# ЗАМОК ДРАКОНА

Математика делает то, что можно, так, как нужно, тогда как информатика делает то, что нужно, так, как можно.

Программистский фольклор

## НАДЁЖНОЕ 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. Роль в обществе компьютеров и программирования, информатизация общества. Взаимосвязь программирования и других областей знания. Применение, злоупотребление и границы компьютерной техники.

### Программа как формализованное описание процесса обработки данных.

### Программное средство

Целью программирования является описание процессов обработки данных (в дальнейшем — просто процессов). Согласно ИФИПа [1]: данные — это представление фактов и идей в формализованном виде, пригодном для передачи и переработке в некоем процессе, а информация — это смысл, который придается данным при их представлении. Обработка данных — 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. 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 — означает 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.

### 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 [2] 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 этом случае правильнее говорить об ошибке в ПС. Разновидностью ошибки в ПС является несогласованность между программами ПС и документацией по их применению. В работе [3] выделяется в отдельное понятие частный случай ошибки в ПС, когда программа не соответствует своей функциональной спецификации (описанию, разрабатываемому 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.

В связи с тем, что задание на ПС обычно формулируется не формально, а также из-за неформализованности понятия ошибки в ПС, нельзя доказать формальными методами (математически) правильность ПС. Нельзя показать правильность ПС и тестированием: как указал Дейкстра [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 ПС — это его способность безотказно выполнять определенные функции при заданных условиях в течение заданного периода времени с достаточно большой вероятностью [5]. При этом под отказом в ПС понимают проявление в нем ошибки [2]. Таким образом, надежная ПС не исключает наличия в ней 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.

Разрабатываемая ПС может обладать различной степенью надежности. Как измерять эту степень? Так же как в технике, степень надежности можно характеризовать [2] вероятностью работы ПС без отказа в течении 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.

### Технология программирования как технология разработки надежных программных средств

В соответствии с обычным значением слова «технология» [6] под технологией программирования будем понимать совокупность производственных процессов, приводящую к созданию требуемого ПС, а также описание этой совокупности 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 об компьютерной технологии программирования).

В литературе имеются и другие, несколько отличающиеся, определения технологии программирования. Эти определения обсуждаются в работе [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 [7]. Именно программной инженерии (Software Engineering) посвящена упомянутая работа [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 прежде всего методы и инструментальные средства разработки ПС с точки зрения достижения определенных целей — они могут использоваться в разных технологических процессах (и в разных технологиях программирования); как эти методы и средства образуют технологические процессы — здесь вопрос второстепенный.

Не следует также путать технологию программирования с методологией программирования [8]. Хотя в обоих случаях изучаются методы, но 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 [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 обсуждать технологию программирования как технологию разработки надёжных ПС. Это означает, что, во-первых, мы будем обсуждать все процессы разработки ПС, начиная с момента возникновения замысла ПС. Во-вторых, нас будут интересовать не только вопросы построения программных конструкций, но и вопросы описания функций и принимаемых решений с точки зрения их человеческого (неформального) восприятия, и, наконец, в качестве продукта технологии мы будем принимать надежную (а не правильную) ПС. Все это будет существенно влиять на выбор методов и инструментальных средств в процессах разработки ПС.

### 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, в котором акценты стали ставится не столько на его эффективности, сколько на удобстве работы с ним для пользователей (не говоря уже о его надежности). Широкое использование компьютерных сетей привело к интенсивному 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 мыслимого 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 (компьютеры первого поколения) была невелика, а программирование для них велось, в основном, в машинном коде. Решались, главным образом, научно-технические задачи (счёт по формулам), задание на программирование уже 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 [10] (для 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.), роль которых в технологии программирования явно преувеличивалась. Надежда на то, что эти языки решат все проблемы при 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 [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. Это стало возможным с использованием коллективной разработки, которая поставила ряд серьезных технологических проблем [11].

В 70-е годы получили широкое распространение информационные системы и базы данных. Этому способствовало очень важное событие, происшедшее в середине 70-ых годов: стоимость хранения одного бита информации на компьютерных носителях стала меньше, чем на традиционных. Интенсивно развивалась технология программирования [2, 8, 12, 13, 14]: 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 [5, 15, 16, 17, 18]. Появляются языки программирования (например, Ада), учитывающие требования технологии программирования [19]. Развиваются методы и языки спецификации ПС [1.20-1.21]. Выходит на передовые позиции объектный подход к разработке ПС [9]. Создаются различные инструментальные среды разработки и сопровождения ПС [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 регулирования доступа к компьютерно-сетевой информации (как технологического, так и юридического и этического характера). 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.

**ЛИТЕРАТУРА**

1. И.Г. Гоулд, Дж.С. Тутилл. Терминологическая работа IFIP (Международная федерация по обработке информации) и ICC (Международный вычислительный центр) // Журн. вычисл. матем. и матем. физ., 1965, №2. — с. 377-386.
2. Г. Майерс. Надежность программного обеспечения. — М.: Мир, 1980.
3. Ian Sommerville. Software Engineering. — Addison-Wesley Publishing Company, 1992.
4. Э. Дейкстра. Заметки по структурному программированию // У. Дал, Э. Дейкстра, К. Хоор. Структурное программирование. — М.: Мир, 1975. — с. 7-97.
5. Criteria for Evalution of Software. — ISO TC97/SC7 #367 (Supersedes Document #327).
6. С.И. Ожегов. Словарь русского языка. — М.: Советская энциклопедия, 1975.
7. Ф.Я. Дзержинский, И.М. Калиниченко. Дисциплина программирования Д: концепция и опыт реализации методических средств программной инженерии. — М.: ЦНИИ информации и технико-экономических исследований по атомной науке и технике, 1988. — с. 9-16.
8. В. Турский. Методология программирования. — М.: Мир, 1981.
9. Г. Буч. Объектно-ориентированное проектирование с примерами применения: пер. с англ. — М.: Конкорд, 1992.
10. Е.А. Жоголев. Система программирования с использованием библиотеки подпрограмм // Система автоматизация программирования. — М.: Физматгиз, 1961. с. 15-52.
11. Ф.П. Брукс, мл. Как проектируются и создаются программные комплексы / Пер. с англ. А.П. Ершова. — М.: Наука, 1979.
12. R.C. Holt. Structure of Computer Programs: A Survey // Proceedings of the IEEE, 1975, 63(6). — p. 879-893.
13. Дж. Хьюз, Дж. Мичтом. Структурный подход к программированию. — М.: Мир, 1980.
14. Е.А. Жоголев. Технологические основы модульного программирования // Программирование, 1980, №2. — с.44-49.
15. Б. Боэм, Дж. Браун, Х. Каспар и др. Характеристики качества программного обеспечения. — М.: Мир, 1981.
16. В.В. Липаев. Качество программного обеспечения. — М.: Финансы и статистика, 1983.
17. Б. Шнейдерман. Психология программирования. — М.: Радио и связь, 1984.
18. Revised version of DP9126 — Criteria of the Evaluation of Software Quality Characteristics. ISO TC97/SC7 #610. — Part 6.
19. В.Ш. Кауфман. Языки программирования. Концепции и принципы. М.: Радио и связь, 1993.
20. Требования и спецификации в разработке программ: пер. с англ. — М.: Мир, 1984.
21. В.Н. Агафонов. Спецификация программ: понятийные средства и их организация. — Новосибирск: Наука (Сибирское отделение), 1987.

# ЗАМОК ДРАКОНА

Математика делает то, что можно, так, как нужно, тогда как информатика делает то, что нужно, так, как можно. Человеку свойственно ошибаться.

Сенека

##

## 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 из одного представления в другое — основная причина ошибок при разработке программных средств. Модель перевода и источники ошибок.

### Интеллектуальные возможности человека

Дейкстра [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 с узнаванием искомого предмета. 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]. Под простой системой будем понимать такую систему, в которой человек может уверенно перебрать все пути взаимодействия между ее элементами, а под сложной системой — такую систему, в которой он этого сделать не в состоянии. Между простыми и сложными системами нет чёткой 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 обо всех связях между её элементами из-за возможных ошибок. Поэтому полезно уметь оценивать сложность системы по числу ее элементов: числом потенциальных путей взаимодействия между её элементами, т.е. n! , где n — число её элементов. Систему назовём малой, если n < 7 (6! = 720 < 1000), систему назовём 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: «всё, 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.

### Неправильный перевод как причина ошибок в программных средствах

При разработке и использовании ПС мы многократно имеем дело [3] с преобразованием (переводом) информации из одной формы в другую (см. Рис. 1). Заказчик формулирует свои потребности в ПС в виде некоторых требований. Исходя из этих требований, разработчик создаёт 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 её 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.

Требования к ПС

Внешнее описание ПС

спецификация базового программного обеспечения

*Спецификация заданной аппаратуры*

тексты программ ПС

Спецификация языка программирования

*пользовательская документация*

Рис. 1. Грубая схема разработки и применения ПС.

На каждом из этих этапов перевод информации может быть осуществлён неправильно, например, из-за неправильного понимания исходного представления информации. Возникнув на одном из этапов ошибка в представлении информации распространяется на последующие этапы разработки и, в конечном счёте, окажется в самом ПС.

### Модель перевода

Чтобы понять природу ошибок при переводе рассмотрим модель [3], изображённую на Рис. 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.

A

###### *M*

*Информация, описывающая процесс преобразования*

*преобразуемая информация*

B

Рис. 2. Модель перевода.

На каждом из этих шагов человек может совершить ошибку разной природы. На первом шаге способность человека «читать между строк» (способность, позволяющая ему понимать текст, содержащий неточности или даже ошибки) может стать причиной ошибки в ПС. Ошибка возникает в том случае, когда при чтении документа A человек, пытаясь восстановить недостающую информацию, видит то, что он ожидает, а не то, что имел в виду автор документа A. В этом случае лучше было бы обратиться к автору документа за разъяснениями. При запоминании информации человек осуществляет её осмысливание (здесь важен его уровень подготовки и знание предметной области, к которой относится документ A). И, если он поверхностно или неправильно поймёт, то информация будет запомнена в искажённом 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 осуществлён неверно. Это обычно происходит при большом объёме плохо организованной информации. И, наконец, на последнем этапе стремление человека поскорее зафиксировать информацию часто приводит к тому, что представление этой информации оказывается неточным, создавая ситуацию для последующей неоднозначной её интерпретации.

### 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).

#### ЛИТЕРАТУРА

1. Э. Дейкстра. Заметки по структурному программированию // У. Дал, Э. Дейкстра, К. Хоор. Структурное программирование. — М.: Мир, 1975. — с. 7-97.
2. Е.А. Жоголев. Технологические основы модульного программирования // Программирование, 1980, №2. — с.44-49.
3. Г.Майерс. Надежность программного обеспечения. — М.: Мир, 1980.

# ЗАМОК ДРАКОНА

Лучшее - враг хорошего.

Народная мудрость

## ОБЩИЕ ПРИНЦИПЫ РАЗРАБОТКИ ПРОГРАММНЫХ СРЕДСТВ

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

### Специфика разработки программных средств

Разработке программных средств присущ ряд специфических особенностей [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 существенно 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.

### Жизненный цикл программного средства

Под жизненным циклом ПС понимают весь период его разработки и эксплуатации (использования), начиная от момента возникновения замысла ПС и кончая прекращением всех видов его использования [1, 2, 3, 4]. Жизненный цикл включает все процессы создания и использования ПС (software process).

Различают следующие стадии жизненного цикла ПС (см. Рис. 1): разработку ПС, производство программных изделий (ПИ) и эксплуатацию ПС.



Рис. 1. Стадии и фазы жизненного цикла 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 управление (management) разработкой 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.

Внешнее описание (Requirements document) ПС является описанием его поведения с точки зрения внешнего по отношению к нему наблюдателю с фиксацией требований относительно его качества. Внешнее описание ПС начинается с определения требований к ПС со стороны пользователей (заказчика).

Конструирование (design) ПС охватывает процессы: разработку архитектуры ПС, разработку структур программ ПС и их детальную спецификацию.

Кодирование (coding) — создание текстов программ на языках программирования, их отладку с тестированием ПС.

На этапе аттестации ПС производится оценка качества ПС, после успешного завершения, которого разработка ПС считается законченной.

Программное изделие (ПИ) — экземпляр или копия, снятая с 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 [1]. Стадия производства ПС в жизненном цикле ПС является, по существу, вырожденной (несущественной), 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 (operation) ПИ по своему назначению. Она состоит из двух параллельно проходящих фаз: фазы применения ПС и фазы сопровождения ПС [4, 5].

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

Сопровождение (maintenance) ПС — это процесс сбора информации о его качестве в эксплуатации, устранения обнаруженных в нем ошибок, его доработки и модификации, а также извещения пользователей о внесенных в него изменениях [1, 4, 5].

### 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 ПС — это совокупность его черт и характеристик, которые влияют на его способность удовлетворять заданные потребности пользователей [6]. Это не означает, что разные ПС должны обладать одной и той же совокупностью таких свойств в их высшей возможной степени. Этому препятствует тот факт, что повышение качества ПС по одному из таких свойств часто может быть достигнуто лишь ценой 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 должны быть, прежде всего, фиксированы критерии отбора требуемых свойств ПС. В настоящее время критериями качества ПС принято считать [6, 7, 8, 9, 10]:

* функциональность,
* надёжность,
* лёгкость 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.

Лёгкость применения — это характеристики ПС, которые позволяют минимизировать усилия пользователя по подготовке исходных данных, применению ПС и оценке полученных результатов, а также вызывать положительные эмоции определённого 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 нём 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 надёжность являются обязательными критериями качества ПС, причём обеспечение надёжности 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.

### Обеспечение надёжности — основной мотив разработки программных средств

Рассмотрим теперь общие принципы обеспечения надёжности ПС, что, как мы уже подчёркивали, является основным мотивом разработки ПС, задающим специфическую окраску всем технологическим процессам разработки ПС. В технике известны четыре подхода обеспечению надёжности [11]:

* 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:

* борьбе со сложностью;
* обеспечении точности перевода;
* преодоления барьера между пользователем и разработчиком;
* обеспечения 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 средств приводит к её усложнению (иногда — значительному), что в какой-то мере мешает методам предупреждения ошибок.

### 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 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.

### 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 [11]. Лучшим руководством по решению задач он считает книгу Пойа «Как решать задачу» [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.

### 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. Поэтому следует — привлекать пользователя в процессы принятия решений при разработке ПС, — тщательно освоить особенности его работы (лучше всего — побывать в его «шкуре»).

### 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 учётом 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).

#### ЛИТЕРАТУРА

1. Е.А. Жоголев. Введение в технологию программирования (конспект лекций). — М.: «ДИАЛОГ-МГУ», 1994.
2. М. Зелковец, А. Шоу, Дж. Гэннон. Принципы разработки программного обеспечения. — М.: Мир, 1982, с. 11.
3. К. Зиглер. Методы проектирования программных систем. — М.: Мир, 1985, с. 15-23.
4. Дж. Фокс. Программное обеспечение и его разработка. — М.: Мир, 1985, с. 53-67, 125-130.
5. Ian Sommerville. Software Engineering. — Addison-Wesley Publishing Company, 1992.
6. Criteria for Evalution of Software. — ISO TC97/SC7 #383.
7. Revised version of DP9126 — Criteria for Evalution of Software Quality Characteristics. — ISO TC97/SC7 #610. — Part 6.
8. Б. Боэм, Дж. Браун, Х. Каспар и др. Характеристики качества программного обеспечения. — М.: Мир, 1981.
9. В.В. Липаев. Качество программного обеспечения. — М.: Финансы и статистика, 1983.
10. Б. Шнейдерман. Психология программирования. — М.: Радио и связь, 1984. — с. 99-103.
11. Г.Майерс. Надежность программного обеспечения. — М.: Мир, 1980.
12. Д. Пойа. Как решать задачу. — М.: Наука, 1961.

# ЗАМОК ДРАКОНА

Не переходи мост, пока не дошел до него.

Народная пословица

## 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

### 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 этапа формулирования требований к ПС, на котором, исходя из довольно смутных пожеланий заказчика, должен быть получен документ, достаточно точно определяющий задачи разработчиков ПС. Этот документ мы называем внешним описанием ПС (в литературе его часто называют спецификацией требований [1]).

Очень часто требования к ПС путают с требованиями к процессам его разработки (к технологическим процессам). Последние включать во внешнее описание не следует, 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 (руководстве) 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 особенно серьезных мер по их предупреждению.

Исходным документом для разработки внешнего описания ПС являются определение требований 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 [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 [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. Иногда для прояснения действительных потребностей пользователей 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 [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 [1], но она, как правило, включает и требования к технологическим процессам). Она, в отличие от функциональной спецификации, реализуется неформализованно 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 разрабатываемого 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 = 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 вопрос, 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.

### Определение требований к программному средству

Определение требования к ПС являются исходным документом разработки ПС — заданием, выражающем 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 знающего специальных программистских обозначений. 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 [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 определения требований к ПС [2]:

* управляемый пользователем,
* контролируемый пользователем,
* независимый от пользователя.

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 и контролю за тем, чтобы формулируемые требования действительно выражали его потребности в ПС. В конечном счёте 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.

### Спецификация качества программного средства

Разработка спецификации качества сводится, по существу, к построению своеобразной модели качества разрабатываемой ПС [2, 3]. В этой модели должен быть перечень всех тех достаточно элементарных свойств, которые требуется обеспечить в разрабатываемом 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 разрабатываемому 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 [3, 4, 5, 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: временнaя эффективность, эффективность по памяти, эффективность по устройствам.

Сопровождаемость. С данным критерием связано много различных примитивов качества. Однако их можно распределить по двум группам, выделив два подкритерия качества: изучаемость и модифицируемость. Изучаемость — это характеристики ПС, которые позволяют минимизировать усилия по изучению и пониманию программ и документации ПС. Модифицируемость — это характеристики ПС, которые упрощают внесение в него необходимых изменений и доработок.

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

Модифицируемость: расширяемость, структурированность, модульность.

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

Ниже даются определения используемых примитивов качества ПС [3, 4, 5].

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

Точность — мера, характеризующая приемлемость величины погрешности в выдаваемых программами ПС результатах с точки зрения предполагаемого их использования.

Автономность — свойство, характеризующее способность ПС выполнять предписанные функции без помощи или поддержки других компонент программного обеспечения.

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

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

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

Информативность — 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 — свойство, характеризующее степень, в которой ПС облегчает задание или описание входных данных, а также обеспечивает выдачу полезных сведений в форме и с содержанием, простыми для понимания.

Временнaя эффективность — мера, характеризующая способность ПС выполнять возложенные на него функции за определенный отрезок времени.

Эффективность по памяти — мера, характеризующая способность ПС выполнять возложенные на него функции при определенных ограничениях на используемую память.

Эффективность по устройствам — мера, характеризующая экономичность использования устройств машины для решения поставленной задачи.

С-документировапнность — 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 — 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.

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

Удобочитаемость — свойство, характеризующее легкость восприятия текста программ ПС (отступы, фрагментация, формативность).

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

Модульность — 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 — свойство, характеризующее способность ПС работать на разнообразном аппаратном обеспечении (различных типах, марках, моделях ЭВМ).

### 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 с точки зрения внешнего наблюдателя (пользователя) случаи, когда ПС не сможет нормально выполнить ту или иную свою функцию (например, при обнаружении ошибки во время взаимодействия с пользователем, или при попытке применить какую-либо функцию к данным, не удовлетворяющим соотношениям, указанным в ее спецификации, или при получении результата, нарушающего заданное ограничение). Для каждого такого случая должна быть определена (описана) реакция ПС.

### 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 её проверка со стороны разработчика требований к ПС, а смежный контроль функциональной спецификации — это её 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 эту 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.

#### ЛИТЕРАТУРА

1. Ian Sommerville. Software Engineering. — Addison-Wesley Publishing Company, 1992.
2. Г.Майерс. Надежность программного обеспечения. — М.: Мир, 1980. с. 49-77.
3. Е.А. Жоголев. Введение в технологию программирования (конспект лекций). — М.: «ДИАЛОГ-МГУ», 1994.
4. Criteria for Evalution of Software. — ISO TC97/SC7 #383.
5. Revised version of DP9126 — Criteria for Evalution of Software Quality Characteristics. — ISO TC97/SC7 #610. — Part 6.
6. Б. Боэм, Дж. Браун, Х. Каспар и др. Характеристики качества программного обеспечения. — М.: Мир, 1981.

# ЗАМОК ДРАКОНА

Все, что вообще может быть сказано, должно быть сказано ясно, а о чем невозможно говорить, о том следует молчать.

##### Л. Витгенштейн

## МЕТОДЫ СПЕЦИФИКАЦИИ СЕМАНТИКИ ФУНКЦИЙ

Основные подходы к спецификации семантики функций. Табличный подход, метод таблиц решений. Алгебраический подход: операционная, денотационная и аксиоматическая семантика.

### Основные подходы к спецификации семантики функций

Для спецификации семантики функций используются следующие подходы: табличный, алгебраический и логический (1), а также графический (5.2).

Табличный подход для определения функций хорошо известен еще со средней школы. Он базируется на использовании таблиц. В программировании эти методы получили развитие в методе таблиц решений.

Алгебраический подход базируется на использовании равенств для определения функций. В этом случае для определения некоторого набора функций строится система равенств вида:

 L1= R1,

(1) .………..

 Ln = Rn.

где Li и Ri, i = 1, ..., n, некоторые выражения, содержащие предопределенные операции, константы, переменные, от которых зависят определяемые функции (формальные параметры этих функций), и вхождения самих этих функций. Семантика определяемых функций извлекается в результате интерпретации этой системы равенств. Эта интерпретация может производиться по-разному (базироваться на разных системах правил), что порождает разные семантики. В настоящее время активно исследуются операционная, денотационная и аксиоматическая семантики.

Третий подход, логический, базируется на использовании предикатов — функций, аргументами которых могут быть значения различных типов, а результатами которых являются логические значения (ИСТИНА и ЛОЖЬ). В этом случае набор функций может определяться с помощью системы предикатов. Заметим, что система равенств алгебраического подхода может быть задана с помощью следующей системы предикатов:

 РАВНО(L1, R1),

(2) ……………….

 РАВНО(Ln, Rn),

где предикат РАВНО истинен, если равны значения первого и второго его аргументов. Это говорит о том, что логический подход располагает большими возможностями для определения функций, однако он требует от разработчиков ПС умения пользоваться методами математической логики, что, к сожалению, не для всех коллективов разработчиков оказывается приемлемым. Более подробно этот подход в нашем курсе лекций мы рассматривать не будем.

Графический подход также известен еще со средней школы. Но в данном случае речь идет не о задании функции с помощью графика, хотя при данном уровне развития компьютерной техники ввод в компьютер таких графиков возможен и они могли бы использоваться (с относительно небольшой точностью) для задания функций. В данном случае речь идет о графическом задании различных схем, выражающих сложную функцию через другие функции, связанными с какими-либо компонентами заданной схемы. Графическая схема может определять ситуации, когда для вычисления представляемой ею функции должны применяться связанные с этой схемой более простые функции. Графическая схема может достаточно точно и формализовано определять часть семантики функции. Примером такой схемы может быть схема переходов состояний конечного автомата, такая, что в каждом из этих состояний должна выполняться некоторая дополнительная функция, указанная в схеме.

### Метод таблиц решений

Метод таблиц решений базируется на использовании таблиц следующего вида (см. Табл. 1).

Верхняя часть этой таблицы определяет различные ситуации, в которых требуется выполнять некоторые действия (операции). Каждая строка этой части задаёт ряд значений некоторой указанной переменной или некоторого указанного условия в первом поле (столбце) этой строки. Таким образом, первый столбец этой части представляет собой список переменных или условий, от значений которых зависит выбор определяемых ситуаций. В каждом следующем столбце указывается комбинация значений этих переменных (условий), определяющая конкретную ситуацию. При этом последний столбец определяет ситуацию, отличную от предыдущих, т.е. для любых других комбинаций значений (будем обозначать их звездочкой \*), отличных от первых, определяется одна и та же, (m+1)-ая, ситуация. Впрочем, в некоторых таблицах решений этот столбец может отсутствовать. Эта часть таблицы решений аналогична соответствующей части таблицы, определяющей какую-либо функцию обычным способом - в ней задаются комбинации значений аргументов функции, которым ставится в соответствие значения этой функции.

Табл. 1. Общая схема таблиц решений

|  |  |
| --- | --- |
| Переменные / условия | Ситуации (комбинации значений) |
| x1 | a1,1 | a1,2 | ... | a1,m | \* |
| x2 | a2,1 | a2,2 | ... | a2,m | \* |
| .….. | .…………………………………………. |
| xn | an,1 | an,2 | ... | an,m | \* |
| S1 | u1,1 | u1,2 | ... | u1,m | u1,m+1 |
| S2 | u2,1 | u2,2 | ... | u2,m | u1,m+1 |
| .….. | .………………………………………… |
| s1k | uk,1 | uk,2 | ... | uk,m | uk,m+1 |
| Действия | Комбинации выполняемых действий |

Нижняя часть таблицы решений определяет действия, которые требуется выполнить в той или иной ситуации, определяемой в верхней части таблицы решений. Она также состоит из нескольких (k) строк, каждая из которых связана с каким-либо одним конкретным действием, указанным в первом поле (столбце) этой строки. В остальных полях (столбцах) этой строки (т.е., для ui j, i = 1, ..., m+1, j = 1, ..., k) указывается, следует ли (ui,j = '+') выполнять это действие в данной ситуации или не следует (ui,j = '–'). Таким образом, первый столбец нижней части этой таблицы представляет собой список обозначений действий, которые могут выполняться в той или иной ситуации, определяемой этой таблицей. В каждом следующем столбце этой части указывается комбинация действий, которые следует выполнить в ситуации, определяемой в том же столбце верхней части таблицы решений. Для ряда таблиц решений эти действия могут выполняться в произвольном порядке, но для некоторых таблиц решений этот порядок может быть предопределен, например, в порядке следования соответствующих строк в нижней части этой таблицы.

Табл. 2. Таблица решений «Светофор у пешеходной дорожки».

|  |  |
| --- | --- |
| Условия | Ситуации |
| Состояние светофора |  |  |  |  |  |  |  |  |
| T = Tкр | Нет | Нет | Да | \* | \* | \* | \* | \* |
| T = Tжёл | \* | \* | \* | Нет | Да | \* | \* | \* |
| T > Tзел | \* | \* | \* | \* | \* | Нет | Да | Да |
| Появление привилегированной машины | Нет | Да | \* | \* | \* | \* | Нет | Да |
| Включить  | – | – | – | – | – | – | + | – |
| Включить  | – | + | + | – | – | – | – | – |
| Включить  | – | – | – | – | + | – | – | – |
| T := 0 | – | + | + | – | + | – | + | – |
| T := T+1 | + | – | – | + | – | + | – | + |
| Освобождение пешеходной дорожки | – | – | – | + | – | – | – | – |
| Пропуск пешеходов | + | + | + | – | – | – | – | – |
| Пропуск машин | – | – | – | – | – | + | + | + |
| Действия | Комбинации выполняемых действий  |

Рассмотрим в качестве примера описание работы светофора у пешеходной дорожки. Переключение светофора в нормальных ситуациях должно производиться через фиксированное для каждого цвета число единиц времени (Tкр — для красного цвета, Tжёл — для жёлтого, Tзел — для зелёного). У светофора имеется счетчик таких единиц. При переключении светофора в счетчике устанавливается 0. Работа светофора усложняется необходимостью пропускать привилегированные машины (об их появлении на светофор поступает специальный сигнал) с минимальной задержкой, но при обеспечении безопасности пешеходов. Приведенная на Табл. 2 таблица решений описывает работу такого светофора и порядок движения у него в каждую единицу времени. Звездочка (\*) в этой таблице означает произвольное значение соответствующего условия.

### Операционная семантика

В операционной семантике алгебраического подхода к описанию семантики функций рассматривается следующий частный случай системы равенств (1):

 f1(x1, x2, ..., xk) = E1,

 f2(x1, x2, ..., xk) = E2,

 (3) .........……………

 fn(x1, x2, ... , xk) = En,

где в левых частях равенств явно указаны определяемые функции с формальными параметрами, включающими (для простоты) обозначения всех входных данных x1, x2, ..., xk, а правые части представляют собой выражения, содержащие, вообще говоря, вхождения этих функций с аргументами, задаваемыми некоторыми выражениями, зависящими от входных данных x1, x2, ..., xk.

Операционная семантика интерпретирует эти равенства как систему подстановок. Под подстановкой

| s E | | T

выражения (терма) T в выражение E вместо символа s (в частности, переменной) будем понимать переписывание выражения E с заменой каждого вхождения в него символа s на выражение T. Каждое равенств

fi(x1, x2, ..., xk) = Ei

задает в параметрической форме множество правил подстановок вида

|x1, x2, ..., xkfi(T1, T2, ..., Tk) -> Ei | |T1, T2, ..., Tk

где T1, T2, ..., TK — конкретные аргументы (значения или определяющие их выражения) данной функции. Это правило допускает замену вхождения левой его части в какое-либо выражение на его правую часть.

Интерпретация системы равенств (3) для получения значений определяемых функций в рамках операционной семантики производится следующим образом. Пусть задан набор входных данных (аргументов) d1, d2, ..., dk. На первом шаге осуществляется подстановка этих данных в левые и правые части равенств с выполнением там, где это возможно, предопределенных операций и с выписыванием получаемых в результате этого равенств. На каждом следующем шаге просматриваются правые части полученных равенств. Если правая часть является каким-либо значением, то оно и является значением функции, указанной в левой части этого равенства. В противном случае правая часть является выражением, содержащим вхождения каких-либо определяемых функций с теми или иными наборами аргументов. Если для такого вхождения соответствующая функция с данным набором аргументов имеется в левой части какого-либо из полученных равенств, то либо вместо этого вхождения подставляется значение правой части этого равенства, если оно уже вычислено, с выполнением, где это возможно, предопределённых операций. Либо не производится никаких изменений, если значение этой правой части ещё не вычислено. В том же случае, если эта функция с данным набором аргументов не является левой частью никакого из полученных равенств, то формируется (и дописывается к имеющимся) новое равенство. Оно получается из исходного равенства для данной функции с подстановкой в него вместо параметров указанных аргументов этой функции. Эти шаги осуществляются до тех пор, пока все определяемые функции не будут иметь вычисленные значения.

В качестве примера операционной семантики рассмотрим определение функции факториала F(n) = n! Она определяется следующей системой равенств:

F(0) = 1, F(n) = F(n–1)Чn.

Для вычисления значения F(3) осуществляются следующие шаги:

1-й шаг:

F(0) = 1,

F(3) = F(2)Ч3.

2-й шаг:

F(0) =1,

F(3) = F(2)Ч3,

F(2) = F(1)Ч2.

3-й шаг:

F(0) = 1,

F(3) = F(2)Ч3,

F(2) = F(1)Ч2,

F(1) = F(0)Ч1.

4-й шаг:

F(0) = 1,

F(3) = F(2)Ч3,

F(2) = F(1)Ч2,

F(1) = 1.

5-й шаг:

F(0) = 1,

F(3) = F(2)Ч3,

F(2) = 2,

F(1) = 1.

6-й шаг:

F(0) = 1,

F(3) = 3,

F(2) = 2,

F(1) = 1.

Значение F(3) на 6-ом шаге получено.

### Денотационная семантика

В денотационной семантике алгебраического подхода рассматривается также система равенств вида (3), которая интерпретируется как система функциональных уравнений, а определяемые функции являются некоторым решением этой системы. В классической математике изучению функциональных уравнений (в частности, интегральных уравнений) уделяется большое внимание и связано с построением достаточно глубокого математического аппарата. Применительно к программированию этими вопросами серьезно занимался Д. Скотт [3].

Основные идеи денотационной семантики проиллюстрируем на более простом случае, когда система равенств (5.3) является системой языковых уравнений:

 X1 = φ1,1  φ1,2  ...  φ1,k1,

 X2 = φ2,1  φ2,2  ...  φ2,k2,

(4) .....…………………………

 Xn= φn,1  φn,2  ...  φn,kn,

причем i-ое уравнение при ki = 0 имеет вид

Xi = 

Как известно, формальный язык — это множество цепочек в некотором алфавите. Такую систему можно рассматривать как одну из интерпретаций набора правил некоторой грамматики, представленную в форме Бэкуса-Наура (каждое из приведенных уравнений является аналогом некоторой такой формулы). Пусть фиксирован некоторый алфавит A = {a1, a2, …, am} терминальных символов грамматики, из которых строятся цепочки, образующие используемые в системе (4) языки. Символы X1, X2, ..., Xn являются метапеременными грамматики, здесь будут рассматриваться как переменные, значениями которых являются языки (множества значений этих метапеременных). Символы φi,j, i = 1, ..., n, j = 1, ..., kj, обозначают цепочки в объединенном алфавите терминальных символов и метапеременных:

φi,j  (A | { X1, X2, ..., Xn})\* .

Цепочка φi,j рассматривается как некоторое выражение, определяющее значение, являющееся языком (множеством цепочек в алфавите A). Такое выражение определяется следующим образом. Если значения X1, X2, ..., Xn заданы, то цепочка

φ = Z1 Z2 ... Zk, Zi(A | { X1, X2, ..., Xn }),

обозначает сцепление множеств Z1 Z2 ... Zk, причём вхождение в эту цепочку символа aj представляет множество из одного элемента {aj}. Это означает, что φ определяет множество цепочек

{ p1 p2 ... pk | pjZj, j = 1, ..., k},

причём цепочка

p1, p2, ..., pk

представляет собой последовательность выписанных друг за другом цепочек p1, p2, ..., pk. Таким образом, каждая правая часть уравнений системы (4) представляет собой объединение множеств цепочек.

Решением системы (4) является набор значений (языков)

L1, L2, ..., Ln

переменных X1, X2, ..., Xn, для которых все уравнения системы (4) превращаются в тождество.

Рассмотрим в качестве примера частный случай системы (4), состоящий из одного уравнения

X = a X  b X  c

с алфавитом A = {a, b, c}. Решением этого уравнения является язык

L = { φ c | φ{a, b}\*}.

Система (4) может иметь несколько решений. Так в рассмотренном примере помимо L решениями являются также

L1 = L  {φ a | φ{a, b}\*}

и

L2 = L  { φ b | φ{a, b}\*}.

В соответствии с денотационной семантикой в качестве определяемого решения системы (4) принимается наименьшее. Решение (L1, L2, ..., Ln) системы (4) называется наименьшим, если для любого другого решения (L′1, L′2, ..., L′n) выполняется

L1  L′1, L2  L′2, ..., Ln  L′n.

Так в рассмотренном примере наименьшим (а значит, определяемым денотационной семантикой) является решение L.

В качестве метода решения систем уравнений (3) и (4) можно использовать метод последовательных приближений. Сущность этого метода для системы (4) заключается в следующем. Обозначим правые части уравнений системы (4) операторами Ti(X1, X2, ..., Xn). Тогда система (4) примет вид

 X1= T1(X1, X2, ..., Xn),

 X2= T2(X1, X2, ..., Xn),

(5) ………………………

 Xn= Tn(X1, X2, ..., Xn).

В качестве начального приближения решения этой системы примем набор языков (L1[0], ..., Ln[0]) = (, , ..., ). Каждое следующее приближение определяется по формуле:

(L1[0], ..., Ln[0]) = (T1(L1[i–1], ..., Ln[i–1]), …………….. (Tn(L1[i–1], ..., Ln[i–1])).

Так как операции объединения и сцепления множеств являются монотонными функциями относительно отношения порядка Н, то этот процесс сходится к решению (L1, ..., Ln) системы (5), т.е.

(L1, ..., Ln)= (T1(L1, ..., Ln), ..., Tn(L1, ..., Ln))

и это решение является наименьшим. Это решение называют ещё наименьшей неподвижной точкой системы операторов

T1, T2, ..., Tn.

В рассмотренном примере этот процесс даёт следующую последовательность приближений:

L[0] = , L[1] = {c}, L[2]= {c, ac, bc},

L[3] = {c, ac, bc, aac, abc, bac, bbc},

…………………………………………

Этот процесс сходится к указанному выше наименьшему решению L.

### Аксиоматическая семантика

В аксиоматической семантике алгебраического подхода система (5) интерпретируется как набор аксиом в рамках некоторой формальной логической системы, в которой есть правила вывода и / или интерпретации определяемых объектов.

Для интерпретации системы (1) вводится понятие аксиоматического описания (S, E) — логически связанной пары понятий: S — сигнатура используемых в системе (1) символов функций f1, f2, ..., fm и символов констант (нульместных функциональных символов) c1, c2, ..., cm, а E — набор аксиом, представленный системой (1). Предполагается, что каждая переменная xi, i = 1, ..., k, и каждая константа cj, j =1, ..., l, используемая в E, принадлежит к какому-либо из типов данных t1, t2, ..., tr, а каждый символ fi, i =1, ..., m, представляет функцию, типа

ti1 \* ti2 \* ... \* tik → ti0.

Такое аксиоматическое описание получит конкретную интерпретацию, если будут заданы конкретные типы данных ti = t′i, i = 1, ..., r, и конкретные значения констант ci = c′i, i = 1, ..., l. В таком случае говорят, что задана одна конкретная интерпретация A символов сигнатуры S, называемая алгебраической системой

A = (t′1, ..., t′r, f ′1, ..., f ′r, с′1, ..., с′ r),

где f ′i, i = 1, ..., m, конкретная функция, представляющая символ fi. Таким образом, аксиоматическое описание (S, E) определяет класс алгебраических систем (частный случай: одну алгебраическую систему), удовлетворяющих системе аксиом E, т.е. превращающих равенства системы E в тождества после подстановки в них f ′i, i = 1, ..., m, и ci = c′i, i = 1, ..., l, вместо fi и ci соответственно.

В программировании в качестве алгебраической системы можно рассматривать, например, тип данных, при этом определяемые функции представляют операции, применимые к данным этого типа. Так К. Хоор построил аксиоматическое определение набора типов данных [4], которые потом Н. Вирт использовал при создании языка Паскаль.

В качестве примера рассмотрим систему равенств:

УДАЛИТЬ(ДОБАВИТЬ(m,d))=m,

ВЕРХ(ДОБАВИТЬ(m,d))=d,

УДАЛИТЬ(ПУСТ)=ПУСТ,

ВЕРХ(ПУСТ)=ДНО,

где УДАЛИТЬ, ДОБАВИТЬ, ВЕРХ — символы функций, а ПУСТ и ДНО — символы констант, образующие сигнатуру этой системы. Пусть D, D1 и М — некоторые типы данных, такие, что mM, dD, ПУСТM, ДНО D1, а функциональные символы представляют функции следующих типов:

УДАЛИТЬ: M → M,

ДОБАВИТЬ: M \* D → M,

ВЕРХ: M → D1.

Данная сигнатура вместе с указанной системой равенств, рассматриваемой как набор аксиом, образует некоторое аксиоматическое описание.

С помощью этого аксиоматического описания определим абстрактный тип данных, называемый магазином, задав следующую интерпретацию символов её сигнатуры: пусть D — множество значений, которые могут быть элементами магазина, D1 = D | {ДНО}, а M — множество состояний магазина,

M = {d1, d2, ..., dn | dniD, i = 1, ..., n, ni0},

ПУСТ = {}, ДНО — особое значение (зависящее от реализации магазина), не принадлежащее D. Тогда указанный набор аксиом определяют свойства магазина.

С аксиоматической семантикой связана логика равенств (эквациональная логика), изучаемая в курсе «Математическая логика». Эта логика содержит правила вывода из заданного набора аксиом других формул (равенств).

### Аксиоматическая семантика

Как уже отмечалось, функциональная спецификация представляет собой математически точное, но, как правило, не формальное описание поведения ПС. Однако, формализованное представление функциональной спецификации имеет ряд достоинств, главным из которых является возможность применять некоторые виды автоматизированного контроля функциональной спецификации.

Под языком спецификаций понимается формальный язык, предназначенный для спецификации функций. В нем используется ряд средств, позволяющих фиксировать синтаксис и выражать семантику описываемых функций. Различие между языками программирования и языками спецификации может быть весьма условным: если язык спецификаций имеет реализацию на компьютере, позволяющую как-то выполнять представленные на нем спецификации (например, с помощью интерпретатора), то такой язык является и языком программирования, может быть, и не позволяющий создавать достаточно эффективные программы. Однако, для языка спецификаций важно не эффективность выполнения спецификации (программы) на компьютере, а её выразительность. Язык спецификации, не являющийся языком программирования, может быть, тем не менее, полезен в процессе разработки ПС (для автоматизации контроля, тестирования и т.п.).

Язык спецификации может базироваться на каком-либо из рассмотренных нами методов описания семантики функций, а также поддерживать спецификацию функций для какой-либо конкретной предметной области.

#### ЛИТЕРАТУРА

1. В.Н. Агафонов. Спецификация программ: понятийные средства и их организация. — Новосибирск: Наука (Сибирское отделение), 1987. — c. 30-73.
2. Ian Sommerville. Software Engineering. — Addison-Wesley Publishing Company, 1992.
3. Д. Скотт. Теория решеток, типы данных и семантика. // Данные в языках программирования. — М.: Мир, 1982. — c. 25-53.
4. К. Хоор. О структурной организации данных // У. Дал, Э. Дейкстра, К. Хоор. Структурное программирование. — М.: Мир, 1975. — c. 98-197.

# ЗАМОК ДРАКОНА

Разделяй и властвуй!

Латинское изречение

## АРХИТЕКТУРА ПРОГРАММНОГО СРЕДСТВА

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

### Понятие архитектуры программного средства

Архитектура ПС — это его строение как оно видно (или должно быть видно) извне 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.

### Основные классы архитектур программных средств

Различают следующие основные классы архитектур программных средств [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 её 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 её 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 никак 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) слоёв;
* 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 слоёв;
* 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 [2]. Эта операционная система состоит из четырех слоёв (см. Рис. 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 устройств ввода-вывода.

КОМПЬЮТЕР

Прикладные программы

*3: Управление входными и выходными потоками данных*

*2: Обеспечение связи с консолью операора*

*1: Управление памятью*

*0: Диспетчеризация и синхронизация процессов*

Рис. 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 путём передачи друг другу некоторых сообщений.

Простейшей разновидностью такой архитектуры является конвейер, средства, для организации которого имеются в операционной системе UNIX [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. 2). Конвейер обрабатывает некоторый поток сообщений. Каждое сообщение этого потока поступает на ввод первой программе, которая, обработав его, передаёт переработанное сообщение 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 передаёт переработанное сообщение следующей программе, а последняя программа конвейера выводит результат работы всего конвейера (результирующее сообщение). Таким образом, в конвейере, состоящим из n программ, может одновременно находиться в обработке до n 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. 2. Конвейер параллельно действующих программ.

1-ая программа

Поток сообщений

Поток сообщений

2-ая программа

Поток сообщений

*n*-ая программа

Поток сообщений

Поток 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 её требованию. Сообщение, переданное какой-либо программой некоторому порту, уже не будет принадлежать этой программе (и использовать её ресурсы), но оно не будет принадлежать и никакой другой программе, пока в порядке очереди не будет передано какой-либо программе по её 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. 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. 3. Пример программной системы с портами сообщений.

*Программа* ***A***

*Программа* ***B***

*Программа* ***C***

**…………………**

*Порт* ***U***

*Сообщение 1*

*Сообщение 2*

**………………**

*Сообщение m*

*Порт* ***W***

*Сообщение 1*

*Сообщение 2*

**………………**

*Сообщение n*

Программные системы с портами сообщений могут быть как жесткой конфигурации, так и гибкой конфигурации. В системах с портами жесткой конфигурации с каждой программой могут быть жестко связаны один или несколько входных портов. 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 должна производиться ее предварительная настройка с помощью специальной программной компоненты, осуществляющая совмещение каждого выходного виртуального порта с каким-либо входным виртуальным портом на основании информации, задаваемой пользователем. 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.

### 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 слоёв 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 слоя (например, обеспечивающий параллельное выполнение этих процедур). В коллективе параллельно действующих программ для управления портами сообщений требуется специальная программная подсистема. Такие программные компоненты никаких внешних функций не выполняют — они реализуют функции, возникшие в результате разработки архитектуры программного средства. В связи с этим такие функции мы будем называть архитектурными.

### Контроль архитектуры программных средств

Для контроля архитектуры ПС используется смежный контроль и ручная имитация.

Смежный контроль архитектуры ПС сверху — это её контроль разработчиками внешнего описания: разработчиками спецификации качества и разработчиками функциональной спецификации. Смежный контроль архитектуры ПС снизу — это её 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.

#### ЛИТЕРАТУРА

1. Г.Майерс. Надежность программного обеспечения. — М.: Мир, 1980. — с. 78-91.
2. E.W. Dijkstra. The Structure of the THE-Multiprogramming // Communications of the ACM. — 1968, 11(5). — pp. 341-346.
3. М. Кристиан. Введение в операционную систему UNIX. — М.: Финансы и статистика, 1985. — с. 46-49.

# ЗАМОК ДРАКОНА

Разделяй и властвуй!

Латинское изречение

## 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.

### Цель модульного программирования

Приступая к разработке каждой программы ПС, следует иметь ввиду, что она, как правило, является большой системой, поэтому мы должны принять меры для её упрощения. Для этого такую программу разрабатывают по частям, которые называются программными модулями [1, 2]. А сам такой метод разработки программ называют модульным программированием [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 разделён от других модулей 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 «Замок дракона. Лекция 3 — Методы борьбы со сложностью»), и обеспечения независимости компонент системы, и использования 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).

### Основные характеристики программного модуля

Не всякий программный модуль способствует упрощению программы [2]. Выделить хороший с этой точки зрения модуль является серьезной творческой задачей. Для оценки приемлемости выделенного модуля используются некоторые критерии. Так, Хольт [4] предложил следующие два общих таких критерия:

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

Майерс [5] предлагает использовать более конструктивные характеристики программного модуля для оценки его приемлемости: размер модуля; прочность модуля; сцепление с другими модулями; рутинность модуля (независимость от предыстории обращений к нему).

Размер модуля измеряется числом содержащихся в нем операторов (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 [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 — они появились только в более поздних языках. Так в языке программирования Ада средством задания информационно прочного модуля является пакет [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 [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 [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, хранящего следы предыдущих обращений к нему. Майерс [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.

### 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, во-первых, синтаксическую спецификацию его входов, позволяющую построить на используемом языке программирования синтаксически правильное обращение к нему (к любому его входу), и, во-вторых, 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 её 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 [1, 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. На первый взгляд такой порядок разработки программы 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 текстов используемых им модулей — для этого достаточно, чтобы каждый используемый модуль был лишь специфицирован (в объёме, 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 её концептуальную целостность и формируется в процессе ее разработки. При восходящей разработке эта глобальная информация для модулей нижних уровней ещё 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 таком порядке разработки программы вся необходимая глобальная информация формируется своевременно, т.е. ликвидируется весьма неприятный источник просчетов при программировании модулей. Существенно облегчается и тестирование модулей, производимое при нисходящем тестировании программы. 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 все модули, к которым может обращаться головной модуль, заменяются их имитаторами (так называемые заглушки [5]). Каждый имитатор модуля представляется весьма простым программным фрагментом, сигнализирующим, в основном, о самом факте обращения к имитируемому модулю с необходимой для правильной работы программы обработкой значений его входных параметров (иногда с их распечаткой) и с выдачей, если это необходимо, заранее запасенного подходящего результата. 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 на сам этот модуль и 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 заменяется программированием достаточно простых имитаторов используемых в программе модулей. Кроме того, имитаторы удобно использовать для подыгрывания процессу подбора тестов путем задания нужных результатов, выдаваемых имитаторами.

Некоторым недостатком нисходящей разработки, приводящим к определенным затруднениям при её 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 её нужно развивать.

В рассмотренных методах восходящей и нисходящей разработок (которые мы будем называть классическими) модульная древовидная структура программы должна разрабатываться до начала программирования модулей. Однако такой подход вызывает ряд возражений: представляется сомнительным, чтобы до программирования модулей можно было разработать структуру программы достаточно точно и содержательно. На самом деле этого делать не обязательно. Например, модульная структура при конструктивном и архитектурном подходах к разработке программ [7] формируется в процессе программирования модулей.

Конструктивный подход к разработке программы представляет собой модификацию нисходящей разработки, при которой модульная древовидная структура программы формируется в процессе программирования модуля. Сначала программируется головной модуль, исходя из спецификации программы в целом, причем спецификация программы является одновременно и спецификацией ее головного модуля, так как последний полностью берёт 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) создаётся спецификация реализующего ее фрагмента программы, который в дальнейшем может быть представлен некоторым поддеревом модулей. Важно заметить, что здесь также ответственность за выполнение выделенной функции берёт 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 её головного модуля) формируется верхняя начальная часть дерева, например, такая, которая показана на рис. Рис. 1.

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. Рис. 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. Далее 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 создаётся в расчёте 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 путём 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.

В классическом методе нисходящей разработки рекомендуется сначала все модули разрабатываемой программы запрограммировать, а уж затем начинать нисходящее их тестирование [5]. 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. В этом методе каждый запрограммированный модуль начинают сразу же тестировать до перехода к программированию другого модуля.

Все эти методы имеют ещё различные разновидности в зависимости от того, в какой последовательности обходятся узлы (модули) древовидной структуры программы в процессе её разработки [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 [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, контроль над выходом 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 путём реализовать тот или иной вариант (сначала самый простейший) нормально действующей программы. В связи с этим такая разновидность конструктивной реализации получила название метода целенаправленной конструктивной реализации. Достоинством этого метода является то, что уже на достаточно ранней стадии создаётся работающий вариант разрабатываемой программы. Психологически это играет роль допинга, резко повышающего эффективность разработчика. Поэтому этот метод является весьма привлекательным.

Подводя итог сказанному, на Рис. 3 представлена общая схема классификации рассмотренных методов разработки структуры программы.

### Контроль структуры программы

Для контроля структуры программы можно использовать три метода [5]:

* статический контроль,
* смежный контроль,
* сквозной контроль.

Статический контроль состоит в оценке структуры программы с точки зрения, хорошо ли программа разбита на модули с учётом 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 характер осуществления этих методов контроля зависит от выбранного метода разработки структуры программы: при классическом подходе они применяются до начала программирования модулей, а при конструктивном и архитектурном подходах — в процессе программирования модулей (в подходящие моменты времени).

#### ЛИТЕРАТУРА

1. Дж. Хьюз, Дж. Мичтом. Структурный подход к программированию. — М.: Мир, 1980. — с. 29-71.
2. В. Турский. Методология программирования. — М.: Мир, 1981. — с.90-164.
3. Е.А. Жоголев. Технологические основы модульного программирования // Программирование, 1980, №2. — с.44-49.
4. R.C.Holt. Structure of Computer Programs: A Survey // Proceedings of the IEEE, 1975, 63(6). — p. 879-893.
5. Г.Майерс. Надежность программного обеспечения. — М.: Мир, 1980. — с. 92-113.
6. Я.Пайл. АДА — язык встроенных систем. М.: Финансы и статистика, 1984. — с. 67-75.
7. М. Зелковец, А. Шоу, Дж. Гэннон. Принципы разработки программного обеспечения. — М.: Мир, 1982, с. 65-71.
8. А.Л.Фуксман. Технологические аспекты создания программных систем. М.: Статистика, 1979. — с. 79-94.

**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 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 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 ввиду, что программа должна быть понятной не только компьютеру, но и человеку: и разработчик модуля, и лица, проверяющие модуль, и текстовики, 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. Впервые на это обратил внимание 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, посуществу, 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, 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, не снижая основных достоинств структурного программирования. Запутывают программу как раз "неструктурные" операторы перехода, особенно переход на оператор, расположенный 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. Иногда программирование модуля начинают с построения его блок-схемы, описывающей в общих чертах логику его работы. Однако современная технология программирования не рекомендует этого делать. Хотя блок-схемы позволяют весьма наглядно представить логику работы модуля, при их 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 может делаться 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 следующем шаге производится уточнение и детализация одного из понятий (будем называть его уточняемым), использованного (как правило, не формализованно) 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 выражены в конечном счете 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 последовательности операторов 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.

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).

|  |
| --- |
|  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.

|  |
| --- |
|  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, УДОВЛЕТВОРЯЮЩЕЙ 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 прочитывается 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.

**Литература к лекции 8.**

8.1. Г.Майерс. Надежность программного обеспечения. - М.: Мир, 1980. - С. 127-154.

8.2. Э.Дейкстра. Заметки по структурному программированию// У.Дал, Э.Дейкстра, К.Хоор. Структурное программирование. - М.: Мир, 1975. - С. 24-97.

8.3. Н.Вирт. Систематическое программирование. - М.: Мир, 1977. - С. 94-164.

**Лекция 9. ДОКАЗАТЕЛЬСТВО СВОЙСТВ ПРОГРАММ**

Понятие обоснования программ. Формализация свойств программ, триады Хоора. Правила для установления свойств оператора присваивания, условного и составного операторов. Правила для установления свойств оператора цикла, понятие инварианта цикла. Завершимость выполнения программы.

**9.1. Обоснования программ. Формализация свойств программ.**

Для повышения надежности программных средств весьма полезно снабжать программы дополнительной информацией, с использованием которой можно существенно повысить уровень контроля ПС. Такую информацию можно задавать в форме неформализованных или формализованных утверждений, привязываемых к различным фрагментам программ. Будем называть такие утверждения обоснованиями программы. Неформализованные обоснования программ могут, например, объяснять мотивы принятия тех или иных решений, что может существенно облегчить поиск и исправление ошибок, а также изучение программ при их сопровождении. Формализованные же обоснования позволяют доказывать некоторые свойства программ как вручную, так и контролировать (устанавливать) их автоматически.

Одной из используемых в настоящее время концепций формальных обоснований программ является использование так называемых триад Хоора. Пусть S - некоторый обобщенный оператор над информационной средой IS, P и Q - некоторые предикаты (утверждения) над этой средой. Тогда запись {P}S{Q} и называют триадой Хоора, в которой предикат P называют предусловием, а предикат Q - постусловием относительно оператора S. Говорят, что оператор (в частности, программа) S обладает свойством {P}S{Q}, если всякий раз, когда перед выполнением оператора S истинен предикат P, после выполнения этого оператора S будет истинен предикат Q.

Простые примеры свойств программ:

(9.1) {n=0} n:=n+1 {n=1},

(9.2) {n<m} n:=n+k {n<m+k},

(9.3) {n<m+k} n:=3\*n {n<3\*(m+k)},

(9.4) {n>0} p:=1; m:=1;

 ПОКА m /= n ДЕЛАТЬ

 m:=m+1; p:=p\*m

 ВСЕ ПОКА

 {p=n!}.

Для доказательства свойства программы S используются свойства простых операторов языка программирования (мы здесь ограничимся пустым оператором и оператором присваивания) и свойствами управляющих конструкций (композиций), с помощью которых строится программа из простых операторов (мы здесь ограничимся тремя основными композициями структурного программирования, см. Лекцию 8). Эти свойства называют обычно правилами верификации программ.

**9.2. Свойства простых операторов.**

Для пустого оператора справедлива

Теорема 9.1. Пусть P - предикат над информационной средой. Тогда имеет место свойство {P}{P}.

Доказательство этой теоремы очевидно: пустой оператор не изменяет состояние информационной среды (в соответствии со своей семантикой), поэтому его предусловие сохраняет истинность и после его выполнения.

Для оператора присваивания справедлива

Теорема 9.2. Пусть информационная среда IS состоит из переменной X и остальной части информационной среды RIS:

IS = {X, RIS}.

Тогда имеет место свойство

{Q(F(X, RIS), RIS)} X:= F(X, RIS) {Q(X, RIS)} ,

где F(X, RIS) - некоторая однозначная функция, Q - предикат.

Доказательство. Пусть до выполнения оператора присваивания был истинен предикат Q(F(X0, RIS0), RIS0), где {X0, RIS0} - некоторое произвольное состояние информационной среды IS, тогда после выполнения оператора присваивания будет истинен предикат Q(X, RIS), так как X получит значение F(X0, RIS0), а состояние RIS не изменяется данным оператором присваивания, и, следовательно, после выполнения этого оператора присваивания в этом случае

Q(X, RIS)=Q(F(X0, RIS0), RIS0).

В силу произвольности выбора состояния информационной среды теорема доказана.

Примером свойства оператора присваивания может служит пример 9.1.

**9.3. Свойства основных конструкций структурного программирования**.

Рассмотрим теперь свойства основных конструкций структурного программирования: следования, разветвления и повторения.

Свойства следования выражает следующая

Теорема 9.3. Пусть P, Q и R - предикаты над информационной средой, а S1 и S2 - обобщенные операторы, обладающие соответственно свойствами

{P}S{Q} и {Q}S2{R}.

Тогда для составного оператора

S1; S2<.blockquote>

имеет место свойство

{P} S1; S2 {R} .

Доказательство. Пусть для некоторого состояния информационной среды перед выполнением оператора S1 истинен предикат P. Тогда в силу свойства оператора S1 после его выполнения будет истинен предикат Q. Так как по семантике составного оператора после выполнения оператора S1 будет выполняться оператор S2, то предикат Q будет истинен и перед выполнением оператора S2. Следовательно, после выполнения оператора S2 в силу его свойства будет истинен предикат R, а так как оператор S2 завершает выполнение составного оператора (в соответствии с его семантикой), то предикат R будет истинен и после выполнения данного составного оператора, что и требовалось доказать.

Например, если имеют место свойства (9.2) и (9.3), то имеет

место и свойство

{n<m} n:=n+k; n:=3\*n {n<3\*(m+k)}.

Свойство разветвления выражает следующая

Теорема 9.4. Пусть P, Q и R - предикаты над информационной средой, а S1 и S2 - обобщенные операторы, обладающие соответственно свойствами

{P,Q} S1{R} и {`P,Q} S2 {R}.

Тогда для условного оператора

ЕСЛИ P ТО S1 ИНАЧЕ S2 ВСЕ ЕСЛИ

имеет место свойство

{Q} ЕСЛИ P ТО S1 ИНАЧЕ S2 ВСЕ ЕСЛИ {R} .

Доказательство. Пусть для некоторого состояния информационной среды перед выполнением условного оператора истинен предикат Q. Если при этом будет истинен также и предикат P, то выполнение условного оператора в соответствии с его семантикой сводится к выполнению оператора S1. В силу же свойства оператора S1 после его выполнения (а в этом случае - и после выполнения условного оператора) будет истинен предикат R. Если же перед выполнением условного оператора предикат P будет ложен (а Q, по-прежнему, истинен), то выполнение условного оператора в соответствии с его семантикой сводится к выполнению оператора S2. В силу же свойства оператора S2 после его выполнения (а в этом случае - и после выполнения условного оператора) будет истинен предикат R. Тем самым теорема полностью доказана.

Прежде чем переходить к свойству конструкции повторения следует отметить полезную для дальнейшего

Теорему 9.5. Пусть P, Q, P1 и Q1 - предикаты над информационной средой, для которых справедливы импликации

P1=>P и Q=>Q1,

и пусть для оператора S имеет место свойство {P}S{Q}.Тогда имеет место свойство {P1}S{Q1} .

Эту теорему называют еще теоремой об ослаблении свойств.

Доказательство. Пусть для некоторого состояния информационной среды перед выполнением оператора S истинен предикат P1. Тогда будет истинен и предикат P (в силу импликации P1=>P). Следовательно, в силу свойства оператора S после его выполнения будет истинен предикат Q, а значит и предикат Q1 (в силу импликации Q=>Q1). Тем самым теорема доказана.

Свойство повторения выражает следующая

Теорема 9.6. Пусть I, P, Q и R - предикаты над информационной средой, для которых справедливы импликации

P=>I и (I,`Q)=>R ,

и пусть S - обобщенный оператор, обладающий свойством {I}S{I}.

Тогда для оператора цикла

ПОКА Q ДЕЛАТЬ S ВСЕ ПОКА

имеет место свойство

{P} ПОКА Q ДЕЛАТЬ S ВСЕ ПОКА {R} .

Предикат I называют инвариантом оператора цикла.

Доказательство. Для доказательства этой теоремы достаточно доказать свойство

{I} ПОКА Q ДЕЛАТЬ S ВСЕ ПОКА {I,`Q}

(по теореме 9.5 на основании имеющихся в условиях данной теоремы импликаций). Пусть для некоторого состояния информационной среды перед выполнением оператора цикла истинен предикат I. Если при этом предикат Q будет ложен, то оператора цикла будет эквивалентен пустому оператору (в соответствии с его семантикой) и в силу теоремы 9.1 после выполнения оператора цикла будет справедливо утверждение (I,`Q). Если же перед выполнением оператора цикла предикат Q будет истинен, то оператор цикла в соответствии со своей семантикой может быть представлен в виде составного оператора S; ПОКА Q ДЕЛАТЬ S ВСЕ ПОКА

В силу свойства оператора S после его выполнения будет истинен предикат I, и возникает исходная ситуация для доказательства свойства оператора цикла: предикат I истинен перед выполнением оператора цикла, но уже для другого (измененного) состояния информационной среды (для которого предикат Q может быть либо истинен либо ложен). Если выполнение оператора цикла завершается, то применяя метод математической индукции мы за конечное число шагов, придем к ситуации, когда перед его выполнением будет справедливо утверждение (I,`Q). А в этом случае, как было доказано выше, это утверждение будет справедливо и после выполнения оператора цикла. Теорема доказана.

Например, для оператора цикла из примера (9.4) имеет место свойство

 {n>0, p=1, m=1} ПОКА m /= n ДЕЛАТЬ

 m:= m+1; p:= p\*m

 ВСЕ ПОКА {p= n!}.

Это следует из теоремы 9.6, так как инвариантом этого оператора цикла является предикат p=m! и справедливы импликации(n>0, p=1, m=1) => p=m! и (p=m!, m=n) => p=n!

**9.4. Завершимость выполнения программы.**

Одно из свойств программы, которое нас может интересовать, чтобы избежать возможных ошибок в ПС, является ее завершимость, т.е. отсутствие в ней зацикливания при тех или иных исходных данных. В рассмотренных нами структурированных программах источником зацикливания может быть только конструкция повторения. Поэтому для доказательства завершимости программы достаточно уметь доказывать завершимость оператора цикла. Для этого полезна следующая

Теорема 9.7. Пусть F - целочисленная функция, зависящая от состояния информационной среды и удовлетворяющая следующим условиям:

(1) если для данного состояния информационной среды истинен предикат Q, то ее значение положительно;

(2) она убывает при изменении состояния информационной среды в результате выполнения оператора S.

Тогда выполнение оператора цикла

ПОКА Q ДЕЛАТЬ S ВСЕ ПОКА завершается.

Доказательство. Пусть is - состояние информационной среды перед выполнением оператора цикла и пусть F(is)=k. Если предикат Q(is) ложен, то выполнение оператора цикла завершается. Если же Q(is) истинен, то по условию теоремы k>0. В этом случае будет выполняться оператор S один или более раз. После каждого выполнения оператора S по условию теоремы значение функции F уменьшается, а так как перед выполнением оператора S предикат Q должен быть истинен (по семантике оператора цикла), то значение функции F в этот момент должно быть положительно (по условию теоремы). Поэтому в силу целочисленности функции F оператор S в этом циклене может выполняться более k раз. Теорема доказана.

Например, для рассмотренного выше примера оператора циклаусловиям теоремы 9.7 удовлетворяет функция f(n, m)= n-m. Так как перед выполнением оператора цикла m=1, то тело этого цикла будет выполняться (n-1) раз, т.е. этот оператор цикла завершается.

**9.5. Пример доказательства свойства программы.**

На основании доказанных правил верификации программ можно доказывать свойства программ, состоящих из операторов присваивания и пустых операторов и использующих три основные композиции структурного программирования. Для этого анализируя структуру программы и используя заданные ее пред- и постусловия необходимо на каждом шаге анализа применять подходящее правило верификации. В случае применения композиции повторения потребуется подобрать подходящий инвариант цикла.

В качестве примера докажем свойство (9.4). Это доказательство будет состоять из следующих шагов.

(Шаг 1). n>0 => (n>0, p - любое, m - любое).

(Шаг 2). Имеет место

{n>0, p - любое, m - любое} p:=1 {n>0, p=1, m - любое}.

-- По теореме 9.2.

(Шаг 3). Имеет место

{n>0, p=1, m - любое} m:=1 {n>0, p=1, m=1}.

-- По теореме 9.2.

(Шаг 4). Имеет место

{n>0, p - любое, m - любое} p:=1; m:=1 {n>0, p=1, m=1}.

-- По теореме 9.3 в силу результатов шагов 2 и 3.

Докажем, что предикат p=m! является инвариантом цикла, т.е. {p=m!} m:=m+1; p:=p\*m {p=m!}.

(Шаг 5). Имеет место {p=m!} m:=m+1 {p=(m-1)!}.

-- По теореме 9.2, если представить предусловие в виде {p=((m+1)-1)!}.

(Шаг 6). Имеет место {p=(m-1)!} p:=p\*m {p=m!}.

-- По теореме 9.2, если представить предусловие в виде {p\*m=m!}.

(Шаг 7). Имеет место инвариант цикл

{p=m!} m:=m+1; p:=p\*m {p=m!}.

-- По теореме 9.3 в силу результатов шагов 5 и 6.

(Шаг 8). Имеет место

{n>0, p=1, m=1} ПОКА m /= n ДЕЛАТЬ

m:= m+1; p:= p\*m

ВСЕ ПОКА {p= n!}.

-- По теореме 9.6 в силу результата шага 7 и имея в виду, что (n>0, p=1, m= 1)=>p=m!; (p=m!, m=n)=>p=n!.

(Шаг 9). Имеет место

{n>0, p - любое, m - любое} p:=1; m:=1;

ПОКА m /= n ДЕЛАТЬ

m:= m+1; p:= p\*m

ВСЕ ПОКА {p= n!}.

-- По теореме 9.3 в силу результатов шагов 3 и 8.

(Шаг 10). Имеет место свойство (9.4) по теореме 9.5 в силу результатов шагов 1 и 9.

**Литература к лекции 9.**

9.1. С.А. Абрамов. Элементы программирования. - М.: Наука, 1982. С. 85-94.

9.2. М. Зелковец, А. Шоу, Дж. Гэннон. Принципы разработки программного обеспечения. - М.: Мир, 1982. С. 98-105.

**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. Указанный 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**.

Успех отладки в значительной степени предопределяет рациональная организация тестирования. При отладке 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: 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, 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 при использовании в качестве тестов только части этих наборов некоторые участки программ ПС могут не работать ни на каком тесте и, значит, 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.

Оптимальная стратегия проектирования тестов расположена внутри интервала между этими крайними подходами, но ближе к левому краю. Она включает проектирование значительной части тестов по спецификациям, исходя из принципов: 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 каждый особый случай или на каждую исключительную ситуацию, указанные в спецификациях, - хотя бы один тест. Но она требует также проектирования некоторых тестов и по текстам программ, исходя из принципа (как минимум): 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. Автономная отладка означает тестирование только какой-то части программы, входящей в ПС, с поиском и исправлением в ней фиксируемых при тестировании ошибок. Она фактически включает отладку каждого 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.**

В данном разделе даются общие рекомендации по организации отладки. 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. Этот феномен подтверждает важность раннего обнаружения ошибок и необходимость тщательного контроля принимаемых решений на каждом этапе разработки ПС.

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. Избегайте невоспроизводимых тестов, документируйте их пропуск через компьютер; детально изучайте результаты каждого теста.

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 модуля.**

При автономной отладке 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, построенная специально для тестирования отлаживаемого модуля. Эта программа лишь частично совпадает с отлаживаемой программой, кроме случая, когда отлаживается последний модуль отлаживаемой программы. По мере продвижения отладки программы все большую часть окружения очередного отлаживаемого модуля будут составлять уже отлаженные модули этой программы, а при отладке последнего модуля этой программы окружение отлаживаемого модуля будет целиком состоять из всех остальных (уже отлаженных) модулей отлаживаемой программы (без каких-либо) отладочных модулей, т.е. в этом случае тестироваться будет сама отлаживаемая программа. Такой процесс наращивания отлаживаемой программы отлаженными и отлаживаемыми 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 всегда будет содержать только один отладочный модуль (кроме случая, когда отлаживается последний модуль отлаживаемой программы), который будет головным в тестируемой программе и который 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, может осуществлять ввод 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 имитаторы всех модулей, к которым может обращаться отлаживаемый модуль, а также имитаторы тех модулей, к которым могут обращаться отлаженные модули отлаживаемой программы (включенные в это окружение), но которые еще не отлажены. Некоторые из этих имитаторов при отладке одного модуля могут изменяться для разных тестов.

На самом деле в окружении отлаживаемого модуля во многих случаях могут содержаться отладочные модули обоих типов по ниже следующим соображениям. Как восходящее, так и нисходящее тестирование имеет свои достоинства и свои недостатки.

К достоинствам восходящего тестирования относятся

простота подготовки тестов и

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 часто приходится составлять для разных тестов много ведущих отладочных модулей);

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, требует высокой квалификации исполнителя-тестовика, 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 позволяет при разумном подходе воспользоваться достоинствами как восходящего, так и нисходящего тестирования и в значительной степени нейтрализовать их недостатки. Этот эффект является проявлением более общего принципа: наибольшего технологического эффекта можно добиться, сочетая нисходящие и восходящие методы разработки программ ПС. Именно для поддержки этого метода и предназначен архитектурный подход к разработке программ (см. лекцию 7): слой квалифицированно разработанных и тщательно оттестированных модулей существенно облегчает реализацию семейства программ в соответствующей предметной области и их последующую модернизацию.

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 самым сильным 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 тест 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. Убедитесь по тексту модуля, что для каждого цикла существует тест, для которого тело цикла не выполняется, тест, для которого тело цикла выполняется один раз, и тест, для которого тело цикла выполняется максимальное число раз. Добавьте недостающие тесты.

Шаг 4. Проверьте по тексту модуля его чувствительность к отдельным особым значениям входных данных - все такие значения должны входить в тесты. 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. Но так как их может быть слишком много, то желательно бы протестировать хотя бы 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) внешней 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. следующую лекцию). 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. Точность, устойчивость, защищенность, временная эффективность, в какой-то мере - эффективность по памяти, эффективность по устройствам, расширяемость и, частично, независимость от устройств могут тестироваться. 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 неудобств применения ПС. Этот этап непосредственно предшествует подключению пользователя к завершению разработки ПС (тестированию 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 требований 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 тестирование производится с помощью контрольных задач - типовых задах, для которых известен результат решения. В тех случаях, когда разрабатываемое ПС должно придти на смену другому варианту ПС, который 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.

**Литература к лекции 10.**

10.1. Г. Майерс. Надежность программного обеспечения. - М.: Мир, 1980. - С. 171-262.

10.2. Д. Ван Тассел. Стиль, разработка, эффективность, отладка и испытание программ. - М.: Мир, 1985. - С. 179-295.

10.3. Дж. Хьюз, Дж. Мичтом. Структурный подход к программированию. - М.: Мир, 1980. - С. 254-268.

10.4. Дж. Фокс. Программное обеспечение и его разработка. - М.: Мир, 1985. - С. 227-241.

10.5. М. Зелковиц, А. Шоу, Дж. Гэннон. Принципы разработки программного обеспечения. - М.: Мир, 1982. - С. 105-116.

10.6. Ю.М. Безбородов. Индивидуальная отладка программ. - М.: Наука, 1982. - С. 9-79.

10.7. В.В. Липаев. Тестирование программ. - М.: Радио и связь, 1986. - С. 15-47.

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

10.9. Э. Дейкстра. Заметки по структурному программированию. //У. Дал, Э. Дейкстра, К. Хоор. Структурное программирование. - М.: Мир, 1975. - С. 7-13.

**Лекция 11. ОБЕСПЕЧЕНИЕ ФУНКЦИОНАЛЬНОСТИ И НАДЕЖНОСТИ ПРОГРАММНОГО СРЕДСТВА**

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

На предыдущих лекция 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 о 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, насколько эта спецификация реализована в данном 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 требуемого ПС должна быть рассчитана на практически полезное применение любыми пользователями, для которых оно предназначено. Поэтому главный принцип обеспечении функциональности такого ПС заключается в том, чтобы с самого начала разрабатывать ПС таким образом, как будто требуется ПС в полном объеме, до тех пор, пока разработчики не будут иметь дело 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).

В силу имеющихся в разрабатываемом ПС ошибок достигнутая при обеспечении его функциональности завершенность (в соответствии со спецификацией его качества) на самом деле может быть не такой, как ожидалось. Можно лишь говорить, что эта 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 (с учетом того, что проявление имеющейся в ПС ошибки является функцией исходных данных), которая пока еще ждет соответствующих теоретических исследований.

**11.3. Обеспечение точности программного средства.**

Обеспечение этого примитива 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 (так называемыми 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.**

Этот примитив качества обеспечивается на этапе спецификации качества принятием решения об использовании в разрабатываемом ПС какого-либо подходящего базового программного обеспечения или не использовании в нем никакого базового программного обеспечения. При этом приходится учитывать как его надежность, так и требуемые для его использования ресурсы. При повышенных требованиях к надежности разрабатываемого ПС надежность базового программного обеспечения, имеющегося в распоряжении разработчиков, может оказаться неудовлетворительной, поэтому от его использования приходиться отказываться, а реализацию его функций в требуемом объеме приходится включать в состав ПС. Аналогичные решения приходится принимать при жестких ограничениях на используемые ресурсы (по критерию эффективности ПС).

**11.5. Обеспечение устойчивости программного средства.**

Этот примитив качества обеспечивается с помощью так называемого защитного программирования. Вообще говоря, защитное программирование применяется для повышения надежности ПС при программировании модуля 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. В связи с этим в конец этого модуля включаются фрагменты второго рода - обработчики соответствующих исключительных ситуаций, которые 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, мягкую остановку управляемых ПС устройств во избежание их поломки при аварийном прекращении выполнения программы).

Применение защитного программирования модулей приводит 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 является не очень злободневной задачей (с учетом уровня достигнутой надежности компьютеров). Но все же полезно знать ее решение. Это обеспечивается организацией так называемых "двойных-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. Копия состояния изменяемой в этом процессе памяти каждой опорной точке записывается во вторичную память с некоторой контрольной суммой (числом, вычисляемым как некоторая функция от этого состояния) в том случае, когда будет считаться, что обработка данных от предыдущей опорной точки до данной (т.е. один "просчет") произведена правильно (без сбоев компьютера). Для того, чтобы это выяснить, производится два таких "просчета". После первого "просчета" вычисляется и запоминается указанная контрольная сумма, а затем восстанавливается состояние памяти для предыдущей опорной точки 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 для предыдущей опорной точки). 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 относится прежде всего к операционным системам или к программам, выполняющим частично их функции. Различают две разновидности этой защиты:

защита от отказов,

защита от злонамеренного влияния "чужой" программы.

При появлении мультипрограммного режима работы компьютера в его памяти может одновременно находится в стадии выполнения несколько программ, попеременно получающих управление в результате возникающих прерываний (так называемое квазипараллельное выполнение программ). Одна из таких программ (обычно: операционная система) занимается обработкой прерываний и управлением мультипрограммным режимом. В каждой из таких программ могут возникать отказы (проявляться ошибки), которые могут повлиять на выполнение функций другими программами. Поэтому управляющая программа (операционная система) должна обеспечить защиту себя и других программ от такого влияния. Для этого 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. Причем к привилегированным операциям относятся операции изменения защиты памяти и режима функционирования, а также доступа к внешней информационной среде. Начальное включение компьютера и любое прерывание должно автоматически включать привилегированный режим и отмену защиты памяти. В этом случае управляющая программа (операционная система) может полностью защитить себя от влияния других программ, если все точки передачи управления при начальном включении и прерываниях будут принадлежать этой программе, если она никакой другой программе не позволит работать в привилегированном режиме (при передаче управления любой другой программе будет включаться только рабочий режим) и если она полностью защитит 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, т.е. неотделима от самого компьютера. Наличие временного прерывания позволяет управляющей программе защититься от зацикливания в других программах (без такого прерывания она могла бы просто лишиться возможности управлять).

Защита от отказов "своей" программы обеспечивается надежностью этой программы, на что ориентирована вся технология программирования, обсуждаемая в настоящем курсе лекций.

Защита от ошибок пользователя (помимо ошибок входных данных, см. обеспечение устойчивости ПС) обеспечивается выдачей предупредительных сообщений о попытках изменить состояние внешней информационной среды с требованием подтверждения этих действий, а также возможностью восстановления состояния отдельных компонент внешней информационной среды. Последнее базируется на осуществлении архивирования изменений состояния внешней информационной среды.

Защита от несанкционированного доступа обеспечивается использованием секретных слов (паролей). В этом случае каждому пользователю предоставляются определенные информационные и процедурные ресурсы (услуги), для использования которых требуется предъявления ПС некоторого пароля, ранее зарегистрированного в ПС этим пользователем. Другими словами, пользователь как бы "вешает замок" на выделенные ему ресурсы, "ключ" от которого имеется только у этого пользователя. Однако, в отдельных случаях могут быть предприняты настойчивые попытки взломать такую защиту, если защищаемые ресурсы представляют для кого-то чрезвычайную ценность. Для такого случая приходится предпринимать дополнительные меры для защиты от взлома защиты.

Защита от взлома защиты связана с использованием в ПС специальных программистских приемов, затрудняющих преодоление защиты от несанкционированного доступа. Использование обычных паролей оказывается недостаточной, когда речь идет о чрезвычайно настойчивом стремлении (например, преступного характера) добиться доступа к ценной информации. Во-первых, потому, что информацию о паролях, которую использует ПС для защиты от несанкционированного доступа, "взломщик" этой защиты относительно легко может достать, если он имеет доступ к самому этому ПС. Во-вторых, используя компьютер, можно осуществлять достаточно большой перебор возможных паролей с целью найти подходящий для доступа к интересующей информации. Защититься от такого взлома можно следующим образом. Секретное слово (пароль) или просто секретное целое число X знает только владелец защищаемой информации, а для проверки прав доступа в компьютере хранится другое число Y=F(X), однозначно вычисляемое ПС при каждой попытке доступа к этой информации при предъявлении секретного слова. При этом функция F может быть хорошо известной всем пользователям ПС, однако она обладает таким свойством, что восстановление слова X по Y практически невозможно: при достаточно большой длине слова X (например, в несколько сотен знаков) для этого требуется астрономическое время. Такое число Y будем называть электронной (компьютерной) подписью владельца секретного слова X (а значит, и защищаемой информации).

Другая разновидность такой защиты связана с защитой сообщений, пересылаемых по компьютерным сетям, преднамеренных (или злонамеренных) искажений. Такое сообщения может перехватываться на "перевалочных" пунктах компьютерной сети и подменяться другим сообщением от автора перехваченного сообщения. Такая ситуация возникает прежде всего при осуществлении банковских операций с использованием компьютерной сети. Путем подмены такого сообщения, являющего распоряжением владельца банковского счета о выполнении некоторой банковской операции деньги с его счета могут быть переведены на счет "взломщика" защиты (своеобразный вид компьютерного ограбления банка). Защиту от такого взлома защиты можно осуществить следующим образом [11.4]. Наряду с функцией F, определяющей компьютерную подпись владельца секретного слова X, которую знает адресат защищаемого сообщения (если только ее владелец является клиентом этого адресата), в ПС определена другая функция Stamp, по которой отправитель сообщения должен вычислить число S=Stamp(X,R), используя секретное слово X и текст передаваемого сообщения R. Функция Stamp также считается хорошо известной всем пользователям ПС и обладает таким свойством, что по S практически невозможно ни восстановить число X, ни подобрать другое сообщение R с соответствующей компьютерной подписью. Само передаваемое сообщение (вместе со своей защитой) должно иметь вид:

R Y S ,

причем Y (компьютерная подпись) позволяет адресату установить истинность клиента, а S как бы скрепляет защищаемое сообщение Rс компьютерной подписью Y. В связи с этим будем называть число S электронной (компьютерной) печатью. В ПС определена еще одна функция Notary, по которой получатель защищаемого сообщения проверяет истинность передаваемого сообщения:

Notary(R,Y,S).

Эта позволяет однозначно установить, что сообщение R принадлежит владельцу секретного слова X.

Защита от защиты необходима в том случае, когда пользователь забыл (или потерял) свой пароль. Для такого случая должна быть предусмотрена возможность для особого пользователя (администратора ПС), отвечающего за функционирования системы защиты, производить временное снятие защиты от несанкционированного доступа для хозяина забытого пароля с целью дать ему возможность зафиксировать новый пароль.

**Литература к лекции 11.**

11.1. И.С. Березин, Н.П. Жидков. Методы вычислений, т.т. 1 и 2. - М.: Физматгиз, 1959.

11.2. Н.С. Бахвалов, Н.П. Жидков, Г.М.Кобельков. Численные методы. - М.: Наука, 1987.

11.3. Г. Майерс. Надежность программного обеспечения. - М.: Мир, 1980. С. 127-154.

11.4. А.Н. Лебедев. Защита банковской информации и современная криптография//Вопросы защиты информации, 2(29), 1995.

**Лекция 12. ОБЕСПЕЧЕНИЕ КАЧЕСТВА 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, обеспечения его наличия в ПС может потребовать своих подходов и методов разработки ПС или отдельных его частей. Кроме того, отмечалась также противоречивость критериев качества ПС 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. Но во всяком случае полезно 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

П-документированость ПС определяет состав пользовательской документации

В предыдущей лекции было уже рассмотрено обеспечение двух из пяти примитивов качества (устойчивость и защищенность), которые определяют легкость применения ПС.

П-документированность и информативность определяют состав и качество пользовательской документации (см. следующую лекцию).

Коммуникабельность обеспечивается созданием подходящего пользовательского интерфейса и соответствующей реализации исключительных ситуаций. В чем здесь проблема?

**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 эффективность ПС. При этом постоянно приходится разрешать противоречие между временной эффективностью и эффективностью по памяти. Поэтому весьма важно, чтобы в спецификации качества было явно указано количественное соотношение между показателями этих примитивов качества или хотя бы заданы количественные границы для одного из этих показателей. И все же разные программные модули по-разному влияют на эффективность ПС в целом: и по вкладу в общие затраты ПС по времени и памяти, и по влиянию на разные примитивы качества (одни модули могут сильно влиять на достижение временной эффективности 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:

сначала нужно разработать надежное ПС, а уж потом добиваться требуемой его эффективности в соответствии со спецификацией качества этого ПС [12.2, 12.3];

для повышения эффективности ПС используйте прежде всего оптимизирующий компилятор - это может обеспечить требуемую эффективность [12.3];

если достигнутая эффективность ПС не удовлетворяет спецификации его качества, то найдите самые критические модули с точки зрения требуемой эффективности ПС (в случае временнoй эффективности для этого потребуется получить распределение по модулям времени работы ПС путем соответствующих измерений во время выполнения ПС); эти модули и попытайтесь оптимизировать в первую очередь путем их ручной переделки [12.3];

не занимайтесь оптимизацией модуля, если этого не требуется для достижения требуемой эффективности ПС.

12.4. Обеспечение сопровождаемости.

С-документированность, информативность и понятность определяют состав и качество документации по сопровождению (см. следующую лекцию). Кроме того, относительно текстов программ (модулей) можно сделать следующие рекомендации.

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 [12.3];

используйте осмысленные (мнемонические) и устойчиво различимые имена (оптимальная длина имени - 4-12 литер, цифры - в конце), не используйте сходные имена и ключевые слова [12.2, 12.3];

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 [12.2]);

не бойтесь использовать не обязательные скобки (скобки обходятся дешевле, чем ошибки [12.3 ];

размещайте не больше одного оператора в строке; для прояснения структуры модуля используйте дополнительные пробелы (отступы) в начале каждой строки [12.2, 12.3];

избегайте трюков, т.е. таких приемов программирования, когда создаются фрагменты модуля, основной эффект которых не очевиден или скрыт (завуалирован), например, побочные эффекты функций [12.2].

Расширяемость обеспечивается созданием подходящего инсталятора.

Структурированность и модульность упрощают как понимание текстов программ, так и их модификацию.

12.5. Обеспечение мобильности.

**Литература к лекции 12.**

12.1. Ian Sommerville. Software Engineering. - Addison-Wesley Publishing Company, 1992. P.

12.2. Г.Майерс. Надежность программного обеспечения. - М.: Мир, 1980. С. 127-154.

12.3. Д.Ван Тассел. Стиль, разработка, эффективность, отладка и испытание программ. - М.: Мир, 1985. С. 8-44, 117-178.

12.4. Документация пользователя программного обеспечения/Стандарт ANSI/IEEE 1063-1987.

**Лекция 13. ДОКУМЕНТИРОВАНИЕ ПРОГРАММНЫХ СРЕДСТВ**

**13.1. Документация, создаваемая в процессе разработки программных средств.**

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

Планы, оценки, расписания. Эти документы создаются менеджерами для прогнозирования и управления процессами разработки и сопровождения.

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 данного 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 (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 должны быть проверены (протестированы) на соответствие программам ПС. Эти 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 применить данное ПС. Она необходима, если ПС предполагает какое-либо взаимодействие с пользователями. К такой документации относятся документы, которыми руководствуется пользователь при инсталяции ПС (при установке ПС с соответствующей настройкой на среду применения ПС), при применении ПС для решения своих задач и при управлении ПС (например, когда данное ПС взаимодействует 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 данное 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 документ 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 инсталяции ПС. Предназначено для системных администраторов. Он должен детально предписывать, как устанавливать системы в конкретной среде. Он должен содержать описание машинно-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 системных администраторов. Оно должно описывать сообщения, генерируемые, когда ПС взаимодействует с другими системами, и как реагировать 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 же 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). Команда разработчиков-сопроводителей должна будет изучать эту документацию, чтобы понять строение и процесс разработки модернизируемого ПС, и внести в эту документацию 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 (Requirements document).

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

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.

Документация второй группы содержит

Руководство по сопровождению ПС (system maintenance guide), которое описывает известные проблемы вместе с ПС, описывает, какие части системы являются аппаратно- и программно-зависимыми, и как развитие ПС принято в расчет в его строении (конструкции).

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 зафиксированы в базе данных управления конфигурацией.

**Литература к лекции 13.**

13.1. Ian Sommerville. Software Engineering. - Addison-Wesley Publishing Company, 1992. P.

13.2. ANSI/IEEE Std 1063-1988, IEEE Standard for Software User Documentation.

13.3. ANSI/IEEE Std 830-1984, IEEE Guide for Software Requirements Specification.

13.4. ANSI/IEEE Std 1016-1987, IEEE Recommended Practice for Software Design Description.

13.5. ANSI/IEEE Std 1008-1987, IEEE Standard for Software Unit Testing.

13.6. ANSI/IEEE Std 1012-1986, IEEE Standard for Software Verification and Validation Plans.

13.7. ANSI/IEEE Std 983-1986, IEEE Guide for Software Quality Assurance Planning.

13.8. ANSI/IEEE Std 829-1983, IEEE Standard for Software Test Documentation.

**Лекция 14. АТТЕСТАЦИЯ ПРОГРАММНОГО СРЕДСТВА**

Назначение аттестации программного средства. Испытания и оценка качества программного средства. Виды испытаний и методы оценки качества программного средства.

**14.1. Назначение аттестации программного средства.**

Аттестация ПС - это авторитетное подтверждение качества ПС [14.1]. Обычно для аттестации ПС создается представительная (аттестационная) комиссия из экспертов, представителей заказчика и представителей разработчика. Эта комиссия проводит испытания ПС с целью получения необходимой информации для оценки его качества. Под испытанием ПС мы будем понимать процесс проведения комплекса мероприятий, исследующих пригодность ПС для успешной его эксплуатации (применения и сопровождения) в соответствии с требованиями заказчика [14.2]. Этот комплекс включает проверку полноты и точности программной документации, изучение и обсуждение других ее свойств, а также необходимое тестирование программ, входящих в состав ПС, и, в частности, соответствия этих программ имеющейся документации.

На основе информации, полученной во время испытаний ПС, прежде всего должно быть установлено, что ПС выполняет декларированные функции, а также должно быть установлено, в какой степени ПС обладает декларированными примитивами и критериями качества. Таким образом, оценка качества ПС является основным содержанием процесса аттестации. Произведенная оценка качества ПС фиксируется в соответствующем решении аттестационной комиссии.

**14.2. Виды испытаний программного средства.**

Известны следующие виды испытаний ПС [14.2,14.3], проводимых с целью аттестации ПС:

испытания компонент ПС;

системные испытания;

приемо-сдаточные испытания;

полевые испытания;

промышленные испытания.

Испытания компонент ПС - это проверка (тестирование) работоспособности отдельных подсистем ПС. Проводятся только в исключительных случаях по специальному решению аттестационной комиссии.

Системные испытания ПС - это проверка (тестирование) работоспособности ПС в целом. Может включать те же виды тестирования, что и при комплексной отладке ПС (см. лекцию 10). Проводится по решению аттестационной комиссии, если возникают сомнения в качестве проведения отладки разработчиками ПС.

Приемо-сдаточные испытания являются основным видом испытаний при аттестации ПС. Именно с этих испытаний начинает работу аттестационная комиссия. Эти испытания начинаются с изучения представленной документации, в том числе, и документации по тестированию и отладке ПС. Если в документации отсутствуют достаточно полные результаты тестирования ПС, аттестационная комиссия может принять решение о проведении системных испытаний ПС или о прекращении процесса аттестации с рекомендацией разработчику провести дополнительное (более полное) тестирование ПС. Кроме того, во время этих испытаний могут выборочно пропускаться тесты разработчиков, а также контрольные задачи пользователей (см. лекцию 10) и дополнительные тесты, подготовленные комиссией для оценки качества аттестуемого ПС.

Полевые испытания ПС - это демонстрация ПС вместе с технической системой, которой управляет эта ПС, узкому кругу заказчиков в реальных условиях и осуществляется тщательное наблюдение за поведением ПС [12.3]. Заказчикам должна быть предоставлена возможность задания собственных контрольных примеров, в частности, с выходов в критические режимы работы технической системы, а также с вызовом в ней аварийных ситуаций. Это дополнительные испытания, проводимые по решению аттестационной комиссии только для некоторых ПС, управляющих определенными техническими системами.

Промышленные испытания ПС - это процесс передачи ПС в постоянную эксплуатацию пользователям. Представляет собой период опытной эксплуатации ПС (см. лекцию 10) пользователями со сбором информации об особенностях поведения ПС и ее эксплуатационных характеристиках. Это завершающие испытания ПС, которые проводятся по решению аттестационной комиссии, если на предшествующих испытаниях получена недостаточно полная или надежная информация для оценки качества аттестуемого ПС.

**14.3. Методы оценки качества программного средства.**

Оценка качества ПС по каждому из критериев сводится к оценке каждого из примитивов, связанных с этим критерием качества ПС, в соответствии с их конкретизацией, произведенной в спецификации качества этого ПС [12.4]. Методы оценки примитивов качества ПС можно разделить на четыре группы:

непосредственное измерение показателей примитива качества;

обработка программ и документации ПС специальными программными инструментами (процессорами);

тестирование программ ПС;

экспертная оценка на основании изучения программ и документации ПС.

Непосредственное измерение показателей примитива качества производится путем подсчета числа вхождений в тот или иной программный документ характерных единиц, объектов, конструкций и т.п., а также путем измерения времени работы различных устройств и объема занятой памяти ЭВМ при выполнении контрольных примеров. Например, некоторым показателем эффективности по памяти может быть число строк программы на языке программирования, а некоторым показателем эффективности по времени может быть время ответа на запрос. Использование каких-либо показателей для примитивов качества может определяться в спецификации качества ПС. Метод непосредственного измерения показателей примитива качества может сочетаться с использованием тестирования программ.

Для установления наличия у ПС некоторых примитивов качества могут использоваться определенные программные инструментальные средства. Такие программные инструменты обрабатывают тексты программ или программной документации с целью контроля каких-либо примитивов качества или получения некоторых показателей этих примитивов качества. Для оценки структурированности программ ПС, если они программировались на подходящем структурном диалекте базового языка программирования, достаточно было бы их пропустить через конвертер структурированных программ, осуществляющий синтаксический и некоторый семантический контроль этого диалекта и переводящий тексты этих программ на входной язык базового транслятора. Однако таким путем в настоящее время удается контролировать лишь небольшое число примитивов качества, да и то в редких случаях. В ряде случаев вместо программных инструментов, контролирующих качество ПС, полезнее применять инструменты, осуществляющие преобразование представления программ или программной документации. Таким, например, является форматер программ, приводящий тексты программ к удобочитаемому виду, - обработка текстов программ ПС таким инструментом может автоматически обеспечить наличие соответствующего примитива качества у ПС.

Для оценки некоторых примитивов качества ПС используется тестирование. К таким примитивам относится прежде всего завершенность ПС, а также его точность, устойчивость, защищенность и другие примитивы качества. В ряде случаев для оценки отдельных примитивов качества ПС тестирование применяется в сочетании с другими методами. Так для оценки качества документации по применению ПС (П-документированности) тестирование применяется в сочетании с экспертной оценкой этой документации. Если при комплексной отладке ПС было проведено достаточно полное тестирование, то эти же тесты могут быть использованы и при аттестации ПС. В этом случае аттестационная комиссия может воспользоваться протоколами тестирования, проведенного при комплексной отладки. Однако и в этом случае необходимо выполнить какие-либо новые тесты или хотя бы повторно некоторые старые. Если же тестирование при комплексной отладке будет признано недостаточно полным, то необходимо провести более полное тестирование. В этом случае может быть принято решение о проведении испытаний компонент или системных испытаний ПС, а также о возврате ПС разработчикам на доработку. Весьма важно, чтобы для оценки ПС по критерию легкости применения было проведено (во время отладки и аттестации ПС) полное тестирование по тестам, подготовленным на основании документации по применению, а по критерию сопровождаемости - по тестам, подготовленным по каждому из документов, предлагаемых для сопровождения ПС.

Для оценки большинства примитивов качества ПС в настоящее время можно применять только метод экспертных оценок. Этот метод заключается в следующем: назначается группа экспертов, каждый из этих экспертов в результате изучения представленной документации составляет свое мнение об обладании ПС требуемым примитивом качества, а затем голосованием членов этой группы устанавливается оценка требуемого примитива качества ПС. 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). При этом группа экспертов должна руководствоваться конкретизацией этого примитива и указанием о способе его оценки, сформулированными в спецификации качества аттестуемого ПС.

Литература к лекции 14.

14.1. Г.Майерс. Надежность программного обеспечния. - М.: Мир, 1980. - С. 174-175.

14.2. В.В.Липаев. Тестирование программ. - М.: Радио и связь, 1986. - С. 231-245.

14.3. Д.Ван Тассел. Стиль, разработка, эффективность, отладка и испытание программ. - М.: Мир, 1985. - С. 281-283.

14.4. Б.Шнейдерман. Психология программирования. - М.: Радио и связь, 1984. - С. 99-127.

**Лекция 15.ОЪЕКТНЫЙ ПОДХОД К РАЗРАБОТКЕ ПРОГРАММНЫХ СРЕДСТВ**

**15.1. Объекты и отношения в программировании. Сущность объектного подхода к разработке программных средств.**

Окружающий нас мир состоит из объектов и отношений между ними [13.1]. Объект воплощает некоторую сущность и имеет некоторое состояние, которое может изменяться со временем как следствие влияния других объектов, находящихся с данным в каком-либо отношении. Он может иметь внутреннюю структуру: состоять из других объектов, также находящихся между собой в некоторых отношениях. Исходя из этого можно построить иерархическое строение мира из объектов. Однако, при каждом конкретном рассмотрении окружающего нас мира некоторые объекты считаются неделимыми ("точечными"), причем в зависимости от целей рассмотрения такими (неделимыми) могут приниматься объекты разного уровня иерархии. Отношение связывает некоторые объекты: можно считать, что объединение этих объектов обладает некоторым свойством. Если отношение связывает n объектов, то такое отношение называется n-местным (n-арным). На каждом месте объединения объектов, которые могут быть связаны каким-либо конкретным отношением, могут находиться разные объекты, но вполне определенные (в этом случае говорят: объекты определенного класса). Одноместное отношение называется свойством объекта (соответствующего класса). Состояние объекта может быть изучено по значению свойств этого объекта или неявно по значению свойств объединений объектов, связываемых вместе с данным тем или иным отношением.

В процессе познания или изменения окружающего нас мира мы всегда принимаем в рассмотрение ту или иную упрощенную модель мира (модельный мир), в которую включаем некоторые из объектов и некоторые из отношений окружающего нас мира и, как правило, одного уровня иерархии. Каждый объект, имеющий внутреннюю структуру, может представлять свой модельный мир, включающий объекты этой структуры и отношения, которые их связывают. Таким образом, окружающий нас мир, можно рассматривать (в некотором приближении) как иерархическую структуру модельных миров.

В настоящее время в процессе познания или изменения окружающего нас мира широко используется компьютерная техника для обработки различного рода информации. В связи с этим применяется компьютерное (информационное) представление объектов и отношений. Каждый объект информационно может быть представлен некоторой структурой данных, отображающей его состояние. Свойства этого объекта могут задаваться непосредственно в виде отдельных компонент этой структуры, либо специальными функциями над этой структурой данных. N-местные отношения для N>1 можно представить либо в активной форме, либо в пассивной форме. В активной форме N-местное отношение представляется некоторым программным фрагментом, реализующим либо N-местную функцию (определяющую значение свойства соответствующего объединения объектов), либо процедуру, осуществляющую по состоянию представлений объектов, связываемых представляемым отношением, изменение состояний некоторых из них. В пассивной форме такое отношение может быть представлено некоторой структурой данных (в которую могут входить и представления объектов, связываемых этим отношением), интерпретируемую на основании принятых соглашений по общим процедурам, независящим от конкретных отношений (например, реляционная база данных). В любом случае представление отношения определяет некоторые действия по обработке данных.

При исследовании модельного мира пользователь может по-разному получать (или захотеть получать) информацию от компьютера. При одном подходе его могут интересовать получение информации об отдельных свойствах интересующих его объектов или результатах какого-либо взаимодействия между некоторыми объектами. Для этого он заказывает разработку того или иного ПС, выполняющего интересующие его функции, или некоторую информационную систему, способной выдавать информацию об интересующих его отношениях, используя при этом соответствующую базу данных. В начальный период развития компьютерной техники (при не достаточно высокой мощности компьютеров) такой подход к использованию компьютеров был вполне естественным. Именно он и провоцировал функциональный (реляционный) подход к разработке ПС, который был подробно рассмотрен в предшествующих лекциях. Сущность этого подхода состоит в систематическом использовании декомпозиции функций (отношений) для построения структуры ПС и текстов программ, входящих в него. При этом сами объекты, к которым применялись заказываемые и реализуемые функции, представлялись фрагментарно (в том объеме, который необходим для выполнения этих функций) и в форме, удобной для реализации этих функций. Тем самым не обеспечивалось цельного и адекватного компьютерного представления модельного мира, интересующего пользователя: отображение его на используемые ПС могло оказаться для пользователя достаточно трудоемкой задачей, попытки незначительного расширения объема и характера информации об интересующем пользователя модельном мире. получаемой от таких ПС, могло привести к серьезной их модернизации. Такой подход к разработке ПС поддерживается большинством используемых языков программирования, начиная от языков ассемблеров и процедурных языков (ФОРТРАН, Паскаль) до функциональных языков (ЛИСП) и языков логического программирования (ПРОЛОГ).

При другом подходе к исследованию модельного мира с помощью компьютера пользователя может интересовать наблюдение за изменением состояний объектов в результате их взаимодействий. Это требует достаточно цельного представления в компьютере интересующего пользователя объекта, а программные компоненты, реализующие отношения, в которых участвует этот объект, явно связываются с ним. Для реализации такого подхода приходилось строить программные средства, моделирующих процессы взаимодействия объектов (модельного мира). С помощью традиционных средств разработки это оказалось довольно трудоемкой задачей. Правда, появились языки программирования, специально ориентированные на такое моделирование, но это лишь частично упростило задачу разработки требуемых ПС. Наиболее полно отвечает решению этой задачи объектный подход к разработке ПС. Сущность его состоит в систематическом использовании декомпозиции объектов при построении структуры ПС и текстов программ, входящих в него. При этом функции (отношения), выполняемые таким ПС, выражались через отношения объектов разных уровней, т. е. их декомпозиция существенно зависела от декомпозиции объектов.

Говоря об объектном подходе следует также четко понимать о какого рода объектах идет речь: объектах модельного мира пользователя, об их информационном представлении, об объектах программы, с помощью которых строится ПС. Кроме того, следует различать собственно объекты (объекты "пассивные") и субъекты (объекты "активные").

**15.2. Объекты и субъекты в программировании.**

**15.3. Объектный и субъектный подходы к разработке программных средств.**

Декарт отмечал, что люди обычно имеют объектно-ориентированный взгляд на мир ([29] в [13.3]).

Считают, что объектно-ориентированного проектирование основано на принципах [13.3, стр. 31]:

выделение абстракций,

ограничение доступа,

модульность,

иерархия,

типизация,

параллельность,

устойчивость.

Но все это может применяться и при функциональном подходе.

Следует различать достоинства и недостатки общего объектного подхода и его частного случая - субъектно-ориентированного подхода.

Достоинства общего объективного подхода:

Естественное отображение реального мира на строение ПС (естественное восприятие человеком возможностей ПС, не нужно "выдумывать" строение ПС, а использовать естественные аналогии).

Использование достаточно содержательных структурных единиц ПС (объект как целостность неизбыточных ассоциаций, инфомационно-прочные модули).

Снижение трудоемкости разработки ПС за счет использования нового уровня абстракций (использование иерархии "непрограммных" абстракций при разработке ПС: классификация объектов реального мира, метод аналогий в природе) как новый уровень наследования.

**15.4. Объектный подход к разработке внешнего описания и архитектуры программного средства.**

Объектно-ориентированное проектирование - метод, использующий объектную декомпозицию; объектно-ориентированный подход имеет свою систему условных обозначений и предлагает богатый набор логических и физических моделей для проектирования систем высокой степени сложности. [13.3, стр. 30].

.....На объектный подход оказал объектно-ориентированный анализ (ООА). ООА направлен на создание моделей, более близких к реальности, с использованием объектно-ориентированного подхода; это методология, при которой требования формируются на основе понятий классов и объектов, составляющих словарь предметной области. [2, стр.42].

Особенности объектно-ориентированного программирования.

Объекты, классы, поведение объекта, свойства, события.

**Литература к лекции 15.**

15.1. К. Фути, Н. Судзуки. Языки программирования и схемотехника СБИС. - М.: Мир, 1988. С. 85-98.

15.2. Ian Sommerville. Software Engineering. - Addison-Wesley Publishing Company, 1992. P. ?-?

15.3. Г.Буч. Объектно-ориентированное проектирование с примерами применения: пер. с англ. - М.: Конкорд, 1992.

15.4. В.Ш.Кауфман. Языки программирования. Концепции и принципы. М.: Радио и связь, 1993.