Е. А. Жоголев

ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЯ

МОСКВА
НАУЧНЫЙ МИР
2004

УДК 681.142
ББК 22.18
Ж.78

Рецензенты:
Кузнецов С.Д., д.т.н., профессор
Машечкин И.В., д.ф.-м.н., профессор

Жоголев Е. А.
Ж.78 Технология программирования.
– М., Научный Мир, 2004, 216 с.

ISBN 5-89176-265-X

В основу данной книги положен курс лекций по технологии программирования, читавшихся автором в течение ряда лет студентам факультета Вычислительной математики и кибернетики МГУ. В ней обсуждается совокупность процессов, приводящая к созданию требуемого программного продукта (программного средства, ПС). Рассматриваются наиболее распространенные приёмы и методы, используемые в таких процессах, а также возникающие в них проблемы. Даются рекомендации по организации этих процессов и по решению конкретных возникающих в них задач. Отдельная глава посвящена инструментальным средствам, поддерживающим разработку программных продуктов.

Значительное внимание уделено понятию качества программного средства и обеспечению требуемого его качества, в частности, его надёжности (включая защиту информации). Большой интерес представляют также приведенные в конце книги толковый словарь основных используемых терминов, словарь англоязычных терминов и обширная библиография по данной тематике.

Книга будет полезна для программистов, приступающих к разработке больших программных систем, а также окажет поддержку студентам высших учебных заведений, изучающим технологию программирования.

РФФИ

Публикуется при финансовой поддержке Российского фонда фундаментальных исследований (проект 04-01-14056)

УДК 681.142
ББК 22.18

ISBN 5-89176-265-X

Евгений Андреевич Жоголев

Коллектив факультета Вычислительной математики и кибернетики МГУ им. М. В. Ломоносова, вся программистская общественность России понесли тяжелую утрату. 24 июня 2003 года ушел из жизни один из первых российских программистов, Заслуженный профессор Московского университета, Заслуженный работник высшей школы РФ, доктор физико-математических наук Евгений Андреевич ЖОГОЛЕВ.

Вся научная работа и жизнь Евгения Андреевича были неразрывно связаны с Московским университетом, где он трудился более 50 лет, сначала в Вычислительном центре МГУ, а затем на кафедре системного программирования факультета Вычислительной математики и кибернетики, и прошел путь от старшего лаборанта до заслуженного профессора. В годы появления первых отечественных ЭВМ Евгений Андреевич вместе с коллегами приложил много сил и творческой энергии для формирования в Вычислительном центре МГУ коллектива первых программистов. Его научные идеи и практические работы в ВЦ МГУ "вдохнули жизнь" в уникальные отечественные машины: были разработаны система стандартных подпрограмм на ЭВМ М-2, практически первый загрузчик подпрограмм; реализована стандартная составляющая программа (редактор связей) для ЭВМ "Стрела"; разработаны система команд и базовое программное обеспечение для малой троичной ЭВМ "Сетунь". Широко известны и признаны программистами работы Евгения Андреевича по проблемам модульного программирования, синтаксического анализа и методике построения синтаксически

управляемых трансляторов, технологии программирования. Его заслуги перед Отечеством отмечены государственными наградами.

Неоценимый вклад внес Евгений Андреевич в становление и совершенствование учебного процесса на факультете Вычислительной математики и кибернетики МГУ, опыт которого был широко использован во многих вузах страны. Учебник "Курс программирования", написанный Евгением Андреевичем совместно с Н. П. Трифоновым в 1964 году, был первым в стране стабильным учебником по программированию, выдержавшим в 60-е и 70-е годы 3 издания. Не одно поколение студентов училось основам программирования по этой книге. Работая в методологическом Совете факультета и являясь секретарем методсовета программистских кафедр, Евгений Андреевич всегда стремился сохранить лучшие традиции отечественной школы программистов, много сил и энергии приложил к созданию новых учебных программ и лекционных курсов. Широкой программистской общественности хорошо известны его регулярные публикации в журнале “Программирование”.

Принципиальность, доброжелательность, неиссякаемый оптимизм и любовь к спорту были присущи ему всегда. Многие из коллег и учеников Евгения Андреевича знали его как человека, еще со студенческих лет пишущего замечательные стихи, глубоко интересующегося проблемами русского языка и истории.

Память о Евгении Андреевиче Жоголеве, замечательном человеке и крупном ученом, навсегда сохранится в наших сердцах.

Декан факультета ВМиК МГУ,
академик РАН Е. И. Моисеев
Главный редактор
журнала "Программирование"
чл.-корр. РАН В. П. Иванников

Биографическая справка

Жоголев Евгений Андреевич родился 15 февраля 1930 г. в г. Сенгилей Ульяновской обл. Окончил механико-математический факультет МГУ (1952).

Кандидат физико-математических наук (1963), доктор физико-математических наук (1983). Профессор кафедры системного программирования факультета ВМиК МГУ (1989). Заслуженный профессор Московского университета (1997). Награжден медалями "За

трудовую доблесть" (1961), "За доблестный труд в ознаменование 100-летия со дня рождения В. И. Ленина" (1970), "Ветеран труда" (1985), "Москва 850" (1997). Награжден Большой серебряной медалью ВДНХ за участие в разработке малой цифровой вычислительной машины "Сетунь" (1962).

Работал в МГУ с 1952 г.: старший лаборант, младший научный сотрудник, инженер, старший инженер, заведующий отделом ВЦ МГУ; с 1986 г. преподавал на кафедре системного программирования – с 1989 г. в должности профессора.

Область научных интересов: системы программирования, технология программирования, инструментальные системы программирования. Участие в разработке системы стандартных программ на ЭВМ М-2 (1954–56). В рамках этой системы разработан загрузчик подпрограмм (1955). Разработана стандартная составляющая программа (редактор связей) для ЭВМ "Стрела" (1957–61). Предложены алгоритм синтаксического анализа и методика построения синтаксически управляемых трансляторов (1961–65). Сформулированы и развиты принципы многоязычной системы модульного программирования СИМПР (1964–87). Разработана и развивается концепция систем обосновательного гиперпрограммирования в качестве некоторого семейства инструментальных систем программирования (с 1974 г.).

Читал лекционные курсы: "Технология программирования", "Инструментальные системы программирования".

Подготовил 15 кандидатов наук и 1 доктора наук.

Автор более 100 научных и учебных публикаций, в том числе:

1. Е. А. Жоголев, Н. П. Трифонов. Курс программирования. -М.: Наука, 1964.

2. Е. А. Жоголев. Принципы построения многоязычной системы модульного программирования // Кибернетика, 1974, №4.

3. Е. А. Жоголев. Система обосновательного гиперпрограммирования // Программирование, 1993, №1.

4. Е. А. Жоголев. Объектная организация систем гиперпрограммирования // Программирование, 1997, №5.

5. Е. А. Жоголев. Лекции по технологии программирования: учебное пособие. -М.: изд. отдел ф-та ВМиК МГУ, 2001.

Предисловие

Как сказано в работе [39, с. 23], к началу третьего тысячелетия информатика стала чрезвычайно актуальной и популярной областью. В течение последних пятидесяти лет она является определяющей технологией нашего времени. Компьютеры проникли во все сферы человеческой деятельности и стали движущей силой экономического развития во всём мире. Постоянно появляются новые информационные технологии, а существующие очень быстро устаревают после своего возникновения.

Решающим фактором функционирования информационных технологий является программное обеспечение (ПО) компьютеров. Рынок программных продуктов вышел на первое место по объёму продаж. Однако разработка программного продукта является весьма трудоёмким и специфическим процессом и, как правило, осуществляется большими коллективами. Фирма (коллектив), разрабатывающая программный продукт, имеет достаточно специфическую организацию, а сама разработка ведётся поэтапно в определённой последовательности. В этом случае говорят, что разработка программного продукта ведётся по той или иной технологии программирования. Существует много различных таких технологий, которые приспособлены к тем или иным классам программных продуктов или к различным особенностям коллективов разработчиков. Тем не менее, в основе всех этих технологий лежит много общего, в связи с этим правомерно говорить о технологии программирования вообще. Именно этим вопросам и посвящена настоящая книга.

Понятие технологии в русском языке имеет ясное определение [36], однако понятие технологии программирования требует некоторого уточнения прежде всего из-за необходимости определения, что следует считать продуктом этой технологии. Кроме того, появление этого термина в русскоязычной научной литературе вызвано в значительной степени не всегда адекватным переводом иноязычной литературы по программированию, что привело к различным определениям (толкованиям) этого понятия. Это уточнение делается в первой главе книги. Тем не менее, уже сейчас можно сказать (в соответствии с общепринятым в русском языке пониманием термина «технология»), что предметом книги является изучение производственных процессов, приводящих к созданию требуемого программного продукта. В частности, обсуждаются вопросы, из каких производственных процессов состоит эта технология,

на каких принципах они строятся, какие методы и инструментальные средства используются в этих процессах. В книге излагается система понятий, на которой базируются конкретные технологии программирования, и содержание различных процессов этих технологий. Рассматриваются наиболее распространенные приёмы и методы, используемые в таких процессах, а также возникающие в них проблемы. Даются рекомендации по организации этих процессов и по решению конкретных возникающих в них задач. В конце книги обсуждается вопрос о компьютерной поддержке разработки программных продуктов, даётся обзор инструментальных программных средств, используемых в этой разработке. Хотя в книге и не излагается конкретная технология программирования, предложенный материал позволит ориентироваться во многих существующих и вновь возникающих конкретных технологиях программирования, а также легко осваивать выбранную такую технологию.

Настоящая книга базируется на издании лекций [22, 23], читавшихся автором в течение ряда лет на факультете Вычислительной математики и кибернетики МГУ. По содержанию она полностью соответствует программе курса «Технология программирования», утверждённой для студентов программистских кафедр. От читателя требуются знания в рамках вводных курсов лекций по программированию, архитектуре компьютеров и операционным системам для них, а также некоторый опыт составления относительно небольших программ и практической работы на компьютерах.

При написании настоящей книги пришлось использовать информацию из самых разнообразных областей программирования. Некоторые из них базировались на автономной системе понятий со своей независимой терминологией. В связи с этим пришлось проделать определённую работу по созданию единой терминологии для используемой системы понятий. Эта работа усложнялась ещё и тем, что многие термины появились в нашей стране в результате перевода соответствующих англоязычных терминов, причём этот перевод часто не учитывал специфику системы понятий, используемых в терминологии программирования. Даже в простых случаях не адекватный перевод термина иногда создавал у русскоязычных читателей искажённое представление о смысле соответствующего понятия. Так, например, при переводе английского слова «plan» русским словом «план» появился термин «план тестирования», весьма не точно отражающий смысл соответствующего понятия, тогда как у этого английского слова есть другой (менее очевидный) пе-

ревод русским словом «схема», а термин «схема тестирования» уже совершенно точно отражает суть этого понятия. Более сложная ситуация сложилась с термином «design». Широко используемый в технологии программирования его перевод русскоязычным термином «проектирование» совершенно неприемлем из-за особенностей программного продукта и специфики его разработки (см. гл. 3). Получается, что мы начинаем «проектировать» программный продукт после этапа разработки его спецификации (подробного описания как «черного ящика»). С другой стороны, творческий процесс продолжается и на этапе кодирования программ, т. е. можно считать, что проектом программы является сама программа. Кроме того, с технологией программирования связан другой термин «project», с которым связан, например, термин «управление проектом», так что в русскоязычных текстах по технологии программирования не всегда ясно, о каком проектировании идет речь. Мы используем для термина «design» русскоязычный термин «конструирование».

Книга будет полезна для программистов, приступающих к разработке больших программных систем, а также для студентов высших учебных заведений, изучающих технологию программирования.

Автор пользуется случаем выразить благодарность всем преподавателям факультета Вычислительной математики и кибернетики МГУ, способствовавшим формированию настоящей книги. Особенно автор признателен доценту кафедры системного программирования Е. А. Кузьменковой, которая неоднократно читала лекции по учебному пособию, лежащему в основе данной книги, и сделала много ценных замечаний, учтенных автором в окончательном варианте текста книги.

Е. А. Жоголев

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

Программистский фольклор

Глава 1
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. Надёжность программного средства. Технология программирования как технология разработки надёжных программных средств. Технология программирования и информатизация общества.

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

Целью программирования является описание процессов обработки данных. Согласно IFIP [14]: *данные* (*data*) – это представление фактов и идей в формализованном виде, пригодном для передачи и переработке в некоем процессе, а *информация* (*information*) – это смысл, который придаётся данным при их представлении. *Обработка данных* (*data processing*) – это выполнение систематической последовательности действий с данными. Данные представляются и хранятся на так называемых *носителях данных*. Совокупность носителей данных, используемых при какой-либо обработке данных, будем называть *информационной средой*. Набор всех данных, содержащихся в какой-либо момент в информационной среде, будем называть *состоянием* этой информационной среды. *Процесс обработки данных* можно определить как последовательность сменяющих друг друга состояний некоторой информационной среды.

Описать процесс обработки данных – значит определить последовательность состояний заданной информационной среды. Если мы хотим, чтобы по заданному описанию требуемый процесс порождался автоматически на каком-либо компьютере, необходимо, чтобы это опи-

сание 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 (programming language), с которого она автоматически переводится на язык соответствующего компьютера с помощью другой программы, называемой транслятором (translator). Человеку (программисту), прежде чем составить программу на удобном для него языке программирования, приходится проделывать большую подготовительную работу по уточнению постановки задачи, выбору метода её 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 её как-то фиксировать в виде отдельных документов (часто не формализованных, рассчитанных только для восприятия человеком).

Обычно программы разрабатываются в расчёте на то, чтобы ими могли пользоваться люди, не участвующие в их разработке (их называют пользователями). Для освоения программы пользователем помимо её текста требуется определённая дополнительная документация. Программа или логически связанная совокупность программ на носителях данных, снабжённая 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 её выполнения, а также понять, что означают получаемые результаты (или каков эффект выполнения этой программы). Кроме того, программная документация помогает разобраться в самой программе, что необходимо, например, при ее модификации.

1.2. Неконструктивность понятия правильной программы

Таким образом, можно считать, что продуктом технологии программирования является ПС, содержащее программы, выполняющие требуемые функции. Здесь под «программой» часто понимают правильную программу, т. е. программу, не содержащую ошибок. Однако

понятие ошибки в программе трактуется в среде программистов неоднозначно. Согласно Майерсу [35, с. 10–13] будем считать, что в программе имеется ошибка, если она не выполняет того, что разумно ожидать от неё пользователю. «Разумное ожидание» пользователя формируется на основании документации по применению этой программы. Следовательно, понятие ошибки в программе является существенно не формальным. В ПС программы и документация взаимно увязаны, образуют некоторую целостность. Поэтому правильнее говорить об ошибке не в программе, а в ПС в целом: будем считать, что в ПС имеется ошибка (software error), если оно не выполняет того, что разумно ожидать от него пользователю. В частности, разновидностью ошибки в ПС является несогласованность между программами ПС и документацией по их применению. В работе [65] выделяется в отдельное понятие частный случай ошибки в ПС, когда программа не соответствует своей функциональной спецификации (описанию, разрабатываемому на этапе, предшествующем непосредственному программированию). Такая ошибка в указанной работе называется дефектом программы. Однако выделение такой разновидности ошибки в отдельное понятие вряд ли оправданно, так как причиной ошибки может оказаться сама функциональная спецификация, а не программа.

Так как задание на разработку ПС обычно формулируется не формально, а также из-за того, что понятия ошибки в ПС не формализовано, то нельзя доказать формальными методами (математически) правильность ПС. Нельзя показать правильность ПС и тестированием: как указал Дейкстра [16], 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. Надёжность программного средства

Альтернативой правильного ПС является надёжное ПС. Надёжность (reliability) ПС – это его способность безотказно выполнять определённые функции при заданных условиях в течение заданного периода времени с достаточно большой вероятностью [58]. При этом под отказом ПС понимают проявление в нём ошибки [35, с. 10–13]. Таким образом, надёжное ПС не исключает наличия в нём 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 путём тестирования, а также при практическом применении. Таким образом, фактически мы можем разрабатывать лишь надёжные, а не правильные ПС.

ПС может обладать различной степенью надёжности. Как измерять эту степень? Так же как в технике, степень надёжности можно характеризовать [35, с. 10–13] вероятностью работы ПС без отказа в течение определённого 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 надёжности 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 надёжности ПС иногда используют дополнительные показатели, учитывающие стоимость (вред) для пользователя каждого отказа.

1.4. Технология программирования как технология разработки надёжных программных средств

В соответствии с обычным значением слова «технология» [36] 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 неё 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, компьютеров (в этом случае будем говорить о компьютерной технологии программирования).

В литературе имеются и другие, несколько отличающиеся, определения технологии программирования. Эти определения обсуждаются в работе [19]. 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 [19, с. 9–16]. Именно программной инженерии (software engineering) посвящена упомянутая работа [65]. Главное различие между технологией

бота [65]. 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 непосредственно используются в этих процессах (их применение и образуют технологические процессы). Тогда как в программной инженерии изучаются различные методы и инструментальные средства разработки ПС с точки зрения достижения определённых целей – эти методы и средства могут использоваться в разных технологических процессах (и в разных технологиях программирования).

Не следует также путать технологию программирования с методологией программирования [42]. 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, с. 25] методология программирования определяется как совокупность механизмов, применяемых в процессе разработки программного обеспечения и объединенных одним общим философским подходом).

Имея в виду, что надёжность является неотъемлемым атрибутом ПС, мы будем рассматривать технологию программирования как технологию разработки надёжного ПС. Это означает, что

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

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

1.5. Информатизация общества и технология программирования

Технология программирования играла разные роли на разных этапах развития программирования. По мере повышения мощности компьютеров и развития средств и методологии программирования росла и сложность решаемых на компьютерах задач, что привело к повышенному вниманию к технологии программирования. Резкое удешевление стоимости компьютеров и, в особенности, стоимости хранения информации на компьютерных носителях привело к широкому внедрению компьютеров практически во все сферы человеческой деятельности, что существенно изменило направленность технологии программирования. Человеческий фактор стал играть в ней решающую роль. Сформировалось достаточно глубокое понятие качества ПС, причем предпочтение стало отдаваться не столько эффективности ПС, сколько удобству работы с ним для пользователей (не говоря уже о его надёжности). Широкое использование компьютерных сетей привело к интенсивному развитию распределённых 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. Начался этап глубокой и полной информатизации (компьютеризации) человеческого общества. Всё это ставит перед технологией программирования новые и достаточно трудные проблемы.

Сделаем краткую характеристику развития программирования по десятилетиям.

В пятидесятые годы мощность компьютеров (первого поколения) была невелика, а программирование для них велось, в основном, в машинном коде. Решались, главным образом, научно-технические задачи (счет по формулам), задание на программирование содержало, как правило, достаточно точную постановку задачи. Использовалась интуитивная технология программирования: почти сразу приступали к составлению программы по заданию, при этом часто задание несколько раз изменялось (что сильно увеличивало время и без того итерационного процесса составления программы), минимальная документация оформлялась уже после того, как программа начинала работать. Тем не менее, именно в этот период родилась фундаментальная для технологии программирования концепция модульного программирования [20], ориен-

тированная на преодоление трудностей программирования в машинном коде. Появились первые языки программирования высокого уровня, из которых только ФОРТРАН пробился для использования в следующие десятилетия.

В шестидесятые годы можно было наблюдать бурное развитие и широкое использование языков программирования высокого уровня (АЛГОЛ 60, ФОРТРАН, КОБОЛ и др.), значение которых в технологии программирования явно преувеличивалось. Надежда на то, что эти языки решат все проблемы, возникающие в процессе разработки больших программ, не оправдалась. В результате повышения мощности компьютеров и накопления опыта программирования на языках высокого уровня быстро росла сложность решаемых на компьютерах задач, в результате чего обнаружилась ограниченность языков, проигнорировавших модульную организацию программ. И только ФОРТРАН, бережно сохранивший возможность модульного программирования, гордо прошествовал в следующие десятилетия (все его ругали, но его пользователи отказаться от его услуг не могли из-за накопления грандиозного 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 [16]. Это было уже началом серьёзных размышлений над методологией и технологией программирования. Появление в компьютерах второго поколения прерываний привело к развитию мультипрограммирования и созданию больших программных систем. Широко стала использоваться коллективная разработка, которая поставила ряд серьёзных технологических проблем [8].

В семидесятые годы получили широкое распространение информационные системы и базы данных. К середине 70-х годов стоимость хранения одного бита информации на компьютерных носителях стала меньше, чем на обычных (ранее использовавшихся) носителях. Это резко повысило интерес к компьютерным системам хранения данных. Началось интенсивное развитие технологии программирования [21, 35, 42, 48, 61], прежде всего, в следующих направлениях:

- обоснование и широкое внедрение нисходящей разработки и структурного программирования;
- развитие абстрактных типов данных и модульного программирования (в частности, возникновение идеи разделения спецификации и

реализации модулей и использование модулей, скрывающих структуры данных);

- исследование проблем обеспечения надёжности и мобильности ПС;
- создание методики управления коллективной разработкой ПС;
- появление инструментальных программных средств (программных инструментов) поддержки технологии программирования.

Восьмидесятые 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 [7, 30, 49, 58, 63]. Появляются языки программирования (например, Ада), учитывающие требования технологии программирования [26]. Развиваются методы и языки спецификации ПС [2, 41]. Начинается бурный процесс стандартизации технологических процессов и, прежде всего, документации, создаваемой в этих процессах [34]. Выходит на передовые позиции объектный подход к разработке ПС [9]. Создаются различные инструментальные среды разработки и сопровождения ПС [65]. Развивается концепция компьютерных сетей.

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

Вопросы к главе 1

1.1. Что такое *информационная среда обработки данных*?
1.2. Что такое *программное средство (ПС)*?
1.3. Что такое *ошибка в ПС*?
1.4. Что такое *надёжность ПС*?
1.5. Что такое *технология программирования*?

Человеку свойственно ошибаться.
Сенека

Глава 2
ИСТОЧНИКИ ОШИБОК В ПРОГРАММНЫХ СРЕДСТВАХ

Интеллектуальные возможности человека, используемые при разработке программных систем. Понятия о простых и сложных системах, о малых и больших системах. Неправильный перевод информации из одного представления в другое – основная причина ошибок при разработке программных средств. Модель перевода и источники ошибок. Основные пути борьбы с ошибками.

2.1. Интеллектуальные возможности человека

Дейкстра [16] выделяет три интеллектуальные возможности человека, используемые при разработке ПС:

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

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

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

действия между ее элементами. В силу ограниченности человека к перебору будем различать простые и сложные системы [21]. Под простой будем понимать такую систему, в которой человек может уверенно перебирать все пути взаимодействия между её элементами, а под сложной будем понимать такую систему, в которой он этого делать не в состоянии. Между простыми и сложными системами нет четкой границы, поэтому можно говорить и о промежуточном классе систем: к таким системам относятся программы, о которых программистский фольклор утверждает, что «в каждой отлаженной программе имеется хотя бы одна ошибка».

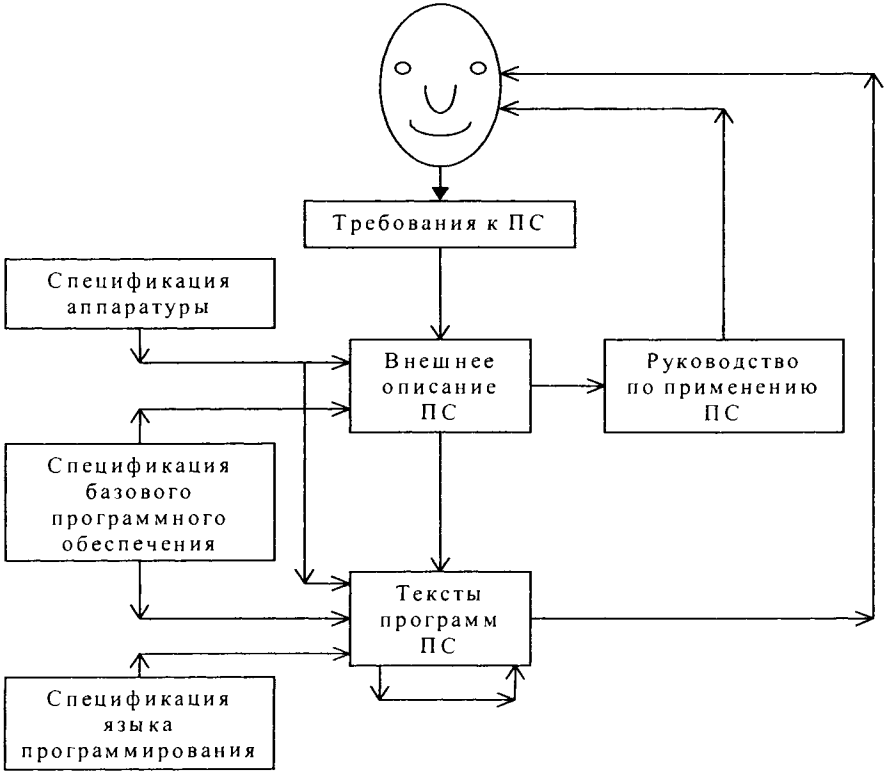


Рис. 2.1. Грубая схема разработки и применения ПС

При разработке ПС мы не всегда можем уверенно знать о всех связях между её элементами из-за возможных ошибок. Поэтому полезно

уметь оценивать сложность системы по числу её элементов, а именно по числу потенциальных путей взаимодействия между её элементами, т. е. n!, где n – число её элементов. Систему назовем малой, если $n < 7$ ($6! = 720 < 1000$), систему назовем большой, если $n > 7$. При $n=7$ имеем промежуточный класс систем. Малая система всегда проста, а большая может быть как простой, так и сложной. Человек, создавший большую систему (систему требований, систему управления, программную систему и т. п.), рискует увязнуть в переборе большого числа вариантов. 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.

2.2. Неправильный перевод информации как причина ошибок в программных средствах

При разработке и использовании ПС мы многократно имеем дело [35, с. 22–28] с преобразованием (переводом) информации из одной формы в другую (см. рис. 2.1). Заказчик формулирует свои потребности в ПС в виде некоторых требований. Исходя из этих требований, разработчик создает внешнее описание ПС, используя при этом спецификацию (описание) заданной аппаратуры и, возможно, спецификацию базового программного обеспечения. На основании внешнего описания и спецификации языка программирования создаются тексты программ ПС на этом языке. По внешнему описанию ПС разрабатывается также и пользовательская документация. Текст каждой программы является исходной информацией при любом ее преобразовании, в частности, при исправлении в ней ошибки.

Пользователь на основании документации выполняет ряд действий для применения ПС и осуществляет интерпретацию получаемых ре-

зультатов. Везде здесь, а также в ряде других процессах разработки ПС, имеет место указанный перевод информации.

На каждом из этих этапов перевод информации может быть осуществлён неправильно. Возникнув на одном из этапов разработки ПС, ошибка в представлении информации преобразуется в новые ошибки результатов, полученных на последующих этапах разработки, и, в конечном счёте, окажется в ПС.

2.3. Модель перевода

Чтобы понять природу ошибок при переводе рассмотрим модель [35, с. 22–28], изображённую на рис. 2.2. На ней человек осуществляет перевод информации из представления A в представление B. При этом он совершает четыре основных шага перевода:

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

На каждом из этих шагов человек может совершить ошибку разной природы. На первом шаге способность человека «читать между строк» (способность, которая часто оказывается полезной, позволяя ему понимать текст, содержащий неточности или даже ошибки) может стать причиной ошибки в ПС. Ошибка возникает в том случае, когда при чтении документа 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 информацию он запомнит 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 шаге

стремление человека быстрее зафиксировать информацию часто приводит к тому, что представление этой информации оказывается неточным, создавая ситуацию для последующей неоднозначной ее интерпретации.

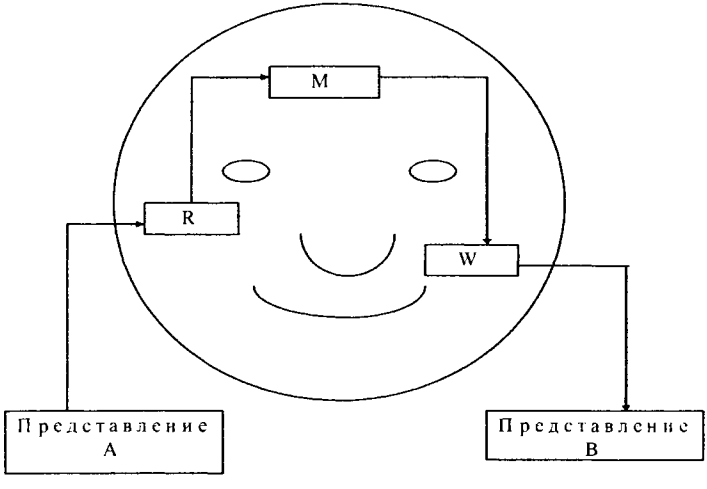


Рис. 2.2. Модель перевода

2.4. Основные пути борьбы с ошибками

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

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

Вопросы к главе 2

2.1. Что такое *простая и сложная системы*?
2.2. Что такое *малая и большая системы*?

2.3. Перечислите основные шаги, осуществляемые человеком при переводе информации из одного представления в другое, и какого рода ошибки он может совершать на этих шагах?

Лучшее – враг хорошего.
Народная мудрость

Глава 3
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 программных средств. Обеспечение точности перевода. Преодоление барьера между пользователем и разработчиком. Обеспечение контроля правильности принимаемых решений.

3.1. Специфика разработки программных средств

Разработка программных средств имеет ряд специфических особенностей [22].

- 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, как известно, переход от неформального к формальному существенно неформален.
- Разработка ПС носит *творческий характер* (на каждом шаге приходится делать какой-либо выбор, принимать какое-либо решение), а не сводится к выполнению какой-либо последовательности регламентированных действий. Тем самым, эта разработка ближе к процессу проектирования каких-либо технических устройств, но никак не к их массовому производству. Этот творческий характер разработки ПС сохраняется до самого её 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 приёмов, методов и средств.

- Продукт разработки имеет и другую специфическую особенность: ПС при своём использовании (эксплуатации) не расходуется и не расходует используемых ресурсов.

Эти особенности превращают разработку программных средств в уникальный вид человеческой деятельности. Первая особенность означает, что разработка ПС является в значительной степени формализацией описаний требуемых процессов обработки данных, при этом сохраняется опасность, что полученное формальное описание будет недостаточно точно отражать неформальное описание требуемых процессов обработки данных. Вторая особенность позволяет заметить сходство разработки ПС с проектированием технического устройства, но это сходство продолжается до самого конца разработки ПС. Другими словами, можно сказать, что ПС является своим собственным проектом, а процесс его производства является вырожденным. В связи с этим весьма осторожно следует относиться к так называемому индустриальному подходу к разработке программных средств (точно так же, как бы мы относились к индустриальному подходу к проектированию). Третья особенность означает, что статическая форма представления программного продукта слишком мало говорит о его приемлемости для пользователя, ценности и надёжности. Всё это может быть выявлено только в результате его применения на компьютере. Четвёртая особенность отражает уникальные свойства информации: ПС как информационный объект после каждого его применения сохраняется в неизменном состоянии (не уменьшается, не «устает», не «стареет»). Его надёжность не связана со временем или интенсивностью его применения. Каждое его применение связано с работой компьютера, на котором ПС для выполнения своих функций использует память, каналы ввода и вывода и другие ресурсы компьютера. После применения этого ПС все эти ресурсы сохраняются в компьютере и могут использоваться при применении другого ПС.

3.2. Жизненный цикл программного средства

Под *жизненным циклом* ПС (*software life cycle*) понимают весь период его разработки и эксплуатации (использования), начиная от момента возникновения замысла ПС и кончая прекращением всех видов его использования [22, 24, 25, 44]. Жизненный цикл охватывает довольно сложный процесс создания и использования ПС. Этот процесс может

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

В настоящее время можно выделить пять основных подходов к организации процесса создания и использования ПС [65].

- *Водопадный подход.* При таком подходе разработка состоит из цепочки этапов. На каждом этапе создаются документы, используемые на последующем этапе. В исходном документе фиксируются требования к ПС. В конце этой цепочки создаются программы, включаемые в ПС.
- *Исследовательское программирование.* При этом подходе предполагается быстрая (насколько это возможно) реализация рабочих версий программ ПС, выполняющих лишь в первом приближении требуемые функции. После экспериментального применения реализованных программ производится их модификация с целью сделать их более полезными для пользователей. Этот процесс повторяется до тех пор, пока ПС не будет достаточно приемлемо для пользователей. Такой подход применялся на ранних этапах развития программирования, когда технологии программирования не придавали большого значения (использовалась интуитивная технология). В настоящее время этот подход применяется для разработки таких ПС, для которых пользователи не могут точно сформулировать требования (например, для разработки систем искусственного интеллекта).
- *Прототипирование (разработка прототипа).* Этот подход моделирует начальную фазу исследовательского программирования (вплоть до создания рабочих версий программ, предназначенных для проведения экспериментов) с целью установить требования к ПС. В дальнейшем должна последовать разработка ПС по установленным требованиям в рамках какого-либо другого подхода (например, водопадного).
- *Формальные преобразования.* Этот подход включает разработку формальных спецификаций ПС и превращение их в программы путем корректных преобразований. На этом подходе базируется компьютерная технология (CASE-технология) разработки ПС.
- *Сборочное программирование.* При этом подходе предполагается, что ПС конструируется, главным образом, из программных компонент, которые уже существуют. Должно быть некоторое хранилище (библиотека) таких компонент, каждая из которых может многократно использоваться в разных ПС. Такие компоненты называются

повторно используемыми (*reusable*). Процесс разработки ПС при данном подходе состоит скорее из сборки программ из компонент, чем из их программирования [32].

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

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

Стадия *разработки ПС* (*software development*) состоит из этапов его внешнего описания, конструирования ПС, кодирования (программирование в узком смысле) ПС и аттестации ПС. Всем этим этапам сопутствуют процессы документирования и управления ПС (*software management*). Этапы конструирования и кодирования часто перекрываются, иногда довольно сильно. Это означает, что кодирование некоторых частей программного средства может быть начато до завершения этапа конструирования.

Этап *внешнего описания* ПС включает процессы, приводящие к созданию документа, который мы будем называть *внешним описанием ПС* (*software requirements document*). Этот документ является описанием поведения ПС с точки зрения внешнего по отношению к нему наблюдателя с фиксацией требований относительно его качества. Внешнее описание ПС начинается с анализа и определения требований к ПС со сто-

роны пользователей (заказчика), а также включает процессы спецификации этих требований.

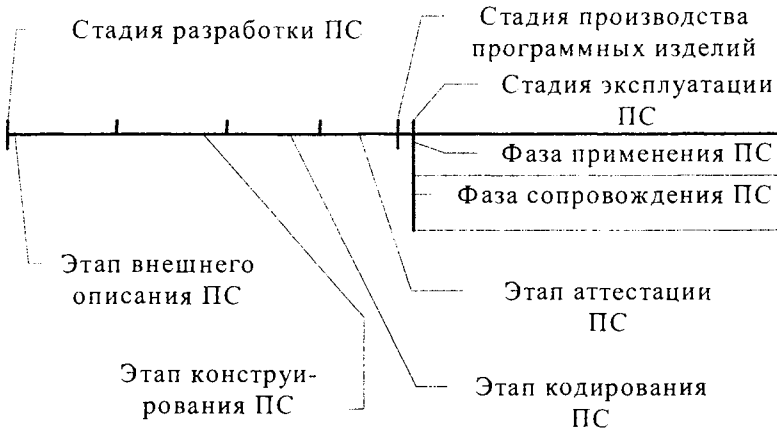


Рис. 3.1. Структура жизненного цикла ПС

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

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

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

Программное изделие (ПИ) – экземпляр или копия разработанного ПС. *Изготовление* ПИ – это процесс генерации и/или воспроизведения (снятия копии) программ и программных документов ПС с целью их поставки пользователю для применения по назначению. *Производство* ПИ – это совокупность работ по обеспечению изготовления требуемого количества ПИ в установленные сроки [22]. Стадия производства ПИ в жизненном цикле ПС является, по существу, вырожденной (не сущест-

венной), так как представляет собой рутинную работу, которая может быть выполнена автоматически и без ошибок. Этим она принципиально отличается от стадии производства различной техники, поэтому в литературе эту стадию, как правило, не включают в жизненный цикл ПС.

Стадия *эксплуатации* ПС охватывает процессы хранения, внедрения и сопровождения ПС, а также транспортировки и применения ПИ по своему назначению. Она состоит из двух параллельно проходящих фаз: фазы применения ПС и фазы сопровождения ПС [44, 65].

Применение ПС (*software operation*) – это использование ПС для решения практических задач на компьютере путём выполнения его программ.

Сопровождение ПС (*software maintenance*) – это процесс сбора информации о качестве ПС в эксплуатации, устранения обнаруженных в нём ошибок, его доработки и модификации, а также извещения пользователей о внесённых в него изменениях [22, 44, 65].

3.3. Понятие качества программного средства

Каждое ПС должно выполнять определённые функции, т. е. делать то, что задумано. Хорошее ПС должно обладать еще целым рядом свойств, позволяющим успешно его использовать в течение длительного периода, т. е. обладать определённым качеством. Качество (quality) ПС – это совокупность его черт и характеристик, которые влияют на его способность удовлетворять заданным потребностям пользователей [59]. Из этого определения не следует, что разные ПС должны обладать одной и той же совокупностью таких свойств с их наилучшими показателями, поскольку улучшение одного из таких свойств ПС часто может быть достигнуто лишь ценой изменения стоимости, сроков завершения разработки и ухудшения других свойств этого ПС. Качество ПС является удовлетворительным, когда оно обладает указанными свойствами в такой степени, в какой гарантируется успешное его использование.

Совокупность свойств, 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 фиксированы критерии отбора требуемых свойств ПС. В настоящее время критериями качества ПС (criteria of software quality) принято считать [7, 30, 49, 59, 63]:

- лёгкость применения,
- надёжность,
- функциональность,
- эффективность,
- сопровождаемость,
- мобильность.

Функциональность (*functionality*) *ПС* – это способность ПС выполнять набор функций, удовлетворяющих заданным или подразумеваемым потребностям пользователей. Этот набор определяется во внешнем описании ПС.

Надёжность ПС подробно обсуждалась в первой главе.

Лёгкость применения (*easy of use*) *ПС* – это характеристики ПС, позволяющие минимизировать усилия пользователя по подготовке исходных данных, применению ПС и оценке полученных результатов, а также вызывающие положительные эмоции определённого или подразумеваемого пользователя.

Эффективность (*efficiency*) *ПС* – это отношение уровня услуг, предоставляемых ПС пользователю при заданных условиях, к объёму используемых ресурсов.

Сопровождаемость (*maintainability*) *ПС* – 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 (*portability*) *ПС* – это способность ПС быть перенесённым из одной среды применения в другую, в частности, с одного компьютера на другой.

Функциональность и надёжность являются обязательными критериями качества ПС, причем 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. Обеспечение этих критериев будет обсуждаться в подходящих частях книги.

3.4. Обеспечение надёжности – основной мотив разработки программных средств

Рассмотрим теперь общие принципы обеспечения надёжности ПС, что является основным мотивом разработки ПС, задающим специфиче-

скую окраску всем технологическим процессам разработки ПС. В технике известны четыре подхода обеспечению надёжности [35]:

- 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) или хотя бы свести их количество к минимуму. Проведенное рассмотрение природы ошибок при разработке ПС позволяет для достижения этой цели сконцентрировать внимание на следующих вопросах:

- упрощение создаваемой ПС,
- обеспечение точности перевода,
- преодоление барьера между пользователем и разработчиком в понимании информации, которой они обмениваются,
- обеспечение контроля принимаемых решений.

Этот подход связан с организацией процессов разработки ПС, т. е. с технологией программирования. И хотя, как мы уже отмечали, гарантировать отсутствие ошибок в ПС невозможно, но в рамках этого подхода можно достигнуть приемлемого уровня надёжности 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 её выполнения. Самоисправление ошибки в программе означает не только обнаружение отказа в процессе её 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 (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, до-

бавление в программу дополнительных фрагментов приводит к ее усложнению (иногда – значительному), что в какой-то мере мешает методам предупреждения ошибок.

3.5. Методы упрощения создаваемой ПС

Мы уже обсуждали в главе 2 сущность вопроса упрощения систем при разработке ПС. Известны два общих метода упрощения систем:

- обеспечения независимости компонент системы,
- использование в системах иерархических структур.

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

3.6. Обеспечение точности перевода

Обеспечение точности перевода направлено на достижение однозначности интерпретации документов различными разработчиками, а также пользователями ПС. Это требует при переводе информации придерживаться определённых правил (определённой дисциплины). Майерс предлагает использовать общую дисциплину решения задач, рассматривая перевод как решение задачи [35]. Лучшим руководством по решению задач он считает книгу Пойа «Как решать задачу» [38]. В соответствии с этим весь процесс перевода можно разбить на следующие этапы:

- обеспечение понимания задачи;
- составление плана (включая цели и методы решения);
- выполнение плана (с проверкой правильности каждого шага);
- анализ полученного решения.

Подробно обсуждать этот вопрос мы здесь не будем.

3.7. Преодоление барьера между пользователем и разработчиком

Как обеспечить, чтобы ПС выполняла то, что пользователю разумно ожидать от нее? Для этого разработчикам необходимо правильно понять, во-первых, чего хочет пользователь, и, во-вторых, знать его уровень подготовки и окружающую его обстановку. Ясное описание соответствующей сферы деятельности пользователя или интересующей его проблемной области во многом облегчает достижение разработчиками этой цели. При разработке ПС следует привлекать пользователя для участия в процессах принятия решений, а также тщательно освоить особенности его работы (лучше всего – побывать в его "шкуре").

3.8. Контроль принимаемых решений

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

Вопросы к главе 3

3.1. Что такое *жизненный цикл ПС*?

3.2. Что такое *внешнее описание ПС*?

3.3. Что такое *сопровождение ПС*?
3.4. Что такое *качество ПС*?
3.5. Что такое *смежный контроль*?

Не переходи мост, пока не дошёл до него.
Народная пословица

Глава 4
ВНЕШНЕЕ ОПИСАНИЕ ПРОГРАММНОГО СРЕДСТВА

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

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

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

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

Внешнее описание ПС играет роль точной постановки задачи, решение которой должно обеспечить разрабатываемое ПС. Более того, оно должно содержать всю информацию, которую необходимо знать пользователю для применения ПС. Оно является исходным документом для трёх 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 счёте, 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 упомянутые параллельных процесса. Это требует принятия особенно серьёзных мер по их предупреждению.

Исходным документом для разработки внешнего описания ПС является *определение требований* к ПС. Через этот документ передается от заказчика (пользователя) к разработчику основная информация относительно требуемого ПС, поэтому формирование этого документа представляет собой довольно длительный и трудный итерационный процесс взаимодействия между заказчиком и разработчиком. 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 определённой формулировки [65], 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 бывает полезным разработка упрощенной версии требуемого ПС, называемую *прототипом* ПС. Анализ "пробного" применения прототипа позволяет выявить действительные потребности пользователей и существенно уточнить требования к ПС.

В определении внешнего описания сразу 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 [35], которые он должен стремиться достигнуть при разработке этого ПС. Эту часть внешнего описания будем называть *спецификацией качества* ПС (ее часто называют *нефункциональной спецификацией* [65], но в этом случае она включает и требования к технологическим процессам). Она, в отличие от функцио-

нальной спецификации, представляется в неформализованном виде и играет роль тех ориентиров, которые в значительной степени определяют выбор подходящих альтернатив при реализации функций ПС, а также определяет стиль всех документов и программ требуемого ПС. Тем самым, спецификация качества играет решающую роль в обеспечении требуемого качества ПС.

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

внешнее описание ПС = определение требований к ПС
+ спецификация качества ПС
+ функциональная спецификация ПС

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

4.2. Определение требований к программному средству

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

Определение требований представляет собой смесь фрагментов на естественном языке, различных таблиц и диаграмм. Такая смесь должна быть понятной пользователю, не ориентирующемуся в специальных программистских понятиях. Обычно в определении требований не содержится формализованных фрагментов, кроме случаев достаточно для этого подготовленных пользователей (например, с математической подготовкой) – 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 [65]. Поэтому важной задачей при создании определения требований является установление контекста использования ПС, включающего связи между этим ПС, аппаратурой и людьми. Лучше всего этот контекст в определении требований представить в графической форме (в виде диаграмм) с добавлением описаний сущностей используемых объектов (блоков ПС, компонент аппаратуры, персонала и т. п.) и характеристики связей между ними.

Известны три способа разработки определения требований к ПС [35]:

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

В *управляемой пользователем* разработке, определения требований к ПС формулируются 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 состоит, в основном, в выяснении того, насколько понятны ему эти требования, и в соответствующей критике рассматриваемого документа. Это может приводить к созданию нескольких редакций этого документа в процессе заключения указанного договора.

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

ПС. Разработанные требования, как правило, утверждаются представителем пользователя.

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

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

4.3. Спецификация качества программного средства

Разработка спецификации качества сводится, по существу, к построению своеобразной модели качества требуемого ПС [22, 35]. В этой модели должен быть перечень всех тех свойств, которые необходимо обеспечить в этом ПС и которые в совокупности образуют приемлемое для пользователя качество ПС. При этом каждое из этих свойств должно быть в достаточной мере конкретизировано с учётом определения требований к ПС, а также с учётом возможности оценки его наличия у разработанного ПС или оценки степени, в которой ПС обладает этим свойством.

Для конкретизации качества ПС по каждому из критериев используется стандартизованный набор достаточно простых свойств ПС, разработанных ISO [7, 22, 59, 63], 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: завершённость.

Надёжность: завершённость, точность, автономность, устойчивость, защищённость.

Лёгкость применения: П-документированность, информативность (применительно к документации по применению), коммуникабельность, устойчивость, защищённость.

Эффективность: временна́я эффективность, эффективность по ресурсам (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 модифици-

руемость. *Изучаемость* – это характеристики ПС, которые позволяют минимизировать усилия по изучению и пониманию программ и документации ПС. *Модифицируемость* – это характеристики ПС, которые позволяют автоматически настраивать на условия применения ПС или упрощают внесение в него вручную необходимых изменений и доработок.

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

Модифицируемость: расширяемость, лёгкость изменения, структурированность, модульность.

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

Ниже определяются используемые примитивы качества ПС [22, 59, 63]. Здесь приводится достаточно большой список, состоящий из 19 примитивов качества, которые следует рассматривать как независимые (самостоятельные) понятия. Некоторые их комбинации образуют более крупные понятия – критерии и подкритерии качества. Такие комбинации можно рассматривать как определённые системы. Но ни одна из этих комбинаций не содержит более шести примитивов (см. выше), так что правило, сформулированное в п. 2.1, здесь не нарушается.

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

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

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

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

Защищённость (*defensiveness*) *ПС* – свойство, характеризующее способность ПС противостоять преднамеренным или нечаянным деструктивным (разрушающим) действиям пользователя.

П-документированность (*u. documentation*) *ПС* – свойство, характеризующее наличие, полноту, понятность, доступность и наглядность

учебной, инструктивной и справочной документации, необходимой для применения ПС.

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

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

Временна́я эффективность (*time efficiency*) *ПС* – мера, характеризующая способность ПС выполнять возложенные на него функции в течение определённого отрезка времени.

Эффективность по ресурсам (*resource efficiency*) *ПС* – мера, характеризующая способность ПС выполнять возложенные на него функции при определённых ограничениях на используемые ресурсы (память).

Эффективность по устройствам (*device efficiency*) *ПС* – мера, характеризующая экономичность использования устройств компьютера для решения поставленной задачи.

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

Понятность (*understandability*) *ПС* – 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 ISO [59], как согласованность, самодокументированность, четкость и, собственно, понятность (текстов программ).

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

Удобочитаемость (*readability*) *ПС* – свойство, характеризующее лёгкость восприятия текста программ ПС (отступы, фрагментация, форматированность).

Расширяемость (*augmentability*) *ПС* – свойство, характеризующее способность ПС к наращиванию объёма памяти для хранения данных или расширению функциональных возможностей отдельных компонент.

Лёгкость изменения (*modifiability*) *ПС* – мера, характеризующая ПС с точки зрения простоты внесения необходимых изменений и доработок на всех этапах и стадиях жизненного цикла ПС.

Модульность (*modularity*) *ПС* – 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 ПС от устройств (*device independence*) – свойство, характеризующее способность ПС работать на разнообразном аппаратном обеспечении (различных типах, марках, моделях компьютеров).

4.4. Функциональная спецификация программного средства

С учётом назначения функциональной спецификации ПС и тяжёлых 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, так что этим вопросам будет посвящена отдельная глава.

Функциональная спецификация состоит из трёх частей:

- описания внешней информационной среды, к которой должны применяться программы разрабатываемой ПС;
- определение функций ПС, определённых 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 её объектами.

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

4.5. Методы контроля внешнего описания программного средства

Разработка внешнего описания обязательно должна завершаться проведением тщательного и разнообразного контроля его правильности. Цель этого процесса состоит в поиске как можно большего числа ошибок, сделанных на этом этапе. Учитывая, что результатом этого

этапа будет, как правило, еще неформализованный текст, 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 проверки 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 его изучение и проверка разработчиками документации по применению и разработчиками комплекта тестов.

Пользовательский контроль внешнего описания – это 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 представляет собой своеобразный динамический контроль внешнего описания, точнее говоря, функциональной спецификации ПС. Для этого необходимо подготовить исходные данные (тесты) и на основании функциональной спецификации осуществить имитацию поведения (работы) разрабатываемого ПС. При этом такую имитацию осуществляет специально назначенный разработчик, выполняющий, по существу, роль будущих программ ПС. Разновидностью такого контроля является имитация за терминалом. В этом случае данные вводятся в компьютер человеком, играющим роль пользователя, и они передаются с помощью несложной программы на другой терминал,

за которым сидит разработчик, играющий роль программ ПС. Полученные результаты передаются через компьютер на первый терминал.

Вопросы к главе 4

4.1. Что такое *определение требований к ПС*?
4.2. Что такое *спецификации качества ПС*?
4.3. Что такое *устойчивость* (*robustness*) *ПС*?
4.4. Что такое *защищённость* (*defensiveness*) *ПС*?
4.5. Что такое *коммуникабельность* (*communicativeness*) *ПС*?
4.6. Что такое *функциональная спецификация ПС*?
4.7. Что такое *ручная имитация внешнего описания ПС*?

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

Л. Витгенштейн

Глава 5
МЕТОДЫ СПЕЦИФИКАЦИИ СЕМАНТИКИ ФУНКЦИЙ

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

5.1. Основные подходы к спецификации семантики функций

Для описания (*спецификации*) семантики функций используются следующие подходы: табличный, алгебраический и логический [2, с. 30–73], а также графический [65].

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

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

$$\begin{aligned} &L1=R1,\\ &\dots \\ &Ln=Rn. \end{aligned} \qquad (5.1)$$

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 осуществляться 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, у которых аргументами могут быть значения различных типов, а результатами являются логические значения (ИСТИНА и ЛОЖЬ). В этом случае набор функций может определяться с помощью системы предикатов. Заметим, что систему равенств алгебраического подхода можно задать с помощью следующей системы предикатов:

PABHO(L1, R1),

. . . (5.2)

PABHO(Ln, Rn),

где предикат PABHO истинен, если равны значения первого и второго его аргументов. Это говорит о том, что логический подход располагает не меньшими возможностями для определения функций, однако он требует от разработчиков ПС умения пользоваться методами математической логики, что, к сожалению, не для всех разработчиков оказывается приемлемым. Более подробно этот подход 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. Здесь 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 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.

5.2. Метод таблиц решений

Метод таблиц решений базируется на использовании таблиц следующего вида (Рис. 5.1).

Верхняя часть этой таблицы определяет различные ситуации, в которых требуется выполнять некоторые действия (операции). Каждая строка этой части задает ряд значений некоторой переменной или некоторого условия. 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 звёздочкой *), отличных от первых, определяется одна и та же, (m+1)-ая, ситуация. Впрочем, в некоторых таблицах решений этот столбец может отсутствовать.

Переменные условия	Ситуации (комбинации значений)				
x1	a[1,1]	a[1,2]	...	a[1,m]	*
x2	a[2,1]	a[2,2]	...	a[2,m]	*
...			...		
xn	a[n,1]	a[n,2]	...	a[n,m]	*
s1	u[1,1]	u[1,2]	...	u[1,m]	u[1,m+1]
s2	u[2,1]	u[2,2]	...	u[2,m]	u[2,m+1]
...			...		
sk	u[k,1]	u[k,2]		u[k,m]	u[k,m+1]
Действия	Комбинации выполняемых действий				

Рис. 5.1. Общая схема таблиц решений

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 u[i, j], i=1, ... m+1, j=1, ... k) указывается, следует ли выполнять (при u[i, j]= '+') это действие в данной ситуации или не следует (при u[i, 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 предопределён, например, в порядке следования строк в нижней части этой таблицы.

Условия !	Ситуации							
Состояние светофора	Кр	Кр	Кр	Жел	Жел	Зел	Зел	Зел
T=Ткр	Нет	Нет	Да	*	*	*	*	*
T=Тжел	*	*	*	Нет	Да	*	*	*
T> Тзел	*	*	*	*	*	Нет	Да	Да
Появление привилегированной машины	Нет	Да	*	*	*	*	Нет	Да
Включить красный	–	–	–	–	–	–	+	–
Включить желтый	–	+	+	–	–	–	–	–
Включить зеленый	–	–	–	–	+	–	–	–
T=0	–	+	+	–	+	–	+	–
T:=T+1	+	–	–	+	–	+	–	+
Освобождение пешеходной дорожки	–	–	–	+	–	–	–	–
Пропуск пешеходов	+	+	+	–	–	–	–	–
Пропуск машин	–	–	–	–	–	+	+	+
Действия	Комбинации выполняемых действий							

Рис. 5.2. Таблица решений "Светофор у пешеходной дорожки"

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 (Ткр – для красного цвета, Тжёл – для жёлтого, Тзел – для зелёного). У светофора имеется счётчик таких единиц. При переключении светофора в счётчике устанавливается 0. Работа светофора

усложняется необходимостью пропускать привилегированные машины (при их появлении на светофор поступает специальный сигнал) с минимальной задержкой, но при обеспечении безопасности пешеходов. Приведённая на рис. 5.2 таблица решений описывает работу конкретного светофора, управляющего заданным порядком движения в каждую единицу времени. Звёздочка (*) в этой таблице означает произвольное значение соответствующего условия.

5.3. Операционная семантика

В операционной семантике алгебраического подхода к описанию семантики функций рассматривается следующий частный случай системы равенств (5.1):

$$
\begin{aligned}
&f1(x1, x2, \dots, xk) = E1,\\
&\dots\dots\dots\dots\\
&fn(x1, x2, \dots, xk) = En,
\end{aligned}
\qquad (5.3)
$$

где в левых частях равенств явно указаны определяемые функции, каждая из которых зависит (для простоты) от одних и тех же параметров

$$x1, x2, \dots, xk,$$

а правые части этих равенств представляют собой выражения, содержащие, вообще говоря, вхождения этих функций, т. е. определяемые функции могут быть *рекурсивными*. Поэтому определяемая функция может иметь дополнительные вхождения в левые части этой системы равенств с постоянными значениями некоторых параметров. Таким образом, каждый параметр xj вхождения определяемой функции fi в левую часть равенств (5.3) будем понимать либо как переменную yj, либо как константу cij для $i=1, \dots, n$, $j=1, \dots, k$, причём совокупность переменных

$$y1, y2, \dots, yk$$

представляет входные данные, для которых требуется вычислять определяемые функции.

Операционная семантика интерпретирует эти равенства как систему подстановок. Под подстановкой

$$E \Big|_{T}^{S}$$

выражения (терма) T в выражение E вместо символа s будем понимать *переписывание* выражения E с заменой каждого вхождения в него символа s на выражение T. Каждое равенство

$$fi(x1, x2, \dots, xk) = Ei,\ i=1, \dots, n,$$

задает в параметрической форме множество правил подстановок вида

$$fi(T1, T2, \dots, Tk) \to Ei \left| \begin{matrix} x1, x2, \dots, xk \\ \\ T1, T2, \dots, Tk \end{matrix} \right., \quad i=1, \dots, n,$$

где T1, T2, ... , Tk – конкретные аргументы (значения или определяющие их выражения) данной функции. Каждое такое правило допускает замену в каком-либо выражении вхождения его левой части на её правую часть.

Интерпретация системы равенств (5.3) 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, предопределённых операций и с *переписыванием* получаемых в результате этого равенств. В результате этого будет сформирована исходная *преобразуемая* система равенств.

На каждом следующем шаге по *текущей преобразуемой* системе равенств производится формирование *новой преобразуемой* системы равенств (которая становится *текущей*) путем *переписывания* этих равенств со следующими преобразованиями.

(1) Если правая часть очередного равенства является каким-либо значением, то это равенство не изменяется (это значение и является значением функции, указанной в левой части этого равенства).

(2) В противном случае правая часть является выражением, содержащим вхождения каких-либо определяемых функций с теми или иными наборами аргументов. В этом случае для каждого вхождения функции в эту правую часть (с конкретным набором аргументов) просматриваются левые части преобразуемых равенств и выполняются следующие действия.

(2.1) Если для этого вхождения в *текущей преобразуемой* системе равенств находится совпадающая с ним левая часть некоторого равенства, то проверяется правая часть этого равенства

(2.1.1) и в случае, если она является уже вычисленным значением, производится подстановка этого значения вместо указанного вхождения определяемой функции,

(2.1.2) если же эта правая часть не является вычисленным значением, то указанное вхождение *переписывается* в неизменном виде.

(2.2) В том же случае, если для указанного вхождения в *текущей преобразуемой* системе равенств не находится совпадающей с ним левой части никакого равенства, то к новой *преобразуемой* системе равенств дописывается *новое* равенство. Это равенство получается из исходного равенства для определяемой функции

fi(y1, y2, ..., yk), i=1, ... , n,

вхождение которой в данный момент исследуется, путем подстановки аргументов этой функции из исследуемого вхождения вместо параметров y1, y2, ..., yk этой функции (с выполнением предопределённых операций там, где это возможно).

Эти шаги будут осуществляться до тех пор, пока все определяемые функции не будут иметь вычисленные значения.

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

F(0)=1,
F(n)=F(n-1)*n.

Для вычисления значения F(3) осуществляются следующие шаги.

1-й шаг: F(0)=1, F(3)=F(2) *3.
2-й шаг: F(0)=1, F(3)=F(2) *3, F(2)=F(1) *2.
3-й шаг: F(0)=1, F(3)=F(2) *3, F(2)=F(1) *2, F(1)=F(0) *1.
4-й шаг: F(0)=1, F(3)=F(2) *3, F(2)=F(1)*2, F(1)=1.
5-й шаг: F(0)=1, F(3)=F(2) *3, 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-м шаге получено.

5.4. Денотационная семантика

В денотационной семантике алгебраического подхода рассматривается также система равенств вида (5.3), которая интерпретируется как система функциональных уравнений, а определяемые функции являются некоторым решением этой системы. В классической математике изу-

чению функциональных уравнений (в частности, интегральных уравнений) уделяется большое внимание и оно связано с построением достаточно глубокого математического аппарата. Применительно к программированию этими вопросами серьёзно занимался Д. Скотт [40].

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

$$\begin{aligned} &X1= phi[1,1] \cup phi[1,2] \cup \ldots \cup phi[1,k1],\\ &X2= phi[2,1] \cup phi[2,2] \cup \ldots \cup phi[2,k2],\\ &\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\\ &Xn= phi[n,1] \cup phi[n,2] \cup \ldots \cup phi[n,kn], \end{aligned} \tag{5.4}$$

причём i-е уравнение при ki=0 имеет вид

$$Xi=\varnothing.$$

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 (5.4) языки. Символы X1, X2, ... , Xn, являющиеся метапеременными грамматики, здесь будут рассматриваться как переменные, значениями которых являются языки (множества значений этих метапеременных). Символы phi[i,j], i=1,....n, j=1,....kj, обозначают цепочки в объединённом алфавите терминальных символов и метапеременных:

$$phi[i,j] \in (A \cup \{X1, X2, \ldots, Xn\})^* .$$

Цепочка phi[i,j] рассматривается как некоторое выражение, определяющее значение, являющееся языком. Такое выражение определяется следующим образом. Если значения X1, X2, ... , Xn заданы, то цепочка

$$phi= Z1\ Z2 \ldots Zk\ ,\ Zi \in (A \cup \{X1, X2, \ldots, Xn\}) \text{ для } i=1, \ldots, k$$

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

$$\{p1\ p2 \ldots pk \mid pj \in Zj,\ j=1, \ldots, k\},$$

причём цепочка p1 p2 ... pk представляет собой последовательность записанных друг за другом цепочек p1, p2, ... , pk (результат выполнения операции *конкатенации* цепочек). Таким образом, каждая правая часть уравнений системы (5.4) представляет собой объединение множеств цепочек.

Решением системы (5.4) является набор языков

(L1, L2, ... , Ln),

если все уравнения системы (5.4) превращаются в тождество при X1= L1, X2= L2, ... , Xn= Ln.

Рассмотрим в качестве примера частный случай системы (5.4), состоящий из одного уравнения

$X= a X \cup b X \cup X \cup c$

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

L={ phi c | phi ∈ {a, b}*}.

Система (5.4) может иметь несколько решений. Так в рассмотренном примере помимо L решениями являются также

L1=L ∪ {phi a | phi ∈ {a, b}*}

и

L2=L ∪ {phi b | phi ∈ {a, b}*}.

В соответствии с денотационной семантикой в качестве *определяемого* решения системы (5.4) принимается наименьшее решение. Решение

(L1, L2, ... , Ln)

системы (5.4) называется *наименьшим*, если для любого другого решения

(L1', L2',..., Ln')

выполняются соотношения

$L1 \subseteq L1', L2 \subseteq L2', \ldots , Ln \subseteq Ln'$.

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

В качестве метода решения систем уравнений (5.3) и (5.4) можно использовать метод последовательных приближений. Сущность этого метода для системы (5.4) заключается в следующем. Обозначим правые части уравнений системы (5.4) операторами

Ti(X1, X2, ... , Xn), i=1, … , n.

Тогда система (5.4) примет вид

$$
\begin{aligned}
&X1=T1(X1, X2, \dots, Xn),\\
&X2=T2(X1, X2, \dots, Xn),\\
&\dots\dots\dots\dots\\
&Xn=Tn(X1, X2, \dots, Xn).
\end{aligned}
\qquad (5.5)
$$

В качестве начального приближения решения этой системы принимается набор языков

$$(L1[0], \dots, Ln[0]) = (\varnothing,\varnothing,\dots,\varnothing).$$

Каждое следующее приближение будет определяться по формуле:

$$(L1[i],\dots,Ln[i])= (T1(L1[i-1], \dots, Ln[i-1]),$$

$$\dots\dots\dots\dots$$

$$Tn(L1[i-1], \dots, Ln[i-1])).$$

Так как операции объединения и сцепления множеств являются монотонными функциями относительно отношения порядка $\subseteq$, то этот процесс сходится к решению

$$(L1, \dots, Ln)$$

системы (5.5), т.е.

$$(L1, \dots, Ln)= (T1(L1, \dots, Ln), \dots, Tn(L1, \dots, Ln))$$

и это решение является наименьшим. Это решение называют еще *наименьшей неподвижной точкой* системы операторов T1, T2, ... , Tn.

В рассмотренном примере этот процесс даёт следующую последовательность приближений:

$L[0]= \varnothing$,

$L[1]= \{c\}$, $L[2]= \{c, ac, bc\}$,

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

. . . . . .

Этот процесс сходится к указанному выше наименьшему решению L.

С помощью денотационной семантики можно определять более широкий класс грамматики по сравнению с формой Бэкуса–Наура. Так в форме Бэкуса–Наура не определены правила вида

$$X::= X,$$

тогда как уравнение вида

$$X= X$$

имеет вполне корректную интерпретацию в денотационной семантике.

5.5. Аксиоматическая семантика

В аксиоматической семантике алгебраического подхода система (5.1) интерпретируется как набор аксиом в рамках некоторой формальной логической системы, в которой есть правила вывода и/или интерпретации определяемых объектов.

Для интерпретации системы (5.1) вводится понятие *аксиоматического описания* (S, E) – логически связанной пары понятий: S – сигнатура используемых в системе (5.1) символов функций f1, f2, ... , fm и символов констант (нульместных функциональных символов) c1, c2, ... , cl, а E – набор аксиом, представленный системой (5.1). Предполагается, что каждая переменная xi, i=1, ... , k, и каждая константа ci, i=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=ti', i=1, ... , r, и конкретные значения констант ci=ci', i=1, ... , l. В таком случае говорят, что задана одна конкретная интерпретация **A** символов сигнатуры S, называемая *алгебраической системой*

A=(t1', ... , tr', f1', ... , fm', c1', ... , cl'),

где fi', 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 в тождества равенства системы E после подстановки в них fi', i=1, ... , m, и ci', i=1, ... , l, вместо fi и ci соответственно.

В программировании в качестве алгебраической системы можно рассматривать, например, тип данных, при этом определяемые функции представляют собой операции, применимые к данным этого типа. Так, К. Хоор построил аксиоматическое определение набора типов данных [47], 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 M – некоторые типы данных, такие, что $m \in M$, $d \in D$, ПУСТ $\in$ M, ДНО $\in$ D1, а функциональные символы представляют функции следующих типов:

УДАЛИТЬ: $M \to M$,

ДОБАВИТЬ: $M * D \to M$,

ВЕРХ: $M \to D1$.

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*. Для этого зададим следующую интерпретацию символов её сигнатуры: пусть D – множество значений, которые могут быть элементами магазина, $D1=D \cup \{ДНО\}$, а M – множество состояний магазина, $M=\{d1, d2, \ldots, dn \mid di \in D, i=1, \ldots, n, n\geq 0\}$, ПУСТ={ }, ДНО – особое значение (зависящее от реализации магазина), не принадлежащее D. Тогда указанный набор аксиом определяет 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.

5.6. Языки спецификаций

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 ПС* понимается формальный язык, предназначенный для спецификации функций. В нём 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 нём спецификации (например, с помощью интерпретатора), то такой язык является и языком программирования,

может быть, и не позволяющим создавать эффективные программы. Однако для языка спецификаций важно не эффективность выполнения спецификации на компьютере, а её выразительность. Язык спецификации, не являющийся языком программирования, также может быть полезен в процессе разработки ПС (для автоматизации контроля, тестирования и т. п.).

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

Упражнения к главе 5

5.1. Функции

function F(x, y: integer): integer;
function G(x, y: integer): integer;
function R(x, y: integer): integer;

определены с помощью операционной семантики равенствами:

$R(x, y) = x*(y - 1)$,
$F(x, y) = R(x + 1, y) - R(x, y - 1)$,
$G(x, y) = F(x, R(x, y))$.

Найти значения G(3, 3).

5.2. Функции

function F(n: integer): integer;
function G(n: integer): integer;

определены с помощью операционной семантики равенствами:

$F(0)=1$,
$G(0)=2$,
$F(n)=G(n-1)$,
$G(n)=F(n-1) + G(n-1)$.

Найти значения F(3) и G(3).

5.3. Формальные языки E и T определены над алфавитом

{'a', '*', '&', '<', '>'}

с помощью денотационной семантики равенствами

E= T ⋃ '*' T ⋃ E '&' T,
T= 'a' ⋃ 'a*' ⋃ '<' E '>'.

Какие из следующих строк

'*a&*a*&a*',
'*a&<a&a*>',

'*<*a*&a>&<*a*>*'

принадлежат языку E и какие из них не принадлежат языку E.

5.4. Тип R определён с помощью следующей аксиоматической семантики.

Описания:

```
type R= record P1, P2, P3: CHAR end;
function READ(S: R): CHAR;   {READ: R → CHAR}
function SHIFT(S: R): R;   {SHIFT: R → R}
function ADD(S: R, C: CHAR): R;  {ADD: R * CHAR → R}
function REMOVE(S: R): R;   {REMOVE: R → R}
var X, Y, Z: CHAR;
U: R;
```

Аксиомы:

```
SHIFT(ADD(ADD(ADD(U, X), Y), Z)) =
          ADD(ADD(ADD(U,Y), Z), X);
REMOVE(U) = SHIFT(ADD(U, '#'));
READ(SHIFT(ADD(U, X))) = X;
```

Найти значение:

```
READ(SHIFT(SHIFT(REMOVE(ADD(ADD(U, 'a'), 'b'))))) =
```

Разделяй и властвуй!
Латинское изречение

Глава 6
АРХИТЕКТУРА ПРОГРАММНОГО СРЕДСТВА

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

6.1. Понятие архитектуры программного средства

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

Основные задачи разработки архитектуры ПС:

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

С учетом принимаемых на этом этапе решений производится дальнейшая конкретизация функциональных спецификаций, включая в необходимых случаях описание интерфейса между выделенными подсистемами и пользователем (пользовательского интерфейса).

6.2. Основные классы архитектур программных средств

Различают следующие основные классы архитектур программных средств [35]:

- цельная программа;
- комплекс автономно выполняемых программ;
- слоистая программная система;
- коллектив параллельно действующих 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 ка-

кую-либо ярко выраженную функцию, реализация которой не представляется слишком сложной. Описание такой архитектуры сводится, в основном, к описанию пользовательского интерфейса.

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 применяются к одной и той же информационной среде.

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

Слоистая программная система состоит из 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 определённый интерфейс, ничего не зная о внутреннем строении всех предшествующих слоев;
- каждый слой располагает определёнными ресурсами, которые он либо скрывает от других слоёв, 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 [60]. Эта операционная система состоит из четырёх слоёв (см. рис. 6.1). На нулевом слое производится обработка всех прерываний и выделение централь-

ного процессора программам (процессам) в пакетном режиме. Только этот уровень осведомлён 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 и вывода.

	Прикладные программы
3:	Управление входными и выходными потоками данных
2:	Обеспечение связи с консолью оператора
1:	Управление памятью
0:	Диспетчеризация и синхронизация процессов
	Компьютер

Рис. 6.1. 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 производится их синхронизация. Обычно взаимодействие между такими процессами производится путём передачи друг другу некоторых сообщений.

Простейшей разновидностью такой архитектуры является конвейер. Возможности для организации конвейера имеются, например, в операционной системе UNIX [28]. *Конвейер* представляет собой последова-

тельность программ, в которой стандартный вывод каждой программы, кроме самой последней, связан со стандартным вводом следующей программы этой последовательности (см. рис. 6.2). Конвейер обрабатывает некоторый поток сообщений. Каждое сообщение этого потока поступает на ввод первой программе, которая переработанное сообщение передаёт 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 и приступает к обработке следующего сообщения. 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).

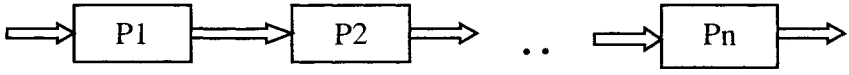


Рис. 6.2. Конвейер параллельно действующих программ

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

Пример программной системы с портами сообщений приведен на рис. 6.3. 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.

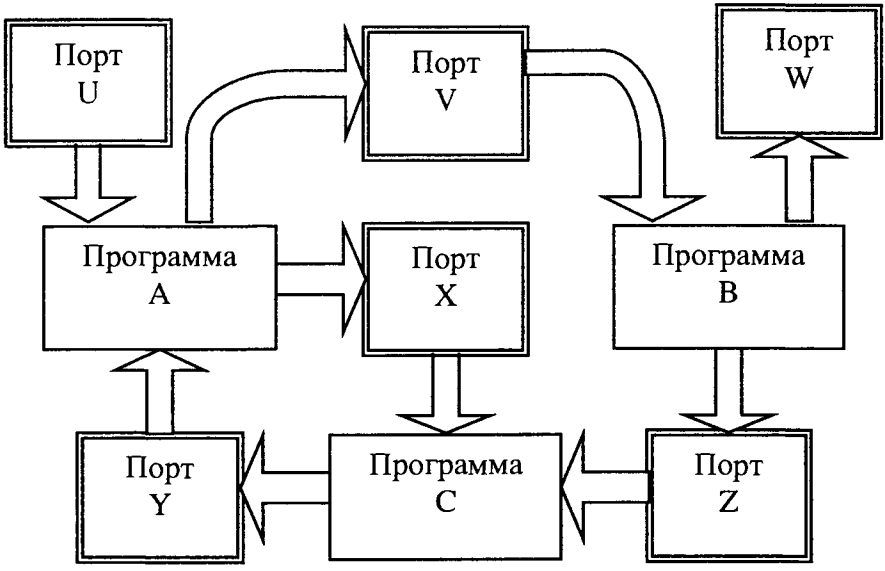


Рис. 6.3. Пример программной системы с портами сообщений

Программные системы с портами сообщений могут быть как жёсткой конфигурации, так и гибкой конфигурации. В системах с портами *жёсткой* конфигурации каждая программа жёстко связывается с одним или с несколькими входными портами. Для передачи сообщения такая программа должна явно указать адрес передачи: имя программы и имя её входного порта. В этом случае при изменении конфигурации системы придётся 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 предварительная настройка программ по информации, задаваемой пользователем. Эта настройка производится с помощью специальной программной

компоненты, осуществляющей совмещение каждого выходного виртуального порта одной программы с каким-либо входным виртуальным портом другой. Тем самым, в этом случае при изменении конфигурации системы не требуется какой-либо ручной корректировки используемых программ. Однако в этом случае требуется иметь специальную программную компоненту, осуществляющую настройку системы.

6.3. Архитектурные функции

Для обеспечения взаимодействия между подсистемами в ряде случаев не требуется создавать какие-либо дополнительные программные компоненты (помимо реализации внешних функций) – для этого может быть достаточно заранее фиксированных соглашений и стандартных возможностей базового программного обеспечения (операционной системы). Так, в комплексе автономно выполняемых программ для обеспечения взаимодействия достаточно описания (спецификации) общей внешней информационной среды и возможностей операционной системы для запуска программ. В слоистой программной системе может оказаться достаточным спецификации выделенных программных слоев и обычного аппарата обращения к процедурам. В программном конвейере взаимодействие между программами также может обеспечивать операционная система (как это делается 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 реализуют не внешние функции ПС, а функции, возникшие в результате разработки архитектуры этого ПС для поддержки взаимодействия между выделенными программными подсистемами. Функцию, поддерживающую взаимодействие между программными подсистемами, выделенными в архи-

тектуре ПС, и выполняемую программной компонентой ПС, видимой из-вне ПС, будем называть *архитектурной функцией.*

6.4. 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 ее контроль потенциальными разработчиками программных подсистем, входящих в состав ПС в соответствии с разработанной архитектурой.

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

Вопросы к главе 6

6.1. Что такое *архитектура ПС*?
6.2. Какие классы архитектур Вы знаете?
6.3. Что такое *архитектурная функция*?

Смотри в корень!
Козьма Прутков

Глава 7
РАЗРАБОТКА СТРУКТУРЫ ПРОГРАММЫ И МОДУЛЬНОЕ ПРОГРАММИРОВАНИЕ

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

7.1. Цель модульного программирования

Приступая к разработке каждой программы ПС, следует иметь в виду, что она, как правило, является большой системой, поэтому мы должны принять меры для ее упрощения. Для этого такую программу разрабатывают по частям, которые называются программными модулями [22, 42, 48]. А сам такой метод разработки программ называют *модульным* программированием [21]. *Программный модуль* – это любой фрагмент описания процесса, оформляемый как самостоятельный программный продукт, пригодный для использования в описаниях разных процессов. Это означает, что каждый программный модуль программируется, компилируется и отлаживается отдельно от других модулей программы, и тем самым, физически разделен 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 упрощения систем (см. гл. 3, п. 3.5): и обеспечение независимости компонент системы, и использование иерархических структур. Для воплощения первого метода формулируются определенные требования, которым должен удовлетворять программный модуль, т. е. выявляются основные характеристики «хорошего» программного модуля. Для воплощения второго метода

используют древовидные модульные структуры программ (включая деревья со сросшимися ветвями).

7.2. Основные характеристики программного модуля

Не всякий программный модуль способствует упрощению программы [43]. Выделить хороший с этой точки зрения модуль является серьёзной творческой задачей. Для оценки приемлемости выделенного модуля используются разные критерии. Так, Хольт [61] предложил следующие два общих таких критерия:

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

Майерс [35] предлагает для оценки приемлемости программного модуля использовать более конструктивные его характеристики:

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

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

Прочность модуля – это мера его внутренних связей. Чем выше прочность модуля, тем больше связей он может спрятать от внешней по отношению к нему части программы и, следовательно, тем больший вклад в упрощение программы он может внести. Для оценки степени прочности модуля Майерс [35] 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. 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 определённую функцию. При реализации этой функции такой модуль может использовать и другие модули. Такой класс программных модулей рекомендуется для использования.

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

В модульных языках программирования как минимум имеются средства для задания функционально прочных модулей (например, модуль типа FUNCTION в языке ФОРТРАН). Средства же для задания информационно прочных модулей в ранних языках программирования отсутствовали. Эти средства появились только в более поздних языках. Так, в языке программирования Ада средством задания информационно прочного модуля является пакет [37].

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 [35] 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 [35]) – это случай, когда данные передаются модулю либо при обращении к нему как значения его параметров, либо как результат его обращения к другому модулю для вычисления некоторой функции. Такой вид сцепления модулей реализуется на языках программирования при использовании обращений к процедурам (функциям).

Рутинность модуля – это его независимость от предыстории обращений к нему. Модуль будем называть *рутинным*, если результат (эффект) обращения к нему зависит только от значений его параметров (и не зависит от предыстории обращений к нему). Модуль будем называть *зависящим от предыстории*, если результат (эффект) обращения к нему зависит от внутреннего состояния этого модуля, изменяемого в результате предыдущих обращений к нему. Майерс [35] не рекомендует использовать зависящие от предыстории (непредсказуемые) модули, так как они провоцируют появление в программах хитрых (неуловимых) ошибок. Однако такая рекомендация является неконструктивной, так как во многих случаях именно зависящий от предыстории модуль является лучшей реализаций информационно прочного модуля. Поэтому более приемлема следующая (более осторожная) рекомендация:

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

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

7.3. Методы разработки структуры программы

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

- синтаксическую спецификацию его входов, позволяющую построить на используемом языке программирования синтаксически правильное обращение к нему (к любому его входу);
- функциональную спецификацию модуля (описание семантики функций, выполняемых этим модулем по каждому из его входов).

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

В процессе разработки программы ее модульная структура может по-разному формироваться и использоваться для определения порядка программирования и отладки модулей, указанных в этой структуре. Поэтому можно говорить о разных методах разработки структуры программы. Обычно в литературе обсуждаются два метода [24, 48]: метод восходящей разработки и метод нисходящей разработки.

Метод *восходящей разработки* заключается в следующем. Сначала строится модульная структура программы в виде дерева. Затем поочерёдно программируются (кодируются) 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 таком же (восходящем) порядке, в каком велось их программирование.

Такой порядок разработки программы, на первый взгляд, кажется вполне естественным: каждый модуль при программировании выражается через уже запрограммированные непосредственно подчиненные модули, а при тестировании использует уже отлаженные модули. Одна-

ко современная технология не рекомендует такой порядок разработки программы. Во-первых, для программирования какого-либо модуля совсем не требуется наличия текстов используемых им модулей – для этого достаточно, чтобы каждый используемый модуль был лишь специфицирован (в объёме, позволяющем построить правильное обращение к нему), а для его тестирования 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 неё, но глобальным для её модулей соображениям (принципам реализации, предположениям, структурам данных и т. п.), что определяет её концептуальную целостность и формируется в процессе её 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 поочерёдно программируются (кодируются) 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 поочерёдное тестирование и отладка в таком же (нисходящем) порядке.

Нисходящее тестирование и отладка означает, что сначала тестируется головной модуль программы, который представляет всю тестируемую программу и поэтому тестируется при «естественном» состоянии информационной среды, при котором начинает выполняться эта программа. При этом те модули, к которым может обращаться головной,

заменяются их имитаторами (так называемыми заглушками [35]). Каждый *имитатор модуля* представляется весьма простым программным фрагментом, который, в основном, сигнализирует о самом факте обращения к имитируемому модулю, производит необходимую для правильной работы программы обработку значений его входных параметров (иногда с их распечаткой) и выдает, если это необходимо, заранее запасённый подходящий результат. После завершения тестирования и отладки головного и любого последующего модуля производится переход к тестированию одного из модулей, которые в данный момент представлены имитаторами, если таковые имеются. Для этого имитатор выбранного для тестирования модуля заменяется самим этим модулем и, кроме того, добавляются имитаторы тех модулей, к которым может обращаться выбранный для тестирования модуль. Каждый такой модуль будет тестироваться при «естественных» состояниях информационной среды, возникающих к моменту обращения к этому модулю при выполнении тестируемой программы. Таким образом, большой объем «отладочного» программирования при восходящем тестировании заменяется программированием достаточно простых имитаторов используемых в программе модулей. Кроме того, имитаторы удобно использовать для того, чтобы подыгрывать процессу подбора тестов путем задания нужных результатов, выдаваемых имитаторами.

При таком порядке разработки программы вся необходимая глобальная информация формируется своевременно, т. е. ликвидируется весьма неприятный источник просчетов при программировании модулей. Некоторым недостатком нисходящей разработки, приводящим к определенным затруднениям при её применении, является необходимость абстрагироваться от базовых возможностей используемого языка программирования, путём выдумывания абстрактных операций, 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.

Особенностью рассмотренных методов восходящей и нисходящей разработок (которые мы будем называть *классическими*) является требование, чтобы модульная структура программы была разработана до начала программирования (кодирования) модулей. Это требование находится в полном соответствии с водопадным подходом к разработке ПС, так как разработка модульной структуры программы и ее кодиро-

вание производятся на разных этапах разработки ПС: первая завершает этап конструирования ПС, а второе – открывает этап кодирования. Однако эти методы вызывают ряд возражений: представляется сомнительным, чтобы до программирования модулей можно было разработать структуру программы достаточно точно и содержательно. На самом деле это делать не обязательно, если несколько модернизировать водопадный подход. Ниже предлагаются конструктивный и архитектурный подходы к разработке программ [21], в которых модульная структура формируется в процессе программирования (кодирования) модулей.

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

Конструктивный подход к разработке программы представляет собой модификацию нисходящей разработки, при которой модульная древовидная структура программы формируется в процессе программирования (кодирования) модулей. Разработка программы при конст-

руктивном подходе начинается с программирования головного модуля, исходя из спецификации программы в целом. При этом спецификация программы принимается в качестве спецификации её головного модуля, который 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) создается спецификация реализующего её фрагмента программы, который в дальнейшем может быть представлен некоторым поддеревом модулей. Важно заметить, что здесь также ответственность за выполнение выделенной функции несет головной (может быть, и единственный) модуль этого поддерева, так что спецификация выделенной функции является одновременно и спецификацией головного модуля этого поддерева. В головном модуле программы для обращения к выделенной функции строится обращение к головному модулю указанного поддерева в соответствии с созданной его спецификацией. Таким образом, на первом шаге разработки программы (при программировании её головного модуля) формируется верхняя начальная часть дерева, например, такая, которая показана на рис. 7.1.

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 очередной шаг построения дерева программы, например, такой, который показан на рис. 7.2.

Архитектурный подход к разработке программы представляет собой модификацию восходящей разработки, при которой модульная структура программы формируется в процессе программирования (кодирования) 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. Эти функции специфицируются, а затем и программируются в виде отдельных программных модулей. Так как процесс выделения таких функций связан с накоплением и обобщением опыта решения задач в заданной предметной области, то обычно сначала выделяются и реали-

зуются 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 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 [35], что опять-таки находится в полном соответствии с водопадным подходом. Однако такой порядок разработки не представляется достаточно обоснованным: тестирование и отладка модулей может привести к изменению спецификации подчиненных модулей и даже к изменению самой модульной структуры программы, так что в этом случае программирование некоторых модулей может оказаться бесполезно проделанной работой. Нам представляется более рациональным другой порядок разработки программы, известный в литературе как метод *нисходящей реализации*, что представляет собой некоторую модификацию водопадного подхода. В этом методе каждый запрограммированный модуль начинают сразу же тестировать до перехода к программированию другого модуля.

Все эти методы имеют еще различные разновидности в зависимости от того, в какой последовательности обходятся узлы (модули) древовидной структуры программы в процессе её разработки [48]. 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.

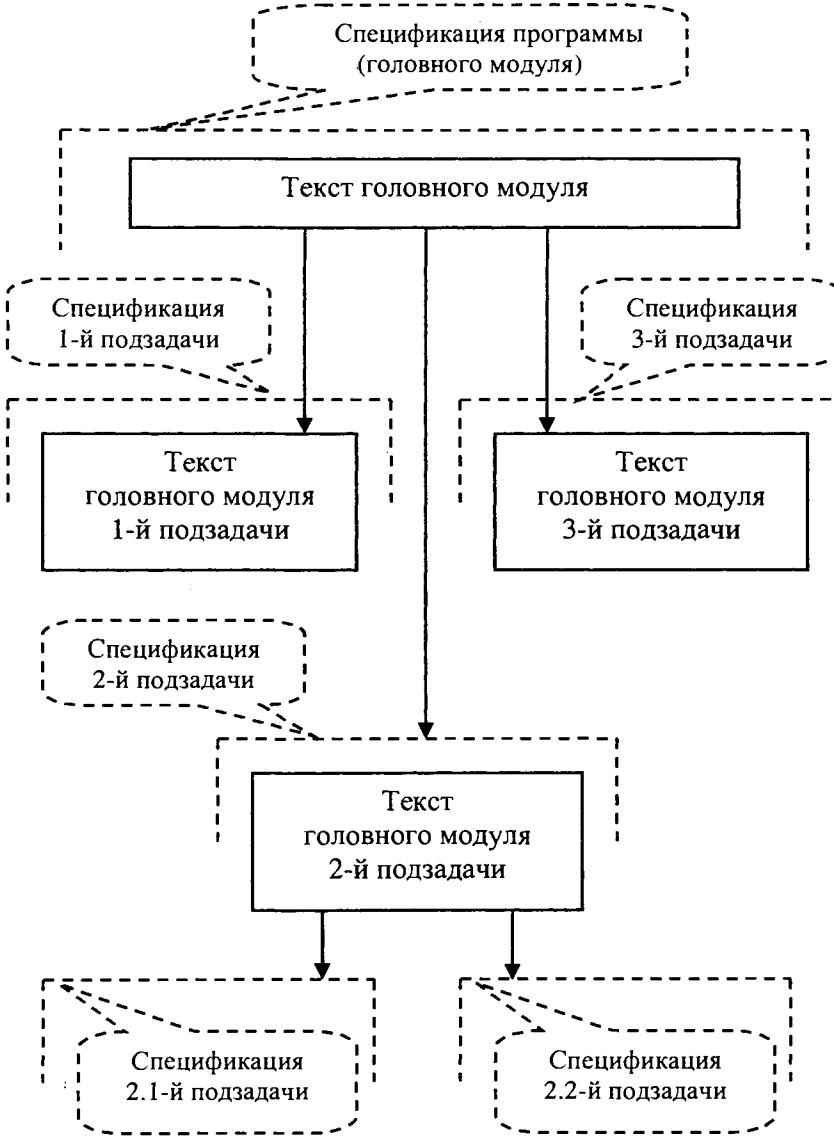


Рис. 7.2. Второй шаг формирования модульной структуры программы при конструктивном подходе

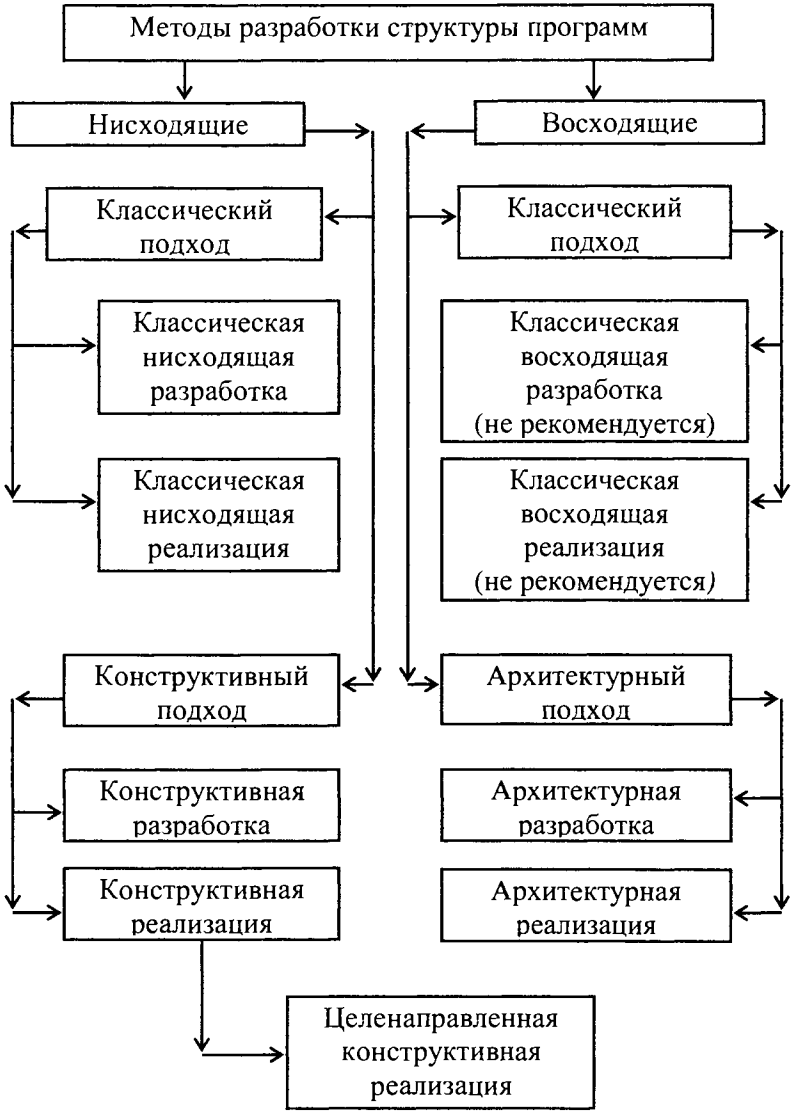


Рис. 7.3. Классификация методов разработки структуры программ

Так, при конструктивной реализации для обхода дерева программы целесообразно следовать идеям Фуксмана, которые он использовал в предложенном им методе вертикального слоения [45]. Сущность такого обхода заключается в следующем. В рамках конструктивного подхода сначала реализуются только те модули, которые необходимы для самого простейшего варианта программы, которая может нормально выполняться только для весьма ограниченного множества наборов входных данных, но для таких данных эта задача будет решаться до конца. Вместо других модулей, на которые в такой программе имеются ссылки, в эту программу вставляются лишь их имитаторы, обеспечивающие, в основном, сигнализацию о выходе за пределы этого частного случая. Затем к этой программе добавляются реализации некоторых других модулей (в частности, вместо некоторых из имеющихся имитаторов), обеспечивающих нормальное выполнение для некоторых других наборов входных данных. И этот процесс продолжается поэтапно до полной реализации требуемой программы. Таким образом, обход дерева программы производится с целью кратчайшим путем реализовать тот или иной вариант (сначала самый простейший) нормально действующей реализации получила название метода *целенаправленной конструктивной реализации*. Достоинством этого метода является то, что уже на достаточно ранней стадии создается работающий вариант разрабатываемой программы. Психологически это играет роль допинга, резко повышающего эффективность труда разработчика. Поэтому этот метод является весьма привлекательным.

Для пояснения сказанного, на рис. 7.3 показана общая классификация рассмотренных методов разработки структуры программы.

7.4. Контроль структуры программы

Для контроля структуры программы можно использовать три метода [35]:

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

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

Смежный контроль сверху – это контроль со стороны разработчиков архитектуры и внешнего описания ПС. Смежный контроль снизу –

это контроль спецификации модулей со стороны разработчиков этих модулей.

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

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

Вопросы к главе 7

7.1. Что такое *программный модуль*?
7.2. Что такое *прочность программного модуля*?
7.3. Что такое *сцепление программного модуля*?
7.4. Что такое *рутинность программного модуля*?
7.5. В чём заключается сущность нисходящей разработки структуры программы?
7.6. Что представляет собой конструктивный подход к разработке структуры программы?
7.7. Что представляет собой метод целенаправленной конструктивной реализации?

Переход от неформального к формальному существенно неформален.
М.Р. Шура-Бура

Глава 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 [35]:

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

Первый шаг разработки программного модуля в значительной степени представляет собой смежный контроль структуры программы снизу: изучая спецификацию модуля, разработчик должен убедиться, что она ему понятна и достаточна для разработки этого модуля. В завершение 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 найдётся 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 для построения текста модуля важно пользоваться технологически обоснованной и практически проверенной дисциплиной программирования. Впервые на это обратил внимание Дейкстра [16], сформулировав и обосновав основные принципы структурного программирования. На этих принципах базируются многие дисциплины программирования, широко применяемые на практике [11, 12, 18, 48]. Наиболее распространенной является дисциплина пошаговой детализации [12], которая подробно обсуждается в п.п. 8.2 и 8.3.

Следующий шаг разработки модуля связан с приведением текста модуля к завершённому 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 [35]. 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); он реализует общий принцип, сформулированный для обсуждаемой технологии программирования, о необходимости контроля принимаемых решений на каждом этапе разработки ПС (см. гл. 3). Методы проверки модуля обсуждаются в п. 8.4.

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

8.2. Структурное программирование

При программировании модуля следует иметь в виду, что программа должна быть понятной не только компьютеру, но и человеку: и

разработчик модуля, и лица, проверяющие модуль, и лица, 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 логику работы модуля сколь угодно сильно, тем самым, сделать модуль трудно понимаемым для человека и, как следствие этого, сделать его ненадёжным или трудно сопровождаемым. Поэтому необходимо принимать меры для выбора подходящих языковых средств и следовать определенной дисциплине программирования. В связи с этим Дейкстра [16] и предложил строить программу как композицию из нескольких типов управляющих конструкций (структур), которые позволяют сильно повысить *понятность* логики работы программы. Программирование с использованием только таких конструкций назвали *структурным*.

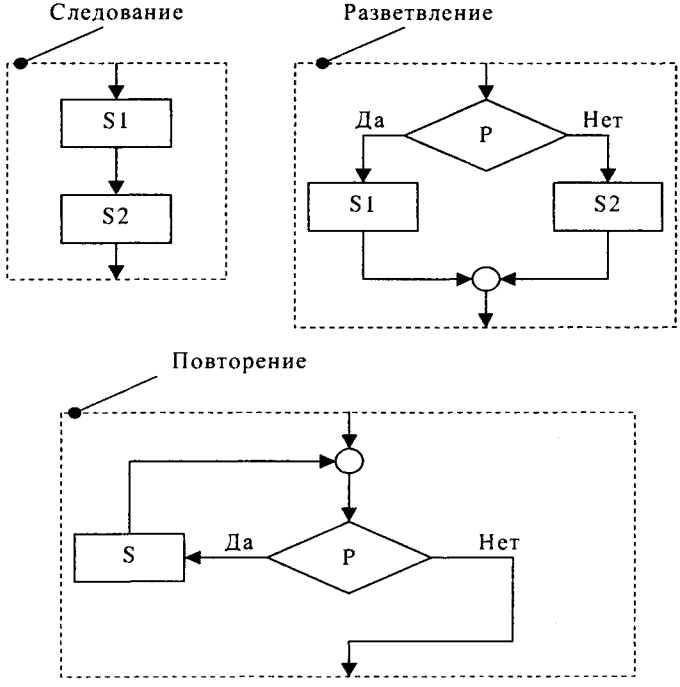


Рис. 8.1. Основные конструкции структурного программирования

Основными конструкциями структурного программирования являются: *следование*, *разветвление* и *повторение* (см. рис. 8.1). Компонентами этих конструкций являются обобщённые операторы (узлы обработки [48]) S, S1, S2 и условие (предикат) P. В качестве обобщённого 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 обобщённый оператор имеет только один вход и один выход.

Весьма важно также, что эти конструкции являются уже математическими объектами (что, по существу, и объясняет причину успеха структурного программирования). Доказано, что для каждой неструктурированной программы можно построить функционально эквивалентную (т. е. решающую ту же задачу) структурированную программу. Для структурированных программ можно математически доказывать определенные свойства, что позволяет обнаруживать в программе некоторые ошибки. Этому вопросу будет посвящена отдельная глава.

Структурное программирование иногда называют еще "программированием без GO TO". Однако дело здесь не в операторе GO TO, а в его беспорядочном использовании. Очень часто при воплощении структурного программирования на некоторых языках программирования (например, на ФОРТРАНе) оператор перехода (GO TO) используется для реализации структурных конструкций, что не нарушает принципов структурного программирования. Запутывают программу как раз "неструктурные" операторы перехода, особенно переход к оператору, расположенному в тексте модуля выше (раньше) выполняемого оператора перехода. Тем не менее, попытка избежать оператора перехода в некоторых простых случаях может привести к слишком громоздким структурированным программам, что не улучшает их ясность и содержит опасность появления в тексте модуля дополнительных ошибок. Поэтому можно рекомендовать избегать употребления оператора перехода всюду, где это возможно, но не ценой ясности программы [35].

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 (обобщённого

оператора) и тем самым лишь локально нарушающего структурированность программы. Большие трудности (и усложнение структуры) вызывает структурная реализация реакции на возникающие исключительные 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 более низких уровней. Вполне приемлемой с технологической точки зрения является следующая «неструктурная» реализация реакции на исключительные ситуации [17]. Обработчики исключительных ситуаций помещаются в конце той или иной структурной единицы. Каждый такой обработчик после окончания своей работы должен производить выход из той структурной единицы, в конце которой он помещен. Обращение к такому обработчику производится оператором перехода из этой структурной единицы (включая любую вложенную в неё структурную единицу).

8.3. Пошаговая детализация и понятие о псевдокоде

Структурное программирование дает рекомендации о том, каким должен быть текст модуля. Возникает вопрос, как должен действовать программист, чтобы построить такой текст. Часто программирование модуля начинают с построения его блок-схемы, описывающей в общих чертах логику его работы. Однако технология программирования не рекомендует этого делать без подходящей компьютерной поддержки. Хотя блок-схемы позволяют весьма наглядно представить логику работы модуля, при их ручном кодировании на языке программирования возникает весьма специфический источник ошибок: отображение существенно двумерных структур, какими являются блок-схемы, на линейный текст, представляющий модуль, содержит опасность искажения логики работы модуля, тем более, что психологически довольно трудно сохранить высокий уровень внимания разработчика при повторном рассмотрении блок-схемы. Исключением может быть случай, когда для построения блок-схем используется графический редактор, а сами блок-схемы формализованы настолько, что по ним автоматически генерируется текст на языке программирования (как, например, это делается в Р-технологии [11]).

В качестве основного метода построения текста модуля технология программирования рекомендует *пошаговую детализацию* [12, 35, 48]. 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*) 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 иметь *уточнения* (т. е. в конечном счёте, будут выражены на базовом языке программирования). Последним шагом является получение текста модуля на базовом языке программирования путем замены всех вхождений уточняемых понятий заданными их описаниями и выражение всех вхождений конструкций структурного программирования средствами этого языка программирования.

Пошаговая детализация связана с использованием частично формализованного языка для представления указанных описаний, который получил название *псевдокода* [10, 48]. Этот язык позволяет использовать все конструкции структурного программирования, которые оформляются формализованно, вместе с неформальными фрагментами на естественном языке для представления обобщённых операторов и условий. В качестве обобщённых 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 [35];
- раздел (совокупность) описаний на базовом языке, причем вместо описаний процедур и функций – только их внешнее оформление;
- неформальное обозначение последовательности операторов тела модуля как одного обобщённого оператора (см. ниже), а также нефор-

мальное обозначение тела каждого описания процедуры или функции как одного обобщённого оператора;

- последнее предложение (конец) модуля на базовом языке [35].

Внешнее оформление описания процедуры или функции представляется аналогично.

Следование:

обобщённый_оператор
обобщённый_оператор

Разветвление:

ЕСЛИ *условие* ТО
обобщённый_оператор
ИНАЧЕ
обобщённый_оператор
ВСЁ ЕСЛИ

Повторение:

ПОКА *условие* ДЕЛАТЬ
обобщённый_оператор
ВСЁ ПОКА

Рис. 8.2. Основные конструкции структурного программирования на псевдокоде

Для каждого неформального обобщённого оператора должно быть создано отдельное описание, выражающее логику его работы (детализирующее его содержание) с помощью композиции основных конструкций структурного программирования и других обобщённых операторов. В качестве заголовка такого описания должно быть неформальное обозначение детализируемого обобщенного оператора. Основные конструкции структурного программирования могут быть представлены так, как это показано на 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).

В качестве обобщённого оператора на псевдокоде можно использовать частные случаи оператора перехода, показанные на рис. 8.3.

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 обобщённого оператора

Последовательность обработчиков исключительных ситуаций (исключений) задается в конце модуля или описания процедуры (функции). Каждый такой обработчик имеет вид:

ИСКЛЮЧЕНИЕ *имя_исключения*
обобщённый_оператор
ВСЁ 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.

```
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.
    ВСЁ ЕСЛИ
  ВСЁ ПОКА
  ЕСЛИ ЗАПИСИ НЕ УДАЛЕНЫ ТО
    НАПЕЧАТАТЬ "ЗАПИСИ НЕ УДАЛЕНЫ".
  ИНАЧЕ
    НАПЕЧАТАТЬ "УДАЛЕНО n ЗАПИСЕЙ".
  ВСЁ ЕСЛИ
```

Рис. 8.4. Пример одного шага детализации на псевдокоде

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.
Подробно пошаговая детализация программ изложена в работе [12].

Идею пошаговой детализации приписывают иногда Дейкстре. Однако Дейкстра предлагал принципиально отличающийся метод построения текста модуля [16], 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 её терминах производить детализацию всех уточняемых понятий. 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 [17].

8.4. Контроль программного модуля

Применяются следующие методы контроля программного модуля:

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

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

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

Доказательству свойств программы (в частности, свойств программного модуля) посвящена следующая глава. Здесь лишь отметим, что этот метод применяется пока очень редко.

Вопросы к главе 8

8.1. Что такое *структурное программирование*?
8.2. Что такое *пошаговая детализация программного модуля*?
8.3. Что такое *псевдокод*?

Встречаются две подруги. Одна говорит другой:

– С моим мужем творится что-то странное. Приходит с работы, наливает полную ванну воды, берет удочку и весь вечер ловит рыбу.

– А почему ты не обратишься к врачу?

– Надо бы. Но так хочется свежей рыбки!

Анекдот

Глава 9
ДОКАЗАТЕЛЬСТВО СВОЙСТВ ПРОГРАММЫ

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

9.1. Обоснования программ. Формализация понятия свойства программы

Для повышения надёжности 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 определённые свойства программы как вручную, так и автоматически.

Одной из используемых в настоящее время концепций формальных обоснований программ является использование так называемых триад Хоора [1, 24]. Пусть S – некоторый обобщённый оператор над информационной средой IS, а 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 – *постусло-*

вием 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:

Пример 9.1. {n=0} n:= n+1{n=1},

Пример 9.2. {n<m} n:= n + k {n<m+k},

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

Пример 9.4. {n>0} p:=1; m:=1;

```
ПОКА m < > n ДЕЛАТЬ
  m:=m+1; p:= p*m
ВСЕ ПОКА
```

{p= n!}.

Для доказательства какого-либо свойства программы S используются свойства простых операторов языка программирования (мы здесь ограничимся пустым оператором и оператором присваивания) и свойствами основных конструкций (композиций), с помощью которых строится программа из простых операторов (мы здесь ограничимся тремя основными конструкциями структурного программирования, см. гл. 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* IS. *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 (X0, RIS0) – некоторое произвольное состояние информационной среды IS, и пусть перед выполнением оператора присваивания предикат Q(F(X0, RIS0), RIS0) является истинным. 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, в этом случае после выполнения этого оператора присваивания имеем

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

В силу произвольности выбора состояния информационной среды теорема доказана.

Примером свойства оператора присваивания может служить пример 9.1.

9.3. Свойства основных конструкций структурного программирования

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

Свойство следования выражает следующая

Теорема 9.3. *Пусть* P, Q и R – *предикаты над информационной средой* IS, *а* S1 *и* S2 – *обобщённые операторы, обладающие соответственно свойствами*

{P}S{Q} *и* {Q}S2{R}.

Тогда для составного оператора

S1; S2

справедливо свойство

{P} S1; S2 {R} .

Доказательство. Пусть для некоторого состояния информационной среды IS 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 справедливы свойства примера 9.2 и примера 9.3, то *справедливо* 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* IS, *а* S1 *и* S2 – *обобщённые операторы, обладающие соответственно свойствами*

{P,Q} S1{R} *и* {¬P,Q} S2 {R}.

Тогда для условного оператора

ЕСЛИ P TO S1ИНАЧЕ S2 ВСЁ ЕСЛИ

справедливо свойство

{Q} ЕСЛИ P TO S1ИНАЧЕ S2 ВСЁ ЕСЛИ {R} .

Доказательство. Пусть для некоторого состояния информационной среды IS 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, что для дальнейшего нам будет полезна

Теорема 9.5. *Пусть* P, Q, P1 *и* Q1 – *предикаты над информационной средой* IS, *для которых справедливы импликации*

P1⇒ P *и* Q⇒ Q1,

и пусть для оператора S *справедливо свойство* {P}S{Q}.*Тогда справедливо свойство* {P1}S{Q1} .

Эту теорему называют еще теоремой *об ослаблении свойств программы*.

Доказательство. Пусть для некоторого состояния информационной среды IS перед выполнением оператора S истинен предикат P1. Тогда будет истинен и предикат P (в силу импликации P1⇒ P). Следова-

тельно, в силу свойства оператора S после его выполнения будет истинен предикат Q, а значит и предикат Q1 (в силу импликации $Q \Rightarrow Q1$). Тем самым теорема доказана.

Свойство повторения выражает следующая

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

$$P \Rightarrow I \text{ и } (I, \neg Q) \Rightarrow R,$$

и пусть S – *обобщённый оператор, обладающий свойством* {I}S{I}. *Тогда для оператора цикла*

ПОКА Q ДЕЛАТЬ S ВСЁ ПОКА

имеет место свойство

{P} ПОКА Q ДЕЛАТЬ S ВСЁ ПОКА {R} .

Предикат I называют *инвариантом* оператора цикла.

Доказательство. Для доказательства этой теоремы достаточно доказать свойство

{I} ПОКА Q ДЕЛАТЬ S ВСЁ ПОКА $\{I, \neg Q\}$

(по теореме 9.5 на основании имеющихся в условиях данной теоремы импликаций). Пусть для некоторого состояния информационной среды IS перед выполнением оператора цикла истинен предикат I. Если при этом предикат Q будет ложен, то оператор цикла будет эквивалентен пустому оператору (в соответствии с его семантикой) и, в силу теоремы 9.1, после выполнения оператора цикла будет справедливо утверждение $(I, \neg Q)$. 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 ВСЁ 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 IS (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 придём к ситуации, когда перед его выполнением будет справедливо утверждение $(I, \neg Q)$. А в этом случае, как было доказано выше, это утверждение будет справедливо и после выполнения оператора цикла. Теорема дока-

зана. Например, для оператора цикла из примера 9.4 *справедливо* свойство

```
{n>0, p=1, m=1}
ПОКА  m <> n  ДЕЛАТЬ
        m:=m+1; p:= p*m
ВСЁ ПОКА
{p= n!}.
```

Это следует из теоремы 9.6, так как инвариантом этого оператора цикла является предикат p= m! и справедливы импликации

$$(n>0,\ p=1,\ m=1) \Rightarrow p= m! \ \text{ и } \ (p= m!,\ m= n) \Rightarrow p= n!$$

9.4. Завершаемость выполнения программы

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

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

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

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

Тогда выполнение оператора цикла

ПОКА Q ДЕЛАТЬ S ВСЁ 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 предикат 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 цикле не может выполняться более k раз. Теорема доказана.

Например, для рассмотренного выше примера оператора цикла условиям 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* (см. п. 9.1) 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 её 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 примера 9.4. Это доказательство будет состоять из следующих шагов.

(Шаг 1). n>0 ⇒ (n>0, p – любое, m – любое).

(Шаг 2). По теореме 9.2 имеем

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

(Шаг 3). По теореме 9.2 имеем

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

(Шаг 4). По теореме 9.3 в силу результатов шагов 2 и 3 имеем

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

Докажем, что предикат p= m! является инвариантом цикла, т.е.

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

(Шаг 5). По теореме 9.2 имеем

{p= m!} m:= m+1 {p= (m–1)!},

если представить предусловие в виде

{p= ((m+1)–1)!}.

(Шаг 6). По теореме 9.2, если представить предусловие в виде

{p*m=m!}, имеем

{p= (m–1)!} p:= p*m {p= m!}.

(Шаг 7). По теореме 9.3 в силу результатов шагов 5 и 6 имеем инвариант цикла

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

(Шаг 8). 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!

имеем

```
{n>0, p=1, m=1}
ПОКА  m < > n  ДЕЛАТЬ
        m:= m+1; p:= p*m
ВСЁ ПОКА
{p= n!}.
```

(Шаг 9). По теореме 9.3 в силу результатов шагов 3 и 8 имеем

```
{n>0, p – любое, m – любое}
p:=1; m:=1;
ПОКА  m < > n  ДЕЛАТЬ
        m:= m+1; p:= p*m
ВСЁ ПОКА
{p= n!}.
```

(Шаг 10). По теореме 9.5 в силу результатов шагов 1 и 9 получаем свойство примера 9.4.

Вопросы и упражнения к главе 9

9.1. Что такое *триада Хоора*?

9.2. Что такое *свойство программы*?

9.3. Пусть заданы описания

```
const n= <конкретное целое значение>;
var k, m: integer;
x: array[1..n] of integer;
```

Доказать свойство программы:

```
{n>0}
m:= x[1]
k:=1;
ПОКА k<n ДЕЛАТЬ
        k:= k+1;
        ЕСЛИ  x[k]<m ТО
                m:= x[k]
        ВСЁ ЕСЛИ
ВСЁ ПОКА;
{n>0 & m<= x[i] для всех i, 1<=i<= n}
```

Лишь та – ошибка, что не исправляется.
Конфуций

Глава 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 трёх 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 под отладкой часто понимают [10, 35, 48] только как процесс поиска и исправления ошибок (без тестирования), факт наличия которых устанавливается при тестировании. Иногда тестирование и отладку считают синонимами [24, 44]. В русскоязычной литературе в понятие отладки обычно включают и тестирование [4, 22, 31], поэтому мы будем следовать сложившейся традиции. Впрочем, совместное рассмотрение в данной главе этих процессов делает указанное разночтение не столь существенным. Следует, однако, отметить, что тестирование используется и как часть процесса аттестации ПС (см. гл. 14).

10.2. Принципы и виды отладки программного средства

Успех отладки ПС в значительной степени предопределяет рациональная организация тестирования. При отладке ПС отыскиваются и устраняются, в основном, те ошибки, наличие которых в ПС устанавливается при тестировании. Как было уже отмечено, тестирование не может доказать правильность ПС [16], в лучшем случае оно может продемонстрировать наличие в нём ошибки. Другими словами, нельзя гарантировать, что тестированием ПС практически выполнимым набором тестов можно установить наличие каждой имеющейся в ПС ошибки. Поэтому возникает две задачи. Первая задача: подготовить такой набор тестов и применить к ним ПС, чтобы обнаружить в нём по возможности бóльшее число ошибок. Однако, чем дольше продолжается процесс тестирования (и отладки в целом), тем бóльшей 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 надёжности 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, отведённом на тестирование) обнаруживать бóльшее число ошибок в ПС, необходимо заранее составлять *схему тестирования* (т. е. составлять некоторую схему тестов). Кроме того, при составлении такой схемы необходимо использовать рациональную стратегию проектирования [35] 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 [35]. Левый крайний подход заключается в том, что тесты проектируются только на основании изучения спецификаций ПС (внешнего описания, описания архитектуры и спецификации модулей). Строение модулей при этом никак не учитывается, т. е. они рассматриваются как черные ящики. Фактически такой подход требует полного перебора всех наборов входных данных, так как в противном случае некоторые участки программ ПС могут не ра-

ботать при пропуске любого теста, а это значит, что содержащиеся в них ошибки не будут проявляться. Однако тестирование ПС полным множеством наборов входных данных практически неосуществимо. Правый крайний подход заключается в том, что тесты проектируются на основании изучения текстов программ с целью протестировать все пути выполнения каждой программ ПС. Если принять во внимание наличие в программах циклов с переменным числом повторений, то различных путей выполнения программ ПС может оказаться также чрезвычайно много, так что их тестирование также будет практически неосуществимо.

Рис. 10.1. Спектр подходов к проектированию тестов

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

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

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

Оптимальную стратегию проектирования тестов можно конкретизировать на основании следующего принципа: для каждого программного документа (включая тексты программ), входящего в состав ПС, должны проектироваться свои тесты с целью выявления в нём ошибок. Во всяком случае, этот принцип необходимо соблюдать в соответствии с определением ПС и содержанием понятия технологии программирования как технологии разработки надёжных ПС (см. гл. 1). В связи с этим Майерс даже определяет разные виды тестирования [35] в зависимости от вида программного документа, на основании которого строятся тесты. В нашей стране различают [22] два основных вида отладки (включая тестирование): автономную и комплексную отладку ПС. *Автономная* отладка ПС означает последовательное раздельное тестирование различных частей программ, входящих в ПС, с поиском и исправлением в них фиксируемых при тестировании ошибок. Она фактически включает отладку каждого программного модуля и отладку сопряжения модулей. *Комплексная* отладка означает тестирование ПС в целом с поиском и исправлением фиксируемых при тестировании ошибок во всех документах (включая тексты программ), относящихся к ПС в целом. К таким документам относятся определение требований к ПС, спецификация качества ПС, функциональная спецификация ПС, описание архитектуры ПС и тексты программ ПС.

10.3. Заповеди отладки программного средства

В этом разделе даются общие рекомендации по организации отладки ПС. Но сначала следует отметить некоторый феномен [35]: *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 предупреждения ошибок на предыдущих этапах разработки.

Ниже приводятся рекомендации по организации отладки в форме заповедей [22, 35].

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

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

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

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

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

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

10.4. Автономная отладка программного средства

При автономной отладке ПС каждый модуль на самом деле тестируется в некотором программном окружении, кроме случая, когда отлаживаемая программа состоит только из одного модуля. Это окружение состоит [22] из других модулей, часть которых является модулями отлаживаемой программы, которые уже отлажены, а часть – модулями, управляющими отладкой (*отладочными* модулями, см. ниже). Таким образом, при автономной отладке тестируется всегда некоторая программа (*тестируемая программа*), построенная специально для тестирования отлаживаемого модуля. Эта программа лишь частично совпадает с отлаживаемой программой, кроме случая, когда отлаживается последний модуль отлаживаемой программы. В процессе автономной отладки ПС производится наращивание тестируемой программы отлаженными модулями: при переходе к отладке следующего модуля в его программное окружение добавляется модуль, отладка которого была перед этим закончена. Такой процесс наращивания программного окружения отлаженными модулями называется *интеграцией* программы [35]. 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. гл. 7) это окружение будет содержать только один отладочный модуль (кроме случая, когда отлаживается последний модуль отлаживаемой программы), который будет головным в тестируемой программе. Такой отладочный модуль называют *ведущим* (или драйвером [35]). Ведущий отладочный модуль подготавливает информационную среду для тестирования отлаживаемого модуля (т.е. формирует её состояние, требуемое для тестирования этого модуля, в частности, путём ввода некоторых тестовых данных), осуществляет обращение к отлаживаемому модулю и после окончания его работы выдает необходимые сообщения. При отладке одного модуля для разных тестов могут составляться разные ведущие отладочные модули.

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

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

К *достоинствам восходящего тестирования* относятся:

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

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

Недостатками восходящего тестирования являются следующие его особенности:

- тестовые данные готовятся, как правило, не в той форме, которая рассчитана на пользователя (кроме случая, когда отлаживается последний, головной, модуль отлаживаемой программ);
- большой объем отладочного программирования (при отладке одного модуля приходится составлять много ведущих отладочных модулей,

формирующих подходящее состояние информационной среды для разных тестов);

- необходимость специального тестирования сопряжения модулей.

К *достоинствам нисходящего тестирования* относятся следующие его особенности:

- большинство тестов готовится в форме, рассчитанной на пользователя;
- во многих случаях относительно небольшой объем отладочного программирования (имитаторы модулей, как правило, весьма просты и каждый пригоден для большого числа, нередко – для всех, тестов);
- отпадает необходимость тестирования сопряжения модулей.

Недостатком нисходящего тестирования является то, что тестовое состояние информационной среды перед обращением к отлаживаемому модулю готовится косвенно – оно является результатом применения уже отлаженных модулей к тестовым данным или данным, выдаваемым имитаторами. Это, во-первых, затрудняет подготовку тестов и требует высокой квалификации *тестовика* (разработчика тестов), а во-вторых, делает затруднительным или даже невозможным реализацию полной схемы 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 гл. 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. Однако представляется более целесообразной следующая модификация нисходящего тестирования: перед интеграцией отлаживаемого модуля, для которого не все тестовые состояния информационной среды удалось реализовать, его дополнительно отдельно тестируют для остальных требуемых состояний информационной среды с использованием ведущего отладочного модуля (как в восходящем тестировании). В этом случае в программном окружении отлаживаемого модуля наряду с ведущим отладочным модулем могут присутствовать и отладочные имитаторы.

Часто применяют также комбинацию восходящего и нисходящего тестирования, которую называют методом сандвича [35]. Сущность этого метода заключается в одновременном осуществлении как восходящего, так и нисходящего тестирования, пока эти два процесса тестирования не встретятся на каком-либо модуле где-то в середине структуры отлаживаемой программы. Этот метод при разумном порядке тестирования позволяет воспользоваться достоинствами как восходящего, так и нисходящего тестирования, а также в значительной степени нейтрализовать их недостатки.

Весьма важным при автономной отладке является тестирование сопряжения модулей. Дело в том, что спецификация каждого модуля программы, кроме головного, используется в этой программы в двух ситуациях: во-первых, при разработке текста (иногда говорят: тела) этого модуля и, во-вторых, при написании обращения к этому модулю в

других модулях программы. И в том, и в другом случае в результате ошибки может быть нарушено требуемое соответствие заданной спецификации модуля. Такие ошибки требуется обнаруживать и устранять. Для этого и предназначено тестирование сопряжения модулей. При нисходящем тестировании это делается попутно каждым пропускаемым тестом, что считают достоинством нисходящего тестирования. При восходящем тестировании обращение к отлаживаемому модулю производится не из модулей отлаживаемой программы, а из ведущего отладочного модуля. В связи с этим существует опасность, что ведущий 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 [35].

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

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

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

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

10.5. Комплексная отладка программного средства

Как уже было сказано выше, при комплексной отладке тестируется ПС в целом, причем тесты готовятся по каждому из документов ПС [22]. Тестирование этих документов производится, как правило, в порядке, обратном их разработке. Исключение составляет лишь тестирование документации по применению, которая разрабатывается по внешнему описанию параллельно с разработкой текстов программ – это тестирование лучше производить после завершения тестирования внешнего описания. Тестирование при комплексной отладке представляет собой применение ПС к конкретным данным, которые в принципе могут возникнуть у пользователя (в частности, все тесты готовятся в форме, рассчитанной на пользователя), но, возможно, в моделируемой (а не в реальной) среде. Например, некоторые недоступные при комплексной отладке устройства ввода и вывода могут быть заменены их программными имитаторами.

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

Тестирование внешних функций. Целью тестирования является поиск расхождений между функциональной спецификацией и совокупностью программ ПС. Несмотря на то, что все эти программы автономно уже отлажены, указанные расхождения могут быть, например, из-за того, что внутренние спецификации программ не соответствуют 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 тести-

рования. Ясно лишь, что далеко не каждый примитив качества ПС может быть испытан тестированием (об оценке качества ПС см. гл. 14). Завершённость 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 надёжности ПС. Однако методика такого тестирования еще требует своей разработки. Могут тестироваться такие примитивы качества, как точность, устойчивость, защищённость, временна́я эффективность, в какой-то мере – эффективность по памяти, эффективность по устройствам, расширяемость и, частично, независимость от устройств. Каждый из этих видов тестирования имеет свою специфику и заслуживает отдельного рассмотрения. Мы здесь ограничимся лишь их перечислением. Лёгкость применения ПС (критерий качества, включающий несколько примитивов качества, см. гл. 4) оценивается при тестировании документации по применению ПС.

Тестирование документации по применению ПС. Целью тестирования является поиск несогласованности между документацией по применению ПС и совокупностью программ ПС, а также выявление неудобств, возникающих при применении ПС. Этот этап непосредственно предшествует подключению пользователя к завершению разработки ПС (тестированию определения требований к ПС и аттестации ПС), поэтому весьма важно разработчикам сначала самим воспользоваться ПС так, как это будет делать пользователь [35]. Все тесты на этом этапе готовятся исключительно на основании только документации по применению ПС. Прежде всего должны тестироваться возможности ПС так же, 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 относительно лёгкости 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 [35] как один из путей преодоления барьера между разработчиком и пользователем (см. гл. 3). Обычно это тестирование производится с помощью контрольных задач – типовых за-

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

Вопросы к главе 10

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

Плохо не клади, вора в грех не вводи.
Народная пословица

Глава 11
ОБЕСПЕЧЕНИЕ ФУНКЦИОНАЛЬНОСТИ И НАДЁЖНОСТИ ПРОГРАММНОГО СРЕДСТВА

Функциональность и надёжность как обязательные критерии качества программного средства. Обеспечение завершённости программного средства. Защитное программирование и обеспечение устойчивости программного модуля. Виды защиты и обеспечение защищённости программного средства.

11.1. Функциональность и надёжность как обязательные критерии качества программного средства

В предыдущих главах 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. гл. 4). Правда, занимаясь реализацией функциональной спецификации ПС, мы тем самым обсудили основные вопросы обеспечения критерия функциональности. Объявив надёжность ПС основным его атрибутом (см. гл. 1), мы выбрали *предупреждение ошибок* в качестве основного подхода для обеспечения надёжности ПС (см. гл. 3) и обсудили его воплощение на разных этапах разработки ПС. Таким образом, проявлялся тезис об обязательности функциональности и надёжности 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 глава. Обеспечение других критериев качества будет обсуждаться в следующей главе.

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

11.2. Обеспечение завершённости программного средства

Завершённость ПС является общим примитивом качества ПС для выражения и функциональности, и надёжности ПС, причем для функциональности она является единственным примитивом (см. гл. 4).

Функциональность ПС определяется его функциональной спецификацией. Завершённость как примитив качества ПС является мерой того, в какой степени эта спецификация реализована в разработанном ПС. Обеспечение этого примитива в полном объёме означает реализацию каждой из функций, определённой в функциональной спецификации, со всеми указанными 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 *упрощённая* (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 упрощённой версии ПС не есть разработка его *прототипа*. Прототип разрабатывается для того, чтобы лучше понять условия применения будущего ПС [65], 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. Упрощённая же версия требуемого ПС должна быть рассчитана на *практически полезное* применение любыми пользователями, для которых предназначено это ПС.

Главный принцип обеспечения функциональности такого ПС заключается в том, чтобы с самого начала разрабатывать ПС таким образом, как будто требуется ПС в полном объёме, до тех пор, пока не придётся непосредственно 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 объёме. Можно отложить лишь реализацию тех программных подсистем (определённых 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 упрощённая реализация некоторых программных модулей, опускающая реализацию некоторых

деталей соответствующих функций. Однако такие модули с технологической точки зрения лучше рассматривать как своеобразные (далеко продвинутые) их имитаторы.

Достигнутый при обеспечении функциональности ПС уровень его завершённости на самом деле может быть не таким, как ожидалось, из-за ошибок, оставшихся в этом ПС. Можно лишь говорить, что требуемая завершённость достигнута с некоторой вероятностью, определяемой объёмом и качеством проведённого тестирования. Для повышения этой вероятности необходимо продолжить тестирование и отладку ПС. Однако оценивание такой вероятности является весьма специфической задачей, которая пока еще ждёт своих теоретических исследований.

11.3. Обеспечение точности программного средства

Обеспечение этого примитива качества связано с действиями над значениями вещественных типов (точнее говоря, со значениями, представляемыми с некоторой погрешностью). Обеспечить требуемую точность при вычислении значения той или иной функции – значит получить это значение с погрешностью, не выходящей за рамки заданных границ. Видами погрешности, методами их оценки и методами достижения требуемой точности (так называемыми *приближёнными вычислениями*) занимается вычислительная математика [3, 6]. 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 так называемой *неустранимой погрешности*),
- от *погрешности округления* (неточности выполнения используемых в методе операций).

Значение погрешности округления зависит от того, как запрограммированы выражения: погрешность округления при вычислении выражений может сильно зависеть от порядка выполнения операций. Так, выражения $(a-b)+(c-d)$ и $(a+c)-(b+d)$ не эквиваленты с точки зрения погрешности округления – ошибка округления при вычислении первого выражения может быть существенно меньше, чем при вычислении второго, если a, b, c и d близкие положительные числа. Для уменьшения

погрешности округления при вычислении полиномов часто используют схему Горнера

$$(\ldots (a*x+b)*x+ \ldots p)*x+r.$$

11.4. Обеспечение автономности программного средства

Вопрос об автономности программного средства решается путем принятия решения о возможности использования в требуемом ПС какого-либо подходящего базового программного обеспечения. Надёжность имеющегося в распоряжении разработчиков базового программного обеспечения для целевого компьютера может не отвечать требованиям к надёжности разрабатываемого ПС. Поэтому от использования такого программного обеспечения приходится отказываться, а его функции в требуемом объёме приходится реализовывать в рамках требуемого ПС. Аналогичное решение приходится принимать при жёстких ограничениях на используемые ресурсы (по критерию эффективности ПС). Такое решение может быть принято уже в процессе разработки спецификации качества ПС, иногда – на этапе конструирования ПС.

11.5. Обеспечение устойчивости программного средства

Этот примитив качества ПС обеспечивается с помощью так называемого *защитного программирования*. Вообще говоря, защитное программирование применяется при программировании модуля для повышения надёжности ПС в более широком смысле. Как утверждает Майерс [35], «защитное программирование основано на важной предпосылке: худшее, что может сделать модуль, – это принять неправильные входные данные и затем вернуть неверный, но правдоподобный результат». Для того чтобы этого избежать, в текст модуля включают *проверки* его входных и выходных данных на их корректность в соответствии со спецификацией этого модуля, в частности, должны быть проверены выполнение ограничений на входные и выходные данные и соотношений между ними, указанные в спецификации модуля. В случае отрицательного результата проверки возбуждается соответствующая исключительная ситуация. Для обработки таких ситуаций в конец этого модуля включаются фрагменты второго рода – *обработчики* соответствующих исключительных ситуаций. Эти обработчики, помимо выдачи необходимой диагностической информации, могут принять меры либо по исключению ошибки в данных (например, потребовать их повторного ввода), либо по ослаблению влияния ошибки (например, во избежание

поломки устройств, управляемых с помощью данного ПС, при аварийном прекращении выполнения программы осуществляют мягкую их остановку).

Применение защитного программирования модулей приводит к снижению эффективности ПС как по времени, так и по памяти. Поэтому необходимо разумно регулировать степень применения защитного программирования в зависимости от требований к надёжности и эффективности ПС, сформулированных 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 надёжности ПС (см. п. 3.4 главы 3). Такой случай защитного программирования применяется крайне редко (только если требования к надёжности ПС чрезвычайно высоки).

11.6. Обеспечение защищённости программного средства

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.

11.6.1. Защита от сбоев аппаратуры. В настоящее время этот вид защиты является не очень злободневной задачей (с учетом уровня достигнутой надёжности компьютеров). Но всё же полезно знать её решение. Это обеспечивается организацией так называемых «двойных или тройных просчётов». Для этого весь процесс обработки данных, определяемый ПС, разбивается по времени на интервалы так называемыми

«опорными точками». Длина этого интервала не должна превосходить половины среднего времени безотказной работы компьютера. В начале каждого такого интервала во вторичную память записывается с некоторой контрольной суммой копия состояния изменяемой в этом процессе памяти («опорная точка»). Обработку данных от этой опорной точки до следующей, называют одним «просчётом». Чтобы убедиться в том, что этот просчёт произведен правильно (без сбоев компьютера), производится второй такой просчёт. Для этого после первого просчёта вычисляется и запоминается указанная контрольная сумма, а затем восстанавливается состояние памяти по опорной точке и делается второй просчёт. После второго просчёта вычисляется снова указанная контрольная сумма, которая затем сравнивается с контрольной суммой первого просчёта. Если эти две контрольные суммы совпадают, то второй просчёт считается правильным, в противном случае контрольная сумма второго просчёта также запоминается и производится третий просчёт (с предварительным восстановлением состояния памяти по опорной точке). Если контрольная сумма третьего просчёта совпадет с контрольной суммой одного из первых двух просчётов, то третий просчёт считается правильным, в противном случае требуется инженерная проверка компьютера.

11.6.2. Защита от влияния «чужой» программы. В мультипрограммном режиме работы компьютера в его памяти может одновременно находиться 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. Здесь под «чужой» программой понимается программа (или какой-либо программный фрагмент), выполняемая параллельно (или квазипараллельно) по отношению к защищаемой программе (или ее фрагменту). Этот вид защиты необходимо обеспечивать, чтобы эффект выполнения защищаемой программы не зависел от того, какие программы выполняются параллельно с ней. Обеспечение такой защиты относится, прежде всего, к функциям операционных систем.

Различают две разновидности этой защиты:

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

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

- защиту памяти,
- два режима функционирования компьютера: привилегированный и рабочий (пользовательский),
- два вида операций: привилегированные и ординарные.
- корректную реализацию прерываний и начального включения компьютера,
- временнóе прерывание.

Защита памяти означает возможность программным путем задавать для каждой программы недоступные для нее участки памяти. В *привилегированном* режиме могут выполняться любые операции (как ординарные, так и привилегированные), а в *рабочем* режиме – только ординарные. Попытка выполнить привилегированную операцию, а также обратиться к защищенной памяти в рабочем режиме вызывает соответствующее прерывание. К *привилегированным* операциям относятся операции изменения защиты памяти и режима функционирования, а также доступа к внешней информационной среде. *Корректная реализация* прерываний и начального включения компьютера означает обязательную установку привилегированного режима и отмену защиты памяти. *Временнóе прерывание* дает возможность управляющей программе регулярно получать управления даже в том случае, когда в параллельно выполняемых программах не возникает других прерываний (например, в результате зацикливания). Без такого прерывания она могла бы просто лишиться возможности управлять.

В этих условиях управляющая программа может полностью защитить себя от влияния отказов других программ. Для этого достаточно, чтобы

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

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

Тогда никто не помешает ей выполнять любые реализованные в ней функции защиты других программ (в том числе и доступа к внешней информационной среде). Для облегчения решения этой задачи часть такой программы помещается в постоянную память.

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

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

11.6.3. Защита от отказов «своей» программы. Обеспечивается надёжностью ПС, на что ориентирована вся технология программирования, обсуждаемая в настоящей книге.

11.6.4. Защита от ошибок пользователя. Здесь идет речь не об ошибочных данных, поступающих от пользователя ПС, – защита от них

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

11.6.5. Защита от несанкционированного доступа. Каждому пользователю ПС предоставляет определённые информационные и процедурные ресурсы (услуги), причем у разных пользователей ПС предоставленные им ресурсы могут отличаться, иногда очень существенно. Этот вид защиты должен обеспечить, чтобы каждый пользователь ПС мог использовать только то, что ему предоставлено (санкционировано). Для этого ПС в своей внешней информационной среде может хранить информацию о своих пользователях и о предоставленных им правах использования ресурсов, а также предоставлять пользователям определенные возможности задания этой информации. Защита от несанкционированного доступа к ресурсам ПС осуществляется с помощью так называемых *паролей* (секретных слов). При этом предполагается, что каждый пользователь знает только свой пароль, зарегистрированный в ПС этим пользователем. Для доступа к выделенным ему ресурсам он должен предъявить ПС свой пароль. Другими словами, пользователь как бы "вешает замок" на предоставленные ему права доступа к ресурсам, "ключ" от которого имеется только у этого пользователя.

Различают две разновидности такой защиты:

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

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

в ПС, которая базируется на операционной системе, не обеспечивающей полную защиту от влияния чужих программ.

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

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

Другая разновидность защиты от взлома защиты связана с защитой сообщений, пересылаемых по компьютерным сетям. Такое сообщение может представлять команду на дистанционный доступ к ценной ин-

формации, и этот доступ отправитель сообщения хочет защитить от возможных искажений. Например, при осуществлении банковских операций с использованием компьютерной сети. Использование компьютерной подписи в такой ситуации недостаточно, так как защищаемое сообщение может быть перехвачено «взломщиком» (например, на «перевалочных» пунктах компьютерной сети) и подменено другим сообщением с сохранением компьютерной подписи (или пароля).

Защиту от такого взлома защиты можно осуществить следующим образом [29]. Наряду с функцией F, определяющей компьютерную подпись владельца пароля X, в ПС определены ещё две функции: Stamp и Notary. При передаче сообщения отправитель, помимо компьютерной подписи Y=F(X), должен вычислить ещё другое число S=Stamp(X,R), где X – пароль, а R – текст передаваемого сообщения. Здесь также предполагается, что функция Stamp хорошо известна всем пользователям ПС и обладает таким свойством, что по S практически невозможно ни восстановить число X, ни подобрать другой текст сообщения R с заданной компьютерной подписью Y. При этом передаваемое сообщение (вместе со своей защитой) должно иметь вид:

(R Y S),

причем Y (компьютерная подпись) позволяет получателю сообщения установить истинность клиента, а S как бы скрепляет защищаемый текст сообщения R с компьютерной подписью Y. В связи с этим число S будем называть *электронной* (*компьютерной*) *печатью*. Функция Notary(R,Y,S) проверяет истинность защищаемого сообщения:

(R,Y,S).

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

11.6.6. Защита от защиты. Защита от несанкционированного доступа может создать нежелательную ситуацию для самого владельца прав доступа к ресурсам ПС – он не сможет воспользоваться этими правами, если забудет свой пароль («ключ»). Для защиты интересов пользователя в таких ситуациях и предназначена защита от защиты. Для обеспечения такой защиты ПС должно иметь привилегированного пользователя – *администратора ПС*. Администратор ПС должен, в частности, отвечать за функционирование защиты ПС: именно он должен формировать контингент пользователей данного экземпляра ПС, предоставляя каждому из этих пользователей определённые права доступа

к ресурсам ПС. В ПС должна быть привилегированная операция (для администратора), позволяющая временно снимать защиту от несанкционированного доступа для пользователя с целью фиксации требуемого пароля («замка»).

Вопросы к главе 11

11.1. Что такое *защитное программирование*?

11.2. Какие виды защиты ПС от искажения информации Вы знаете?

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

11.4. Что такое *компьютерная подпись*?

11.5. Что такое *компьютерная печать*?

К чему ищу так славу я?
Известно, в славе нет блаженства.
Но хочет всё душа моя
Во всём дойти до совершенства.

М.Ю. Лермонтов

Глава 12
ОБЕСПЕЧЕНИЕ КАЧЕСТВА ПРОГРАММНОГО СРЕДСТВА

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

12.1. Общая характеристика процесса обеспечения качества программного средства

Как уже говорилось в главе 4, спецификация качества определяет основные ориентиры (цели), которые на всех этапах разработки ПС так или иначе влияют при принятии различных решений на выбор подходящего варианта. Однако каждый примитив качества имеет свои особенности такого влияния, тем самым обеспечение его наличия в ПС может потребовать своих подходов и методов разработки ПС или отдельных его частей. Кроме того, ранее уже отмечалась противоречивость критериев качества ПС, а также и выражающих их примитивов качества: хорошее обеспечение одного какого-либо примитива качества ПС может существенно затруднить или сделать невозможным обеспечение некоторых других из этих примитивов. Поэтому существенная часть процесса обеспечения качества ПС состоит из поиска приемлемых компромиссов. Эти компромиссы частично должны быть определены уже в спецификации качества ПС: модель качества ПС должна конкретизировать требуемую степень присутствия в ПС каждого его примитива качества и определять приоритеты достижения этих степеней.

Обеспечение качества осуществляется в каждом технологическом процессе: принятые в нем решения в той или иной степени оказывают влияние на качество ПС в целом. В частности и потому, что значительная часть примитивов качества связана не столько со свойствами про-

грамм, входящих в ПС, сколько со свойствами документации. В силу отмеченной противоречивости примитивов качества весьма важно придерживаться выбранных приоритетов в их обеспечении. При этом следует придерживаться двух общих принципов:

- сначала необходимо обеспечить требуемую функциональность и надёжность 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 определён в спецификации качества ПС.

Обеспечение функциональности и надёжности ПС было рассмотрено в предыдущей главе. Ниже обсуждается обеспечение других критериев качества ПС.

12.2. Обеспечение лёгкости применения программного средства

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

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

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

Возбуждение исключительной ситуации во многих случаях означает, что возникла необходимость информировать пользователя о ходе выполнения программы. При этом выдаваемая пользователю информация должна быть простой для понимания (см. гл. 4). Однако исключительные ситуации возникают обычно на достаточно низком уровне модульной структуры программы, а создать понятное для пользователя сообщение можно, как правило, на более высоких уровнях этой структуры, где известен контекст, в котором были активизированы действия, приведшие к возникновению исключительной ситуации. Обработку исключительных ситуаций внутри модуля мы уже обсуждали в главе 8. Для обработки возникшей исключительной ситуации в другом модуле приходится принимать непростые решения. Применяемый часто способ передачи информации о возникшей исключительной ситуации по цепочке обращений к программным модулям (в обратном направлении) является тяжеловесным: он требует дополнительных проверок после возврата из модуля и часто усложняет само обращение к этим модулям за счет задания дополнительных параметров. Приемлемым решением является включение в операционную среду выполнения программ (в *исполнительную поддержку*) возможностей прямой передачи этой информации обработчикам исключительных ситуаций по динамически формируемой очереди таких обработчиков.

Пользовательский интерфейс представляет собой языковое средство взаимодействия пользователя с ПС. При разработке пользовательского интерфейса следует учитывать потребности, опыт и способности пользователя [65]. Поэтому потенциальные пользователи должны быть вовлечены в процесс разработки такого интерфейса. Большой эффект здесь дает прототипирование такого интерфейса. При этом пользователи должны получить доступ к прототипам пользовательского интерфейса, а их оценка различных возможностей используемого прототипа должна существенно учитываться при создании окончательного варианта пользовательского интерфейса.

В силу большого разнообразия пользователей и видов ПС существует много различных стилей пользовательских интерфейсов, при разработке которых могут использоваться разные принципы и подходы. Однако следующие важнейшие принципы следует соблюдать всегда [65]:

- базирование на терминах и понятиях, знакомых пользователю,
- единообразие,

- возможность исправлять собственные ошибки,
- возможность получения справочной информации как по запросу пользователя, так и генерируемой ПС.

В настоящее время широко распространены командные и графические пользовательские интерфейсы.

Командный пользовательский интерфейс предоставляет пользователю возможность обращаться к ПС с некоторым заданием (запросом), представляемым некоторым текстом (командой) на специальном командном языке (языке заданий). Достоинствами такого интерфейса является возможность его реализации на дешёвых алфавитно-цифровых терминалах и возможность минимизации требуемого от пользователя ввода с клавиатуры. Недостатками такого интерфейса являются необходимость изучения командного языка и достаточно большая вероятность ошибки пользователя при задании команды. В связи с этим командный пользовательский интерфейс обычно выбирают только опытные пользователи. Такой интерфейс позволяет им осуществлять быстрое взаимодействие с компьютером и предоставляет возможность объединять команды в процедуры и программы (см. например, язык Shell операционной системы Unix [28]).

Графический пользовательский интерфейс предоставляет пользователю возможности:

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

Графический пользовательский интерфейс позволяет

- размещать на экране множество различных окон, в которые можно выводить информацию автономно (независимо от вывода в другие окна);
- использовать графические объекты, называемые *пиктограммами* (или *иконками*), для обозначения различных информационных объектов или процессов;
- использовать *экранный указатель* для выбора объектов (или их элементов), размещённых на экране; экранный указатель управляется (перемещается) с помощью клавиатуры или мыши.

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

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

12.3. Обеспечение эффективности программного средства

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

С учетом сказанного, рекомендуется придерживаться следующих принципов для обеспечения эффективности ПС [10, 35]:

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

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

12.4. Обеспечение сопровождаемости программного средства

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

Изучаемость (подкритерий качества) ПС определяется составом и качеством документации по сопровождению ПС и выражается через такие примитивы качества ПС как *С-документированность*, *информативность*, *понятность*, *структурированность* и *удобочитаемость*. Последние два примитива качества и, в значительной степени, понятность связаны с текстами программных модулей. Вопрос о документации по сопровождению будет обсуждаться в следующей главе. Здесь мы лишь сделаем некоторые общие рекомендации относительно текстов программ (модулей).

При окончательном оформлении текста программного модуля целесообразно придерживаться следующих рекомендаций, определяющих практически оправданный стиль программирования [10, 35]:

- используйте в тексте модуля комментарии, объясняющие особенности принимаемых решений; по-возможности, включайте коммента-

рии (хотя бы в краткой форме) на самой ранней стадии разработки текста модуля;

- используйте осмысленные (мнемонические) и устойчиво различимые имена (оптимальная длина имени – 4-12 литер, цифры – в конце), не используйте сходные имена и ключевые слова;
- соблюдайте осторожность в использовании констант, уникальная константа должна иметь единственное вхождение в текст модуля: при её объявлении или, в крайнем случае, при инициализации переменной в качестве константы;
- не бойтесь использовать необязательные скобки – они обходятся дешевле, чем ошибки;
- размещайте не больше одного оператора в строке; для прояснения структуры модуля используйте дополнительные пробелы (отступы) в начале каждой строки; этим обеспечивается *удобочитаемость* текста модуля;
- избегайте *трюков*, т.е. таких приёмов программирования, когда создаются фрагменты модуля, основной эффект которых не очевиден или скрыт (завуалирован), например, побочные эффекты функций.

Структурированность текста модуля существенно упрощает его понимание. Обеспечение этого примитива качества подробно обсуждалось в главе 8. *Удобочитаемость* текста модуля может быть обеспечена автоматически путем применения специального программного инструмента – *форматера*.

Модифицируемость (подкритерий качества) ПС определяется, частично, некоторыми свойствами документации, а также свойствами, реализуемыми программным путём, и выражается через такие примитивы качества ПС, как *расширяемость*, *лёгкость изменения*, *структурированность* и *модульность*.

Расширяемость обеспечивается возможностями автоматически настраиваться на условия применения ПС по информации, задаваемой пользователем. К таким условиям относятся прежде всего конфигурация компьютера, на котором будет применяться ПС (в частности, объём и структура его памяти), а также требования конкретного пользователя к функциональным возможностям ПС (например, требования, которые определяют режим применения ПС или конкретизируют структуру информационной среды). К этим возможностям можно отнести и возможность добавления к ПС определённых компонент. Для реализации таких возможностей в ПС часто включается дополнительная компонента

(подсистема), называемая *инсталлятором*. Инсталлятор осуществляет прием от пользователя необходимой информации и настройку ПС по этой информации. Обычно решение о включении в ПС такой компоненты принимается в процессе разработки архитектуры ПС.

Модифицируемость (примитив качества) обеспечивается такими свойствами документации и свойствами, реализуемыми программным путём, которые облегчают внесение изменений и доработок в документацию и программы ПС вручную (возможно, с определённой компьютерной поддержкой). В спецификации качества могут быть указаны некоторые приоритетные направления и особенности развития ПС. Эти указания должны быть учтены при разработке архитектуры ПС и модульной структуры его программ. Общая проблема сопровождения ПС – обеспечить, чтобы все его компоненты (на всех уровнях представления) оставались согласованными в каждой новой версии ПС. Этот процесс обычно называют *управлением конфигурацией* (*configuration management*). Чтобы помочь управлению конфигурацией, необходимо, чтобы связи и зависимости между документами и их частями фиксировались в специальной документации по сопровождению [13]. Эта проблема усложняется, если в процессе доработки может находиться сразу несколько версий ПС (в разной степени завершённости). Тогда без компьютерной поддержки довольно трудно обеспечить согласованность документов в разных конфигурациях. Поэтому в таких случаях в ПС включается дополнительная компонента (подсистема), которую можно назвать *конфигуратором*. С такой компонентой связывают специальную базу данных (или специальный раздел в базе данных), в которой фиксируются связи и зависимости между документами и их частями для всех версий ПС. Обычно решение о включении в ПС такой компоненты принимается в процессе разработки архитектуры ПС. Для обеспечения этого примитива качества в документацию по сопровождению включают специальное руководство, которое описывает, какие части ПС являются аппаратно- и программно-зависимыми, и как возможное развитие ПС учтено в его строении (конструкции).

Структурированность и *модульность* упрощают ручную модификацию программ ПС.

12.5. Обеспечение мобильности

Проблема мобильности возникает из-за того, что быстрое развитие компьютерной техники и аппаратных средств делает жизненный цикл

многих больших программных средств (программных систем) намного продолжительнее периода «морально» оправданного существования компьютеров и аппаратуры, для которых первоначально создавались эти программные средства. Поэтому обеспечение критерия мобильности для таких ПС является весьма важной задачей.

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

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

Если ПС зависит от устройств (аппаратуры), то в спецификации качества должна быть описана эта компьютерно-аппаратная среда (будем ее называть *аппаратной платформой* [34]). Избавиться от этой зависимости можно за счет такого примитива качества ПС, как автономность. Как правило, ПС строится в рамках некоторой *операционной системы* (ОС), которая может спрятать специфику аппаратной платформы и, тем самым, сделать ПС независимым от устройств. Но тогда ПС не будет обладать свойством автономности. В этом случае в спецификации качества должна быть указана та программная среда, над которой строится ПС (будем эту среду называть *операционной платформой* [34]). Таким образом, мобильность ПС будет непосредственно связана с мобильностью используемой ОС: перенос ПС на другую аппаратную платформу осуществляется автоматически, если будет осуществлен перенос на эту платформу используемой ОС. Но обеспечение мобильности ОС является самостоятельной и довольно трудной задачей.

Итак, для обеспечения мобильности ПС нужно решить две задачи:

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

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

Рис. 12.1. Рекомендуемая архитектура мобильного ПС

Модульность ПС позволяет сформировать указанные слои, выделяя программные модули с требуемыми свойствами. *Модульность* и *структурированность* оболочки и ядра позволяют обеспечить эти слои

свойством *лёгкости изменения*. При этом желательно, чтобы каждый модуль этих слоёв был ориентирован на реализацию каких-либо функций управления четко выделенной компоненты аппаратно-операционной среды. Для этого используются такие методы как унификация интерфейсов, стандартизация протоколов и т. п. [34].

Вопросы к главе 12

12.1. Какие задачи приходиться решать при обеспечении коммуникабельности ПС?

12.2. Какие возможности предоставляет пользователю графический пользовательский интерфейс?

12.3. Как нужно действовать для обеспечения эффективности ПС?

12.4. Что такое *инсталятор программного средства*?

12.5. Что такое *управление конфигурацией ПС*?

12.6. Какая архитектура ПС рекомендуется для обеспечения его мобильности?

В начале было Слово...
Библия, Новый завет,
От Иоанна святое благовествование

Глава 13
ДОКУМЕНТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА

Виды документов, создаваемых и используемых в процессе разработки программных средств. Состав и характеристика пользовательской документации. Категории пользователей. Виды документов по сопровождению программных средств.

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

При разработке ПС создаётся и используется большой объём 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 [65]:

- Документация управления разработкой ПС.
- Документация ПС.

Документация управления разработкой ПС (*software process documentation*) протоколирует процессы разработки и сопровождения *программного средства*, определяет условия этой разработки, исполнителей и порядок выполнения отдельных её работ (заданий), регулирует отношения между этими исполнителями внутри коллектива разработчиков, а также между коллективом разработчиков и *менеджерами ПС* (*software managers*) – лицами, управляющими разработкой ПС. Эти документы могут быть следующих типов [65]:

- *Планы, оценки, расписания.* Эти документы создаются менеджерами для прогнозирования и управления процессами разработки и сопровождения ПС.
- *Отчеты об использовании ресурсов в процессе разработки.* Они создаются менеджерами.

- *Стандарты.* Эти документы предписывают разработчикам, каким принципам, правилам, соглашениям они должны следовать в процессе разработки ПС. Стандарты могут быть как международными или национальными, так и специально созданными для организации, в которой ведётся разработка ПС.
- *Рабочие документы.* Это основные технические документы, определяющие связи 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.

Документация ПС (*software product documentation*) представляет собой совокупность документов, входящих в состав *программного средства*, которые описывают *программное средство* как с точки зрения применения его программ пользователями, так и с точки зрения его построения разработчиками и сопроводителями. Здесь следует отметить, что эти документы будут использоваться не только на стадии эксплуатации ПС (в её 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 применять данное ПС [56, 65]. Она необходима, если ПС предполагает какое-либо взаимодействие с пользователями. К такой документации относятся документы, которыми должен руководствоваться пользователь при *инсталляции* ПС (при установке ПС с соответствующей настройкой на среду применения ПС), при применении ПС для решения своих задач и при управлении ПС (например, когда разрабатываемое ПС будет взаимодействовать с другими системами). Эти документы частично затра-

гивают вопросы сопровождения ПС, но не касаются вопросов, связанных с изменением программ.

В связи с этим следует различать две категории пользователей ПС: ординарных пользователей ПС и администраторов ПС. *Ординарный пользователь ПС* (*end-user*) – это лицо, использующее (применяющее) ПС для решения своих задач. 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*) – это лицо, которое управляет использованием ПС ординарными пользователями и осуществляет сопровождение ПС, не связанное с изменением его программ. Например, он может регулировать права доступа к ПС между ординарными пользователями, поддерживать связь с поставщиками ПС или выполнять определённые 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 пользователей* здесь понимается контингент пользователей ПС, у которого есть необходимость в определённой пользовательской документации ПС [56]. Хороший пользовательский документ должен существенно учитывать особенности аудитории пользователей, для которой он предназначен. Пользовательская документация должна содержать документы, необходимые для каждой такой аудитории. Под *режимом использования* документа понимается способ, определяющий, каким образом используется этот документ. Обычно пользователю весьма 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 [56, 65] можно считать типичным следующий состав пользовательской документации для достаточно больших ПС:

- *Общее функциональное описание ПС.* Оно дает краткую характеристику функциональных возможностей ПС и предназначено для пользователей, которые должны решить, насколько необходимо им данное ПС.

- *Руководство по инсталляции ПС.* Предназначено для администраторов ПС. Оно должно детально предписывать, как устанавливать системы в конкретной среде, в частности, должно содержать описание компьютерно-считываемого носителя, на котором поставляется ПС, файлы, представляющие ПС, и требования к минимальной конфигурации аппаратуры.
- *Инструкция по применению ПС.* Предназначена для ординарных пользователей и содержит необходимую информацию по применению ПС, организованную в форме, удобной для её изучения.
- *Справочник по применению ПС.* Предназначен для ординарных пользователей и содержит необходимую информацию по применению ПС, организованную в форме удобной для избирательного поиска отдельных деталей.
- *Руководство по управлению ПС.* Предназначено для администраторов ПС. Оно должно описывать, какие сообщения генерирует ПС, когда оно взаимодействует с другими системами, и как должен реагировать администратор на эти сообщения. Кроме того, если ПС использует системную аппаратуру, этот документ может объяснять, как сопровождать эту аппаратуру.

Как уже говорилось ранее (см. гл. 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, [56]), 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*) – совокупность документов, которые объясняют разработчиками и сопроводителями, как они должны действовать, чтобы изменять данное *про-*

граммное средство. Эта документация необходима, если ПС ориентировано на то, что в процессе его использования допускается внесение в него определенных изменений. Как уже отмечалось, сопровождение – это продолжающаяся разработка. Поэтому в случае необходимости изменения 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). Чтобы понять строение и процесс разработки изменяемого ПС, команда разработчиков-сопроводителей должна изучить эту документацию, а затем внести в неё необходимые изменения, повторяя в значительной степени технологические процессы, с помощью которых создавалось первоначальное ПС.

Документацию по сопровождению ПС можно разбить на две группы:

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

(2) документация, помогающая 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 её программы (подсистемы).
- Для каждой программы ПС – описание ее модульной структуры, включая внешнюю спецификацию каждого включенного в нее модуля.
- Для каждого модуля – его спецификация и описание его строения (*design description*).
- Тексты модулей на выбранном языке программирования.
- 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 определённых свойств программ.

Для обеспечения приемлемого качества этой документации полезно следовать общепринятым рекомендациям и стандартам [50 - 55].

Документация второй группы содержит

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

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

Вопросы к главе 13

13.1. Что такое *менеджер ПС*?
13.2. Что такое *ординарный пользователь ПС*?
13.3. Что такое *администратор ПС*?
13.4. Что такое *руководство по инсталляции ПС*?
13.5. Что такое *руководство по управлению ПС*?
13.6. Что такое *руководство по сопровождению ПС*?

Губа не дура, язык не лопата.
Народная поговорка

Глава 14
УПРАВЛЕНИЕ РАЗРАБОТКОЙ И АТТЕСТАЦИЯ ПРОГРАММНОГО СРЕДСТВА

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

14.1. Назначение и процессы управления разработкой программного средства

Управление разработкой ПС – это деятельность, направленная на обеспечение необходимых условий для работы коллектива разработчиков ПС, на планирование и контроль деятельности этого коллектива с целью обеспечения требуемого качества ПС, выполнения сроков и бюджета разработки ПС [33, 65]. Часто эту деятельность называют также *управлением программным проектом* (*software project management*). Здесь под *программным проектом* (*software project*) понимают всю совокупность работ, связанную с разработкой ПС, а ход выполнения этих работ называют *развитием программного проекта* (*software project progress*). К необходимым условиям работы коллектива относятся помещения, аппаратно-программные средства разработки, документация и материально-финансовое обеспечение. Планирование и контроль включают в себя разбиение всего процесса разработки ПС на отдельные конкретные работы (задания), подбор и расстановку исполнителей, установление сроков и порядка выполнения этих работ, оценку качества выполнения каждой работы. Финальной частью этой деятельности является организация и проведение аттестации (сертификации) ПС, которой завершается стадия разработки ПС. Влияние правильной расстановки исполнителей на обеспечение надёжности ПС уже обсуждалось в главе 2. Вопросы документации обсуждались в предыдущей главе. Теперь кратко обсудим другие вопросы управления разработкой.

Хотя виды деятельности по управлению разработкой ПС могут быть весьма разнообразными в зависимости от специфики разрабаты-

ваемого ПС и организации работ по его созданию, можно выделить некоторые общие процессы (виды деятельности) по управлению разработкой ПС:

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

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

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

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

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

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

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

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

Подбор и оценка персонала коллектива разработчиков ПС – это деятельность, связанная с формированием коллектива разработчиков ПС. Имеющийся в распоряжении штат разработчиков далеко не всегда будет подходящим по квалификации и опыту работы для данного проекта. Поэтому приходится частично вербовать требуемый персонал, а частично организовывать дополнительное обучение имеющихся разработчиков. В любом случае в формируемом коллективе хотя бы один его член должен иметь опыт разработки программных средств (систем), сопоставимых с ПС, который требуется разработать. Это поможет избежать многих простых ошибок в развитии проекта.

14.2. Структура управления разработкой программных средств

Разработка ПС обычно производится в организации, в которой одновременно могут вестись разработки ряда других программных средств. Для управления всеми этими программными проектами используется иерархическая структура управления. Традиционная структура такого рода обсуждена в работе [65]. Она представлена на рис. 14.1.

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

Рис. 14.1. Структура управления разработкой программных средств.

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

По каждому программному проекту назначается свой менеджер, который управляет развитием этого проекта. Ему непосредственно под-

чинены лидеры бригад разработчиков. *Менеджер проекта* осуществляет планирование и составление расписаний работы этих бригад по разработке соответствующего ПС (см. п. 14.3).

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

Наиболее распространены три подхода к организации бригад разработчиков [8, 49, 65]:

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

В *обычной бригаде* старший программист (*лидер бригады*) непосредственно руководит работой младших программистов. Недостатки такой организации непосредственно связаны со спецификой разработки ПС: программисты разрабатывают сильно связанные части программной подсистемы, сам процесс разработки состоит из многих этапов, каждый из которых требует особенных способностей от программиста, ошибки отдельного программиста могут препятствовать работе других программистов. Успех работы такой бригады достигается в том случае, когда её руководитель является компетентным программистом, способным предъявлять к членам бригады разумные требования и умеющим поощрять хорошую работу. В *неформальной демократической бригаде* поручаемая ей работа обсуждается совместно всеми её членами, а задания между её членами распределяются согласованно в зависимости от способностей и опыта этих членов. Один из членов этой бригады является *лидером (руково-*

дителем) бригады, но он также выполняет и некоторые задания, распределяемые между членами бригады. Неформальные демократические бригады могут весьма успешно справляться с порученной им работой, если большинство членов бригады являются опытными и компетентными специалистами. Если же неформальная демократическая бригада состоит, в основном, из неопытных и некомпетентных членов, в деятельности бригады могут возникать большие трудности. Без наличия в бригаде хотя бы одного квалифицированного и авторитетного члена, способного координировать и направлять работу членов бригады, эти трудности могут привести к неудаче проекта.

В *бригаде ведущего программиста* за разработку порученной программной подсистемы несет полную ответственность один человек, называемый *ведущим программистом (chief programmer)* и являющийся *лидером бригады*: он сам конструирует эту подсистему, составляет и отлаживает необходимые программы, пишет документацию к подсистеме. Ведущий программист выбирается из числа опытных и одарённых программистов. Все остальные члены такой бригады, в основном, создают условия для наиболее продуктивной работы ведущего программиста. Организацию такой бригады обычно сравнивают с хирургической бригадой [8, 65]. Ядро бригады ведущего программиста составляют три члена бригады: помимо ведущего программиста в него входит дублер ведущего программиста и администратор базы данных разработки. *Дублер ведущего программиста (backup programmer)* также является квалифицированным и опытным программистом, способным выполнить любую работу ведущего программиста, но сам он эту работу не делает. Главная его обязанность – быть в курсе всего, что делает ведущий программист. Он выступает в роли оппонента ведущего программиста при обсуждении его идей и предложений, но решения по всем обсуждаемым вопросам принимает единолично ведущий программист. *Администратор базы данных разработки (librarian)* отвечает за сопровождение всей документации (включая версии программ), возникающей в процессе разработки программной подсистемы, и снабжает членов бригады информацией о текущем состоянии разработки. Эта работа выполняется с помощью соответствующей инструментальной компьютерной поддержки (см. гл. 16). В зависимости от объема и характера порученной работы в бригаду могут быть включены дополнительные члены, такие как

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

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

Важное место в управлении разработкой ПС отводится управлению обеспечением качества. Для руководства этой деятельностью назначается специальный менеджер, подчиненный непосредственно директору, – *менеджер по качеству*. Ему непосредственно подчинены формируемые бригады по контролю качества. Эти бригады работают с отдельными проектами, но непосредственно соответствующим менеджерам проектов не подчинены, сохраняя тем самым свою независимость от них.

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

Существенную роль в управлении качеством ПС играют программные стандарты [33, 65]. Они фиксируют удачный опыт высоко квалифицированных специалистов по разработке ПС для различных их классов и для разных моделей их качества. Следование подходящим стандартам может существенно облегчить достижение поставленных целей относительно качества ПС, а также упростить смотр по контролю качества. Кроме того, стандарты способствуют формированию взаимопонимания внутри коллектива разработчиков и упрощают процесс обучения новых членов этого коллектива.

Различают два вида таких стандартов:

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

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

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

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

Бригада по контролю качества состоит из ассистентов (рецензентов) по качеству ПС. Она проводит смотры тех или иных частей ПС или всего ПС в целом с целью поиска возникающих проблем в процессе его

разработки. Смотру подлежат все программные компоненты и документы, включаемые в ПС, а также процессы их разработки. В процессе смотра учитываются требования, сформулированные в спецификации качества ПС, в частности, проверяется соответствие исследуемого документа или технологического процесса стандартам, указанным в этой спецификации. В результате смотра формулируются замечания, которые могут фиксироваться письменно или просто передаваться разработчикам устно.

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

14.3. Планирование и составление расписаний по разработке программного средства

Общее представление об этой деятельности можно составить по её описанию на псевдокоде (см. гл. 8), приведенном на рис. 14.2 (см. также [65]). Это описание показывает, что планирование и составление расписаний по разработке ПС представляет собой итеративный процесс, который заканчивается только после прекращения работ по самому программному проекту.

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

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

- разделение всей работы, необходимой для выполнения проекта, на отдельные самостоятельно выполняемые задания;
- составление сетевого графика выполнения заданий;
- составление гистограммы выполнения заданий;
- расстановка исполнителей заданий.

```
Определить проектные ограничения.
Сделать начальную оценку параметров проекта.
Установить вехи развития проекта и их сроки.
ПОКА  проект не является завершённым или
   прекращённым (аннулированным)
ДЕЛАТЬ
   Составить расписание проекта.
   Инициировать процессы, соответствующие
      расписанию.
   ПОДОЖДАТЬ.
   Просмотреть развитие проекта.
   Скорректировать параметры проекта.
   Оценить влияние изменения параметров проекта
      на расписание проекта.
   Уточнить проектные ограничения и сроки.
   ЕСЛИ  возникли проблемы  ТО
      Инициировать технический пересмотр и
         возможную ревизию проекта.
   ВСЁ ЕСЛИ
ВСЁ ПОКА
```

Рис. 14.2. Описание на псевдокоде процесса планирования и составления расписаний по разработке ПС

При выделении самостоятельных заданий для каждого из них оценивается время его выполнения и его зависимость от других заданий с точки зрения порядка выполнения. *Сетевой график* представляет собой

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

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

14.4. Аттестации программного средства

Завершающим этапом разработки ПС является аттестация ПС, подводящая итог всей разработке. *Аттестация* (*certification*) ПС – это авторитетное подтверждение качества ПС [22, 35]. Обычно для аттестации ПС создаётся аттестационная комиссия из экспертов, представителей заказчика и представителей разработчика. Эта комиссия проводит *приёмо-сдаточные испытания* ПС с целью получения необходимой информации для оценки его качества. Под *испытанием* ПС здесь понимают [22, 34] процесс проведения комплекса мероприятий, исследующих пригодность ПС для успешной его эксплуатации (применения и

сопровождения) в соответствии с требованиями заказчика. В этом процессе проверяется полнота и исследуется качество представленной программной документации, производится необходимое тестирование программ, входящих в состав ПС, а также исследуются и другие свойства ПС, декларированные в его спецификации качества. На основе полученной информации комиссия должна установить, в какой степени ПС выполняет декларированные функции и в какой степени ПС обладает декларированными примитивами и критериями качества. Решение аттестационной комиссии о произведённой оценке качества ПС фиксируется в соответствующем документе (сертификате), который подписывается членами комиссии.

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

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

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

Для оценки некоторых примитивов качества ПС используется *тестирование* [22, 31, 34, 35]. К таким примитивам относятся прежде всего завершённость ПС, а также его точность, устойчивость, защищённость и другие примитивы качества. Этот вопрос уже обсуждался в главе 10. Однако во время приёмо-сдаточных испытаний нет необходи-

мости проведения тестирования ПС в полном объёме (это может слишком дорого стоить). Аттестационная комиссия должна прежде всего изучить предъявленную документацию по проведённому разработчиками тестированию ПС и лишь выборочно пропустить предъявленные тесты. Дополнительные тесты составляются, если у комиссии возникают определённые сомнения в полноте тестирования, проведённого разработчиками. Кроме того, обычно работоспособность и некоторые показатели качества ПС демонстрируются на решении контрольных задач, предъявляемых заказчиком.

В некоторых случаях для оценки качества ПС проводятся дополнительные полевые и промышленные испытания [10, 31]. *Полевые* испытания ПС – это демонстрация ПС вместе с технической системой, которой управляет это ПС, с обеспечением тщательного наблюдения за поведением ПС. Заказчикам должна быть предоставлена возможность задания собственных контрольных примеров, в частности, с выходом в критические режимы работы технической системы, а также с вызовом в ней аварийных ситуаций. *Промышленные* испытания ПС – это процесс передачи ПС в постоянную эксплуатацию пользователям, представляющий собой опытную эксплуатацию ПС (см. гл. 10) пользователями со сбором информации об особенностях поведения ПС и его эксплуатационных характеристиках.

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

Аттестация ПС похожа на смотр различных компонент ПС в процессе управления качеством ПС, однако имеются и существенные различия [65]. Во-первых, смотр проводится менее представительной группой специалистов. Во-вторых, в процессе смотра не производится

полной оценки качества ПС, а выявляются лишь связанные с обозреваемой компонентой (документом) отдельные просчёты и нарушения требований относительно качества ПС. При этом не требуется немедленного устранения выявленных недостатков, если они не мешают проведению последующих работ. *Целью* же аттестации является проверка и фиксация реальных показателей качества предъявленного ПС [34]. Если аттестационная комиссия подтверждает, что предъявленное ПС соответствует всем требованиям относительно его качества, сформулированным во внешнем описании этого ПС, то считается, что его разработка завершена успешно и заказчик обязан принять это ПС. Если же будут обнаружены отступления от этих требований, то должны приниматься определённые решения о продолжении или прекращении разработки предъявленного ПС, но это уже вопрос взаимоотношений между заказчиком и разработчиками. Таким образом, аттестационная комиссия, подписывая документ о произведённой оценке качества ПС, принимает на себя определенную ответственность за гарантию качества этого ПС. Но здесь имеются определённые правовые проблемы, обсуждение которых выходит за рамки темы этой главы.

Вопросы к главе 14

14.1. Что такое *управление разработкой ПС*?

14.2. Что такое *менеджер программного проекта*?

14.3. Что такое *неформальная демократическая бригада разработчиков ПС*?

14.4. Что такое *бригада ведущего программиста*?

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

14.6. Что такое *аттестация ПС*?

Не умножай число имеющихся сущностей.
Вильям Оккам

Глава 15
ОЪЕКТНЫЙ ПОДХОД К РАЗРАБОТКЕ ПРОГРАММНЫХ СРЕДСТВ

Сущность объектного подхода к разработке программных средств. Объектное моделирование как содержание этапа внешнего описания при объектном подходе. Особенности этапов конструирования и кодирования программного средства при объектном подходе.

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

Окружающий нас мир состоит из объектов и отношений между ними [46]. Согласно В. Далю [15] объект (предмет) – это всё, что представляется чувствам (объект вещественный) или уму (объект умственный). Таким образом, *объект* воплощает некоторую сущность и имеет некоторое состояние, которое может изменяться со временем как следствие влияния других объектов, находящихся с первым в каких-либо отношениях. 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, 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 определённые (в этом случае говорят: объекты определённого класса). Одноместное отношение называется *простым свойством объекта* (соответствующего класса). Многоместное отношение объектов будем называть *ассоциативным свойством объекта*, если этот объект участвует в этом

отношении. Состояние объекта может быть изучено по значению простых или ассоциативных свойств этого объекта. Множество всех объектов, которые обладают каким-то общим набором свойств, называется *классом объектов*.

В процессе познания или изменения окружающего нас мира мы всегда принимаем в рассмотрение ту или иную упрощённую модель мира (тот или иной *модельный мир*), в которую включаем объекты и отношения некоторых интересующих нас классов из окружающего нас мира. Каждый объект, имеющий внутреннюю структуру, может представлять свой модельный мир, включающий объекты этой структуры и отношения, которые их связывают.

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. Простые свойства этого объекта могут задаваться непосредственно в виде отдельных компонент этой структуры, либо специальными функциями над этой структурой данных. Ассоциативные свойства (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 соответствующего свойства, либо процедуру, осуществляющую изменение состояний некоторых из представлений объектов, связываемых представляемым отношением. В пассивной форме такое отношение может быть представлено некоторой структурой данных, интерпретируемой на основании принятых соглашений с помощью общих процедур, независящих от конкретных отношений (например, реляционная база данных). В любом случае представление отношения определяет некоторые действия по обработке данных.

При исследовании модельного мира пользователи могут поразному получать информацию от компьютера.

В одних случаях пользователей может интересовать получение информации об отдельных свойствах определённых объектов или результаты какого-либо взаимодействия между некоторыми объектами модельного мира. Для удовлетворения таких запросов разрабатываются соответствующие ПС, которые выполняют интересующие пользовате-

лей функции, или подходящие информационные системы, способные выдавать информацию об интересующих пользователей отношениях. В начальный период развития компьютерной техники (при не достаточно высокой мощности компьютеров) такой подход к исследованию модельного мира был вполне естественным. Именно он и провоцировал *реляционный* (*процедурный*, *функциональный* и т. п.) подход к разработке ПС, который был подробно рассмотрен в предшествующих главах. Сущность этого подхода состоит в преимущественном использовании *декомпозиции отношений* (*процедур, функций*) для описания и построения ПС. При этом сами объекты модельного мира, с которыми связаны 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 обеспечивалась эффективная реализация требуемых функций, но не создавалось цельного и адекватного компьютерного представления модельного мира, интересующего пользователя. Попытки даже незначительного расширения объёма и характера информации об этом модельном мире, которую можно получить от ПС, могло потребовать серьёзной переделки этого ПС.

В других случаях пользователя может интересовать наблюдение за изменением состояний объектов модельного мира в результате их взаимодействий. Это требует использования подходящих информационных моделей таких объектов, создания программных средств, моделирующих процессы взаимодействия объектов модельного мира, и предоставление пользователю доступа к этим информационным моделям (к *пользовательским* объектам). С помощью традиционных методов разработки это оказалось довольно трудоёмкой задачей. Наиболее полно отвечает решению этой задачи *объектный* подход к разработке ПС. Сущность его состоит в преимущественном использовании *декомпозиции объектов* для описания и построения ПС. При этом функции (операции), выполняемые таким ПС, будут выражаться через операции над объектами других уровней, т. е. их декомпозиция будет существенно зависеть от декомпозиции объектов.

С точки зрения разработчиков ПС следует различать следующие категории объектов (и, соответственно, их классов):

- объекты модельного (вещественного или умственного) мира,
- информационные модели объектов реального мира (будем называть их *пользовательскими объектами*),

- объекты процесса выполнения программ,
- объекты процесса разработки ПС (*технологические объекты программирования*).

Следует заметить, что только пользовательские объекты и объекты процесса выполнения программы являются программными объектами, т. е. объектами непосредственно используемыми в программах. Под *программным объектом* будем понимать некоторый фрагмент информационной среды, который пригоден для хранения данных определённого типа и с которым связан некоторый набор применимых к нему операций. *Пользовательский объект* размещается во внешней информационной среде и пользователь может непосредственно применять связанные с ним операции.

В зависимости от способа представления в компьютере модельного мира и характера взаимодействия с ним со стороны пользователя следует различать пассивные и активные программные объекты. Программный объект будем называть *пассивным*, если его состояние (содержащиеся в нём данные) не имеет программных частей, способных находиться в процессе выполнения. Операции над таким объектом применяются под воздействием некоторой *внешней* по отношению к этому объекту *активной силы*, исходящей либо от пользователя, либо от какого-либо программного фрагмента в процессе его выполнения. Программный объект будем называть *активным*, если его состояние (содержащиеся в нём данные) имеют программные части, способные находиться в процессе выполнения. Активный объект, у которого какие-либо программные части находятся в активном состоянии, способен воспринимать сообщения или сигналы из операционной среды, в которую он погружён, и самостоятельно выполнять некоторые операции как реакцию на эти сообщения или сигналы. Таким образом, можно считать, что активный объект обладает *внутренней активной силой*.

Когда говорят об *объектно-ориентированном* подходе к разработке ПС, имеют в виду объектный подход с ориентацией на описание объектов модельного мира и построением их информационных моделей, причем используются, в основном, активные объекты. При этом многие процессы разработки ПС приобретают специфические («объектные») черты:

- использование системы понятий, позволяющих описывать объекты и их классы,
- декомпозиция объектов является основным средством упрощения ПС,

- использование внепрограммных абстракций для упрощения процессов разработки,
- предпочтение (приоритет) разработки структуры данных перед реализацией функций.

Основные из этих специфических особенностей разработки ПС покажем в рамках водопадной модели технологии программирования.

15.2. Особенности объектного подхода к разработке внешнего описания программного средства

При объектном подходе этап внешнего описания ПС оказывается существенно более ёмким и содержательным по сравнению с реляционным подходом [9, 27, 43, 64].

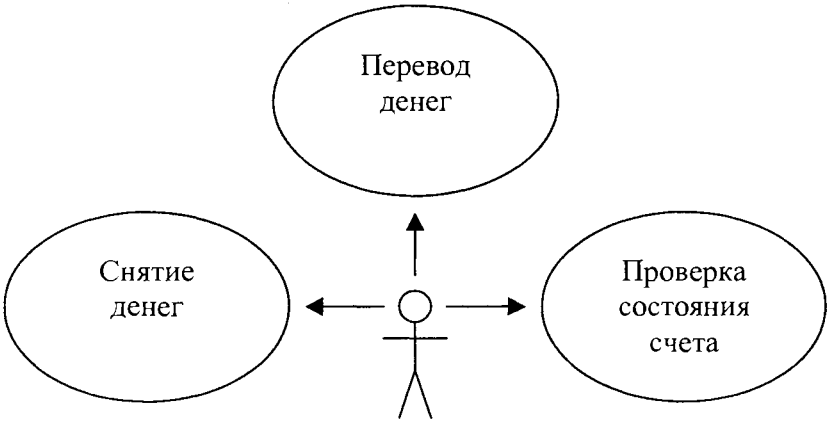


Рис. 15.1. Диаграмма вариантов использования для банкомата

Определение требований заключается в неформальном описании модельного мира, который пользователь собирается изучать или просто использовать с помощью требуемого ПС. При этом повышается роль прототипирования, которое при этом подходе часто окупается за счёт уменьшения объёма работы на последующих этапах разработки ПС. Часто определение требований завершается разработкой *диаграммы вариантов использования*, в которой фиксируются, в каких случаях будет использоваться ПС. Это позволяет при разработке ПС ограничиться только такими ее функциональными возможностями, которые поддер-

живают эти случаи (варианты) использования. По существу, диаграмма вариантов использования представляет собой описание некоторого *контекста использования* (см. гл. 4). На рис. 15.1. приведён пример диаграммы вариантов использования [27].

Существенно изменяется содержание процесса спецификации требований: вместо разработки функциональной спецификации ПС создается формальное описание модельного мира, состоящее из трех частей:

- объектной модели,
- динамической модели,
- функциональной модели.

Назначение этих частей можно образно определить следующим образом [64]: объектная модель определяет то, с чем что-то случается; динамическая модель определяет, когда это случается; функциональная модель определяет то, что случается.

Объектная модель показывает статическую объектную структуру модельного мира, который должно представлять разрабатываемое программное средство (программная система). Она включает определения используемых классов объектов и отношений между этими классами, а также определение используемых объектов этих классов и отношения между этими объектами.

Обычно *класс объектов* в объектной модели представляется в виде тройки

(Имя класса, Список атрибутов, Список операций)

Каждый атрибут представляется некоторым именем и может принимать значения определённого типа. По существу, атрибут класса выражает некоторое простое свойство объектов этого класса. Представление некоторых простых свойств объектов атрибутами весьма удобно, особенно когда по значениям этих атрибутов осуществляется классификация объектов. Операции, указываемые в представлении класса, отражают другие свойства объектов этого класса (как простые, так и ассоциативные). Они показывают, что можно делать с объектами этого класса (или что могут делать сами эти объекты).

В объектной модели *отношение между объектами* некоторых классов обобщаются в *отношения между* этими *классами*. При этом используются, как правило, только одноместные и двуместные отношения между объектами. Более сложные отношения приводят к неоправданному усложнению объектных моделей, а с другой стороны, такие отношения всегда могут быть сведены к двуместным за счет определе-

ния дополнительных классов. Одноместные отношения между объектами называют *атрибутами объекта*, причем некоторые атрибуты объекта получаются из атрибутов класса присвоением им конкретных значений. Отношение между двумя (и более) объектами называют *связями*, а их обобщение (отношение между классами) обычно называют *ассоциациями*. Ассоциации определяют допустимые связи между объектами. Различают следующие виды ассоциаций:

- взаимодействие состояний объектов,
- агрегирование (структурирование) объектов,
- абстрагирование (порождение) классов.

Ассоциация «взаимодействие», по существу, означает, что объекты классов, находящихся в таком отношении, могут быть параметрами некоторых операций. Ассоциация «агрегирование» означает, что объект одного из классов, находящихся в таком отношении, включает в себя (или может включать как часть) объекты другого из этих классов. Ассоциация «абстрагирование» означает, что один из классов, находящихся в таком отношении, наследует свойства другого из этих классов, но может обладать также и другими (дополнительными) свойствами.

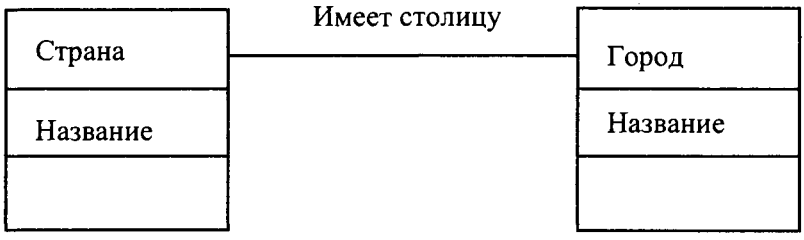


Рис. 15.2. Пример отношения между классами объектов

Для представления объектной модели часто используются графические языки спецификации объектов (например, язык UML [43]). На таких языках классы и объекты задаются прямоугольниками, в которых указывается специфицирующая их информация. Для задания отношений между двумя классами соответствующие им прямоугольники связываются линией, снабжённой различными графическими значками и некоторыми надписями. Графические значки специфицируют характер (вид) отношения между этими классами, а надписи обеспечивают полную идентификацию этого отношения (делают его конкретным). На-

пример, на рис. 15.2 заданное отношение между классами Страна и Город имеет характер «один к одному». Более конкретно это отношение означает, что каждый объект класса Страна обязательно связан отношением «имеет столицей» с одним и только одним объектом класса Город, и этот объект класса Город не связан таким отношением ни с каким другим объектом класса Страна.

Для описания декомпозиции объектов используется отношение вида агрегирования. Например, отношение «программа состоит из одного или нескольких модулей» представлено на рис. 15.3.

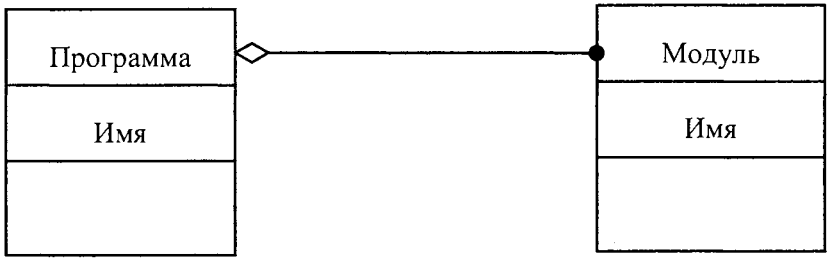


Рис. 15.3. Пример отношения агрегирования между классами объектов

Следует заметить, что объектная модель соответствует описанию внешней информационной среды при реляционном подходе.

Динамическая модель показывает допустимые последовательности изменений состояний объектов из объектной модели того модельного мира, который должно представлять разрабатываемое программное средство (программная система). Она описывает последовательности операций в ответ на внешние сигналы (взаимодействия) без рассмотрения того, что эти операции делают. Динамическая модель необходима, если в соответствующей объектной модели имеются активные объекты.

Основные понятия динамической модели: события и состояния объектов. Под *событием* здесь понимается элементарное воздействие одного объекта на другой, происходящее в определённый момент времени. Одно событие может логически предшествовать другому или быть не связанным с другим. Другими словами, события в динамической модели *частично упорядочены*. Под *состоянием объекта* здесь понимается совокупность значений атрибутов объекта и представления текущих связей этого объекта с другими объектами. Состояние объекта

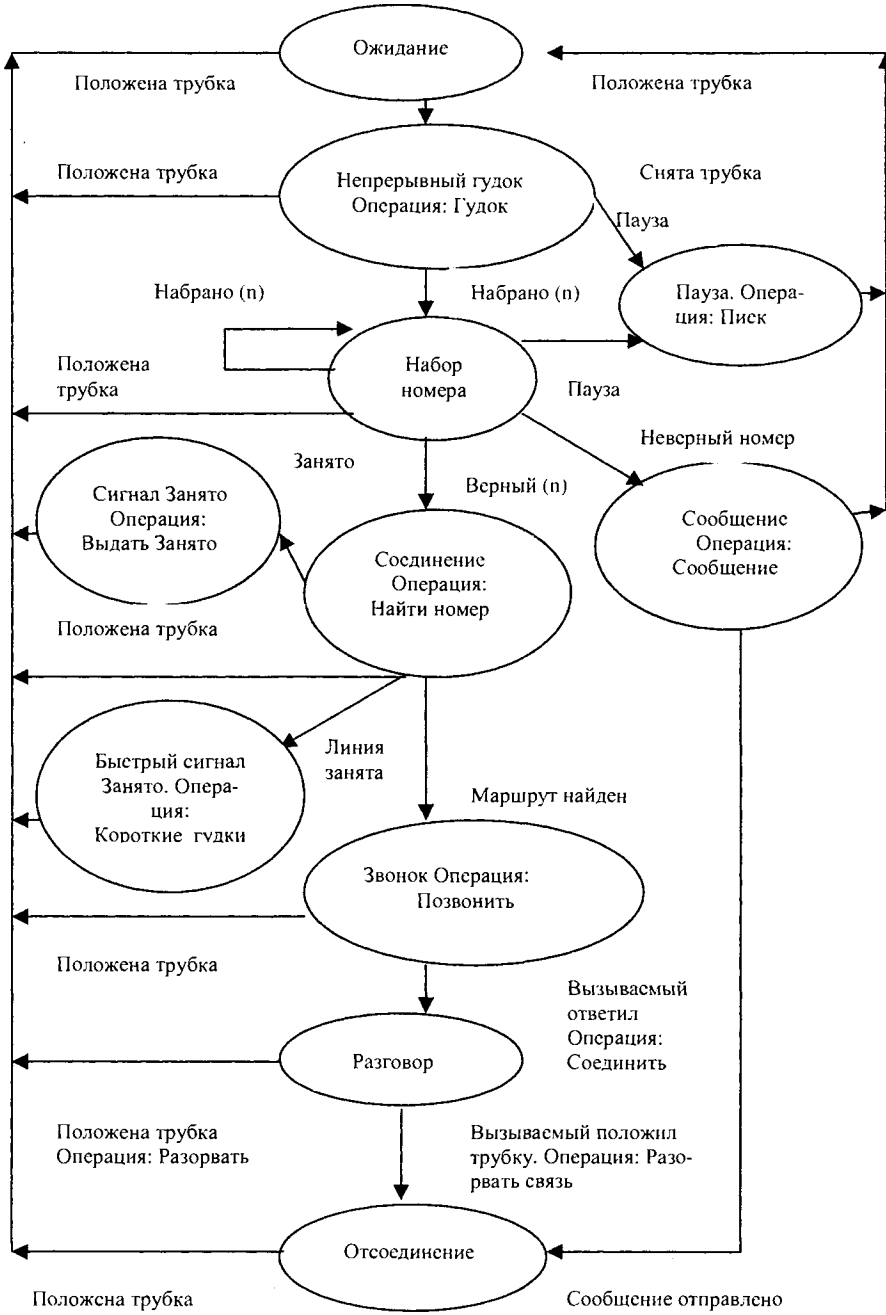
связывается с интервалом времени между некоторыми двумя событиями, на которые реагирует этот объект. Объект *переходит* из одного состояния в другое в результате реакции на некоторое событие (в конце интервала, связанного с этим состоянием).

В связи с этим в динамической модели для каждого класса активных объектов строится своя *диаграмма состояний*. Она представляет собой граф, вершинами которого являются состояния, а дугами – переходы между этими состояниями (переходы часто обозначаются именами событий). Некоторые переходы могут быть связаны с условиями, разрешающими эти переходы. *Условие* – это предикат, зависящий от значений некоторых атрибутов объекта. Каждое условие указывается на дуге, переходом по которой управляет это условие. Существенно, что в диаграмме состояний с некоторыми состояниями или событиями связываются определенные операции. Операция, связываемая с событием, обозначает реакцию объекта на это событие и считается, что она выполняется мгновенно (в точке некоторого временнóго интервала). Такая операция называется *действием*. Операция, связываемая с состоянием, выполняется в рамках временнóго интервала, с которым связано это состояние (т. е. имеет продолжительность, ограниченную этим интервалом). Такая операция называется *деятельностью*. Диаграмма состояний определяет управление активизацией указанных операций. Таким образом, диаграмма состояний описывает поведение одного класса объектов. Ниже приведена диаграмма состояний класса на примере состояний телефонной линии.

Динамическая модель в целом объединяет все диаграммы состояний с помощью событий между классами.

Функциональная модель показывает, как вычисляются выходные значения из входных данных без указания порядка, в котором эти значения вычисляются. Она определяет все операции, условия и ограничения, используемые в объектной и динамической моделях (*внешние операции*). Функциональная модель соответствует определению *внешних функций* при реляционном подходе к разработке ПС.

Для определения крупных операций в функциональной модели используются *потоковые диаграммы* (*диаграммы потоков данных*), позволяющие выразить эти операции через более простые операции. Основными понятиями потоковых диаграмм являются *процессы*, *объекты* и *потоки данных*. Потоковая диаграмма – это граф, вершинами которого являются объекты или процессы, а дугами – потоки данных.


Ожидание
Положена трубка
Положена трубка
Положена трубка
Непрерывный гудок
Операция: Гудок
Снята трубка
Пауза
Набрано (n)
Набрано (n)
Пауза. Операция: Писк
Положена трубка
Набор номера
Пауза
Неверный номер
Занято
Сигнал Занято
Операция:
Выдать Занято
Верный (n)
Сообщение
Операция:
Сообщение
Соединение
Операция:
Найти номер
Положена трубка
Линия занята
Быстрый сигнал Занято. Операция:
Короткие гудки
Маршрут найден
Звонок Операция:
Позвонить
Положена трубка
Вызываемый ответил
Операция:
Соединить
Разговор
Положена трубка
Операция: Разорвать
Вызываемый положил трубку. Операция: Разорвать связь
Отсоединение
Положена трубка
Сообщение отправлено

Процессы преобразуют данные, поступающие от одних объектов и направляе мые для хранения в другие объекты. Эти процессы воплощают внутренние операции, через которые выражается операция, определяемая данной потоковой диаграммой. Объекты могут быть пассивными (хранилищами данных) и активными. Пассивные объекты используются только для хранения данных, а активные объекты используются как для хранения, так и для преобразования данных. Потоки данных определяют допустимые направления перемещения данных и типы перемещаемых данных.

Процессы могут выражаться терминальными операциями (определяемыми непосредственно) или с помощью других потоковых диаграмм. Таким образом, потоковые диаграммы являются иерархическими.

Терминальные операции определяются так же, как и при реляционном подходе (см. главы 4 и 5). Впрочем, и диаграммы потоков данных используются при реляционном подходе.

Таким образом, основным содержанием этапа внешнего описания при объектном подходе является объектное моделирование. При этом широко используются формальные языки спецификаций, в том числе и графические. Одним из наиболее употребительных в настоящее время таких языков является язык UML [43].

15.3. Особенности объектного подхода на этапе конструирования программного средства

На этапе конструирования при объектном подходе продолжается процесс объектного моделирования: уточняются (в терминах описания программных систем) модели, построенные на этапе внешнего описания и производится дальнейшая декомпозиция объектов [9, 43, 64].

В процессе разработки объектной архитектуры ПС выделяются все объекты, с информационными моделями которых собирается непосредственно работать пользователь, и завершается их программная спецификация, а так же определяется их пользовательский интерфейс. Такие объекты становятся *пользовательскими*. Классы таких объектов или отдельные активные объекты образуют архитектурные подсистемы. Определяется метод взаимодействия между этими подсистемами.

В случае использования активных объектов основным классом архитектур при объектном подходе является коллектив параллельно действующих программ (см. гл. 6), причем здесь роль программ выполняют как раз эти активные объекты, а способ управления передачей со-

общений зависит от выбранного подкласса таких архитектур. Типичной архитектурой такого класса является архитектура «клиент-сервер» (см. рис. 15.4). В такой системе один из активных объектов, называемый сервером, выполняет определенные программные услуги по запросам других активных объектов, называемых клиентами. Такой запрос передается серверу с помощью сообщения от клиента, результат выполнения сервером запроса передаётся соответствующему клиенту с помощью другого сообщения.

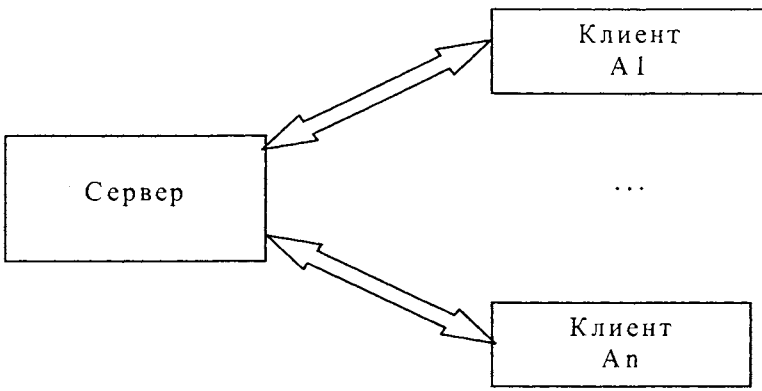


Рис. 15.4. Архитектура «Клиент-Сервер»

Дальнейшая разработка структуры программных подсистем и их кодирование на языках программирования может осуществляться уже в рамках реляционного подхода на соответствующих языках программирования [26] – пользователь внутреннюю организацию этих подсистем уже «не видит». Однако во многих случаях существуют сильные аргументы за то, чтобы продолжить объектную декомпозицию этих подсистем. Объектная структура этих подсистем может быть существенно более понятной разработчику, чем их структура при реляционном подходе. Кроме того, продолжение объектной декомпозиции и использование основных понятий и методов объектного подхода при дальнейшей разработке ПС представляется «естественным», так как весь процесс разработки становится единообразным (концептуально целостным).

При объектном подходе в модульной структуре программы используются преимущественно информационно прочные программные модули. Информационно прочный программный модуль, реализующий

какой-либо класс объектов или логически связанную совокупность таких классов, часто называют компонентом ПС.

15.4. Особенности объектного подхода на этапе кодирования программного средства

На этапе кодирования при объектном подходе используются языки программирования уже другого типа – *объектно-ориентированные* [5, 26]. Считается, что язык программирования поддерживает объектно-ориентированное программирование, если он включает конструкции [5] для:

- инкапсуляции и абстракции данных,
- наследования,
- динамического полиморфизма.

Однако самое существенное различие между реляционными и объектно-ориентированными языками программирования заключается в следующем: в качестве основных средств декомпозиции программ в реляционных языках используются описания функций и процедур (декомпозиция отношений), а в объектно-ориентированных языках – описание классов объектов (декомпозиция объектов). Объекты, возникающие в программах при такой декомпозиции архитектурных подсистем, мы будем называть *объектами процесса выполнения программ*.

Изменяется и технология разработки программного модуля. При реляционном подходе для разработки модулей (в основном, реализующих функцию или процедуру) рекомендовались структурное программирование и пошаговая детализация (см. гл. 8). При объектно-ориентированном подходе для разработки компонента (информационно прочного модуля) более подходит предложенная Дейкстрой [16] нисходящая технология разработки *слоистой* структуры модуля. В этой технологии каждый модуль рассматривается состоящим из некоторой упорядоченной совокупности программных слоев (ср. со слоистой программной системой в гл. 6), причем каждый слой является реализацией некоторой абстрактной машины (в нашем случае это можно рассматривать как реализацию класса или логически связанной совокупности классов). Таким образом, модуль порождается определённой цепочкой иерархически связанных абстрактных машин (так называемыми «бусами» [16]). Разработка модуля начинается с разработки абстрактной машины самого верхнего уровня. Реализация каждой абстрактной машины выражается в терминах ниже стоящей абстрактной машины (ниже

стоящего программного слоя) или в терминах выбранного языка программирования (при реализации абстрактной машины самого нижнего уровня).

Вопросы к главе 15

15.1. В чем заключается сущность объектного подхода к разработке ПС?

15.2. Какие категории объектов можно выделить с точки зрения разработчиков ПС?

15.3. Что такое *объектная модель ПС*?

15.4. Что такое *динамическая модель ПС*?

15.5. Что такое *диаграмма состояний класса*?

15.6. Что такое *функциональная модель ПС*?

15.7. Что такое *компонент ПС*?

Умный в гору не пойдёт,
умный гору обойдёт.
Народная пословица (от И.Б. Задыхайло)

Глава 16
КОМПЬЮТЕРНАЯ ПОДДЕРЖКА РАЗРАБОТКИ И СОПРОВОЖДЕНИЯ ПРОГРАММНЫХ СРЕДСТВ

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

16.1. Инструменты разработки программных средств

При разработке программных средств используется в той или иной мере компьютерная поддержка процессов разработки и сопровождения ПС [65]. Это достигается путем представления хотя бы некоторых программных документов ПС (прежде всего, программ) на компьютерных носителях данных (например, на дискетах) и предоставлению в распоряжение разработчика ПС специальных ПС или включенных в состав компьютера специальных устройств, созданных для какой-либо обработки таких документов. В качестве такого специального ПС можно указать компилятор с какого-либо языка программирования. Компилятор избавляет разработчика ПС от необходимости писать программы на языке компьютера, который для разработчика ПС был бы крайне неудобен, – вместо этого он составляет программы на удобном ему языке программирования, которые соответствующий компилятор автоматически переводит на язык компьютера. В качестве специального устройства, поддерживающего процесс разработки ПС, можно указать, например, эмулятор какого-либо языка. Эмулятор позволяет выполнять (интерпретировать) программы на языке, отличном от языка компьютера, поддерживающего разработку ПС, например, на языке компьютера, для которого эта программа предназначена.

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

Инструменты разработки ПС могут использоваться в течение всего жизненного цикла ПС [22] для работы с разными программными документами. Так, текстовый редактор может использоваться для разработки практически любого программного документа. С точки зрения функций, которые инструменты выполняют при разработке ПС, их можно разбить на следующие четыре группы:

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

Редакторы поддерживают конструирование (формирование) тех или иных программных документов на различных этапах жизненного цикла. Как уже упоминалось, для этого можно использовать один какой-нибудь универсальный текстовый редактор. Однако более сильную поддержку могут обеспечить специализированные редакторы: для каждого вида документов – свой редактор. В частности, на ранних этапах разработки в документах могут широко использоваться графические средства описания (диаграммы, схемы и т. п.). В таких случаях весьма полезными могут быть графические редакторы. На этапе программирования (кодирования) вместо текстового редактора может оказаться более удобным синтаксически управляемый редактор, ориентированный на используемый язык программирования.

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

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

Инструменты, поддерживающие процесс выполнения программ, позволяют выполнять на компьютере описания процессов или отдельных их частей, представленных в виде, отличном от машинного кода, или машинный код с дополнительными возможностями его интерпретации. Примером такого инструмента является эмулятор кода другого

компьютера. К этой группе инструментов следует отнести и различные отладчики. По существу, каждая система программирования содержит программную подсистему, поддерживающую выполнение программ (она выполняет программные фрагменты, наиболее типичные для языка программирования, и обеспечивает стандартную реакцию на возникающие при выполнении программ исключительные ситуации). Такую подсистему, которую мы называем *исполнительной поддержкой программ*, также можно рассматривать как инструмент данной группы.

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

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

Часто разработка ПС производится на том же компьютере, на котором оно будет применяться. Это достаточно удобно по следующим соображением. Во-первых, в этом случае разработчик имеет дело только с компьютерами одного типа. А, во-вторых, в разрабатываемое ПС могут включаться компоненты самой инструментальной среды. Однако это не всегда возможно. Например, компьютер, на котором должно применяться ПС, может быть неудобен для поддержки разработки ПС или его мощность недостаточна для обеспечения функционирования требуемой инструментальной среды. Кроме того, такой компьютер может быть недоступен для разработчиков этого ПС (например, он постоянно занят другой работой, которую нельзя прерывать, или он находится ещё в стадии разработки). В таких случаях применяется так называемый *инструментально-объектный подход* к разработке и использованию ПС [65]. Сущность его заключается в том, что ПС разрабатывается на одном компьютере, называемом *инструментальным*, а применяться будет на другом компьютере, называемом *целевым* (или *объектным*).

Совокупность инструментальных сред разработки и сопровождения ПС можно разбивать на разные классы, которые различаются значением следующих признаков:

- ориентированность на конкретный язык программирования,
- специализированность,
- комплексность,
- ориентированность на конкретную технологию программирования,
- ориентированность на коллективную разработку,
- интегрированность.

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

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

Комплексность инструментальной среды показывает: поддерживает ли она все процессы разработки и сопровождения ПС или нет. В первом случае продукция этих процессов должна быть согласована. Более того, поддержка инструментальной средой фазы сопровождения ПС означает, что она должна поддерживать работу сразу с несколькими вариантами ПС, ориентированными на разные условия применения ПС и на разную связанную с ним аппаратуру, в частности, она должна поддерживать *управление конфигурацией ПС* [13, 65].

Ориентированность на конкретную технологию программирования показывает: ориентирована ли инструментальная среда на фиксированную технологию программирования [22] либо нет. В первом случае структура и содержание её информационной среды, а также набор её инструментов существенно зависит от выбранной технологии (*технологическая определенность*). Во втором случае инструментальная среда поддерживает самые общие операции разработки ПС, не зависящие от выбранной технологии программирования.

Ориентированность на коллективную разработку показывает: поддерживает ли инструментальная среда управление (*management*) работой коллектива или нет. В первом случае она обеспечивает для разных членов этого коллектива разные права доступа к различным фрагментам продукции технологических процессов и поддерживает работу *менеджеров* [65] по управлению коллективом разработчиков. Во втором случае она ориентирована на поддержку работы лишь отдельных пользователей.

Интегрированность инструментальной среды показывает: является ли она интегрированной в каком-либо смысле или нет. Инструментальная среда считается *интегрированной*, если она является интегрированной хотя бы в одном из трех *видов интегрированности*:

- интегрированность по пользовательскому интерфейсу,
- интегрированность по данным,
- интегрированность по действиям (функциям).

Интегрированность инструментальной среды по пользовательскому интерфейсу означает, что все её инструменты объединены единым пользовательским интерфейсом. *Интегрированность инструментальной среды по данным* означает, что её инструменты действуют в соответствии с фиксированной информационной схемой (моделью), определяющей зависимость друг от друга различных используемых в среде фрагментов данных (информационных объектов). Это свойство инст-

рументальной среды позволяет осуществлять контроль полноты и актуальности программных документов разрабатываемого ПС и управлять порядком их разработки. *Интегрированность инструментальной среды по действиям* означает, что, во-первых, в системе имеются общие части всех инструментов и, во-вторых, одни инструменты при выполнении своих функций могут обращаться к другим инструментам.

Инструментальную среду разработки и сопровождения ПС, интегрированную хотя бы по данным или по действиям, будем называть *инструментальной системой разработки и сопровождения ПС*. При этом интегрированность по данным предполагает наличие в системе специализированной базы данных, предназначенной для хранения всех данных, связанных с разработкой ПС в течение всего его жизненного цикла [65]. Такую базу данных будем называть *репозиторием* инструментальной системы разработки и сопровождения ПС.

16.3. Основные классы инструментальных сред разработки и сопровождения программных средств

В настоящее время выделяют [65] три основных класса инструментальных сред разработки и сопровождения ПС (рис. 16.1):

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

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

Рабочее место для компьютерной технологии программирования ориентировано прежде всего на поддержку ранних этапов разработки ПС (системного анализа и спецификаций) и автоматической генерации программ по спецификациям [57, 65]. Оно существенно использует

свойства специализированности, ориентированности на конкретную технологию программирования и, как правило, интегрированности. Более поздние рабочие места компьютерной технологии обладают также свойством комплексности ([57]). Что же касается языковой ориентированности, то вместо языков программирования они ориентированы на те или иные языки спецификаций. Свойством ориентированности на коллективную разработку указанные рабочие места в настоящее время, как правило, не обладают.

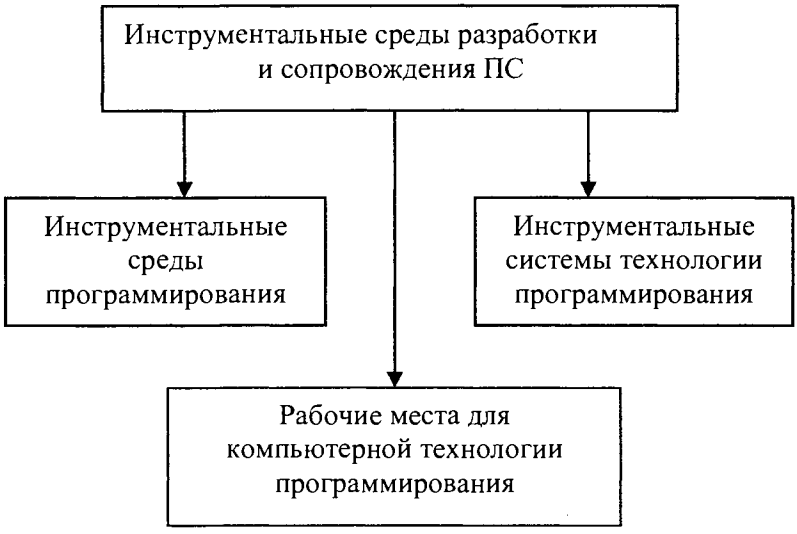


Рис. 16.1. Основные классы инструментальных сред разработки и сопровождения ПС

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

16.4. Инструментальные среды программирования

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

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

- среды общего назначения,
- языково-ориентированные среды.

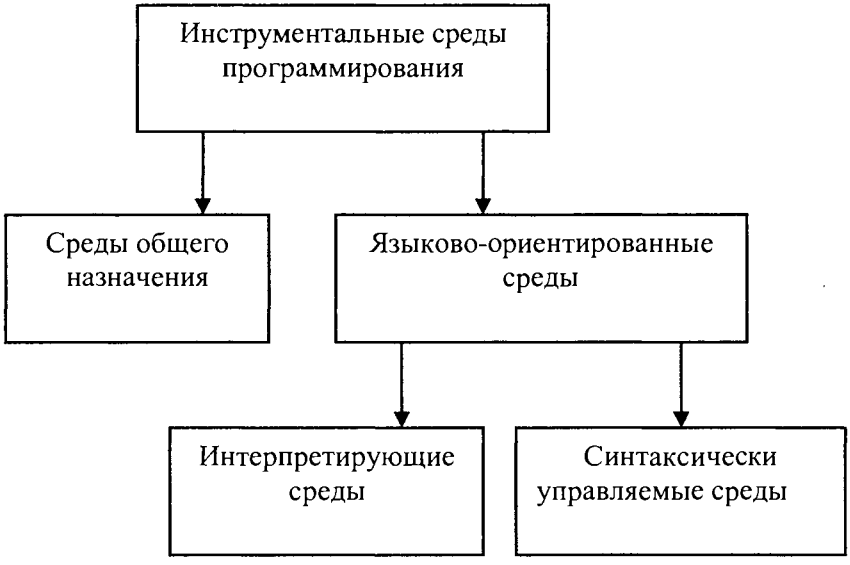


Рис. 16.2. Классификация инструментальных сред программирования

Инструментальная среда программирования *общего назначения* содержит набор программных инструментов, поддерживающих разработку программ на разных языках программирования (например, тексто-

вый редактор, редактор связей или интерпретатор языка целевого компьютера) и обычно представляет собой некоторое расширение возможностей используемой операционной системы. Для программирования в такой среде на каком-либо языке программирования потребуются дополнительные инструменты, ориентированные на этот язык (например, компилятор).

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

- интерпретирующие среды,
- синтаксически управляемые среды.

Интерпретирующая инструментальная среда программирования осуществляет интерпретацию программ на данном языке программирования, т. е. содержит прежде всего интерпретатор языка программирования, на который эта среда ориентирована. Такая среда необходима для языков программирования интерпретирующего типа (таких, как Лисп), но может использоваться и для других языков (например, на инструментальном компьютере). *Синтаксически управляемая* инструментальная среда программирования базируется на знании синтаксиса языка программирования, на который она ориентирована. В такой среде вместо текстового используется синтаксически управляемый редактор, позволяющий пользователю использовать различные шаблоны синтаксических конструкций (в результате этого разрабатываемая программа всегда будет синтаксически правильной). Одновременно с программой такой редактор формирует (в памяти компьютера) её синтаксическое дерево, которое может использоваться другими инструментами.

16.5. Понятие компьютерной технологии программирования и её рабочие места

Имеются некоторые трудности в выработке строгого определения CASE-технологии (*компьютерной технологии программирования*). CASE – это абревиатура от английского Computer-Aided Software Engineering (буквально: Компьютерно-помогаемая инженерия программирования). Но без помощи (поддержки) компьютера ПС уже давно не

разрабатываются (используется хотя бы компилятор). В действительности, в это понятие вкладывается более узкий (специальный) смысл, который постепенно размывается (как это всегда бывает, когда какое-либо понятие не имеет строгого определения). Первоначально под CASE-технологией понималась [57] инженерия ранних этапов разработки ПС (определение требований, разработка внешнего описания и архитектуры ПС) с использованием программной поддержки (программных инструментов). Теперь под CASE-технологией может пониматься [57] и инженерия всего жизненного цикла ПС (включая и его сопровождение), но только в том случае, когда программы частично или полностью генерируются по документам, полученным на указанных ранних этапах разработки. В этом случае CASE-технология стала принципиально отличаться от ручной (традиционной) технологии программирования: изменилось не только содержание технологических процессов, но и сама их совокупность.

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

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

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

На наш взгляд, главное отличие ручной технологии программирования от компьютерной заключается в следующем. Ручная технология ориентирована на разработку документов, одинаково понимаемых разными разработчиками ПС, тогда как компьютерная технология ориентирована на обеспечение семантического понимания (интерпретации) документов программной поддержкой компьютерной технологии. Семантическое понимание документов даёт программной поддержке возможность автоматически генерировать программы. В связи с этим существенной частью компьютерной технологии становится использование формальных языков уже на ранних этапах разработки ПС: как для

спецификации программ, так и для спецификации других документов. Например, широко используются формальные графические языки спецификаций. Именно это позволяет рационально изменить и саму совокупность технологических процессов разработки и сопровождения ПС.

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

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

С учётом сказанного *жизненный цикл ПС для компьютерной технологии* можно представить [57] схемой, приведенной на рис. 16.3.

Прототипирование ПС является необязательным этапом жизненного цикла ПС при компьютерной технологии, что на рис. 16.3 показано пунктирной стрелкой. Однако использование этого этапа во многих случаях и соответствующая компьютерная поддержка этого этапа является характерной для компьютерной технологии.

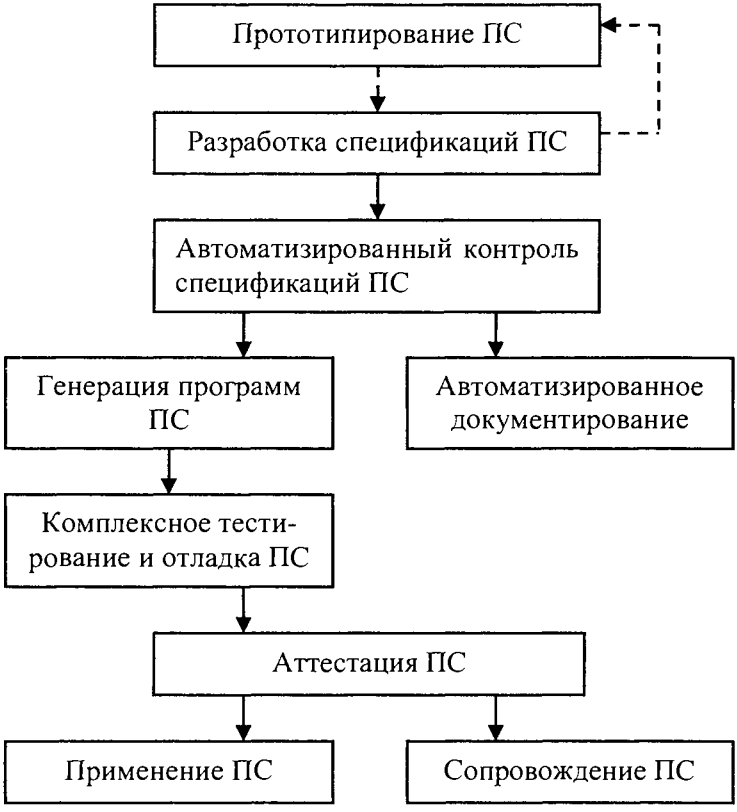


Рис. 16.3. Жизненный цикл программного средства для компьютерной технологии программирования

В некоторых случаях прототипирование делается после (или в процессе) разработки спецификаций ПС, например, в случае прототипиро-

вания пользовательского интерфейса. Это показано на рис. 16.3 пунктирной возвратной стрелкой. Хотя возврат к предыдущим этапам мы допускаем на любом этапе, но здесь это показано явно, так как прототипирование является особым подходом к разработке ПС (см. гл. 3). Прототипирование пользовательского интерфейса позволяет заменить косвенное описание взаимодействия между пользователем и ПС при ручной технологии (при разработке внешнего описания ПС) прямым выбором пользователем способа и стиля этого взаимодействия с фиксацией всех необходимых деталей. По существу, на этом этапе производится точное описание пользовательского интерфейса, понятное программной поддержке компьютерной технологии, причем с ответственным участием пользователя. Всё это базируется на использовании для ПС настраиваемой оболочки с обширной библиотекой заготовок различных фрагментов и деталей экрана. Такое прототипирование, по-видимому, является лучшим способом преодоления барьера между пользователем и разработчиком.

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

Автоматизированный контроль спецификаций ПС использует то обстоятельство, что значительная часть спецификаций представляется на формальных языках. Это позволяет автоматически осуществлять различные виды контроля: синтаксический и частичный семантический контроль спецификаций, контроль полноты и состоятельности схем и диаграмм (в частности, все их элементы должны быть идентифициро-

ваны и отражены в словаре именованных сущностей), сквозной контроль сбалансированности уровней спецификаций и другие виды контроля в зависимости от возможностей языков спецификаций.

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

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

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

Аттестация ПС имеет прежнее содержание.

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

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

- конструкторы пользовательских интерфейсов;
- инструмент работы со словарем именованных сущностей;
- графические и тестовые редакторы спецификаций;
- анализаторы спецификаций;
- генератор программ;
- документаторы.

16.6. Инструментальные системы технологии программирования

Инструментальная система технологии программирования – это интегрированная совокупность программных и аппаратных инструментов, поддерживающая все процессы разработки и сопровождения больших ПС в течение всего его жизненного цикла в рамках определённой технологии программирования. Выше уже отмечалось (см. п. 14.3), что она помимо интегрированности обладает ещё свойствами комплексности и ориентированности на коллективную разработку. Это означает, что она базируется на согласованности продукции технологических процессов. Тем самым, инструментальная система в состоянии обеспечить, по крайней мере, контроль полноты (комплектности) создаваемой документации (включая набор программ) и согласованности её изменения (версионности). Поддержка инструментальной системой фазы сопровождения ПС означает, что она должна обеспечивать *управление конфигурацией ПС* [13, 65]. Кроме того, инструментальная система поддерживает управление работой коллектива разработчиков и для разных членов этого коллектива обеспечивает разные права доступа к различным фрагментам продукции технологических процессов, а также поддерживает работу *менеджеров* [65] по управлению коллективом разработчиков. Инструментальные системы технологии программирования представляют собой достаточно большие и дорогие ПС, чтобы как-то была оправдана их инструментальная перегруженность. Поэтому набор включаемых в них инструментов тщательно отбирается с учётом потребностей предметной области, используемых языков и выбранной технологии программирования.

С учётом обсуждённых свойств инструментальных систем технологии программирования можно выделить три их основные компоненты:

- репозиторий,
- инструментарий,
- интерфейсы.

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

части всех инструментов (*ядра*) и структурных (иногда иерархически связанных) классов инструментов.

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

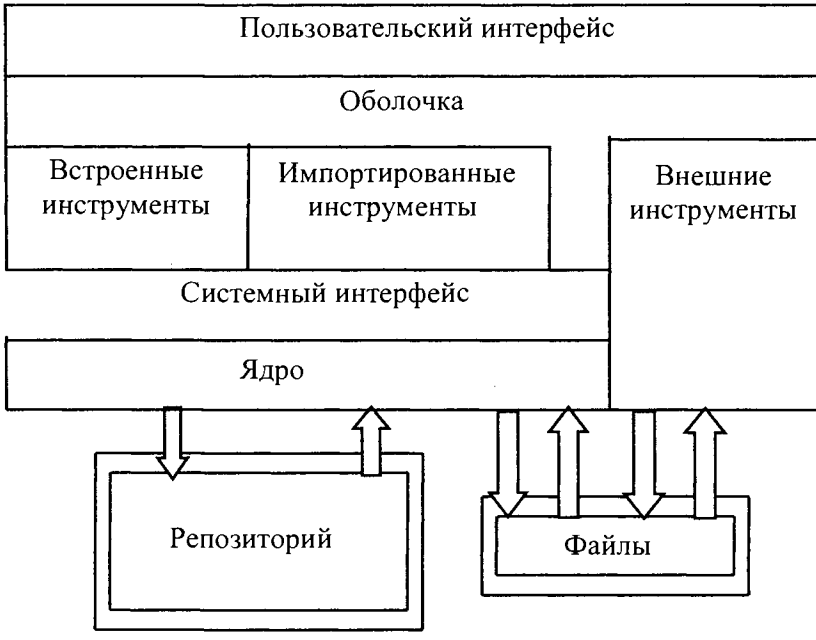


Рис. 16.4. Общая архитектура инструментальных систем технологии программирования.

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

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

Инструментальная система поддержки программного проекта – это открытая система, способная поддерживать разработку ПС на разных языках программирования после соответствующего её расширения про-

граммными инструментами, ориентированными на выбранный язык. Набор инструментов такой системы поддерживает управление разработкой ПС, а также содержит независимые от языка программирования инструменты, поддерживающие разработку ПС (текстовые и графические редакторы, генераторы отчётов и т. п.). Кроме того, он содержит инструменты расширения системы. Ядро такой системы обеспечивает, в частности, доступ к репозиторию.

Языково-зависимая инструментальная система технологии программирования – это такая инструментальная система технологии программирования, которая поддерживает разработку и сопровождение ПС на каком-либо одном языке программирования и существенно использует в организации своей работы специфику этого языка. Эта специфика может сказываться и на возможностях ядра (в том числе и на структуре репозитория), и на требованиях к оболочке и инструментам.

Примером такой системы является среда поддержки программирования на языке Ада (APSE [62]). Прежде всего эта система базируется на понятиях языка Ада, включая организацию репозитория и стиля пользовательского интерфейса. При построении системы предполагается, что все её инструменты, кроме некоторых машинно-зависимых, написаны на языке Ада. В системе имеется компилятор с языка Ада в качестве встроенного инструмента, а ядро системы содержит поддержку выполнения программ, оттранслированных с языка Ада. Таким образом, любой импортируемый инструмент, написанный на Аде, включается в систему путем его трансляции с помощью встроенного компилятора. Требуется лишь, чтобы он был согласован с системным интерфейсом. Предполагается, что спецификация системного интерфейса представлена спецификацией на языке Ада одного или нескольких пакетов. На каждом компьютере ядро системы построено таким образом, чтобы обеспечить корректное функционирование этого интерфейса во всех случаях, когда к нему производится обращение в соответствии со спецификацией этого интерфейса на Аде из программ, оттранслированных с языка Ада. Тем самым, обеспечивается высокая степень мобильности этой системы: при переносе её на другой компьютер требуется лишь реализация соответствующего ядра, компилятора с языка Ада на этот компьютер, оболочки, реализующей пользовательский интерфейс, и необходимые машинно-зависимые инструменты.

Вопросы к главе 16

16.1. Что такое *программный инструмент разработки ПС*?

16.2. Что такое *аппаратный инструмент разработки ПС*?

16.3. Что такое *инструментальная среда разработки и сопровождения ПС*?

16.4. Что такое *инструментально-объектный подход к ПС*?

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

16.6. Что такое *интегрированность инструментальной среды разработки и сопровождения ПС*?

16.7. Какие виды интегрированности инструментальной среды разработки и сопровождения ПС Вы знаете?

16.8. Что такое *репозиторий инструментальной среды разработки и сопровождения ПС*?

16.9. Что такое *инструментальная среда программирования*?

16.10. Что такое *языково-ориентированная инструментальная среда программирования*?

16.11. Что такое *компьютерная технология программирования*?

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

16.13. Что такое *рабочее место для компьютерной технологии программирования*?

16.14. Что такое *инструментальная система технологии программирования*?

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

16.16. Что такое *ядро инструментальной системы технологии программирования*?

16.17. Что такое *встроенный инструмент инструментальной системы технологии программирования*?

16.18. Что такое *импортируемый инструмент инструментальной системы технологии программирования*?

16.19. Что такое *оболочка инструментальной системы технологии программирования*?

ТОЛКОВЫЙ СЛОВАРЬ ОСНОВНЫХ ПОНЯТИЙ

А

Автономная отладка программного средства – последовательное раздельное *тестирование* различных частей *программ*, входящих в *программное средство*, с поиском и исправлением в них фиксируемых при тестировании *ошибок*.

Автономность программного средства – примитив *качества программного средства,* который представляет собой свойство, характеризующее способность *программного средства* выполнять предписанные функции без помощи или поддержки других компонент программного обеспечения.

Администратор программного средства – лицо, которое управляет использованием *программного средства ординарными пользователями* и осуществляет *сопровождение программного средства*, не связанное с изменением его программ.

Активный программный объект – *программный объект*, состояние которого имеет программные части, способные находиться в процессе выполнения.

Аппаратная платформа программного средства – компьютерно-аппаратная среда, от которой может зависеть *программное средство.*

Аппаратный инструмент разработки программного средства – устройство компьютера, специально предназначенное для поддержки разработки и функционирования *программного средства.*

Архитектура программного средства – строение *программного средства* как оно видно (или должно быть видно) из-вне его.

Архитектурная функция программного средства – функция, поддерживающая взаимодействие между программными подсистемами, выделенными в *архитектуре программного средства*, и выполняемая программной компонентой *программного средства*, видимой из-вне его.

Аттестация программного средства – авторитетное подтверждение *качества программного средства*.

Б

Бригада ведущего программиста – особо организованная *бригада разработчиков программного средства*, в которой всю разработку порученной бригаде части *программного средства* выполняет один программист высокой квалификации (*ведущий программист*), а остальные члены такой бригады создают условия для успешной работы этого программиста и выполняют отдельные его поручения.

Бригада разработчиков программного средства – часть коллектива разработчиков *программного средства*, выполняющих какую-либо одну относительно независимую работу по разработке *программного средства*.

В

Веха развития программного проекта – конечная точка некоторого этапа или процесса *разработки программного средства*, с которой связывается выдача промежуточного продукта, представляющего собой некоторый чётко определённый документ.

Внешнее описание программного средства – описание поведения *программного средства* с точки зрения внешнего по отношению к нему наблюдателя с фиксацией требований относительно его *качества*.

Временна́я эффективность программного средства – примитив *качества программного средства*, который представляет собой меру, характеризующую способность *программного средства* выполнять возложенные на него функции в течение определённого отрезка времени.

Д

Данные – представление фактов и идей в формализованном виде, пригодном для передачи и переработки в некоем *процессе*.

Документация по сопровождению программного средства – совокупность документов, которые объясняют разработчикам и сопроводителям, как они должны действовать, чтобы изменять данное *программное средство*.

Документация программного средства – совокупность документов, входящих в состав *программного средства*, которые описывают *программное средство* как с точки зрения применения его программ пользователями, так и с точки зрения его построения разработчиками и сопроводителями.

Документация управления разработкой программного средства – совокупность документов, которые протоколируют процессы разработки и сопровождения *программного средства*, определяют условия этой разработки, исполнителей и порядок выполнения отдельных её работ (заданий), регулируют отношения между этими исполнителями внутри коллектива разработчиков, а также между коллективом разработчиков и *менеджерами программного средства.*

Дублер ведущего программиста – член *бригады ведущего программиста*, который должен быть в курсе всего, что делает *ведущий программист*, выступать в роли его оппонента при обсуждении его идей и предложений, а также быть способным выполнить любую работу ведущего программиста.

Ж

Жизненный цикл программного средства – весь период разработки и эксплуатации (использования) *программного средства*, начиная от момента возникновения замысла о нём и кончая прекращением всех видов его использования.

З

Завершаемость программы – свойство *программы*, означающее отсутствие в ней зацикливания при любых исходных *данных*.

Завершённость программного средства – примитив *качества программного средства,* который представляет собой свойство, характеризующее степень обладания *программным средством* всеми необходимыми частями и чертами, требующимися для выполнения своих явных и неявных функций.

Защищённость программного средства – примитив *качества программного средства,* который представляет собой свойство, характеризующее способность *программного средства* противостоять преднамеренным или нечаянным деструктивным (разрушающим) действиям пользователя.

И

Изучаемость программного средства – характеристики *программного средства*, которые позволяют минимизировать усилия по изучению и пониманию программ и документации *программного средства*; является подкритерием *сопровождаемости программного средства*.

Имитатор программного модуля – отладочный программный модуль, замещающий какой-либо модуль программы в процессе её нисходящей *автономной отладки*.

Инсталлятор программного средства – компонента (подсистема) *программного средства*, осуществляющая автоматическую настройку *программного средства* на условия его применения по информации, задаваемой пользователем.

Инструмент разработки программного средства – аппаратный или *программный инструмент разработки программного средства*.

Инструментальная система поддержки программного проекта – открытая *инструментальная система технологии программирования*, способная поддерживать разработку *программных средств* на разных языках программирования после соответствующего её расширения *программными инструментами*, ориентированными на выбранный язык.

Инструментальная система разработки и сопровождения программного средства – инструментальная среда разработки и сопровождения программного средства, обладающая свойством *интегрированности по данным* или *интегрированности по действиям*.

Инструментальная система технологии программирования – инструментальная система разработки и сопровождения программного средства, поддерживающая все процессы разработки и сопровождения большого *программного средства* в течение всего его *жизненного цикла* в рамках определённой *технологии программирования*.

Инструментальная среда программирования – инструментальная среда разработки и сопровождения программного средства, поддерживающая процессы программирования (кодирования *программ*), *тестирования* и *отладки программного средства*.

Инструментальная среда разработки и сопровождения программного средства – логически связанная совокупность *инструментов разработки программного средства.*

Интегрированность инструментальной среды – свойство *инструментальной среды разработки и сопровождения программного средства*, заключающееся в обладании некоторой не пустой комбинацией из трех свойств: *интегрированность инструментальной среды по пользовательскому интерфейсу*, *интегрированность инструментальной среды по данным*, *интегрированность инструментальной среды по действиям.*

Интегрированность инструментальной среды по данным – свойство *инструментальной среды разработки и сопровождения программного средства*, означающее, что её *инструменты* действуют в соответствии с фиксированной информационной схемой (моделью), определяющей зависимость друг от друга различных используемых в среде фрагментов *данных* (информационных объектов).

Интегрированность инструментальной среды по действиям – свойство *инструментальной среды разработки и сопровождения программного средства*, означающее, что в среде имеются общие части всех её *программных инструментов* и при выполнении своих функций одни *инструменты* могут обращаться к другим *инструментам.*

Интегрированность инструментальной среды по пользовательскому интерфейсу – свойство *инструментальной среды разработки и сопровождения программного средства*, означающее, что все её *инструменты* объединены единым *пользовательским интерфейсом.*

Интерпретирующая инструментальная среда программирования – *языково-ориентированная инструментальная среда программирования*, осуществляющая интерпретацию *программ* на данном языке программирования, т.е. содержит прежде всего интерпретатор языка программирования, на который эта среда ориентирована.

Информативность программного средства – примитив *качества программного средства,* который представляет собой свойство, характеризующее наличие в составе *программного средства информации*, необходимой и достаточной для понимания назначе-

ния *программного средства*, принятых предположений, существующих ограничений, входных *данных* и результатов работы отдельных компонент, а также текущего состояния *программ* в процессе их функционирования.

Информатика – неустановившееся понятие: в узком смысле – наука об *обработке информации*, преимущественно с помощью компьютеров; в широком смысле – *информационные технологии*; в рамках настоящей книги используется как перевод англоязычного термина *Computer Science*.

Информационная среда – совокупность *носителей данных*, используемых при какой-либо *обработке данных*.

Информационные технологии – неустановившееся понятие: обозначает широкую сферу человеческой деятельности (в науке, технике и производстве), охватывающую исследования теоретических и методологических основ, разработку и создание технологий информационной индустрии, связанных со сбором, производством, обработкой, передачей, распространением, хранением, представлением, использованием, защитой различных видов *информации*; в рамках настоящей книги используется как перевод англоязычного термина *Computing*.

Информационно прочный программный модуль – *программный модуль*, выполняющий (реализующий) несколько операций (функций) над одной и той же структурой *данных*, которая считается неизвестной вне этого модуля.

Информация – смысл, который придаётся *данным* при их представлении.

Исполнительная поддержка программ – операционная среда выполнения *программ* (программная подсистема, поддерживающая выполнение программ).

К

Качество программного средства – совокупность черт и характеристик *программного средства*, которые влияют на его способность удовлетворять заданным потребностям пользователей.

Кодирование программного средства – этап *разработки программного средства*, на котором осуществляются процессы программирования (кодирования *программ*), *тестирования* и *отладки программного средства*.

Коммуникабельность программного средства – примитив *качества программного средства*, который представляет собой свойство, характеризующее степень, в которой *программное средство* облегчает задание или описание входных *данных*, и способность выдавать полезные сведения в достаточно простой форме и с простым для понимания содержанием.

Комплексная отладка программного средства – *тестирование программного средства* в целом с поиском и исправлением фиксируемых при тестировании *ошибок* во всех документах (включая тексты *программ*), относящихся к *программному средству* в целом.

Компьютерная технология программирования – *технология программирования*, в которой используются *программные инструменты* для разработки формализованных спецификаций *программ* и других документов (включая и графические спецификации) с последующей автоматической генерацией значительной части *программ* и документов по этим спецификациям.

Конструирование программного средства – этап *разработки программного средства*, на котором осуществляются процессы разработки *архитектуры программного средства* и структуры его *программ*, включая спецификацию *программных модулей*, входящих в эти *программы*.

Л

Лёгкость изменения программного средства – примитив *качества программного средства*, который представляет собой меру, характеризующую *программное средство* с точки зрения простоты внесения необходимых изменений и доработок на всех этапах и стадиях *жизненного цикла программного средства*.

Лёгкость применения программного средства – критерий *качества программного средства*, который представляет собой характеристики *программного средства*, позволяющие минимизировать усилия пользователя по подготовке исходных данных, применению *программного средства* и оценке полученных результатов, а также вызывать положительные эмоции определённого или подразумеваемого пользователя.

М

Менеджер программного средства – лицо, управляющее *разработкой программного средства*.

Мобильность программного средства – критерий *качества программного средства*, который представляет собой способность *программного средства* быть перенесённым из одной среды в другую, в частности, с одного компьютера на другой.

Модифицируемость – характеристики *программного средства*, которые позволяют автоматически настраивать его на условия применения *программного средства* или упрощают внесение в него вручную необходимых изменений и доработок; является подкритерием *сопровождаемости программного средства*.

Модульное программирование – метод программирования, в котором *программа* разделяется на автономно разрабатываемые части (*программные модули*).

Модульность программного средства – примитив *качества программного средства*, который представляет собой свойство, характеризующее *программное средство* с точки зрения организации его *программ* из таких дискретных компонент, что изменение одной из них оказывает минимальное воздействие на другие компоненты.

Н

Надёжность программного средства – критерий *качества программного средства*, который представляет собой способность *программного средства* безотказно выполнять определённые функции при заданных условиях в течение заданного периода времени с достаточно большой вероятностью.

Независимость программного средства от устройств – примитив *качества программного средства*, который представляет собой свойство, характеризующее способность *программного средства* работать на разнообразном аппаратном обеспечении (различных типах, марках, моделях компьютеров).

Носитель данных – устройство или физическая среда, в которой представляются и хранятся некоторые *данные*.

О

Обработка данных – выполнение систематической последовательности действий с *данными*.

Операционная платформа программного средства – программная среда (в частности, операционная система), над которой строится *программное средство*.

Операционная система – совокупность программных компонент (*программное средство*), которые совместно управляют ресурсами компьютера и процессами обработки данных, использующих эти ресурсы.

Ординарный пользователь программного средства – лицо, использующее (*применяющее*) ПС для решения своих задач.

Отказ программного средства – проявление *ошибки в программном средстве*.

Отладка программного средства – деятельность, направленная на обнаружение и исправление *ошибок в программном средстве* с использованием процессов выполнения его *программ*.

Отладочный программный модуль – *программный модуль*, включаемый в программу в процессе её *автономной отладки* для организации *тестирования* какого-либо другого *программного модуля*.

Ошибка в программном средстве – состояние *программного средства*, в котором оно не выполняет того, что разумно ожидать от него пользователю.

П

Пассивный программный объект – *программный объект*, состояние которого не имеет программных частей, способных находиться в процессе выполнения.

П-документированность программного средства – примитив *качества программного средства*, который представляет собой свойство, характеризующее наличие, полноту, понятность, доступность и наглядность учебной, инструктивной и справочной документации, необходимой для *применения программного средства*.

Пользователь программного средства – *ординарный пользователь программного средства* или *администратор программного средства*.

Пользовательская документация программного средства – совокупность документов, объясняющая пользователям, как они должны действовать, чтобы применить данное *программное средство*.

Пользовательский интерфейс программного средства – языковое средство взаимодействия пользователя с *программным средством*.

Пользовательский объект в программном средстве – некоторый фрагмент внешней *информационной среды программного средства*, который пригоден для хранения *данных* определённого типа и с которым связан некоторый набор операций, применимых к нему *пользователем*.

Понятность программного средства – примитив *качества программного средства,* который представляет собой свойство, характеризующее степень, в которой *программное средство* 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.

Применение программного средства – использование *программного средства* для решения практических задач на компьютере путём выполнения его *программ*.

Программа – формализованное описание *процесса обработки данных*.

Программное средство – *программа* или логически связанная совокупность *программ* на *носителях данных*, снабжённая программной документацией.

Программный инструмент разработки программного средства – *программное средство*, предназначенное для поддержки разработки или сопровождения других *программных средств*.

Программный модуль – любой фрагмент описания *процесса обработки данных*, оформляемый как самостоятельный программный продукт, пригодный для использования в описаниях разных *процессов обработки данных*.

Программный объект – некоторый фрагмент *информационной среды*, который пригоден для хранения *данных* определённого типа и с которым связан некоторый набор применимых к нему операций.

Программный проект – вся совокупность работ, связанная с разработкой *программного средства*.

Процесс обработки данных – последовательность сменяющих друг друга *состояний* некоторой *информационной среды*.

Псевдокод – частично формализованный язык, предназначенный для описания процессов.

Р

Рабочее место для компьютерной технологии программирования – *инструментальная среда разработки и сопровождения программ-*

ного средства, поддерживающая этапы *жизненного цикла компьютерной технологии разработки программного средства*.

Развитие программного проекта – ход выполнения работ, связанных с *разработкой программного средства*.

Разработка программного средства – стадия *жизненного цикла программного средства*, включающая все процессы, связанные с созданием *программного средства*.

Расширяемость программного средства – примитив *качества программного средства*, который представляет собой свойство, характеризующее способность *программного средства* к наращиванию объёма памяти для хранения *данных* или расширению функциональных возможностей отдельных компонент.

Рутинный программный модуль – программный модуль, результат (эффект) обращения к которому зависит только от значений его параметров (и не зависит от предыстории обращений к нему).

С

С-документированность программного средства – примитив *качества программного средства*, который представляет собой свойство, характеризующее с точки зрения наличия документации, отражающей требования к *программному средству* и результаты различных этапов разработки данного *программного средства*, включающие возможности, ограничения и другие черты *программного средства*, а также их обоснование.

Синтаксически управляемая инструментальная среда программирования – языково-ориентированная инструментальная среда программирования, базирующаяся на знании синтаксиса языка программирования, на который она ориентирована.

Система – совокупность взаимодействующих (находящихся в отношениях) друг с другом элементов.

Сопровождаемость программного средства – критерий *качества программного средства*, который представляет собой характеристики *программного средства*, позволяющие минимизировать усилия по внесению изменений для устранения в нём ошибок и по его модификации в соответствии с изменяющимися потребностями пользователей.

Сопровождение программного средства – процесс сбора информации о качестве *программного средства* в эксплуатации, устранения об-

наруженных в нём *ошибок*, его доработки и модификации, а также извещения пользователей о внесённых в него изменениях.

Состояние информационной среды – совокупность *данных*, содержащихся в какой-либо момент в *информационной среде*.

Спецификация – в литературе по программированию используется как описание (часто формализованное) каких-либо аспектов *программных средств*, например: *спецификация программы, спецификация семантики функций, спецификация качества программного средства.*

Структурированность программного средства – примитив *качества программного средства,* который представляет собой свойство, характеризующее *программы программного средства* с точки зрения организации взаимосвязанных их частей в единое целое определённым образом.

Т

Тестирование программного средства – процесс выполнения его *программ* на *тестовых наборах данных.*

Тестовый набор данных (тест) программы – набор *данных*, для которого заранее известен результат применения или известны правила поведения данной *программы.*

Технология программирования – совокупность производственных процессов, приводящая к созданию требуемого *программного средства*, а также описание этой совокупности процессов.

Точность программного средства – примитив *качества программного средства,* который представляет собой меру, характеризующую приемлемость величины погрешности в выдаваемых *программным средством* результатах с точки зрения предполагаемого их использования.

Транслятор – программное средство, осуществляющее перевод *программ* с какого-либо *языка программирования* на другой *язык программирования* или на язык соответствующего компьютера.

У

Удобочитаемость программного средства – примитив *качества программного средства,* который представляет собой свойство, характеризующее лёгкость восприятия текста *программ программного средства.*

Управление конфигурацией программного средства – деятельность, обеспечивающая согласованность компонент (на всех уровнях их представления) *программного средства* в процессе создания его новых версий.

Управление программным проектом – управление разработкой программного средства.

Управление разработкой программного средства – деятельность, направленная на обеспечение необходимых условий для работы коллектива разработчиков *программного средства*, на планирование и контроль деятельности этого коллектива с целью обеспечения требуемого *качества программного средства*, выполнения сроков и бюджета разработки *программного средства*.

Устойчивость программного средства – примитив *качества программного средства,* который представляет собой свойство, характеризующее способность *программного средства* продолжать корректное функционирование, несмотря на задание неправильных (ошибочных) входных *данных*.

Ф

Функционально прочный программный модуль – программный модуль, выполняющий (реализующий) одну какую-либо определённую функцию.

Функциональность программного средства – критерий *качества программного средства,* который представляет собой способность *программного средства* выполнять набор функций, удовлетворяющих заданным или подразумеваемым потребностям пользователей.

Э

Эффективность программного средства по ресурсам – примитив *качества программного средства,* который представляет собой меру, характеризующую способность *программного средства* выполнять возложенные на него функции при определённых ограничениях на используемые ресурсы (память).

Эффективность программного средства по устройствам – примитив *качества программного средства,* который представляет собой меру, характеризующую экономичность использования устройств компьютера для решения поставленной задачи.

Эффективность программного средства – критерий *качества программного средства,* который представляет собой отношение уровня услуг, предоставляемых *программным средством* пользователю при заданных условиях, к объёму используемых ресурсов.

Я

Язык программирования – формализованный язык, предназначенный для описания *процессов обработки данных.*

Языково-зависимая инструментальная система технологии программирования – инструментальная система технологии программирования, поддерживающая разработку *программных средств* на каком-либо одном языке программирования и существенно использующая в организации своей работы специфику этого языка.

Языково-ориентированная инструментальная среда программирования – инструментальная среда программирования, поддерживающая разработку *программного средства* на каком-либо одном языке программирования, причём знания об этом языке существенно использовались при построении этой среды.

СЛОВАРЬ ИСПОЛЬЗУЕМЫХ АНГЛОЯЗЫЧНЫХ ТЕРМИНОВ

A

Accountability – информативность (программного средства).
Accuracy – точность (программного средства).
Augmentability – расширяемость(программного средства).

B

Backup programmer – дублер ведущего программиста (в *бригаде ведущего программиста*).

C

Certfication – аттестация (программного средства).
Chief programmer team – бригада ведущего программиста.
Communicativeness – коммуникабельность (программного средства).
Completeness – завершённость (программного средства).
Computer Science – информатика (в узком смысле); компьютерная наука.
Computing – информационные технологии; информатика (в широком смысле).
Conciseness – компактность (*программного средства*).
Consistency – согласованность (*программного средства*).
Criteria of software quality – критерии *качества программного средства* (программного обеспечения).

D

Data – данные.
Data processing – обработка данных.
Defect – дефект, неисправность.
Defensiveness – защищённость (программного средства).
Description of the system architecture – описание архитектуры системы, описание *архитектуры программного средства.*
Device efficiency – эффективность по устройствам (программного средства) по устройствам.
Device independence – независимость (программного средства) от устройств.

Documentation – документированность, *С-документированность* (*программного средства*).

E

Easy of use – лёгкость применения (*программного средства*).
Efficiency – эффективность (*программного средства*).
End-user – ординарный пользователь (*программного средства*).
Environments – среда.

F

Functionality – функциональность (*программного средства*).

H

Hardware – аппаратура, аппаратное оборудование.

I

Implementation – реализация.
Information – информация.

L

Legibility – четкость (*программного средства*).
Librarian – администратор базы *данных* разработки (в *бригаде ведущего программиста*).

M

Maintainability – сопровождаемость (*программного средства*).
Modifiability – лёгкость изменения (*программного средства*).
Modularity – модульность (*программного средства*).

O

Operating system(OS) – операционная система(ОС).

P

Plan – план, схема.
Portability – мобильность (*программного средства*).
Programming language – язык программирования.

R

Readability – удобочитаемость (программного средства).

Reliability – надёжность (программного средства).

Resource efficiency – эффективность (программного средства) по ресурсам.

Reusable – повторно используемая (компонента *программного средства*).

Robustness – устойчивость (программного средства).

Run-time environment – среда выполнения (*программы*).

Run-time support – Исполнительная поддержка (программ), поддержка периода выполнения (*программ*).

S

Self-containedness – автономность (программного средства).

Self-descriptiveness – самодокументированность (*программного средства*).

Software – программное средство, программное обеспечение.

Software certfication – аттестация (сертификация) *программного средства.*

Software coding – кодирование программного средства; программирование (в узком смысле).

Software configuration management – управление конфигурацией программного средства.

Software debugging – отладка программного средства.

Software design – конструирование программного средства.

Software design description – описание строения (конструкции) *программного средства.*

Software development – разработка программного средства.

Software development environment – инструментальная среда разработки программного средства.

Software documentation – С-документированность программного средства.

Software engineering – программная инженерия.

Software engineering environment – инструментальная система технологии программирования.

Software error – ошибка в программном средстве.

Software life cycle – жизненный цикл программного средства.

Software maintenance – сопровождение программного средства.

Software management – управление *программным средством.*

Software manager – менеджер программного средства.

Software operation – применение программного средства.

Software process – процесс, затрагивающий *разработку* и *сопровождение программного средства.*

Software product documentation – документация программного средства.

Software process documentation – документация управления разработкой программного средства.

Software project – программный проект.

Software project management – управлением программным проектом.

Software project progress – развитие программного проекта.

Software project progress milestone – веха развития программного проекта.

Software quality – качество программного средства.

Software quality assurance – обеспечение *качества программного средства.*

Software quality manager – менеджер по *качеству программного средства.*

Software requireents definition – определение требований к *программному средству.*

Software requirements document – внешнее описание программного средства, документ требований к *программному средству.*

Software requirements specification – спецификация требований к *программному средству.*

Software specification – спецификация *программного средства.*

Software testing – тестирование программного средства.

Software user – пользователь программного средства.

Specification – спецификация.

Structuredness – структурированность (*программного средства*).

System administrator – администратор программного средства, системный администратор.

System documentation – документация по сопровождению (*программного средства*), системная документация.

System maintenance guide – руководство по сопровождению программного средства.

T

Test – тест, тестировать.
Testing – тестирование, испытание.
Time efficiency – временна́я эффективность (*программного средства*).
Tool – инструмент.
Toolset – инструментарий.
Top-down testing – нисходящее *тестирование.*
Translator – транслятор.

U

Understandability – понятность (*программного средства*).
Usability – полезность (*программного средства*).
User documentation – пользовательская документация (*программного средства*).
User interface – пользовательский интерфейс (*программного средства*).
U. documentation – П-документированность (*программного средства*).

V

Validation documents – документы установления достоверности (*программного средства*).
Verification – проверка, верификация.

W

Workbench – рабочее место.

СПИСОК ЛИТЕРАТУРЫ

1. Абрамов С.А. Элементы программирования. – М.: Наука, 1982.
2. Агафонов В.Н. Спецификация программ: понятийные средства и их организация. – Новосибирск: Наука (Сибирское отделение), 1987.
3. Бахвалов Н.С., Жидков Н.П., Кобельков Г.М. Численные методы. – М.: Наука, 1987.
4. Безбородов Ю.М. Индивидуальная отладка программ. – М.: Наука, 1982.
5. Бен-Ари М. Языки программирования. Практический сравнительный анализ. Пер. с англ. – М.: Мир, 2000.
6. Березин И.С., Жидков Н.П. Методы вычислений, т.т. 1 и 2. – М.: Физматгиз, 1959.
7. Боэм Б., Браун Дж., Каспар Х. и др. Характеристики качества программного обеспечения. Пер. с англ. – М.: Мир, 1981.
8. Брукс Ф.П., мл. Как проектируются и создаются программные комплексы. – М.: Наука, 1979.
9. Буч Г. Объектно-ориентированное проектирование с примерами применения. Пер. с англ. – М.: Конкорд, 1992.
10. Ван Тассел Д. Стиль, разработка, эффективность, отладка и испытание программ. Пер. с англ. – М.: Мир, 1985.
11. Вельбицкий И.В., Ходаковский В.Н., Шолмов Л.И.. Технологический комплекс производства программ на машинах ЕС ЭВМ и БЭСМ-6. – М.: Статистика, 1980.
12. Вирт Н. Систематическое программирование. Пер. с англ. – М.: Мир, 1977.
13. Горбунов-Посадов М.М. Конфигурации программ. Рецепты безболезненных изменений. – М.: «Малип», 1994.
14. Гоулд И.Г., Тутилл Дж.С. Терминологическая работа IFIP (Международная федерация по обработке информации) и ICC (Международный вычислительный центр) // Журн. вычисл. матем. и матем. физ., 1965, #2. – с. 377-386.
15. Даль В. Толковый словарь русского языка. – М.: Советская энциклопедия, 1975
16. Дейкстра Э. Заметки по структурному программированию / У. Дал, Э. Дейкстра, К. Хоор. Структурное программирование. Пер. с англ. – М.: Мир, 1975. – с. 7-97.

17. Джехани Н. Язык Ада. Пер. с англ. – М.: Мир, 1988.
18. Дзержинский Ф.Я., Тер-Сааков А.И.. Технология программирования – структурный подход. – М.: Цниатоминформ, 1978.
19. Дзержинский Ф.Я., Калиниченко И.М. Дисциплина программирования Д: концепция и опыт реализации методических средств программной инженерии. – М.: ЦНИИ информации и технико-экономических исследований по атомной науке и технике, 1988.
20. Жоголев Е.А. Система программирования с использованием библиотеки подпрограмм / Система автоматизация программирования. – М.: Физматгиз, 1961. – с. 15-52.
21. Жоголев Е.А. Технологические основы модульного программирования // Программирование, 1980, #2. – с. 44-49.
22. Жоголев Е.А. Введение в технологию программирования (конспект лекций). – М.: "ДИАЛОГ-МГУ", 1994.
23. Жоголев Е.А. Лекции по технологии программирования: учебное пособие. – М.: Изд. отдел факультета ВМиК МГУ, 2001.
24. Зелковец М., Шоу А., Гэннон Дж. Принципы разработки программного обеспечения. Пер. с англ. – М.: Мир, 1982.
25. Зиглер К. Методы проектирования программных систем. Пер. с англ. – М.: Мир, 1985.
26. Кауфман В.Ш. Языки программирования. Концепции и принципы. – М.: Радио и связь, 1993.
27. Крачтен Ф. Введение в RATIONAL UNIFIED PROCESS. Пер. с англ. – М.: Изд. Дом «Вильямс», 2002.
28. Кристиан М. Введение в операционную систему UNIX. Пер. с англ. – М.: Финансы и статистика, 1985.
29. Лебедев А.Н. Защита банковской информации и современная криптография // Вопросы защиты информации, 2(29), 1995.
30. Липаев В.В. Качество программного обеспечения. – М.: Финансы и статистика, 1983.
31. Липаев В.В. Тестирование программ. – М.: Радио и связь, 1986.
32. Липаев В.В., Позин Б.А., Штрик А.А. Технология сборочного программирования. – М.: Радио и связь, 1992.
33. Липаев В.В. Управление разработкой программных средств. Методы, стандарты, технология. – М.: Финансы и статистика, 1993.
34. Липаев В.В., Филиппов Е.Н. Мобильность программ и данных в открытых информационных системах. – М.: Научная книга, 1997.

35. Майерс Г. Надежность программного обеспечения. Пер. с англ. – М.: Мир, 1980.
36. Ожегов С.И. Словарь русского языка. – М.: Советская энциклопедия, 1975.
37. Пайл Я. АДА – язык встроенных систем. – М.: Финансы и статистика, 1984.
38. Пойа Д. Как решать задачу. Пер. с англ. – М.: Наука, 1961.
39. Рекомендации по преподаванию информатики в университетах. – СПб: Изд-во Санкт-Петербургского ГУ, 2002.
40. Скотт Д. Теория решеток, типы данных и семантика / Данные в языках программирования. Пер. с англ. – М.: Мир, 1982.
41. Требования и спецификации в разработке программ. Сборник статей. Пер. с англ. под ред. В.Н. Агафонова. – М.: Мир, 1984.
42. Турский В. Методология программирования. Пер. с англ. – М.: Мир, 1981.
43. Фаулер М., Скотт К. UML в кратком изложении. Пер. с англ. – М.: Мир, 1999.
44. Фокс Дж. Программное обеспечение и его разработка. Пер. с англ. – М.: Мир, 1985.
45. Фуксман А.Л. Технологические аспекты создания программных систем. – М.: Статистика, 1979.
46. Фути К., Судзуки Н. Языки программирования и схемотехника СБИС. Пер. с англ. – М.: Мир, 1988.
47. Хоор К. О структурной организации данных / У. Дал, Э. Дейкстра, К. Хоор. Структурное программирование. Пер. с англ. – М.: Мир, 1975. – с.98-197.
48. Хьюз Дж., Мичтом Дж. Структурный подход к программированию. Пер. с англ. – М.: Мир, 1980.
49. Шнейдерман Б. Психология программирования. Пер. с англ. – М.: Радио и связь, 1984.
50. ANSI/IEEE Std 829-1983, IEEE Standard for Software Test Documentation.
51. ANSI/IEEE Std 830-1984, IEEE Guide for Software Requirements Specification.
52. ANSI/IEEE Std 983-1986, IEEE Guide for Software Quality Assurance Planning.

53. ANSI/IEEE Std 1012-1986, IEEE Standard for Software Verification and Validation Plans.
54. ANSI/IEEE Std 1008-1987, IEEE Standard for Software Unit Testing.
55. ANSI/IEEE Std 1016-1987, IEEE Recommended Practice for Software Design Description.
56. ANSI/IEEE Std 1063-1988, IEEE Standard for Software User Documentation.
57. CASE: Компьютерное проектирование программного обеспечения. – Издательство Московского университета, 1994.
58. Criteria for Evaluation of Software. – ISO TC97/SC7 #367 (Supersedes Document #327).
59. Criteria for Evaluation of Software. ISO TC97/SC7 #383.
60. Dijkstra E.W. The Structure of the THE-Multiprogramming // Communications of the ACM. –1968, 11(5). – p. 341-346.
61. Holt R.C. Structure of computer programs: A Survey // Proceedings of the IEEE, 1975, 63(6). – p. 879-893.
62. Requirements for Ada Programming Support Environments. – USA: DoD, Stoneman, 1980.
63. Revised version of DP9126 – Criteria of the evaluation of software quality characteristics. ISO TC97/SC7 #610. Part 6.
64. Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorenzen W. Objekt-Oriented Modeling and Design. – Prentice Hall. 1991.
65. Sommerville I. Software Engineering. – Addison-Wesley Publishing Company, 1992.

СОДЕРЖАНИЕ

Нучное издание

Автор:
доктор физ.-мат. наук, профессор
ЖОГОЛЕВ Евгений Андреевич

"Научный мир"
Тел./факс (007)(095)291-28-47. E-mail: naumir@ben.irex.ru
Internet: http://195.178.196.201/ N_M/n_m.htm
ЛР № 03221 от 10.11.2000

Подписано к печати 20.03.2004. Формат 60x90/16. Гарнитура Таймс.
Печать офсетная. Усл. печ. л. 13,5. Тираж 500 экз. Заказ [illegible]
Издание отпечатано в типографии ООО "Галлея-Принт".
Москва, 5-я Кабельная, 26