Министерство науки и высшего образования РФ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
АМУРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
(ФГБОУ ВО «АмГУ»)

Технологии и методы программирования
сборник учебно-методических материалов

для направления подготовки 10.03.01 «Информационная безопасность»

Благовещенск, 2019

Печатается по решению
редакционно-издательского совета
факультета математики и информатики
Амурского государственного
университета

Составитель: Акилова И.М.

Технологии и методы программирования: сборник учебно-методических материалов для направления подготовки для направления подготовки 10.03.01 «Информационная безопасность». – Благовещенск: Амурский гос. ун-т, 2019.

КРАТКОЕ ИЗЛОЖЕНИЕ ЛЕКЦИОННОГО МАТЕРИАЛА

Лекция №1

НАДЕЖНОЕ ПРОГРАММНОЕ СРЕДСТВО КАК ПРОДУКТ |ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ. ИСТОРИЧЕСКИЙ И СОЦИАЛЬНЫЙ КОНТЕКСТ ПРОГРАММИРОВАНИЯ.

Понятие информационной среды процесса обработки данных. Программа как формализованное описание процесса. Понятие о программном средстве. Понятие ошибки в программном средстве. Неконструктивность понятия правильной программы. Надежность программного средства]. Технология программирования как технология разработки надежных программных средств. Технология программирования и информатизация общества.

1.1. Программа как формализованное описание процесса обработки данных. Программное средство.

Целью программирования является описание процессов обработки данных (в дальнейшем - просто *процессов}*. Согласно ИФИПа - [1.1]: *данные (data)* - это представление фактов и идей в формализованном виде, пригодном для передачи и переработке в некоем процессе, а *информация (Information)* - это смысл, который придается данным при их представлении. *Обработка данных (data processing)* - это выполнение систематической последовательности действий с данными. Данные представляются и хранятся на т.н. *носителях данных.* Совокупность носителей данных, используемых при какой-либо обработке данных, будем называть *информационной средой (data medium).* Набор данных, содержащихся в какой-либо момент в информационной среде, будем называть *состоянием* этой информационной среды. *Процесс* можно определить как последовательность сменяющих друг друга состояний некоторой информационной среды.

Описать процесс - это значит определить последовательность состояний заданной информационной среды. Если мы хотим, чтобы по заданному описанию требуемый процесс порождался *автоматически* на каком-либо компьютере, необходимо, чтобы это описание было *формализованным.* Такое описание называется *программой.* С другой стороны, программа должна быть понятной и человеку, так как и при разработке программ, и при их использовании часто приходится выяснять, какой именно процесс она порождает. Поэтому программа составляется на удобном для человека формализованном *языке программирования, с* которого она автоматически переводится на язык соответствующего компьютера с помощью другой программы, называемой *транслятором.* Человеку *(программисту),* прежде чем составить программу на удобном для него языке программирования, приходится проделывать большую подготовительную работу по уточнению постановки задачи, выбору метода ее решения, выяснению специфики применения требуемой программы, прояснению общей организации разрабатываемой программы и многое другое. Использование этой информации может существенно упростить задачу понимания программы человеком, поэтому весьма полезно ее как-то фиксировать в виде отдельных документов (часто не формализованных, рассчитанных только для восприятия человеком).

Обычно программы разрабатываются в расчете на то, чтобы ими могли пользоваться люди, не участвующие в их разработке (их называют *пользователями).* Для освоения программы пользователем помимо ее текста требуется определенная дополнительная документация. Программа или логически связанная совокупность программ на носителях данных, снабженная программной документацией, называется *программным средством (ПС).* Программа позволяет осуществлять некоторую автоматическую обработку данных на компьютере. Программная документация позволяет понять, какие функции выполняет та или иная программа ПС, как подготовить исходные данные и запустить требуемую программу в процесс ее выполнения, а также: что означают получаемые результаты (или каков эффект выполнения этой программы). Кроме того, программная документация помогает разобраться в самой программе, что необходимо, например, при ее модификации.

1.2. Неконструктивность понятия правильной программы.

Таким образом, можно считать, что продуктом технологии программирования является ПС. содержащее программы, выполняющие требуемые функции. Здесь под «программой» часто понимают правильную программу, т.е. программу, не содержащую ошибок. Однако, понятие ошибки в программе трактуется в среде программистов неоднозначно. Согласно Майерсу [1.2, стр. 10-13] будем считать, что в программе имеется *ошибка,* если она не выполняет того, что разумно ожидать от нее пользователю. «Разумное ожидание» пользователя формируется на основании документации по применению этой программы. Следовательно, понятие ошибки в программе является существенно не формальным. В ПС программы и документация взаимно увязаны, образуют некоторую целостность. Поэтому правильнее говорить об ошибке не в программе, а в ПС в целом: будем считать, что в ПС имеется *ошибка (software error),* если оно не выполняет того, что разумно ожидать от него пользователю. В частности, разновидностью ошибки в ПС является несогласованность между программами ПС и документацией по их применению. В работе [1.3] выделяется в отдельное понятие частный случай ошибки в ПС. когда программа не соответствует своей функциональной спецификации (описанию, разрабатываемом) на этапе, предшествующему непосредственному программированию). Такая ошибка в указанной работе называется *дефектом программы.* Однако выделение такой разновидности ошибки в отдельное понятие вряд ли оправданно, так как причиной ошибки может оказаться сама функциональная спецификация, а не программа.

Так как задание на ПС обычно формулируемся не формально, а также из-за того, что понятия ошибки в ПС не формализовано, то нельзя доказать формальными методами (математически) правильность ПС. Нельзя показать правильность ПС и тестированием: как указал Дейкстра [1.4], тестирование может лишь продемонстрировать наличие в ПС ошибки. Поэтому понятие правильной ПС неконструктивно в том смысле, что после окончания работы над созданием ПС мы не сможем убедиться, что достигли цели.

1.3. Надежность программного средства.

Альтернативой правильного ПС является *надежное* ПС. *Надежность (reliabiliiv)* ПС - это его способность безотказно выполнять определенные функции при заданных условиях в течение заданного периода времени с достаточно большой вероятностью [1.5]. При этом под *отказом* в ПС понимают проявление в нем ошибки [1.2, стр. 10-13]. Таким образом, надежное ПС не исключает наличия в нем ошибок - важно лишь, чтобы эти ошибки при практическом применении этого ПС в заданных условиях проявлялись достаточно редко. Убедиться, что ПС обладает таким свойством можно при его испытании путем тестирования, а также при практическом применении. Таким образом, фактически мы можем разрабатывать лишь надежные, а не правильные ПС.

ПС может обладать различной степенью надежности. Как измерять эту степень? Так же как в технике, степень надежности можно характеризовать [1.2. стр. 10-13] вероятностью работы ПС без отказа в течение определенного периода времени. Однако в силу специфических особенностей ПС определение этой вероятности наталкивается на ряд трудностей по сравнению с решением этой задачи в технике. Позже мы вернемся к более обстоятельному обсуждению этого вопроса.

При оценке степени надежности ПС следует также учитывать последствия каждого отказа. Некоторые ошибки в ПС могут вызывать лишь некоторые неудобства при его применении, тогда как другие ошибки могут иметь катастрофические последствия, например, угрожать человеческой жизни. Поэтому для оценки надежности ПС иногда используют дополнительные показатели, учитывающие стоимость (вред) для пользователя каждого отказа.

1.4. Технология программирования как технология разработки надежных программных средств.

В соответствии с обычным значением слова «технология» [1.6] *под технологией программирования (programming technology)* будем понимать совокупность производственных процессов, приводящую к созданию требуемого ПС, а также описание этой со-

вокупности процессов. Другими словами, технологию программирования мы будем понимать здесь в широком смысле как технологию разработки *программных средств,* включая в нее все процессы, начиная с момента зарождения идеи этого средства, и, в частности, связанные с созданием необходимой программной документации. Каждый процесс этой совокупности базируется на использовании каких-либо методов и средств, например, компьютер (в этом случае будем говорить о *компьютерной* технологии программирования).

В литературе имеются и другие, несколько отличающиеся, определения технологии программирования. Эти определения обсуждаются в работе [1.7]. Используется в литературе и близкое к технологии программирования понятие *программной инженерии,* определяемой как систематический подход к разработке, эксплуатации, сопровождению и изъятию из обращения программных средств [1.7, стр. 9-16]. Именно программной инженерии *(software engineering)* посвящена упомянутая работа [1.3]. Главное различие между технологией программирования и программной инженерией как дисциплинами для изучения заключается в способе рассмотрения и систематизации материала. В технологии программирования акцент делается на изучении процессов разработки ПС *(технологических процессов]* и порядке их прохождения - методы и инструментальные средства разработки ПС *используются* в этих процессах (их применение и образуют технологические процессы). Тогда как в программной инженерии изучаются различные методы и инструментальные средства разработки ПС с точки зрения достижения определенных целей — эти методы и средства могут использоваться в разных технологических процессах (и в разных технологиях программирования).

Не следует также путать технологию программирования с методологией программирования [1.8]. В технологии программирования методы рассматриваются «сверху» - с точки зрения организации технологических процессов, а в методологии программирования методы рассматриваются «снизу» - с точки зрения основ их построения (в работе [1.9, стр. 25] методология программирования определяется как совокупность механизмов, применяемых в процессе разработки программного обеспечения и объединенных одним общим философским подходом).

Имея ввиду, что надежность является неотъемлемым атрибутом ПС, мы будем рассматривать технологию программирования как технологию разработки *надежных* ПС. Это означает, что

- мы будем рассматривать все процессы разработки ПС, начиная с момента возникновения замысла ПС;
- нас будут интересовать не только вопросы построения программных конструкций, но и вопросы описания функций и принимаемых решений с точки зрения их человеческого (неформального) восприятия:
- в качестве продукта технологии принимается надежная (далеко не всегда правильная) ПС.

Такой взгляд на технологию программирования будет существенно влиять на организацию технологических процессов, на выбор в них методов и инструментальных средств.

1.5. Технология программирования и информатизация общества.

Технологии программирования играло разную роль на разных этапах развития программирования. По мере повышения мощности компьютеров и развития средств и методологии программирования росла и сложность решаемых на компьютерах задач, что привело к повышенному вниманию к технологии программирования. Резкое удешевление стоимости компьютеров и, в особенности, стоимости хранения информации на компьютерных носителях привело к широкому внедрению компьютеров практически во все сферы человеческой деятельности, что существенно изменило направленность технологии программирования. Человеческий фактор стал играть в ней решающую роль. Сформировалось достаточно глубокое понятие качества ПС, причем предпочтение стало отдаваться не столько эффективности ПС, сколько удобству работы с ним для пользова-

телей (не говоря уже о его надежности). Широкое использование компьютерных сетей привело к интенсивном) развитию распределенных вычислений, дистанционного доступа к информации и электронного способа обмена сообщениями между людьми. Компьютерная техника из средства решения отдельных задач все более превращается в средство информационного моделирования реального и мыслимою мира, способное просто отвечать людям на интересующие их вопросы. Начинается этап глубокой и полной информатизации (компьютеризации) человеческого общества. Все это ставит перед технологией программирования новые и достаточно трудные проблемы.

Сделаем краткую характеристику развития программирования по десятилетиям.

В 50-е годы мощность компьютеров (первого поколения) была невелика, а программирование для них велось, в основном, в машинном коде. Решались, главным образом, научно-технические задачи (счет по формулам), задание на программирование содержало, как правило, достаточно точную постановку задачи. Использовалась интуитивная технология программирования: почти сразу приступали к составлению программы по заданию, при этом часто задание несколько раз изменялось (что сильно увеличивало время и без того итерационного процесса составления программы), минимальная документация оформлялась уже после того, как программа начинала работать. Тем не менее, именно в этот период родилась фундаментальная для технологии программирования концепция модульного программирования [1.10], ориентированная на преодоления трудностей программирования в машинном коде. Появились первые языки программирования высокого уровня, из которых только ФОРТРАН пробился для использования в следующие десятилетия.

В 60-е годы можно было наблюдать бурное развитие и широкое использование языков программирования высокого уровня (АЛГОЛ 60, ФОРТРАН. КОБОЛ и др.), значение которых в технологии программирования явно преувеличивалась. Надежда на то, что эти языки решат все проблемы, возникающие в процессе разработки больших программ, не оправдалась. В результате повышения мощности компьютеров и накопления опыта программирования на языках высокого уровня быстро росла сложность решаемых на компьютерах задач, в результате чего обнаружилась ограниченность языков, проигнорировавших модульную организацию программ. И только ФОРТРАН, бережно сохранивший возможность модульного программирования, гордо прошествовал в следующие десятилетия (все его ругали, но его пользователи отказаться от его услуг не могли из-за грандиозного накопления фонда программных модулей, которые с успехом использовались в новых программах). Кроме того, было понято, что важно не только то, на каком языке мы программируем, но и то, как мы программируем [1.4]. Это было уже началом серьезных размышлений над методологией и технологией программирования. Появление в компьютерах 2-го поколения прерываний привело к развитию мультипрограммирования и созданию больших программных систем. Широко стала использоваться коллективная разработка, которая поставила ряд серьезных технологических проблем [1.1 1].

В 70-е годы получили широкое распространение информационные системы и базы данных. К середине 70-ых годов стоимость хранения одного бита информации на компьютерных носителях стала меньше, чем на традиционных носителях. Это резко повысило интерес к компьютерным системам хранения данных. Началось интенсивное развитие технологии программирования [1.2, 1.8. 1.12-1.14], прежде всего, в следующих направлениях:

- обоснование и широкое внедрение нисходящей разработки и структурного программирования.
- развитие абстрактных типов данных и модульного программирования (в частности, возникновение идеи разделения спецификации и реализации модулей и использование модулей, скрывающих структуры данных).
- исследование проблем обеспечения надежности и мобильности ПС.

• создание методики управления коллективной разработкой ПС.

• появление инструментальных программных средств (программных инструментов) поддержки технологии программирования.

80-е годы характеризуются широким внедрением персональных компьютеров во все сферы человеческой деятельности и тем самым созданием обширного и разнообразного контингента пользователей ПС. Это привело к бурному развитию пользовательских интерфейсов и созданию четкой концепции качества ПС [1.5, 1.15-1.18]. Появляются языки программирования (например, Ада), учитывающие требования технологии программирования [1-19]. Развиваются методы и языки спецификации ПС [1.20-1.21]. Выходит на передовые позиции объектный подход к разработке ПС [1.9]. Создаются различные инструментальные среды разработки и сопровождения ПС [1.3]. Развивается концепция компьютерных сетей.

90-е годы знаменательны широким охватом всего человеческого общества международной компьютерной сетью, персональные компьютеры стали подключаться к ней как терминалы. Это поставило ряд проблем (как технологического, так и юридического и этического характера) регулирования доступа к информации компьютерных сетей. Остро встала проблема защиты компьютерной информации и передаваемых по сети сообщений. Стали бурно развиваться компьютерная технология (CASE-технология) разработки ПС и связанные с ней формальные методы спецификации программ. Начался решающий этап полной информатизации и компьютеризации общества.

Упражнения к лекции 1.

1.1. Что такое информационная среда программы?
1 .2. Что такое программное средство (II С)?
1.3. Что такое ошибка в ПС ?
1.4. Что такое надежность ПС?
1.5. Что такое технология программирования?
1.6. Этапы развития технологии программирования.

Лекция 2.

ИСТОЧНИКИ ОШИБОК В ПРОГРАММНЫХ СРЕДСТВАХ

Интеллектуальные возможности человека, используемые при разработке программных систем. Понятия о простых и сложных системах, о малых и больших системах. Неправильный перевод информации іп одного представления в другое - основная причина ошибок при разработке программных средств. Модель перевода и источники ошибок.

2.1. Интеллектуальные возможности человека.

Дейкстра [2.1] выделяет три интеллектуальные возможности человека, используемые при разработке ПС:

• способность к перебору,
• способность к абстракции,
• способность к математической индукции.

Способность человека к перебору связана с возможностью последовательного переключения внимания с одного предмета на другой, позволяя *узнавать* искомый предмет. Эта способность весьма ограничена - в среднем человек может уверенно (не сбиваясь) перебирать в пределах 1000 предметов (элементов). Человек должен научиться действовать с учетом этой своей ограниченности. Средством преодоления этой ограниченности является его способность к абстракции, благодаря которой человек может объединять разные предметы или экземпляры в одно понятие, заменять множество элементов одним элементом (другого рода). Способность человека к математической индукции позволяет ему справляться с бесконечными последовательностями.

При разработке ПС человек имеет дело с системами. Под *системой* будем понимать совокупность взаимодействующих (находящихся в отношениях) друг с другом элементов. ПС можно рассматривать как пример системы. Логически связанный набор программ является другим примером системы. Любая отдельная программа также является системой. Понять

систему - значит осмысленно перебрать все пути взаимодействия между ее элементами. В силу ограниченности человека к перебору будем различать простые и сложные системы [2.2]. Под *простой* будем понимать такую систему, в которой человек может уверенно перебирать все пути взаимодействия между ее элементами, а под *сложной* будем понимать такую систему, в которой он этого делать не в состоянии. Между простыми и сложными системами нет четкой границы, поэтому можно говорить и о промежуточном классе систем: к таким системам относятся программы, о которых программистский фольклор утверждает, что "в каждой отлаженной программе имеется хотя бы одна ошибка".

При разработке ПС мы не всегда можем уверенно знать о всех связях между ее элементами из-за возможных ошибок. Поэтому полезно уметь оценивать сложность системы по числу ее элементов: числом потенциальных путей взаимодействия между ее элементами, т.е. n! , где n - число ее элементов. Систему назовем *малой,* если $n < 7$ ($6! = 720 < 1000$), систему назовем *большой,* если $n > 7$. При $n=7$ имеем промежуточный класс систем. Малая система всегда проста, а большая может быть как простой, так и сложной. Задача технологии программирования - научиться делать большие системы простыми.

Полученная оценка простых систем по числу элементов широко используется на практике. Так, для руководителя коллектива весьма желательно, чтобы в нем не было больше шести взаимодействующих между собой подчиненных. Весьма важно также следовать правилу: *«все, что может быть сказано, должно быть сказано в шести пунктах или меньше».* Этому правилу мы будем стараться следовать в настоящем пособии: всякие перечисления взаимосвязанных утверждений (набор рекомендаций, список требований и т.п.) будут соответствующим образом группироваться и обобщаться. Полезно ему следовать и при разработке ПС.

2.2. Неправильный перевод как причина ошибок в программных средствах.

При разработке и использовании ПС мы многократно имеем дело [2.3, стр. 22-28] с преобразованием (переводом) информации из одной формы в другую (см. рис.2.1). Заказчик формулирует свои потребности в ПС в виде некоторых требований. Исходя из этих требований, разработчик создает внешнее описание ПС, используя при этом спецификацию (описание) заданной аппаратуры и, возможно, спецификацию базового программного обеспечения. На основании внешнего описания и спецификации языка программирования создаются тексты программ ПС на этом языке. По внешнему описанию ПС разрабатывается также и пользовательская документация. Текст каждой программы является исходной информацией при любом ее преобразовании, в частности, при исправлении в ней ошибки. Пользователь на основании документации выполняет ряд действий для применения ПС и осуществляет интерпретацию получаемых результатов. Везде здесь, а также в ряде других процессах разработки ПС, имеет место указанный перевод информации.

На каждом из этих этапов перевод информации может быть осуществлен неправильно, например, из-за неправильного понимания исходного представления информации. Возникнув на одном из этапов разработки ПС, ошибка в представлении информации преобразуется в новые ошибки результатов, полученных на последующих этапах разработки, и, в конечном счете, окажется в ПС.

2.3. Модель перевода.

Чтобы понять природу ошибок при переводе рассмотрим модель [2.3, стр. 22-28], изображенную на рис.2.2. На ней человек осуществляет перевод информации из представления A в представление B. При этом он совершает четыре основных шага перевода:

- он получает информацию, содержащуюся в представлении A, с помощью своего читающего механизма R;
- он запоминает полученную информацию в своей памяти M;
- он выбирает из своей памяти преобразуемую информацию и информацию, описывающую процесс преобразования, выполняет перевод и посылает результат своему пишущему механизму W;
- с помощью этого механизма он фиксирует представление B.

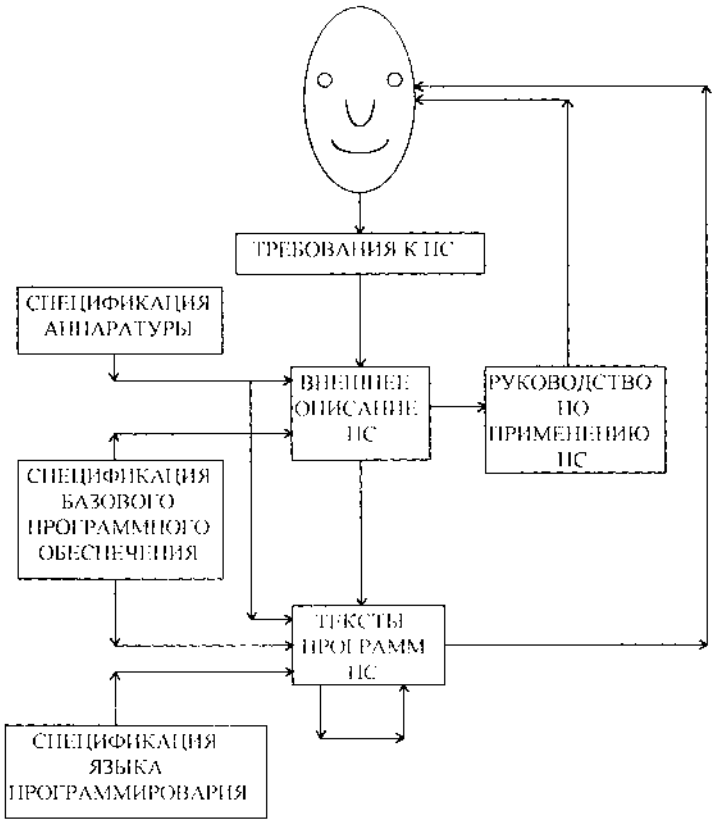


Рис. 2.1. Грубая схема разработки и применения ПС.

.

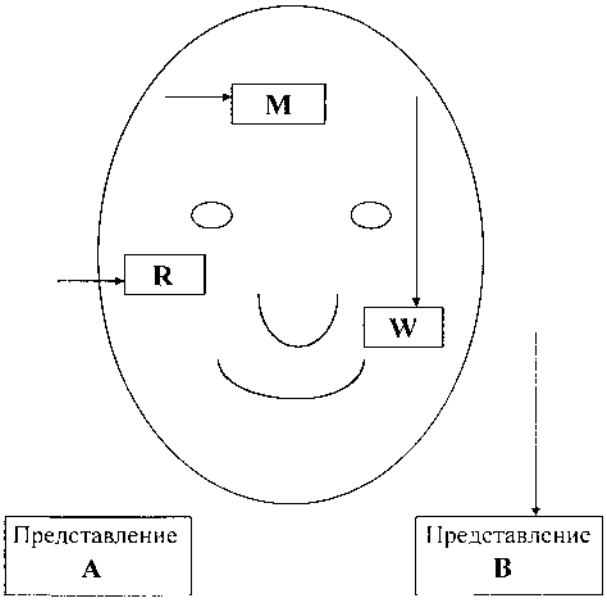


Рис 2.2. Модель перевода.

На каждом из этих шагов человек может совершить ошибку разной природы. На первом шаге способность человека "читать между строк" (способность, которая часто оказывается полезной, позволяя ему понимать текст, содержащий неточности или даже ошибки) может стать причиной ошибки в ПС. Ошибка возникает в том случае, когда при чтении документа А человек, пытаясь восстановить недостающую информацию, видит то, что он ожидает, а не то, что имел в виду автор документа А. В этом случае лучше было бы обратиться к автору документа за разъяснениями. При запоминании информации человек осуществляет ее осмысливание (здесь важен его уровень подготовки и знание предметной области, к которой относится документ А). И. если он поверхностно или неправильно поймет, то информация

будет запомнена в искаженном виде. На третьем этапе забывчивость человека может привести к тому, что он может выбрать из своей памяти не всю преобразуемую информацию или не все правила перевода, в результате чего перевод будет осуществлен неверно. Это обычно происходит при большом объеме плохо организованной информации. И, наконец, на последнем этапе стремление человека быстрее зафиксировать информацию часто приводит к тому, что представление этой информации оказывается неточным, создавая ситуацию для последующей неоднозначной ее интерпретации.

2.4. Основные пути борьбы с ошибками.

Учитывая рассмотренные особенности действий человека при переводе можно указать следующие пути борьбы с ошибками:

- сужение пространства перебора (упрощение создаваемых систем),
- обеспечение требуемого уровня подготовки разработчика (это функции менеджеров коллектива разработчиков),
- обеспечение однозначности интерпретации представления информации,
- контроль правильности перевода (включая и контроль однозначности интерпретации).

Упражнения к лекции 2.

2.1. Что такое простая и сложная системы?

2.2. Что такое малая и большая системы[9]

Лекция 3.

ОБЩИЕ ПРИНЦИПЫ РАЗРАБОТКИ ПРОГРАММНЫХ СРЕДСТВ

Специфика разработки программных средств. Жизненный цикл программного средства. Понятие качества программного средства. Обеспечение надежности - основной мотив разработки программного средства. Методы борьбы со сложностью. Обеспечение точности перевода. Преодоление барьера между пользователем и разработчиком. Обеспечение контроля правильности принимаемых решений.

3.1. Специфика разработки программных средств.

Разработка программных средств имеет ряд специфических особенностей [3.1].

- Прежде всего, следует отметить некоторое противостояние: неформальный характер требований к ПС (постановки задачи) и понятия ошибки в нем, но *формализованный* основной объект разработки - программы ПС. Тем самым разработка ПС содержит определенные этапы формализации, а переход от неформального к формальному существенно неформален.
- Разработка ПС носит *творческий характер* (на каждом шаге приходится делать какой-либо выбор, принимать какое-либо решение), а не сводится к выполнению какой-либо последовательности регламентированных действий. Тем самым эта разработка ближе к процессу проектирования каких-либо сложных устройств, но никак не к их массовому производству. Этот творческий характер разработки ПС сохраняется до самого ее конца.
- Следует отметить также особенность продукта разработки. Он представляет собой некоторую совокупность текстов (т.е. *статических* объектов), смысл же (семантика) этих текстов выражается процессами обработки данных и действиями пользователей, запускающих эти процессы (т.е. является *динамическим).* Это предопределяет выбор разработчиком ряда специфичных приемов, методов и средств.
- Продукт разработки имеет и другую специфическую особенность: ПС при своем использовании (эксплуатации) не расходуется и не расходует используемых ресурсов.

3.2. Жизненный цикл программного средства.

Под *жизненным циклом* ПС *(software life cycle)* понимают весь период его разработки и эксплуатации (использования), начиная от момента возникновения замысла ПС и кончая прекращением всех видов его использования [3.1-3.4]. Жизненный цикл охватывает довольно сложный процесс создания и использования ПС *(soffrvare process).* Этот процесс

может быть организован по- разному для разных классов ПС и в зависимости от особенностей коллектива разработчиков.

В настоящее время можно выделить 5 основных подходов к организации процесса создания и использования ПС [3.5].

• Водопадный подход. При таком подходе разработка ПС состоит из цепочки этапов. На каждом этапе создаются документы, используемые на последующем этапе. В исходном документе фиксируются требования к ПС. В конце этой цепочки создаются программы, включаемые в ПС.

• Исследовательское программирование. Этот подход предполагает быструю (насколько это возможно) реализацию рабочих версий программ ПС, выполняющих лишь в первом приближении требуемые функции. После экспериментального применения реализованных программ производится их модификация с целью сделать их более полезными для пользователей. Этот процесс повторяется до тех пор, пока ПС не будет достаточно приемлемо для пользователей. Такой подход применялся на ранних этапах развития программирования, когда технологии программирования не придавали большого значения (использовалась интуитивная технология). В настоящее время этот подход применяется для разработки таких ПС, для которых пользователи не могут точно сформулировать требования (например, для разработки систем искусственного интеллекта).

• Прототипирование. Этот подход моделирует начальную фазу исследовательского программирования вплоть до создания рабочих версий программ, предназначенных для проведения экспериментов с целью установить требования к ПС. В дальнейшем должна последовать разработка ПС по установленным требованиям в рамках какого-либо другого подхода (например, водопадного).

• Формальные преобразования. Этот подход включает разработку формальных спецификаций ПС и превращение их в программы путем корректных преобразований. На этом подходе базируется компьютерная технология (CASE-технология) разработки ПС.

• Сборочное программирование. Этот подход предполагает, что ПС конструируется, главным образом, из компонент, которые уже существуют. Должно быть некоторое хранилище (библиотека) таких компонент, каждая из которых может многократно использоваться в разных ПС. Такие компоненты называются *повторно используемыми (reusable]*. Процесс разработки ПС при данном подходе состоит скорее из сборки программ из компонент, чем из их программирования.

В нашем курсе лекций мы, в основном, будем рассматривать водопадный подход с некоторыми модификациями. Во-первых, потому, что в этом подходе приходиться иметь дело с большинством процессов программной инженерии, а, во-вторых, потому, что в рамках этого подхода создается большинство больших программных систем. Именно этот подход рассматривается в качестве индустриального подхода разработки программного обеспечения. Исследовательское программирование исходит из взгляда на программирование как на искусство. Оно применяется тогда, когда водопадный подход не применим из-за того, что не удается точно сформулировать требования к ПС. В нашем курсе мы этот подход рассматривать не будем. Прототипирование рассматривается как вспомогательный подход, используемый в рамках других подходов, в основном, для прояснения требований к ПС. Компьютерной технологии (включая обсуждение жизненного цикла ПС, созданного по этой технологии) будет посвящена отдельная лекция. Сборочное программирование мы в нашем курсе рассматривать не будем, хотя о повторно используемых программных модулях мы говорить будем, обсуждая свойства программных модулей.

В рамках водопадного подхода различают следующие стадии жизненного цикла ПС (см. рис. 3.1): разработку ПС, производство программных изделий (ПИ) и эксплуатацию ПС.

Стадия *разработки (development)* ПС состоит из этапа его внешнего описания, этапа конструирования ПС, этапа кодирования (программирование в узком смысле) ПС и

этапа аттестации ПС. Всем этим этапам сопутствуют процессы документирования и управления *(management)* ПС. Этапы конструирования и кодирования часто перекрываются, иногда довольно сильно. Это означает, что кодирование некоторых частей программного средства может быть начато до завершения этапа конструирования.

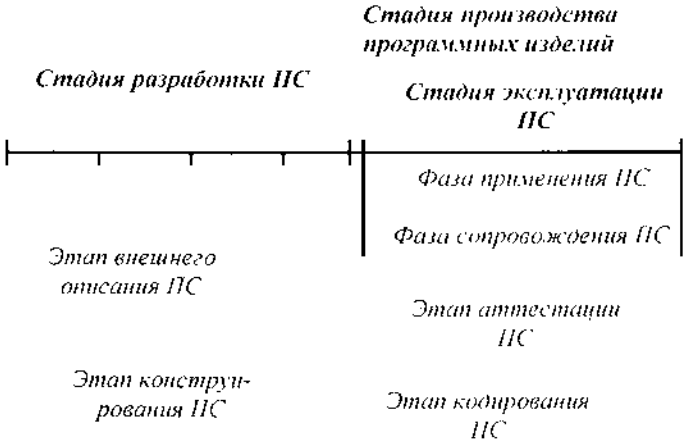


Рис. 3.1. Стадии и фазы жизненного цикла ПС.

Этап *внешнего описания* ПС включает процессы, приводящие к созданию некоторого документа, который мы будем называть *внешним описанием (requirements document) ПС.* Этот документ является описанием поведения ПС с точки зрения внешнего по отношению к нему наблюдателя с фиксацией требований относительно его качества. Внешнее описание ПС начинается с анализа и определения требований к ПС со стороны пользователей (заказчика), а также включает процессы спецификации этих требований. *Конструирование (design)* ПС охватывает процессы: разработку архитектуры ПС, разработку структур программ ПС и их детальную спецификацию.

Кодирование (coding) ПС включает процессы создания текстов программ на языках программирование, их отладку с тестированием ПС.

На этапе *аттестации (acceptance)* ПС производится оценка качества ПС. Если эта оценка оказывается приемлемой для практического использования ПС, то разработка ПС считается законченной. Это обычно оформляется в виде некоторого документа, фиксирующего решение комиссии, проводящей аттестацию ПС.

Программное изделие (ПИ) - экземпляр или копия разработанного ПС. *Изготовление* ПИ - это процесс генерации и/или воспроизведения (снятия копии) программ и программных документов ПС с целью их поставки пользователю для применения по назначению. *Производство* ПИ - это совокупность работ по обеспечению изготовления требуемого количества ПИ в установленные сроки [3.1]. Стадия производства ПИ в жизненном цикле ПС является, по-существу, вырожденной (не существенной), так как представляет рутинную работу, которая может быть выполнена автоматически и без ошибок. Этим она принципиально отличается от стадии производства различной техники. В связи с этим в литературе эту стадию, как правило, не включают в жизненный цикл ПС.

Стадия *эксплуатации* ПС охватывает процессы хранения, внедрения и сопровождения ПС, а также транспортировки и применения ПИ по своему назначению. Она состоит из двух параллельно проходящих фаз: фазы применения ПС и фазы сопровождения ПС [3.4, 3.5].

Применение (operation) ПС - это использование ПС для решения практических задач на компьютере путем выполнения ее программ.

Сопровождение (maintenance) ПС - это процесс сбора информации качестве ПС в эксплуатации, устранения обнаруженных в нем ошибок, его доработки и модификации, а также извещения пользователей о внесенных в него изменениях [3.1,3.4,3.5].

3.3. Понятие качества программного средства.

Каждое ПС должно выполнять определенные функции, т.е. делать то, что задумано. Хорошее ПС должно обладать еще целым рядом свойств, позволяющим успешно его использовать в течении длительного периода, т.е. обладать определенным качеством. *Качество (quality)* ПС - это совокупность его черт и характеристик, которые влияют на его способность удовлетворять заданные потребности пользователей [3.6]. Это не означает, что разные ПС должны обладать одной и той же совокупностью таких свойств в их наивысшей степени. Этому препятствует тот факт, что повышение качества ПС по одному из таких свойств часто может быть достигнуто лишь пеной изменения стоимости, сроков завершения разработки и снижения качества этого ПС по другим его свойствам. Качество ПС является удовлетворительным, когда оно обладает указанными свойствами в такой степени, чтобы гарантировать успешное его использование.

Совокупность свойств ПС. которая образует удовлетворительное для пользователя качество ПС, зависит от условий и характера эксплуатации этого ПС. т.е. от позиции, с которой должно рассматриваться качество этого ПС. Поэтому при описании качества ПС, прежде всего, должны быть фиксированы *критерии* отбора требуемых свойств ПС. В настоящее время *критериями качества* //С *(criteria ofsofnvare quality')* принято считать [3.6-3.10]:

- функциональность,
- надежность,
- легкость применения,
- эффективность,
- сопровождаемость,
- мобильность.

Функциональность - это способность ПС выполнять набор функций, удовлетворяющих заданным или подразумеваемым потребностям пользователей. Набор указанных функций определяется во внешнем описании ПС.

Надежность подробно обсуждалась в первой лекции.

Легкость применения - это характеристики ПС, которые позволяют минимизировать усилия пользователя по подготовке исходных данных, применению ПС и оценке полученных результатов, а также вызывать положительные эмоции определенного или подразумеваемого пользователя.

Эффективность - это отношение уровня услуг, предоставляемых ПС пользователю при заданных условиях, к объему используемых ресурсов.

Сопровождаемость - это характеристики ПС, которые позволяют минимизировать усилия по внесению изменений для устранения в нем ошибок и по его модификации в соответствии с изменяющимися потребностями пользователей.

Мобильность - это способность ПС быть перенесенным из одной среды (окружения) в другую, в частности, с одной ЭВМ на другую.

Функциональность и надежность являются обязательными критериями качества ПС, причем обеспечение надежности будет красной нитью проходить по всем этапам и процессам разработки ПС. Остальные критерии используются в зависимости от потребностей пользователей в соответствии с требованиями к ПС - их обеспечение будет обсуждаться в подходящих разделах курса.

3.4. Обеспечение надежности - основной мотив разработки программных средств.

Рассмотрим теперь общие принципы обеспечения надежности ПС, что, как мы уже подчеркивали, является основным мотивом разработки ПС, задающим специфическую окраску всем технологическим процессам разработки ПС. В технике известны четыре подхода обеспечению надежности [3.1 1]:

- предупреждение ошибок;
- самообнаружение ошибок;
- самоисправление ошибок;

• обеспечение устойчивости к ошибкам.

Целью подхода предупреждения ошибок - не допустить ошибок в готовых продуктах, в нашем случае - в ПС. Проведенное рассмотрение природы ошибок при разработке ПС позволяет для достижения этой цели сконцентрировать внимание на следующих вопросах:

• борьба со сложностью,
• обеспечение точности перевода,
• преодоление барьера между пользователем и разработчиком,
• обеспечение контроля принимаемых решений.

Этот подход связан с организацией процессов разработки ПС, т.е. с технологией программирования. И хотя, как мы уже отмечали, гарантировать отсутствие ошибок в ПС невозможно, но в рамках этого подхода можно достигнуть приемлемого уровня надежности ПС.

Остальные три подхода связаны с организацией самих продуктов технологии, в нашем случае - программ. Они учитывают возможность ошибки в программах. Самообнаружение ошибки в программе означает, что программа содержит средства обнаружения отказа в процессе ее выполнения. Самоисправление ошибки в программе означает не только обнаружение отказа в процессе ее выполнения, но и исправление последствий этого отказа, для чего в программе должны иметься соответствующие средства. Обеспечение устойчивости программы к ошибкам означает, что в программе содержатся средства, позволяющие локализовать область влияния отказа программы, либо уменьшить его неприятные последствия, а иногда предотвратить катастрофические последствия отказа. Однако, эти подходы используются весьма редко (может быть, относительно чаще используется обеспечение устойчивости к ошибкам). Связано это, во-первых, с тем. что многие простые методы, используемые в технике в рамках этих подходов, неприменимы в программировании, например, дублирование отдельных блоков и устройств (выполнение двух копий одной и той же программы всегда будет приводить к одинаковому эффекту - правильному или неправильному). А, во-вторых, добавление в программу дополнительных фрагментов приводит к ее усложнению (иногда - значительному), что в какой-то мере мешает методам предупреждения ошибок.

3.5. Методы борьбы со сложностью.

Мы уже обсуждали в лекции 2 сущность вопроса борьбы со сложностью при разработке ПС. Известны два общих метода борьбы со сложностью систем:

• обеспечения независимости компонент системы;
• использование в системах иерархических структур.

Обеспечение независимости компонент означает разбиение системы на такие части, между которыми должны остаться по возможности меньше связей. Одним из воплощений этого метода является модульное программирование. Использование в системах иерархических структур позволяет локализовать связи между компонентами, допуская их лишь между компонентами, принадлежащими смежным уровням иерархии. Этот метод, по-существу, означает разбиение большой системы на подсистемы, образующих малую систему. Здесь существенно используется способность человека к абстрагированию.

3.6. Обеспечение точности перевода.

Обеспечение точности перевода направлено на достижение однозначности интерпретации документов различными разработчиками, а также пользователями ПС. Это требует придерживаться при переводе определенной дисциплины. Майерс предлагает использовать общую дисциплину решения задач, рассматривая перевод как решение задачи [3.11]. Лучшим руководством по решению задач он считает книгу Пойа "Как решать задачу" [3.12]. В соответствии с этим весь процесс перевода можно разбить на следующие этапы:

• Поймите задачу;
• Составьте план (включая цели и методы решения);
• Выполните план (проверяя правильность каждого шага);

• Проанализируйте полученное решение.

3.7. Преодоление барьера между пользователем и разработчиком.

Как обеспечить, чтобы ПС выполняла то, что пользователю разумно ожидать от нее? Для этого необходимо правильно понять, во-первых, чего хочет пользователь, и, во-вторых, его уровень подготовки и окружающую его обстановку. При разработке ПС следует привлекать пользователя для участия в процессах принятия решений, а также тщательно освоить особенности его работы (лучше всего - побывать в его "шкуре").

3.8. Контроль принимаемых решений.

Обязательным шагом в каждом процессе (этапе) разработки ПС должна быть проверка правильности принятых решений. Это позволит обнаруживать и исправлять ошибки на самой ранней стадии после ее возникновения, что, во-первых, существенно снижает стоимость ее исправления и, во-вторых, повышает вероятность правильного ее устранения.

С учетом специфики разработки ПС необходимо применять везде, где это возможно,

• смежный контроль,

• сочетание как статических, так и динамических методов контроля.

Смежный контроль означает, проверку полученного документа лицами, не участвующими в его разработке, с двух сторон: во-первых, со стороны автора исходного для контролируемого процесса документа, и, во-вторых, лицами, которые будут использовать полученный документ в качестве исходного в последующих технологических процессах. Такой контроль позволяет обеспечивать однозначность интерпретации полученного документа.

Сочетание статических и динамических методов контроля означает, что нужно не только контролировать документ как таковой, но и проверять, какой процесс обработки данных он описывает. Это отражает одну из специфических особенность ПС (статическая форма, динамическое содержание).

Упражнения к лекции 3.

3.1. Что такое жизненный цикл программного средства (ПС)?

3.2. Что такое внешнее описание ПС?

3.3. Что такое сопровождение ПС?

3.4. Что такое качество ПС'?

5.1. Что такое смежный контроль?

Лекция 4.

ВНЕШНЕЕ ОПИСАНИЕ ПРОГРАММНОГО СРЕДСТВА

Понятие внешнего описания, его назначение и роль в обеспечении качества программного средства. Определение требований к программному средству. Спецификация качества программного средства. Основные примитивы качества программного средства. Функциональная спецификация программного средства. Контроль внешнего описания.

4.1. Назначение внешнего описания программного средства и его роль в обеспечении качества программного средства.

Разработчикам больших программных средств приходится решать весьма специфические и трудные проблемы, особенно, если это ПС должно представлять собой программную систему нового типа, в плохо компьютеризированной предметной области. Разработка ПС начинается с процесса формулирования требований к ПС, в котором, исходя из довольно смутных пожеланий заказчика, должен быть получен документ, достаточно точно определяющий задачи разработчиков ПС. Этот документ мы называем *внешним описанием ПС.*

Очень часто требования к ПС путают с требованиями к процессам его разработки (технологическим процессам). Последние не следует включать во внешнее описание, если только они не связаны с оценкой качества ПС. В случае необходимости требования к технологическим процессам можно оформить в виде самостоятельного документа, который будет использоваться при управлении разработкой ПС.

Внешнее описание ПС играет роль точной постановки задачи, решение которой должно обеспечить разрабатываемое ПС. Более того, оно должно содержать всю информацию, которую необходимо знать пользователю для применения ПС. Оно является исходным документом для трех параллельно протекающих процессов: разработки текстов (конструированию и кодированию) программ, входящих в ПС, разработки документации по применению ПС и разработки существенной части комплекта тестов для тестирования ПС. Ошибки и неточности во внешнем описании, в конечном счете, трансформируются в ошибки самой ПС и обходятся особенно дорого, во-первых, потому, что они делаются на самом раннем этапе разработки ПС, и, во-вторых, потому, что они распространяются на три параллельных процесса. Это требует принятия особенно серьезных мер по их предупреждению.

Исходным документом для разработки внешнего описания ПС являются *определение требовании* к ПС. Но так как через этот документ передается от заказчика (пользователя) к разработчику основная информация относительно требуемого ПС, то формирование этого документа представляет собой довольно длительный и трудный итерационный процесс взаимодействия между заказчиком и разработчиком, с которого и начинается этап разработки требований к ПС [4.2]. Трудности, возникающие в этом процессе, связаны с тем, что пользователи часто плохо представляют, что им на самом деле нужно: использование компьютера в "узких" местах деятельности пользователей может на самом деле потребовать принципиального изменения всей технологии этой деятельности (о чем пользователи, как правило, и не догадываются). Кроме того, проблемы, которые необходимо отразить в определении требований, могут не иметь определенной формулировки [4.1], что приводит к постепенному изменению понимания разработчиками этих проблем. В связи с этим определению требований часто предшествует процесс *системного анализа,* в котором выясняется, насколько целесообразно и реализуемо "заказываемое" ПС, как повлияет такое ПС на деятельность пользователей и какими особенностями оно должно обладать. Для выявления действительных потребностей пользователей иногда приходится разрабатывать упрощенную версию требуемого ПС, называемую *прототипом* ПС. Анализ "пробного" применения прототипа позволяет иногда существенно уточнить требования к ПС.

В определении внешнего описания легко бросаются в глаза две самостоятельные его части. Описание поведения ПС определяет функции, которые должна выполнять ПС, и потому его называют *функциональной спецификацией* ПС. Функциональная спецификация определяет допустимые фрагменты программ, реализующих декларированные функции. Требования к качеству ПС должны быть сформулированы так, чтобы разработчику были ясны цели [4.2], которые он должен стремиться достигнуть при разработке этого ПС. Эту часть внешнего описания будем называть *спецификацией качества* ПС (в литературе ее часто называют *нефункциональной спецификацией* [4.1], но она, как правило, включает и требования к технологическим процессам). Она, в отличие от функциональной спецификации, представляется в неформализованном виде и играет роль тех ориентиров, которые в значительной степени определяют выбор подходящих альтернатив при реализации функций ПС, а также определяет стиль всех документов и программ требуемого ПС. Тем самым, спецификация качества играет решающую роль в обеспечении требуемого качества ПС.

Обычно разработка спецификации качества предшествует разработке функциональной спецификации ПС, так как некоторые требования к качеству ПС могут предопределять включение в функциональную спецификацию специальных функций, например, функции защиты от несанкционированного доступа к некоторым объектам информационной среды. Таким образом, структуру внешнего описания ПС можно выразить формулой:

Внешнее описание ПС = определение требований
+ спецификация качества ПС
+ функциональная спецификация ПС

Таким образом, внешнее описание определяет, что должно делать ПС и какими внешними свойствами оно должно обладать. Оно не отвечает на вопросы, как должно быть устроено это ПС и как обеспечить требуемые его внешние свойства. Оно должно достаточно точно и полно определять задачи, которые должны решить разработчики требуемого ПС. В то же время оно должно быть понято представителем пользователем - на его основании заказчиком достаточно часто принимается окончательное решение на заключение договора на разработку ПС. Внешнее описание играет большую роль в обеспечении требуемого качества ПС, так как спецификация качества ставит для разработчиков ПС конкретные ориентиры, управляющие выбором приемлемых решений при реализации специфицированных функций.

4.2. Определение требований к программному средству.

Определение требования к ПС являются исходным документом разработки ПС - заданием, отражающем в абстрактной форме потребности пользователя. Они в общих чертах определяют замысел ПС, характеризуют условия его использования. Неправильное понимание потребностей пользователя трансформируются в ошибки внешнего описания. Поэтому разработка ПС начинается с создания документа, достаточно полно характеризующего потребности пользователя и позволяющего разработчику адекватно воспринимать эти потребности.

Определение требований представляет собой смесь фрагментов на естественном языке, различных таблиц и диаграмм. Такая смесь, должна быть понятной пользователю, не ориентирующемуся в специальных программистских понятиях. Обычно в определении требований не содержится формализованных фрагментов, кроме случаев достаточно для этого подготовленных пользователей (например, математически) - формализация этих требований составляет содержание дальнейшей работы коллектива разработчиков.

Неправильное понимание требований заказчиком, пользователями и разработчиками связано обычно с различными взглядами на роль требуемого ПС в среде его использования [4.1]. Поэтому важной задачей при создании определения требований является установление контекста использования ПС, включающего связи между этим ПС, аппаратурой и людьми. Лучше всего этот контекст в определении требований представить в графической форме (в виде диаграмм) с добавлением описаний сущностей используемых объектов (блоков ПС, компонент аппаратуры, персонала и т.п.) и характеристики связей между ними.

Известны три способа разработки определения требований к ПС [4.2]:

- управляемая пользователем разработка,
- контролируемая пользователем разработка,
- независимая от пользователя разработка.

В *управляемой пользователем* разработке определения требований к ПС определяются заказчиком, представляющим организацию пользователей. Это происходит обычно в тех случаях, когда организация пользователей (заказчик) заключает договор на разработку требуемого ПС с коллективом разработчиков и требования к ПС являются частью этого договора. Роль разработчика ПС в создании этих требований сводится, в основном, в выяснении того, насколько понятны ему эти требования с соответствующей критикой рассматриваемого документа. Это может приводить к созданию нескольких редакций этого документа в процессе заключения указанного договора.

В *контролируемой пользователем* разработке требования к ПС формулируются разработчиком при участии представителя пользователей. Роль пользователя в этом случае сводится к информированию разработчика о своих потребностях в ПС, а также к контролю того, чтобы формулируемые требования действительно выражали его потребности в ПС. Разработанные требования, как правило, утверждаются представителем пользователя.

В *независимой от пользователя* разработке требования к ПС определяются без какого-либо участия пользователя (на полную ответственность разработчика). Это происходит обычно тогда, когда разработчик решает создать ПС широкого применения в расчете на то, разработанное им ПС найдет спрос на рынке программных средств.

С точки зрения обеспечения надежности ПС наиболее предпочтительным является контролируемая пользователем разработка.

4.3. Спецификация качества программного средства.

Разработка спецификации качества сводится, по существу, к построению своеобразной модели качества требуемого ПС [4.2, 4.3]. В этой модели должен быть перечень всех тех достаточно элементарных свойств, которые необходимо обеспечить в требуемом ПС и которые в совокупности образуют приемлемое для пользователя качество ПС. При этом каждое из этих свойств должно быть в достаточной степени конкретизировано с учетом определения требований к ПС и возможности оценки его наличия у разработанного ПС или необходимой степени обладания им этим ПС.

Для конкретизации качества ПС по каждому из критериев используется стандартизованный набор достаточно простых свойств ПС [4.3-4.6], однозначно интерпретируемых разработчиками. Такие свойства мы будем называть *примитивами качества* ПС. Некоторые из примитивов могут использоваться по нескольким критериям. Ниже приводится зависимость критериев качества от примитивов качества ПС.

Функциональность: завершенность.

Надежность: завершенность, точность, автономность, устойчивость, защищенность.

Легкость применения: П-документированность, информативность (только применительно к документации по применению), коммуникабельность, устойчивость, защищенность.

Эффективность: временная эффективность, эффективность по ресурсам (по памяти), эффективность по устройствам.

Сопровождаемостъ: С данным критерием связано много различных примитивов качества. Однако их можно распределить по двум группам, выделив два подкритерия качества: изучаемость и модифицируемость.

Изучаемюсть - это характеристики ПС, которые позволяют минимизировать усилия по изучению и пониманию программ и документации ПС.

Модифицируемость - это характеристики ПС, которые позволяют автоматически настраивать на условия применения ПС или упрощают внесение в него вручную необходимых изменений и доработок.

Изучаемость: С-документированность, информативность (здесь применительно к документации по сопровождению), понятность, структурированность, удобочитаемость.

Модифицируемость: расширяемость, модифицируемость (в узком смысле, как примитив качества), структурированность, модульность.

Мобильность: независимость от устройств, автономность, структурированность, модульность.

Ниже даются определения используемых примитивов качества ПС [4.3-4.5].

Завершенность (completeness) свойство, характеризующее степень обладания ПС всеми необходимыми частями и чертами, требующимися для выполнения своих явных и неявных функций.

Точность (accuracy) - мера, характеризующая приемлемость величины погрешности в выдаваемых программами ПС результатах с точки зрения предполагаемого их использования.

Автономность (self-containedness) свойство, характеризующее способность ПС выполнять предписанные функции без помощи или поддержки других компонент программного обеспечения.

Устойчивость (robustness) - свойство, характеризующее способность ПС продолжать корректное функционирование, несмотря на задание неправильных (ошибочных) входных данных.

Защищенность (defensiveness) - свойство, характеризующее способность ПС противостоять преднамеренным или нечаянным деструктивным (разрушающим) действиям пользователя.

П-документированность (u. documentation) - свойство, характеризующее наличие, полноту, понятность, доступность и наглядность учебной, инструктивной и справочной документации, необходимой для применения ПС.

Информативность (accountability) - свойство, характеризующее наличие в составе ПС информации, необходимой и достаточной для понимания назначения ПС, принятых предположений, существующих ограничений, входных данных и результатов работы отдельных компонент, а также текущего состояния программ в процессе их функционирования.

Коммуникабельность (communicativeness) - свойство, характеризующее степень, в которой ПС облегчает задание или описание входных данных, и способность выдавать полезные сведения в достаточно простой форме и с простым для понимания содержанием.

Временная эффективность (time efficiency) -- мера, характеризующая способность ПС выполнять возложенные на него функции в течение определенного отрезка времени.

Эффективность по ресурсам (resource efficiency} - мера, характеризующая способность ПС выполнять возложенные на него функции при определенных ограничениях на используемые ресурсы (используемую память).

Эффективность по устройствам (device efficiency) мера, характеризующая экономичность использования устройств машины для решения поставленной задачи.

С-документированнностъ (documentation) — свойство, характеризующее с точки зрения наличия документации, отражающей требования к ПС и результаты различных этапов разработки данной ПС, включающие возможности, ограничения и другие черты ПС, а также их обоснование.

Понятность (understandability) - свойство, характеризующее степень, в которой ПС позволяет изучающему его лицу понять его назначение, сделанные допущения и ограничения, входные данные и результаты работы его программ, тексты этих программ и состояние их реализации. Этот примитив качества синтезирован нами из таких примитивов ИСО [4.4], как согласованность, самодокументированность, четкость и, собственно, понятность (текстов программ).

Структурированность (structuredness) свойство, характеризующее программы ПС с точки зрения организации взаимосвязанных их частей в единое целое определенным образом (например, в соответствии с принципами структурного программирования).

Удобочитаемость (readability) - свойство, характеризующее легкость восприятия текста программ ПС (отступы, фрагментация, форматированность).

Расширяемость (augmentability) - свойство, характеризующее способность ПС к использованию большего объема памяти для хранения данных или расширению функциональных возможностей отдельных компонент.

Модифицируемость (modisability) - мера, характеризующая ПС с точки зрения простоты внесения необходимых изменений и доработок на всех этапах и стадиях жизненного цикла ПС.

Модульность (modularity} - свойство, характеризующее ПС с точки зрения организации его программ из таких дискретных компонент, что изменение одной из них оказывает минимальное воздействие на другие компоненты.

Независимость от устройств (device independence) свойство, характеризующее способность ПС работать на разнообразном аппаратном обеспечении (различных типах, марках, моделях ЭВМ).

4.4. Функциональная спецификация программного средства.

С учетом назначения функциональной спецификации ПС и тяжелых последствий неточностей и ошибок в этом документе, функциональная спецификация должна быть математически точной. Это не означает, что она должна быть формализована настолько, что

по ней можно было бы автоматически генерировать программы, решающие поставленную задачу. А означает лишь, что она должна базироваться на понятиях, построенных как математические объекты, и утверждениях, однозначно понимаемых разработчиками ПС. Достаточно часто функциональная спецификация формулируется на естественном языке. Тем не менее, использование математических методов и формализованных языков при разработке функциональной спецификации весьма желательно, поэтому этим вопросам будет посвящена отдельная лекция.

Функциональная спецификация состоит из трех частей:

- описания внешней информационной среды, к которой должны применяться программы разрабатываемой ПС;
- определение функций ПС, определенных на множестве состояний этой информационной среды (такие функции будем называть *внешними функциями* ПС);
- описание нежелательных (исключительных) ситуаций, которые могут возникнуть при выполнении программ ПС, и реакций на эти ситуации, которые должны обеспечить соответствующие программы.

В первой части должны быть определены на концептуальном уровне все используемые каналы ввода и вывода и все информационные объекты, к которым будет применяться разрабатываемое ПС, а также существенные связи между этими информационными объектами. Примером описания информационной среды может быть концептуальная схема базы данных или описание сети датчиков и приборов, которой должна управлять разрабатываемая ПС.

Во второй части вводятся обозначения всех определяемых функций, специфицируются все входные данные и результаты выполнения каждой определяемой функции, включая указание их типов и заданий всех соотношений (или ограничений), которым должны удовлетворять эти данные и результаты. И, наконец, определяется семантика каждой из этих функций, что является наиболее трудной задачей функциональной спецификации ПС. Обычно эта семантика описывается неформально на естественном языке - примерно так, как это делается при описании семантики многих языков программирования. Эта задача может быть в ряде случаев существенно облегчена при достаточно четком описании внешней информационной среды, если внешние функции задают какие-либо манипуляции с ее объектами.

В третьей части должны быть перечислены все существенные случаи, когда ПС не сможет нормально выполнить ту или иную свою функцию (с точки зрения внешнего наблюдателя). Примером такого случая может служить обнаружение ошибки во время взаимодействия с пользователем, или попытка применить какую-либо функцию к данным, не удовлетворяющим соотношениям, указанным в ее спецификации, или получение результата, нарушающего заданное ограничение. Для каждого такого случая должна быть определена (описана) реакция ПС.

4.5. Методы контроля внешнего описания программного средства.

Разработка внешнего описания обязательно должна завершаться проведением тщательного и разнообразного контроля правильности внешнего описания. Целью этого процесса является найти как можно больше ошибок, сделанных на этом этапе. Учитывая, что результатом этого этапа является, как правило, еще неформализованный текст, то здесь на первый план выступают психологические факторы контроля. Можно выделить следующие методы контроля, применяемые на этом этапе:

- статический просмотр,
- смежный контроль,
- пользовательский контроль,
- ручная имитация.

Первый метод предполагает внимательное прочтение текста внешнего описания разработчиком с целью проверка его полноты и непротиворечивости, а также выявления других неточностей и ошибок.

Смежный контроль спецификации качества сверху - это ее проверка со стороны разработчика требований к ПС, а смежный контроль функциональной спецификации это ее проверка разработчиками требований к ПС и спецификации качества. Смежный контроль внешнего описания снизу - это его изучение и проверка разработчиками архитектуры ПС и текстов программ, а также разработчиками документации по применению и разработчиками комплекта тестов.

Пользовательский контроль внешнего описания выражает участие пользователя (заказчика) в принятии решений при разработке внешнего описания и его контроле. Если разработка требований к ПС велась под управлением пользователя, то пользовательский контроль внешнего описания, по существу, означает его смежный контроль сверху. Однако, если представителю пользователя оказывается трудно самостоятельно разобраться во внешнем описании, создается специальная группа разработчиков, выполняющая роль пользователя (и взаимодействующая с ним) для проведения такого контроля.

Ручная имитация выражает своеобразный динамический контроль внешнего описания, точнее говоря, функциональной спецификации ПС. Для этого необходимо подготовить исходные данные (тесты) и на основании функциональной спецификации осуществить имитацию поведения (работы) разрабатываемого ПС. При этом такую имитацию осуществляет специально назначенный разработчик, выполняющий, по существу, роль будущих программ ПС. Разновидностью такого контроля является имитация за терминалом. В этом случае данные вводятся в компьютер человеком, играющего роль пользователя, и передаются с помощью несложной программы на другой терминал, за которым сидит разработчик, выполняющий роль программ ПС. Полученные результаты передаются через компьютер на первый терминал.

Упражнения к лекции 4.

4.1. Что такое устойчивость (robustness) программного средства (ПС)?

4.2. Что такое защищенность (defensiveness) ПС?

4.3. Что такое коммуникабельность (communicativeness) ПС?

Лекция 5.

АРХИТЕКТУРА ПРОГРАММНОГО СРЕДСТВА

Понятие архитектуры и задачи ее описания. Основные классы архитектур программных средств. Взаимодействие между подсистемами и архитектурные функции. Контроль архитектуры программных средств.

5.1. Понятие архитектуры программного средства.

Архитектура ПС - это его строение как оно видно (или должно быть видно) из-вне его, т.е. представление ПС как системы, состоящей из некоторой совокупности взаимодействующих подсистем. В качестве таких подсистем выступают обычно отдельные программы. Разработка архитектуры является первым этапом борьбы со сложностью ПС, на котором реализуется принцип выделения относительно независимых компонент.

Основные задачи разработки архитектуры ПС:

- Выделение программных подсистем и отображение на них внешних функций (заданных во внешнем описании) ПС;
- определение способов взаимодействия между выделенными программными подсистемами.

С учетом принимаемых на этом этапе решений производится дальнейшая конкретизация и функциональных спецификаций.

5.2. Основные классы архитектур программных средств.

Различают следующие основные классы архитектур программных средств [5.1]:

- цельная программа;
- комплекс автономно выполняемых программ;

- слоистая программная система;
- коллектив параллельно выполняемых программ.

Цельная программа представляет вырожденный случай архитектуры ПС: в состав ПС входит только одна программа. Такую архитектуру выбирают обычно в том случае, когда ПС должно выполнять одну какую-либо ярко выраженную функцию и ее реализация не представляется слишком сложной. Естественно, что такая архитектура не требует какого-либо описания (кроме фиксации класса архитектуры), так как отображение внешних функций на эту программу тривиально, а определять способ взаимодействия не требуется (в силу отсутствия какого-либо внешнего взаимодействия программы, кроме как взаимодействия ее с пользователем, а последнее описывается в документации по применению ПС).

Комплекс автономно выполняемых программ состоит из набора программ, такого, что:

- любая из этих программ может быть активизирована (запущена) пользователем;
- при выполнении активизированной программы другие программы этого набора не могут быть активизированы до тех пор, пока не закончит выполнение активизированная программа;
- все программы этого набора применятся к одной и той же информационной среде.

Таким образом, программы этого набора по управлению не взаимодействуют - взаимодействие между ними осуществляется только через общую информационную среду.

Слоистая программная система состоит из некоторой упорядоченной совокупности программных подсистем, называемых *слоями,* такой, что:

- на каждом слое ничего не известно о свойствах (и даже существовании) последующих (более высоких) слоев;
- каждый слой может взаимодействовать по управлению (обращаться к компонентам) с непосредственно предшествующим (более низким) слоем через заранее определенный интерфейс, ничего не зная о внутреннем строении всех предшествующих слоев;
- каждый слой располагает определенными ресурсами, которые он либо скрывает от других слоев, либо предоставляет непосредственно последующему слою (через указанный интерфейс) некоторые их абстракции.

Таким образом, в слоистой программной системе каждый слой может реализовать некоторую абстракцию данных. Связи между слоями ограничены передачей значений параметров обращения каждого слоя к смежному снизу слою и выдачей результатов этого обращения от нижнего слоя верхнему. Недопустимо использование глобальных данных несколькими слоями.

В качестве примера рассмотрим использование такой архитектуры для построения операционной системы. Такую архитектуру применил Дейкстра при построении операционной системы THE [5.2]. Эта операционная система состоит из четырех слоев (см. рис. 5.1). На нулевом слое производится обработка всех прерываний и выделение центрального процессора программам (процессам) в пакетном режиме. Только этот уровень осведомлен о мультипрограммных аспектах системы. На первом слое осуществляется управление страничной организацией памяти. Всем вышестоящим слоям предоставляется виртуальная непрерывная (не страничная) память. На втором слое осуществляется связь с консолью (пультом управления) оператора. Только этот слой знает технические характеристики консоли. На третьем слое осуществляется буферизация входных и выходных потоков данных и реализуются так называемые абстрактные каналы ввода и вывода, так что прикладные программы не знают технических характеристик ввода и вывода.

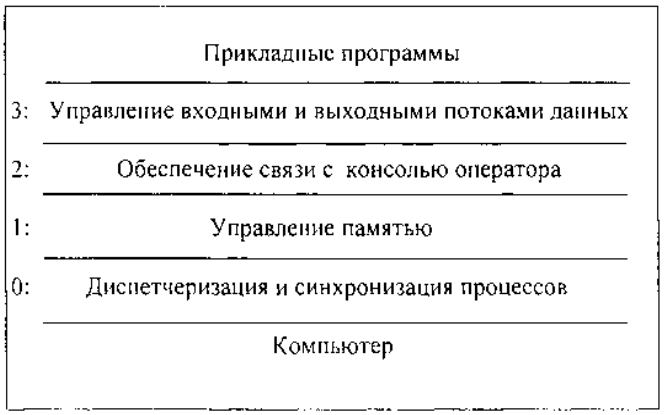


Рис. 5.1. Архитектура операционной системы THE.

Коллектив параллельно действующих программ представляет собой набор программ, способных взаимодействовать между собой, находясь одновременно в стадии выполнения. Это означает, что такие программы, во-первых, вызваны в оперативную память, активизированы и могут попеременно разделять по времени один или несколько центральных процессоров, а во-вторых, осуществлять между собой динамические (в процессе выполнения) взаимодействия, на базе которых производиться их синхронизация. Обычно взаимодействие между такими процессами производится путем передачи друг другу некоторых сообщений.

Простейшей разновидностью такой архитектуры является конвейер. Возможности для организации конвейера имеются, например, в операционной системе UNIX [5.3]. *Конвейер* представляет собой последовательность программ, в которой стандартный вывод каждой программы, кроме самой последней, связан со стандартным вводом следующей программы этой последовательности (см. рис. 5.2). Конвейер обрабатывает некоторый поток сообщений. Каждое сообщение этого потока поступает на ввод первой программе, которая переработанное сообщение передает следующей программе, а сама начинает обработку очередного сообщения потока. Таким же образом действует каждая программа конвейера: получив сообщение от предшествующей программы и, обработав его, она передает переработанное сообщение следующей программе и приступает к обработке следующего сообщения. Последняя программа конвейера выводит результат работы всего конвейера (результирующее сообщение). Таким образом, в конвейере, состоящим из п программ, может одновременно находиться в обработке до п сообщений. Конечно, в силу того, что разные программы конвейера могут затратить на обработку очередных сообщений разные отрезки времени, необходимо обеспечить каким-либо образом синхронизацию этих процессов (некоторые процессы могут находиться в стадии ожидания либо возможности передать переработанное сообщение, либо возможности получить очередное сообщение).

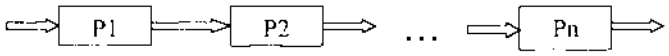


Рис. 5.2 Конвейер параллельно действующих программ.

В общем случае коллектив параллельно действующих программ может быть организован в систему с портами сообщений. *Порт сообщений* представляет собой программную подсистему, обслуживающую некоторую очередь сообщений: она может принимать на хранение от программы какое-либо сообщение, ставя его в очередь, и может выдавать очередное сообщение другой программе по ее требованию. Сообщение, переданное какой-либо программой некоторому порту, уже не будет принадлежать этой программе (и использовать ее ресурсы), но оно не будет принадлежать и никакой другой программе, пока в порядке очереди не будет передано какой-либо программе по ее запросу. Таким образом, программа, передающая сообщение не будет находиться в стадии ожидания пока программа, принимающая это сообщение, не будет готова его обрабатывать (если только не будет переполнен принимающий порт).

Пример программной системы с портами сообщений приведен на рис. 5.3. Порт U может рассматриваться как порт вводных сообщений для представленного на этом рисунке

коллектива параллельно действующих программ, а порт W - как порт выводных сообщений для этого коллектива программ.

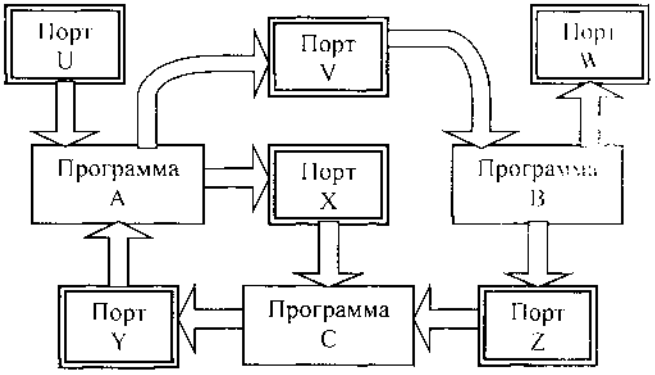


Рис. 5.3. Пример программной системы с портами сообщений.

Программные системы с портами сообщений могут быть как *жесткой* конфигурации, так и *гибкой* конфигурации. В системах с портами жесткой конфигурации каждая программа жестко связывается с одним или с несколькими входными портами. Для передачи сообщения такая программа должна явно указать адрес передачи: имя программы и имя ее входного порта. В этом случае при изменении конфигурации системы придется корректировать используемые программы: изменять адреса передач сообщений. В системах с портами гибкой конфигурации с каждой программой связаны как входные, так и выходные *виртуальные* порты. Перед запуском такой системы программы на основании информации, задаваемой пользователем, должна производиться ее предварительная настройка с помощью специальной программной компоненты, осуществляющей совмещение каждого выходного виртуального порта одной программы с каким-либо входным виртуальным портом другой. Тем самым при изменении конфигурации системы в этом случае не требуется какой-либо корректировки используемых программ - необходимые изменения должны быть отражены в информации для настройки. Однако в этом случае требуется иметь специальную программную компоненту, осуществляющую настройку системы.

5.3. Архитектурные функции.

Для обеспечения взаимодействия между подсистемами в ряде случаев не требуется создавать какие-либо дополнительные программные компоненты (помимо реализации внешних функций) - для этого может быть достаточно заранее фиксированных соглашений и стандартных возможностей базового программного обеспечения (операционной системы). Так в комплексе автономно выполняемых программ для обеспечения взаимодействия достаточно описания (спецификации) общей внешней информационной среды и возможностей операционной системы для запуска программ. В слоистой программной системе может оказаться достаточным спецификации выделенных программных слоев и обычный аппарат обращения к процедурам. В программном конвейере взаимодействие между программами также может обеспечивать операционная система (как это имеет место в операционной системе UNIX).

Однако в ряде случаев для обеспечения взаимодействия между программными подсистемами может потребоваться создание дополнительных программных компонент. Так для управления работой комплекса автономно выполняемых программ часто создают специализированный командный интерпретатор, более удобный (в данной предметной области) для подготовки требуемой внешней информационной среды и запуска требуемой программы, чем базовый командный интерпретатор используемой операционной системы. В слоистых программных системах может быть создан особый аппарат обращения к процедурам слоям (например, обеспечивающий параллельное выполнение этих процедур). В коллективе параллельно действующих программ для управления портами сообщений требуется специальная программная подсистема. Такие программные компоненты реализуют не внешние

функции ПС, а функции, возникшие в результате разработки архитектуры этого ПС. В связи с этим такие функции мы будем называть *архитектурными.*

5.4. Контроль архитектуры программных средств.

Для контроля архитектуры ПС используется смежный контроль и ручная имитация.

Смежный контроль архитектуры ПС сверху - это ее контроль разработчиками внешнего описания: разработчиками спецификации качества и разработчиками функциональной спецификации. Смежный контроль архитектуры ПС снизу - это ее контроль потенциальными разработчиками программных подсистем, входящих в состав ПС в соответствии с разработанной архитектурой.

Ручная имитация архитектуры ПС производится аналогично ручной имитации функциональной спецификации, только целью этого контроля является проверка взаимодействия между программными подсистемами. Так же как и в случае ручной имитации функциональной спецификации ПС должны быть сначала подготовлены тесты. Затем группа разработчиков должна для каждого такого теста имитировать работу каждой программной подсистемы, входящей в состав ПС. При этом работу каждой подсистемы имитирует один какой-либо разработчик (не автор архитектуры), тщательно выполняя все взаимодействия этой подсистемы с другими подсистемами (точнее, с разработчиками, их имитирующими) в соответствии с разработанной архитектурой ПС. Тем самым обеспечивается имитационное функционирование ПС в целом в рамках проверяемой архитектуры.

Упражнения к лекции 5.

5.1 Что такое архитектура программного средства?

5.2 Что такое архитектурная функция?

Лекция 7.

РАЗРАБОТКА СТРУКТУРЫ ПРОГРАММЫ И МОДУЛЬНОЕ ПРОГРАММИРОВАНИЕ

Цель разработки структуры программы. Понятие программного модуля. Основные .характеристики программного модуля. Методы разработки структуры программы. Спецификация программного модуля. Контроль структуры программы.

7.1. Цель модульного программирования.

Приступая к разработке каждой программы ПС, следует иметь в виду, что она, как правило, является большой системой, поэтому мы должны принять меры для ее упрощения. Для этого такую программу разрабатывают по частям, которые называются программными модулями [7.1, 7.2]. А сам такой метод разработки программ называют *модульным* программированием [7.3]. *Программный модуль* - это любой фрагмент описания процесса, оформляемый как самостоятельный программный продукт, пригодный для использования в описаниях процесса. Это означает, что каждый программный модуль программируется, компилируется и отлаживается отдельно от других модулей программы, и тем самым, физически разделен с другими модулями программы. Более того, каждый разработанный программный модуль может включаться в состав разных программ, если выполнены условия его использования, декларированные в документации по этому модулю. Таким образом, программный модуль может рассматриваться и как средство борьбы со сложностью программ, и как средство борьбы с дублированием в программировании (т.е. как средство накопления и многократного использования программистских знаний).

Модульное программирование является воплощением в процессе разработки программ обоих общих методов борьбы со сложностью (см. лекцию 3, п. 3.5): и обеспечение независимости компонент системы, и использование иерархических структур. Для воплощения первого метода формулируются определенные требования, которым должен удовлетворять программный модуль, т.е. выявляются основные характеристики «хорошего» программного модуля. Для воплощения второго метода используют древовидные модульные структуры программ (включая деревья со сросшимися ветвями).

7.2. Основные характеристики программного модуля.

Не всякий программный модуль способствует упрощению программы [7.2]. Выделить хороший с этой точки зрения модуль является серьезной творческой задачей. Для оценки приемлемости выделенного модуля используются некоторые критерии. Так, Хольт [7.4] предложил следующие два общих таких критерия:

- хороший модуль снаружи проще, чем внутри;
- хороший модуль проще использовать, чем построить.

Майерс [7.5] предлагает для оценки приемлемости программного модуля использовать более конструктивные его характеристики:

- размер модуля,
- прочность модуля,
- сцепление с другими модулями,
- рутинность модуля (независимость от предыстории обращений к нему).

Размер модуля измеряется числом содержащихся в нем операторов или строк. Модуль не должен быть слишком маленьким или слишком большим. Маленькие модули приводят к громоздкой модульной структуре программы и могут не окупать накладных расходов, связанных с их оформлением. Большие модули неудобны для изучения и изменений, они могут существенно увеличить суммарное время повторных трансляций программы при отладке программы. Обычно рекомендуются программные модули размером от нескольких десятков до нескольких сотен операторов.

Прочность модуля - это мера его внутренних связей. Чем выше прочность модуля, тем больше связей он может спрятать от внешней по отношению к нему части программы и, следовательно, тем больший вклад в упрощение программы он может внести. Для оценки степени прочности модуля Майерс [7.5] предлагает упорядоченный по степени прочности набор из семи классов модулей. Самой слабой степенью прочности обладает модуль, *прочный по совпадению.* Это такой модуль, между элементами которого нет осмысленных связей. Такой модуль может быть выделен, например, при обнаружении в разных местах программы повторения одной и той же последовательности операторов, которая и оформляется в отдельный модуль. Необходимость изменения этой последовательности в одном из контекстов может привести к изменению этого модуля, что может сделать его использование в других контекстах ошибочным. Такой класс программных модулей не рекомендуется для использования. Вообще говоря, предложенная Майерсом упорядоченность по степени прочности классов модулей не бесспорна. Однако, это не очень существенно, так как только два высших по прочности класса модулей рекомендуются для использования. Эти классы мы и рассмотрим подробнее.

Функционально прочный модуль - это модуль, выполняющий (реализующий) одну какую-либо определенную функцию. При реализации этой функции такой модуль может использовать и другие модули. Такой класс программных модулей рекомендуется для использования.

Информационно прочный модуль - это модуль, выполняющий (реализующий) несколько операций (функций) над одной и той же структурой данных (информационным объектом), которая считается неизвестной вне этого модуля. Для каждой из этих операций в таком модуле имеется свой вход со своей формой обращения к нему. Такой класс следует рассматривать как класс программных модулей с высшей степенью прочности. Информационно прочный модуль может реализовывать, например, абстрактный тип данных.

В модульных языках программирования как минимум имеются средства для задания функционально прочных модулей (например, модуль типа FUNCTION в языке ФОРТРАН). Средства же для задания информационно прочных модулей в ранних языках программирования отсутствовали. Эти средства появились только в более поздних языках. Так в языке программирования Ада средством задания информационно прочного модуля является пакет [7.6].

Сцепление модуля - это мера его зависимости по данным от других модулей. Характеризуется способом передачи данных. Чем слабее сцепление модуля с другими модулями, тем сильнее его независимость от других модулей. Для оценки степени сцепления Майерс предлагает [7.5] упорядоченный набор из шести видов сцепления модулей. Худшим видом сцепления модулей является *сцепление по содержимому.* Таким является сцепление двух модулей, когда один из них имеет прямые ссылки на содержимое другого модуля (например, на константу, содержащуюся в другом модуле). Такое сцепление модулей недопустимо. Не рекомендуется использовать также *сцепление по общей области* - это такое сцепление модулей, когда несколько модулей используют одну и ту же область памяти. Такой вид сцепления модулей реализуется, например, при программировании на языке ФОРТРАН с использованием блоков COMMON. Единственным видом сцепления модулей, который рекомендуется для использования современной технологией программирования, является *параметрическое сцепление* (сцепление по данным по Майерсу [7.5]) - это случай, когда данные передаются модулю либо при обращении к нему как значения его параметров,
либо как результат его обращения к другому модулю для вычисления некоторой функции. Такой вид сцепления модулей реализуется на языках программирования при использовании обращений к процедурам (функциям).

Рутинность модуля - это его независимость от предыстории обращений к нему. Модуль будем называть *рутинным,* если результат (эффект) обращения к нему зависит только от значений его параметров (и не зависит от предыстории обращений к нему). Модуль будем называть *зависящим от предыстории,* если результат (эффект) обращения к нему зависит от внутреннего состояния этого модуля, изменяемого в результате предыдущих обращений к нему. Майерс [7.5] не рекомендует использовать зависящие от предыстории (непредсказуемые) модули, так как они провоцируют появление в программах хитрых (неуловимых) ошибок. Однако такая рекомендация является неконструктивной, так как во многих случаях именно зависящий от предыстории модуль является лучшей реализаций информационно прочного модуля. Поэтому более приемлема следующая (более осторожная) рекомендация:

- всегда следует использовать рутинный модуль, если это не приводит к плохим (не рекомендуемым) сцеплениям модулей;
- зависящие от предыстории модули следует использовать только в случае, когда это необходимо для обеспечения параметрического сцепления;
- в спецификации зависящего от предыстории модуля должна быть четко сформулирована эта зависимость таким образом, чтобы было возможно прогнозировать поведение (эффект выполнения) данного модуля при разных последующих обращениях к нему.

В связи с последней рекомендацией может быть полезным определение внешнего представления (ориентированного на информирование человека) состояний зависящего от предыстории модуля. В этом случае эффект выполнения каждой функции (операции), реализуемой этим модулем, следует описывать в терминах этого внешнего представления, что существенно упростит прогнозирование поведения данного модуля.

7.3. Методы разработки структуры программы.

Как уже отмечалось выше, в качестве модульной структуры программы принято использовать древовидную структуру, включая деревья со сросшимися ветвями. В узлах такого дерева размещаются программные модули, а направленные дуги (стрелки) показывают статическую подчиненность модулей, т.е. каждая дуга показывает, что в тексте модуля, из которого она исходит, имеется ссылка на модуль, в который она входит. Другими словами, каждый модуль может обращаться к подчиненным ему модулям, т.е. выражается через эти модули. При этом модульная структура программы, в конечном счете, должна включать и совокупность спецификаций модулей, образующих эту программу. *Спецификация* программного модуля содержит

• синтаксическую спецификацию его входов, позволяющую построить на используемом языке программирования синтаксически правильное обращение к нему (к любому его входу),

• функциональную спецификацию модуля (описание семантики функций, выполняемых этим модулем по каждому из его входов)

Функциональная спецификация модуля строится так же, как и функциональная спецификация ПС.

В процессе разработки программы ее модульная структура может по-разному формироваться и использоваться для определения порядка программирования и отладки модулей, указанных в этой структуре. Поэтому можно говорить о разных методах разработки структуры программы. Обычно в литературе обсуждаются два метода [7.1, 7.7]: метод восходящей разработки и метод нисходящей разработки.

Метод *восходящей разработки* заключается в следующем. Сначала строится модульная структура программы в виде дерева. Затем поочередно программируются модули программы, начиная с модулей самого нижнего уровня (листья дерева модульной структуры программы), в таком порядке, чтобы для каждого программируемого модуля были уже запрограммированы все модули, к которым он может обращаться. После того, как все модули программы запрограммированы, производится их поочередное тестирование и отладка в принципе в таком же (восходящем) порядке, в каком велось их программирование. Такой порядок разработки программы на первый взгляд кажется вполне естественным: каждый модуль при программировании выражается через уже запрограммированные непосредственно подчиненные модули, а при тестировании использует уже отлаженные модули. Однако, современная технология не рекомендует такой порядок разработки программы. Во-первых, для программирования какого-либо модуля совсем не требуется наличия текстов используемых им модулей - для этого достаточно, чтобы каждый используемый модуль был лишь специфицирован (в объеме, позволяющем построить правильное обращение к нему), а для тестирования его возможно (и даже, как мы покажем ниже, полезно) используемые модули заменять их имитаторами (заглушками). Во-вторых, каждая программа в какой-то степени подчиняется некоторым внутренним для нее, но глобальным для ее модулей соображениям (принципам реализации, предположениям, структурам данных и т.п.), что определяет ее концептуальную целостность и формируется в процессе ее разработки. При восходящей разработке эта глобальная информация для модулей нижних уровней еще не ясна в полном объеме, поэтому очень часто приходится их перепрограммировать, когда при программировании других модулей производится существенное уточнение этой глобальной информации (например, изменяется глобальная структура данных). В-третьих, при восходящем тестировании для каждого модуля (кроме головного) приходится создавать ведущую программу (модуль), которая должна подготовить для тестируемого модуля необходимое состояние информационной среды и произвести требуемое обращение к нему. Это приводит к большому объему «отладочного» программирования и в то же время не дает никакой гарантии, что тестирование модулей производилось именно в тех условиях, в которых они будут выполняться в рабочей программе.

Метод *нисходящей разработки* заключается в следующем. Как и в предыдущем методе сначала строится модульная структура программы в виде дерева. Затем поочередно программируются модули программы, начиная с модуля самого верхнего уровня (головного), переходя к программированию какого-либо другого модуля только в том случае, если уже запрограммирован модуль, который к нему обращается. После того, как все модули программы запрограммированы, производится их поочередное тестирование и отладка в таком же (нисходящем) порядке. При этом первым тестируется головной модуль программы, который представляет всю тестируемую программу и поэтому тестируется при «естественном» состоянии информационной среды, при котором начинает выполняться эта программа. При этом те модули, к которым может обращаться головной, заменяются их имитаторами (так называемыми заглушками [7.5]). Каждый *имитатор модуля* представляется

весьма простым программным фрагментом, который, в основном, сигнализирует о самом факте обращения к имитируемому модулю, производит необходимую для правильной работы программы обработку значений его входных параметров (иногда с их распечаткой) и выдает, если это необходимо, заранее запасенный подходящий результат. После завершения тестирования и отладки головного и любого последующего модуля производится переход к тестированию одного из модулей, которые в данный момент представлены имитаторами, если таковые имеются. Для этого имитатор выбранного для тестирования модуля заменяется самим этим модулем и, кроме того, добавляются имитаторы тех модулей, к которым может обращаться выбранный для тестирования модуль. При этом каждый такой модуль будет тестироваться при «естественных» состояниях информационной среды, возникающих к моменту обращения к этому модулю при выполнении тестируемой программы. Таким образом, большой объем «отладочного» программирования при восходящем тестировании заменяется программированием достаточно простых имитаторов используемых в программе модулей. Кроме того, имитаторы удобно использовать для того, чтобы подыгрывать процессу подбора тестов путем задания нужных результатов, выдаваемых имитаторами. При таком порядке разработки программы вся необходимая глобальная информация формируется своевременно, т.е. ликвидируется весьма неприятный источник просчетов при программировании модулей. Некоторым недостатком нисходящей разработки, приводящим к определенным затруднениям при ее применении, является необходимость абстрагироваться от базовых возможностей используемого языка программирования, выдумывая абстрактные операции, которые позже нужно будет реализовать с помощью выделенных в программе модулей. Однако способность к таким абстракциям представляется необходимым условием разработки больших программных средств, поэтому ее нужно развивать.

Особенностью рассмотренных методов восходящей и нисходящей разработок (которые мы будем называть *классическими*) является требование, чтобы модульная структура программы была разработана до начала программирования (кодирования) модулей. Это требование находится в полном соответствии с водопадным подходом к разработке ПС, так как разработка модульной структуры программы и ее кодирование производятся на разных этапах разработки ПС: первая завершает этап конструирования ПС, а второе -открывает этап кодирования. Однако эти методы вызывают ряд возражений: представляется сомнительным, чтобы до программирования модулей можно было разработать структуру программы достаточно точно и содержательно. На самом деле это делать не обязательно, если несколько модернизировать водопадный подход. Ниже предлагаются конструктивный и архитектурный подходы к разработке программ [7.3], в которых модульная структура формируется в процессе программирования (кодирования) модулей.

Конструктивный подход к разработке программы представляет собой модификацию нисходящей разработки, при которой модульная древовидная структура программы формируется в процессе программирования модулей. Разработка программы при конструктивном подходе начинается с программирования головного модуля, исходя из спецификации программы в целом. При этом спецификация программы принимается в качестве спецификации ее головного модуля, который полностью берет на себя ответственность за выполнение функций программы. В процессе программирования головного модуля, в случае, если эта программа достаточно большая, выделяются подзадачи (внутренние функции), в терминах которых программируется головной модуль. Это означает, что для каждой выделяемой подзадачи (функции) создается спецификация реализующего ее фрагмента программы, который в дальнейшем может быть представлен некоторым поддеревом модулей. Важно заметить, что здесь также ответственность за выполнение выделенной функции несет головной (может быть, и единственный) модуль этого поддерева, так что спецификация выделенной функции является одновременно и спецификацией головного модуля этого поддерева. В головном модуле программы для обращения к выделенной функции строится обращение к головному модулю указанного поддерева в соответствии с созданной

его спецификацией. Таким образом, на первом шаге разработки программы (при программировании ее головного модуля) формируется верхняя начальная часть дерева, например, такая, которая показана на рис. 7.1.

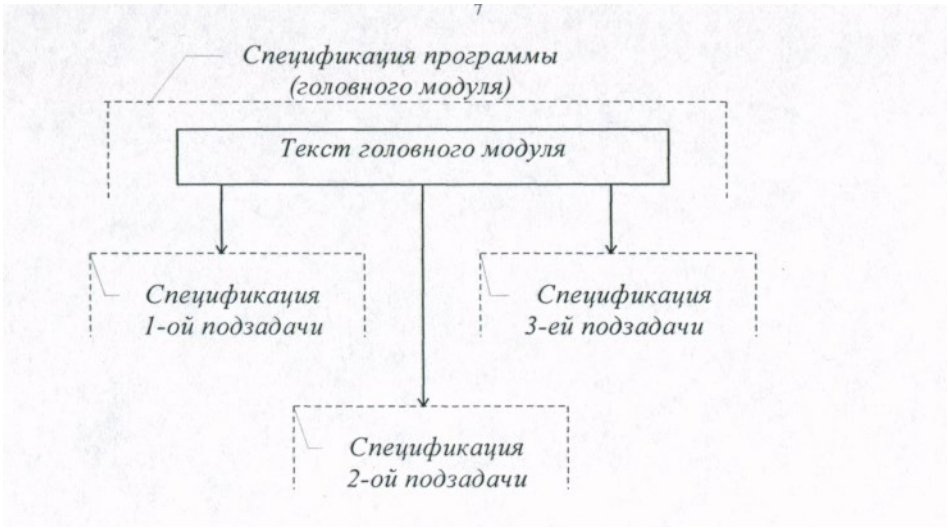
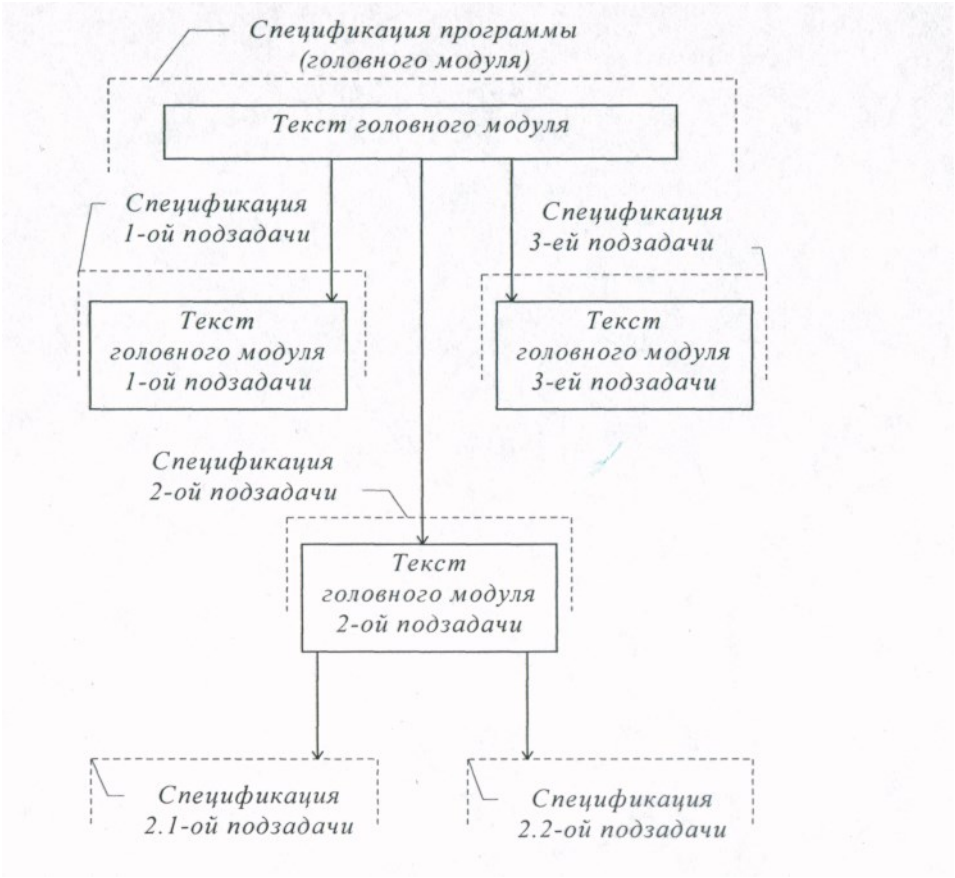


Рис. 7.1. Первый шаг формирования модульной структуры программы при конструктивном подходе

Аналогичные действия производятся при программировании любого другого модуля, который выбирается из текущего состояния дерева программы из числа специфицированных, но пока еще не запрограммированных модулей. В результате этого производится очередное доформирование дерева программы, например, такое, которое показано на рис. 7.2.

Архитектурный подход к разработке программы представляет собой модификацию восходящей разработки, при которой модульная структура программы формируется в процессе программирования модуля. Но при этом ставится существенно другая цель разработки: повышение уровня используемого языка программирования, а не разработка конкретной программы. Это означает, что для заданной предметной области выделяются типичные функции, каждая из которых может использоваться при решении разных задач в этой области, и специфицируются, а затем и программируются отдельные программные модули, выполняющие эти функции. Так как процесс выделения таких функций связан с накоплением и обобщением опыта решения задач в заданной предметной области, то обычно сначала выделяются и реализуются отдельными модулями более простые функции, а затем постепенно появляются модули, использующие ранее выделенные функции. Такой набор модулей создается в расчете на то, что при разработке той или иной программы заданной предметной области в рамках конструктивного подхода могут оказаться приемлемыми некоторые из этих модулей. Это позволяет существенно сократить трудозатраты на разработку конкретной программы путем подключения к ней заранее заготовленных и проверенных на практике модульных структур нижнего уровня. Так как такие структуры могут многократно использоваться в разных конкретных программах, то архитектурный подход может рассматриваться как путь борьбы с дублированием в программировании. В связи с этим программные модули, создаваемые в рамках архитектурного подхода, обычно параметризуются для того, чтобы усилить применимость таких модулей путем настройки их на параметры.

Рис. 7.2. Второй шаг формирования модульной структуры программы



при конструктивном подходе.

В классическом методе нисходящей разработки рекомендуется сначала все модули разрабатываемой программы запрограммировать, а уж затем начинать нисходящее их тестирование [7.5], что опять-таки находится в полном соответствии с водопадным подходом. Однако такой порядок разработки не представляется достаточно обоснованным: тестирование и отладка модулей может привести к изменению спецификации подчиненных модулей и даже к изменению самой модульной структуры программы, так что в этом случае программирование некоторых модулей может оказаться бесполезно проделанной работой. Нам представляется более рациональным другой порядок разработки программы, известный в литературе как метод *нисходящей реализации,* что представляет некоторую модификацию водопадного подхода. В этом методе каждый запрограммированный модуль начинают сразу же тестировать до перехода к программированию другого модуля.

Все эти методы имеют еще различные разновидности в зависимости от того, в какой последовательности обходятся узлы (модули) древовидной структуры программы в процессе ее разработки [7.1]. Это можно делать, например, по слоям (разрабатывая все модули одного уровня, прежде чем переходить к следующему уровню)/При нисходящей разработке дерево можно обходить также в лексикографическом порядке (сверху вниз, слева направо). Возможны и другие варианты обхода дерева. Так, при конструктивной реализации для обхода дерева программы целесообразно следовать идеям Фуксмана, которые он использовал в предложенном им методе вертикального слоения [7.8]. Сущность такого обхода заключается в следующем. В рамках конструктивного подхода сначала реализуются только те модули, которые необходимы для самого простейшего варианта программы, которая может нормально выполняться только для весьма ограниченного множества наборов входных данных, но для таких данных эта задача будет решаться до конца. Вместо других модулей, на которые в такой программе имеются ссылки, в эту программу вставляются лишь их имитаторы, обеспечивающие, в основном, сигнализацию о выходе за пределы этого частного случая. Затем к этой программе добавляются реализации некоторых других модулей (в

частности, вместо некоторых из имеющихся имитаторов), обеспечивающих нормальное выполнение для некоторых других наборов входных данных. И этот процесс продолжается поэтапно до полной реализации требуемой программы. Таким образом, обход дерева программы производится с целью кратчайшим путем реализовать тот или иной вариант (сначала самый простейший) нормально действующей программы. В связи с этим такая разновидность конструктивной реализации получила название *целенаправленной конструктивной реализации.* Достоинством этого метода является то, что уже на достаточно ранней стадии создается работающий вариант разрабатываемой программы. Психологически это играет роль допинга, резко повышающего эффективность разработчика. Поэтому этот метод является весьма привлекательным.

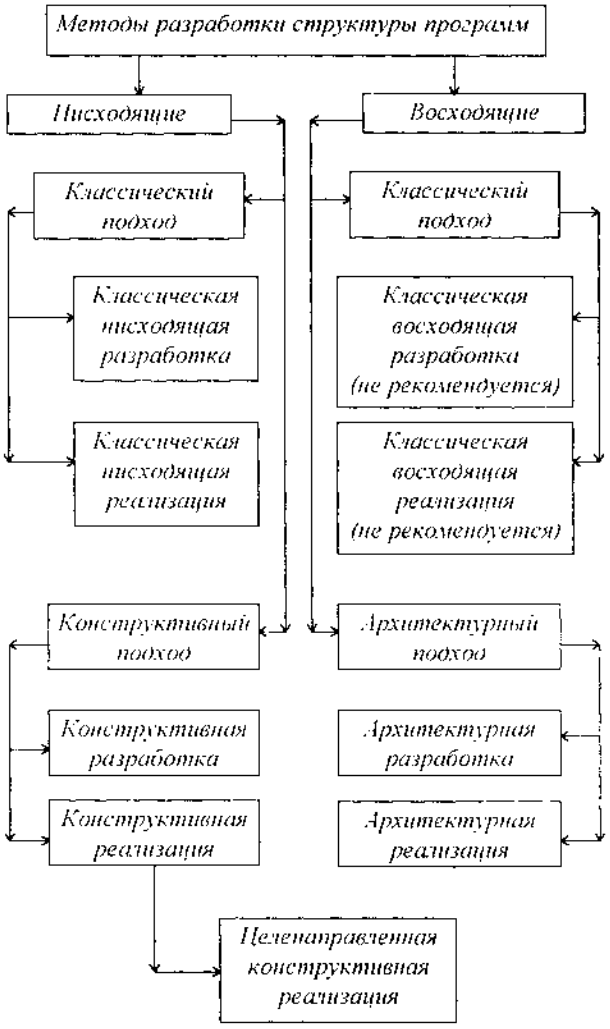


Рис. 7.3. Классификация методов разработки структуры программ.

На рис. 7.3 представлена общая классификация рассмотренных методов разработки структуры программы.

7.4. Контроль структуры программы.

Для контроля структуры программы можно использовать три метода [7.5]:

- статический контроль,
- смежный контроль,
- сквозной контроль.

Статический контроль состоит в оценке структуры программы, насколько хорошо программа разбита на модули с учетом значений рассмотренных выше основных характеристик модуля.

Смежный контроль сверху - это контроль со стороны разработчиков архитектуры и внешнего описания ПС. Смежный контроль снизу - это контроль спецификации модулей со стороны разработчиков этих модулей.

Сквозной контроль - это мысленное прокручивание (проверка) структуры программы при выполнении заранее разработанных тестов. Является видом динамического контроля так же, как и ручная имитация функциональной спецификации или архитектуры ПС.

Следует заметить, что указанный контроль структуры программы производится в рамках водопадного подхода разработки ПС, т.е. при классическом подходе. При конструктивном и архитектурном подходах контроль структуры программы осуществляется в процессе программирования (кодирования) модулей в подходящие моменты времени.

Упражнения к лекции 7.

7.1. *Что такое программный модуль?*

7.2. *Что такое прочность программного модуля?*

7.3. *Что такое сцепление программного модуля?*

Лекция 8.

РАЗРАБОТКА ПРОГРАММНОГО МОДУЛЯ

Порядок разработки программного модуля. Структурное программирование и пошаговая детализация. Понятие о псевдокоде. Контроль программного модуля.

8.1. Порядок разработки программного модуля.

При разработке программного модуля целесообразно придерживаться следующего порядка [8.1]:

- изучение и проверка спецификации модуля, выбор языка программирования;
- выбор алгоритма и структуры данных;
- программирование (кодирование) модуля;
- шлифовка текста модуля;
- проверка модуля;
- компиляция модуля.

Первый шаг разработки программного модуля в значительной степени представляет собой смежный контроль структуры программы снизу: изучая спецификацию модуля, разработчик должен убедиться, что она ему понятна и достаточна для разработки этого модуля. В завершении этого шага выбирается язык программирования: хотя язык программирования может быть уже предопределен для всего ПС, все же в ряде случаев (если система программирования это допускает) может быть выбран другой язык, более подходящий для реализации данного модуля (например, язык ассемблера).

На втором шаге разработки программного модуля необходимо выяснить, не известны ли уже какие-либо алгоритмы для решения поставленной и или близкой к ней задачи. И если найдется подходящий алгоритм, то целесообразно им воспользоваться. Выбор подходящих структур данных, которые будут использоваться при выполнении модулем своих функций, в значительной степени предопределяет логику и качественные показатели разрабатываемого модуля, поэтому его следует рассматривать как весьма ответственное решение.

На третьем шаге осуществляется построение текста модуля на выбранном языке программирования. Обилие всевозможных деталей, которые должны быть учтены при реализации функций, указанных в спецификации модуля, легко могут привести к созданию весьма запутанного текста, содержащего массу ошибок и неточностей. Искать ошибки в таком модуле и вносить в него требуемые изменения может оказаться весьма трудоемкой задачей. Поэтому весьма важно для построения текста модуля пользоваться технологически обоснованной и практически проверенной дисциплиной программирования. Впервые на это обратил внимание Дейкстра [8.2], сформулировав и обосновав основные принципы структурного программирования. На этих принципах базируются многие дисциплины программирования, широко применяемые на практике [8.3-8.6]. Наиболее распространенной является дисциплина пошаговой детализации [8.3], которая подробно обсуждается в разделах 8.2 и 8.3.

Следующий шаг разработки модуля связан с приведением текста модуля к завершенному виду в соответствии со спецификацией качества ПС. При программировании модуля разработчик основное внимание уделяет правильности реализации функций модуля, оставляя недоработанными комментарии и допуская некоторые нарушения требований к стилю программы. При шлифовке текста модуля он должен отредактировать имеющиеся в тексте комментарии и, возможно, включить в него дополнительные комментарии с целью обеспечить требуемые примитивы качества [8.1]. С этой же целью производится редактирование текста программы для выполнения стилистических требований.

Шаг проверки модуля представляет собой ручную проверку внутренней логики модуля до начала его отладки (использующей выполнение его на компьютере), реализует общий принцип, сформулированный для обсуждаемой технологии программирования, о необходимости контроля принимаемых решений на каждом этапе разработки ПС (см. лекцию 3). Методы проверки модуля обсуждаются в разделе 8.4.

И, наконец, последний шаг разработки модуля означает завершение проверки модуля (с помощью компилятора) и переход к процессу отладки модуля.

8.2. Структурное программирование.

При программировании модуля следует иметь в виду, что программа должна быть понятной не только компьютеру, но и человеку: и разработчик модуля, и лица, проверяющие модуль, и тестовики, готовящие тесты для отладки модуля, и сопроводители ПС, осуществляющие требуемые изменения модуля, вынуждены будут многократно разбирать логику работы модуля. В современных языках программирования достаточно средств, чтобы запутать эту логику сколь угодно сильно, тем самым, сделать модуль трудно понимаемым для человека и, как следствие этого, сделать его ненадежным или трудно сопровождаемым. Поэтому необходимо принимать меры для выбора подходящих языковых средств и следовать определенной дисциплине программирования. В связи с этим Дейкстра [8.2] и предложил строить программу как композицию из нескольких типов управляющих конструкций (структур), которые позволяют сильно повысить понимаемость логики работы программы. Программирование с использованием только таких конструкций назвали *структурным.*

Основными конструкциями структурного программирования являются: следование, разветвление и повторение (см. Рис. 8.1). Компонентами этих конструкций являются обобщенные операторы (узлы обработки [8.5]) S, S1, S2 и условие (предикат) P. В качестве обобщенного оператора может быть либо простой оператор используемого языка программирования (операторы присваивания, ввода, вывода, обращения к процедуре), либо фрагмент программы, являющийся композицией основных управляющих конструкций структурного программирования. Существенно, что каждая из этих конструкций имеет по управлению только один вход и один выход. Тем самым, и обобщенный оператор имеет только один вход и один выход.

Весьма важно также, что эти конструкции являются уже математическими объектами (что, по существу, и объясняет причину успеха структурного программирования). Доказано, что для каждой неструктурированной программы можно построить функционально эквивалентную (т.е. решающую ту же задачу) структурированную программу.

Для структурированных программ можно математически доказывать некоторые свойства,

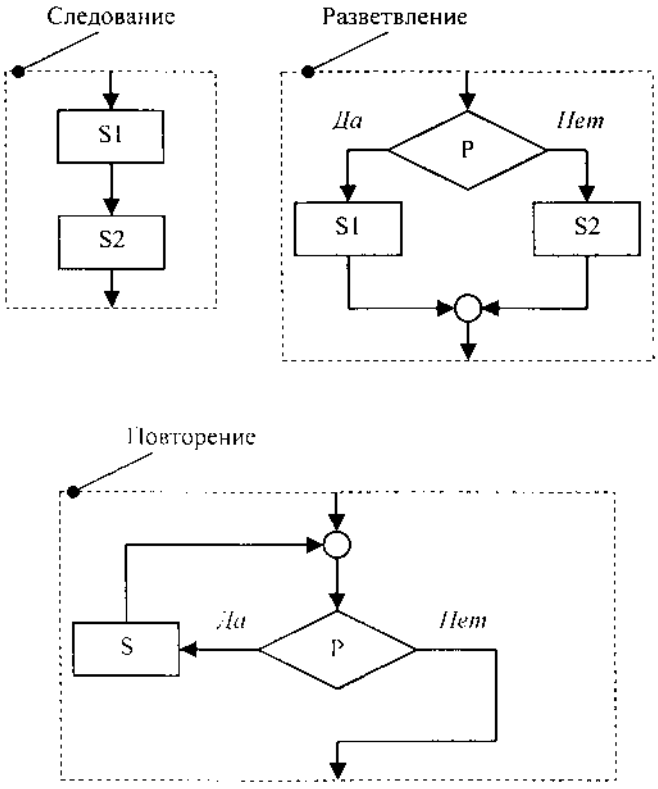


Рис. 8.1. Основные управляющие конструкции структурного программирования.

что позволяет обнаруживать в программе некоторые ошибки. Этому вопросу будет посвящена отдельная лекция.

Структурное программирование иногда называют еще "программированием без GO TO". Однако дело здесь не в операторе GO TO, а в его беспорядочном использовании. Очень часто при воплощении структурного программирования на некоторых языках программирования (например, на ФОРТРАНе) оператор перехода (GO TO) используется для реализации структурных конструкций, что не нарушает принципов структурного программирования. Запутывают программу как раз "неструктурные" операторы перехода, особенно переход к оператору, расположенному в тексте модуля выше (раньше) выполняемого оператора перехода. Тем не менее, попытка избежать оператора перехода в некоторых простых случаях может привести к слишком громоздким структурированным программам, что не улучшает их ясность и содержит опасность появления в тексте модуля дополнительных ошибок. Поэтому можно рекомендовать избегать употребления оператора перехода всюду, где это возможно, но не ценой ясности программы [8.1].

К полезным случаям использования оператора перехода можно отнести выход из цикла или процедуры по особому условию, "досрочно" прекращающего работу данного цикла или данной процедуры, т.е. завершающего работу некоторой структурной единицы (обобщенного оператора) и тем самым лишь локально нарушающего структурированность программы. Большие трудности (и усложнение структуры) вызывает структурная реализация реакции на возникающие исключительные (часто ошибочные) ситуации, так как при этом требуется не только осуществить досрочный выход из структурной единицы, но и произвести необходимую обработку (исключение) этой ситуации (например, выдачу подходящей диагностической информации). Обработчик исключительной ситуации может находиться на любом уровне структуры программы, а обращение к нему может производиться с разных нижних уровней. Вполне приемлемой с технологической точки зрения является следующая "неструктурная" реализация реакции на исключительные ситуации [8.7]. Обработчики исключительных ситуаций помещаются в конце той или иной структурной единицы и каждый такой обработчик программируется таким образом, что после окончания своей работы производит выход из той структурной единицы, в конце которой он помещен. Обращение к такому обработчику производится оператором перехода из данной структурной единицы (включая любую вложенную в нее структурную единицу).

8.3. Пошаговая детализация и понятие о псевдокоде.

Структурное программирование дает рекомендации о том, каким должен быть текст модуля. Возникает вопрос, как должен действовать программист, чтобы построить такой текст. Часто программирование модуля начинают с построения его блок-схемы, описывающей в общих чертах логику его работы. Однако современная технология программирования не рекомендует этого делать без подходящей компьютерной поддержки. Хотя блок-схемы позволяют весьма наглядно представить логику работы модуля, при их ручном кодировании на языке программирования возникает весьма специфический источник ошибок: отображение существенно двумерных структур, какими являются блок-схемы, на линейный текст, представляющий модуль, содержит опасность искажения логики работы модуля, тем более, что психологически довольно трудно сохранить высокий уровень внимания при повторном ее рассмотрении. Исключением может быть случай, когда для построения блок-схем используется графический редактор и они формализованы настолько, что по ним автоматически генерируется текст на языке программирования (как, например, это делается в Р-технологии [8.6]).

В качестве основного метода построения текста модуля современная технология программирования рекомендует *пошаговую детализацию* [8.1, 8.3, 8.5]. Сущность этого метода заключается в разбиении процесса разработки текста модуля на ряд шагов. На первом шаге описывается общая схема работы модуля в обозримой линейной текстовой форме (т.е. с использованием очень крупных понятий), причем это описание не является полностью формализованным и ориентировано на восприятие его человеком. На каждом следую-

щем шаге производится уточнение и детализация одного из понятий (будем называть его *уточняемым*), в каком либо описании, разработанном на одном из предыдущих шагов. В результате такого шага создается описание выбранного уточняемого понятия либо в терминах базового языка программирования (т.е. выбранного для представления модуля), либо в такой же форме, что и на первом шаге с использованием новых уточняемых понятий. Этот процесс завершается, когда все уточняемые понятия будут *уточнения* (т.е. в конечном счете будут выражены на базовом языке программирования). Последним шагом является получение текста модуля на базовом языке программирования путем замены всех вхождений уточняемых понятий заданными их описаниями и выражение всех вхождений конструкций структурного программирования средствами этого языка программирования.

Пошаговая детализация связана с использованием частично формализованного языка для представления указанных описаний, который получил название *псевдокода* [8.5, 8.8]. Этот язык позволяет использовать все конструкции структурного программирования, которые оформляются формализованно, вместе с неформальными фрагментами на естественном языке для представления обобщенных операторов и условий. В качестве обобщенных операторов и условий могут задаваться и соответствующие фрагменты на базовом языке программирования.

Головным описанием на псевдокоде можно считать внешнее оформление модуля на базовом языке программирования, которое должно содержать:

- начало модуля на базовом языке, т.е. первое предложение или заголовок (спецификацию) этого модуля [8.1];
- раздел (совокупность) описаний на базовом языке, причем вместо описаний процедур и функций - только их внешнее оформление;
- неформальное обозначение последовательности операторов тела модуля как одного обобщенного оператора (см. ниже), а также неформальное обозначение тела каждого описания процедуры или функции как одного обобщенного оператора;
- последнее предложение (конец) модуля на базовом языке [8.1].

Внешнее оформление описания процедуры или функции представляется аналогично. Впрочем, если следовать Дейкстре [8.2], раздел описаний лучше также представить здесь неформальным обозначением, произведя его детализацию в виде отдельного описания.

Неформальное обозначение обобщенного оператора на псевдокоде производится на естественном языке произвольным предложением, раскрывающим в общих чертах его содержание. Единственным формальным требованием к оформлению такого обозначения является следующее: это предложение должно занимать целиком одно или несколько графических (печатных) строк и завершаться точкой (или каким-либо другим знаком, специально выделенным для этого).

```
Следование:
    обобщенный_оператор
    обобщенный_оператор

Разветвление:
    ЕСЛИ условие ТО
      обобщенный_оператор
   ИНАЧЕ
      обобщенный_оператор
ВСЕ ЕСЛИ

Повторение:
    ПОКА условие ДЕЛАТЬ
```

```
    обобщенный_оператор
  ВСЕ ПОКА
```

Рис. 8.2. Основные конструкции структурного программирования на псевдокоде.

Для каждого неформального обобщенного оператора должно быть создано отдельное описание, выражающее логику его работы (детализирующее его содержание) с помощью композиции основных конструкций структурного программирования и других обобщенных операторов. В качестве заголовка такого описания должно быть неформальное обозначение детализируемого обобщенного оператора. Основные конструкции структурного программирования могут быть представлены в следующем виде (см. рис. 8.2). Здесь условие может быть либо явно задано на базовом языке программирования в качестве булевского выражения, либо неформально представлено на естественном языке некоторым фрагментом, раскрывающим в общих чертах смысл этого условия. В последнем случае должно быть создано отдельное описание, детализирующее это условие, с указанием в качестве заголовка обозначения этого условия (фрагмента на естественном языке).

```
Выход из повторения (цикла):
    ВЫЙТИ
Выход из процедуры (функции):
    ВЕРНУТЬСЯ
Переход на обработку исключительной ситуации:
    ВОЗБУДИТЬ имя  исключения
```

Рис. 8.3. Частные случаи оператора перехода в качестве обобщенного оператора.

В качестве обобщенного оператора на псевдокоде можно использовать указанные выше частные случаи оператора перехода (см. рис. 8.3). Последовательность обработчиков исключительных ситуаций (исключений) задается в конце модуля или описания процедуры (функции). Каждый такой обработчик имеет вид:

```
ИСКЛЮЧЕНИЕ имя_исключения
  обобщенный_оператор
ВСЕ ИСКЛЮЧЕНИЕ
```

Отличие обработчика исключительной ситуации от процедуры без параметров заключается в следующем: после выполнения процедуры управление возвращается к оператору, следующему за обращением к ней, а после выполнения исключения управление возвращается к оператору, следующему за обращением к модулю или процедуре (функции), в конце которого (которой) помещено данное исключение.

Рекомендуется на каждом шаге детализации создавать достаточно содержательное описание, но легко обозримое (наглядное), так чтобы оно размещалось на одной странице текста. Как правило, это означает, что такое описание должно быть композицией пяти-шести конструкций структурного программирования. Рекомендуется также вложенные конструкции располагать со смещением вправо на несколько позиций (см. рис. 8.4). В результате можно получить описание логики работы по наглядности вполне конкурентное с блок-схемами, но обладающее существенным преимуществом - сохраняется линейность описания.

```
УДАЛЕНИЕ В ФАЙЛЕ ЗАПИСЕЙ ДО ПЕРВОЙ, УДОВЛЕТВОРЯЮ-
ЩЕЙ ЗАДАННОМУ ФИЛЬТРУ:

УСТАНОВИТЬ НАЧАЛО ФАЙЛА.
ПОКА  НЕ КОНЕЦ ФАЙЛА ДЕЛАТЬ
    ПРОЧИТАТЬ ОЧЕРЕДНУЮ ЗАПИСЬ.
    ЕСЛИ  ОЧЕРЕДНАЯ ЗАПИСЬ УДОВЛЕТВОРЯЕТ
          ФИЛЬТРУ ТО
```

```
    ВЫЙТИ
  ИНАЧЕ
    УДАЛИТЬ ОЧЕРЕДНУЮ ЗАПИСЬ ИЗ ФАЙЛА.
  ВСЕ ЕСЛИ
 ВСЕ ПОКА
 ЕСЛИ ЗАПИСИ НЕ УДАЛЕНЫ ТО
   НАПЕЧАТАТЬ "ЗАПИСИ НЕ УДАЛЕНЫ".
 ИНАЧЕ
   НАПЕЧАТАТЬ "УДАЛЕНО н ЗАПИСЕЙ".
 ВСЕ ЕСЛИ
```

Рис. 8.4. Пример одного шага детализации на псевдокоде.

Идею пошаговой детализации приписывают иногда Дейкстре [8.1]. Однако Дейкстра предлагал принципиально отличающийся метод построения текста модуля [8.2], который нам представляется более глубоким и перспективным. Во-первых, вместе с уточнением операторов он предлагал постепенно (по шагам) уточнять (детализировать) и используемые структуры данных. Во-вторых, на каждом шаге он предлагал создавать некоторую виртуальную машину для детализации и в ее терминах производить детализацию всех уточняемых понятий, для которых эта машина позволяет это сделать. Таким образом, Дейкстра предлагал, по существу, детализировать по горизонтальным слоям, что является перенесением его идеи о слоистых системах (см. лекцию 6) на уровень разработки модуля. Такой метод разработки модуля поддерживается в настоящее время пакетами языка АДА [8.7] и средствами объектно-ориентированного программирования [8.9].

8.4. Контроль программного модуля.

Применяются следующие методы контроля программного модуля:

- статическая проверка текста модуля;
- сквозное прослеживание;
- доказательство свойств программного модуля.

При статической проверке текста модуля этот текст просматривается с начала до конца с целью найти ошибки в модуле. Обычно для такой проверки привлекают, кроме разработчика модуля, еще одного или даже нескольких программистов. Рекомендуется ошибки, обнаруживаемые при такой проверке исправлять не сразу, а по завершению чтения текста модуля.

Сквозное прослеживание представляет собой один из видов динамического контроля модуля. В нем также участвуют несколько программистов, которые вручную прокручивают выполнение модуля (оператор за оператором в той последовательности, какая вытекает из логики работы модуля) на некотором наборе тестов.

Доказательству свойств программ посвящена следующая лекция. Здесь следует лишь отметить, что этот метод применяется пока очень редко.

Упражнения к лекции 8.

8.1. Что такое структурное программирование?

8.2. Что такое пошаговая детализация программного модуля'?

8.3. Что такое псевдокод?

Лекция № 10.

ТЕСТИРОВАНИЕ И ОТЛАДКА ПРОГРАММНОГО СРЕДСТВА

Основные понятия. Стратегия проектирования тестов. Заповеди отладки. Автономная отладка и тестирование программного модуля. Комплексная отладка и тестирование программного средства.

10.1. Основные понятия.

Отладка ПС это деятельность, направленная на обнаружение и исправление ошибок в ПС с использованием процессов выполнения его программ. *Тестирование* ПС — это процесс выполнения его программ на некотором наборе данных, для которого заранее известен результат применения или известны правила поведения этих программ. Ука-

занный набор данных называется *тестовым* или просто *тестом.* Таким образом, отладку можно представить в виде многократного повторения трех процессов: тестирования, в результате которого может быть констатировано наличие в ПС ошибки, поиска места ошибки в программах и документации ПС и редактирования программ и документации с целью устранения обнаруженной ошибки. Другими словами: Отладка = Тестирование + Поиск ошибок + Редактирование.

В зарубежной литературе отладку часто понимают [10.1-10.3] только как процесс поиска и исправления ошибок (без тестирования), факт наличия которых устанавливается при тестировании. Иногда тестирование и отладку считают синонимами [10.4,10.5]. В нашей стране в понятие отладки обычно включают и тестирование [10.6 -10.8], поэтому мы будем следовать сложившейся традиции. Впрочем, совместное рассмотрение в данной лекции этих процессов делает указанное разночтение не столь существенным. Следует, однако, отметить, что тестирование используется и как часть процесса аттестации ПС (см. лекцию 14).

10.2. Принципы и виды отладки программного средства.

Успех отладки ПС в значительной степени предопределяет рациональная организация тестирования. При отладке ПС отыскиваются и устраняются, в основном, те ошибки, наличие которых в ПС устанавливается при тестировании. Как было уже отмечено, тестирование не может доказать правильность ПС [10.9], в лучшем случае оно может продемонстрировать наличие в нем ошибки. Другими словами, нельзя гарантировать, что тестированием ПС практически выполнимым набором тестов можно установить наличие каждой имеющейся в ПС ошибки. Поэтому возникает две задачи. Первая задача: подготовить такой набор тестов и применить к ним ПС, чтобы обнаружить в нем по возможности большее число ошибок. Однако чем дольше продолжается процесс тестирования (и отладки в целом), тем большей становится стоимость ПС. Отсюда вторая задача: определить момент окончания отладки ПС (или отдельной его компоненты). Признаком возможности окончания отладки является полнота охвата пропущенными через ПС тестами (т.е. тестами, к которым применено ПС) множества различных ситуаций, возникающих при выполнении программ ПС, и относительно редкое проявление ошибок в ПС на последнем отрезке процесса тестирования. Последнее определяется в соответствии с требуемой степенью надежности ПС, указанной в спецификации его качества.

Для оптимизации набора тестов, т.е. для подготовки такого набора тестов, который позволял бы при заданном их числе (или при заданном интервале времени, отведенном на тестирование) обнаруживать большее число ошибок в ПС, необходимо, во-первых, заранее планировать этот набор и, во-вторых, использовать рациональную стратегию планирования (проектирования [10.1]) тестов. Проектирование тестов можно начинать сразу же после завершения этапа внешнего описания ПС. Возможны разные подходы к выработке стратегии проектирования тестов, которые можно условно графически разместить (см. рис. 9.1) между следующими двумя крайними подходами [10.1]. Левый крайний подход заключается в том, что тесты проектируются только на основании изучения спецификаций ПС (внешнего описания, описания архитектуры и спецификации модулей). Строение модулей при этом никак не учитывается, т.е. они рассматриваются как черные ящики. Фактически такой подход требует полного перебора всех наборов входных данных, так как в противном случае некоторые участки программ ПС могут не работать при пропуске любого теста, а это значит, что содержащиеся в них ошибки не будут проявляться. Однако тестирование ПС полным множеством наборов входных данных практически неосуществимо. Правый крайний подход заключается в том, что тесты проектируются на основании изучения текстов программ с целью протестировать все пути выполнения каждой программ ПС. Если принять во внимание наличие в программах циклов с переменным числом повторений, то различных путей выполнения программ ПС может оказаться также чрезвычайно много, так что их тестирование также будет практически неосуществимо.

Рис. 10.1. Спектр подходов к тестированию тестов.

Оптимальная стратегия проектирования тестов расположена внутри интервала между этими крайними подходами, но ближе к левому краю. Она включает проектирование значительной части тестов по спецификациям, но она требует также проектирования некоторых тестов и по текстам программ. При этом в первом случае эта стратегия базируется на принципах:

- на каждую используемую функцию или возможность - хотя бы один тест,
- на каждую область и на каждую границу изменения какой-либо входной величины - хотя бы один тест,
- на каждую особую (исключительную) ситуацию, указанную в спецификациях, - хотя бы один тест.

Во втором случае эта стратегия базируется на принципе: каждая команда каждой программы ПС должна проработать хотя бы на одном тесте.

Оптимальную стратегию проектирования тестов можно конкретизировать на основании следующего принципа: для каждого программного документа (включая тексты программ), входящего в состав ПС, должны проектироваться свои тесты с целью выявления в нем ошибок. Во всяком случае, этот принцип необходимо соблюдать в соответствии с определением ПС и содержанием понятия технологии программирования как технологии разработки надежных ПС (см. лекцию 1). В связи с этим Майерс даже определяет разные виды тестирования [10.1] в зависимости от вида программного документа, на основании которого строятся тесты. В нашей стране различаются [10.8] два основных вида отладки (включая тестирование): автономную и комплексную отладку ПС. *Автономная* отладка ПС означает последовательное раздельное тестирование различных частей программ, входящих в ПС, с поиском и исправлением в них фиксируемых при тестировании ошибок. Она фактически включает отладку каждого программного модуля и отладку сопряжения модулей. *Комплексная* отладка означает тестирование ПС в целом с поиском и исправлением фиксируемых при тестировании ошибок во всех документах (включая тексты программ ПС), относящихся к ПС в целом. К таким документам относятся определение требований к ПС, спецификация качества ПС, функциональная спецификация ПС, описание архитектуры ПС и тексты программ ПС.

10.3. Заповеди отладки программного средства.

В этом разделе даются общие рекомендации по организации отладки ПС. Но сначала следует отметить некоторый феномен [10.1], который подтверждает важность предупреждения ошибок на предыдущих этапах разработки: по мере роста числа обнаруженных и исправленных ошибок в ПС *растет* также относительная вероятность существования в нем необнаруженных ошибок. Это объясняется тем, что при росте числа ошибок, обнаруженных в ПС, уточняется и наше представление об общем числе допущенных в нем ошибок, а значит, в какой-то мере, и о числе необнаруженных еще ошибок.

Ниже приводятся рекомендации по организации отладки в форме заповедей [10.1, 10.8].

Заповедь 1. Считайте тестирование ключевой задачей разработки ПС, поручайте его самым квалифицированным и одаренным программистам; нежелательно тестировать свою собственную программу.

Заповедь 2. Хорош тот тест, для которого высока вероятность обнаружить ошибку, а не тот, который демонстрирует правильную работу программы.

Заповедь 3. Готовьте тесты как для правильных, так и для неправильных данных.

Заповедь 4. Документируйте пропуск тестов через компьютер; детально изучайте результаты каждого теста; избегайте тестов, пропуск которых нельзя повторить.

Заповедь 5. Каждый модуль подключайте к программе только один раз; никогда не изменяйте программу, чтобы облегчить ее тестирование.

Заповедь 6. Пропускайте заново все тесты, связанные с проверкой работы какой-либо программы ПС или ее взаимодействия с другими программами, если в нее были внесены изменения (например, в результате устранения ошибки).

10.4. Автономная отладка программного средства.

При автономной отладке ПС каждый модуль на самом деле тестируется в некотором программном окружении, кроме случая, когда отлаживаемая программа состоит только из одного модуля. Это окружение состоит [10.8] из других модулей, часть которых является модулями отлаживаемой программы, которые уже отлажены, а часть модулями, управляющими отладкой *(отладочными* модулями, см. ниже). Таким образом, при автономной отладке тестируется всегда некоторая программа *(тестируемая программа),* построенная специально для тестирования отлаживаемого модуля. Эта программа лишь частично совпадает с отлаживаемой программой, кроме случая, когда отлаживается последний модуль отлаживаемой программы. В процессе автономной отладки ПС производится наращивание тестируемой программы отлаженными модулями: при переходе к отладке следующего модуля в его программное окружение добавляется последний отлаженный модуль. Такой процесс наращивания программного окружения отлаженными модулями называется *интеграцией* программы [10.1]. Отладочные модули, входящие в окружение отлаживаемого модуля, зависят от порядка, в каком отлаживаются модули этой программы, от того, какой модуль отлаживается и, возможно, от того, какой тест будет пропускаться.

При восходящем тестировании (см. лекцию 7) это окружение будет содержать только один отладочный модуль (кроме случая, когда отлаживается последний модуль отлаживаемой программы), который будет головным в тестируемой программе. Такой отладочный модуль называют *ведущим* (или драйвером [10.1]). Ведущий отладочный модуль подготавливает информационную среду для тестирования отлаживаемого модуля (т. е. формирует ее состояние, требуемое для тестирования этого модуля, в частности, путем ввода некоторых тестовых данных), осуществляет обращение к отлаживаемому модулю и после окончания его работы выдает необходимые сообщения. При отладке одного модуля для разных тестов могут составляться разные ведущие отладочные модули.

При нисходящем тестировании (см. лекцию 7) окружение отлаживаемого модуля в качестве отладочных модулей содержит *отладочные имитаторы* (заглушки) некоторых еще не отлаженных модулей. К таким модулям относятся, прежде всего, все модули, к которым может обращаться отлаживаемый модуль, а также еще не отлаженные модули, к которым могут обращаться уже отлаженные модули (включенные в это окружение). Некоторые из этих имитаторов при отладке одного модуля могут изменяться для разных тестов.

На практике в окружении отлаживаемого модуля могут содержаться отладочные модули обоих типов, если используется смешанная стратегия тестирования. Это связано с тем, что как восходящее, так и нисходящее тестирование имеет свои достоинства и свои недостатки.

К *достоинствам восходящего тестирования* относятся:

- простота подготовки тестов,
- возможность полной реализации плана тестирования модуля.

Это связано с тем, что тестовое состояние информационной среды готовится непосредственно перед обращением к отлаживаемому модулю (ведущим отладочным модулем).

Недостатками восходящего тестирования являются следующие его особенности:

- тестовые данные готовятся, как правило, не в той форме, которая рассчитана на пользователя (кроме случая, когда отлаживается последний,

головной, модуль отлаживаемой программ);

• большой объем отладочного программирования (при отладке одного модуля приходится составлять много ведущих отладочных модулей, формирующих подходящее состояние информационной среды для разных тестов);

• необходимость специального тестирования сопряжения модулей.

К *достоинствам нисходящего тестирования* относятся следующие его особенности:

• большинство тестов готовится в форме, рассчитанной на пользователя;

• во многих случаях относительно небольшой объем отладочного программирования (имитаторы модулей, как правило, весьма просты и каждый пригоден для большого числа, нередко - для всех, тестов);

• отпадает необходимость тестирования сопряжения модулей.

Недостатком нисходящего тестирования является то, что тестовое состояние информационной среды перед обращением к отлаживаемому модулю готовится косвенно — оно является результатом применения уже отлаженных модулей к тестовым данным или данным, выдаваемым имитаторами. Это, во-первых, затрудняет подготовку тестов и требует высокой квалификации тестовика (разработчика тестов), а во-вторых, делает затруднительным или даже невозможным реализацию полного плана тестирования отлаживаемого модуля. Указанный недостаток иногда вынуждает разработчиков применять восходящее тестирование даже в случае нисходящей разработки. Однако чаще применяют некоторые модификации нисходящего тестирования, либо некоторую комбинацию нисходящего и восходящего тестирования. Исходя из того, что нисходящее тестирование, в принципе, является предпочтительным, остановимся на приемах, позволяющих в какой-то мере преодолеть указанные трудности.

Прежде всего, необходимо организовать отладку программы таким образом, чтобы как можно раньше были отлажены модули, осуществляющие ввод данных, - тогда тестовые данные можно готовить в форме, рассчитанной на пользователя, что существенно упростит подготовку последующих тестов. Далеко не всегда этот ввод осуществляется в головном модуле, поэтому приходится в первую очередь отлаживать цепочки модулей, ведущие к модулям, осуществляющим указанный ввод (ср. с методом целенаправленной конструктивной реализации в лекции 7). Пока модули, осуществляющие ввод данных, не отлажены, тестовые данные поставляются некоторыми имитаторами: они либо включаются в имитатор как его часть, либо вводятся этим имитатором.

При нисходящем тестировании некоторые состояния информационной среды, при которых требуется тестировать отлаживаемый модуль, могут не возникать при выполнении отлаживаемой программы ни при каких входных данных. В этих случаях можно было бы вообще не тестировать отлаживаемый модуль, так как обнаруживаемые при этом ошибки не будут проявляться при выполнении отлаживаемой программы ни при каких входных данных. Однако так поступать не рекомендуется, так как при изменениях отлаживаемой программы (например, при сопровождении ПС) не использованные для тестирования отлаживаемого модуля состояния информационной среды могут уже возникать, что требует дополнительного тестирования этого модуля (а этого при рациональной организации отладки можно было бы не делать, если сам данный модуль не изменялся). Для осуществления тестирования отлаживаемого модуля в указанных ситуациях иногда используют подходящие имитаторы, чтобы создать требуемое состояние информационной среды. Чаще же пользуются модифицированным вариантом нисходящего тестирования, при котором отлаживаемые модули перед их интеграцией предварительно тестируются отдельно (в этом случае в окружении отлаживаемого модуля появляется ведущий отладочный модуль, наряду с имитаторами модулей, к которым может обращаться отлаживаемый модуль). Однако, представляется более целесообразной другая модификация нисходящего тестирования: после завершения нисходящего тестирования отлаживаемого модуля для достижимых

тестовых состояний информационной среды следует его отдельно протестировать для остальных требуемых состояний информационной среды.

Часто применяют также комбинацию восходящего и нисходящего тестирования, которую называют методом *сандвича* [10.1]. Сущность этого метода заключается в одновременном осуществлении как восходящего, так и нисходящего тестирования, пока эти два процесса тестирования не встретятся на каком-либо модуле где-то в середине структуры отлаживаемой программы. Этот метод при разумном порядке тестирования позволяет воспользоваться достоинствами как восходящего, так и нисходящего тестирования, а также в значительной степени нейтрализовать их недостатки.

Весьма важным при автономной отладке является тестирование сопряжения модулей. Дело в том, что спецификация каждого модуля программы, кроме головного, используется в этой программы в двух ситуациях: во-первых, при разработке текста (иногда говорят: тела) этого модуля и, во-вторых, при написании обращения к этому модулю в других модулях программы. И в том, и в другом случае в результате ошибки может быть нарушено требуемое соответствие заданной спецификации модуля. Такие ошибки требуется обнаруживать и устранять. Для этого и предназначено тестирование сопряжения модулей. При нисходящем тестировании тестирование сопряжения осуществляется попутно каждым пропускаемым тестом, что считают достоинством нисходящего тестирования. При восходящем тестировании обращение к отлаживаемому модулю производится не из модулей отлаживаемой программы, а из ведущего отладочного модуля. В связи с этим существует опасность, что последний модуль может приспособиться к некоторым "заблуждениям" отлаживаемого модуля. Поэтому, приступая (в процессе интеграции программы) к отладке нового модуля, приходится тестировать каждое обращение к ранее отлаженному модулю с целью обнаружения несогласованности этого обращения с телом соответствующего модуля (и не исключено, что виноват в этом ранее отлаженный модуль). Таким образом, приходится частично повторять в новых условиях тестирование ранее отлаженного модуля, при этом возникают те же трудности, что и при нисходящем тестировании.

Автономное тестирование модуля целесообразно осуществлять в четыре последовательно выполняемых шага [10.1].

Шаг 1. На основании спецификации отлаживаемого модуля подготовьте тесты для каждой возможности и каждой ситуации, для каждой границы областей допустимых значений всех входных данных, для каждой области изменения данных, для каждой области недопустимых значений всех входных данных и каждого недопустимого условия.

Шаг 2. Проверьте текст модуля, чтобы убедиться, что каждое направление любого разветвления будет пройдено хотя бы на одном тесте. Добавьте недостающие тесты.

Шаг 3. Проверьте текст модуля, чтобы убедиться, что для каждого цикла существуют тесты, обеспечивающие, по крайней мере, три следующие ситуации: тело цикла не выполняется ни разу, тело цикла выполняется один раз и тело цикла выполняется максимальное число раз. Добавьте недостающие тесты.

Шаг 4. Проверьте текст модуля, чтобы убедиться, что существуют тесты, проверяющие чувствительность к отдельным особым значениям входных данных. Добавьте недостающие тесты.

10.5. Комплексная отладка программного средства.

Как уже было сказано выше, при комплексной отладке тестируется ПС в целом, причем тесты готовятся по каждому из документов ПС [10.8]. Тестирование этих документов производится, как правило, в порядке, обратном их разработке. Исключение составляет лишь тестирование документации по применению, которая разрабатывается по внешнему описанию параллельно с разработкой текстов программ - это тестирование лучше производить после завершения тестирования внешнего описания. Тестирование при комплексной отладке представляет собой применение ПС к конкретным данным, которые в принципе могут возникнуть у пользователя (в частности, все тесты готовятся в форме, рассчитанной на

пользователя), но, возможно, в моделируемой (а не в реальной) среде. Например, некоторые недоступные при комплексной отладке устройства ввода и вывода могут быть заменены их программными имитаторами.

Тестирование архитектуры ПС. Целью тестирования является поиск несоответствия между описанием архитектуры и совокупностью программ ПС. К моменту начала тестирования архитектуры ПС должна быть уже закончена автономная отладка каждой подсистемы. Ошибки реализации архитектуры могут быть связаны, прежде всего, с взаимодействием этих подсистем, в частности, с реализацией архитектурных функций (если они есть). Поэтому хотелось бы проверить все пути взаимодействия между подсистемами ПС. При этом желательно хотя бы протестировать все цепочки выполнения подсистем без повторного вхождения последних. Если заданная архитектура представляет ПС в качестве малой системы из выделенных подсистем, то число таких цепочек будет вполне обозримо.

Тестирование внешних функций. Целью тестирования является поиск расхождений между функциональной спецификацией и совокупностью программ ПС. Несмотря на то, что все эти программы автономно уже отлажены, указанные расхождения могут быть, например, из-за несоответствия внутренних спецификаций программ и их модулей (на основании которых производилось автономное тестирование) функциональной спецификации ПС. Как правило, тестирование внешних функций производится так же, как и тестирование модулей на первом шаге, т.е. как черного ящика.

Тестирование качества ПС. Целью тестирования является поиск нарушений требований качества, сформулированных в спецификации качества ПС. Это наиболее трудный и наименее изученный вид тестирования. Ясно лишь, что далеко не каждый примитив качества ПС может быть испытан тестированием (об оценке качества ПС см. лекцию 14). Завершенность ПС проверяется уже при тестировании внешних функций. На данном этапе тестирование этого примитива качества может быть продолжено, если требуется получить какую-либо вероятностную оценку степени надежности ПС. Однако, методика такого тестирования еще требует своей разработки. Могут тестироваться такие примитивы качества, как точность, устойчивость, защищенность, временная эффективность, в какой-то мере - эффективность по памяти, эффективность по устройствам, расширяемость и, частично, независимость от устройств. Каждый из этих видов тестирования имеет свою специфику и заслуживает отдельного рассмотрения. Мы здесь ограничимся лишь их перечислением. Легкость применения ПС (критерий качества, включающий несколько примитивов качества, см. лекцию 4) оценивается при тестировании документации по применению ПС.

Тестирование документации по применению ПС. Целью тестирования является поиск несогласованности документации по применению и совокупностью программ ПС, а также выявление неудобств, возникающих при применении ПС. Этот этап непосредственно предшествует подключению пользователя к завершению разработки ПС (тестированию определения требований к ПС и аттестации ПС), поэтому весьма важно разработчикам сначала самим воспользоваться ПС так, как это будет делать пользователь [10.1]. Все тесты на этом этапе готовятся исключительно на основании только документации по применению ПС. Прежде всего, должны тестироваться возможности ПС как это делалось при тестировании внешних функций, но только на основании документации по применению. Должны быть протестированы все неясные места в документации, а также все примеры, использованные в документации. Далее тестируются наиболее трудные случаи применения ПС с целью обнаружить нарушение требований относительности легкости применения ПС.

Тестирование определения требовании к ПС. Целью тестирования является выяснение, в какой мере ПС не соответствует предъявленному определению требований к нему. Особенность этого вида тестирования заключается в том, что его осуществляет организация-покупатель или организация-пользователь ПС [10.1] как один из путей преодоления барьера между разработчиком и пользователем (см. лекцию 3). Обычно это те-

стирование производится с помощью контрольных задач - типовых задач, для которых известен результат решения. В тех случаях, когда разрабатываемое ПС должно придти на смену другой версии ПС, которая решает хотя бы часть задач разрабатываемого ПС, тестирование производится путем решения общих задач с помощью как старого, так и нового ПС (с последующим сопоставлением полученных результатов). Иногда в качестве формы такого тестирования используют *опытную* эксплуатацию ПС — ограниченное применение нового ПС с анализом использования результатов в практической деятельности. По существу, этот вид тестирования во многом перекликается с испытанием ПС при его аттестации (см. лекцию 14), но выполняется до аттестации, а иногда и вместо аттестации.

Упражнения к лекции 10.

10.1. Что такое отладка программного средства?
10.2. Что такое тестирование программного средства?
10.3. Что такое автономная отладка программного средства?
10.4. Что такое комплексная отладка программного средства?
10.5. Что такое ведущий отладочный модуль?
10.6. Что такое отладочный имитатор программного модуля?

Лекция № 11.

ОБЕСПЕЧЕНИЕ ФУНКЦИОНАЛЬНОСТИ И НАДЕЖНОСТИ ПРОГРАММНОГО СРЕДСТВА

11.1. Функциональность и надежность как обязательные критерии качества программного средства.

На предыдущих лекциях мы рассмотрели все этапы разработки ПС, кроме его аттестации. При этом мы не касались вопросов обеспечения качества ПС в соответствии с его спецификацией качества (см. лекцию 4). Правда, занимаясь реализацией функциональной спецификации ПС, мы тем самым обсудили основные вопросы обеспечения критерия функциональности. Объявив надежность ПС основным его атрибутом (см. лекцию 1), мы выбрали *предупреждение ошибок* в качестве основного подхода для обеспечения надежности ПС (см. лекцию 3) и обсудили его воплощение на разных этапах разработки ПС. Таким образом, проявлялся тезис об обязательности функциональности и надежности ПС как критериев его качества.

Тем не менее, в спецификации качества ПС могут быть дополнительные характеристики этих критериев, обеспечение которых требуют специального обсуждения. Этим вопросам и посвящена настоящая лекция. Обеспечение других критериев качества будет обсуждаться в следующей лекции.

Ниже обсуждаются обеспечение примитивов качества ПС, выражающих критерии функциональности и надежности ПС.

11.2. Обеспечение завершенности программного средства.

Завершенность ПС является общим примитивом качества ПС для выражения и функциональности и надежности ПС, причем для функциональности она является единственным примитивом (см. лекцию 4).

Функциональность ПС определяется его функциональной спецификацией. Завершенность ПС как примитив его качества является мерой того, в какой степени эта спецификация реализована в разрабатываемом ПС. Обеспечение этого примитива в полном объеме означает реализацию каждой из функций, определенной в функциональной спецификации, со всеми указанными там деталями и особенностями. Все рассмотренные ранее технологические процессы показывают, как это может быть сделано.

Однако в спецификации качества ПС могут быть определены несколько уровней реализации функциональности ПС: может быть определена некоторая упрощенная (начальная или стартовая) версия, которая должна быть реализована в первую очередь; могут быть также определены и несколько промежуточных версий. В этом случае возникает дополнительная технологическая задача: организация наращивания функциональности ПС. Здесь важно отметить, что разработка упрощенной версии ПС не есть разработка его

прототипа. Прототип разрабатывается для того, чтобы лучше понять условия применения будущего ПС [11.1], уточнить его внешнее описание. Он рассчитан на избранных пользователей и поэтому может сильно отличаться от требуемого ПС не только выполняемыми функциями, но и особенностями пользовательского интерфейса. Упрощенная же версия разрабатываемого ПС должна быть рассчитана на *практически полезное* применение любыми пользователями, для которых предназначено это ПС. Поэтому главный принцип обеспечения функциональности такого ПС заключается в том, чтобы с самого начала разрабатывать ПС таким образом, как будто требуется ПС в полном объеме, до тех пор, когда разработчики будут иметь дело непосредственно с теми частями или деталями ПС, реализацию которых можно отложить в соответствии со спецификацией его качества. Тем самым, и внешнее описание и описание архитектуры ПС должно быть разработано в полном объеме. Можно отложить лишь реализацию тех программных подсистем (определенных в архитектуре разрабатываемого ПС), функционирования которых не требуется в начальной версии этого ПС. Реализацию же самих программных подсистем лучше всего производить методом целенаправленной конструктивной реализации, оставляя в начальной версии ПС подходящие имитаторы тех программных модулей, функционирование которых в этой версии не требуется. Допустима также упрощенная реализация некоторых программных модулей, опускающая реализацию некоторых деталей соответствующих функций. Однако такие модули с технологической точки зрения лучше рассматривать как своеобразные их имитаторы (хотя и далеко продвинутые).

Достигнутый при обеспечении функциональности ПС уровень его завершенности на самом деле может быть не таким, как ожидалось, из-за ошибок, оставшихся в этом ПС. Можно лишь говорить, что требуемая завершенность достигнута с некоторой вероятностью, определяемой объемом и качеством проведенного тестирования. Для того чтобы повысить эту вероятность, необходимо продолжить тестирование и отладку ПС. Однако, оценивание такой вероятности является весьма специфической задачей, которая пока еще ждет соответствующих теоретических исследований.

11.3. Обеспечение точности программного средства.

Обеспечение этого примитива качества связано с действиями над значениями вещественных типов (точнее говоря, со значениями, представляемыми с некоторой погрешностью). Обеспечить требуемую точность при вычислении значения той или иной функции - значит получить это значение с погрешностью, не выходящей за рамки заданных границ. Видами погрешности, методами их оценки и методами достижения требуемой точности (т.н. *приближенными вычислениями]* занимается вычислительная математика [11.1, 11.2]. Здесь мы лишь обратим внимание на некоторую структуру погрешности: погрешность вычисленного значения *(полная погрешность)* зависит

- от *погрешности используемого метода* вычисления (в которую мы включаем и неточность используемой модели),
- от погрешности представления используемых данных (от т.н. *неустранимой погрешности),*
- от *погрешности округления* (неточности выполнения используемых в методе операций).

11.4. Обеспечение автономности программного средства.

Вопрос об автономности программного средства решается путем принятия решения о возможности использования в разрабатываемом ПС какого-либо подходящего базового программного обеспечения. Надежность имеющегося в распоряжении разработчиков базового программного обеспечения для целевого компьютера может не отвечать требованиям к надежности разрабатываемого ПС. Поэтому от использования такого программного обеспечения приходиться отказываться, а его функции в требуемом объеме приходится реализовывать в рамках разрабатываемого ПС. Аналогичное решение приходится принимать при жестких ограничениях на используемые ресурсы (по критерию эффективности ПС).

Такое решение может быть принято уже в процессе разработки спецификации качества ПС, иногда — на этапе конструирования ПС.

11.5. Обеспечение устойчивости программного средства.

Этот примитив качества ПС обеспечивается с помощью так называемого *защитного программирования.* Вообще говоря, защитное программирование применяется при программировании модуля для повышения надежности ПС в более широком смысле. Как утверждает Майерс [11.3], «защитное программирование основано на важной предпосылке: худшее, что может сделать модуль, - это принять неправильные входные данные и затем вернуть неверный, но правдоподобный результат». Для того, чтобы этого избежать, в текст модуля включают проверки его входных и выходных данных на их корректность в соответствии со спецификацией этого модуля, в частности, должны быть проверены выполнение ограничений на входные и выходные данные и соотношений между ними, указанные в спецификации модуля. В случае отрицательного результата проверки возбуждается соответствующая исключительная ситуация. Для обработки таких ситуаций в конец этого модуля включаются фрагменты второго рода обработчики соответствующих исключительных ситуаций. Эти обработчики помимо выдачи необходимой диагностической информации, могут принять меры либо по исключению ошибки в данных (например, потребовать их повторного ввода), либо по ослаблению влияния ошибки (например, во избежание поломки устройств, управляемых с помощью данного ПС, при аварийном прекращении выполнения программы осуществляют мягкую их остановку).

Применение защитного программирования модулей приводит к снижению эффективности ПС как по времени, так и по памяти. Поэтому необходимо разумно регулировать степень применения защитного программирования в зависимости от требований к надежности и эффективности ПС, сформулированным в спецификации качества разрабатываемого ПС. Входные данные разрабатываемого модуля могут поступать как непосредственно от пользователя, так и от других модулей. Наиболее употребительным случаем применения защитного программирования является применение его для первой группы данных, что и означает реализацию устойчивости ПС. Это нужно делать всегда, когда в спецификации качества ПС имеется требование об обеспечении устойчивости ПС. Применение защитного программирования для второй группы входных данных означает попытку обнаружить ошибку в других модулях во время выполнения разрабатываемого модуля, а для выходных данных разрабатываемого модуля - попытку обнаружить ошибку в самом этом модуле во время его выполнения. По существу, это означает частичное воплощение подхода самообнаружения ошибок для обеспечения надежности ПС, о чем шла речь в лекции 3. Этот случай защитного программирования применяется крайне редко - только в том случае, когда требования к надежности ПС чрезвычайно высоки.

11.6. Обеспечение защищенности программных средств.

Различают следующие виды защиты ПС от искажения информации:

- защита от сбоев аппаратуры;
- защита от влияния «чужой» программы;
- защита от отказов «своей» программы;
- защита от ошибок оператора (пользователя);
- защита от несанкционированного доступа;
- защита от защиты.

Защита от сбоев аппаратуры. В настоящее время этот вид защиты является не очень злободневной задачей (с учетом уровня достигнутой надежности компьютеров). Но все же полезно знать ее решение. Это обеспечивается организацией т.н. «двойных или тройных просчетов». Для этого весь процесс обработки данных, определяемый ПС, разбивается по времени на интервалы так называемыми «опорными точками». Длина этого интервала не должна превосходить половины среднего времени безотказной работы компьютера. В начале каждого такого интервала во вторичную память записывается с некоторой контрольной суммой копия состояния изменяемой в этом процессе памяти («опорная точка»).

Для того, чтобы убедиться, что обработка данных от одной опорной точки до следующей (т.е. один «просчет») произведена правильно (без сбоев компьютера), производится два таких «просчета». После первого «просчета» вычисляется и запоминается указанная контрольная сумма, а затем восстанавливается состояние памяти по опорной точке и делается второй «просчет». После второго «просчета» вычисляется снова указанная контрольная сумма, которая затем сравнивается с контрольной суммой первого «просчета». Если эти две контрольные суммы совпадают, второй просчет считается правильным, в противном случае контрольная сумма второго «просчета» также запоминается и производится третий «просчет» (с предварительным восстановлением состояния памяти по опорной точке). Если контрольная сумма третьего «просчета» совпадет с контрольной суммой одного из первых двух «просчетов», то третий просчет считается правильным, в противном случае требуется инженерная проверка компьютера.

Защита от влияния «чужой» программы. При появлении мультипрограммного режима работы компьютера в его памяти может одновременно находиться в стадии выполнения несколько программ, попеременно получающих управление в результате возникающих прерываний (т.н. квазипараллельное выполнение программ). Одна из таких программ (обычно: операционная система) занимается обработкой прерываний и управлением мультипрограммным режимом. Здесь под «чужой» программой понимается программа (или какой-либо программный фрагмент), выполняемая параллельно (или квазипараллельно) по отношению к защищаемой программе (или ее фрагменту). Этот вид защиты должна обеспечить, чтобы эффект выполнения защищаемой программы не зависел от того, какие программы выполняются параллельно с ней, и относится, прежде всего, к функциям операционных систем.

Различают две разновидности этой защиты:

- защита от отказов «чужой» программы,
- защита от злонамеренного влияния «чужой» программы.

Защита от отказов «чужой» программы означает, что на выполнение функций защищаемой программой не будут влиять отказы (проявления ошибок), возникающие в параллельно выполняемых программах. Для того чтобы управляющая программа (операционная система) могла обеспечить защиту себя и других программ от такого влияния, аппаратура компьютера должна реализовывать следующие возможности:

- защиту памяти,
- два режима функционирования компьютера: привилегированный и
- рабочий (пользовательский),
- два вида операций: привилегированные и ординарные,
- корректную реализацию прерываний и начального включения компьютера,
- временное прерывание.
- Защита памяти означает возможность программным путем задавать для каждой программы недоступные для нее участки памяти. В привилегированном режиме могут выполняться любые операции (как ординарные, так и привилегированные), а в рабочем режиме – только ординарные. Попытка выполнить привилегированную операцию, а также обратиться к защищенной памяти в рабочем режиме вызывает соответствующее прерывание. К привилегированным операциям относятся операции изменения защиты памяти и режима функционирования, а также доступа к внешней информационной среде. Корректная реализация прерываний и начального включения компьютера означает обязательную установку привилегированного режима и отмену защиты памяти. В этих условиях управляющая программа (операционная система) может полностью защитить себя от влияния отказов других программ. Для этого достаточно, чтобы:

- все точки передачи управления при начальном включении компьютера и при прерываниях принадлежали этой программе,
- она не позволяла никакой другой программе работать в привилегированном режиме (при передаче управления любой другой программе должен вклю-

чаться только рабочий режим),

- она полностью защищала свою память (содержащую, в частности, всю ее управляющую информацию, включая так называемые вектора прерываний) от других программ.

Тогда никто не помешает ей выполнять любые реализованные в ней функции защиты других программ (в том числе и доступа к внешней информационной среде). Для облегчения решения этой задачи часть такой программы помещается в постоянную память. Наличие временного прерывания позволяет управляющей программе защититься от зацикливания в других программах (без такого прерывания она могла бы просто лишиться возможности управлять).

Защита от злонамеренного влияния «чужих» программ означает, что изменение внешней информационной среды, предоставленной защищаемой программе, со стороны другой, параллельно выполняемой программы будет невозможно или сильно затруднено без ведома защищаемой программы. Для этого операционная система должна обеспечить подходящий контроль доступа к внешней информационной среде. Необходимым условием обеспечение такого контроля является обеспечения защиты от злонамеренного влияния «чужих» программ хотя бы самой операционной системы. В противном случае такой контроль можно было бы обойти путем изменения операционной системы со стороны «злонамеренной» программы.

Этот вид защиты включает, в частности, и защиту от т.н. «компьютерных вирусов», под которыми понимают фрагменты программ, способные в процессе своего выполнения внедряться (копироваться) в другие программы (или в отдельные программные фрагменты). «Компьютерные вирусы», обладая способностью к размножению (к внедрению в другие программы), при определенных условиях вызывают изменение эффекта выполнения «зараженной» программы, что может привести к серьезным деструктивным изменениям ее внешней информационной среды. Операционная система, будучи защищенной от влияния «чужих» программ, может ограничить доступ к программным фрагментам, хранящимся во внешней информационной среде. Так, например, может быть запрещено изменение таких фрагментов любыми программами, кроме некоторых, которые знает операционная система, или, другой вариант, может быть разрешено только после специальных подтверждений программы (или пользователя).

Защита от отказов «своей» программы. Обеспечивается надежностью ПС, на что ориентирована вся технология программирования, обсуждаемая в настоящем курсе лекций.

Защита от ошибок пользователя. Здесь идет речь не об ошибочных данных, поступающих от пользователя ПС, - защита от них связана с обеспечением устойчивости ПС, а о действиях пользователя, приводящих к деструктивному изменению состояния внешней информационной среды ПС, несмотря на корректность используемых при этом данных. Защита от таких действий, частично, обеспечивается выдачей предупредительных сообщений о попытках изменить состояние внешней информационной среды ПС с требованием подтверждения этих действий. Для случаев же, когда такие ошибки совершаются, может быть предусмотрена возможность восстановления состояния отдельных компонент внешней информационной среды ПС на определенные моменты времени. Такая возможность базируется на ведении (формировании) архива состояний (или изменений состояния) внешней информационной среды.

Защита от несанкционированного доступа. Каждому пользователю ПС предоставляет определенные информационные и процедурные ресурсы (услуги), причем у разных пользователей ПС предоставленные им ресурсы могут отличаться, иногда очень существенно. Этот вид защиты должен обеспечить, чтобы каждый пользователь ПС мог использовать только то, что ему предоставлено (санкционировано). Для этого ПС в своей внешней информационной среде может хранить информацию о своих пользователях и предоставленным им правах использования ресурсов, а также предоставлять пользователям определенные возможности формирования этой информации. Защита от несанкциониро-

ванного доступа к ресурсам ПС осуществляется с помощью т.н. *паролей* (секретных слов). При этом предполагается, что каждый пользователь знает только свой пароль, зарегистрированный в ПС этим пользователем. Для доступа к выделенным ему ресурсам он должен предъявить ПС свой пароль. Другими словами, пользователь как бы "вешает замок" на предоставленные ему права доступа к ресурсам, "ключ" от которого имеется только у этого пользователя.

Различают две разновидности такой защиты:

- простая защита от несанкционированного доступа,
- защита от взлома защиты.

Простая защита от несанкционированного доступа обеспечивает защиту от использования ресурсов ПС пользователем, которому не предоставлены соответствующие права доступа или который указал неправильный пароль. При этом предполагается, что пользователь, получив отказ в доступе к интересующим ему ресурсам, не будет предпринимать попыток каким-либо несанкционированным образом обойти или преодолеть эту защиту. Поэтому этот вид защиты может применяться и в ПС, которая базируется на операционной системе, не обеспечивающей полную защиту от влияния «чужих» программ.

Защита от взлома защиты -- это такая разновидность защиты от несанкционированного доступа, которая существенно затрудняет преодоление этой защиты. Это связано с тем, что в отдельных случаях могут быть предприняты настойчивые попытки взломать защиту от несанкционированного доступа, если защищаемые ресурсы представляют для кого-то чрезвычайную ценность. Для такого случая приходится предпринимать дополнительные меры защиты. Во-первых, необходимо обеспечить, чтобы такую защиту нельзя было обойти, т. е. должна действовать защита от влияния «чужих» программ. Во-вторых, необходимо усилить простую защиту от несанкционированного доступа использованием в ПС специальных программистских приемов, в достаточной степени затрудняющих подбор подходящего пароля или его вычисление по информации, хранящейся во внешней информационной среде ПС. Использование обычных паролей оказывается недостаточной, когда речь идет о чрезвычайно настойчивом стремлении добиться доступа к ценной информации. Если требуемый пароль («замок») в явном виде хранится во внешней информационной среде ПС, то "взломщик" этой защиты относительно легко может его достать, имея доступ к этому ПС. Кроме того, следует иметь в виду, что с помощью современных компьютеров можно осуществлять достаточно большой перебор возможных паролей с целью найти подходящий.

Защититься от этого можно следующим образом. Пароль (секретное слово или просто секретное целое число) X должен быть известен только владельцу защищаемых прав доступа и он не должен храниться во внешней информационной среде ПС. Для проверки прав доступа во внешней информационной среде ПС хранится другое число Y=F(X), однозначно вычисляемое ПС по предъявленному паролю X. При этом функция F может быть хорошо известной всем пользователям ПС, однако она должна обладать таким свойством, что восстановление слова X по Y практически невозможно: при достаточно большой длине слова X (например, в несколько сотен знаков) для этого может потребоваться астрономическое время. Такое число Y будем называть *электронной (компьютерной] подписью* владельца пароля X (а значит, и защищаемых прав доступа).

Другой способ защиты от взлома защиты связан с защитой сообщений, пересылаемых по компьютерным сетям. Такое сообщение может представлять команду на дистанционный доступ к ценной информации, и этот доступ отправитель сообщения хочет защитить от возможных искажений. Например, при осуществлении банковских операций с использованием компьютерной сети. Использование компьютерной подписи в такой ситуации недостаточно, так как защищаемое сообщение может быть перехвачено «взломщиком» (например, на «перевалочных» пунктах компьютерной сети) и подменено другим сообщением с сохранением компьютерной подписи (или пароля).

Защиту от такого взлома защиты можно осуществить следующим образом [11.4]. Наряду с функцией F, определяющей компьютерную подпись владельца пароля X, в ПС определены еще две функции: Stamp и Notary. При передаче сообщения отправитель, помимо компьютерной подписи Y=F(X), должен вычислить еще другое число S=Stamp(X,R), где X - пароль, а R - текст передаваемого сообщения. Здесь также предполагается, что функция Stamp хорошо известна всем пользователям ПС и обладает таким свойством, что по S практически невозможно ни восстановить число X, ни подобрать другой текст сообщения R с заданной компьютерной подписью Y. При этом передаваемое сообщение (вместе со своей защитой) должно иметь вид: (R Y S), причем Y (компьютерная подпись) позволяет получателю сообщения установить истинность клиента, а S как бы скрепляет защищаемый текст сообщения R с компьютерной подписью Y. В связи с этим будем называть число S *электронной (компьютерной) печатью*. Функция Notary(R,Y,S).проверяет истинность защищаемого сообщения:

(R,Y,S).

Эта позволяет получателю сообщения однозначно установить, что текст сообщения R принадлежит владельцу пароля X.

Защита от защиты. Защита от несанкционированного доступа может создать нежелательную ситуацию для самого владельца прав доступа к ресурсам ПС - он не сможет воспользоваться этими правами, если забудет (или потеряет) свой пароль («ключ»). Для защиты интересов пользователя в таких ситуациях и предназначена защита от защиты. Для обеспечения такой защиты ПС должно иметь привилегированного пользователя, называемого *администратором ПС.* Администратор ПС должен, в частности, отвечать за функционирование защиты ПС: именно он должен формировать контингент пользователей данного экземпляра ПС, предоставляя каждому из этих пользователей определенные права доступа к ресурсам ПС. В ПС должна быть привилегированная операция (для администратора), позволяющая временно снимать защиту от несанкционированного доступа для пользователя с целью фиксации требуемого пароля («замка»).

Упражнения к лекции 11.

11.1. Что такое защитное программирование?

11.2. Какие виды защиты программного средства от искажения информации Вы знаете?

11.3. Какие требования предъявляются к компьютеру, чтобы можно было обеспечить защиту программы от отказов другой программы в мультипрограммном режиме?

11.4. Что такое компьютерная подпись?

11.5.Что такое компьютерная печать?

Лекция 12.

ОБЕСПЕЧЕНИЕ КАЧЕСТВА ПРОГРАММНОГО СРЕДСТВА

Общий обзор. Реализация пользовательского интерфейса и обеспечение легкости применения программного средства. Обеспечение эффективности программного средства. Обеспечение сопровождаемости и управление конфигурацией программного средства. Аппаратно-операционные платформы и обеспечение мобильности программного средства.

12.1. Общая характеристика процесса обеспечения качества программного средства.

Как уже отмечалось в лекции 4, спецификация качества определяет основные ориентиры (цели), которые на всех этапах разработки ПС так или иначе влияют при принятии различных решений на выбор подходящего варианта. Однако каждый примитив качества имеет свои особенности такого влияния, тем самым, обеспечение его наличия в ПС может потребовать своих подходов и методов разработки ПС или отдельных его частей. Кроме того, уже отмечалась ранее противоречивость критериев качества ПС, а также и выражающих их примитивов качества: хорошее обеспечение одного какого-либо примитива

качества ПС может существенно затруднить или сделать невозможным обеспечение некоторых других из этих примитивов. Поэтому существенная часть процесса обеспечения качества ПС состоит из поиска приемлемых компромиссов. Эти компромиссы частично должны быть определены уже в спецификации качества ПС: модель качества ПС должна конкретизировать требуемую степень присутствия в ПС каждого его примитива качества и определять приоритеты достижения этих степеней.

Обеспечение качества осуществляется в каждом технологическом процессе: принятые в нем решения в той или иной степени оказывают влияние на качество ПС в целом. В частности и потому, что значительная часть примитивов качества связана не столько со свойствами программ, входящих в ПС, сколько со свойствами документации. В силу отмеченной противоречивости примитивов качества весьма важно придерживаться выбранных приоритетов в их обеспечении. При этом следует придерживаться двух общих принципов:

- сначала необходимо обеспечить требуемую функциональность и надежность ПС, а затем уже доводить остальные критерии качества до приемлемого уровня их присутствия в ПС;
- нет никакой необходимости и, может быть, даже вредно добиваться более высокого уровня присутствия в ПС какого-либо примитива качества, чем тот, который определен в спецификации качества ПС.

Обеспечение функциональности и надежности ПС было рассмотрено в предыдущей лекции. Ниже обсуждается обеспечение других критериев качества ПС.

12.2. Обеспечение легкости применения программного средства.

Легкость применения, в значительной степени, определяется составом и качеством пользовательской документации, а также некоторыми свойствами, реализуемыми программным путем.

С пользовательской документацией связаны такие примитивы качества ПС, как *П-документированностъ* и *информативность*. Обеспечением ее качества занимаются обычно технические писатели. Этот вопрос будет обсуждаться в следующей лекции. Здесь лишь следует заметить, что там речь будет идти об автономной по отношению к программам документации. В связи с этим следует обратить внимание на широко используемый в настоящее время подход информирования пользователя в интерактивном режиме (в процессе применения программ ПС). Такое информирование во многих случаях оказывается более удобным для пользователя, чем с помощью автономной документации, так как позволяет пользователю без какого-либо поиска вызывать необходимую информацию за счет использования контекста ее вызова. Такой подход к информированию пользователя является весьма перспективным.

Программным путем реализуются такие примитивы качества ПС как *коммуникабельность, устойчивость* и *защищенность*. Обеспечение устойчивости и защищенности уже было рассмотрено в предыдущей лекции. Коммуникабельность обеспечивается соответствующей реализацией обработки исключительных ситуаций и созданием подходящего пользовательского интерфейса.

Возбуждение исключительной ситуации во многих случаях означает, что возникла необходимость информировать пользователя о ходе выполнения программы. При этом выдаваемая пользователю информация должна быть простой для понимания (см. лекцию 4). Однако исключительные ситуации возникают обычно на достаточно низком уровне модульной структуры программы, а создать понятное для пользователя сообщение можно, как правило, на более высоких уровнях этой структуры, где известен контекст, в котором были активизированы действия, приведшие к возникновению исключительной ситуации. Обработку исключительных ситуаций внутри модуля мы уже обсуждали в лекции 8. Для обработки возникшей исключительной ситуации в другом модуле приходится принимать не простые решения. Применяемый часто способ передачи информации о возникшей исключительной ситуации по цепочке обращений к программным модулям (в обратном

направлении) является тяжеловесным: он требует дополнительных проверок после возврата из модуля и часто усложняет само обращение к этим модулям за счет задания дополнительных параметров. Приемлемым решением является включение в операционную среду выполнения программ (в *исполнительную поддержку]* возможностей прямой передачи этой информации обработчикам исключительных ситуаций по динамически формируемой очереди таких обработчиков.

Пользовательский интерфейс представляет средство взаимодействия пользователя с ПС. При разработке пользовательского интерфейса следует учитывать потребности, опыт и способности пользователя [12.1]. Поэтому потенциальные пользователи должны быть вовлечены в процесс разработки такого интерфейса. Большой эффект здесь дает его прототипирование. При этом пользователи должны получить доступ к прототипам пользовательского интерфейса, а их оценка различных возможностей используемого прототипа должна существенно учитываться при создании окончательного варианта пользовательского интерфейса.

В силу большого разнообразия пользователей и видов ПС существует множество различных стилей пользовательских интерфейсов, при разработке которых могут использоваться разные принципы и подходы. Однако следующие важнейшие принципы следует соблюдать всегда [12.1]:

- пользовательский интерфейс должен базироваться на терминах и понятиях, знакомых пользователю;
- пользовательский интерфейс должен быть единообразным;
- пользовательский интерфейс должен позволять пользователю исправлять собственные ошибки;
- пользовательский интерфейс должен позволять получение пользователем справочной информации: как по его запросу, так и генерируемой ПС.

В настоящее время широко распространены командные и графические пользовательские интерфейсы.

Командный пользовательский интерфейс предоставляет пользователю возможность обращаться к ПС с некоторым заданием (запросом), представляемым некоторым текстом (командой) на специальном командном языке (языке заданий). Достоинствами такого интерфейса является возможность его реализации на дешевых алфавитно-цифровых терминалах и возможность минимизации требуемого от пользователя ввода с клавиатуры. Недостатками такого интерфейса являются необходимость изучения командного языка и достаточно большая вероятность ошибки пользователя при задании команды. В связи с этим командный пользовательский интерфейс обычно выбирают только опытные пользователи. Такой интерфейс позволяет им осуществлять быстрое взаимодействие с компьютером и предоставляет возможность объединять команды в процедуры и программы (см. например, язык Shell операционной системы Unix [12.2]).

Графический пользовательский интерфейс предоставляет пользователю возможности:

- обращаться к ПС путем выбора на экране подходящего графического или текстового объекта,
- получать от ПС информацию на экране в виде графических и текстовых объектов,
- осуществлять прямые манипуляции с графическими и текстовыми объектами, представленными на экране.

Графический пользовательский интерфейс позволяет

- размещать на экране множество различных окон, в которые можно выводить информацию независимо;
- использовать графические объекты, называемые *пиктограммами* (или *иконами),* для обозначения различных информационных объектов или процессов;
- использовать *экранный указатель* для выбора объектов (или их элементов),

размещенных на экране; экранный указатель управляется (перемещается) с помощью клавиатуры или мыши.

Достоинством графического пользовательского интерфейса является возможность создания удобной и понятной пользователю модели взаимодействия с ПС (панель управления, рабочий стол и т.п.) без необходимости изучения какого-либо специального языка. Однако его разработка требует больших трудозатрат, сравнимых с трудозатратами по создания самого ПС. Кроме того, возникает серьезная проблема по переносимости ПС на другие операционные системы, так как графический интерфейс существенно зависит от возможностей *(графической пользовательской платформы],* предоставляемых операционной системой для его создания.

Графический пользовательский интерфейс обобщает такие виды пользовательского интерфейса, как интерфейс типа меню и интерфейс прямого манипулирования.

12.3. Обеспечение эффективности программного средства.

Эффективность ПС обеспечивается принятием подходящих решений на разных этапах его разработки, начиная с разработки его архитектуры. Особенно сильно на эффективность ПС (особенно по памяти) влияет выбор структуры и представления данных. Но и выбор алгоритмов, используемых в тех или иных программных модулях, а также особенности их реализации (включая выбор языка программирования) может существенно повлиять на эффективность ПС. При этом постоянно приходится разрешать противоречие между *временной эффективностью* и *эффективностью по памяти (ресурсам).* Поэтому весьма важно, чтобы в спецификации качества были явно указаны приоритеты или количественное соотношение между показателями этих примитивов качества. Следует также иметь в виду, что разные программные модули по-разному влияют на эффективность ПС в целом: одни модули могут сильно влиять на временную эффективность и практически не влиять на эффективность по памяти, а другие могут существенно влиять на общий расход памяти, не оказывая заметного влияния на время работы ПС. Более того, это влияние (прежде всего, в отношении временной эффективности) заранее (до окончания реализации ПС) далеко не всегда можно правильно оценить.

С учетом сказанного, рекомендуется придерживаться следующих принципов для обеспечения эффективности ПС [12.3, 12.4]:

- сначала нужно разработать надежное ПС, а потом уж заниматься доведением его эффективности до требуемого уровня в соответствии с его спецификацией качества;
- для повышения эффективности ПС, прежде всего, нужно использовать оптимизирующий компилятор - это может обеспечить требуемую эффективность;
- если эффективность ПС не удовлетворяет спецификации его качества, то найдите самые критические модули с точки зрения требуемой эффективности ПС; эти модули и попытайтесь оптимизировать в первую очередь путем их ручной переделки;
- не следует заниматься оптимизацией модуля, если этого не требуется для достижения требуемой эффективности ПС.

Для отыскания критических модулей с точки зрения временной эффективности ПС потребуется получить распределение по модулям времени работы ПС путем соответствующих измерений во время выполнения ПС. Это может быть сделано с помощью динамического анализатора (специального программного инструмента), который может определить частоту обращения к каждому модулю в процессе применения ПС.

12.4. Обеспечение сопровождаемое™ программного средства.

Обеспечение сопровождаемости ПС сводится к обеспечению изучаемости ПС и к обеспечению его модифицируемости.

Изучаемость (подкритерий качества) ПС определяется составом и качеством документации по сопровождению ПС и выражается через такие примитивы качества ПС как *С-*

документированностъ, информативность, понятность, структурированность и *удобочитаемость*. Последние два примитива качества и, в значительной степени, понятность связаны с текстами программных модулей. Вопрос о документации по сопровождению будет обсуждаться в следующей лекции. Здесь мы лишь сделаем некоторые общие рекомендации относительно текстов программ (модулей).

При окончательном оформлении текста программного модуля целесообразно придерживаться следующих рекомендаций, определяющих практически оправданный стиль программирования [12.3, 12.4]:

- используйте в тексте модуля комментарии, проясняющие и объясняющие особенности принимаемых решений; по-возможности, включайте комментарии (хотя бы в краткой форме) на самой ранней стадии разработки текста модуля;
- используйте осмысленные (мнемонические) и устойчиво различимые имена (оптимальная длина имени - 4-12 литер, цифры - в конце), не используйте сходные имена и ключевые слова;
- соблюдайте осторожность в использовании констант (уникальная константа должна иметь единственное вхождение в текст модуля: при ее объявлении или, в крайнем случае, при инициализации переменной в качестве константы);
- не бойтесь использовать необязательные скобки - они обходятся дешевле, чем ошибки;
- размещайте не больше одного оператора в строке; для прояснения структуры модуля используйте дополнительные пробелы (отступы) в начале каждой строки; этим обеспечивается *удобочитаемость* текста модуля;
- избегайте *трюков,* т.е. таких приемов программирования, когда создаются фрагменты модуля, основной эффект которых не очевиден или скрыт (завуалирован), например, побочные эффекты функций.

Структурированность текста модуля существенно упрощает его понимание. Обеспечение этого примитива качества подробно обсуждалось в лекции 8.

Удобочитаемость текста модуля может быть обеспечена автоматически путем применения специального программного инструмента - *форматера.*

Модифицируемость (подкритерий качества) ПС определяется, частично, некоторыми свойствами документации, и свойствами, реализуемые программным путем, и выражается через такие примитивы качества ПС как *расширяемость, модифицируемость, структурированность* и *модульность.*

Расширяемость обеспечивается возможностями автоматически настраиваться на условия применения ПС по информации, задаваемой пользователем. К таким условиям относятся, прежде всего, конфигурация компьютера, на котором будет применяться ПС (в частности, объем и структура его памяти), а также требования конкретного пользователя к функциональным возможностям ПС (например, требования, которые определяют режим применения ПС или конкретизируют структуру информационной среды). К этим возможностям можно отнести и возможность добавления к ПС определенных компонент. Для реализации таких возможностей в ПС часто включается дополнительная компонента (подсистема), называемая *инсталятором.* Инсталятор осуществляет прием от пользователя необходимой информации и настройку ПС по этой информации. Обычно решение о включении в ПС такой компоненты принимается в процессе разработки архитектуры ПС.

Модифицируемость (примитив качества) обеспечивается такими свойствами документации и свойствами, реализуемые программным путем, которые облегчают внесение изменений и доработок в документацию и программы ПС ручным путем (возможно, с определенной компьютерной поддержкой). В спецификации качества могут быть указаны некоторые приоритетные направления и особенности развития ПС. Эти указания должны

быть учтены при разработке архитектуры ПС и модульной структуры его программ. Общая проблема сопровождения ПС - обеспечить, чтобы все его компоненты (на всех уровнях представления) оставались согласованными в каждой новой версии ПС. Этот процесс обычно называют *управлением конфигурацией (configuration management).Чтобы* помочь управлению конфигурацией, необходимо, чтобы связи и зависимости между документами и их частями фиксировать в специальной документации по сопровождению [12.5]. Эта проблема усложняется, если в процессе доработки может находиться сразу несколько версий ПС (в разной степени завершенности).Тогда без компьютерной поддержки довольно трудно обеспечить согласованность документов в разных конфигурациях. Поэтому в таких случаях в ПС включается дополнительная компонента (подсистема), называемая *конфигуратором.* С такой компонентой связывают специальную базу данных (или специальный раздел в базе данных), в которой фиксируются связи и зависимости между документами и их частями для всех версий ПС. Обычно решение о включении в ПС такой компоненты принимается в процессе разработки архитектуры ПС. Для обеспечения этого примитива качества в документацию по сопровождению включают специальное руководство, которое описывает, какие части ПС являются аппаратно- и программно-зависимыми, и как возможное развитие ПС учтено в его строении (конструкции).

Структурированность и *модульность* упрощают ручную модификацию программ ПС.

12.5. Обеспечение мобильности.

Проблема мобильности возникает из-за того, что быстрое развитие компьютерной техники и аппаратных средств делает жизненный цикл многих больших программных средств (программных систем) намного продолжительнее периода «морально» оправданного существования компьютеров и аппаратуры, для которых первоначально создавались эти программные средства. Поэтому обеспечение критерия мобильности для таких ПС является весьма важной задачей.

Мобильность ПС определяется такими примитивами качества ПС как *независимость от устройств, автономность, структурированность* и *модульность.*

Если бы ПС обладало таками примитивами качества, как *независимость от устройств* и *автономность,* и его программы были бы представлены на машинно-независимом языке программирования, то перенос ПС в другую среду обеспечивался бы перетрансляцией (перекомпиляцией) его программ в этой среде. Однако трудно представить реальное ПС, обладающее таким качеством. Тем не менее, таким качеством могут обладать отдельные части программ ПС и даже весьма значительные. А это уже явный намек на то, каким путем следует добиваться мобильности ПС.

Если ПС зависит от устройств (аппаратуры), то в спецификации качества должна быть описана эта компьютерно-аппаратная среда (будем ее называть *аппаратной платформой* [12.6]). Избавится от этой зависимости можно за счет такого примитива качества ПС как автономность. Как правило, ПС строится в рамках некоторой операционной системы (ОС), которая может спрятать специфику аппаратной платформы и, тем самым, сделать ПС независимым от устройств. Но тогда ПС не будет обладать свойством автономности. В этом случае в спецификации качества должна быть описана эта программная среда, над которой строится ПС (будем эту среду называть *операционной платформой* [12.6]). Таким образом, мобильность ПС будет непосредственно связано с мобильностью используемой ОС: перенос ПС на другую аппаратную платформу осуществляется автоматически, если будет осуществлен перенос на эту платформу используемой ОС. Но обеспечение мобильности ОС является самостоятельной и довольно трудной задачей.

Таким образом, для обеспечения мобильности ПС нужно решить две задачи:

- выделение по возможности наибольшей части программ ПС, обладающей свойствами независимости от устройств и автономности (другими словами, независимой от *аппаратно-операционной платформы);*
- обеспечение сопровождаемости для остальных частей программ ПС.

Для решения этих задач целесообразно выбрать в качестве архитектуры ПС слоистую систему (см. рис. 12.1). *Основной слой,* реализующий основные функции ПС, должен быть независимым от аппаратно-операционной платформы. Выделяется также слой (часто называемый *ядром* ПС), который включает программные модули, зависящие от аппаратно-операционной платформы. Этот слой должен обеспечивать, в частности, доступ к внешней информационной среде ПС. Между этими слоями должен быть определен интерфейс, независимый от аппаратно-операционной платформы и обеспечивающий правила обращения из основного слоя к модулям ядра. Будем называть этот интерфейс *системным.* Использование графических пользовательских интерфейсов требует выделение еще одного программного слоя, зависящего от той части аппаратно-операционной платформы *(графической пользовательской платформы),* на которой строятся пользовательские интерфейсы. Будем называть этот слой *оболочкой* ПС. Между оболочкой и основным слоем также должен быть определен интерфейс, независимый от графической пользовательской платформы и обеспечивающий правила обращения из оболочки к модулям основного слоя.

Модульность ПС позволяет сформировать указанные слои, выделяя программные модули с требуемыми свойствами и распределяя их между указанными слоями. *Модульность* и *структурированность* оболочки и ядра позволяют обеспечить эти слои свойством модифицируемости. При этом желательно, чтобы каждый модуль этих слоев был ориентирован на реализацию каких-либо функций управления четко выделенной компоненты аппаратно-операционной среды. Для этого используются такие методы как унификация интерфейсов, стандартизация протоколов и т.п. [12.6].

Упражнения к лекции 12.

12.1. Какие задачи приходиться решать при обеспечении коммуникабельности ПС?

12.2. Какие возможности предоставляет пользователю графический пользовательский интерфейс?

12.3. Как нужно действовать для обеспечения эффективности ПС?

12.4. Что такое инсталятор программного средства (ПС)?

12.5. Что такое управление конфигурацией ПС?

12.6. Что такое ядро ПС?

12. 7. Что такое оболочка ПС?

Лекция 13.

ДОКУМЕНТИРОВАНИЕ ПРОГРАММНЫХ СРЕДСТВ

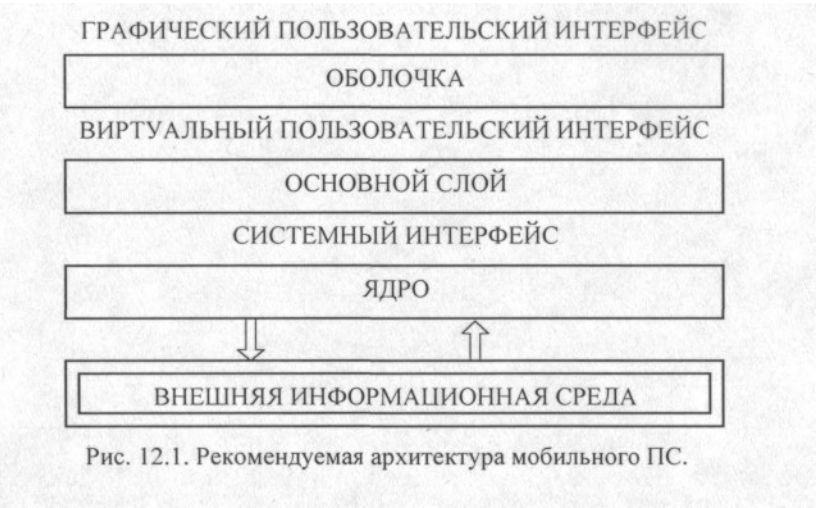


Рис. 12.1. Рекомендуемая архитектура мобильного ПС.

13.1. Документация, создаваемая и используемая в процессе разработки программных средств.

При разработке ПС создается и используется большой объем разнообразной документации. Она необходима как средство передачи информации между разработчиками ПС, как средство управления разработкой ПС и как средство передачи пользователям информации, необходимой для применения и сопровождения ПС. На создание этой документации приходится большая доля стоимости ПС.

Эту документацию можно разбить на две группы [13.1]:

- Документы управления разработкой ПС.
- Документы, входящие в состав ПС.

Документы управления разработкой ПС (software process documentation) управляют и протоколируют процессы разработки и сопровождения ПС, обеспечивая связи внутри коллектива разработчиков ПС и между коллективом разработчиков и *менеджерами ПС (software managers)* - лицами, управляющими разработкой ПС. Эти документы могут быть следующих типов [13.1]:

- *Планы, оценки, расписания.* Эти документы создаются менеджерами для прогнозирования и управления процессами разработки и сопровождения ПС.
- *Отчеты об использовании ресурсов в процессе разработки.* Создаются менеджерами.
- *Стандарты.* Эти документы предписывают разработчикам, каким принципам, правилам, соглашениям они должны следовать в процессе разработки ПС. Эти стандарты могут быть как международными или национальными, так и специально созданными для организации, в которой ведется разработка ПС.
- *Рабочие документы.* Это основные технические документы, обеспечивающие связь между разработчиками. Они содержат фиксацию идей и проблем, возникающих в процессе разработки, описание используемых стратегий и подходов, а также рабочие (временные) версии документов, которые должны войти в ПС.
- *Заметки и переписка.* Эти документы фиксируют различные детали взаимодействия между менеджерами и разработчиками.

Документы, входящие в состав ПС(software product documentation), описывают программы ПС как с точки зрения их применения пользователями, так и с точки зрения их разработчиков и сопроводителей (в соответствии с назначением ПС). Здесь следует отметить, что эти документы будут использоваться не только на стадии эксплуатации ПС (в ее фазах применения и сопровождения), но и на стадии разработки для управления процессом разработки (вместе с рабочими документами) - во всяком случае, они доля быть проверены (протестированы) на соответствие программам ПС. документы образуют два комплекта с разным назначением:

- Пользовательская документация ПС (П-документация).
- Документация по сопровождению ПС (С-документация).

13.2. Пользовательская документация программных средств.

Пользовательская документация ПС (user documentation) объясняет пользователям, как они должны действовать, чтобы примем разрабатываемое ПС [13.1, 13.2.]. Она необходима, если ПС предполагает какое-либо взаимодействие с пользователями. К такой документации относятся документы, которыми должен руководствоваться пользователь *инсталляции* ПС (при установке ПС с соответствующей настройкой на среду применения ПС), при применении ПС для решения своих задач и управлении ПС (например, когда разрабатываемое ПС будет взаимодействовать с другими системами). Эти документы частично затрагивают вопросы сопровождения ПС, но не касаются вопросов, связанных с модификацией программ.

В связи с этим следует различать две категории пользователей ПС: ординарных пользователей ПС и администраторов ПС. *Ординарный пользователь ПС (end-user)* использует ПС для решения своих задач (в своей предметной области). Это может быть инженер, проектирующий техническое устройство, или кассир, продающий железнодорожные билеты с помощью ПС. Он может и не знать многих деталей работы компьютера или принципов программирования. *Администратор ПС(system administrator)* управляет использованием ПС ординарными пользователями и осуществляет сопровождение ПС, не связанное с модификацией программ. Например, он может регулировать права доступа к ПС между ординарными пользователями, поддерживать связь с поставщиками ПС или выполнять определенные действия, чтобы поддерживать ПС в рабочем состоянии, если оно включено как часть в другую систему.

Состав пользовательской документации зависит от аудиторий пользователей, на которые ориентировано разрабатываемое ПС, и от режима использования документов. Под *аудиторией* здесь понимается контингент пользователей ПС, у которого есть необходимость в определенной пользовательской документации ПС [13.2]. Удачный пользовательский до-

кумент существенно зависит от точного определения аудитории, для которой он предназначен. Пользовательская документация должна содержать информацию, необходимую для каждой аудитории. Под *режимом использования* документа понимается способ, определяющий, каким образом используется этот документ. Обычно пользователю достаточно больших программных систем требуются либо документы для изучения ПС (использование в виде *инструкции),* либо для уточнения некоторой информации (использование в виде *справочника).*

В соответствии с работами [13.1, 13.2] можно считать типичным следующий состав пользовательской документации для достаточно больших ПС:

- *Общее функциональное описание ПС.* Дает краткую характеристику функциональных возможностей ПС. Предназначено для пользователей, которые должны решить, насколько необходимо им данное ПС.
- *Руководство по инсталляции ПС.* Предназначено для администраторов ПС. Оно должно детально предписывать, как устанавливать системы в конкретной среде, в частности, должно содержать описание компьютерно-считываемого носителя, на котором поставляется ПС, файлы, представляющие ПС, и требования к минимальной конфигурации аппаратуры.
- *Инструкция по применению ПС.* Предназначена для ординарных пользователей. Содержит необходимую информацию по применению ПС, организованную в форме удобной для ее изучения.
- *Справочник по применению ПС.* Предназначен для ординарных пользователей. Содержит необходимую информацию по применению ПС, организованную в форме удобной для избирательного поиска отдельных деталей.
- *Руководство по управлению ПС.* Предназначено для администраторов ПС. Оно должно описывать сообщения, генерируемые, когда ПС взаимодействует с другими системами, и как должен реагировать администратор на эти сообщения. Кроме того, если ПС использует системную аппаратуру, этот документ может объяснять, как сопровождать эту аппаратуру.

Как уже говорилось ранее (см. лекцию 4), разработка пользовательской документации начинается сразу после создания внешнего описания. Качество этой документации может существенно определять успех ПС. Она должна быть достаточно проста и удобна для пользователя (в противном случае это ПС, вообще, не стоило создавать). Поэтому, хотя черновые варианты (наброски) пользовательских документов создаются основными разработчиками ПС, к созданию их окончательных вариантов часто привлекаются профессиональные технические писатели. Кроме того, для обеспечения качества пользовательской документации разработан ряд стандартов (см. например, [13.2]), в которых предписывается порядок разработки этой документации, формулируются требования к каждому виду пользовательских документов и определяются их структура и содержание.

13.3. Документация по сопровождению программных средств.

Документация по сопровождению ПС (system documentation) описывает ПС с точки зрения ее разработки. Эта документация необходима, если ПС предполагает изучение того, как оно устроена (сконструирована), и модернизацию его программ. Как уже отмечалось, сопровождение - это продолжающаяся разработка. Поэтому в случае необходимости модернизации ПС к этой работе привлекается специальная команда разработчиков-сопроводителей. Этой команде придется иметь дело с такой документацией, которая определяла деятельность команды первоначальных (основных) разработчиков ПС, - с той лишь разницей, что эта документация для команды разработчиков-сопроводителей будет, как правило, чужой (она создавалась другой командой). Чтобы понять строение и процесс разработки модернизируемого ПС, команда разработчиков-сопроводителей должна изучить эту документацию, а затем внести в нее необходимые изменения повторяя в значительной степени технологические процессы, с помощью которых создавалось первоначальное ПС.

Документация по сопровождению ПС можно разбить на две группы:

(1) документация, определяющая строение программ и структур данных ПС и технологию их разработки;

(2) документацию, помогающую вносить изменения в ПС. Документация первой группы содержит итоговые документы каждого технологического этапа разработки ПС. Она включает следующие документы:

• Внешнее описание ПС (Requirement document).

• Описание архитектуры ПС (description of the system architecture), включая внешнюю спецификацию каждой ее программы (подсистемы).

• Для каждой программы ПС - описание ее модульной структуры, включая внешнюю спецификацию каждого включенного в нее модуля.

• Для каждого модуля - его спецификация и описание его строения (design description).

• Тексты модулей на выбранном языке программирования (program source code listings).

• Документы установления достоверности ПС (validation documents), описывающие, как устанавливалась достоверность каждой программы ПС и как информация об установлении достоверности связывалась с требованиями к ПС.

Документы установления достоверности ПС включают, прежде всего, документацию по тестированию (схема тестирования и описание комплекта тестов), но могут включать и результаты других видов проверки ПС, например, доказательства свойств программ. Для обеспечения приемлемого качества этой документации полезно следовать общепринятым рекомендациям и стандартам [13.3 - 13.8].

Документация второй группы содержит

• *Руководство по сопровождению ПС* (system maintenance guide), которое описывает особенности реализации ПС (в частности, трудности, которые пришлось преодолевать) и как учтены возможности развития ПС в его строении (конструкции). В нем также фиксируются, какие части ПС являются аппаратно- и программно-зависимыми.

Общая проблема сопровождения ПС - обеспечить, чтобы все его представления шли в ногу (оставались согласованными), когда ПС изменяется. Чтобы этому помочь, связи и зависимости между документами и их частями должны быть отражены в руководстве по сопровождению, и зафиксированы в базе данных управления конфигурацией.

Упражнения к лекции 13.

13.1. Что такое менеджер программного средства?

13.2. Что такое ординарный пользователь программного средства?

13.3. Что такое администратор программного средства?

13.4. Что такое руководство по инсталляции программного средства?

13.5. Что такое руководство по управлению программным средством?

13.6. Что такое руководство по сопровождению программного средства?

Лекция № 14.

УПРАВЛЕНИЕ РАЗРАБОТКОЙ И АТТЕСТАЦИЯ ПРОГРАММНОГО СРЕДСТВА.

Назначение управления разработкой программного средства и его основные процессы. Структура управления разработкой программных средств. Подходы к организации бригад разработчиков. Управление качеством программного средства. Аттестация программного средства и характеристика методов оценки качества программного средства.

14.1. Назначение и процессы управления разработкой программного средства.

Управление разработкой ПС (software management) - это деятельность, направленная на обеспечение необходимых условий для работы коллектива разработчиков ПС, на планирование и контроль деятельности этого коллектива с целью обеспечения требуемого качества ПС, выполнения сроков и бюджета разработки ПС [14.1, 14.2]. Часто эту деятельность называют также *управлением программным проектом (software project management)*. Здесь под *программным проектом (software project)* понимают всю совокупность работ, связанную с разработкой ПС, а ход выполнения этих работ называют *развитием программного проекта (software project progress)*. К необходимым условиям работы коллектива относятся помещения, аппаратно-программные средства разработки, документация и материально-финансовое обеспечение. Планирование и контроль предполагает разбиение всего процесса разработки

ПС на отдельные конкретные работы (задания), подбор и расстановка исполнителей, установление сроков и порядка выполнения этих работ, оценка качества выполнения каждой работы. Финальной частью этой деятельности является организация и проведения аттестации (сертификации) ПС, которой завершается стадия разработки ПС. Влияние правильной расстановки исполнителей на обеспечение надежности ПС уже обсуждалось в лекции 2. Вопросы документации обсуждались в предыдущей лекции. Другие вопросы управления разработки ПС кратко обсудим в настоящей лекции.

Хотя виды деятельности по управлению разработкой ПС могут быть весьма разнообразными в зависимости от специфики разрабатываемого ПС и организации работ по его созданию, можно выделить некоторые общие процессы (виды деятельности) по управлению разработкой ПС:

- составление плана-проспекта по разработке ПС;
- планирование и составление расписаний по разработке ПС;
- управление издержками по разработке ПС;
- текущий контроль и документирование деятельности коллектива по разработке ПС.
- подбор и оценка персонала коллектива разработчиков ПС.

Составление плана-проспекта по разработке ПС включает формулирование предложений о том, как выполнять разработку ПС. Прежде всего, должно быть зафиксировано, для кого разрабатывается ПС:

- для внешнего заказчика,
- для других подразделений той же организации,
- или является инициативной внутренней разработкой.

В плане-проспекте должны быть установлены общие очертания работ по создания ПС и оценена стоимость разработки, а также предоставляемые для разработки ПС материально-финансовые ресурсы и временные ограничения. Кроме того, он должен включать обоснование, какого рода коллективом должно разрабатываться ПС (специальной организацией, отдельной бригадой и т.п.). И, наконец, должны быть сформулированы необходимые технологические требования (включая, возможно, и выбор подходящей технологии программирования).

Планирование и составление расписаний по разработке ПС - это деятельность, связанная с распределением работ между исполнителями и по времени их выполнения в рамках намеченных сроков и имеющихся ресурсов. Более подробно этот процесс будет рассмотрен в п. 14.3.

Управление издержками по разработке ПС - это деятельность, направленная на обеспечение подходящей стоимости разработки в рамках выделенного бюджета. Она включает оценивание стоимости разработки проекта в целом или отдельных его частей, контроль выполнения бюджета, выбор подходящих вариантов расходования бюджета. Эта деятельность тесно связана с планированием и составлением расписаний в течение всего периода выполнения проекта. Основными источниками издержек являются:

- затраты на аппаратное оборудование (hardware);
- затраты на вербовку и обучение персонала;
- затраты на оплату труда разработчиков.

Текущий контроль и документирование деятельности коллектива по разработке ПС - это непрерывный процесс слежения за ходом развития проекта, сравнения действительных состояния и издержек с запланированными, а также документирования различных аспектов развития проекта (см. лекцию 13). Этот процесс помогает вовремя обнаружить затруднения и предсказать возможные проблемы в развитии проекта.

Подбор и оценка персонала коллектива разработчиков ПС – это деятельность, связанная с формированием коллектива разработчиков ПС. Имеющийся в распоряжении штат разработчиков далеко не всегда будет подходящим по квалификации и опыту работы для данного проекта. Поэтому приходится, частично, вербовать подходящий персонал, а, частично, организовывать дополнительное обучение имеющихся разработчиков. В любом случае в фор-

мируемом коллективе хотя бы один его член должен иметь опыт разработки программных средств (систем), сопоставимых с ПС, который требуется разработать. Это поможет избежать многих простых ошибок в развитии проекта.

14.2. Структура управления разработкой программных средств.

Разработка ПС обычно производится в организации, в которой одновременно могут вестись разработки ряда других программных средств. Для управления всеми этими программными проектами используется иерархическая структура управления. Традиционная структура такого рода обсуждена в работе.

Во главе этой иерархии находится *директор* (или вице-президент) программистской организации, отвечающий за управление всеми разработками программных средств. Ему непосредственно подчинены несколько менеджеров сферы разработок и один менеджер по качеству программных средств. В результате общения с потенциальными заказчиками директор принимает решение о начале выполнения какого-либо программного проекта, поручая его одному из менеджеров сферы разработок, а также решение о прекращение того или иного проекта. Он участвует в обсуждении общих организационных требований (ограничений) к программному проекту и возникающих проблем, решение которых требует использование общих ресурсов программистской организации или изменения заказчиком общих требований.

Менеджер сферы разработок отвечает за управление разработками программных средств (систем) определенного типа, например, программные системы в сфере бизнеса, экспертные системы, программные инструменты и инструментальные системы, поддерживающие процессы разработки программных средств, и другие. Ему непосредственно подчинены менеджеры проектов, относящихся к его сфере. Получив поручение директора по выполнению некоторого проекта, он организует формирование коллектива исполнителей по этому проекту (в частности, необходимую вербовку и обучение персонала). Он участвует в обсуждении плана-проспекта программного проекта, относящегося к сфере разработок, за которую он отвечает, а также в обсуждении и решении возникающих проблем в развитии этого проекта. Он организует обобщение опыта разработок программных средств в его сфере и накопление программных средств и документов для повторного использования.

По каждому программному проекту назначается свой менеджер, который управляет развитием этого проекта. Ему непосредственно подчинены лидеры бригад разработчиков. *Менеджер проекта* осуществляет планирование и составление расписаний работы этих бригад по разработке соответствующего ПС (см. следующий раздел).

Считается крайне нецелесообразным разработка большого ПС (программной системы) одной большой единой бригадой разработчиков. Для этого имеется ряд серьезных причин. В частности, в большой бригаде время, затрачиваемое на общение между ее членами, может быть больше времени, затрачиваемого на собственно разработку. Отрицательное влияние оказывает большая бригада на строение ПС и на интерфейс между отдельными его частями. Все это приводит к снижению надежности ПС. Поэтому обычно большой проект разбивается на несколько относительно независимых подпроектов таким образом, чтобы каждый подпроект мог быть выполнен отдельной небольшой бригадой разработчиков (обычно считается, что в бригаде не должно быть больше 8-10 членов). При этом архитектура ПС должна быть такой, чтобы между программными подсистемами, разрабатываемыми независимыми бригадами, был достаточно простой и хорошо определенный системный интерфейс.

Наиболее употребительны три подхода к организации бригад разработчиков [14.1, 14.3, 14.4]:

- обычные бригады,
- неформальные демократические бригады,
- бригады ведущего программиста.

В *обычной бригаде* старший программист *(лидер бригады)* непосредственно руководит работой младших программистов. Недостатки такой организации непосредственно связаны со спецификой разработки ПС: программисты разрабатывают сильно связанные части про-

граммной подсистемы, сам процесс разработки состоит из многих этапов, каждый из которых требует особенных способностей от программиста, ошибки отдельного программиста могут препятствовать работе других программистов. Успех работы такой бригады достигается в том случае, когда ее руководитель является компетентным программистом, способным предъявлять к членам бригады разумные требования и умеющим поощрять хорошую работу.

В *неформальной демократической бригаде* поручаемая ей работа обсуждается совместно всеми ее членами, а задания между ее членами распределяются согласованно в зависимости от способностей и опыта этих членов. Один из членов этой бригады является *лидером (руководителем) бригады,* но он также выполняет и некоторые задания, распределяемые между членами бригады. Неформальные демократические бригады могут весьма успешно справляться с порученной им работой, если большинство членов бригады являются опытными и компетентными специалистами. Если же неформальная демократическая бригада состоит, в основном, из неопытных и некомпетентных членов, в деятельности бригады могут возникать большие трудности. Без наличия в бригаде хотя бы одного квалифицированного и авторитетного члена, способного координировать и направлять работу членов бригады, эти трудности могут привести к неудаче проекта.

В *бригаде ведущего программиста* за разработку порученной программной подсистемы несет полную ответственность один человек, называемый *ведущим программистом (chief programmer)* и являющийся *лидером бригады:* он сам конструирует эту подсистему, составляет и отлаживает необходимые программы, пишет документацию к подсистеме. Ведущий программист выбирается из числа опытных и одаренных программистов. Все остальные члены такой бригады, в основном, создают условия для наиболее продуктивной работы ведущего программиста. Организацию такой бригады обычно сравнивают с хирургической бригадой [14.1, 14.4]. Ядро бригады ведущего программиста составляют три члена бригады: помимо ведущего программиста в него входит дублер ведущего программиста и администратор базы данных разработки. *Дублер ведущего программиста (backup programmer)* также является квалифицированным и опытным программистом, способным выполнить любую работу ведущего программиста, но сам он эту работу не делает. Главная его обязанность - быть в курсе всего, что делает ведущий программист. Он выступает в роли оппонента ведущего программиста при обсуждении его идей и предложений, но решения по всем обсуждаемым вопросам принимает единолично ведущий программист. *Администратор базы данных разработки (librarian)* отвечает за сопровождение всей документации (включая версии программ), возникающей в процессе разработки программной подсистемы, и снабжает членов бригады информацией о текущем состоянии разработки. Эта работа выполняется с помощью соответствующей инструментальной компьютерной поддержки (см. лекцию 16). В зависимости от объема и характера порученной работы в бригаду могут быть включены дополнительные члены, такие как

- распорядитель бригады, выполняющий административные функции;
- технический редактор, осуществляющий доработку и техническое редактирование документов, написанных ведущим программистом;
- инструментальщик, отвечающий за подбор и функционирование программных средств, поддерживающих разработку программной подсистемы;
- тестовик, готовящий подходящий набор тестов для отладки разрабатываемой программной подсистемы;
- один или несколько младших программистов, осуществляющих кодирование отдельных программных компонент по спецификациям, разработанным ведущим программистом.

Кроме того, к работе бригады может привлекаться для консультации эксперт по языку программированию.

Важное место в управлении разработкой ПС отводится управление обеспечением качества. Для руководства этой деятельностью назначается специальный менеджер, подчиненный непосредственно директору, - *менеджер по качеству.* Ему непосредственно подчинены

формируемые бригады по контролю качества. Эти бригады работают с отдельными проектами, но непосредственно соответствующим менеджерам проектов не подчинены, сохраняя тем самым свою независимость от них.

Управление обеспечением качества означает контроль качества каждой работы, выполняемой разработчиками в рамках программного проекта, контроль каждого документа, включаемого в ПС. Качество ПС не может быть добавлено к ПС после того, как оно будет уже создано. Качество ПС формируется постепенно в процессе всей разработки ПС, в каждой отдельной работе, выполняемой по программному проекту. Поэтому для каждой такой работы, прежде чем она получит одобрение и будет считаться завершенной, организуется *смотр (review)* соответствующей бригадой по контролю качества. Этот смотр существенно отличается от контроля, осуществляемого разработчиками в конце каждого этапа разработки, так как последний является техническим процессом, связанным с обнаружением ошибок, тогда как смотр по контролю качества является функцией управления разработкой и связан с оценкой того, насколько результаты этой работы согласуются с декларированными требованиями относительно качества ПС.

Существенную роль в управлении качеством ПС играют программные (софтверные) стандарты [14.1, 14.2]. Они фиксируют удачный опыт высоко квалифицированных специалистов по разработке ПС для различных их классов и для разных моделей их качества. Следование подходящим стандартам может существенно облегчить достижение поставленных целей относительно качества ПС, а также упростить смотр по контролю качества. Кроме того, стандарты способствуют формированию взаимопонимания внутри коллектива разработчиков и упрощают процесс обучения новых членов этого коллектива.

Различают два вида таких стандартов:

- стандарты ПС (программного продукта),
- стандарты процесса создания и использования ПС.

Стандарты ПС определяют некоторые свойства, которыми должны обладать программы или документы ПС, т.е. определяют в какой-то степени качество ПС. При спецификации качества (см. лекцию 4) для конкретизации какого-либо примитива качества иногда достаточно указать, какому стандарту он должен соответствовать, в других случаях привязка примитива качества к стандарту может потребовать лишь незначительной дополнительной конкретизации этого примитива. Привязка примитивов качества к тем или иным стандартам сильно упрощает контроль и оценку качества ПС. К стандартам ПС относятся, прежде всего, стандарты на языки программирования, на состав документации, на структуру различных документов, на различные форматы и другие.

Стандарты процесса создания и использования ПС определяют, как должен проводится этот процесс, т.е. подход к разработке ПС, структуру жизненного цикла ПС и его технологические процессы. Хотя эти стандарты непосредственно не определяют качества ПС, однако считается, что качество ПС существенно зависит от качества процесса его разработки. Эти стандарты проще контролировать, поэтому повсеместно используются для управления качеством ПС.

В лекции 13 уже отмечалось, что эти стандарты могут быть как международными или национальными, так и специально созданными для организации, в которой ведется разработка ПС. Разработка последних стандартов является одной из функций управления обеспечением качества ПС.

Бригада по контролю качества состоит из ассистентов (рецензентов) по качеству ПС. Она проводит смотры тех или иных частей ПС или всего ПС в целом с целью поиска возникающих проблем в процессе его разработки. Смотру подлежат все программные компоненты и документы, включаемые в ПС, а также процессы их разработки. В процессе смотра учитываются требования, сформулированные в спецификации качества ПС, в частности, проверяется соответствие исследуемого документа или технологического процесса стандартам, указанным в этой спецификации. В результате смотра формулируются замечания, которые могут фиксироваться письменно или просто передаваться разработчикам устно.

Для смотра каждой конкретной программной компоненты или документа ПС создается комиссия (группа) во главе с *председателем {chairman),* который отвечает за организацию смотра. Он должен иметь достаточный опыт конструирования ПС, чтобы быть готовым принять ответственность за важные технические решения. В эту комиссию включаются два или три ассистента по качеству ПС, один из которых должен быть ответственным за запись решений, сделанных в течение смотра. К смотру обычно привлекаются разработчик исследуемой компоненты или исследуемого документа ПС, а также новые члены коллектива разработчиков в целях их обучения.

14.3. Планирование и составление расписаний по разработке ПС.

Общее представление об этой деятельности можно составить по **ее** описанию на псевдокоде (см. лекцию 8), приведенном на рис. 14.1. Это описание показывает, что планирование и составление расписаний по разработке ПС представляет собой итеративный процесс, который заканчивается только после прекращения работ по самому программному проекту.

В начале этого описания оцениваются общий срок разработки ПС, используемые штаты исполнителей, предельный бюджет и другие ограничения (условия) разработки. С учетом этого фиксируются начальные параметры проекта (его структура и распределение функций). Должны быть также определены «вехи развития проекта» и их сроки. *Веха развития проекта (project progress milestone)* - это конечная точка некоторого этапа или процесса, с которой связывается выдача некоторого промежуточного продукта, представляющего собой некоторый четко определенный документ. Вехи развития проекта обеспечивают возможность контроля развития проекта и возможность модификации расписаний проекта.

```
Определить ограничения, с которыми проект
должен быть доведен до конца.
Сделать начальную оценку параметров проекта.
Установить вехи развития проекта и их сроки.
ПОКА проект не является завершенным или
        прекращенным (аннулированным)
ДЕЛАТЬ
        Составить расписание проекта.
        Инициировать процессы, соответствующие
        расписанию.
        ПОДОЖДАТЬ.
        Просмотреть развитие проекта.
        Скорректировать параметры проекта.
        Оценить влияние изменения параметров
        проекта на расписание проекта.
        Уточнить ограничения и сроки.
        ЕСЛИ возникли проблемы ТО
                Инициировать технический пере-
                смотр и возможную ревизию проекта.
        ВСЕ ЕСЛИ
ВСЕ ПОКА
```

Рис. 14.1. Описание на псевдокоде процесса планирования и состав расписаний по разработке ПС.

Далее начинается итерационный процесс, основу которого составляет повторяющиеся составления расписаний. Составление расписания заключается:

- в разделении всей работы, необходимой для выполнения проекта, на отдельные самостоятельно выполняемые задания;
- в составлении сетевого графика выполнения заданий;
- в составлении гистограммы выполнения заданий;
- в расстановке исполнителей заданий.

При выделении самостоятельных заданий для каждого из них оценивается время его выполнения и его зависимость от других заданий с точки зрения порядка выполнения. *Сетевой график* представляет собой схему (сеть) путей выполнения заданий с указанием времени выполнения каждого задания и с расстановкой вех развития проекта. В сетевом графике должен быть определен *критический путь,* представляющий собой такой путь заданий, суммарное время выполнения которых является наибольшим. *Гистограмма выполнения заданий (activity bar chart)* содержит для каждого задания свою временную полосу от момента, когда выполнение этого задания может быть начато, и до момента, когда выполнение этого задания должно быть закончено. В такой полосе фиксируется как продолжительность выполнения самого задания, так и возможный запас времени для завершения его выполнения. Это дает возможность модифицировать план развития проекта в определенных рамках без изменения общих сроков выполнения проекта. При расстановке исполнителей оценивается для каждого исполнителя соответствие его квалификации и опыта характеру предлагаемой работы. Особое внимание уделяется расстановке исполнителей заданий, находящихся на критическом пути.

Спустя некоторое время (обычно 2-3 недели) после активизации процессов, указанных в расписании, производится обозрение (просмотр) хода развития проекта и отмечаются возникшие противоречия. С учетом этого производится пересмотр (уточнение) параметров проекта и оценивается влияние измененных параметров на расписание проекта. Если окажется, что эти изменения увеличивают время разработки ПС, необходимо обсудить с заказчиком возможность изменения ограничений проекта и срока его завершения. В том случае, когда заказчик не может пойти на подходящие изменения, производится технический пересмотр проекта с целью поиска альтернативных подходов к разработке ПС.

14.4 Аттестации программного средства.

Завершающим этапом разработки ПС является аттестация ПС, подводящая итог всей разработке. *Аттестация (certification)* ПС - это авторитетное подтверждение качества ПС [14.5, 14.6]. Обычно для аттестации ПС создается аттестационная комиссия из экспертов, представителей заказчика и представителей разработчика. Эта комиссия проводит *приемосдаточные испытания* ПС с целью получения необходимой информации для оценки его качества. Под *испытанием* ПС здесь понимают [14.6, 14.7] процесс проведения комплекса мероприятий, исследующих пригодность ПС для успешной его эксплуатации (применения и сопровождения) в соответствии с требованиями заказчика. В этом процессе проверяется полнота и исследуется качество представленной программной документации, производится необходимое тестирование программ, входящих в состав ПС, а также исследуются и другие свойства ПС, декларированные в его спецификации качества. На основе полученной информации комиссия должна установить, в какой степени ПС выполняет декларированные функции и в какой степени ПС обладает декларированными примитивами и критериями качества. Решение аттестационной комиссии о произведенной оценке качества ПС фиксируется в соответствующем документе (сертификате), который подписывается членами комиссии.

Таким образом, оценка качества ПС является основным содержанием процесса аттестации. Прежде всего, следует отметить, что оценка качества ПС производится по предъявленной спецификации его качества, т.е. оценивается только декларированное разработчиками качество ПС. При этом оценка качества ПС по каждому из критериев сводится к оценке каждого из примитивов качества, связанному с этим критерием качества ПС, в соответствии с их конкретизацией в спецификации качества этого ПС (см. лекцию 4). Различают следующие группы методов оценки примитивов качества ПС:

- непосредственное измерение показателей примитива качества;
- тестирование программ ПС;
- экспертная оценка на основании изучения программ и документации ПС

Непосредственное измерение показателей примитива качества производится путем проверки соответствия предъявленной документации (включая тексты программ на языке программирования) стандартам или явным требованиям, указанным в спецификации качества ПС, а также путем измерения времени работы различных устройств и используемых ресурсов при выполнении контрольных (тестовых) задач. Например, некоторым показателем эффективности по памяти может быть число строк программы на языке программирования, а некоторым показателем временной эффективности может быть время ответа на запрос пользователя.

Для оценки некоторых примитивов качества ПС используется *тестирование* [14.5-14.8]. К таким примитивам относятся, прежде всего завершенность ПС, а также его точность, устойчивость, защищенность и другие примитивы качества. Этот вопрос уже обсуждался в лекции 10. Однако во время приемо-сдаточных испытаний нет необходимости проведения тестирование ПС в полном объеме (это может слишком дорого стоить). Аттестационная комиссия должна, прежде всего, изучить предъявленную документацию по проведенному разработчиками тестированию ПС и лишь выборочно пропустить предъявленные тесты. Дополнительные тесты составляются, если у комиссии возникают определенные сомнения в полноте тестирования, проведенного разработчиками. Кроме того, обычно работоспособность и некоторые показатели качества ПС демонстрируются на решении контрольных задач, предъявляемых заказчиком.

В некоторых случаях для оценки качества ПС проводятся дополнительные полевые или промышленные испытания [14.8, 14.9]. *Полевые* испытания ПС – это демонстрация ПС вместе с технической системой, которой управляет эта ПС, с обеспечением тщательного наблюдения за поведением ПС. Заказчикам должна быть предоставлена возможность задания собственных контрольных примеров, в частности, с выходом в критические режимы работы технической системы, а также с вызовом в ней аварийных ситуаций. *Промышленные* испытания ПС - это процесс передачи ПС в постоянную эксплуатацию пользователям, представляющий собой опытную эксплуатацию ПС (см. лекцию 10) пользователями со сбором информации об особенностях поведения ПС и ее эксплуатационных характеристиках.

Многие примитивы качества ПС трудно уловимы с точки зрения их (объективной) оценки. В этих случаях иногда применяют *метод экспертных оценок.* Этот метод заключается в следующем. Назначается группа экспертов и каждый из этих экспертов в результате изучения представленной документации составляет свое мнение об обладании ПС требуемым примитивом качества. Затем голосованием членов этой группы устанавливается оценка требуемого примитива качества ПС, т.е. получаемая оценка является некоторым усреднением совокупности субъективных оценок. Эта оценка может производиться как по двухбалльной системе ("обладает" - "не обладает"), так и учитывать степень обладания ПС этим примитивом качества (например, производиться по пятибальной системе) в соответствии с требованиями относительно этого примитива, сформулированными в спецификации качества аттестуемого ПС.

Аттестация ПС похожа на смотр различных компонент ПС в процессе управления качеством ПС, однако, имеются и существенные различия [14.1]. Во-первых, смотр проводится менее представительной группой специалистов. Во-вторых, в процессе смотра не производится полной оценки качества ПС, а выявляются лишь отдельные просчеты и нарушения требований относительно качества ПС, связанные с обозреваемой компонентой (документом), при этом не требуется немедленного устранения выявленных недостатков, если они не мешают проведения последующих работ. *Целью* же аттестации является проверка и фиксация реальных показателей качества предъявленного ПС [14.7]. Если аттестационная комиссия подтверждает, что предъявленное ПС соответствует всем требованиям относительно его качества, сформулированным во внешнем описании этого ПС, то считается, что его разра-

ботка завершена успешно и заказчик обязан принять это ПС. Если же будут обнаружены отступления от этих требований, то должны приниматься определенные решения о продолжении или прекращении разработки предъявленного ПС, но это уже вопрос взаимоотношений между заказчиком и разработчиками. Таким образом, аттестационная комиссия, подписывая документ о произведенной оценке качества ПС, принимает на себя определенную ответственность за гарантию качества этого ПС. Но здесь имеются определенные правовые проблемы, обсуждение которых выходит за рамки темы этой лекции.

Упражнения к лекции 14.

14.1. Что такое управление разработкой ПС?

14.2. Что такое менеджер программного проекта?

14.3. Что такое неформальная демократическая бригада разработчиков ПС?

14.4. Что такое бригада ведущего программиста?

14.5. Что такое смотр программной компоненты (программного документа)?

14.6. Что такое аттестация ПС?

Лекция № 15.

КОМПЬЮТЕРНАЯ ПОДДЕРЖКА РАЗРАБОТКИ И СОПРОВОЖДЕНИЯ ПРОГРАММНЫХ СРЕДСТВ

Программные инструменты в жизненном цикле программных средств. Инструментальные среды и инструментальные системы поддержки разработки программных средств, их классификация. Компьютерная технология (CASE-технология) разработки программных средств и ее рабочие места. Общая архитектура инструментальных систем технологии программирования

16.1. Инструменты разработки программных средств.

При разработке программных средств используется в той или иной мере компьютерная поддержка процессов разработки **и** сопровождения ПС [16.1]. Это достигается путем представления хотя бы некоторых программных документов ПС (прежде всего, программ) на компьютерных носителях данных (например, на дискетах) и предоставлению в распоряжение разработчика ПС специальных ПС или включенных в состав компьютера специальных устройств, созданных для какой-либо обработки таких документов. В качестве такого специального ПС можно указать компилятор с какого-либо языка программирования. Компилятор избавляет разработчика ПС от необходимости писать программы на языке компьютера, который для разработчика ПС был бы крайне неудобен, - вместо этого он составляет программы на удобном ему языке программирования, которые соответствующий компилятор автоматически переводит на язык компьютера. В качестве специального устройства, поддерживающего процесс разработки ПС, можно указать, например, эмулятор какого-либо языка. Эмулятор позволяет выполнять (интерпретировать) программы на языке, отличном от языка компьютера, поддерживающего разработку ПС, например, на языке компьютера, для которого эта программа предназначена.

ПС, предназначенное для поддержки разработки других ПС, будем называть *программным инструментом* разработки ПС, а устройство компьютера, специально предназначенное для поддержки разработки ПС, будем называть *аппаратным инструментом* разработки ПС.

Инструменты разработки ПС могут использоваться в течение всего жизненного цикла ПС [16.2] для работы с разными программными документами. Так текстовый редактор может использоваться для разработки практически любого программного документа. С точки зрения функций, которые инструменты выполняют при разработке ПС, их можно разбить на следующие четыре группы:

- редакторы,
- анализаторы,
- преобразователи,
- инструменты, поддерживающие **процесс выполнения** программ.

Редакторы поддерживают конструирование (формирование) тех или иных программных документов на различных этапах жизненного цикла. Как уже упоминалось, для этого

можно использовать один какой-нибудь универсальный текстовый редактор. Однако, более сильную поддержку могут обеспечить специализированные редакторы: для каждого вида документов - свой редактор. В частности, **на** ранних этапах разработки в документах могут широко использоваться графические средства описания (диаграммы, схемы и т.п.). В таких случаях весьма полезными могут быть графические редакторы. На этапе программирования (кодирования) вместо текстового редактора может оказаться более удобным синтаксически управляемый редактор, ориентированный на используемый язык программирования.

Анализаторы производят либо *статическую* обработку документов, осуществляя различные виды их контроля, выявление определенных их свойства накопление статистических данных (например, проверку соответствия документов указанным стандартам), либо *динамический* анализ программ (например, с целью выявление распределения времени работы программы по программным модулям).

Преобразователи позволяют автоматически приводить документы к другой форме представления (например, форматеры) или переводить документ одного вида к документу другого вида (например, конверторы или компиляторы), синтезировать какой-либо документ из отдельных частей и т.п.

Инструменты, поддерживающие процесс выполнения программ, позволяют выполнять на компьютере описания процессов или отдельных их частей, представленных в виде, отличном от машинного кода, или машинный код с дополнительными возможностями его интерпретации. Примером такого инструмента является эмулятор кода другого компьютера. К этой группе инструментов следует отнести и различные отладчики. По существу, каждая система программирования содержит программную подсистему периода выполнения, которая выполняет программные фрагменты, наиболее типичные для языка программирования, и обеспечивает стандартную реакцию на возникающие при выполнении программ исключительные ситуации (такую подсистему мы будем называть *исполнительной поддержкой).* Такую подсистему также можно рассматривать как инструмент данной группы.

16.2. Инструментальные среды разработки и сопровождения программных средств и принципы их классификации.

Компьютерная поддержка процессов разработки и сопровождения ПС может производиться не только за счет использования отдельных инструментов (например, компилятора), но **и** за счет использования некоторой логически связанной совокупности программных **и** аппаратных инструментов. Такую совокупность будем называть *инструментальной средой разработки и сопровождения ПС.*

Часто разработка ПС производится на том же компьютере, на котором оно будет применяться. Это достаточно удобно. Во-первых, в этом случае разработчик имеет дело только с компьютерами одного типа. А, во-вторых, в разрабатываемое ПС могут включаться компоненты самой инструментальной среды. Однако, это не всегда возможно. Например, компьютер, на котором должно применяться ПС, может быть неудобен для поддержки разработки ПС или его мощность недостаточна для обеспечения функционирования требуемой инструментальной среды. Кроме того, такой компьютер может быть недоступен для разработчиков этого ПС (например, он постоянно занят другой работой, которую нельзя прерывать, или он находится еще в стадии разработки). В таких случаях применяется так называемый *инструментально-объектный подход* [16.1]. Сущность его заключается в том, что ПС разрабатывается на одном компьютере, называемым *инструментальным,* а применяться будет на другом компьютере, называемым *целевым* (или *объектным).*

Инструментальная среда не обязательно должна функционировать на том компьютере, на котором должно будет применяться разрабатываемое с помощью ее ПС.

Совокупность инструментальных сред можно разбивать на разные классы, которые различаются значением следующих признаков:

- ориентированность на конкретный язык программирования,
- специализированность,
- комплексность,

- ориентированность на конкретную технологию программирования,
- ориентированность на коллективную разработку,
- интегрированность.

Ориентированность на конкретный язык программирования (языковая ориентированность) показывает: ориентирована ли среда на какой-либо конкретный язык программирования (и на какой именно) или может поддерживать программирование на разных языках программирования. В первом случае информационная среда и инструменты существенно используют знание о фиксированном языке *(глобальная ориентированность),* в силу чего они оказываются более удобным для использования или предоставляют дополнительные возможности при разработке ПС. Но в этом случае такая среда оказывается не пригодной для разработки программ на другом языке. Во втором случае инструментальная среда поддерживает лишь самые общие операции и, тем самым, обеспечивает не очень сильную поддержку разработки программ, но обладает свойством *расширения (открытости).* Последнее означает, что в эту среду могут быть добавлены отдельные инструменты, ориентированные на тот или иной конкретный язык программирования, но эта ориентированность будет лишь *локальной* (в рамках лишь отдельного инструмента).

Специализированность инструментальной среды показывает: ориентирована ли среда на какую-либо предметную область или нет. В первом случае информационная среда и инструменты существенно используют знание о фиксированной предметной области, в силу чего они оказываются более удобными для использования или предоставляют дополнительные возможности при разработке ПС для этой предметной области. Но в этом случае такая инструментальная среда оказывается не пригодной или мало пригодной для разработки ПС для других предметных областей. Во втором случае среда поддерживает лишь самые общие операции для разных предметных областей. Но в этом случае такая среда будет менее удобной для конкретной предметной области, чем специализированная на эту предметную область.

Комплексность инструментальной среды показывает: поддерживает ли она все процессы разработки и сопровождения ПС или нет. В первом случае продукция этих процессов должна быть согласована. Поддержка инструментальной средой фазы сопровождения ПС, означает, что она должна поддерживать работу сразу с несколькими вариантами ПС, ориентированными на разные условия применения ПС и на разную связанную с ним аппаратуру, т.е. должна обеспечивать *управление конфигурацией ПС* [16.1, 16.3].

Ориентированность на конкретную технологию программирования показывает: ориентирована ли инструментальная среда на фиксированную технологию программирования [16.2] либо нет. В первом случае структура и содержание информационной среды, а также набор инструментов существенно зависит от выбранной технологии *(технологическая определенность).* Во втором случае инструментальная среда поддерживает самые общие операции разработки ПС, не зависящие от выбранной технологии программирования.

Ориентированность на коллективную разработку показывает: поддерживает ли среда управление (management) работой коллектива или нет. В первом случае она обеспечивает для разных членов этого коллектива разные права доступа к различным фрагментам продукции технологических процессов и поддерживает работу *менеджеров* [16.1] по управлению коллективом разработчиков. Во втором случае она ориентирована на поддержку лишь отдельных пользователей.

Интегрированность инструментальной среды показывает: является ли она интегрированной (и в каком смысле) или нет. Инструментальная среда считается *интегрированной,* если взаимодействие пользователя с инструментами подчиняется единообразным правилам, а сами инструменты действуют по заранее заданной информационной схеме, связаны по управлению или имеют общие части. В соответствие с этим различают три *вида интегрированности:*

- интегрированность по пользовательскому интерфейсу,
- интегрированность по данным,

- интегрированность по действиям (функциям),

Интегрированность по пользовательскому интерфейсу означает, что все инструменты объединены единым пользовательским интерфейсом.

Интегрированность по данным означает, что инструменты действуют в соответствии с фиксированной информационной схемой (моделью) системы, определяющей зависимость друг от друга различных используемых в системе фрагментов данных (информационных объектов). В этом случае может быть обеспечен контроль полноты и актуальности программных документов и порядка их разработки. *Интегрированность по действиям* означает, что, во-первых, в системе имеются общие части всех инструментов и, во-вторых, одни инструменты при выполнении своих функций могут обращаться к другим инструментам.

Инструментальную среду, интегрированную хотя бы по данным или по действиям, будем называть *инструментальной системой.* При этом интегрированность по данным предполагает наличие в системе специализированной базы данных, называемой *репозиторием.* Под *репозиторием* будем понимать центральное компьютерное хранилище информации, связанной с проектом (разработкой) ПС в течение всего **его** жизненного цикла [16.1].

16.3. Основные классы инструментальных сред разработки и сопровождения программных средств.

В настоящее время выделяют [16.1] три основных класса инструментальных сред разработки и сопровождения ПС (рис. 16.1):

- инструментальные среды программирования,
- рабочие места компьютерной технологии,
- инструментальные системы технологии программирования.

Рис.16.1. Основные классы инструментальных сред разработки и сопровождения ПС.

Инструментальная среда программирования предназначена в основном для поддержки процессов программирования (кодирования), тестирования и отладки ПС. Она не обладает рассмотренными выше свойствами комплексности, ориентированности на конкретную технологию программирования, ориентированности на коллективную разработку и, как правило, свойством интегрированности, хотя имеется некоторая тенденция к созданию интегрированных сред программирования (в этом случае их следовало бы называть *системами программирования).* Иногда среда программирования может обладать свойством специализированности. Признак же ориентированности на конкретный язык программирования может иметь разные значения, что существенно используется для дальнейшей классификации сред программирования.

Рабочее место компьютерной технологии ориентировано на поддержку ранних этапов разработки ПС (системного анализа и спецификаций) и автоматической генерации программ по спецификациям. Оно существенно использует свойства специализированности, ориентированости на конкретную технологию программирования и, как правило, интегрированности. Более поздние рабочие места компьютерной технологии обладают также свойством комплексности. Что же касается языковой ориентированности, то вместо языков программирования они ориентированы на те или иные формальные языки спецификаций. Свойством

ориентированности на коллективную разработку указанные рабочие места в настоящее время, как правило, не обладают.

Инструментальная система технологии программирования предназначена для поддержки всех процессов разработки и сопровождения в течение всего жизненного цикла ПС и ориентирована на коллективную разработку больших программных систем с продолжительным жизненным циклом. Обязательными свойствами ее являются комплекенсность, ориентированность на коллективную разработку и интегрированность. Кроме того, она или обладает технологической определенностью или получает это свойство в процессе расширения (настройки). Значение признака языковой ориентированности может быть различным, что используется для дальнейшей классификации этих систем.

16.3. Инструментальные среды программирования.

Инструментальная среда программирования включает, прежде всего текстовый редактор, позволяющий конструировать программы на заданном языке программирования, а также инструменты, позволяющие компилировать или интерпретировать программы на этом языке, тестировать и отлаживать полученные программы. Кроме того, могут быть и другие инструменты, например, для статического или динамического анализа программ. Взаимодействуют эти инструменты между собой через обычные файлы с помощью стандартных возможностей файловой системы.

Различают следующие классы инструментальных сред программирования (см. рис. 16.2):

- среды общего назначения,
- языково-ориентированные среды.

Инструментальные среды программирования *общего назначения* содержат набор программных инструментов, поддерживающих разработку программ на разных языках программирования (например, текстовый редактор, редактор связей или интерпретатор языка целевого компьютера) и обычно представляют собой некоторое расширение возможностей используемой операционной системы. Для программирования в такой среде на каком-либо языке программирования потребуются дополнительные инструменты, ориентированные на этот язык (например, компилятор).

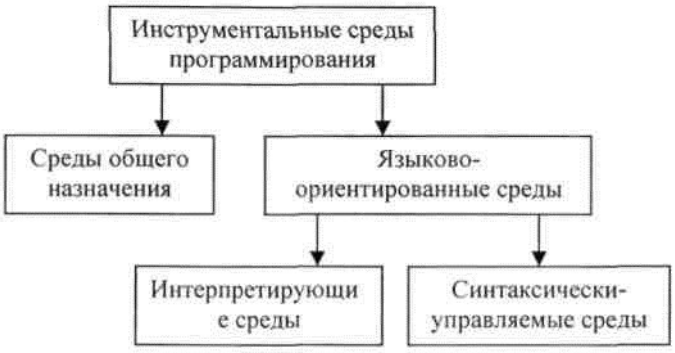


Рис. 16.2. Классификация инструментальных сред программирования.

Языково-ориентированная инструментальная среда программирования предназначена для поддержки разработки ПС на каком-либо одном языке программирования и знания об этом языке существенно использовались при построении такой среды. Вследствие этого в такой среде могут быть доступны достаточно мощные возможности, учитывающие специфику данного языка. Такие среды разделяются на два подкласса:

- интерпретирующие среды,
- синтаксически-управляемые среды.

Интерпретирующая инструментальная среда программирования обеспечивает интерпретацию программ на данном языке программирования, т.е. содержит, прежде всего, интерпретатор языка программирования, на который эта среда ориентирована. Такая среда необходима для языков программирования интерпретирующего типа (таких, как Лисп), но может использоваться и для других языков (например, на инструментальном компьютере). *Синтак-*

сически-управляемая инструментальная среда программирования базируется на знании синтаксиса языка программирования, на который она ориентирована. В такой среде вместо текстового используется синтаксически-управляемый редактор, позволяющий пользователю использовать различные шаблоны синтаксических конструкций (в результате этого разрабатываемая программа всегда будет синтаксически правильной). Одновременно с программой такой редактор формирует (в памяти компьютера) ее синтаксическое дерево, которое может использоваться другими инструментами.

16.4. Понятие компьютерной технологии разработки программных средств и ее рабочие места.

Имеются некоторые трудности в выработке строгого определения CASE-технологии (компьютерной технологии разработки ПС). CASE - это аббревиатура от английского Computer-Aided Software Engineering (Компьютерно-Помогаемая Инженерия Программирования). Но без помощи (поддержки) компьютера ПС уже давно не разрабатываются (используется хотя бы компилятор). В действительности, в это понятие вкладывается более узкий (специальный) смысл, который постепенно размывается (как это всегда бывает, когда какое-либо понятие не имеет строгого определения). Первоначально под CASE понималась [16.4] инженерия ранних этапов разработки ПС (определение требований, разработка внешнего описания и архитектуры ПС) с использованием программной поддержки (программных инструментов). Теперь под CASE может пониматься [16.4] и инженерия всего жизненного цикла ПС (включая и его сопровождение), но только в том случае, когда программы частично или полностью генерируются по документам, полученным на указанных ранних этапах разработки. В этом случае CASE-технология стала принципиально отличаться от ручной (традиционной) технологии разработки ПС: изменилось не только содержание технологических процессов, но и сама их совокупность.

В настоящее время *компьютерную технологию* разработки ПС можно характеризовать [16.1] использованием

- программной поддержки для разработки графических требований графических спецификаций ПС,
- автоматической генерации программ на каком-либо языке программирования или в машинном коде (частично или полностью),
- программной поддержки прототипирования.

Говорят также, что компьютерная технология разработки ПС является "безбумажной", т.е. рассчитанной на компьютерное представление программных документов. Однако, уверенно отличить ручную технологию разработки ПС от компьютерной по этим признакам довольно трудно. Значит, самое существенное в компьютерной технологии не выделено.

На наш взгляд, главное отличие ручной технологии разработки ПС от компьютерной заключается в следующем. Ручная технология ориентирована на разработку документов, одинаково понимаемых разными разработчиками ПС, тогда как компьютерная технология ориентирована на обеспечение семантического понимания (интерпретации) документов программной поддержкой компьютерной технологии. Семантическое понимание документов дает программной поддержке возможность автоматически генерировать программы. В связи с этим существенной частью компьютерной технологии становится использование формальных языков уже на ранних этапах разработки ПС: как для спецификации программ, так и для спецификации других документов. В частности, широко используются формальные графические языки спецификаций. Именно это позволяет рационально изменить и саму совокупность технологических процессов разработки и сопровождения ПС.

Из проведенного обсуждения можно определить *компьютерную технологию разработки ПС* как технологию программирования, в которой используются программные инструменты для разработки формализованных спецификаций программ и других документов (включая и графические спецификации) с последующей автоматической генерацией программ и документов (или хотя бы значительной их части) по этим спецификациям.

Теперь становятся понятными и основные изменения в жизненном цикле ПС для компьютерной технологии. Если при использовании ручной технологии основные усилия по разработке ПС делались на этапах собственно программирования (кодирования) и отладки (тестирования), то при использовании компьютерной технологии - на ранних этапах разработки ПС (определения требований и функциональной спецификации, разработки архитектуры). При этом существенно изменился характер документации. Вместо целой цепочки неформальных документов, ориентированной на передачу информации от заказчика (пользователя) к различным категориям разработчикам, формируются прототип ПС, поддерживающий выбранный пользовательский интерфейс, и формальные функциональные спецификации (иногда и формальные спецификации архитектуры ПС), достаточные для автоматического синтеза (генерации) программ ПС (или хотя бы значительной их части). При этом появилась возможность автоматической генерации части документации, необходимой разработчикам и пользователям. Вместо ручного программирования (кодирования) - автоматическая генерация программ, что делает не нужной автономную отладку и тестирование программ: вместо нее добавляется достаточно глубокий автоматический семантический контроль документации. Появляется возможность автоматической генерации тестов по формальным спецификациям для комплексной *{системной)* отладки ПС. Существенно изменяется и характер сопровождения ПС: все изменения разработчиком-сопроводителем вносятся только в спецификации (включая и прототип), остальные изменения в ПС осуществляются автоматически.

С учетом сказанного *жизненный цикл ПС для компьютерной технологии* можно представить [16.4] следующей схемой (рис. 16.3).

Рис. 16.3. Жизненный цикл программного средства для компьютерной технологии.

Прототипирование ПС является необязательным этапом жизненного цикла ПС при компьютерной технологии, что на рис. 16.3 показано пунктирной стрелкой. Однако использование этого этапа во многих случаях и соответствующая компьютерная поддержка этого этапа является характерной для компьютерной технологии. В некоторых случаях прототипирование делается после (или в процессе) разработки спецификаций ПС, например, в случае прототипирования пользовательского интерфейса. Это показано на рис. 16.3 пунктирной возвратной стрелки. Хотя возврат к предыдущим этап мы допускаем на любом этапе, но здесь это показано явно, так как прототипирование является особым подходом к разработке ПС (см. лекцию 3). Прототипирование пользовательского интерфейса позволяет заменить

косвенное описание взаимодействия между пользователем и ПС при ручной технологии (при разработке внешнего описания ПС) прямым выбором пользователем способа и стиля этого взаимодействия с фиксацией всех необходимых деталей. По существу, на этом этапе производится точное описание пользовательского интерфейса, понятное программной поддержке компьютерной технологии, причем с ответственным участием пользователя. Все это базируется на наличие в программной поддержке компьютерной

технологии настраиваемой оболочки с обширной библиотекой заготовок различных фрагментов и деталей экрана. Такое прототипирование, по-видимому, является лучшим способом преодоления барьера между пользователем и разработчиком.

Разработка спецификаций ПС распадается на несколько разных процессов. Если исключить начальный этап разработки спецификаций (определение требований), то в этих процессах используются методы, приводящие к созданию формализованных документов, т. е. используются формализованные языки спецификаций. При этом широко используются графические методы спецификаций, приводящие к созданию различных схем и диаграмм, которые определяют структуру информационной среды и структуру управления ПС. К таким структурам привязываются фрагменты описания данных и программ, представленные на алгебраических языках спецификаций (например, использующие операционную или аксиоматическую семантику), или логических языках спецификаций (базирующихся на логическом подходе к спецификации программ). Такие спецификации позволяют в значительной степени или полностью автоматически генерировать программы. Существенной частью разработки спецификаций является создание словаря именованных сущностей, используемых в спецификациях.

Автоматизированный контроль спецификаций ПС использует то обстоятельство, что значительная часть спецификаций представляется на формальных языках. Это позволяет автоматически осуществлять различные виды контроля: синтаксический и частичный семантический контроль спецификаций, контроль полноты и состоятельности схем и диаграмм (в частности, все их элементы должны быть идентифицированы и отражены в словаре именованных сущностей), сквозной контроль сбалансированности уровней спецификаций и другие виды контроля в зависимости от возможностей языков спецификаций.

Генерация программ ПС На этом этапе автоматически генерирует скелеты кодов программ ПС или полностью коды этих программ по формальным спецификациям ПС.

Автоматизированное документирование ПС- Предполагает возможность генерации различных форм документов с частичным заполнением их по информации, хранящейся в репозитории. При этом количество видов документов сокращается по сравнению с традиционной технологией.

Комплексное тестирование и отладка ПС На этом этапе тестируются все спецификации ПС и исправляются обнаруженные при этом ошибки. Тесты могут создаваться как вручную, так и автоматически (если это позволяют используемые языки спецификаций) и пропускаются через сгенерированные программы ПС.

Аттестация ПС имеет прежнее содержание.

Сопровождение ПС существенно упрощается, так как основные изменения делаются только в спецификациях.

Рабочее место компьютерной технологии разработки ПС представляет собой инструментальную среду, поддерживающую все этапы жизненного цикла этой технологии. В этой среде существенно используется репозиторий. В репозитории хранится вся информация, создаваемая в процессе разработки ПС (в частности, словарь именованных сущностей и все спецификации). По существу, рабочее место компьютерной технологии является интегрированным хотя бы по пользовательскому интерфейсу и по данным. Основными инструментами такого рабочего места являются:

- конструкторы пользовательских интерфейсов;
- инструмент работы со словарем именованных сущностей;
- графические и тестовые редакторы спецификаций;

- анализаторы спецификаций;
- генератор программ;
- документаторы.

14.5. Инструментальные системы технологии программирования.

Инструментальная система технологии программирования – это интегрированная совокупность программных и аппаратных инструментов, поддерживающая все процессы разработки и сопровождения больших ПС в течение всего его жизненного цикла в рамках определенной технологии. Выше уже отмечалось (см. п. 14.3), что она помимо интегрированности обладает еще свойствами комплексности и ориентированности на коллективную разработку. Это означает, что она базируется на согласованности продукции технологических процессов. Тем самым, инструментальная система в состоянии обеспечить, по крайней мере, контроль полноты (комплексности) создаваемой документации (включая набор программ) и согласованности ее изменения (версионности). Поддержка инструментальной системой фазы сопровождения ПС, означает, что она должна обеспечивать *управление конфигурацией ПС*. Кроме того, инструментальная система поддерживает управление работой коллектива и для разных членов этого коллектива обеспечивает разные права доступа к различным фрагментам продукции технологических процессов и поддерживает работу *менеджеров* [16.1] по управлению коллективом разработчиков. Инструментальные системы технологии программирования представляют собой достаточно большие и дорогие ПС, чтобы как-то была оправданна их инструментальная перегруженность. Поэтому набор включаемых в них инструментов тщательно отбирается с учетом потребностей предметной области, используемых языков и выбранной технологией программирования.

С учетом обсужденных свойств инструментальных систем технологии программирования можно выделить три их основные компоненты:

- репозитории,
- инструментарий,
- интерфейсы.

Инструментарий - набор инструментов, определяющий возможности предоставляемые системой коллективу разработчиков. Обычно этот набор является открытым и структурированным. Помимо минимального набора (*встроенные инструменты),* он содержит средства своего расширения *(импортированными инструментами).* Кроме того, в силу интегрированности по действиям он состоит из некоторой общей части всех инструментов *(ядра)* и структурных (иногда иерархически связанных) классов инструментов.

Интерфейсы разделяются на пользовательский и системные. *Пользовательский* интерфейс обеспечивает доступ разработчикам к инструментарию. Он реализуется *оболочкой* системы. *Системные* интерфейсы обеспечивают взаимодействие между инструментами и их общими частями. Системные интерфейсы выделяются как архитектурные компоненты в связи с открытостью системы - их обязаны использовать новые *{импортируемые)* инструменты, включаемые в систему.

Самая общая архитектура инструментальных систем технологии программирования представлена на рис. 16.4.

Рис. 16.4. Общая архитектура инструментальных систем технологии программирова-

ния.

Различают два класса инструментальных систем технологии программирования: инструментальные системы поддержки проекта и языково-зависимые инструментальные системы.

Инструментальная система поддержки проекта - это открытая система, способная поддерживать разработку ПС на разных языках программирования после соответствующего ее расширения программными инструментами, ориентированными на выбранный язык. Набор инструментов такой системы поддерживает разработкой ПС, а также содержит независимые от языка программирования инструменты, поддерживающие разработку ПС (текстовые и графические редакторы, генераторы отчетов и т.п.). Кроме того, он содержит инструменты расширения системы. Ядро такой системы обеспечивает в частности, доступ к репозиторию.

Языково-зависимая инструментальная система - это система поддержки разработки ПС на каком-либо одном языке программирования, существенно использующая в организации своей работы специфику этого языка. Эта специфика может сказываться и на возможностях ядра (в том числе и на структуре репозитория), и на требованиях к оболочке и инструментам. Примером такой системы является среда поддержки программирования на Аде (APSE [16.5]).

Упражнения к лекции 16.

16.1. Что такое программный инструмент разработки ПС?

16.2. Что такое аппаратный инструмент разработки ПС?

16.3. Что такое инструментальная среда разработки и сопровождения ПС?

16.4. Что такое инструментально-объектный подход к разработке программного средства?

16.5.Какие признаки классификации инструментальных сред разработки и сопровождения ПС Вы знаете?

16.6. Что такое интегрированность инструментальной среды разработки и сопровождения ПС[9]

16.7. Какие виды интегрированности инетр\'менталъной среды разработки и сопровождения ПС Вы знаете?

16.8. Что такое релозитории инструментальной среды разработки и сопровождения ПС?

16.9.Что такое инструментальная среда программирования?

16.10. Что такое языково-ориентированная инструментальная среда программирования?

16.11. Что такое компьютерная технология (CASE-технология) разработки ПС?

16.12.Какие отличия .жизненного цикла ПС при компьютерной технологии программирования от жизненного никла ПС при традиционной (ручной) технологии программирования (при водопадном подходе)?

16.13.Что такое рабочее место компьютерной технологии разработки и сопровождения ПС?

16.14.Что такое инструментальная система технологии программирования?

16.15. Что такое языково-зависимая инструментальная система технологии программирования?

16.16. Что такое ядро инструментальной системы технологии программирования?

16. 17. Что такое встроенный инструмент инструментальной системы технологии программирования?

16.18. Что такое импортируемый инструмент инструментальной системы технологии программирования?

16.19. Что такое оболочка инструментальной системы технологии программирования ?

Литература к лекции 16.

16.1. Ian Sommerville. Software Engineering. - Addison-Wesley Publishing Company, 1992. P. 349-369.

16.2. Е.А. Жоголев. Введение в технологию программирования (конспект лекций). - М.: "ДИАЛОГ-МГУ", 1994.

16.3. М.М. Горбунов-Посадов. Конфигурации программ. Рецепты безболезненных изменений. - М.: «Малип». 1994.

16.4. CASE: Компьютерное проектирование программного обеспечения. - Издательство Московского университета, 1994.

16.5. Requirements for Ada Programming Support Environments. - USA: DoD, Stoneman, 1980.

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНЫМ ЗАНЯТИЯМ

Федеральное агентство по образованию Российской Федерации
АМУРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Факультет математики и информатики

И.М. Акилова, Л.В. Чепак, Е.Н. Архипова

ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЯ.
Программирование на языке Java

Допущено учебно-методическим объединением (*УМО*) *вузов по университетскому политехническому образованию в качестве учебного пособия*

Благовещенск
2007

ББК

Печатается по решению
редакционно-издательского совета
факультета математики и информатики
Амурского государственного университета

Акилова И.М., Чепак Л.В., Архипова Е.Н.

Технология программирования. Программирование на языке Java: Учебное пособие. Допущено учебно-методическим объединением вузов по университетскому политехническому образованию в качестве учебного пособия для студентов высших учебных заведений, обучающихся по специальности 230102 очной формы обучения.

Благовещенск: Амурский гос. ун-т, 2007.

Пособие рекомендуется студентам, изучающим курс «Технологияпрограммирования», а также всем желающим ознакомиться с структурой Java-программы, разработкой простейших апплетов и апплетов двойного назначения, обработкой событий, графикой, анимацией, элементами управления, контейнерами компонентов, реализацией многозадачности в Java, многопотоковыми, автономными и сетевыми приложениями, сокетами и их применением, созданием и использованием сервлетов, работой с файлами и базами данных. Получить навыки программирования на основе приведенных методов.

Рассчитано на преподавателей и студентов.

Рецензенты: В.Д. Епанешников, проф. кафедры автоматики и системотехники ТОГУ, д-р физ.-мат. наук;

А.Н. Рыбалев, доцент кафедры автоматизации производственных процессов и электротехники АмГУ, к.т. наук;

А.Н. Семочкин, начальник управления информационных и телекоммуникационных технологий и информационной безопасности БГПУ, доцент кафедры информатики БГПУ, к.ф.-м. наук.

ВВЕДЕНИЕ

При изучении дисциплины «Технологий и методов программирования» акцент делается на изучение процессов разработки программных средств, в которую включены все этапы, начиная с момента зарождения идеи этого средства. Каждый этап этой совокупности базируется на использовании каких-либо методов и средств.

В настоящее время слово Java стало известно практически всем. С одной стороны, язык Java расширяет возможности разработчиков WWW-серверов, а с другой - помогает программисту превратить WWW в платформу программирования. Основным вкладом нового языка является независимый доступ к исполняемому содержимому - для Java-приложений безразлично, на какой платформе оно работает. Визуализация информации при использовании Java становится все более утонченной, позволяя кому угодно с помощью броузера, поддерживающего Java, увидеть вещи, ранее доступные только в лабораториях.

Учебное пособие «Технология программирования. Программирование на языке Java» составлено для студентов специальности «Автоматизированные системы обработки информации и управления» по курсу «Технология программирования».

В данном пособии рассмотрены основные вопросы программирования на языке Java: структура Java-программы, разработка простейших апплетов и апплетов двойного назначения, классы, создание и использование пакетов, обработка событий от мыши и от клавиату-

ры, графика, цвет, шрифты, анимация, элементы управления и устройства, контейнеры компонентов, реализация многозадачности в Java, многопотоковые, автономные и сетевые приложения, сокеты и их применение, создание и использование сервлетов, работа с файлами и базами данных.

Цели, которые ставит пособие:

1) изучение объектно-ориентированного языка программирования Java;
2) усвоение и закрепление основных приемов и алгоритмов языка;
3) применение навыков программирования для создания программных продуктов с использованием Java-файлов.

В каждой лабораторной работе даются методические указания, содержащие основные теоретические сведения, рассматриваются различные примеры программ, приложений и апплетов. В конце каждой лабораторной работы сформулированы задания для самостоятельного выполнения и контрольные вопросы для самопроверки. В конце методического пособия приведены приложения, содержащие Java-файлы различных апплетов.

СОДЕРЖАНИЕ

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ СТУДЕНТОВ

В процессе подготовки к лабораторным занятиям, студентам необходимо обратить особое внимание на самостоятельное изучение рекомендованной учебно-методической (а также научной и популярной) литературы. Самостоятельная работа с учебниками, учебными пособиями, научной, справочной и популярной литературой, материалами периодических изданий и Интернета, статистическими данными является наиболее эффективным методом получения знаний, позволяет значительно активизировать процесс овладения информацией, способствует более глубокому усвоению изучаемого материала, формирует у студентов свое отношение к конкретной проблеме.

Более глубокому раскрытию вопросов способствует знакомство с дополнительной литературой, рекомендованной преподавателем по каждой теме семинарского или практического занятия, что позволяет студентам проявить свою индивидуальность в рамках выступления на данных занятиях, выявить широкий спектр мнений по изучаемой проблеме.