допустить!

Тестирование программного обеспечения

Имеется серьезное расхождение между тем, как должна вестись разработка программного обеспечения, и тем, как она ведется в действительности. В идеале: четко и ясно сформулированные требования, полные и детальные спецификации, тщательный продуманный проект и тщательная реализация программного обеспечения теоретически создают предпосылки для получения безошибочного программного изделия, пригодного к эксплуатации. Но это далеко от реальности. Обычно большая часть усилий по созданию надежного программного обеспечения падает на фазу тестирования, где принимаются более или менее успешные попытки получения нового программного изделия, правильно обрабатывающего достаточно простые входные данные.

Тестирование — это процесс выполнения программного обеспечения с целью определения места некорректного его функционирования. Оно включает преднамеренное конструирование трудных наборов входных данных, создающих наибольшие возможности для отказа программного продукта. Иногда процесс тестирования сокращают, доведя его до уровня обычной проверки. В результате же неминуемая катастрофа — ненадежное программное обеспечение, недовольный заказчик и т.д.

Основным методом обнаружения ошибок в программном обеспечении является его тестирование. Эффективность тестирования — важнейший фактор, определяющий стоимость и длительность разработки больших программных изделий с заданным качеством. Затраты на тестирование для обнаружения ошибок в программном обеспечении достигают 30—40% общих затрат на его разработку и в значительной степени определяют его качество.

Тестирование — это процесс выполнения программного обеспечения с намерением найти в нем ошибку — довольно необычный процесс (и поэтому трудный), т.к. этот процесс разрушительный.

Цель проверяющего (тестовика) — заставить программное обеспечение сбиться. Он доволен, если это ему удается.

Если программное изделие правильно ведет себя для солид ного набора тестов, нет оснований утверждать, что в нем не ошибок. Просто неизвестно, когда оно не сработает и можно говорить лишь о некотором уровне уверенности в его правиль ности.

Тесты, не способствующие обнаружению ошибок и только подтверждающие корректность функционирования программного изделия, являются не эффективными, т.к. приводят к беспо лезным затратам ресурсов и времени.

Если задача тестовика — продемонстрировать, что в про граммном обеспечении нет ошибок, то подсознание побуждает его выбирать такие тесты, которые с большой вероятностью будут выполняться правильно, и найдет он ошибок не слишком много. Если же его цель — вскрыть наличие ошибок в нем, он найдет значительную их часть!

Необходимо с самого начала не допускать ошибок в про граммном обеспечении. Тогда роль тестирования будет состоять в том, чтобы определить местонахождение немногочисленных ошибок, оставшихся в хорошо спроектированном программном обеспечении.

Тест — это просчитанный вручную или другим способом пример, исходные, промежуточные и конечные результаты которого используются для контроля правильности (живучести) программного обеспечения.

Тест состоит из исходных данных и тех значений (результи тирующих, а иногда и промежуточных), которые должно выдать программное обеспечение при работе по этому тесту. Эти зна чения должны быть записаны в точности в том виде, в котором их должна выдать ЭВМ. Эти значения желательно получить любым путем, не обязательно тем, который реализован в программном обеспечении, т.к. в последнем случае можно не заметить ошибки в алгоритмизации, которые затем не выявит и отладка.

Тесты составляются после разработки алгоритма, но до программирования.

Комплект тестов должен быть таким, чтобы проверить все варианты внешнего эффекта программного изделия и варианты его внутренней работы — алгоритма.

Комплект тестов должен обеспечить прохождение всех ветвей алгоритма по крайней мере по одному разу, а также проверку всех контролирующих предельных и вырожденных случаев. Про верка предельных случаев помогает выявить частые в практике

программирования ошибки, когда цикл недорабатывает или перерабатывает один раз.

В некоторых случаях тестирования рекомендуется производить «миниматоризацию» программы, т.е. разумно сокращать объём данных по сравнению с реальным. Например, создать укороченную базу данных. Проверить матрицу 50×50 вручную невозможно. Поэтому в качестве теста может быть использована матрица 5×5. Точно так же, если некоторая подпрограмма работает в цикле 5 раз, она сможет работать и 105 раз. Правда, при миниматоризации программы могут произойти ситуации: либо существующие в программе ошибки в результате упрощающих изменений станут неявными или временно исчезнут, либо появятся новые ошибки.

Тестовые данные должны подбираться таким образом, чтобы программист был в состоянии вычислить правильный результат вне до начала тестового прогона. Если этого не сделать заблаговременно, то потом очень легко поддаться соблазну считать машинный результат достоверным.

В целом составление тестов — большое искусство, т.к. полностью этот процесс формализации не поддается.

Программные изделия как объекты тестирования имеют ряд особенностей, которые отличают процесс их тестирования от традиционного, применяемого для проверки аппаратуры и других технических изделий.

Для больших программных изделий практически всегда отсутствует полностью определенный и точный эталон для всех тестовых наборов.

В связи с этим для тестирования программного обеспечения в качестве эталона используется ряд косвенных данных, которые не полностью отражают функции и характеристики отлаживаемых программных изделий.

Для больших программных изделий недостижимо исчерпывающее их тестирование, гарантирующее абсолютно полную их проверку.

Поэтому тестирование программного обеспечения проводится в объёмах, минимально необходимых для его проверки в некоторых ограниченных пределах изменения параметров и условий их функционирования.

Ограниченность ресурсов тестирования привела к необходимости упорядочения применяемых методов и конкретных значений параметров с целью получения при тестировании программного обеспечения наибольшей глубины его проверок.

Для ускорения отладки не безразличен порядок пропуска тестов: сначала пропускаются простые тесты...

Простые тесты проверяют начальную цель тестирования — работает ли программное обеспечение вообще.

Тест, который используется для проверки основной ветви программы, должен обнаружить грубые ошибки.

Усложнение тестовых данных должно происходить постепенно.

Процесс тестирования программного обеспечения можно разделить на три этапа:

— проверка в нормальных условиях;
— проверка в экстремальных условиях;
— проверка в исключительных ситуациях.

Каждый из этих трех этапов проверки должен гарантировать получение верных результатов при правильных входных данных и/или выдачу сообщений об ошибках при неправильных входных данных.

Проверка в нормальных условиях. Случаи, когда программное обеспечение должно работать со всеми возможными исходными данными, чрезвычайно редки. Обычно имеют место конкретные ограничения на область изменения данных, в которой программное изделие должно сохранять свою работоспособность. Программное обеспечение выдает правильные результаты для характерных совокупностей исходных данных.

Проверка в экстремальных условиях. Тестовые данные этого этапа включают граничные значения области изменения входных переменных, которые должны восприниматься программным обеспечением как правильные данные.

Типичные примеры — очень большие числа, очень малые числа или отсутствие информации.

Проверка в исключительных ситуациях. Используются исходные данные, значения которых лежат за пределами допустимой области их изменения.

Наихудшая ситуация когда программное обеспечение воспринимает неверные данные как правильные и выдаёт неверный, но правдоподобный результат.

Программное обеснечение должно самоотвергать любые данные, которые оно не в состоянии обрабатывать верно.

Пример тестов. Пусть требуется вычислить длину диагонали параллелепипеда

$$d = \sqrt{a^2 + b^2 + c^2}.$$

Необходимо сформировать тестовые данные для нормальных, экстремальных и исключительных условий.

Стороны параллелепипеда			Примечание
1	1	1	Хороший начальный тест $d=\sqrt{3} = 1,7320508$
1	2	3	Нормальные условия $d=\sqrt{14} \approx 3,74165773$
0	0	0	Результат должен быть равен нулю
0	1	2	Не параллелепипед. Что произойдет?
1	0	3	Неверные данные
3	1	0	Неверные данные
1	-6	3	Неверные данные
A	B	C	Неверные данные

Создание программного обеспечения без ошибок — трудоемкая работа. Поэтому разумно всегда настаивать на выделении необходимого времени для обеспечения надлежащего качества тестирования программного обеспечения. Нарушение этого условия приводит к неполному тестированию и, как следствие, — к ненадежному программному продукту, в то время как его соблюдение дает хорошие результаты.

Различают еще так называемые «Альфа»- и «Бета»-тестирования.

Первоначальное альфа-тестирование программного обеспечения, как правило, осуществляют разработчики в «лабораторных» условиях — последняя возможность разработчиков исправить все обнаруженные ошибки прежде, чем система будет передана в эксплуатацию конечным пользователям.

Поэтому лабораторное тестирование должно проходить максимально полно. Если альфа-тестирование проведено некачественно, общий процесс тестирования может занять продолжительное время, так как исправление ошибок, выявленных на последующих стадиях тестирования, занимает значительно больше времени из-за невозможности исправления их «на лету».

Любые обнаруженные проблемы должны протоколироваться, чтобы хронология проблем и их устранения были доступны при

возникновении последующих вопросов о ранее существовавших проблемах.

Бета-тестирование (опытная эксплуатация) — это следующая фаза общего тестирования, при которой программное обеспечение поставляется ограниченному кругу конечных пользователей для более жесткого тестирования.

Хорошо известно, что пользователи иногда задействуют программное обеспечение не по его целевому назначению. Поэтому пользователи часто будут находить ошибки в тех местах программы, где недели лабораторных испытаний не нашли ничего. Это необходимо ожидать и не отрицать возможности возврата к предыдущей фазе – лабораторному тестированию. Кстати, в данных случаях часто помогают протоколы обнаруженных и фиксированных ошибок.

Если предыдущие стадии тестирования завершены успешно, то приемочный тест становится простой формальностью. Используя информацию о том, что все обнаруженные ошибки уже исправлены, приемочный тест просто подтверждает, что никаких новых проблем не обнаружено, и программное обеспечение готово для выпуска.

Очевидно, что чем больше существует реальных или потенциальных пользователей разработанного программного продукта, тем более важным является приемочный тест. Конечно, если предыдущие стадии не прошли успешно, то приемочный тест является единственной возможностью предотвратить затраты на изменение и дополнение поставляемого продукта.

Методы тестирования

Восходящее тестирование. Программное обеспечение собирается и тестируется снизу вверх.

Только модули самого нижнего уровня тестируются изолированно, автономно.

Затем тестируются модули, непосредственно вызывающие их, которые тестируются не автономно, а вместе с уже проверенными модулями. И так, пока не будет достигнута вершина.

В последнюю очередь тестируется программное обеспечение в целом.

Нисходящее тестирование. Программное обеспечение собирается и тестируется сверху вниз.

Изолированно тестируется только головной модуль.

Затем с ним соединяются один за другим модули, непосредственно вызываемые им, и тестируется полученная комбинация. Так до тех пор, пока не будут собраны и проверены все модули.

Если вызываемый для тестирования модуль еще не существует, то для имитации функций недостающих модулей программируются модули-«заглушки».

При модификации нисходящего подхода требуется, чтобы каждый модуль перед подключением его к комплексу программ прошел автономное тестирование.

Метод большого скачка. Каждый модуль тестируется автономно.

Затем они интегрируются в систему все сразу.

Заметим, что при восходящем и нисходящем подходах каждый раз подключается только один модуль, и если обнаружится ошибка, подозрение в первую очередь падает на последний добавленный модуль.

Так что метод большого скачка значительно усложняет отладку программного обеспечения и приемлем лишь для маленьких, хорошо спроектированных программ.

Метод «сэндвича». Одновременно начинают восходящее и нисходящее тестирование, собирая программу как снизу, так и сверху и встречаясь где-то в середине.

Точка встречи зависит от тестируемого программного обеспечения и должна быть заранее определена при изучении ее структуры.

Это разумный подход к интеграции больших программных изделий, таких как операционная система или пакет прикладных программ.

Аксиомы тестирования

1. Хорош тот тест, который позволяет обнаружить ошибку, а не тот, который демонстрирует правильную работу программы.

2. Тесты, не способствующие обнаружению ошибок и только подтверждающие корректность функционирования программного обеспечения, являются неэффективными, т.к. приводят к бесполезным затратам ресурсов и времени.

3. Одна из самых сложных проблем при тестировании — решить, когда нужно его закончить.

4. Невозможно тестировать свою собственную программу.

Тестирование должно быть в высшей степени разрушительным процессом, но программист не может относиться к своей программе как разрушитель (психологические причины, жесткий график, давление коллектива и т.д.).

5. Необходимая часть всякого теста — описание ожидаемых выходных данных или результатов. Ожидаемые результаты нужно определять заранее.

6. Избегайте невоспроизводимых тестов, не тестируйте «на лету».

7. Никогда не используйте тестов, которые тут же выбрасываются. Более того, тесты следует документировать и хранить в такой форме, чтобы каждый мог использовать их повторно.

8. Готовьте тесты, как для правильных, так и для неправильных входных данных.

9. Детально изучите результаты каждого теста.

10. По мере того, как число ошибок, обнаруженных в некоторых компонентах программного обеспечения увеличивается, растет также относительная вероятность существования в них необнаруженных ошибок.

11. Поручайте тестирование самым опытным и способным программистам.

12. Считайте тестируемость ключевой задачей вашей разработки.

13. Никогда не изменяйте программу, чтобы облегчить ее тестирование.

14. Тестирование, как почти всякая другая деятельность, должно начинаться с постановки целей.

15. Проект программного обеспечения должен быть таким, чтобы каждый модуль подключался к нему только один раз.

16. Для сокращения календарного времени отладки все тесты пропускаются в один выход на ЭВМ независимо от результатов выполнения каждого теста, а затем они обрабатываются.

17. Тест удаляется из дальнейшей работы, если он отработал правильно, т.к. его повторный пропуск не дает ничего нового.

18. Тест возвращается в работу, если вносились изменения в блоки, работающие при этом тесте.

19. Составляя календарный план разработки программного обеспечения, разумно всегда настаивать на выделении необходимого времени для обеспечения надлежащего качества его тестирования.

10. Если программное обеспечение правильно ведет себя для полного набора тестов, нет оснований утверждать, что в нем нет ошибок.

Отладка программного обеспечения

Под *отладкой* понимается процесс, позволяющий получить программное обеспечение, функционирующее с требующимися характеристиками в заданной области входных данных.

Отладка не является разновидностью тестирования, хотя слова «отладка» и «тестирование» часто используют как синонимы. Под ними подразумеваются разные виды деятельности:

— тестирование — деятельность, направленная на обнаружение ошибок;

— отладка направлена на установление точной природы известной ошибки, а затем — на исправление этой ошибки;

— результаты тестирования являются исходными данными для отладки.

Эти два вида деятельности очень тесно связаны и поэтому они обычно рассматриваются совместно.

В результате отладки программное обеспечение должно соответствовать определенной фиксированной совокупности правил и показателей качества, принимаемой для него за эталонную. Иными словами: отладка — это этап разработки, на котором устраняются недостатки только что созданного программного обеспечения.

Основная задача отладки в целом состоит в завершении разработки всего программного обеспечения и в доведении его характеристик до значений, заданных требованиями технического задания (соглашения о требованиях). При этом ПО должно гарантированно удовлетворять всем требованиям не только в диапазоне типичных условий его функционирования, но и при предельных, критических сочетаниях значений всех параметров. Это обеспечивает надежность функционирования ПО при разнообразных произвольных, в том числе, искаженных сочетаниях входных данных.

По оценкам специалистов, в общем времени разработки программного обеспечения, отладка занимает от 50 % до 90% (зависит от результатов проведения предыдущих этапов).

Отладку можно разделить на синтаксическую и семантическую.

Отладка синтаксиса обычно не вызывает трудностей и требует

только аккуратности. Результаты синтаксически правильной программы сравниваются с тестовыми, их несовпадение является признаком наличия семантической ошибки. Это сравнение надо выполнять очень скрупулезно: даже лишний пробел между двумя значениями может дать ключ к поиску ошибки.

В отличие от синтаксической ошибки, *семантическая ошибка не формализуема* и поэтому она составляет основу отладки.

Отладка готовой части (ветви) программного обеспечения может быть начата ранее, чем будет написан весь программный продукт, в результате чего сокращается время разработки ПО, т.е. экономится календарное время его разработки. Временно отсутствующие блоки либо оставляются пустыми, либо заменяются печатью сообщения о том, что программное обеспечение не рассчитано на такие данные, или пишутся временные программные заглушки.

Инструменты отладки программного обеспечения

Основными инструментами отладки служат тесты и отладочные печати.

Тест — это специально подобранные исходные данные в совокупности с теми результатами, которые должно выдавать программное обеспечение при обработке этих данных. Несовпадение результатов программного обеспечения с результатами тестов — признак наличия в нем ошибки.

Но иногда и неправильное программное обеспечение может по нескольким тестам дать правильный результат, поэтому необходимо контролировать и промежуточные результаты, чтобы не упустить взаимное уничтожение ошибок в данном варианте работы программного обеспечения.

Для такого контроля и локализации ошибок в программу вставляются отладочные печати, которые выдают ее промежуточные результаты. Эти печати позволяют проследить логический след программы (где она «ходила») и ее арифметический след (что она вычисляла).

Процесс обнаружения ошибок в программе характеризуется выявлением в ней двух мест: точки обнаружения и точки происхождения ошибки. Точка обнаружения служит отправным пунктом для поиска точки происхождения (место в программе, где возникают условия для появления ошибки).

Основным приемом обнаружения ошибок в программе является фиксация ошибок.

Фиксация ошибок — это работа только с тестами и выданными программой результатами, не заглядывая в схему и в текст программы. Основная задача — по возможности более четко ответить на вопрос «что случилось?». Чем более точно будет сформулирована суть ошибки, тем легче будет ее найти. При этом всегда следует помнить, что несовпадение результатов программы с тестовыми данными может быть из-за ошибки в тесте.

Поиск точек обнаружения и происхождения ошибок в программе осуществляется отладочными печатями.

Любую программу можно разделить на несколько частей.

Основной принцип контроля программы: каждая часть должна «Расписаться» в получении и сдаче управления и предъявить полученные и вычисленные ею значения. Так как результаты одной части программы являются исходными данными для другой, то достаточно ставить печати на переходах из одной части в другую.

Бывает полезным сразу после ввода данных их распечатывать (контроль вводимой информации или эхо-проверка). После входа в процедуру также можно распечатывать значения входных параметров, а после выхода — выходных, т.к. при передаче параметров часты ошибки.

Нежелательно вставлять отладочные печати во внутренних циклах, т.к. они будут работать очень много раз, расходуя бумагу и время.

Если же все-таки необходимо вставлять печать во внутренний цикл, то это делают с условием: IF I< K THEN WRITELN.., где K — константа; эта печать работает только при первых «K» проходах тела цикла.

Каждый оператор отладочной печати должен содержать идентифицирующую часть, чтобы можно было определить, какой оператор печати работал, т.е. получить логический след; их оформить в отдельные строки, чтобы их затем было легко удалить. Для удобства обнаружения отладочной печати в тексте программы ее рекомендуется каким-либо способом выделять (например: (****)WRITELN... (****) или в конце оператора писать: отладка).

Если в отладочных печатях используются циклы, то в качестве параметров этих циклов надо использовать переменные, которые не используются в программе, чтобы не испортить какое-нибудь нужное значение.

По вопросу движения отладочных печатей можно рекомендовать:

- печать вводимых данных остается на все время: ее можно рассматривать как заголовок — единственный способ контроля введенных значений;

- печать параметров процедур удаляется после первого же раза, когда она дала правильный результат при работе всех операторов вызова этой процедуры;

- печать в какой-либо части программы удаляется после того, как эта часть правильно отработала на всех тестах;

- если программа хранится в библиотеке, то отладочные печати лучше не удалять, а превращать в комментарии;

- если вносились изменения в часть программы, из которой удалены печати, то достаточно вставить одну печать после этой части: если она покажет ошибку, то надо проверить те операторы, которые менялись;

- дополнительные печати вставляются, если не удается обнаружить ошибку; этими печатями контролируются значения переменных, участвующих в формировании неверного результата.

К инструментам отладки можно отнести также системные средства ЭВМ — трансляторы и компиляторы с языков программирования, операционную систему, другие программные средства отладки.

Трансляторы, например, указывают на место прерывания отлаживаемой программы, по своему описывают характер ошибки, что, конечно, способствует ведению отладки программ.

Лучшие результаты отладки программы все же дают универсальные ручные методы, пригодные для любых языков программирования.

В одном эксперименте одна группа программистов обнаруживала ошибки с помощью ЭВМ, другая — вручную. Обе группы на обнаружение ошибок затратили примерно одинаковое время.

Методы отладки программного обеспечения

Для локализации и установки точной природы ошибки в программе используют статические и динамические методы отладки программ.

К *статическим методам* относятся методы отладки, при которых не требуется выполнение отлаживаемой программы на ЭВМ. Они обычно требуют больших усилий от программиста и незначительных затрат машинного времени. Они универсальны и

методы для отладки программ, написанных на любом языке программирования и используемых на любой ЭВМ.

Статические методы включают:

— ручную прокрутку программы;

— прокрутку программы программными анализаторами (например, компилятором); автоматизированный анализ программы в этом случае проводится без выполнения ее на ЭВМ и поэтому попадает в категорию «статических»;

— коллективную проверку программ;

— проверку программы программистом-технологом с целью выявления и исправления в ней технологических ошибок.

Экспериментально установлено, что в программах ручными методами удается обнаруживать от 30 до 70 % программных и алгоритмических ошибок из общего числа ошибок, выявленных при отладке. При этом одновременно осуществляется доработка программ с целью улучшения их структуры, логики обработки данных и для снижения сложности последующего автоматизированного тестирования на ЭВМ.

Динамические методы связаны со значительным расходом машинного времени и, возможно, не меньшими затратами труда программиста. В этом случае отладка программ происходит совместно с их выполнением на ЭВМ. Динамические методы отладки программ, как правило, привязаны к конкретной ЭВМ и к конкретному транслятору (компилятору).

К динамическим методам относятся:

— тестирование;

— поиск ошибок с использованием системных средств;

— отладка программы в интерактивном режиме.

Важнейшее правило отладки: не делать следующего выхода на ЭВМ, пока не будет разобрана каждая найденная ошибка. Из этого правила существует единственное исключение: если найдены 5—6 ошибок, которые не дают эффекта, то можно сделать новый выход на машину (устранив эти ошибки), чтобы получить эффект в чистом виде (если он есть),поскольку наложение нескольких ошибок иногда может дать самый неожиданный результат.

Если программист исчерпал все возможности поиска ошибки, но не нашел ее, то как крайнее средство, можно сделать выход на машину, ничего не изменив в программе, но добавив печати, выдающие значения идентификаторов, участвующих в формировании неверного значения. И снова произвести анализ полу-

ченных результатов. Квалификация программиста в области отладки определяется тем, сколько информации об ошибках он сможет получить из одной выдачи ЭВМ.

Ручная прокрутка программы

Ручная прокрутка программы, или проверка программы за столом является универсальным и мощным, но весьма трудоемким средством проверки программы на ошибку и принадлежит к наиболее «старым» традициям в программировании. Этот метод характеризует неавтоматизированную, ручную деятельность программиста.

К этому методу наиболее часто относят:

— анализ листинга программы на наличие ошибок;

— выполнение расчетов для проверки точности результатов;

— «игру за машину», т.е. выполнение программы вручную вместо ЭВМ.

Эти три процесса позволяют уяснить и проверить логику программы (ошибки алгоритмизации) и поток данных.

Ручная прокрутка программы наиболее эффективна, когда ошибка локализована достаточно точно, но почему-то не выявляется другими приемами.

Обычно прокручивают небольшую часть программы — проверяется путь, который прошла программа при тех исходных данных, которые ей были даны до того места, где проявилась ошибка. Затем проводят обратное отслеживание идентификаторов. Обычно обнаруживаются ошибки алгоритмизации типа неучтенного особого случая.

Обратное отслеживание идентификаторов — это ответ на вопрос «*где случилось?*» и отчасти «*почему случилось?*». Ошибка проявляется между последней правильно отработавшей и первой неправильно отработавшей печатью. Этот фрагмент программы является стартовой точкой поиска ошибки и поэтому подлежит исследованию, хотя сама ошибка чаще всего может находиться в другом месте. При отслеживании идентификаторов следует постоянно контролировать описания, т.к. неправильное описание переменных вызывает преобразование присваиваемых значений.

Во время ручной прокрутки программы выписываются все идентификаторы, работающие в данном месте программы и их значения. При каждом изменении значения идентификатора старое значение зачеркивается и на его месте или рядом записывается новое значение. В частности, будут обнаружены неоп-

неделенные переменные, лишние переменные и неверные преобразования их типов. Для массивов можно нарисовать рамку, содержащую нужное количество позиций. Тогда при ошибке типа выхода индекса за границу массива это будет сразу обнаружено, т.к. очередное значение придется брать или записывать за границами этой рамки.

Во время анализа листинга программы необходимо придерживаться концепций контроля утверждений. Утверждение — это предложение, устанавливающее некоторый факт. В программном смысле — это установление истины в процессе выполнения программы.«Локальное» утверждение должно быть истинным в точке его описания, «глобальное» утверждение — в любой точке программы.

Логика поиска ошибки

Обратное отслеживание идентификаторов — наиболее частый прием поиска ошибок — операция довольно трудоемкая, т.к. в место проявления ошибки программа могла прийти несколькими путями, и попытка отследить по всем ветвям значения всех прямых и косвенных участников формирования неверного результата приводит к лавинообразному нарастанию количества подлежащих отслеживанию идентификаторов.

Логические рассуждения помогают отсечь некоторые варианты выполнения программы:

— на выдаче имеем ситуацию 1;

— чтобы она случилась, надо чтобы было 2 и 3 и либо чтобы было 4;

— ситуация 2 не может быть, т.к. .... ; чтобы было 4, надо ....

На каждом шаге такого анализа приходится делать коротенькие ручные программные прокрутки или короткие отслеживания одного-двух идентификаторов.

Логика рассуждения на каждом отдельном шаге обычно сводится к одному из двух заключений:

1) программа прошла по ветви N, т.к. переменная X получила (или изменила) значение x, а это единственная ветвь, где это может быть...;

2) программа заведомо не проходила по ветви N, т.к. на этой ветви переменная X должна получить (или изменить) значение x, чего не наблюдалось.

Подобный анализ двух-трех переменных нередко позволяет

определить единственный вариант выполнения программы даже в сильно ветвленных программах.

Если отладочные печати не позволяют достаточно точно локализовать ошибку, то особого внимания требуют ветви программы, которые работают впервые или в необычных условиях.

Ветви, работавшие впервые, определить легко.

Понятие «необычные условия» формализации не поддается, в этой ситуации надо постараться понять, чем отличается тест с точки зрения каждого блока от тех тестов, которые прошли успешно.

Фрагмент программы правильно выполняется первый раз и неправильно — в последующие разы — типичная ошибка в начальных присвоениях.

Неправильно обрабатывался первый или последний элемент — наиболее вероятная ошибка — не хватает 1 или лишняя 1 в начальном или в конечном значении параметра цикла.

Отладка программного обеспечения в интерактивном режиме

Отладка в интерактивном режиме — это процесс нахождения и исправления ошибок в программе, при котором обеспечивается возможность вмешательства человека в ход выполнения программы.

В этом случае программист может выполнить за короткий сеанс работы за дисплеем то, что обычно требует нескольких выходов на ЭВМ в пакетном режиме.

С другой стороны: у программиста остается меньше времени для выполнения остальных работ таких, как оформление документации или выполнение других проектов, в результате чего производительность его труда не повышается с той же интенсивностью, с какой сокращается время поиска ошибок.

Отлаживая программу интерактивно, некоторые программисты стремятся сразу принять решение о необходимых корректировках программы, поскольку они могут быть сделаны «немедленно». Эти «быстрые» корректировки программы уже через короткое время окажутся неправильными («скороспелыми»). Всегда следует неторопливо и вдумчиво просмотреть все варианты решения корректировок программы. С целью определить как эти корректировки повлияют на другие части программы?

Другая беда отладки программы в интерактивном режиме это

«экспериментальные» исправления типа: «посмотрим, что получится, если...». Их необходимо избегать, т.к. они только в пустую расходуют время. Нужно тщательно исследовать, что именно в программе выполняется верно, а что — неверно, для выработки одной или нескольких гипотез о природе ошибки, и исправить ее.

Программист, в распоряжении которого предоставлен экранный терминал, неистово меняет строки программы одну за другой, что приводит к небольшому результату. Он может вносить беспорядочные, плохо продуманные изменения, которые приведут только к новым ошибкам. В таких случаях лучше встать из-за терминала, распечатать текст программы, позволяющий полностью ее обозреть (для полного понимания того, что происходит с программой), и провести проверку программы за столом.

Автономная отладка частей программы

Автономная отладка частей программы относится к ускоренным методам отладки программ и происходит ценой дополнительных затрат на программирование.

Программа разбивается на относительно независимые части и каждая часть отлаживается отдельно от других, причем все части отлаживаются одновременно, что позволяет сократить время почти во столько раз, сколько частей удалось выделить. Поскольку части программы все же как-то связаны между собой, то каждую часть необходимо дополнить блоками, имитирующими работу отдельных частей(это и есть те самые дополнительные затраты). В простейшем случае это блоки, выводящие заранее просчитанные данные.

Коллективная проверка программы

Коллективная проверка — это процесс, при котором бригада программистов тщательно просматривает программу или ее часть в порядке инспекции (контроль программы, сквозной просмотр и т.п.).

Целью коллективной проверки является повышение надежности программы.

Бригада программистов сообща проверяет программу с разных точек зрения участников проверки, выискивая ошибки, которые могли ускользнуть при менее широком подходе.

Коллективная проверка программ обладает существенным преимуществом: программисты стараются выполнить свою работу

лучше, зная, что программу будут критически рассматривать их коллеги. Кроме того, программисты совершенствуют свои методы и стиль работы в результате обмена мнениями и опытом.

Однако здесь есть и свои трудности. Проверка программы — часто нудное, запутанное дело. Для эффективной проверки необходимы методичность и скрупулезность. По этой причине заинтересовать ее участников весьма сложно, однако стимулировать их чрезвычайно важно, т.к. при отсутствии достаточной внимательности проверка программы теряет смысл.

Кроме того, это дорогой метод. Опыт показывает, что в час может быть проверено около ста операторов исходного текста, а внимание участников через час работы рассеивается.

Для программы большого объема стоимость и длительность коллективной проверки могут стать непреодолимым препятствием. Поэтому в таких обстоятельствах целесообразнее проверять ключевые участки программы, а остальные — выборочно.

В группу контроля обычно входят от 3 до 5 специалистов различной квалификации. Во главе группы должен быть высококвалифицированный программист, ответственный за группу программ, в которую входит данный модуль. Для контроля документов и структурного построения модуля необходимо участие технолога по программированию. Руководство не должно подключаться к этому процессу во избежание проверки программиста, а не его программы.

Тестовые данные заранее подготавливаются разработчиком программы с целью охватить основные маршруты ее исполнения и могут дополняться членами группы в процессе анализа.

Выявленные ошибки желательно устранять до следующего заседания группы и продолжать следующий сквозной просмотр после краткого комментирования проведенных корректировок. В результате подготавливаются и уточняются состав тестов и набор эталонов для тестирования на ЭВМ с исполнением программ.

Система автоматизации отладки программного обеспечения

Поиск некоторых ошибок, содержащихся в структуре данных и, в особенности, в логической структуре программы, можно автоматизировать. Автоматизированный анализ программы на предмет выявления ошибок в ней производится без ее выполнения на ЭВМ и поэтому попадает в категорию «статических».

Система автоматизации отладки программного обеспечения (САО ПО) представляет собой совокупность программных

средств, предназначенных для автоматизации процессов установления факта правильности функционирования разработанных программных изделий, а также для обнаружения, локализации и устранения ошибок в алгоритмах и программах.

САО ПО выполняет следующие функции:

— контроль правильности тестов и заданий на отладку, составленных на входных языках отладки, и выдача информации о месте и характере ошибок;

— выполнение отладочных заданий при помощи системы трансляции заданий, тестов и их исполнения;

— моделирование и интерпретация системы команд управляющей ЭВМ в тех случаях, когда отладка программного обеспечения выполняется на вычислительной машине с иной системой команд;

— выдача оператору результатов отладки и необходимых промежуточных данных на языке отладки после их предварительной обработки;

— корректировка отлаживаемой программы по заданию оператора с целью исправления обнаруженных ошибок.

Ниже перечислены примеры ошибок, которые можно обнаружить автоматизирующими анализаторами программ.

1. Переменные не описаны или описаны неправильно.

2. Переменные используются прежде, чем им было присвоено значение, или присвоенное значение никогда не используется.

3. Используются недопустимые языковые формы (например, арифметика с разнотипными переменными).

4. Нарушение соглашений о наименованиях (переменных, подпрограмм, меток операторов...).

5. Переусложненная структура (циклы или условные операторы слишком глубоко или неправильно вложены).

6. Проверка аргументов подпрограммы (соответствие формального и фактического значений).

7. Противоречия в дереве вызываемых подпрограмм (непредусмотренные циклы).

8. Противоречивость набора глобальных данных (общих блоков).

9. Невыполнимые условия.

10. Потеря управления (например, оператором GO TO передано управление на несуществующий оператор).

11. Незамкнутая логика.

12. Ошибочная логика (например, потенциально бесконечные циклы).

РЕЗЮМЕ

* Под ошибкой в широком смысле слова понимается неправильность, погрешность или неумышленное, невольное искажение объекта или процесса. При этом подразумевается, что известно правильное или неискаженное эталонное состояние объекта, к которому относится ошибка.

* Считается, что если программа не выполняет того, что пользователь от нее ожидает, то в ней имеется ошибка.

* Искажения в тексте программ (первичные ошибки) являются элементами, подлежащими корректировке. Однако непосредственно наличие ошибки обнаруживается по ее вторичным проявлениям — по искажению выходных результатов исполнения программ.

* Ошибку можно отнести к одному из нижеперечисленных классов:системные ошибки, ошибки в выборе алгоритма, алгоритмические ошибки, технологические ошибки, программные ошибки.

* Некачественное определение требований к программе приводит к созданию программы, которая будет правильно решать неверно сформулированную задачу.

* Исправление синтаксических ошибок осуществляется просто — достаточно сравнить формат исправляемой конструкции с синтаксисом в справочнике и исправить его.

* Семантическая ошибка не формализуема и поэтому она составляет основу отладки.

* Защитное программирование основано на важной предпосылке: худшее, что может сделать модуль, — это принять неправильные входные данные и затем вернуть неверный, но правдоподобный результат.

* Если программист убежден, что он может писать программы правильно, так оно и будет. Если же он убежден, что написанная им программа непременно содержит ошибки, — они неизбежны.

* Самый лучший способ организации отладки — это сведение к минимуму необходимости в ней.

* О квалификации программиста судят по числу ошибок, обнаруживаемых в его программном обеспечении после того, как оно передано для широкого использования.

* Лучше иметь репутацию медлительного программиста, но создающего хорошие и тщательно выверенные программные продукты, чем репутацию быстро работающего программиста, но

разрабатывающего программные изделия, испещренные ошибками.

* Ошибку найти и исправить труднее, чем ее не допустить.

* Тестирование — это процесс выполнения программного обеспечения с целью определения места некорректного его функционирования.

* Тест — это просчитанный вручную или другим способом пример, исходные, промежуточные и конечные результаты которого используются для контроля правильности (живучести) программного обеспечения.

* Тесты составляются после разработки алгоритма, но до программирования.

* Если программное изделие правильно ведет себя для солидного набора тестов, еще нет оснований утверждать, что в нем нет ошибок.

* Тесты, не способствующие обнаружению ошибок и только подтверждающие корректность функционирования программного изделия, являются неэффективными, т.к. приводят к бесполезным затратам ресурсов и времени.

* Тест состоит из исходных данных и тех значений (результирующие, а иногда и промежуточные), которые должно выдать программное обеспечение при работе по этому тесту.

* Комплект тестов должен обеспечить проверку всех вариантов внешнего эффекта программного изделия и вариантов его внутренней работы алгоритма.

* Комплект тестов должен быть таким, чтобы все ветви алгоритма были пройдены по крайней мере по одному разу, а так же проверены все контролирующие предельные и вырожденные случаи.

* Составление тестов — это искусство, т.к. полностью этот процесс формализации не поддается.

* Простые тесты проверяют начальную цель тестирования, т.е. работает ли программное обеспечение вообще. Тест, который используется для проверки основной ветви программы, должен обнаружить грубые ошибки. Усложнение тестовых данных должно происходить постепенно.

* Процесс тестирования программного обеспечения можно разделить на три этапа: проверка в нормальных условиях, проверка в экстремальных условиях, проверка в исключительных ситуациях.

* Под отладкой понимается процесс, позволяющий получить

программное обеспечение, функционирующее с нужными характеристиками в заданной области входных данных.

* Отладка не является разновидностью тестирования:

— тестирование — это деятельность, направленная на обнаружение ошибок;

— отладка направлена на установление точной природы известной ошибки, а затем — на исправление этой ошибки;

— результаты тестирования являются исходными данными для отладки.

* Отладка — это этап разработки, на котором устраняются недостатки только что созданного программного обеспечения.

* Процесс обнаружения ошибок в программе характеризуется выявлением в ней двух мест: точки обнаружения и точки происхождения ошибки.

* Для локализации и установки точной природы ошибки в программе используют статические и динамические методы отладки программ.

* К статическим относятся методы отладки, при которых не требуется выполнение отлаживаемой программы на ЭВМ.

* К динамическим относятся методы отладки, при которых отладка программ происходит совместно с их выполнением на ЭВМ.

Они, как правило, привязаны к конкретной ЭВМ и к конкретному транслятору (компилятору).

* Ручная прокрутка программы, или проверка программы за столом является универсальным и мощным, но весьма трудоемким средством проверки программы на ошибку.

* Отладка в интерактивном режиме — это процесс нахождения и исправления ошибок в программе, при котором обеспечивается возможность вмешательства человека в ход выполнения программы.

* Коллективная проверка — это процесс, при котором бригада программистов тщательно просматривает программу или ее часть в порядке инспекции (контроль программы, сквозной просмотр и т.п.).

* Целью коллективной проверки является повышение надежности программного обеспечения.

* Для контроля документов и структурного построения модуля необходимо участие технолога по программированию.

* Логика рассуждения на каждом отдельном шаге обычно сводится к одному из двух заключений:

— программа прошла по ветви N, т.к. переменная X получила (или изменила) значение x, а это единственная ветвь, где это может быть...

— программа заведомо не проходила по ветви N, т.к. на этой ветви переменная X должна получить (или изменить) значение x, чего не наблюдалось.

Темы для повторения

1. Ошибка. Классы ошибок.
2. Первичные и вторичные ошибки.
3. Синтаксические и семантические ошибки.
4. Защитное программирование.
5. Программирование без ошибок.
6. Тестирование программного обеспечения.
7. Понятие теста. Комплект тестов. Пример теста.
8. Три этапа процесса тестирования программного обеспечения.
9. Методы тестирования.
10. «Альфа» и «Бета» тестирования программного обеспечения.
11. Аксиомы тестирования.
12. Отладка программного обеспечения.
13. Статические и динамические методы отладки программного обеспечения.
14. Система автоматизации отладки программного обеспечения.
15. Инструменты отладки.
16. Методы отладки.
17. Ручная прокрутка программы.
18. Логика поиска ошибки.
19. Отладка программного обеспечения в интерактивном режиме.
20. Автономная отладка частей программы.
21. Коллективная отладка частей программы.

Глава 9. Оценка (испытания) программного обеспечения

На этапе оценки (испытания) качества готовое программное обеспечение подвергается строгому системному испытанию со стороны группы лиц, обычно не являющихся разработчиками.

Это делается для гарантии, что готовое программное обеспечение удовлетворяет всем требованиям пользователя и спецификациям, может быть использовано в среде пользователя, свободно от каких-либо дефектов и содержит необходимую документацию, которая точно и полно описывает программный продукт.

Фаза оценки начинается как только все компоненты (модули) отлажены и собраны вместе в одно целое, т.е. после полной отладки готового программного продукта.

Она заканчивается после получения подтверждения, что программное обеспечение прошло все испытания и готово к эксплуатации.

Для программ, создаваемых на уровне продукции производственно-технического назначения и отчуждаемых от разработчика, испытания являются одним из важнейших этапов их жизненного цикла, на котором проверяются и фиксируются достигнутые показатели качества программного обеспечения.

Цель испытаний — определение степени соответствия созданного программного обеспечения техническому заданию (соглашению о требованиях), полученному от заказчика.

За относительно короткий период приемо-сдаточных испытаний трудно провести достаточно полное тестирование, демонстрирующее достигнутое качество большого программного изделия. Поэтому в техническом задании целесообразно задавать условия поэтапной проверки основных компонентов программного обеспечения в процессе его разработки.

В этом случае испытатель получает возможность поэтапно и глубоко знакомиться с создаваемым программным продуктом и подготовиться к его испытаниям. Одновременно уточняются и конкретизируются требования к программному обеспечению и

методика его тестирования на завершающих приемо-сдаточных испытаниях.

Для всесторонней проверки программного обеспечения его опытный образец подвергается испытаниям со стороны:

— главного конструктора (предварительные испытания);

— заказчика-пользователя с участием разработчиков (совместные испытания).

Предварительные испытания программного обеспечения

На *предварительных испытаниях программного обеспечения* производится, по существу, такое же тестирование, что и на его совместных испытаниях, только в меньшем объеме. Эти проверки оформляются документально и являются основанием для предъявления программного обеспечения заказчику на его совместные испытания.

Любые испытания ограничены допустимым объемом проверок и длительностью работы комиссии и поэтому не могут гарантировать всестороннюю проверку изделия. Поэтому, после предварительных испытаний программное обеспечение целесообразно на некоторое время передать на опытную эксплуатацию в типовых условиях («Бета»-тестирование),что позволит более глубоко оценить эксплуатационные характеристики созданного программного продукта и устранить многие в нем ошибки.

Результаты опытной эксплуатации программного обеспечения могут учитываться при совместных испытаниях для их сокращения.

В жизненном цикле программного обеспечения можно выделить следующие виды испытаний:

— опытного образца программного обеспечения на полное соответствие его требованиям технического задания (соглашения о требованиях);

— рабочей версии программного обеспечения, адаптированной к условиям конкретного применения;

— версии модернизированного программного обеспечения при его сопровождении.

Задача испытателей и заказчика при планировании испытаний программного обеспечения состоит в выделении условий и областей изменения переменных, которые наиболее важны для последующего его использования.

При этом разработчик контролирует, чтобы планируемое тестирование не выходило из областей, заданных техническим

заданием (соглашением о требованиях). Испытания программного изделия за пределами технического задания (соглашения о требованиях) могут квалифицироваться как его расширение или могут исключаться по требованию разработчика.

Совместные испытания программного обеспечения

Совместные испытания программного обеспечения проводятся комиссией заказчика, в которой участвуют главный конструктор разработки программного обеспечения и некоторые ведущие его разработчики.

При совместных испытаниях комиссия руководствуется документами:

— техническим заданием (соглашением о требованиях) на ПО;

— действующими государственными и отраслевыми стандартами на проектирование и испытания программных изделий и на техническую документацию разработки;

— программой испытаний по всем требованиям технического задания.

Программа испытаний содержит уточнения требований технического задания для данного программного изделия, набор тестов и должна гарантировать их корректную проверку.

Документация на испытываемое программное обеспечение должна полностью ему соответствовать, обеспечивать познаваемость разрабатываемой системы обслуживаемым персоналом, а также обеспечивать возможность развития и модернизации программного изделия для увеличения его жизненного цикла.

Программа испытаний — это план проведения серии экспериментов. Она разрабатывается с позиции минимизации объема тестирования при заданной и согласованной с заказчиком достоверности получаемых результатов.

Результаты испытаний фиксируются в протоколах, которые обычно содержат следующие разделы:

- назначение тестирования и раздел требований технического задания (соглашения о требованиях), по которому проводится испытание;

- условия проведения тестирования и характеристика исходных данных;

- обобщенные результаты испытаний с оценкой их на соответствие требованиям технического задания (соглашения о

требованиях) и другим руководящим документам;

- выводы о результатах испытаний и степени соответствия созданного программного обеспечения определенному разделу требований технического задания (соглашения о требованиях).

Протоколы по всей программе испытаний обобщаются в соответствующем акте, в результате чего делается заключение о соответствии системы требованиям заказчика и о завершении работы с положительным или отрицательным итогом. При полном выполнении всех требований технического задания заказчик обязан принять программное изделие, и его разработка считается завершенной.

Однако, как уже отмечалось, для больших программных изделий трудно на начальных этапах проектирования предусмотреть и корректно сформулировать все требования в техническом задании (в соглашении о требованиях). Поэтому при отладке и испытаниях программного изделия часто выявляется, что некоторые требования технического задания (соглашения о требованиях) оказываются невыполненными и иногда даже принципиально не могут быть выполнены при самом добросовестном отношении к этому со стороны разработчика. В этом случае необходима совместная работа заказчика и разработчика в поисках компромиссного решения при завершении испытаний и составлении заключения.

Некоторые недостатки программного обеспечения в процессе его испытаний только регистрируются и фиксируются в плане устранения замечаний комиссии, проводившей испытания. Этот план является приложением к акту о результатах испытаний и позволяет отделять последующие доработки от непосредственных испытаний.

Свойства качественного программного обеспечения

Допустим, что кто-то разработал программное обеспечение системы в срок, не превысив при этом предусмотренных затрат. Пусть оно точно и эффективно выполняет все функции, перечисленные в документации. Значит ли это, что оно качественное? По целому ряду характеристик ответ на этот вопрос может быть отрицательным. Например:

— программное обеспечение может оказаться трудным для освоения и модификации, что вызовет дополнительные затраты на его текущее обслуживание, а эти издержки не так уж малы;

— программное обеспечение может оказаться трудным для правильного использования и легким—для неправильного;

— может оказаться, что программное обеспечение в значительной мере привязано к конкретной ЭВМ или плохо стыкуется с ранее разработанными программами.

Существует, однако, целый ряд типичных ситуаций, в которых можно эффективно воздействовать на качество программного обеспечения, и поэтому очень важно знать его основные качественные характеристики.

Управление качеством программного обеспечения — это повышение экономичности его текущего обслуживания. Однако не все качественные показатели программ могут быть выражены непосредственно в терминах затрат, таких, как эксплуатационная надежность, низкий уровень которой приводит к высокой стоимости обслуживания на протяжении всего срока использования программного обеспечения, что связано с частыми корректировками и трудностями приспособления к новым требованиям пользователей.

Еще одна трудность состоит в том, что существующие показатели качества программных средств, как правило, неадекватно отражают те или иные их свойства, определяемые потребностями и предпочтениями будущего пользователя.

В связи с большим разнообразием пользователей невозможно предложить какую-то единую универсальную меру качества программного обеспечения.

Поэтому лучшее, на что может рассчитывать пользователь — это эффективная методика оценки качественных показателей на основе хорошо продуманных детальных вопросников и ранжирования соответствующих характеристик по их важности.

Однако, поскольку метрику программного обеспечения нельзя считать всеобъемлющей, общий результат такой оценки должен рассматриваться скорее как информация к размышлению, чем как окончательные выводы или предписания.

Следовательно, наиболее рациональный способ действий по оценке качества программного обеспечения на сегодня состоит в том, чтобы разработать некоторую систему индикаторов его дефектов и использовать эту систему для определения направлений дальнейшего усовершенствования программных средств, планирования их испытаний, решения вопросов о целесообразности приобретения и организации эксплуатации.

Основную проблему оценки качества программных продуктов представляет тот факт, что многие его частные качественные характеристики противоречивы.

Например:

— увеличение эффективности нередко становится возможным лишь за счет ухудшения мобильности, точности, понятности и удобства эксплуатации;

— повышение точности часто отрицательно сказывается на мобильности вследствие зависимости обеих характеристик от длины машинного слова;

— достижение высокой степени осмысленности может вступать в конфликт с ограничениями на открытость.

Когда возникают подобные конфликтные ситуации, пользователи обычно затрудняются указать, какие же характеристики с их точки зрения являются более существенными.

Повышение жизнеспособности программного обеспечения можно осуществлять следующими четырьмя способами:

— установлением в явном виде целевых параметров качества и их относительных приоритетов;

— применением контрольных таблиц и вопросников по анализу качества;

— введением специальных мер, гарантирующих высокое качество программ;

— использованием специальных средств и методов повышения качества.

Если пользователь предпочитает эффективности программы ее мобильность и удобство эксплуатации, важно довести это до сведения разработчика, причем, сделать это в такой форме, которая позволяла бы в дальнейшем определить, до какой степени желаемые свойства реализованы в готовой системе программного обеспечения.

Множество характеристик качества программного обеспечения может быть представлено в виде дерева, в котором более элементарные характеристики являются необходимым условием существования более обобщенных (см. схему).

Стрелки в нем указывают логическое отношение следования. Например, если программа удобна в эксплуатации, то она обязательно понятна, она обязательно оказывается структурированной, согласованной и осмысленной, открытой и информативной.

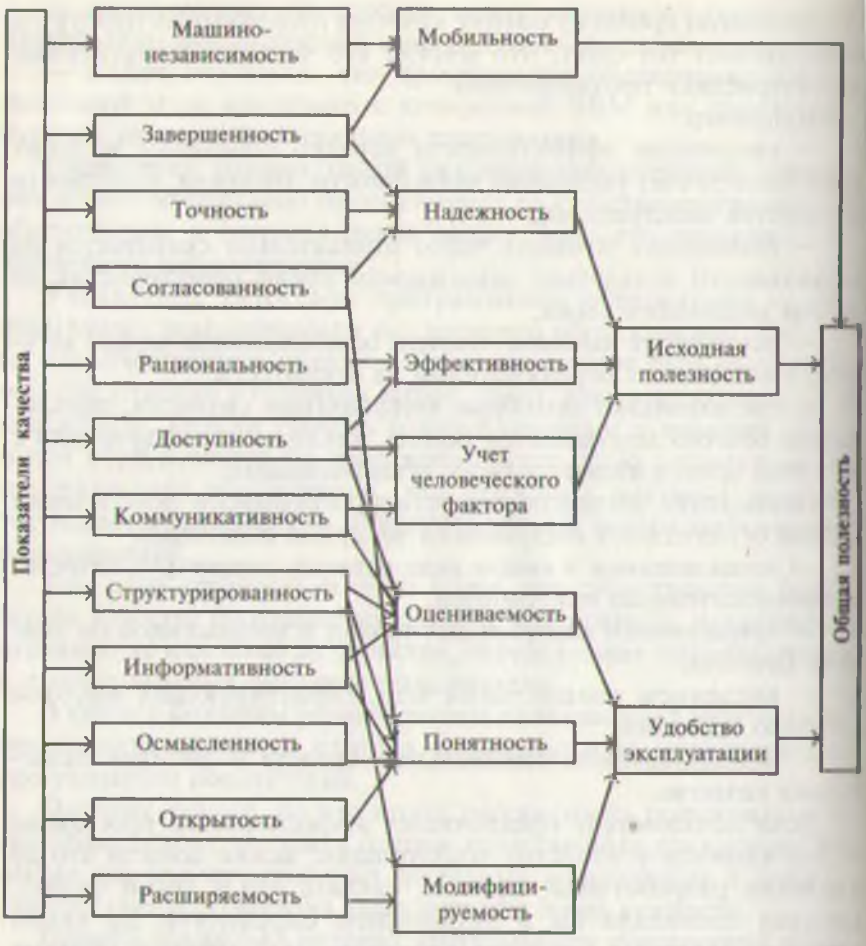


Схема («дерево») характеристик качества программного обеспечения

Характеристики, расположенные на нижнем уровне дерева, представляют собой множество элементарных свойств, которые имеют фундаментальные отличия друг от друга и группируются в наборы соответственно условиям, необходимым для существования конкретных характеристик промежуточного уровня.

Для определения качества программного обеспечения необходимо объявить для него целесообразную совокупность характеристик качества по приоритетам, установить для них один или более измеримых показателей. Например, число (процент завер-

шенности, процент ошибок, выявленных при тестировании и т.п.) или качественное суждение, вынесенное человеком в одной из двух форм: в виде численной оценки или в виде решения типа «да — нет».

Качественные характеристики затем используются для вывода некоторой *интегральной оценки* достоинства всего программного обеспечения.

При этом должны обнаруживаться также всевозможные аномалии характеристик качества, которые следует классифицировать. Серьезные недостатки должны анализироваться и исправляться либо игнорироваться, если выяснится, что данный критерий неприменим к оцениваемому программному продукту, что бывает довольно часто. Например, низкий уровень понятности может быть допустим в случае высокоэффективной программы, работающей в реальном масштабе времени.

Ниже предлагаются описания всех возможных промежуточных и элементарных свойств качественного программного обеспечения.

Понятность. Программное обеспечение обладает свойством *понятности* в той степени, в которой оно позволяет оценивающему лицу понять его назначение.

Из этого определения следует, что человек, проводящий оценивание, должен иметь возможность проникнуть в смысл документации и принципов функционирования программного обеспечения, равно как и понять его взаимосвязи с другими программными средствами и подсистемами.

Под этим определением подразумевается, что всякий программный продукт необходимо создавать с учетом нужд конечного пользователя, условий, оговоренных конкретным документом («Соглашением о требованиях», контрактом и т.п.).

Система программного обеспечения понятна лишь в том случае, если она описана ясным и простым языком, свободным от жаргона и неадекватно определенных терминов или символов, и содержит необходимые ссылки на легкодоступные документы, позволяя читателю разобраться в сложных или новых элементах.

В применении к блок-схемам алгоритмов и машинным программам свойство понятности означает четкость и аккуратность рисунков, расшифровку соответствующей символики, согласованное использование символов, адекватные комментарии или описание одинаковых всюду элементов диалога, а также написания в программах одних и тех же символических имен переменных и применение легко различимых имен.

Завершенность. Программный продукт обладает свойством *завершенности*, если в нем присутствуют все необходимые компоненты, каждый из которых разработан всесторонне.

О документе говорят, что он завершен, если в нем присутствуют все элементы содержания, перечисленные в оглавлении, и это содержание с достаточной полнотой отражает аспекты функционирования систем, соответствующие всем другим характеристикам.

Если база данных содержит архивные данные, то в документацию, кроме структуры базы данных должно быть включено описание всех хранимых сведений. Кроме того, сама база данных не может быть признана завершенной, если она не защищена должным образом от потери информации и не предусматривает наличия необходимого количества резервных копий.

Следовательно, завершенность предполагает замкнутость описания и живучесть программы.

Осмысленность. Программный продукт обладает свойством *осмысленности*, если его документация не содержит избыточной информации.

Лишние фразы и повторы затемняют основную мысль и не позволяют сосредоточить внимание на важных подробностях. Это относится как к документации, так и к самим программам.

Иногда требования осмысленности могут противоречить друг другу, особенно если читатели не являются специалистами.

Мобильность. Программный продукт обладает свойством *мобильности*, если он может легко и эффективно использоваться для работы на ЭВМ иного типа, чем, та, для которой он предназначен.

Свойство мобильности равнозначно свойству автономности, определяющему способность программных средств работать «на самообслуживании», без привлечения дополнительных программных ресурсов.

Потребность в обеспечении мобильности зависит от конкретного проекта, и во многих случаях здесь должны сравниваться затраты и получаемый эффект.

Может быть вполне оправданным назначение некоторого произвольно выбранного уровня мобильности, которому должно удовлетворять программное обеспечение, с определенными затратами для перехода к новым условиям функционирования в будущем.

Полезность. Программный продукт обладает свойством *полезности*, если он удобен для практического применения.

Это свойство имеет две стороны:

1. Программы должны быть написаны так, чтобы была возможность их полного или частичного использования (в случае необходимости) в иных условиях. При этом речь идет не о свойстве мобильности, а об ответе на вопрос: полезна ли функция, выполняемая данным программным продуктом, для других проектов?

Например, если рассматривается информационно-справочная система, важно выяснить, может ли она использоваться не только для тех целей, которые ставились первоначально при ее создании.

2. Необходима надлежащая проработка вопросов обеспечения взаимодействия человека с машиной: должны быть четко определены входные параметры, форматы ввода данных, которые следует делать либо свободными, либо обеспечивающими определенную гибкость.

Описание выходов программ и базы данных должно быть ясным и легко интерпретируемым, позволяющим гибко определять формат и содержание результирующей информации.

Вопросы человеко-машинных интерфейсов важны даже тогда, когда программный продукт не отличается широтой использования и не обладает общей полезностью. Дело в том, что исходные данные готовятся людьми, результаты тоже должны читаться и интерпретироваться людьми, и люди могут привлекаться к операциям ведения базы данных.

В области сопряжения функций и машины существует широкий набор методов и средств, иногда они встречаются в языках высокого уровня, и большая часть этой работы ложится на разработчиков программного обеспечения.

Особенно благоприятные возможности для решения вопросов человеко-машинных интерфейсов представляют программы, предназначенные для работы с интерактивными терминалами типа графических устройств, проекционных дисплеев, индивидуальных пультов проектировщиков и т.п. Всюду, где такие терминалы существуют, они должны использоваться не только по основному назначению, но и для того, чтобы способствовать процессам подготовки исходных данных представления результатов.

Машинонезависимость. Программный продукт обладает свойством *машинонезависимости*, если входящие в него программы могут выполняться на вычислительной машине иной конфигурации, чем та, для которой они непосредственно предназначены.

Надежность. Программный продукт обладает свойством *надежности*, если можно ожидать, что он будет удовлетворительно выполнять необходимые функции в течение определенного времени. Обеспечение надежности предполагает получение ответов на следующие две группы вопросов:

1). Способен ли программный продукт удовлетворить выдвинутым требованиям к нему? Если программный продукт – программа, то достигается ли необходимая точность процедур трансляции, загрузки и выполнения?

2). При функционировании в реальных условиях продолжает ли программа работать правильно в случае исходных данных, существенно отличающихся от тестовых? Как много будет выявлено скрытых ошибок после аттестации программы как работоспособной? Какова вероятность того, что результаты будут содержать необнаруженные ошибки?

Структурированность. Программный продукт обладает свойством *структурированности*, если его взаимосвязанные части организованы в единое целое определенным образом.

Структурированность программы может иметь в своей основе самые различные причины. Например, она может быть разработана в соответствии со специальными стандартами, определенными руководящими принципами и требованиями к интерфейсам, или она может быть написана с использованием языка структурного программирования, или может отражать в своей структуре процесс постепенного эволюционного развития на основе целенаправленных и систематизированных изменений.

Эффективность. Программный продукт обладает свойством *эффективности*, если он выполняет требуемые функции без лишних затрат ресурсов и времени.

Термин «ресурсы» здесь понимается в широком смысле: это может быть оперативная память, общее количество выполняемых команд на одну итерацию решаемой задачи или на один прогон, внешняя память, пропускная способность канала и т.п.

Часто *эффективность* приобретается ценой ухудшения других характеристик так как нередко является машинозависимой характеристикой и определяется свойствами конкретного используемого языка программирования.

Необходимость обеспечения эффективности за счет ухудшения других характеристик программного обеспечения должна особо отмечаться в задании на проектирование ПО.

Точность. Программный продукт обладает свойством *точ-*

ности, если выдаваемые им результаты имеют точность, достаточную с точки зрения основного их назначения.

В качестве отрицательного примера можго привести два оператора

```
SUM: = TIIN/100;      SUM: = SUM + TIIN/100;
```

В результате их действий при некоторых значениях переменных (TIIN = 20 и TIIN = 81) происходит нежелательная потеря целого сума (в результате получается SUM = 0).

Для обеспечения требуемой точности это вычисление можно выполнить оператором

```
SUM: = (TIIN1+ TIIN2)/100;
```

Доступность. Программный продукт обладает свойством *доступности*, если он допускает селективное использование отдельных его компонент.

Отрицательным примером может служить запись

```
GRVITY: = 1.40764548E16/ R**2;
```

Здесь GRVITY – переменная, обозначающая гравитационную постоянную.

Иногда может возникнуть необходимость в использовании иного, не земного значения гравитационной постоянной, например, постоянной характеристикой лунного притяжения.

Для обеспечения широкой доступности программы целесообразна следующая форма выражения:

```
GRVITY = GCONST/R**2;
```

где переменной GCONST по умолчанию при запуске программы присваивается значение $1.4076548*10^{16}$, но пользователь может задавать и/или изменить это значение на новое.

Модифицируемость. Программный продукт обладает свойством *модифицируемости*, если он имеет структуру, позволяющую легко вносить требуемые изменения.

Открытость. Программный продукт обладает свойством *открытости*, если его функции и назначения соответствующих операторов легко понимаются в результате чтения текста программы.

Коммуникативность. Программный продукт обладает свойством *коммуникативности*, если он дает возможность легко опи-

сывать входные данные и выдает информацию, форма и содержание которой просты для понимания и несут полезные сведения.

Отрицательным примером может служить выдаваемое программой число или набор чисел без комментариев.

Информативность. Программный продукт обладает свойством *информативности*, если он содержит информацию, необходимую и достаточную для понимания читающим лицом назначения программных средств, принятых допущений, существующих ограничений, исходных данных, результатов, отдельных компонентов и текущего состояния программ при их функционировании.

Расширяемость. Программный продукт обладает свойством *расширяемости*, если он позволяет увеличивать при необходимости объем памяти для хранения данных или расширять его функции.

Учет человеческого фактора. Программный продукт учитывает человеческий фактор, если он способен выполнять свои функции, не требуя излишних затрат времени со стороны пользователя, неоправданных усилий пользователя по поддержанию процесса функционирования программ и без ущерба для морального состояния пользователя.

РЕЗЮМЕ

* На этапе оценки (испытания) готовое ПО подвергается строгому системному испытанию со стороны группы лиц, обычно не являющихся разработчиками, на предмет оценки его качеств.

* Цель испытаний — определение степени соответствия созданного программного обеспечения техническому заданию (соглашению о требованиях), полученному от заказчика.

* Для всесторонней проверки программного обеспечения его опытный образец подвергается испытаниям со стороны:

— главного конструктора (предварительные испытания);

— пользователя с участием разработчиков (совместные испытания).

* На предварительных испытаниях программного обеспечения производится, по существу, такое же тестирование, что и на его совместных испытаниях, только в меньшем объеме.

* Совместные испытания программного обеспечения проводятся комиссией заказчика, в которой участвуют главный конст-

руктор разработки программного обеспечения и некоторые ведущие его разработчики.

* При полном выполнении всех требований технического задания заказчик обязан принять программное изделие, и его разработка считается завершенной.

* В связи с большим разнообразием пользователей невозможно предложить какую-то единую универсальную меру качества ПО.

* Основную проблему оценки качества программных продуктов представляет тот факт, что многие его частные характеристики качества противоречивы.

* Для определения качества программного обеспечения необходимо объявить для него целесообразную совокупность характеристик качества по приоритетам, установить для них один или более измеримых показателей, которые затем используются для вывода некоторой интегральной оценки достоинства всего программного обеспечения.

Темы для повторения

1. Оценка (испытания) программного обеспечения.
2. Предварительные испытания программного обеспечения.
3. Совместные испытания программного обеспечения.
4. Свойства качественного программного обеспечения.
5. Проблема оценки качества программных продуктов.
6. Определение качества программного обеспечения.
7. Качественные характеристики ПО: понятность, завершенность, осмысленность, мобильность, полезность, машинонезависимость, надежность, структурированность, эффективность, точность, доступность, модифицируемость, открытость, коммуникативность, информативность, расширяемость, учет человеческого фактора.

Глава 10. Использование программного обеспечения

Использование программного обеспечения является *заключительной фазой* его жизненного цикла. В это время выполняются обучение персонала, внедрение, настройка, эксплуатация, сопровождение и, возможно, расширение программного изделия – так называемое *продолжающееся проектирование*. Это то время, когда разработанное программное изделие «живет», т.е. находится в действии и используется (эксплуатируется) эффективно.

Если все предыдущие этапы разработки программного обеспечения были проведены правильно и на достаточно высоком уровне, то его использование будет идти спокойно и гармонично (идеальный вариант).

Эксплуатация программного обеспечения заключается в исполнении, функционировании его на ЭВМ для обработки информации и в получении результатов, являющихся целью его создания, а также, в обеспечении достоверности и надежности выдаваемых данных.

Программное изделие подобно живому организму должно развиваться во времени. В процессе эксплуатации большого программного продукта возможно обнаружение в нем ошибок, и появляется необходимость его модификации и расширения его функций — дальнейшая разработка ПО.

Продолжением разработки программного обеспечения занимается группа сопровождения, которая выполняет следующие функции:

— эксплуатационное обслуживание программного обеспечения;

— развитие функциональных возможностей программного обеспечения;

— повышение эксплуатационных характеристик программного обеспечения;

— тиражирование программного обеспечения;

— перенос программного обеспечения на различные типы вычислительных средств.

Сопровождение программного обеспечения состоит в эксплуатационном обслуживании, развитии функциональных возможностей и повышении эксплуатационных характеристик программного изделия, в тиражировании и переносе его на различные типы вычислительных средств. Сопровождение играет роль необходимой обратной связи с эксплуатацией ПО.

Выявление и исправление ошибок в программном обеспечении, а так же его модификация и расширение его функций, как правило, ведутся одновременно с эксплуатацией текущей версии программного изделия.

После проверки подготовленных корректировок на одном из экземпляров программ очередная версия программного изделия заменяет ранее эксплуатировавшиеся или некоторые из них. При этом процесс эксплуатации программного изделия может быть практически непрерывным, так как замена версии ПИ является кратковременной. Эти обстоятельства приводят к тому, что процесс эксплуатации версии программного изделия обычно идет параллельно и независимо от этапа сопровождения.

Сопровождение программного обеспечения

Программные средства являются одним из наиболее гибких видов промышленных изделий и эпизодически подвергаются изменениям в течение всего времени их использования.

Иногда достаточно при корректировке программного обеспечения внести только одну ошибку для того, чтобы резко снизилась его надежность или его корректность при некоторых исходных данных.

Для сохранения и повышения качества программного обеспечения необходимо регламентировать процесс модификации и поддерживать его соответствующим тестированием и контролем качества. В результате программное изделие со временем обычно улучшается как по функциональным возможностям, так и по качеству решения отдельных задач.

Работы, обеспечивающие контроль и повышение качества, а также развитие функциональных возможностей программ, составляют процесс сопровождения.

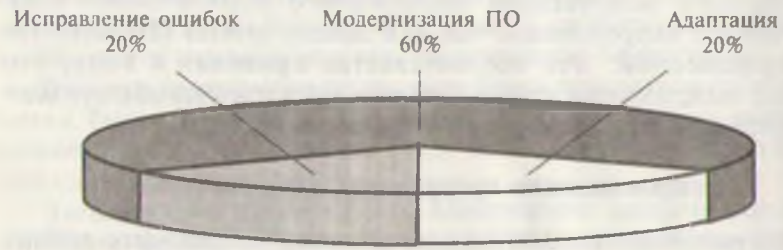
В процессе сопровождения в программное обеспечение вносятся следующие изменения, значительно различающиеся причинами и характеристиками:

— исправление ошибок — корректировка программ, вы-

дающих неправильные результаты в условиях, ограниченных техническим заданием и документацией. Исправление ошибок требуют около 20% общих затрат на сопровождение.

— регламентированная документами адаптация программного обеспечения к условиям конкретного использования, с учетом характеристик внешней среды или конфигурации аппаратуры, на которой предстоит функционировать программам. Адаптация занимает около 20% общих затрат на сопровождение.

— модернизация — расширение функциональных возможностей или улучшение характеристик решения отдельных задач в соответствии с новым или дополнительным техническим заданием на программное изделие. Модернизация занимает до 60% общих затрат на сопровождение.



Затраты на сопровождение программного обеспечения

Первый вид изменений (исправление ошибок) является непредсказуемым и его трудно регламентировать.

Остальные виды корректировок носят упорядоченный характер и проводятся в соответствии с заранее подготавливаемыми планами и документами. Эти корректировки в наибольшей степени изменяют программные изделия и требуют наибольших затрат.

Поэтому изменения, обусловленные ошибками, в большинстве случаев целесообразно по возможности накапливать и реализовывать их, приурочивая к изменениям, регламентированным модернизациями.

Однако некоторые ошибки вызывают необходимость срочного исправления программ. В этих случаях допустимо некоторое отставание корректировки документации при более срочном и регистрируемом исправлении самих программ.

Сопровождение программ — это «ложка дегтя» для каждого программиста, всегда помеха при начале разработки какого-либо

нового проекта, заставляющая отвлекаться от его разработки и возвращаться к старым программам и старым проблемам.

Что делает сопровождение программного обеспечения крайне непривлекательным? Это плохо документированный код, недостаточно полное начальное проектирование и отсутствие внешней документации.

Если все этапы жизненного цикла разработки программного обеспечения выполнялись правильно, то сопровождение не будет вызывать серьезных проблем, а будет элементарной технической поддержкой и модификацией внедренного программного продукта.

Со временем, иногда через десятки лет, сопровождение программного обеспечения прекращается. Это может быть обусловлено: разработкой более совершенных программных средств; прекращением использования сопровождаемого программного продукта; нерентабельным возрастанием затрат на его сопровождение.

Отметим, однако, что программное изделие может долго применяться кем-либо и после прекращения его сопровождения от лица разработчика, потому, что этот некто может плодотворно использовать программное изделие у себя самостоятельно, без помощи разработчика.

Для того чтобы со временем прийти к обоснованному решению о прекращении сопровождения программного обеспечения, необходимо периодически оценивать эффективность его эксплуатации, возможный ущерб от отмены сопровождения. В некоторых случаях решение о прекращении сопровождения принимается при противодействии со стороны отдельных пользователей.

Разработка программного обеспечения «под ключ»

Трудозатраты на разработку программного обеспечения «под ключ» часто измеряют в человеко-месяцах, а в крупных проектах в человеко-годах. Это объясняется тем, что создаваемое программное обеспечение должно практически автоматически выполнять все необходимые операции, кроме включения и выключения компьютера. Почти никогда такие проекты не выполняются одним программистом. В них приходится реализовывать многоуровневую систему отслеживания ошибок и интерактивную помощь оператору. Программные изделия, разработанные «под

ключ», должны «вести» неподготовленного пользователя и обладать некоторым «интеллектом».

С другой стороны, чтобы окупить затраты на разработку программного обеспечения чаще всего предполагается впоследствии его тиражировать. В среднем на проектирование системы «под ключ» уходит около 12 месяцев. Еще два года исправляют ошибки, вкравшиеся в программы. К этому моменту разработчик уже ждет усовершенствованную версию.

В программном обеспечении, разработанном «под ключ» самостоятельные действия пользователя практически должны сводиться к ее активизации. Каждый модуль здесь необходимо продумать до мельчайших деталей, а расчетные формулы, меню и базы данных спроектировать самым тщательным образом. Приступая к выполнению подобного заказа, необходимо приготовиться к длительным стрессам. Из-за высокой стоимости разработки программного обеспечения и связанного с этим психологического дискомфорта заказчик, вероятно, при приемке программного изделия будет не менее привередлив, чем при выборе невесты.

Обычно пользователь будущего программного продукта плохо представляет себе, какой продукт в точности ему нужен и сколько времени займет его разработка. Разработчик программного обеспечения должен быть готов к тому, что заказчик часто хочет получить больше, чем предполагалось первоначально и побыстрее. Вот почему необходимо с самого начала точно определить решаемую задачу с учетом возможности ресурсов, изучить и «прочувствовать» всю систему разрабатываемого продукта. И только после тщательного внешнего проектирования программного обеспечения писать программы, реализующие письменно утвержденные требования пользователя.

Надо помнить о том, что следует решать лишь поставленную задачу, избегая излишнего ее обобщения. Конечная цель разработки программного обеспечения «под ключ» — выполнить заказ, т.е. изготовить законченный по всем правилам программный продукт.

Хороший программист знает, что не всегда его разработка получается всесторонне совершенной, но он добросовестно стремится разработать программное обеспечение на профессиональном уровне. Его изделие правильно, надежно, компактно, обладает высокой скоростью выполнения функций, экономно использует оперативную и дисковую память. В нем реализованы все возможности инструментального языка.

Документация к программному обеспечению

Формальные требования к документации программного обеспечения описаны в ЕСПД (Единая система программной документации), неформально: состав документации к программному обеспечению состоит из описания внешнего эффекта ПО и описания его внутреннего устройства.

Первая часть документации, так называемая «Инструкция пользователю» (или «Руководство пользователю») предназначена для того, кто собирается использовать программное обеспечение, не вникая в подробности его внутреннего устройства; вторая («Руководство программисту») необходима при модификации ПО или при необходимости исправить в нем ошибку.

В целом, документация к программному обеспечению может содержать нижеперечисленные сведения:

1. Наименование ПО и описание задачи, которую оно решает.

2. Область применимости ПО, т.е. класс исходных данных, на который рассчитано ПО и степень контроля за принадлежностью данных этому классу.

3. Описание метода, которым решается задача, его достоинства и недостатки. Должны быть приведены расчетные формулы или алгоритм. При необходимости даются ссылки на литературу, откуда взяты формулы или метод.

4. Режим работы ПО (точки входа в процедуру), сообщения, выдаваемые по ходу его работы, ответы пользователя на них (если это необходимо).

5. Исходные данные, необходимые для работы ПО; а также выдаваемые им результаты; побочный эффект изменения в памяти, который производит ПО и который не является его основной задачей, например, изменение входных параметров.

6. Правила подготовки исходных данных на внешних носителях (если они применяются) и вид выдаваемой информации.

7. Описание структуры данных. Для любой переменной описывается ее назначение, атрибуты (тип, размер массива и т.д.), структура информации в ней, если она не очевидна. Описание переменных должно начинаться с тех, которые служат исходными данными и результатами.

8. Описания форм, объектов. Опись свойств форм и объектов.

9. Тексты программ, процедур (в виде распечатки ЭВМ) с комментариями.

10. Тесты. Включаются только тестовые исходные данные и конечные результаты без промежуточных результатов.

Тесты необходимы при модификации программы, при переносе ее на ЭВМ с той, на которой она отлаживалась.

Во всех случаях сначала запускаются тесты, чтобы убедиться в правильности программы.

11. Возможные модификации программного обеспечения. Указывается назначение модификации и те изменения, которые надо внести.

12. Инструкция (руководство) пользователю.

Инструкция по использованию программного обеспечения

Инструкция по использованию программного обеспечения (или просто «Инструкция пользователю», или «Руководство для пользователя») – это выдержка из полной документации, предназначенная для эксплуатации программного обеспечения. Она представляет собой независимый документ, в котором описывается: что делает программное обеспечение и как им пользоваться.

«Инструкция пользователю» должна содержать всю необходимую для пользователя информацию и должна быть ему понятна без дополнительных материалов (без обращения к другим спецификациям). Следовательно, необходимая для этой инструкции информация переписывается полностью из соответствующих спецификаций.

Первая часть инструкции является описательной и должна содержать:

— наименование программного обеспечения;

— краткое описание программного обеспечения;

— перечень выполняемых им функций;

— краткую характеристику метода (или методов), его достоинство и недостатки;

— полную библиографическую ссылку на полное описание метода;

— описание входных и выходных данных.

Вторая часть документа должна описывать порядок работы с программным обеспечением. Она должна содержать описание всех режимов работы программного обеспечения, а также содержание всех печатей и диагностических сообщений, которые выдаются по ходу выполнения программы.

Следует помнить, что пользователь по своей квалификации не является программистом и поэтому его работа с программным обеспечением описывается на понятном ему языке и достаточно подробно, а именно:

— как запустить программное обеспечение;

— как продолжить работу с программным обеспечением (описывается подробный интерактивный режим его работы с ПО);

— подготовка и ввод исходных данных в программное обеспечение;

— как реагировать на запросы программного обеспечения;

— как вести работу в исключительных ситуациях;

— как реагировать на ошибки;

— как восстановить работу программного обеспечения в случае аварийного его завершения;

— как получить требуемый результат;

— как правильно закончить работу с программным обеспечением (запланированный программой выход).

РЕЗЮМЕ

* Использование программного обеспечения является заключительной фазой его жизненного цикла, во время которого выполняются обучение персонала, внедрение, настройка, эксплуатация, сопровождение и, возможно, расширение программного изделия.

* Если все предыдущие этапы разработки программного обеспечения были проведены правильно и на достаточно высоком уровне, то его использование будет идти спокойно и гармонично (идеальный вариант).

* Сопровождение программного обеспечения состоит в эксплуатационном обслуживании, развитии функциональных возможностей и повышении эксплуатационных характеристик программного изделия, в тиражировании и переносе его на различные типы вычислительных средств.

* Сопровождение играет роль необходимой обратной связи с эксплуатацией.

* Изменения, обусловленные ошибками, в большинстве случаев целесообразно по возможности накапливать и реализовывать их, приурочивая к изменениям, регламентированным модернизациями программного обеспечения.

* Если все этапы жизненного цикла разработки программного обеспечения выполнялись правильно, то сопровождение не будет вызывать серьезных проблем, а будет элементарной технической поддержкой и модификацией внедренного программного продукта.

* Программные изделия, разработанные «под ключ», должны «вести» неподготовленного пользователя и обладать некоторым «интеллектом». Создаваемое программное обеспечение должно практически автоматически выполнять все необходимые операции, кроме включения и выключения компьютера.

* Каждый модуль здесь необходимо продумать до мельчайших деталей, а расчетные формулы, меню и базы данных спроектировать самым тщательным образом.

* Необходимо с самого начала точно определить решаемую задачу с учетом возможности ресурсов, изучить и «прочувствовать» всю систему разрабатываемого продукта. И только после тщательного внешнего проектирования программного обеспечения писать программы, реализующие письменно утвержденные требования пользователя.

Темы для повторения

1. Использование программного обеспечения.
2. Эксплуатация программного обеспечения.
3. Сопровождение программного обеспечения.
4. Разработка программного обеспечения «Под ключ».
5. Документация к программному обеспечению.
6. Инструкция по использованию программного обеспечения.

Практические работы

Основной целью выполнения практических работ является изучение технологии и методов разработки, эффективности, тестирования, отладки и эксплуатации конкретных программ.

В результате выполнения практических работ студенты должны научиться самостоятельно разрабатывать программы на алгоритмическом языке программирования высокого уровня, оформлять документацию к ней и правильно ее оценивать.

Следует заметить, что разработка программ может быть выполнена любыми средствами на любом алгоритмическом языке и на любом компьютере по усмотрению студента — разработчика программы.

Самостоятельно разрабатывая программу, студент также должен научиться:

— проводить беседы с пользователем с целью определения требований к разрабатываемой программе и ее осуществимости;

— разрабатывать внешний и внутренний проект программы;

— оценивать качество программ;

— выполнять тестирование и отладку разрабатываемой программы.

С этой целью студенту рекомендуется определить для себя задачу и на ее основе провести основные этапы разработки конкретной программы. Задачу, например, можно выбрать из предлагаемого ниже перечня задач.

Предлагается выполнить семь работ.

1. Предварительное проектирование.
2. Разработка программного обеспечения.
3. Проведение внешнего проектирования программы.
4. Разработка архитектуры программного обеспечения.
5. Словесное и блок-схемное описание алгоритма.
6. Метод пошаговой детализации.
7. Тестирование и отладка программы.

Работа 1. **Предварительное проектирование**

Цель работы:

— предварительное проектирование конкретной программы;

— составление перечня требований и функциональных характеристик разрабатываемого программного изделия;

— разработка постановки задачи.

Порядок выполнения работы и отчетность

Во время выполнения практической работы необходимо определить потребность в программном изделии, его назначение и основные функциональные характеристики; составить перечень требований к нему.

Работа может быть оформлена в виде документа «Постановка задач».

Работа 2. **Разработка программного обеспечения**

Цель работы:

— определение этапов разработки конкретной программы;

— разработка календарного плана создания конкретной программы.

Порядок выполнения работы и отчетность.

Во время выполнения практической работы необходимо подробно проанализировать этапы разработки конкретной программы (ее жизненный цикл), начиная от возникновения потребности в ней до полного прекращения ее использования вследствие ее морального старения или потери необходимости решения соответствующих задач.

Работа может быть оформлена в виде календарного плана разработки программы по форме:

№	Наименование этапа разработки программы	Срок исполнения		Примечания
		начало	окончание	

Работа 3. **Проведение внешнего проектирования программы**

Цель работы:

— проведение внешнего проектирования конкретной программы;

— построение функциональной схемы;

— разработка взаимодействия разрабатываемой программы с пользователем: сценарий, экранные формы, набор подсказок, и пр.

Порядок выполнения работы и отчетность.

Во время выполнения практической работы необходимо описать ожидаемое поведение разрабатываемого программного изделия с точки зрения внешнего по отношению к нему наблюдателя (обычно — пользователя), то есть осуществить «конструирование» внешних взаимодействий будущего программного продукта с пользователем без конкретизации его внутреннего устройства.

Работа может быть оформлена в виде внешней спецификации.

Работа 4. **Разработка архитектуры программного обеспечения**

Цель работы:

— разработать архитектуру программного изделия.

Порядок выполнения работы и отчетность.

Во время выполнения практической работы необходимо разработать архитектуру программного изделия: спроектировать структуру всех его компонентов, его модульно-иерархическое построение.

Работа может быть оформлена в виде архитектуры программы.

Работа 5. **Словесное и блок-схемное описание алгоритма**

Цель работы:

— разработка алгоритма решения задачи;

— запись алгоритма в словесной форме;

— запись алгоритма в блок-схемной форме.

Порядок выполнения работы и отчетность.

Во время проведения практической работы необходимо разработать алгоритм решения конкретной задачи и представить его в словесной и в блок-схемной формах, которые должны быть отражены в отчете.

Работа 6. **Метод пошаговой детализации**

Цель работы:

— освоить метод пошаговой разработки программ.

Порядок выполнения работы и отчетность.

Во время выполнения практической работы необходимо осуществить пошаговую разработку конкретной программы, описание которой представить в отчете.

Работа 7. **Тестирование и отладка программы**

Цель работы:

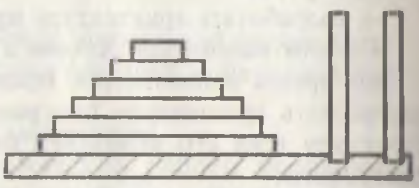
— осуществить тестирование и отладку конкретной программы на алгоритмическом языке высокого уровня.

Порядок выполнения работы и отчетность.

Во время выполнения практической работы необходимо составить набор тестов к конкретной программе и провести ее отладку. Составленный набор тестов необходимо представить в отчете.

Варианты задач

1. «*Ханойская башня*». Доска имеет три колышка. На первом нанизано m дисков убывающего вверх диаметра (см. рисунок).



Расположить диски в том же порядке на другом колышке. Диски можно перекладывать с колышка на колышек по одному. Класть большой диск на меньший не разрешается.

2. «*Пятнадцать*». На квадратном поле размером 4×4 с помощью датчика случайных чисел расставлены 15 фишек с номерами от 1 до 15. Имеется одна свободная позиция. Расставить фишки по возрастанию их номеров. Передвигать фишки можно только на соседнюю свободную позицию.

3. «*Вращающийся квадрат*». Дан квадрат размером 4×4 клетки, в которых с помощью датчика случайных чисел расставлены буквы от A до P. Упорядочить буквы в квадрате по алфавиту. Квадрат имеет подквадраты, которые можно вращать по часовой стрелке на одну клетку. Подквадраты имеют размер 2×2, и указывается номером левой верхней клетки. Предусмотрена операция, которая может быть выполнена один раз: обмен местами двух букв.

4. «*Кости*». Играющий называет любое число в диапазоне от 2 до 12 и ставку, которую он делает в этот ход. Программа с помощью датчика случайных чисел дважды выбирает числа от 1

до 6 («бросает кубик», на гранях которого цифры от 1 до 6). Если сумма выпавших цифр меньше 7 и играющий задумал число меньшее 7, он также выигрывает сделанную ставку. Если сумма выпавших цифр больше 7, он также выигрывает сделанную ставку. Если играющий угадал сумму цифр, он получает в четыре раза больше очков, чем сделанная ставка. Ставка проиграна, если не имеет место ни одна из описанных ситуаций. В начальный момент у играющего 100 очков.

5. «*Игра в слова*». Программа выбирает слово и рисует на экране столько прочерков, сколько букв в этом слове. Нужно отгадать, какое слово загадано программой. В каждый ход играющий указывает одну букву. Если названа буква, входящая в состав слова, она подставляется вместо соответствующего прочерка. В противном случае играющий теряет 1 очко. В начальный момент у играющего 15 очков.

6. «*Ипподром*». Играющий выбирает одну из трех лошадей, состязающихся на бегах, и выигрывает, если его лошадь приходит первой. Скорость передвижения лошадей на разных этапах выбирается программой с помощью датчика случайных чисел.

7. «*Коровы и быки*». Программа выбирает с помощью датчика случайных чисел четырехзначное число с разными цифрами. Угадать это число. На каждом шаге играющий называет четырехзначное число, а программа сообщает, сколько цифр числа угадано («быки») и сколько цифр угадано и стоит на нужном месте («коровы»). Например, если программой задано число 1294, а играющий назвал 1423, он получит ответ «1 корова, 3 быка».

8. «*Подбери ключи*». Перед играющим четыре запертые двери. Открыть все двери, имея десять ключей, каждый из которых может открыть несколько дверей. Предоставляется 14 попыток.

9. Требуется ввести курсор в область экрана (небольшой круг), расположение которого неизвестно играющему. Передвижение курсора сопровождается звуковым сигналом: если приближается к области, то звук становится выше; если удаляется — ниже.

10. Составить программу обучения работе с клавиатурой. Программа должна выдавать на экран буквы, цифры, слова и фразы, которые следует набрать на клавиатуре.

11. Составить программу, помогающую в запоминании исторических дат. Программа должна предлагать вопросы, контролирующие знание дат исторических событий, например, «В каком году была Куликовская битва?». Если ответ правильный, должен быть предложен следующий вопрос. Если ответ не верен, прог-

рамма подскажет правильный ответ, а позднее повторит этот вопрос еще раз.

12. «*Сбей самолет*». По экрану летят вражеские самолеты. Цель сбить их. Пусковая установка находится в нижней части экрана. Пусковую установку можно перемещать по строке налево и направо.

13. «*Морской бой*». На поле 10×10 позиций стоят невидимые вражеские корабли: 4 корабля по 1 клетке, 3 корабля по 2 клетки, 2 корабля по 3 клетки, 1 корабль в 4 клетки. Необходимо поразить каждую из клеток кораблей. Позиции указываются буквами от А до К (по строкам) и от 1 до 10 (по столбцам). Конфигурация и положение кораблей на поле выбираются с помощью датчика случайных чисел. Если клетка корабля угадана играющим верно, она отмечается крестиком; в противном случае — точкой.

14. Составить программу для заучивания слов иностранного языка. Программа должна предлагать слова из некоторого списка на одном языке, обучающийся — дать перевод этого слова на другой язык. Если ответ правильный, предлагается следующий вопрос. Если ответ не верен, программа подскажет правильный ответ, а позднее повторит этот вопрос еще раз.

15. Составить программу для тренировки памяти. Программа должна высветить на экране несколько точек, играющий — указать, в каком порядке эти точки были высвечены. Координаты точек выбираются в программе с помощью датчика случайных чисел.

16. «*Угадай число*». Один из играющих задумывает число от 1 до 1000, другой пытается угадать его за 10 вопросов типа: верно ли, что задуманное число больше такого-то числа. Написать программу, играющую за отгадчика.

17. Составить программу для изучения созвездий. Программа должна построить на экране изображение созвездия, обучающийся — назвать его. Если ответ правильный, должен быть предложен следующий вопрос. Иначе — программа подскажет правильный ответ, а позднее повторит этот вопрос еще раз.

18. Составить программу, помогающую в изучении движения тела, брошенного под углом к горизонту с некоторой начальной скоростью. Играющий, зная расстояние от человека, бросающего камень, до лунки и ширину лунки, должен задать такие значения угла «альфа» и начальной скорости V, чтобы камень попал в лунку. На экране должны изображаться поверхность земли, лунка, камень и траектория полета камня. Расстояние от человека,

бросающего камень, до лунки и ширины лунки выбирать с помощью датчика случайных чисел.

19. Составить программу, помогающую в изучении колебаний математического маятника. Маятник должен двигаться на экране, совершая гармонические колебания, период которых выбран с помощью датчика случайных чисел. Играющий должен указать длину нити, на которой подвешен маятник. Ответ считается правильным, если ошибка не превышает 10%.

20. Написать шахматную программу, играющую за белых:

а) королем и ферзем против короля;

б) королем и двумя ладьями против короля;

в) королем, ферзем и ладьей против короля.

21. Написать программу, играющую в «крестики-нолики».

22. «*100 спичек*». Из кучки, первоначально содержащей 100 спичек, двое играющих поочередно берут по несколько спичек: не менее одной и не более десяти. Проигрывает взявший последнюю спичку.

23. «*Ним*». Имеются три кучки спичек. Двое играющих по очереди делают ходы. Каждый ход заключается в том, что из одной какой-то кучки берется произвольное ненулевое число спичек. Выигрывает взявший последнюю спичку.

24. Компьютер и пользователь поочередно «называют» число, не превышающее 10. Эти числа складываются одно за другим. Выигрывает тот, кто первым достигнет 100. (Обобщить задачу на ввод произвольных чисел вместо 10 и 100).

25. В ряд расположены 15 спичек. Требуется собрать их в 5 кучек по 3 спички в кучке, перекладывая спички по одной. Каждую спичку можно переносить только через три других. Обобщить задачу на ввод произвольных чисел вместо 15 и 3).

26. Разложить спички в три кучки, например, по 12, 10 и 7 спичек в кучке. Компьютер и пользователь поочередно берут из кучек некоторое количество спичек, но только из одной кучки. Можно взять сразу всю кучу. Выигрывает тот, кто последним возьмет спички.

27. Десять спичек положены в один ряд. Распределить их попарно, всего 5 пар, перекладывая по одной спичке через две. (Обобщить задачу на ввод произвольных чисел вместо 10 и 2).

28. Написать программу «калькулятор».

29. Написать программу простейшего текстового редактора.

30. Написать программу простейшего музыкального редактора.

Использованная литература

1. Фокс Дж. Программное обеспечение и его разработка. Пер. с англ. М.: Мир, 1985.

2. Липаев В.В. Проектирование программных средств. М.: Высшая школа, 1990.

3. Российская академия наук. Методы и средства информационной технологии в науке и производстве. М.: Наука, 1992.

4. Российская академия наук. Программирование прикладных систем. М.: Наука, 1992.

5. Шафрин Ю. Основы компьютерной технологии. М.: Наука, 1997.

6. Майерс Г.Надежность программного обеспечения. Пер. с англ. М.: Мир, 1980.

7. Гантер Р. Методы управления проектированием программного обеспечения. Пер. с англ. М.: Мир, 1981.

8. Изосимов Л.В. Рыленко А.Л. Метрическая оценка качества программ. М.: МАИ, 1989.

9. Боэм Б. и др. Характеристики качества программного обеспечения. Пер. с англ. М.: Мир, 1981.

10. Гласс Р. Руководство по надежному программированию. М.: Финансы и статистика, 1982.

11. Ван Тассел Д.Стиль, разработка, эффективность, отладка и испытание программ. Пер. с англ. М.: Мир, 1981.

12. Боэм Б.У.Инженерное проектирование программного обеспечения. Пер. с англ. М.: Радио и связь, 1985.

13. Гласс Р. Нуазо Р.Сопровождение ПО. Пер. с англ. М.: Финансы и статистика, 1982.

14. Тимоти Бад.Объектно-ориентированное программирование в действии. Пер. с англ. С-П.: Питер, 1997.

15. Липаев В.В.Тестирование программ. М.: Радио и связь,1986.

16. Черн С. Готлоб Г.Логическое программирование и базы данных. М.: Мир, 1992.

17. Майерс Г. Искусство тестирования программ. Пер. с англ. М.: Финансы и статистика, 1982.

18. Чернев Д.А. Введение в технологию разработки программного обеспечения. Учебное пособие. Ташкент: ТГТУ, 1995.

19. Косимова Ш.Т., Чернев Д.А. Программалаш технологиясига кириш. Ўқув қўлланма. Тошкент: ТошДТУ, 1997.

20. Чернев Д.А. Технология программирования. Учебник для колледжей. Ташкент: Ўқитувчи, 2003.

Содержание

Глава 10. **Использование программного обеспечения**

ДМИТРИЙ АЛЕКСЕЕВИЧ ЧЕРНЕВ

ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Учебное пособие

Ташкент — «Mehnat» — 2004

Зав. редакцией *А. Бобониязов*
Редактор *Г. Хубларов*
Художественный редактор *О. Баклыкова*
Художник *Х. Кутлуков*
Технический редактор *Т. Смирнова*
Корректор *Г. Усмонова*
Компьютерная верстка *В. Верховцев*

Подписано в печать 24 марта 2004 г. Формат 60x84$^{1}/_{16}$. Гарнитура «Тип Таймс».
Печать офсетная. Усл. печ. л. 14,0. Уч.-изд. л. 14,0.
Тираж 1000 экз. Зак. № 3040.

Издательство «Mehnat», 700129. г. Ташкент, ул. Навои, 30.
Договор № 65-2003.

Оригинал-макет изготовлен в компьютерном отделе издательства «Mehnat».

Отпечатано в типографии №1 Узбекского агентства по печати и информации.
г. Ташкент, ул. Сагбан, тупик 1, дом 2.