ИЗДАТЕЛЬСТВО
МИР

SOFTWARE MAINTENANCE GUIDEBOOK

ROBERT L. CLASS
RONALD A. NOISEUX

Prentice-Hall, Inc.
Englewood Cliffs, New Jersey
1981

Р. Гласс, Р. Нуазо

СОПРОВОЖДЕНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Перевод с английского
канд. техн. наук Г. Л. ВЫШКОВСКОГО

под редакцией
д-ра техн. наук, проф. Ю. А. ЧЕРНЫШЕВА

Москва «Мир» 1983

ББК 32.973
Г 52
УДК 681.3

Гласс Р., Нуазо Р.

Г 52 Сопровождение программного обеспечения: Пер. с англ. — М.: Мир, 1983. — 156 с., ил.

В книге американских авторов рассмотрены вопросы методологии проектирования и применения программного обеспечения с целью повышения эффективности его использования для решения практических задач. Особое внимание уделяется роли программиста, осуществляющего сопровождение программного обеспечения, и всевозможным приемам в его работе, позволяющим сводить количество ошибок в программе до минимума.

Для специалистов в области сопровождения программного обеспечения ЭВМ, а также аспирантов и студентов соответствующих специальностей.

Г $\frac{2405000000-222}{041(01)-83}$ 159-84, ч. 1 ББК 32.973

Редакция литературы по новой технике

© Prentice-Hall International, Englewood Cliffs, 1981

© Перевод на русский язык, «Мир», 1983

ПРЕДИСЛОВИЕ РЕДАКТОРА ПЕРЕВОДА

Бурное развитие вычислительных машин и их использование в различных областях науки и техники привело к созданию большого количества программ. Над их составлением и отладкой работают десятки миллионов высококвалифицированных специалистов. Однако процесс внесения изменений, а также дальнейшее улучшение программного обеспечения и его поддержание в рабочем состоянии, как правило, не выполняется составителями самих программ. Поэтому в последнее время за рубежом появились специалисты, занимающиеся сопровождением программного обеспечения. Это позволило намного увеличить эффективность использования вычислительных машин, работающих как в интерактивном, так и в пакетном режимах. При этом повысилась надежность программного обеспечения, а самое главное стало возможным широкое использование ранее составленных программ при решении новых задач.

Авторы книги уделяют большое внимание организации сопровождения программного обеспечения и требованиям к специалистам, работающим в этой области. Указываются основные характеристики общего и специального программного обеспечения, гарантирующие высокое качество сопровождения. Выдержки из дневника программиста дают представление о рабочем процессе сопровождения.

Приведенные примеры документации можно использовать на практике при программном сопровождении. Обширная библиография и комментарии к ней знакомят читателя с книгами и журнальными публикациями, посвященными сопровождению программного обеспечения.

Книга написана в доступной форме и будет полезна широкому кругу специалистов, занимающихся программированием.

Ю. А. Чернышев

ПРЕДИСЛОВИЕ

> Сопровождение является загадкой программного обеспечения. Огромные средства вкладываются в него. Немногие исследователи и руководители уделяют ему должное внимание. И фактически пока само понятие «сопровождение» определено недостаточно точно.

В этой книге мы попытаемся приподнять завесу таинственности, окружающую процесс сопровождения программного обеспечения.

Книга начинается с определения понятия «сопровождение» и его места в «жизненном цикле программного обеспечения».

Далее излагается необычная точка зрения, утверждающая, что важную роль в процессе сопровождения программного обеспечения играет личность специалиста по сопровождению. Это — невоспетый герой, благодаря скромному труду которого не затихает мерное гудение вычислительных машин в мире, не слишком щедром на похвалы и награды.

Основываясь на главенствующей роли личности, авторы затем переходят к основной теме книги, раскрывающей как известные, так и новые методы работы этих «невоспетых героев». В книге описываются средства и методы, которые должны быть им знакомы, начиная от самых обычных (преобразователей кода) до почти фантастических (суперкомпиляторов). Основной принцип, с помощью которого разработчик программного обеспечения может облегчить задачу специалиста по сопровождению (в дальнейшем мы будем называть его сопровождающим программистом), заключается в том, чтобы делать свою работу правильно с первого раза. В книге приводится множество примеров, в большинстве случаев написанных на недавно созданном в Министерстве обороны США алгоритмическом языке Ада (Ada).

Далее речь в книге идет о руководстве сопровождением программного обеспечения, которое неразрывно связано с процессом сопровождения. Обсуждаются вопросы планирования, организации и управления. Дается принципиально новый подход к проблеме документации, который открывает небывалые возможности, но требует от руководителей серьезного анализа.

И наконец, для более глубокого понимания материала и придания ему практического смысла в книге приводится дневник сопровождающего программиста, представляющий собой подлинную документальную запись всех событий, происходящих день за днем в его трудовой жизни.

В конце каждой главы дан список литературы. В результате перечисляется почти все, что на сегодняшний день написано по

сопровождению. Написано мало, и это лишний раз указывает на недостаточный интерес исследователей к сопровождению. Однако это положение начинает постепенно улучшаться.

Данная книга предназначается для руководителей или сопровождающих программистов, а также для студентов, которые уже имеют представление о программном обеспечении, но нуждаются в расширении своих знаний в области сопровождения. Книга будет полезной специалистам-консультантам при усовершенствовании работы программного обеспечения в соответствующих организациях, а также может быть использована как часть универсального курса по методам программного обеспечения либо в целях повышения квалификации уже работающих, опытных программистов.

Авторы благодарят всех тех, кто внес вклад в создание книги, будь то их идея, статья, книга или даже просто фраза. Мир программного обеспечения станет лучше, потому что все мы прилагаем к этому усилия.

Авторы посвящают эту книгу всем сопровождающим программистам мира — тем невоспетым героям, благодаря которым не умолкает гул вычислительных машин.

Глава 1

ВВЕДЕНИЕ

К несчастью, природа сбоев аппаратных средств ЭВМ и программного обеспечения противоположна по крайней мере в одном: аппаратные средства ЭВМ отказывают из-за отсутствия должного обслуживания, в то время как программное обеспечение приходит в негодность из-за попыток осуществить его[1].

Предлагаем вашему вниманию очень простой тест. Ниже приведен ряд утверждений, связанных с вопросами сопровождения программного обеспечения. В случае вашего согласия (или несогласия) с этими утверждениями поставьте против каждого пункта «да» (или «нет»).

1. Сопровождение программного обеспечения заключается в исправлении ошибок.

2. Поскольку сопровождение следует за созданием программного обеспечения, нет нужды его планировать.

3. Стоимость сопровождения программного обеспечения составляет лишь небольшую часть средств, затраченных на программное обеспечение.

4. Сопровождение программного обеспечения — довольно неинтересный предмет.

Ну вот. А теперь давайте займемся самооценкой. Правильный ответ на каждый из пунктов — «нет». Начиная со 100, вычитайте по 25 очков за каждый неверный ответ. Если вы набрали 100 очков, вам, возможно, незачем читать далее нашу книгу, за исключением, пожалуй, гл. 3, где обсуждаются некоторые технические приемы обучения сопровождению. Если вы набрали 75 очков, то вы прекрасный специалист, и вас можно поздравить.

Но, если вы набрали меньше 50 очков, не огорчайтесь. Вы принадлежите к большинству. Существует множество заблуждений, связанных с пониманием сопровождения программного обеспечения, и вы разделили некоторые из них.

Почему же на все вышеприведенные пункты теста следовало ответить словом «нет»? Разъяснению причин и посвящена наша

---

[1] Fisher, Standish, Initial Thoughts on the Pebbleman Process, Institute for Defense Analyses, January 3, 1979.

книга. Чтобы удовлетворить читательское любопытство, скажем вкратце следующее:

1. Сопровождение программного обеспечения предполагает скорее модификацию программ, нежели просто исправление ошибок в программе.

2. При существующем отношении к сопровождению программного обеспечения правильные решения приходят с опозданием, и в результате переработок программное обеспечение начинает носить непоследовательный и внутренне противоречивый характер.

3. Исследования показывают, что сопровождение программного обеспечения поглощает более половины средств, выделяемых на модификацию всей системы.

4. Значимость затрачиваемых средств делает сопровождение программного обеспечения интересным.

Этим тестом нам хотелось заинтересовать читателя. Как и все мышление, находящееся во власти стереотипов, сопровождение программного обеспечения объято настоящим туманом заблуждений. Порой даже самые опытные специалисты в области вычислительной техники — и теоретики, и практики — бывают далеки от проблем, связанных с сопровождением программного обеспечения. Наша книга ставит своей целью преодолеть недопонимание и стереотипы, а также предложить вниманию заинтересовавшихся читателей практическое руководство по сопровождению программного обеспечения.

Надо отметить, что пункт 4 теста «Сопровождение программного обеспечения — довольно неинтересный предмет» является на самом деле коварным утверждением. Большинство честных людей, связанных с сопровождением программного обеспечения, откровенно поставили «да». И авторы смогут считать свою книгу удавшейся, если, прочитав ее, эти люди переменят свою точку зрения.

А теперь еще один тест. Условия те же: отвечайте «да» или «нет», но будьте внимательны!

1. Важность вопросов сопровождения программного обеспечения стала причиной серьезного их изучения.

2. Методы и приемы работы специалиста по сопровождению программного обеспечения хорошо известны.

3. Сопровождением программного обеспечения обычно занимаются лучшие специалисты.

4. Руководство уделяет большое внимание сопровождению программного обеспечения.

Ну уж на этот раз вас не проведешь! В пункте 3 содержалась подсказка. И конечно, опять правильный ответ на все пункты «нет». Короче говоря, до недавнего времени сопровождению программного обеспечения не уделялось достаточного внимания.

Исследователи не хотят связываться с ним: методика сопровождения программного обеспечения вообще редко их интересует;

связанные с ней задачи считаются неинтересными, и их избегают все, кому это удается. Как правило, им занимается младший технический персонал. И руководство часто вполне удовлетворено существующим положением дел. (Однако многие организации вынуждены выплачивать премии, чтобы стимулировать работу по сопровождению!)

Если первый тест ставил своей задачей разрушить ошибочные представления о сопровождении программного обеспечения, то второй призван подчеркнуть вредные последствия таких представлений. Приходится признать, что мы слишком долго пренебрегали вопросами сопровождения программного обеспечения. Это произошло из-за недооценки его значения. В результате мы имеем целую и очень важную область, которая недостаточно хорошо изучена.

При сложившихся обстоятельствах нелегко написать эффективную и полезную книгу. Во-первых, трудно убедить читателя обратить внимание на книгу с таким компрометирующим названием. Как уже говорилось, к сопровождению программного обеспечения сложилось предвзятое отношение. К тому же нет ни одной монографии по обсуждаемому вопросу, на которую авторы могли бы опираться и ссылаться. Остается одно: быть пионерами этого дела и написать достаточно интересную книгу, способную привлечь внимание читателей, а не отбить у них всякую охоту заниматься проблемами сопровождения программного обеспечения, предоставив читателям ее оценку в качестве очередного теста.

Авторы книги сами долгое время занимались сопровождением программного обеспечения. Их также интересовали теоретические и практические вопросы вычислительной техники. Эта книга — попытка поделиться накопленным опытом, создать определенную базу для дальнейших исследований, дать специалистам и их руководителям основную информацию, необходимую для эффективной работы, и, наконец, создать учебное пособие для изучающих программирование, дающее полное представление о методах, перспективах и всем спектре их будущей деятельности, включая пресловутое «сопровождение программного обеспечения».

1.1. МЕСТО СОПРОВОЖДЕНИЯ В ЖИЗНЕННОМ ЦИКЛЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Дискуссия по вопросу о так называемом жизненном цикле на одной из конференций по вычислительной технике стала поводом для постоянных шуток. Каждый из выступающих непременно считал своим долгом представить в графическом виде свою собственную модель этого понятия. К концу дня один из выступающих остроумно назвал свою версию «обязательной блок-схемой жизненного цикла программного обеспечения».

В области программного обеспечения, как, впрочем, и в других областях техники, нередки модные увлечения. Если какая-либо

проблема заинтересует одного исследователя, то через некоторое время уже 90% исследователей увлекаются ей. В воздухе начинает витать слово, новое или с интересным подтекстом, и все оживляются. Затем появляется новое понятие, которое обещает значительное снижение затрат и улучшение показателей работы программного обеспечения. И число сторонников этой новой идеи быстро растет.

К счастью, большинство таких увлечений не лишено смысла. Возможно, их ценность не всегда прямо пропорциональна числу приверженцев, но все же смысл в них есть. «Жизненный цикл» сопровождения программного обеспечения это и есть именно такое не лишенное смысла новое увлечение.

Основной смысл введения этого понятия состоит в том, чтобы уничтожить распространенное ошибочное мнение, что процесс создания программного обеспечения — это только процесс написания программ. Уже некоторые исследования конца 60-х—начала 70-х годов [1] поставили под сомнение такое представление и обратили внимание на некоторые моменты процесса создания программного обеспечения. Дальнейшие наблюдения показали, что анализ требований, предъявляемых к программному обеспечению и к его разработке, занимал значительно больше времени, чем написание программ. Кроме того, проверка работоспособности уже созданного программного обеспечения также отнимала много времени. В этой связи появление блок-схемы жизненного цикла программного обеспечения было обусловлено стремлением наглядно изобразить новейшие результаты исследований в этом направлении.

Многие первые блок-схемы жизненного цикла не включали в себя понятие «сопровождения программного обеспечения». И если раньше основное внимание ошибочно уделялось составлению программ, то теперь его ошибочно сосредоточили на процессе доработки программ. Только в середине 70-х годов в схемы жизненного цикла стали включать сопровождение, что произвело переворот в существовавших представлениях!

Настало время дать определение. Жизненный цикл программного обеспечения — это весь процесс его создания и применения от начала до конца. Этот процесс состоит из нескольких стадий: определения технических требований, проектирования, программирования (реализации), отладки, контроля и, наконец, сопровождения. Причины, послужившие поводом для появления названия «жизненный цикл», ясны: это тип обобщенного определения, которое учитывает все стадии существования программного обеспечения от начала до конца.

Причины появления и признания этого понятия не в том, что оно что-то отвергает, а что-то проясняет. Важным фактором является стоимость программного обеспечения, и становится очевидным, что объем капиталовложений в одну из стадий жизненного цикла

программного обеспечения неизбежно влияет на средства, затрачиваемые на других его стадиях. Поэтому при разработке программного обеспечения необходимо анализировать весь цикл его создания в целом. Например, увеличение объема капиталовложений на этапе проектирования, безусловно, приведет к снижению стоимости составления программ, их отладки и контроля. Чем лучше составлены программы, тем проще и дешевле их отладка. А тщательный анализ технических требований позволит снизить затраты на всех последующих этапах.

Эта экономическая сторона дела стала очевидной в тот момент, когда важность учета сопровождения перестала ставиться под сомнение. Более тщательная разработка каждой предыдущей стадии неизменно приводила к уменьшению стоимости стадии сопровождения. Если же принять во внимание величину затрат на сопровождение, то это соображение представляется очень важным. Поэтому целый раздел нашей книги (разд. 3.2.2.1) будет посвящен проблеме снижения затрат на сопровождение путем более детальной разработки предшествующих стадий жизненного цикла. Книга содержит множество практических рекомендаций по этому поводу.

Важно отметить, что до того, как был рассмотрен жизненный цикл, экономический аспект проблемы не был очевидным. Разработчик программного обеспечения мог, например, менее тщательно разработать стадию сопровождения, сократив тем самым затраты, и, казалось бы, удешевить разработку. Заказчику же в конечном счете это обойдется дороже, так как придется нести дополнительные расходы на стадии сопровождения. Если же эту проблему рассматривать с позиций «жизненного цикла», то разумно было бы указать исполнителю на его недоработки на начальной стадии проектирования, пусть даже заплатив за это немного больше, а в итоге на этом значительно сократить общие расходы. Доводы экономического характера были основным фактором в признании концепции жизненного цикла.

Введем несколько определений. Для этого рассмотрим основные стадии жизненного цикла программного обеспечения.

Назовем первую из них стадией определения *требований/спецификаций*. Она также может быть названа стадией системного анализа. На этом этапе осмысливается цель и ставится задача. Ее решение может быть найдено на этом же этапе, но это решение необходимо проверить при работе всей системы. Затем его следует всесторонне проанализировать и представить в виде спецификаций для программного обеспечения. Такие спецификации могут считаться первым результатом начальной стадии. Пожалуй, самую большую опасность на этом этапе представляет желание решать только ту часть задачи, которая хорошо определена. Поддавшись такому соблазну, мы можем в итоге прийти к несоответствию результатов разработки целям применения системы. Это в свою очередь вызовет

необходимость формирования новых требований к отдельным частям системы и их значительной доработке, что может привести к «гибели» всей системы. Самой сложной и дорогостоящей работой в создании программного обеспечения является его доработка. Множество программ было выброшено только потому, что они не поддавались доработке. Таким образом, хорошо продуманные требования и спецификации жизненно необходимы для деятельности сопровождающего программиста.

Второй стадией является *проектирование* системы. На этом этапе, согласно техническим требованиям и техническим условиям спецификаций, происходит выбор возможных путей и методов реализации. Здесь решаются вопросы выбора: типа ЭВМ, необходимого объема памяти и состава комплекса технических средств, языка программирования, типа модулей, перечня выполняемых функций, структуры данных и т. п. Все эти компоненты включаются в техническое задание на программное обеспечение. Первичным результатом стадии проектирования является проектное решение. Оно может быть представлено в виде описания, в форме схем или таблиц, на алгоритмическом языке или в любом другом виде. Основной сложностью на данном этапе является решение вопроса: когда остановиться? Обычное заблуждение заключается в стремлении остановиться слишком быстро, что приводит к большому количеству ошибок в проектировании. Не следует впадать и в другую крайность — вникать в бесчисленное множество различных вариантов. Это приведет к напрасной трате времени и средств и будет дублировать этап реализации. Оптимальный вариант следует находить для каждого отдельного случая с учетом конкретных условий работы.

Третья стадия — *программирование*. Проектное решение, полученное на предыдущей стадии, здесь реализуется в виде программ. Постепенно программное обеспечение системы приобретает форму и уже выступает как сущность, способная принимать решения. На этой же стадии решаются все вопросы, связанные с особенностями типа ЭВМ. Разрабатываются отдельные блоки и подключаются к создаваемой системе. Впечатление такое (а возможно, так и есть на самом деле), что создается инструмент, подобный скрипке Страдивари, способный исполнять прекрасную музыку. Небрежность здесь недопустима. Программа состоит из массы сложных элементов, многие из которых взаимосвязаны, а иные заключают в себе головоломку. Поэтому существует риск, что небрежность превратит прекрасную скрипку Страдивари в детскую свистульку.

Четвертая стадия — *отладка* программного обеспечения. На этом этапе начинаем играть на скрипке Страдивари с целью установить, отвечает ли она предъявляемым требованиям. При этом системный программист играет роль Шерлока Холмса, подозреваю-

щего вычислительную машину и созданные программы во всех смертных грехах. В результате этой игры он обнаруживает целый ряд недостатков системы, в число которых входят ошибки, допущенные в технических требованиях, на этапе проектирования, при программировании. Отладка состоит в наведении лоска на вновь созданное программное обеспечение, которое предстоит использовать в ближайшем будущем. Отладка — это тяжелый труд, требующий огромного терпения и заключающийся в апробировании всех требований, всех структурных элементов системы и такого количества всевозможных комбинаций, какое только позволяют здравый смысл и бюджет. Здесь существует соблазн прекратить работу, объявить программу работающей и сбыть ее пользователю. Последствия такого шага могут быть самыми ужасными. Пользователь или сопровождающий программист, получивший такую программу, может потерять веру в нее, что окончательно погубит все дело.

Пятая стадия — *сопровождение*, бедное, всеми нелюбимое сопровождение. Сопровождение — это процесс исправления ошибок, координации всех элементов системы в соответствии с потребностями пользователя, внесения нужных ему изменений. Как мы уже говорили, программисты стремятся избежать этой работы, так как она считается недостаточно творческой. Парадокс в том, что сопровождающий программист, который является самым важным с точки зрения пользователя звеном жизненного цикла сопровождения программного обеспечения, подчас оказывается наименее квалифицированным. В результате прекрасно настроенная скрипка Страдивари превращается в высококачественное полено для растопки очага в зимнее время. Все достижения предыдущих стадий сводятся на нет неумелым сопровождающим программистом.

На рис. 1.1-1 — 1.1-4 рассмотрен жизненный цикл программного

Рис. 1.1-1. Жизненный цикл программного обеспечения: затраты по стадиям.

Рис. 1.1-2. Жизненный цикл программного обеспечения: количество ошибок по стадиям.

обеспечения в различных аспектах, а именно затраты, ошибки и надежность. Из рисунков видно, что сопровождение есть главная стадия, которую следует учитывать при оценке затрат, стадия проектирования системы — основная по количеству совершающихся на ней ошибок, а на стадии приемочных испытаний совместно с сопровождением происходит обнаружение наибольшего числа ошибок. Эти цифры особенно интересны тем, что они подтверждают чисто интуитивные предположения, высказанные ранее.

Было проведено несколько исследований величины затрат на различных стадиях жизненного цикла программного обеспечения [1—3], и, хотя результаты исследований несколько различны, все они отмечают преобладающее значение стадии сопровождения.

Рис. 1.1-3. Жизненный цикл программного обеспечения: обнаружение ошибок по стадиям.

Были также проведены исследования причин возникновения ошибок по стадиям жизненного цикла программного обеспечения [4, 5]. Все они начинаются после стадии определения требований/спецификаций, так как их авторы единодушно рассматривают эту стадию как эталон, по которому судят об ошибках. Все они также учитывают только ошибки, обнаруженные после запуска системы, или ее контрольных испытаний, или ее передачи пользователю. Как видно из рис. 1.1-2, преобладают ошибки, допущенные на стадии проектирования.

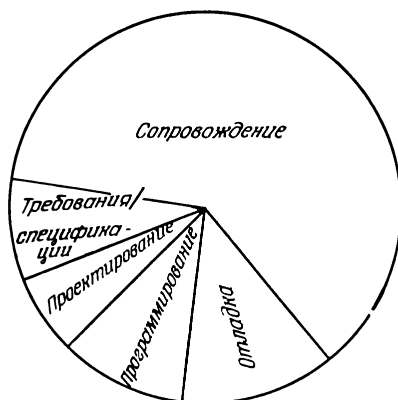

Рис. 1.1-4. Жизненный цикл программного обеспечения: затраты на устранение ошибок по стадиям.

Исследования [14] показывают, что ошибки чаще всего обнаруживаются лишь на последних стадиях жизненного цикла — в основном во время приемочных испытаний или после них (рис. 1.1-3). Затраты на устранение ошибок на каждой последующей стадии значительно возрастают по сравнению с предыдущей [6]. На стадии отладки они уже огромны (рис. 1.1-4).

1.2. МИНИЦИКЛ ПРОЦЕССА СОПРОВОЖДЕНИЯ

До сих пор мы сознательно избегали точного определения термина «сопровождение программного обеспечения». Такой подход может показаться читателю странным, ведь это — основная тема книги. Однако у авторов были для этого основания.

Мы уже могли убедиться, что сопровождение программного обеспечения не совсем то, чем его привыкли считать. Существует мнение, что специалист по сопровождению программного обеспечения, подобно врачу или автомеханику, занимается только устранением ошибок в программе одному ему доступными способами. Действительно, это входит в его задачу. Однако, как показали

последние исследования, круг решаемых им задач значительно шире.

Фактически он выполняет несколько функций: исправляет ошибки в программах, вносит в них изменения по просьбе пользователя, приспосабливает их к работе в новых условиях, одним словом, улучшает программное обеспечение.

Теперь настало время дать определение. Сопровождение программного обеспечения в понимании авторов книги — это такая работа, которая включает внесение необходимых изменений в программы и поддержание уже готового программного обеспечения в рабочем состоянии. При этом сопровождающий программист не может подтянуть разболтавшуюся деталь программного обеспечения с помощью гаечного ключа или заменить ее новой, так как на вид новое программное обеспечение ничем не отличается от использовавшегося ранее. Поэтому сопровождающего программиста нельзя сравнивать с автомехаником. Его труд носит разносторонний характер.

В процессе своей работы он должен: изучить и понять новые требования к системе, вызывающие необходимость ее доработки; выбрать оптимальный способ видоизменения программ, который приведет к удовлетворению новых требований; внести необходимые изменения в программы; проверить работу модифицированного программного обеспечения и взять на себя ответственность за его работу.

Если вы не находите сказанное в этом разделе новым, это вполне естественно. Здесь мы просто обобщили положения разд. 1.1. Таким образом, сопровождающий программист, занимающийся сопровождением программного обеспечения, причастен ко всем стадиям его жизненного цикла.

В некотором смысле слово «обеспечение» обманчиво. Сопровождающий программист играет ведущую творческую роль в жизненном цикле программного обеспечения. Его деятельность носит более разносторонний характер, чем работа системного аналитика, который не программирует, или программиста, который не занимается тестированием.

1.3. УСОВЕРШЕНСТВОВАНИЕ, АДАПТАЦИЯ И КОРРЕКЦИЯ В ПРОЦЕССЕ СОПРОВОЖДЕНИЯ

Существует особый аспект понятия «сопровождение». Мы уже говорили о том, что сопровождение программного обеспечения заключается не только в исправлении ошибок. Изменения в программы чаще приходится вносить из-за изменившихся технических требований к системе, а не по вине программиста, допустившего ошибку. Статистические исследования и практический опыт убедительно это доказывают.

Важные исследования, проведенные в работе [7], показали, что деятельность сопровождающего программиста фактически распадается на три основных этапа. Эти этапы и распределение времени между ними рассматриваются ниже.

1.3.1. УСОВЕРШЕНСТВОВАНИЕ В ПРОЦЕССЕ СОПРОВОЖДЕНИЯ

Этап усовершенствования в процессе сопровождения заключается в усовершенствовании работы программного обеспечения в соответствии с интересами заказчика и пользователя. На этом этапе работа сопровождающего программиста состоит не в устранении ошибок, а во внесении изменений в программное обеспечение. Согласно данным работы [7], этот вид деятельности отнимает около 60% всего времени.

1.3.2. АДАПТАЦИЯ В ПРОЦЕССЕ СОПРОВОЖДЕНИЯ

Программное обеспечение не может существовать само по себе. Оно является составной частью всей системы в целом и обеспечивает четкое взаимодействие отдельных частей системы во время ее работы. Адаптация в процессе сопровождения предполагает приспособление программного обеспечения к изменившимся условиям работы системы. Если происходит изменение команд вычислительной машины или необходимо внести существенные изменения в базу данных (например, увеличить мантиссу числа с пяти до девяти значащих цифр), то программное обеспечение должно быть адаптировано к этим изменениям. По данным исследований [7], 18% времени сопровождающего программиста затрачивается на адаптацию программного обеспечения в процессе сопровождения.

1.3.3. КОРРЕКЦИЯ В ПРОЦЕССЕ СОПРОВОЖДЕНИЯ

Наконец мы добрались до темы, которая, несомненно, заинтересует читателей! Чистое исправление ошибок программного обеспечения, входящее в состав этапа коррекции в процессе сопровождения, занимает лишь 17% времени сопровождающего программиста. Таким образом, мы существенно изменили представление читателей о сопровождении программного обеспечения. (Те из вас, кто до сих пор внимательно читали нашу книгу, возможно, обратили внимание на тот факт, что суммарные затраты по времени меньше 100%. В работе [7] это время оставляется на «разное».)

1.4. ОПРЕДЕЛЕНИЯ

Будем надеяться, что нам удалось сформировать у читателей общее представление о сопровождении программного обеспечения. Нам хотелось также показать, что сопровождение представляет собой особый род деятельности, включающий в себя все стадии

жизненного цикла программного обеспечения и выполняемый скорее ради усовершенствования программного обеспечения, нежели исправления ошибок в нем.

Прежде чем перейти к следующим главам, в которых детально будут разобраны основные аспекты сопровождения программного обеспечения, авторы считают необходимым ввести основные понятия. С этой целью предполагаются следующие определения.

Понятие *сопровождение* уже было определено выше. Однако с целью расширения значения этого понятия мы вводим еще одно определение и предполагаем выбрать любое из них. *Сопровождение* — это такая стадия жизненного цикла программного обеспечения, на которую часто переносятся затраты со стадии проектирования. Иными словами, все, что не доделал разработчик программного обеспечения, должно быть в конечном счете доделано на стадии сопровождения.

Заказчик — это лицо, которое хочет использовать программное обеспечение, и, как только последнее попадает к нему, он становится пользователем.

Система — это комплекс средств, взаимодействие которых приводит к полному решению поставленных задач. Система обычно состоит из большого количества программ и подпрограмм, а также может содержать компоненты, не входящие в программное обеспечение, такие, как аппаратные средства вычислительной машины и документация. В этой книге (и на это мы будем опираться в дальнейшем) рассматриваются решения на уровне системы, а не на уровне программ. Здесь мы не касаемся сопровождения 100-строчных программ. Хотя сопровождение программ такого размера требует не меньших затрат, чем их разработка и создание. В нашей книге рассматриваются задачи сопровождения программного обеспечения, на решение которых требуется несколько человеко-лет.

Сложность — это мера трудности понимания и сопровождения части программного обеспечения. Сложность должна сводиться к минимуму усилиями разработчика и сопровождающего программиста. Это достигается путем постоянного слежения за размерами модулей, входящих в программное обеспечение, за логической структурой этих модулей и за взаимодействием этих модулей между собой. Модулями повышенной сложности считаются такие, которые нельзя понять сразу. Парадоксальным является тот факт, что сложное программное обеспечение часто не поддается осознанию даже его создателем, не говоря уже о сопровождающем программисте.

На рис. 1.4-1 изображена структурная схема всех стадий сопровождения программного обеспечения, которая дает представление читателю о вопросах, обсуждаемых в последующих главах книги. Заметим, что процесс создания системы включает этапы определения технических требований, проектирования и реализации.

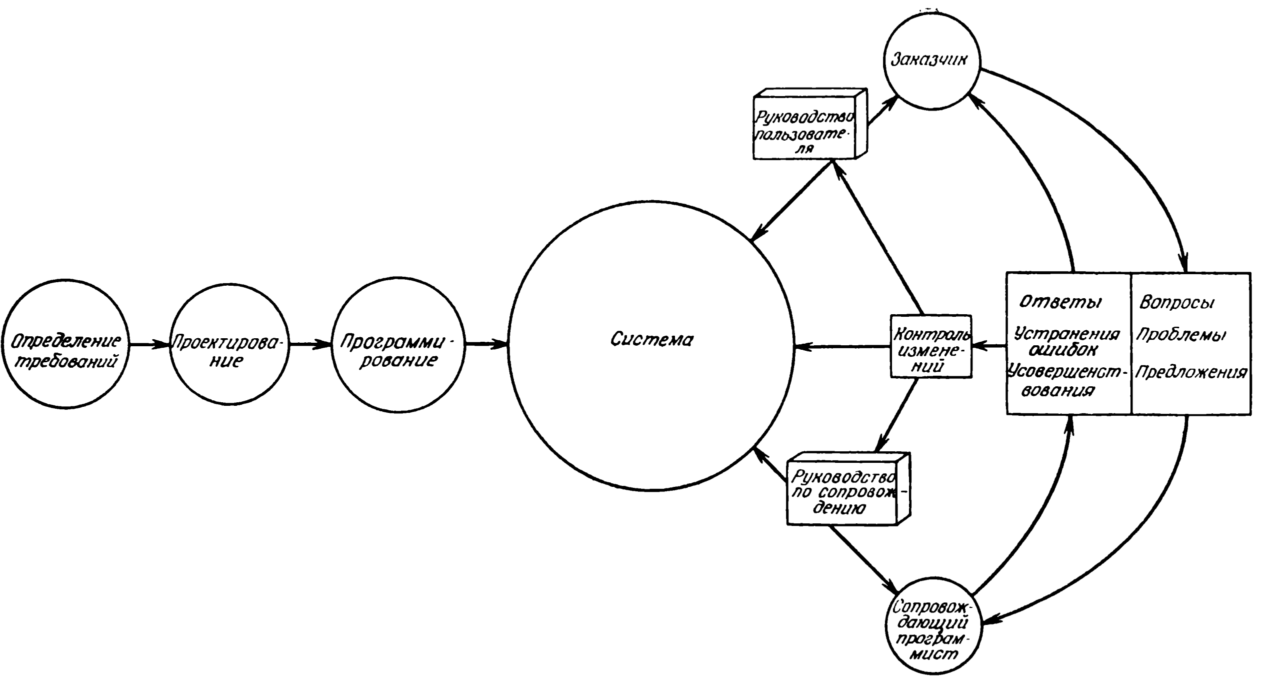

Рис. 1.4-1. **Общая схема** сопровождения программного обеспечения системы.

При этом заказчик и сопровождающий программист взаимодействуют с этой системой каждый со своим документом (руководством пользования и руководством сопровождающего программиста соответственно). Заказчик ставит перед сопровождающим программистом задачи, решения которых (если они вообще возможны), пройдя контроль изменений, включаются в систему. Каждое из приведенных понятий, а также некоторые другие будут подробно исследованы ниже.

ЛИТЕРАТУРА

1. Boehm, The High Cost of Software, Practical Strategies for Developing Large Software Systems, Addison-Wesley, 1975.
   Обсуждаются затраты на программное обеспечение и пути их снижения на разных стадиях жизненного цикла. Программирование считается самой дешевой стадией по сравнению со стадиями разработки технических требований, проектирования и реализации.
2. Alberts, The Economics of Software Quality Assurance, Proceedings of the National Computer Conference, 1976.
3. Lientz, Swanson, Tompkins, Characteristics of Applications Software Maintenance, UCLA Graduate School of Management, 1976.
4. Boehm, Software Design and Structuring, Practical Strategies for Developing Large Software Systems, Addison-Wesley, 1975.
5. Hecht, Sturm, Trattner, Reliability Measurement during Software Development, Proceedings of the AIAA Conference on Computers in Aerospace, 1977.
6. Stanfield, Skrukrud, Software Acquisition Management Guidebook, Software Maintenance Volume, System Development Corp., TM-5772 / 004 / 02, Nov. 1977.
7. Lientz, Swanson, Tompkins, Characteristics of Application Software Maintenance, *Communications of the ACM* (June 1978).
   Даны различные характеристики процесса сопровождения программного обеспечения, полученные на основе исследований 69 вычислительных центров. Определена роль сопровождения в вычислительном центре.

Глава 2

РОЛЬ ЛИЧНОСТИ ПРОГРАММИСТА В ПРОЦЕССЕ СОПРОВОЖДЕНИЯ

До недавнего времени роли личности программиста при создании программного обеспечения уделялось второстепенное значение. На этот счет имелись самые различные точки зрения, начиная от полного игнорирования труда программиста-одиночки [16] (вплоть до введения обезличенного программирования [32]) и кончая утверждением, что личность сопровождающего программиста является основным фактором, определяющим качество программного обеспечения [10, 20, 23, 27, 29]. В некоторых последних исследованиях высказывается промежуточная точка зрения [5, 36].

Однако специалистов, создающих программное обеспечение, мало интересуют эти точки зрения. Бурное развитие вычислительной техники привело к нехватке квалифицированных специалистов. Это положение существовало всегда и продолжает усугубляться. Квалифицированных кадров будет недоставать по меньшей мере в течение последующих 5 лет. Есть учреждения, где укомплектовано лишь 75% штата. Сколько бы мы ни спорили о том, является ли программирование уделом одиночек или это должен быть коллективный труд, ясно одно: сопровождающих программистов не хватает. Многие проблемы программного обеспечения мы вынуждены решать силами средних специалистов, а не талантливых одиночек. Это необходимо принимать во внимание при планировании.

За последние годы было сделано много попыток упростить подготовку специалистов по программному обеспечению путем снижения уровня требований при обучении. В начале 60-х годов говорили о том, что при наличии определенных методов любого человека можно обучить работе сопровождающего программиста. Надежды возлагались на совместное использование Кобола и автоматизированных методов написания программ. Сейчас над этим уже можно посмеяться. Однако и в наши дни находятся и ученые, которые убеждены в возможности существования таких методов, и предприниматели, которые финансируют современных сторонников этой идеи. Иллюзорная надежда на стандартизацию сопровождения программного обеспечения по-прежнему остается заманчивой.

В середине 60-х годов одна ведущая фирма по изготовлению аппаратных средств для вычислительных машин наивно полагала, что ей удалось разработать метод, позволяющий осуществлять

полуавтоматическую разработку программного обеспечения. Она поставила на поток производство компиляторов, ассемблеров и загрузчиков, использовав для этого неквалифицированных программистов. Эксперимент оказался неудачным: полученная в результате продукция не выдержала конкуренции на рынке и фирма потерпела убытки [11].

Итак, проблема заключается в следующем: слишком мало специалистов, способных справиться с комплексом работ по программному обеспечению все возрастающей сложности. Этим объясняется большое количество объявлений, приглашающих на работу квалифицированных программистов, и рост заработной платы программистов, в особенности вновь принятых на работу. (Нередко новичок получает большую зарплату, чем его коллега, проработавший в том же учреждении 1—2 года.)

Особенно эта проблема актуальна в области сопровождения программного обеспечения. Это объясняется тем, что, сколько бы мы ни противопоставляли личность программиста коллективу, количественную и качественную стороны сопровождения программного обеспечения, *оно по-прежнему остается уделом специалиста-одиночки*. Хорошо это или плохо, но сопровождающий программист чаще всего работает один на один с программой и берет всю ответственность за результат этой работы на себя. Это, конечно, не всегда так. Применяется и коллективное программирование. Однако сопровождение программного обеспечения можно сравнить с участком, заросшим сорной травой, в саду программного обеспечения. Этот участок требует своего садовника. Если садовник — человек, работающий по призванию, сорняки не страшны его саду. В противном случае они заполонят весь сад.

Все сказанное является предисловием к разговору о роли личности в сопровождении программного обеспечения. Здесь необходим «человек на своем месте». Проблема требует индивидуального подхода, но пока она мало изучена. И все же она заслуживает особого внимания: здесь расходуется 50% всех средств, вкладываемых в программное обеспечение.

Много слов сказано в адрес сопровождающих программистов. Некоторые из них хвалебные, другие далеко не лестные. Пожалуй, правильнее всего назвать их «невоспетыми героями». Им будет посвящен данный раздел этой книги, в котором мы постараемся рассказать, какими способностями должет обладать сопровождающий программист, какими приемами он должен владеть, какие у него возникают проблемы, какие из них следует решать в первую очередь, какова роль личности программиста и коллектива в процессе создания программного обеспечения и т. д. Короче говоря, мы поговорим о сопровождающем программисте как о личности.

2.1. ПРОБЛЕМА ПОДБОРА СПЕЦИАЛИСТОВ В ПЕРСПЕКТИВЕ

Чтобы показать важность проблемы подбора специалистов, приведем следующий пример[1].

Для сопровождения подбор специалистов имеет подчас далеко идущие последствия. Некоторые ошибки в сопровождении программного обеспечения представляют собой не просто легко устранимые недостатки программирования, а едва уловимые логические ошибки при проектировании или реализации.

Сложность состоит в том, что логическая ошибка может быть допущена на всех этапах, начиная от постановки задачи и кончая реализацией.

Чтобы пояснить вышесказанное, рассмотрим навигационный модуль сверхзвукового самолета. Предположим, что такой модуль должен обеспечить правильное местоположение самолета в любой точке атмосферы земли с точностью до 10 м. Модуль получает входную информацию от автопилота, акселерометров, часов, радаров, датчиков, связанных со спутниками и наземными станциями. Допустим, что все входные устройства функционируют нормально, а результаты работы модуля ошибочны.

Далее предположим, что в действительности ошибка представляет собой суперпозицию сигналов от трех отдельных источников: 1) ошибка в программе, возникшая в результате потери оператора, который присваивает глобальной переменной значение локальной переменной; 2) ошибка, возникающая в результате потери точности вычислений на режимах, где основная матрица системы является плохо обусловленной; 3) систематическая ошибка на траектории север — юг при больших числах Маха, возникающая из-за *еучета* датчиками кориолисовой силы.

Для устранения каждой из этих трех ошибок нужен свой специалист. Только физик, знакомый с законами кинематики и динамики, может понять и устранить ошибку, связанную с кориолисовой силой. Только специалист по численным методам анализа способен устранить ошибку, связанную с потерей точности вычислений, и только опытный программист, хорошо знакомый с правилами использованного языка программирования, сможет найти ошибку в программе и внести в нее необходимые исправления.

Фактически общая ошибка является суперпозицией трех различных по физической природе ошибок, и сомнительно, что сопровождающий программист, обладающий знаниями только в

---

[1] Пример взят из «Initial Thoughts on the Pebbleman Process», где дан анализ требований для оборудования алгоритмического языка Министерства обороны США Ада, представленных Д. А. Фишером (Institute for Defense Analyses) и Т. А. Стендишем (University of California, Irvine).

одной из имеющих отношение к ошибке областей, способен устранить ее.

Аналогичное происходит, если ставится задача усовершенствования системы и связанной с ним документации с целью доведения их до современного уровня. Для успешного решения необходимо искусство системных аналитиков и проектировщиков. Рост числа усовершенствований может привести к тому, что очередная модификация окажется более сложной, чем первоначальный анализ и проектирование системы. Сопровождающий программист, в большинстве случаев подготовленный лишь к работе на стадиях реализации и отладки, не в состоянии выполнить работу, посильную только для комплексной группы специалистов, работавший над системой на более ранних стадиях жизненного цикла. А такая группа к моменту усовершенствования системы уже распущена. Трудности такого рода проявляются особенно в тех случаях, когда усовершенствуемая система требует значительной доработки в области, не связанной с программированием.

Но мы знаем, что комплексная группа, создававшая систему, распущена, и сами обстоятельства заставляют переложить и сопровождение, и усовершенствование на плечи недостаточно компетентных специалистов. В таких случаях остается только один выход: найти способ вновь созвать ту группу специалистов, которая создавала систему, и предложить ей заняться обнаружением ошибок, их устранением и внесением усовершенствований в систему.

Из сказанного становится ясно, что некоторые проблемы сопровождения программного обеспечения и глубоки, и трудно разрешимы, и дорогостоящи.

В одном из отзывов[1] на эту книгу были выдвинуты и другие соображения по затронутому вопросу. Речь идет об уровне знаний и навыках, необходимых при сопровождении программного обеспечения.

1. Сопровождение требует от специалиста больших умственных затрат. По целому ряду причин возникающие задачи трудно ограничить определенными рамками.

2. Сопровождение очень сложно технически. Круг решаемых задач может быть обширным. Он включает программирование, проектирование, архитектуру, общие вопросы.

3. Сопровождение ненадежно, часто необходимая информация бывает либо недоступной, либо ложной. Специалист, начавший работу по сопровождению системы, нередко переходит на другую работу или получает повышение, не закончив написание и отладку программы. Даже готовая программа, попав в чужие руки, перестает быть полезной.

---

[1] Robert J. Rader of The Software Developers.

4. Сопровождение программного обеспечения непрестижно. Люди занимаются этой работой без желания.

5. Сопровождение — это неблагодарная работа. Она делается за кулисами, наедине с программой и машиной. В этой области мало славы, почестей и шансов быстро продвинуться по службе.

6. Сопровождение программного обеспечения основано на программах, написанных в прошлом. Бóльшая часть программного обеспечения несовершенна из-за того, что морально устарела. Кроме того, бóльшая его часть создается специалистами прежде, чем они достигают вершин мастерства.

Все эти соображения убеждают в том, что вряд ли стоит возлагать всю работу по сопровождению программного обеспечения на вновь принятых и малоопытных сотрудников.

2.2. ЛИЧНЫЕ КАЧЕСТВА СОПРОВОЖДАЮЩЕГО ПРОГРАММИСТА

Не стоит, пожалуй, слишком увлекаться образом «невоспетого героя». Так можно создать некий образ суперпрограммиста, похожего на человека, обладающего способностью перепрыгивать через небоскребы и при этом совершать строго определенное число прыжков или способного не отставать от мчащегося локомотива. Такая фантазия, несомненно, будет мешать пониманию вопроса и собьет с толку самого «невоспетого героя».

И все же реальные условия мира сопровождения программного обеспечения во многом напоминают эту фантазию. Ведь сопровождение программного обеспечения — это микромир, в котором, как в капле воды, отражается весь процесс создания программного обеспечения. В этом микромире есть сопровождающий программист, который проводит системный анализ проблемной области и определяет требования, согласно требованиям вносит изменения в программное обеспечение, программирует и отлаживает программы и, наконец, сдает готовую продукцию пользователю. В наш век узкой специализации сопровождающий программист должен уметь делать все, причем делать хорошо.

И более того, сопровождающий программист осуществляет связь с заказчиком, выясняет недостатки обеспечения и справляется о результатах их устранения. Это нелегкая задача, и здесь следует воздать должное «невоспетому герою».

Далее в книге будут показаны те черты характера, которые необходимы сопровождающему программисту. Это довольно редкие качества, и многими из них большинство сопровождающих программистов не обладает.

2.2.1. ГИБКОСТЬ В РАБОТЕ

Сопровождающий программист должен уметь приспосабливаться к стилям программирования, которые не свойственны ему самому. Важно уметь отличать программу, написанную плохо, от программы, которая написана в непривычном стиле. Здесь-то ему и потребуется гибкость. И тогда он не станет напрасно терять время, переделывая то, что ему не по вкусу, кроме, конечно, случая, когда эта переработка необходима в интересах дела. Каждый программист имеет индивидуальный стиль программирования. Программист, обладающий гибкостью, способен признать за другими право на собственный стиль и сможет работать, приспосабливаясь к этому стилю. (Проблема гибкости более подробно освещена с разд. 2.3.)

2.2.2. ШИРОКИЙ ПРОФЕССИОНАЛЬНЫЙ КРУГОЗОР

Чтобы обладать гибкостью, программисту необходимо ознакомиться со всеми существующими стилями. Начинающий сопровождающий программист будет стараться навязать программе свой стиль. Например, если специалист привык работать на Фортране, то, даже программируя на Коболе, он будет стараться написать программу в стиле Фортрана. Сопровождающий программист должен владеть как можно большим количеством машинных языков, чтобы, встречаясь с ними, уметь узнавать тончайшие разновидности стилей программирования.

Кроме того, он должен быть знаком с прикладными дисциплинами, связанными с программированием. Редко удается программисту проработать всю жизнь с одними и теми же заказчиками. Ему, например, нужно быть в курсе того, каким требованиям должен отвечать отчет управляющего фирмой, а каким — отчет начальника отдела.

2.2.3. ТЕРПЕНИЕ

В жизни сопровождающего программиста существует две области, где ему особенно пригодится терпение. Первая из них — связь с заказчиком. Само собой разумеется, что заказчик пытается сделать определенную работу с помощью системы. Ему известно, чего он хочет добиться от системы, но сопровождающий программист лучше его представляет себе, чего тот в действительности добьется от нее и почему. Сопровождающему программисту приходится подчас сталкиваться с очень нетерпеливыми людьми, которые требуют, чтобы им объяснили, почему система не работает. И нередко, если сопровождающий программист недостаточно терпелив, чтобы объяснить, почему желание заказчика не всегда совпадает с возможностями машины, возникают конфликты.

Второй случай, когда сопровождающему программисту приходится запастись терпением, — это когда речь идет о стабильности

системы. Заказчик обычно требует от системы стабильности. Он не переносит неожиданных изменений в принципах работы системы. В результате он может запретить или приостановить введение радикально новой программы, хотя сопровождающий программист не сомневается, что оно может существенно улучшить работу системы.

Возьмем к примеру программу, которая очень сложна, «почти» верна, но не работает. В этом случае сопровождающий программист может пойти двумя путями: 1) переделать всю программу заново и таким образом обойти все трудные места в ней; 2) доработать имеющуюся программу. Исходя из требования стабильности, второй путь более желателен. По мере обнаружения и устранения всех затруднительных мест постепенно вся программа приводится в порядок. При каждом запуске системы вносятся небольшие улучшения, и со временем программа будет отлажена. Очевидно, что для такого подхода сопровождающему программисту потребуется терпение.

2.2.4. САМОСТОЯТЕЛЬНОСТЬ МЫШЛЕНИЯ

Как уже говорилось, системный программист часто работает самостоятельно. Чтобы продуктивно использовать рабочее время, он должен уметь самостоятельно принимать решения. Программист, который нуждается в постоянном руководстве, не может и не должен быть сопровождающим программистом.

2.2.5. ОТВЕТСТВЕННОСТЬ

С самостоятельностью неразрывно связана ответственность. Программист должен чувствовать удовлетворение от своего труда. Одновременно он берет на себя ответственность за качество проделанной работы, иначе он будет непроизводительно расходовать рабочее время. К тому же он может внести беспорядок в уже сделанную ранее работу. Чувство ответственности, пожалуй, самое нужное качество в работе сопровождающего программиста.

2.2.6. СКРОМНОСТЬ И САМОКРИТИЧНОСТЬ

Наш «совершенный программист», обладающий столькими добродетелями, не раз услышит от раздраженного заказчика слова: «Ваша программа никуда не годится». Вот здесь-то ему и понадобится самокритичность, чтобы правильно отнестись к критическим замечаниям. Надо уметь спокойно реагировать, когда вам указывают на ошибки, даже если это очень неприятно. «Неужели опять не годится!» — говорит в таких случаях программист, своей кротостью обезоруживая раздраженного заказчика и превращая его из врага в союзника. Здесь также может помочь чувство юмора.

2.2.7. ТВОРЧЕСКИЙ ПОДХОД К ПРОГРАММНОМУ ОБЕСПЕЧЕНИЮ

Это самое трудное. Сопровождающий программист получает уже существующую программу со всеми ее специфическими особенностями. Ему приходится приспособиться к стилю и содержанию этой программы. Затем появляется необходимость внести в нее коренные изменения. Вот здесь от программиста и ожидают творческого подхода. Причем, внося изменения, требуется как можно меньше нарушать уже существующую структуру программы!

Несомненно, это самая сложная форма творчества. Поскольку главное условие таково: сделай что-то новое, не нарушив старого. Конечно, задача нелегкая, но экономия при этом может быть огромной.

2.2.8. ХОРОШАЯ ПАМЯТЬ

Сопровождающий программист должен всегда держать в памяти всю предшествующую работу. Необходимо зафиксировать хронологию создания системы. Чтобы видеть перспективы системы, следует создать перечень ошибок в хронологическом порядке. Не менее важно вести учет всех вносимых в систему изменений версий решаемых задач. Часто бывает нужно оглянуться на то, что уже было сделано, чтобы понять возможные пути и методы решений (более подробное обсуждение этого вопроса см. в разд. 3.2.3).

2.3. СТИЛИ И СТИЛЕВЫЕ КОНФЛИКТЫ

Многое делается для повышения качества программирования, облегчения труда программистов, упрощения программного обеспечения и решения других подобных задач. Особенно большое внимание уделяется снижению себестоимости, планированию и повышению надежности программного обеспечения. С точки зрения руководства, эти вопросы являются основными.

Однако на техническом уровне, где идет повседневная работа, на первый план выдвигаются иные, более конкретные задачи. В частности, очень важна проблема так называемого стиля. Об этом пока мало говорят.

В 60-х и начале 70-х годов появился ряд исследований о преимуществах модульного программирования (например, [21, 25]). В 70-е годы много писали о структурном программировании [1, 4]. Были работы, раскрывающие само понятие «стиль» [14, 17, 31, 36]. Недавно Ассоциация вычислительных машин Тьюринга (АВМТ) организовала цикл лекций о важности стиля (а также генерации алгоритмов), вводя основные понятия «парадигмы» [9].

Кроме того, большое внимание уделяется созданию и немедленному внедрению новых машинных языков. (Лучшими из используемых в последние годы были языки Паскаль [33, 34], Модула [35], Эвклид [28].)

Но, хотя проблема стиля затрагивается в работах по языкам, само понятие «стиль» — это нечто более широкое. Стиль — это слияние задачи и решения, предлагаемого программистом в одной или нескольких методиках этого решения. В результате программа имеет свой стиль. Стиль неотделим от программы. Все, что сказано выше, еще не является определением понятия «стиль». Мы предлагаем свое определение, хотя и сознаем, что оно не совсем точно. *Стиль — это отпечаток, налагаемый на программу программистом и свойственными ему методами работы.*

Значение стиля велико. Всякая программа имеет свой стиль. Каждому программисту свойственны стилевые особенности. Если программист получает программу, которая не соответствует его стилю, возникает стилевой конфликт. В результате программист отвергает программу, не желая приспосабливаться к ее стилю. При этом он начинает «изобретать велосипед», т. е. переписывает всю программу заново. В 50-е и, позднее, в 70-е годы были сделаны попытки пользоваться общими программами. Они оказались неудачными, и среди множества различных причин главной явилось несоответствие стилей программирования.

В наши дни можно сказать, что проблемы стилевых конфликтов уже не существует. Во-первых, структурное программирование основано на том, что существует лишь один наилучший способ программирования, а значит, если все будут дисциплинированно придерживаться его, проблема стиля будет решена. Во-вторых, существует мнение, что проблема стиля не поддается логическому осмыслению.

Оба эти аргумента легко опровергнуть. Программное обеспечение должно включать структурное программирование, но это не исключает существование стиля. Например, структурное программирование умалчивает о том, каким образом выбрать модули, чтобы в дальнейшем программное обеспечение легко поддавалось модификации и сопровождению. По мнению авторов, именно выбор модулей, а не структура программ влияет на стиль программного обеспечения, хотя важно то и другое.

Тот факт, что затраты на кодирование программ составляют лишь незначительную часть всех затрат на программное обеспечение, явился неприятной неожиданностью для большинства технологов программного обеспечения. В настоящее время это открытие осмысливается. Одним из последствий такого открытия явились попытки сосредоточить усилия исследователей на других стадиях жизненного цикла программного обеспечения, в частности на стадиях определения требований, проектирования, тестирования и верификации. Подобное перенесение затрат может отрицательно сказаться на сопровождении программного обеспечения. Несмотря на то что создание программы, как мы выяснили, обходится дешевле, она не становится от этого менее важной для сопровождаю-

щего программиста. Программа по-прежнему остается тем материалом, с которым ему приходится работать. А если вспомнить, что сопровождение требует 50% всех затрат на программное обеспечение, то создание программы, каких бы незначительных затрат оно ни требовало на стадии проектирования, по-прежнему остается жизненно важной частью всего процесса программного обеспечения.

А также и стиль. И следовательно, стилевые конфликты.

Все вышесказанное, однако, носит общий характер. В дальнейшем мы рассмотрим вопрос более подробно.

В качестве первой попытки устранения рассогласованности стилей проведем рассуждения, положив в основу языки программирования, так как именно их особенности в значительной степени порождают стилевые конфликты. Выбранная нами форма изложения представляет собой исторически сложившийся подход, хотя мы и не убеждены, что она является наилучшей. Проблема стилей заслуживает самого пристального изучения. (На важность этой темы только намекают в литературе [30]. Во многих учреждениях выделяют одного специалиста, в обязанности которого входит соблюдение единого стиля и устранение стилевых несоответствий, где это необходимо.) Одна из лучших работ, касающихся этой темы, противопоставляет традиционное программирование структурному [2]. Степень сложности программного обеспечения тесно связана со стилем [13].

Начнем обсуждение стилей в хронологическом порядке. Это наилучший способ, хотя он и не очень удобен для введения определений.

2.3.1. СТИЛЬ ПРОГРАММИРОВАНИЯ НА АССЕМБЛЕРЕ

Язык программирования Ассемблер предоставляет программисту с точки зрения стиля полную свободу. Но это не означает, что программы на Ассемблере не могут иметь и не имеют стиля, — напротив. Скорее, позволяя полностью исследовать все возможности ЭВМ, Ассемблер порождает огромное стилевое разнообразие.

Например, на Ассемблере можно создать уродливо смодулированную программу самой невероятной структуры, которая разрушает любое понятие о стиле. И наоборот, можно обеспечить безупречную организацию модулей, особенно если использовать предоставляемую Ассемблером возможность хорошей структурности программ.

Исторически, однако, сложилось так, что на Ассемблере кодируют те части программного обеспечения, которые, хотя и оказываются важными в конкретных случаях, в целом считаются стилистически слабыми; к ним относятся:

1. Программы, требующие хитроумных приемов (для повышения эффективности).

2. Самоизменяющиеся программы (для преодоления ограниченности набора команд).

3. Программы, связанные со спецификой аппаратуры ЭВМ (из-за того, что их иногда необходимо и проще кодировать на Ассемблере).

4. Программы, нарушающие стандарт (так как иногда задачи и стандарты бывают несовместимы!).

Таким образом, влияние Ассемблера на стиль программирования можно охарактеризовать как «эклектическое». Следующие разделы проиллюстрируют это понятие с помощью примеров.

2.3.2. СТИЛЬ ПРОГРАММИРОВАНИЯ НА ФОРТРАНЕ

Фортран явился первым языком, который в значительной степени разрушил и продолжает разрушать неразрывную связь программиста с вычислительной машиной. Как бы то ни было, Фортран оказал важное методологическое воздействие на стиль.

Главным достижением Фортрана с точки зрения стиля является его модульность. Он явился первым языком, в котором материализовалось понятие «модульность». Появились подпрограммы. (Хотя сам термин возник еще во времена Ассемблера.) Действительно, широкая возможность использования при написании программ на Фортране подпрограмм и стандартных функций дала право создавать программы, состоящие из отдельных моделей. Хорошо разработанные библиотеки стандартных подпрограмм, которыми снабжены почти все модификации Фортрана, являются ярким примером, иллюстрирующим преимущество модульного принципа программирования.

Однако проблема взаимосвязи данных между этими модулями была решена менее удовлетворительно. Метод *вызывающих последовательностей* был эффективным средством взаимосвязи, однако для больших массивов данных был разработан метод *общих блоков памяти* (COMMON), который нередко является источником ошибок. Общие блоки стали важной и трудной областью Фортрана.

Поименование переменных в Фортране еще более неудовлетворительно. Идентифицирование данных несовершенно, так как ограничено всего шестью символами, которых часто не хватает. Возможность структуризации программы также ограничена, а операторы скорее пронумерованы, чем поименованы. В результате программы на Фортране трудно читать.

Недостатком Фортрана является его непригодность к структуризации данных. Несмотря на очевидную необходимость использования во всех прикладных сферах величин, не связанных с размерами слова памяти, Фортран такой возможности не предоставляет. Даже попытки решить эту проблему путем использования библиотеки подпрограмм, ориентированных на работу с частями слова, приводят к решению задач на уровне операций, тогда как она должна

быть решена на уровне описаний. Таким образом, стиль программирования на Фортране представляет собой смесь привлекательной модульности с серьезными недостатками. Программист, работающий на Фортране, склонен писать трудно читаемые программы с неадекватным описанием.

2.3.3. СТИЛЬ ПРОГРАММИРОВАНИЯ НА КОБОЛЕ

Стиль Кобола является почти зеркальным отражением стиля Фортрана. Он обладает большими возможностями описания данных. Это объясняется возможностью точной спецификации редактирования данных при помощи их описания вместо выполняемых операторов или псевдовыполняемых операторов формата в Фортране, что очень удобно, так как делает язык гибким.

А файлы ввода/вывода Кобола ориентированы на область его применения, что является его преимуществом. Программы, написанные на Коболе, имеют хороший стиль только в том случае, если они решают задачи той прикладной области, для которой предназначен Кобол.

Однако в противоположность Фортрану у него отсутствует модульность. Возможность простой организации параграфов не заменяет передачи параметров и COMMON-связей подпрограмм Фортрана.

Кроме того, Кобол вынуждает программиста к многословию в программе и ставит его в строгие рамки ключевых слов и синтаксических требований. В результате программа пишется медленно, и скорость чтения программы на Коболе увеличивается ненамного.

Положительной стороной Кобол-стиля является доступность определенных крайне гибких общих модулей, таких, как модуль сортировки пакетов и модуль генератора отчетов, что реабилитирует стиль Кобола. Короче говоря, стиль Кобола построен на контрастах: с одной стороны, возможность прекрасной структуризации данных и совершенный ввод/вывод, с другой — слабый синтаксис и отсутствие модульности. Вследствие противоположности языков Фортрана и Кобола стили работающих на них программистов, несомненно, будут несовместимы.

2.3.4. СТИЛЬ ПРОГРАММИРОВАНИЯ НА АЛГОЛЕ

С точки зрения стиля Алгол является особенным языком. Хотя он мало применяется при написании программ (в основном он в ходу в Европе), его влияние на стиль не менее важно, чем влияние Фортрана и Кобола.

Сильной стороной Алгола является структурность его программ. В отличие от больших возможностей структуризации данных Кобола Алгол позволяет уточнить структуру программы с помощью

серии выполняемых операторов, которые можно рассматривать как единое целое или группировать их с целью облегчения чтения программы. Алгол позволяет создавать более удачные условные конструкции, чем Фортран и Кобол, и не уступает им при составлении циклов в программе.

Алгол частично решает задачу описания данных, которая не решена в Фортране, за счет их иерархического представления. Это позволяет локализовать данные в определенной области программы. Кроме того, Алгол позволяет программисту составить до некоторой степени самодокументирующиеся программы с помощью присвоения данным, программе и операторам значимых имен.

Но, к сожалению, Алгол имеет свои недостатки. Его программы не обеспечивают возможности раздельного компилирования — важного элемента модульности, хотя они сильнее, чем у Фортрана, так как можно записывать одну подпрограмму внутри другой.

И Кобол с операторами ввода/вывода, ориентированными на файл, и Фортран со своими форматами имели свой стиль ввода/вывода.

Благодаря Алголу возник третий стиль, отличный от стилей Фортрана и Кобола. В целом этот стиль более удобен, позволяет создавать программы с тонкой структурой, а также предполагает создание больших программ (с большим количеством внутренних подпрограмм), лишенных общих областей ввода/вывода.

2.3.5. ДРУГИЕ СТИЛИ ПРОГРАММИРОВАНИЯ

Прикладные лингвисты старались избавляться от недостатков различных стилей по мере их обнаружения. Они вносили дополнения, стараясь улучшить модульность Фортрана, описательную способность Кобола и структуру Алгола. Например, появилось понятие *кластер*, которое дает доступ к обычному набору функций и связанной с ним базой данных. Появились константы как вычислительные, так и описательные, которые облегчают преобразования. Были введены также языковые формы, позволяющие оптимизировать структуру программы и свести ее сложность до минимума.

Несомненно, что стили Фортрана, Кобола и Алгола несовершенны. Несовершенны и улучшения, о которых сказано выше. Однако наша цель состоит не столько в отыскании лучшего стиля, сколько в раскрытии причин возникновения различных стилей и стилевых конфликтов.

Сопровождающий программист будет работать эффективно только в том случае, если захочет и сумеет приспособиться к существующему разнообразию стилей. В противном случае он принесет больше вреда, чем пользы.

Стилевые конфликты невозможно обсуждать объективно. В большинстве случаев программисту полезно проявить гибкость и

попытаться преодолеть конфликт. Но есть случаи, когда здравый смысл требует проявления настойчивости. Вопросы преодоления стилевых конфликтов требуют больших затрат и поэтому заслуживают пристального внимания.

2.4. ЗАДАЧИ СОПРОВОЖДЕНИЯ И ИХ ПРИОРИТЕТЫ

Внимательный читатель уже мог заметить, что задачи, возникающие в процессе сопровождения программного обеспечения, не так очевидны, как это может показаться на первый взгляд. Исправление ошибок в программном обеспечении — лишь одна из немногих задач, которые приходится решать в процессе его сопровождения. Поскольку задачи сопровождения программного обеспечения сложны и разнообразны, то необходимо установить очередность их решения по степени их важности, т. е. присвоить им приоритеты. Сопровождающий программист может избежать многих досадных ошибок, если будет не просто представлять себе круг требующих решения задач, но в точности знать их приоритеты. Специалист, которому доверяют такую ответственную работу, обязан сам уметь решать, какие задачи важнее.

2.4.1. НАДЕЖНОСТЬ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Под надежностью программного обеспечения будем понимать возможность заказчика использовать программное обеспечение и получать результаты. Обеспечение надежности работы программ является первоочередной задачей сопровождающего программиста. Надежность программного обеспечения стоит любых затрат труда.

Если из-за частых изменений работа программы становится нестабильной, прогоните ее еще раз, а затем не трогайте некоторое время. Копите ваши последующие изменения, но не вносите их до тех пор, пока вновь не поверите в ее работоспособность.

2.4.2. ИСПРАВЛЕНИЕ ОШИБОК

Если в программном обеспечении обнаружены какие-либо ошибки, то обычно (но не всегда) их следует устранить как можно скорее. Эта вторая по значению задача сопровождающего программиста.

Однако существуют исключения. Иногда лучше оставить недоработку, чем перенести сроки передачи системы заказчику. В программном обеспечении существуют различные ошибки, начиная с тех, которые задерживают использование программы, и кончая настолько необычными, что они могут встретиться раз в тысячелетие. Все остальные случаи лежат между этими двумя крайностями.

Уровни приоритетов, присваиваемых ошибкам, зависят от степени их серьезности. В некоторых крайних случаях затраты на поиски ошибок обесценивают смысл самого поиска. В этом случае недостаток никогда не стоит устранять!

2.4.3. ВНЕСЕНИЕ ИЗМЕНЕНИЙ В СИСТЕМУ

Задача внесения изменений в систему аналогична задаче устранения ошибок. Если принять во внимание, что изменение должно быть произведено по распоряжению руководства (разд. 4.2.1), внесение этого изменения становится первостепенной задачей. Однако, так же как и в случае с устранением ошибок, приоритет задачи зависит от характера изменений.

В целом устранение ошибок следует считать более важной задачей, чем внесение изменений в систему. Однако очевидно, что внесение изменений, обладающих высоким уровнем приоритета, — более важная задача, чем устранение ошибок, имеющих низкий приоритет. Вопрос о том, что важнее, должен быть согласован с заказчиком.

2.4.4. СОПРОВОЖДЕНИЕ РАДИ СОПРОВОЖДЕНИЯ

Пожалуй, самым странным является мысль о сопровождении как о самоцели. Ее трудно бывает объяснить руководству, так как все действия, направленные на облегчение процесса сопровождения, не входят в рамки работ для заказчика или не соответствуют требованиям заказчика к программному обеспечению [12]. В некоторых случаях, однако, разбор запутанной программы и расширение области применения программы или прочтения трудной программы могут в конечном счете оказаться более важными задачами, чем устранение ошибок. Чем легче сопровождение программы, тем она более полезна и тем больше срок ее службы. И все же в большинстве случаев задача обеспечения простоты и удобства процесса сопровождения имеет более низкий приоритет, чем задачи, описанные в разд. 2.4.1, 2.4.2 и 2.4.3.

2.4.5. ЭФФЕКТИВНОСТЬ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Основными характеристиками программного обеспечения являются объем памяти, быстродействие, затраты, потребность в сопровождающем персонале и т. п. Поэтому под эффективностью здесь мы будем понимать оптимизацию одной (или более) из этих характеристик. Очевидно, что одна из первоочередных задач состоит в том, чтобы сделать сопровождение программного обеспечения более эффективным.

Оценивая эффективность, следует принимать во внимание степень важности этих трех характеристик. Если для выполнения программы не хватает объема памяти или быстродействия вычислительной

машины, то проблема эффективности становится актуальной. Однако, как правило, проблема эффективности имеет самый низкий приоритет.

2.4.6. ВЕДЕНИЕ ДОКУМЕНТАЦИИ

Опытный программист, занимающийся сопровождением программного обеспечения, понимает, что аккуратное ведение документации является важной частью его повседневной работы. С этой точки зрения программист должен походить на мастерового, который всегда содержит свой инструмент в идеальном порядке.

К сожалению, такое отношение к документации нередко отсутствует. Поэтому следует взять за правило: всякое изменение в программе должно быть зафиксировано в документации сразу же по окончании коррекции программы и еще до того, как в программу будет внесено новое изменение.

Хотя этот раздел идет последним, значение документации стоит далеко не на последнем месте. Скорее — наоборот. Внесение изменений в документацию должно происходить одновременно с исправлением программы.

Объем работ, связанных с ведением документации, будет существенно сокращен, если придерживаться рекомендаций данной книги: документирование должно являться составной частью ЛИСТИНГА программы (об этом говорится подробно в разд. 4.3). Во многих случаях изменения, вносимые в программу, столь незаметны, что их не нужно формально документировать, и поэтому документация не должна быть изменена. При использовании описанного выше подхода к документации поддерживать ее в порядке обычно бывает несложно.

2.5. ОГРАНИЧЕНИЕ ИЗМЕНЕНИЙ

Однажды, выступая по телевидению, один из президентов Соединенных Штатов Америки оправдывал несоответствие платежей по социальному страхованию стоимости жизни тем, что очень трудно изменить вычислительные машины, другими словами, трудно изменить программное обеспечение, управляющее платежами по социальному страхованию. Программисты, привыкшие к тому, что их обвиняют всегда и во всем, отнеслись к этому высказыванию весьма равнодушно, поскольку оно было, с их точки зрения, результатом неосведомленности президента в вопросах вычислительной техники. Однако, по словам одного из ведущих специалистов в области вычислительной техники, «самые простые на первый взгляд изменения порой бывает невообразимо трудно претворить в жизнь, так как это дорого и сложно. Мы нуждаемся в лучшем систем-

ном анализе и более совершенных средствах программирования»[1] .

В предыдущих главах этой книги мы затрагивали проблему изменения программного обеспечения в строго ограниченных рамках. Вернемся к этой теме. Каждая программа обладает своим стилем и структурой. Если программа сделана добросовестно, она имеет гибкий стиль и гибкую (в пределах разумного) структуру. Но гибкость имеет свои границы. Она определяется способностью системного программиста предвидеть те изменения, которые придется вносить в программное обеспечение. А такое предвидение всегда имеет границы. В особенности в наш век, когда человек научился летать, передавать звуки и изображения на расстояние и создавать логические устройства, трудно надеяться, что системный программист способен предугадать все сферы применения своего детища.

Большие затруднения с точки зрения обработки информации вызвало в начале 80-х годов решение о расширении плана почтовых зон (ППЗ) Почтового ведомства США и переводе его на более длинные индексы. В результате все имеющиеся программы, обслуживавшие Почтовое ведомство, устарели и нуждаются в изменении и приведении в соответствие с новым ППЗ. Если бы специалисты по программному обеспечению проявили настойчивость, Почтовому ведомству не позволили бы вносить изменения в план. (Забавно, что именно программисты, которые в середине века были носителями перемен и нововведений, теперь выступают как решительные противники всяческих перемен.)

По крайней мере в некоторой степени такая позиция неверна. Вычислительная техника это все-таки прикладная область, призванная упрощать различные процессы и удешевлять их. В данном случае и изменения в Почтовом ведомстве и в системе социального страхования есть прогрессивные общественные перемены, и здесь требуется проявить гибкость.

В этой связи следует сосредоточить внимание, во-первых, на важности вопроса создания гибкого программного обеспечения, а во-вторых, на возможности внесения изменений в рамках уже существующих систем. Последнее положение более подробно рассматривается в разд. 3.2.2.1, где речь идет о том, как сделать программное обеспечение более гибким. Этот вопрос больше связан с обслуживающим персоналом, чем с правилами программирования и методикой написания программ.

Ввести новое в уже существующую систему и при этом не разрушить старого — очень сложная задача. Программисту приходится принимать во внимание весь объем работ, проделанных его предшественниками, и поддерживать точку зрения, что для того, чтобы приспособить программное обеспечение к новым требовани-

[1] McCracken, ACM President's Letter, *Communications of the ACM,* **21**, No. 12 (December 1978).

ям, его не обязательно ломать. Ему приходится все продумывать самому и ценой напряженной работы искать оригинальное решение, которое может оказаться в противоречии с его личным стилем. Здесь важно не впасть в искушение бросить работу или избрать дорогостоящее решение «с позиции силы».

Это важные требования. Здесь они рассматриваются потому, что от их выполнения во многом зависит способность программного обеспечения удовлетворить требованиям заказчика. Специалист, способный вводить в программу новое, обусловленное не только требованием свыше, но и требованием своих коллег, заслуживает самых лестных отзывов.

Однако, кроме внутренних ограничений (требований, предъявляемых вышестоящими звеньями и своими коллегами), существуют внешние ограничения. О них следует сказать.

Нужды заказчика — одно из внешних ограничений — будут обсуждаться в следующем разделе. Если они оговорены в контракте или внесены в спецификации, сопровождающему программисту неизбежно предстоит иметь с ними дело.

Еще существуют такие внешние факторы, как стоимость машинного времени и риск. Часто бывает, что, если данная модификация системы не вписывается в график работы машины или в смету расходов, она не принимается.

Надежность результата, о которой мы говорили как о задаче программирования, является одновременно и ограничением. Если модификация системы снижает надежность, ее не следует вводить.

Еще одним ограничением является совместимость изменений. Совместима ли уже существующая программа с планируемым изменением? Этот вопрос может стать фактором, ограничивающим изменение. Если внесение изменения превратит четкий, отлаженный как часы механизм в груду старого хлама, то такое изменение вводить не нужно. Программист, знающий свою работу до тонкости, должен сам решить вопрос о целесообразности изменений.

И наконец, документация, как ни странно, часто является фактором, ограничивающим модификацию. Затраты, необходимые для внесения изменений в документацию (например, справочник пользователя), могут сделать модификацию нежелательной.

Умение обойти все эти ограничения — одна из немаловажных задач сопровождающего программиста.

2.6. НУЖДЫ ЗАКАЗЧИКА

Вот еще одна тема, требующая освещения. Однако здесь потребуется вернуться на два десятилетия назад. В истории человечества время от времени появлялись такие профессии, которые котировались выше других. Возьмем к примеру популярных спортсменов или кинозвезд, заработки которых значительно превосходят

самые смелые мечты обычных людей. К таким профессиям относятся и те, которые возбуждают воображение масс. В детстве все мечтают стать пожарниками или полисменами. В более зрелом возрасте появляются мысли о журналистике или преподавании. Двадцать лет назад профессия программиста была также престижной. Программист — человек, способный управлять загадочной человекоподобной машиной, человек, которому подвластно управление обществом, — занимал тогда исключительное положение. Признаться, это было довольно приятно. Можно было играть чувствами неосведомленных людей. Но со временем люди получили реальное представление о работе программиста. Специалистов в этой области стало более 70 тысяч, и почти у каждого есть теперь знакомый программист. Благоговение и восхищение исчезли.

Программисту остается тешить себя мыслью, что он является полновластным хозяином в своей области. Руководство, конечно, постоянно пытается вторгаться в эту область с помощью различных методов административного контроля. И все же справедливо, что программист один способен вникнуть во все тонкости своей работы. Здесь он по-прежнему полновластный хозяин. Но так ли это?

Вспомним, что всякое программное обеспечение делается по заказу, а программирование имеет целью обслуживание заказчика. Для программиста, даже если он купается в лучах славы, очень важно понимать, что он создает лишь инструмент для заказчика, а не абстрактно-красивую, но бесполезную игрушку. Будет лучше, если программист представляет себе свою работу как банковскую систему, которая выписывает ему чеки. Когда его труд не будет оплачен, ему будет легче понять, что заказчик плохо обслужен.

Здесь уместно привести примеры. Один программист, опасаясь, что компания намерена его уволить, решил помешать этому и начал создавать хитрые программы в расчете на то, что станет незаменимым. Тем самым он, несомненно, пренебрег интересами заказчика и в результате был, конечно, уволен [12]!

Другой, более положительный пример. Программист подчинил свое стремление создать более совершенную программу интересам заказчика. Программа представляла собой математическую модель процесса внутреннего сгорания в двигателе ракеты. Надо было, не запуская ракету, установить, какую силу тяги могут развить различные варианты двигателя. В целом заказчика интересовал оптимальный химический состав вещества, из которого следует изготовить двигатель. Программист, зная общую цель заказчика, сумел так построить программу, чтобы учесть в ней все возможные дополнения, которые мог сделать заказчик. В результате программа стала уникально эффективной (такого общего решения еще никто раньше не предлагал), и у программиста появились все основания быть довольным собой.

Иногда нуждами заказчика объясняются на первый взгляд

нелогичные поступки. Рассмотрим в качестве примера систему, которая дает сведения о производстве продукции в цехе. Одна группа сообщений указывает рабочим, какие операции и в каком порядке следует производить. Другая сообщает администрации, какие детали находятся в процессе производства и где. Само собой разумеется, что выходные данные такой программы должны строго соответствовать действительности. В нашем примере это не удалось. Сообщения запаздывали и в течение двух дней из пяти последних в месяце были неправильными. Понимая, что поставлено под сомнение доверие к системе в целом, программист преднамеренно отключил систему и отказался вносить изменения в течение двух недель. Для программиста это был период вынужденного бездействия, но для программы и заказчика он стал жизненно необходимым периодом стабильности. Иногда в интересах заказчика лучше бездействовать, чем делать что-либо позитивное!

Наши примеры поясняют тот факт, что удовлетворение нужд заказчика — хотя и важная, но не имеющая точных рамок задача. Здесь, как нигде, программисту необходимо умение мыслить самостоятельно.

2.7. ЛИЧНАЯ ОТВЕТСТВЕННОСТЬ

Мы уже затрагивали проблему личности и коллектива. У этой проблемы есть по крайней мере один аспект, где значение личности очень велико, каким бы коллективным ни был труд программиста. Когда программист сделал часть программы или документацию к нему, он должен поставить под работой свою подпись. Речь здесь идет не столько о праве авторства, сколько о личной ответственности.

Полезно привести пример. Предположим, что программист *A* — первоклассный специалист, но его слабым местом является документирование, а программист *B*, наоборот, ведет документацию прекрасно, но разрабатывает и кодирует программы посредственно. При чтении справочника по сопровождению программного обеспечения, созданного совместно *A* и *B*, полезно будет знать, кто что сделал и кто за что в ответе. Если мне не ясна запись, сделанная *A*, или программа, выполненная *B*, то по крайней мере будет понятно, к кому обратиться за разъяснением.

Такая подпись должна стоять под каждой страницей документации и под каждым отрезком программы. (И, что не менее важно, должно быть указано число.) Тогда при возникшей необходимости внести изменения в материал легко будет установить имя автора (и число).

ЛИТЕРАТУРА

1. Baker, Structured Programming in a Production Programming Environment, Proceedings of the IEEE International Conference on Reliable Software, 1975.
Рассматривается проблема структурного программирования и стандарт программирования в связи с методикой структурного программирования.
2. Black, Curnow, Katz, Gray, BCS Software Production Data, RADC-TR-77-116, 1977.
Дано определение современному практическому программированию на основе опыта одной компании. Противопоставляется традиционное программирование современным методам, дано подробное описание традиционного стиля программирования.
3. См. [1] к гл. 1.
4. Brooks, The Mythical Man-Month, Addison-Wesley, 1975.
Даны ценнейшие сведения об управлении программным обеспечением и обслуживающим персоналом. Автор делится практическим опытом применения оперативных систем OS/360, определяет роль личности и ее место в коллективе, занимающемся сопровождением программного обеспечения.
5. Human Factors in Software Engineering, *Computer* (December 1979).
Содержит статьи о роли личности в создании программного обеспечения. Авторы (Шнайдерман, Базили, Райтер, Ганпон и др.) исследуют проблемы проектирования интерактивных систем, преимущества современной методики и частоту обращения данных.
6. DeMillo, Lipton, Sayward, Hints on Test Data Selection: Help for the Practicing Programmer, *Computer* (April 1978).
Авторы отстаивают интуитивные методы тестирования, считая, что все остальные подходы к проблеме не учитывают той общеизвестной истины, что большинство проверяемых программ «почти верны».
7. Dijkstra, The Humble Programmer (1972 Turing Award Lecture), Communications of the ACM, 1972.
Обсуждается эволюция профессии программиста. Подчеркивается необходимость повышения качества и надежности методов программирования.
8. A Study of Fundamental Factors Underlying Software Maintenance Problems, ESD-TR-72-121, Vol. 11, 1971.
Даны несколько интервью с программистами по вопросам сопровождения программного обеспечения. Содержатся также дневники программистов и отчеты о различных случаях из их практики.
9. Floyd, The Paradigms of Programming, *Communications of the ACM* (August 1979).
Обсуждается важность основных понятий программирования в развитии программного обеспечения. Основные понятия названы парадигмами (в нашей книге набор таких парадигм заменен понятием «стиль»). В работе развиваются идеи лекции автора, удостоенной премии в 1979 г.
10. Frank, The New Software Economics, *Computerworld* (January 8, 1979).
Автор считает, что дальнейшее повышение продуктивности программного обеспечения маловероятно. Он убежден, что основным фактором дальнейшего повышения продуктивности обеспечения является использование в большей степени индивидуальных возможностей каждого специалиста.
11. Glass, Computing Breakthrough Becomes Breakdown, The Universal Elixir and Other Computing Projects Which Failed, Computing Trends, 1977.
Автор делится опытом использования неквалифицированных программистов для развития программного обеспечения.
12. Glass, Software Reliability Guidebook, Prentice-Hall, 1979.
Дан обзор методов повышения надежности программного обеспечения. Обсуждается взаимосвязь между надежностью обеспечения и сопровождением. Изложена методика создания программного обеспечения, обладающего высокой надежностью.
13. Special Collection on Software Sgience, *IEEE Transactions on Software Engineering* (March 1979).
В статьях сборника обсуждаются вопросы сложности, стиля, сопровождения программного обеспечения.

14. Kernighan, Plauger, The Elements of Programming Style, McGraw-Hill, 1978.
Дается более 100 правил, определяющих хороший стиль. Правила подкреплены примерами и их решениями. Приведем некоторые из них: «При составлении модулей используй подпрограмму», «Данные определяют структуру программы», «Не комментируй плохую программу — перепиши ее».
15. Kenney, Staffing the Albatross Project, *Datamation* (February 1976).
Всесторонний анализ вопросов укомплектовывания штатов для реализации проектов с напряженным бюджетом.
16. Kraft, Programmers, Managers – The Routinization of Computer Programming in the United States, Springer-Verlag, 1977.
Выражается сожаление по поводу потери квалификации специалистами, занятыми в больших программах. Отмечает стандартизацию задач, выполняемых ранее творческими людьми.
17. Ledgard, Programming Proverbs for Fortran Programmers, Hayden, 1975.
Дается подход к проблеме стиля через «пословицы» и правила. Книга содержит правила Фортран-стиля. Автор книги написал подобные книги и о других языках.
18. Lientz, Swanson, Tompkins, Characteristics of Application Software Maintenance, UCLA Graduate School of Management, 1976.
Дается обзор работ 69 вычислительных центров с целью описания характеристик сопровождения программного обеспечения. Изучается в основном оборудование, ориентированное на Кобол и очень нуждающееся в сопровождении.
19. См. [7], гл. 1.
Более современный вариант работы [18].
20. Love, Sheppard, Curtis, Milliman, Borst, First-Year Result from a Research Program on Human Factors in Software Engineering, Proceedings of the National Computer Conference, 1979.
Дается отчет об экспериментах, поставленных для выяснения факторов, влияющих на эффективность работ программиста. Индивидуальные особенности программиста, как видно из экспериментов, имеют важное значение.
21. Myers, Reliable Software through Composite Design, Petrocelli / Charter, 1975.
В работе отстаивается положение, что модульность является ключом к эффективному программированию. Даются рекомендации, как правильно составлять модули и сочетать их между собой. В работе подчеркивается важность стадии разработки.
22. Myers. Module Design and Coding, Software Reliability, Wiley-Interscience, 1976.
В работе обсуждаются языки высокого уровня, а также структурное программирование, его правила и техника программирования.
23. Myers, A Controlled Experiment in Program Testing and Code Walkthroughs / Inspections, *Communications of the ACM* (September 1978).
В работе описывается эксперимент по поиску ошибок с применением тестирования и пересмотра программы. Автор считает этот метод работ полезным и необходимым. Он полагает, что методы тестирования могут быть очень разнообразными, и рекомендует использовать их для сравнения результатов двух специалистов, работающих независимо друг от друга.
24. Overton, Colin., Tillman, Research Toward Ways of Improving Software Maintenance, ESD-TR-73-125, 1973.
В работе описывается эксперимент, позволяющий оценивать объем работ по сопровождению программного обеспечения. Излагается методика обслуживания и ремонта технических средств для программного обеспечения. Обсуждается применение графических терминалов для процесса сопровождения.
25. Parnas, On the Criteria to Be Used in Decomposing Systems into Modules, *Communications of the ACM* (December 1972).
На конкретном примере рассматриваются два способа определения модулей. Рекомендуется модулировать, отдавая предпочтение компоновке информации, таким образом, чтобы модули содержали конструктивные решения, которые могут изменяться.
26. Parnas, The Influence of Software Structure on Reliability, Proceedings of IEEE International Conference on Reliable Software, 1975.

Даны различия между надежностью (с точки зрения оказания услуг пользователю) и корректностью (удовлетворение требованиям спецификаций) программного обеспечения. Подчеркивается важность последней и рекомендуется способ формирования модулей для ее обеспечения. Освещаются вопросы, связанные с определением модуля.

27. Peters, Tripp, Comparing Software Desing Methodologies, Datamation, November 1977.
Представляет собой анализ пяти различных методов проектирования. Дается оценка методов, основанных на применении триады. Автор делает вывод, что ни один из них не является универсальным и отмечает, что в «конечном счете система создается проектировщиком, а не методами».
28. Popek, Horning, Lampson, Mitchell, London, Notes on the Desigh of Euclid, Proceedings of an ACM Conference in Language Design for Reliable Software, 1977.
Дано описание языка программирования под названием Эвклид, который обеспечивает возможность проверки программы. Работа содержит подробный анализ языка Эвклид и связанного с ним определения кластер-модуля.
29. Rubey, Higher Order Languages for Avionics Software – A Survey, Summary and Critique, NAECON 1978.
В работе прослеживается история применения языков высокого уровня в авиации (в реальном масштабе времени). Даются задачи и их решения с помощью языков высокого уровня в противопоставлении Ассемблеру. Автор отмечает тенденцию к применению языков высокого уровня, но высказывается мысль, что «главным в вопросе обеспечения качества программ остается мастерство программиста».
30. Soroka, Letter to the Editor, SIGPLAN Notices, December 1978.
Автор высказывает мысль, что каждый специалист имеет индивидуальный стиль, и предлагает провести дальнейшее исследование по этому вопросу.
31 Van Tassel, Program Style, Design, Efficiency, Debugging and Testing, Prentica-Hall, 1974.
Работа представляет собой практическое руководство по программированию, основанное на обобщении опыта работы большого числа различных профессиональных программистов.
32. Weinberg, The Psychology of Computer Programming, Van Nostrand Reinhold, 1971.
В работе обсуждается вопрос «обезличенного программирования» и индивидуального владения программой. Автор является сторонником коллективного метода создания программ. В книге разбираются случаи из практики.
33. Wirth, The Programming Language Pascal, *Acta Information*, **1** (1971).
Работа знакомит читателей с языком Паскаль, его возможностями и формой.
34. Wirth, An Assessment of the Programming Language Pascal, Proceedings of the International Conference on Reliable Software, June 1975.
Дается оценка возможностей языка Паскаль.
35. Wirth, Modula: A Language for Modular Multiprogramming, *Software Practice and Experience*, **2**,1 (1977).
В работе охарактеризован язык Модула, основанный на языке Паскаль и используемый при написании системных программ для малых вычислительных машин. Приведенные в работе статьи посвящены структуре языка, а также его применению и использованию.
36. Shneiderman, Software Psychology, Winthrop, 1980.
Работа посвящена изучению психологических особенностей системных программистов; она содержит разделы, касающиеся стиля программирования, коллективного программирования и роли личности в коллективе.

Глава 3

ТЕХНИЧЕСКИЙ АСПЕКТ СОПРОВОЖДЕНИЯ

Процесс создания программного обеспечения можно сравнить с процессом воспитания человека. Степень приспособленности систем к выполнению своих функций зависит, так же как у человека, от суммы знаний, заложенных в них на ранних стадиях разработки. В данной главе мы рассмотрим различные методы создания удобного и надежного программного обеспечения.

Если, читая эту главу, вы надеетесь получить ответ на вопрос «Как сократить затраты на сопровождение?» применительно к уже работающей системе, то вы опоздали. Сложность и возможность модификации программы уже определены, а следовательно, определены и затраты. Задайтесь тем же вопросом применительно к вашей будущей системе.

Это не означает, что в существующей системе бесполезно что-либо изменять радикально. Придерживаясь методики, изложенной в книге, можно постепенно превратить трудномодифицируемую систему в такую, которую модифицировать не слишком трудно. Однако, так же как и в случае воспитания ребенка, формирование правильного поведения системы лучше всего начать на самой ранней стадии ее развития.

В предыдущей главе достаточно хорошо сформулированы задачи специалиста по сопровождению и определены приоритеты решений этих задач. В данной главе мы постараемся подробно рассмотреть способы их решения. Рассмотрим также технические приемы сопровождения и будем ими пользоваться. К сожалению, их невозможно определить так же точно, как задачи специалиста по сопровождению программного обеспечения (СПО). Для того чтобы выполнить поставленную перед ним задачу, сопровождающий программист руководствуется определенной схемой, включающей требования пользователя, существующую программу, все доступные ему средства, окружение и его личные возможности.

Сопровождающий программист несет ответственность за все, что относится к программному обеспечению. Наиболее очевидным результатом хорошего сопровождения является безошибочная работа всего программного обеспечения в целом. Справочники пользователя часто включают ошибки, неточности и недоработки, которые приводят к затруднениям при работе пользователя. Исправление

документации отнимает у сопровождающего программиста много времени. Но документирование возникающих в ходе работы проблем и возможных путей их устранения необходимо для работы пользователя. Спецификации программного обеспечения могут также содержать неточности и ошибки. И их исправление входит в функции сопровождающего программиста.

Итак, специалист по СПО встречается с разного рода задачами, которые обычно решаются вручную или с помощью соответствующей данному случаю методики. Однако с увеличением объема программного обеспечения становится все очевиднее, что программист для решения своих задач должен применять новейшую технологию программирования. Это достигается посредством использования методов системного анализа, а также современных методов автоматизации программирования. Системный анализ ведет к лучшей технологии программирования, а автоматизация дает лучшие средства программирования. Разбор технологии и средств программирования является темой настоящей главы.

Но прежде, чем перейти к обсуждению этой темы, рассмотрим работу сопровождающего программиста. Тем самым мы определим требования, предъявляемые к технологии и средствам программирования.

3.1. ЧЕМ ЗАНИМАЕТСЯ СОПРОВОЖДАЮЩИЙ ПРОГРАММИСТ

Одна из трудностей руководителя, контролирующего работу сопровождающего программиста, состоит в том, что он часто не знает, что делает этот программист. Здесь мы постараемся показать весь процесс его работы. Возьмем для примера одну задачу и рассмотрим весь перечень работ, связанных с ее решением. Затем повторим этот процесс много раз и перетасуем его составные части. В результате мы получим представление о том, как много работ выполняет сопровождающий программист.

Специалист по сопровождению обычно тратит много времени, анализируя необработанные данные о предполагаемых причинах ошибок. Обычно все начинается с появления раздраженного заказчика. Претензии могут быть самыми разными в зависимости от типа программного обеспечения системы. Например, пользователь программы, с помощью которой готовится платежная ведомость, может сказать: «У меня цифры не сходятся. В вашей программе ошибка!» В то же время пользователь компилятора может сказать: «Моя программа ведет себя как-то странно. Вероятно, в компиляторе ошибка».

Порой, чтобы отличить настоящие ошибки от воображаемых, сопровождающему программисту приходится потратить немало времени. За это время он составляет определенное мнение о причине ошибок. Он может задать пользователю много вопросов, воспользо-

ваться справочниками (например, справочником пользователя), спецификациями программы, документами по сопровождению, распечатками программ. В течение этого времени он должен установить истину: ошибка ли это пользователя, сбой ли это в программе или программа работает верно, но нечетко. Может быть, допущена неточность в документации? Или все же дело в программе? Учтите также, что размышлять приходится в самый неподходящий момент.

И когда сопровождающий программист занят решением какой-то определенной задачи, его прерывают, и очередной вновь пришедший клиент жизнерадостно спрашивает: «Можно вас на минуточку?» Как тут оставаться спокойным!

Итак, он отправляет своего предыдущего собеседника, наскоро пообещав ему внести исправление в руководство пользователя. Но у этого вновь пришедшего в самом деле что-то серьезное. После 20-минутного разговора и просмотра дополнительной информации оба приходят к выводу, что действительно что-то неверно. Сопровождающий программист берет журнал, в котором фиксируются замечания пользователя, и начинает регистрировать отказ в программе. Заказчик падает духом. «Как, неужели ОН не сможет исправить это за 20 минут?» — думает пользователь. Вслух он ничего не произносит, но эта мысль написана на его лице. Подумав 1—2 мин, сопровождающий программист находит какой-нибудь выход из положения, записывает его в журнал (на случай, если кто-то столкнется с подобной проблемой), а пожалуйста! Еще один довольный заказчик отправляется по своим делам.

Вот теперь он может действительно подумать о причинах ошибки. (Он пришел в субботу в 10 ч 30 мин вечера, чтобы поработать спокойно.) Из своей предварительно разложенной по степени важности стопки бумаг он достает одну — с надписью «Очень срочно!». Берет длинный список возможных ошибок в программах, руководство пользователя, спецификации, документацию и, обложившись всеми этими материалами, принимается за дело. Читает, строит графики, повторяет все это по многу раз и очень быстро (время летит незаметно, когда занимаешься любимым делом) обнаруживает ошибку в программе. Возможно, что так бывает.

А теперь вернемся к реальности. Если бы все программы писались «идеальными программистами», документация строго отражала бы каждый шаг в их мышлении (включая и те варианты, которые были отброшены), программы были бы совершенными, четкими, имели бы простую структуру, простые модули и в них бы не было ошибок (перепутанных строчек и т. д.), то сопровождение было бы просто развлечением. Но в жизни все обстоит иначе. Большинство программ претерпевает значительные изменения. Поэтому найти ту часть программы, где требуется внести исправление, нелегко. Тем более нелегко сделать это исправление.

Однако вернемся к нашей задаче. Есть ошибка в программе. Ее следует устранить. Но это еще не все. Надо провести кропотливую работу, чтобы гарантировать качество. Сопровождающий программист должен проверить исправленную им программу. В большинстве случаев проверка покажет качество всей программы в целом. Добавьте специальную проверку, направленную конкретно на решенную вами задачу. Придется потратить время и на это. Иногда проверка проходит легко. Но не всегда. Иногда обнаруживается дефект в программе. Надо вновь искать ошибку и устранить ее. Наконец все работает! Но закончена ли работа специалиста по СПО? Нет. Теперь следует оформить документацию, написать примечания для пользователя, записать возникшие в ходе работы проблемы. Вот теперь все. К сожалению, по этой схеме сопровождающие программисты никогда не работают.

3.1.1. СОПРОВОЖДАЮЩИЙ ПРОГРАММИСТ И ПОЛЬЗОВАТЕЛЬ

Как вы уже поняли, сопровождающий программист тратит много времени, отвечая на вопросы, устраняя замечания пользователя, анализируя требования и даже помогая в вопросах, не имеющих прямого отношения к программированию. Известно, что пользователь не всегда хорошо разбирается в системе, а сопровождающий программист знает ее до тонкостей. При обучении пользователей можно пойти по двум направлениям: 1) предварительно обучать, как пользоваться системой, или 2) консультировать их во время работы с системой. Часто необходимо и то, и другое. В результате эта работа также ложится на сопровождающего программиста.

Кроме того, сопровождающий программист не всегда находится в непосредственной близости от пользователя. Чем доступнее он для пользователя, тем больше претензий выпадает на его долю.

В любом случае сопровождающий программист оказывает дополнительную помощь пользователю, который постоянно обращается к нему со своими проблемами. Это делает сопровождение более дорогостоящим, так как отнимает у программиста время.

3.1.2. СОПРОВОЖДАЮЩИЙ ПРОГРАММИСТ И ЕГО ЖУРНАЛ

Ошибки, с которыми сталкивается сопровождающий программист, бывают старые и новые. Сначала рассмотрим исправление новых. Следует разработать методику учета всех возникающих в ходе работы затруднений. Описание каждой новой ошибки должно засылаться в базу данных. Там хранятся все случаи, с которыми уже сталкивались программисты. Такая засылка может быть осуществлена вручную или, еще лучше, автоматически. Многие потенциальные ошибки могут быть быстро устранены при обращении к отчету, содержащему список уже известных ошибок.

Каждая ошибка должна быть детально охарактеризована с двух точек зрения: пользователя — о сути ошибки и способе ее устранения и программиста — о ее возможной причине. Кроме того, может потребоваться дополнительная информация, сопроводительная документация и т. п.

В любом случае необходима формальная регистрация всех сбоев и ошибок. Эта тема развивается подробно в разд. 4.2.2.

3.2. КАК РАБОТАЕТ СОПРОВОЖДАЮЩИЙ ПРОГРАММИСТ

В этом разделе мы рассмотрим средства и технику работы сопровождающего программиста. Средства (в основном автоматизированные) включают как традиционное программное обеспечение, с которым работает программист-разработчик, так и уникальные средства, необходимые сопровождающему программисту. Технология программирования (в основном ручная) вырабатывается на протяжении многих лет поколениями программистов.

3.2.1. СРЕДСТВА

Как уже говорилось, сопровождение составляет значительную часть жизненного цикла программного обеспечения, которую чаще, чем другие этапы цикла, не учитывают. Программное обеспечение существует в первую очередь для того, чтобы способствовать все более широкому применению вычислительных машин, чтобы заменить труд человека трудом машины. Но в большинстве случаев в самой области программного обеспечения для решения собственных задач вычислительные машины не применяются.

Следует признать, что «системе» предстоит просуществовать хотя и долгий, но все же не бесконечный отрезок времени. Необходимость сопровождения также следует признать и заранее запланировать.

Существует ли такая часть программного обеспечения, которая бы сразу удовлетворила пользователя, не имела ошибок и не требовала модификации? Вряд ли.

Наличие ошибок и длительность процесса сопровождения требуют долгосрочного планирования. Такое планирование, как и в других областях, требует применения соответствующих (включая машинные) средств. Пришло время дать их проектировщикам программного обеспечения и сопровождающим программистам. Время, когда дамп (вывод данных и контроль правильности выполнения операции вывода путем сопоставления контрольных сумм) был единственным средством сопровождения, прошло. Мы нуждаемся в более эффективных средствах. (Понятие «средства» подробно дается в [23].)

Существуют две основные группы средств, используемых при сопровождении. Первая группа связана непосредственно с технической стороной сопровождения. Она включает: компиляторы; ассемблеры; редакторы связей; операционные системы; захватывание и трассировку; дампы; символьный вывод; анализ по уровням; компараторы; анализаторы, использующие полное множество тестов; средства верификации; средства аттестации; варианты тестирования; таблицы перекрестных ссылок; редакторы текста; преобразователи и препроцессоры.

Вторая группа средств дает возможность сопровождающему программисту выполнять административные функции. Эти средства включают: составление перечня ошибок, ведение журнала отказов, фиксации вносимых поправок, ведение документации и написание отчетов.

Обе категории средств могут применяться либо в пакетном режиме (пользователь ставит задачу и приходит за готовым решением), либо в интерактивном режиме разделения времени (пользователь вмешивается в работу программы с помощью терминала). Применяемые средства в этих случаях будут различны из-за задержки во времени при работе в пакетном режиме.

Авторы рекомендуют при всякой возможности работать в интерактивном режиме, так как он значительно продуктивнее. Однако приходится признать, что в большинстве вычислительных центров реальное положение дел, к сожалению, пока не соответствует этой рекомендации.

Конечно, интерактивный режим имеет свои преимущества и недостатки. Преимуществом работы в интерактивном режиме является возможность для сопровождающего программиста избежать с помощью терминала многократной прогонки программы, которая во многих вычислительных центрах требует несколько дней. При этом ход мысли сопровождающего программиста не прерывается из-за необходимости дожидаться конца прогона программы.

Однако у этого метода есть и недостатки. Сопровождающий программист, работая в пакетном режиме, может оформить документацию, пока у него идет программа. В интерактивном режиме такой возможности у него нет. Кроме того, в результате оперативного устранения ошибки с помощью терминала может быть принято поспешное, плохо продуманное решение. Чтобы этого не случилось, программист должен быть очень дисциплинированным и внутренне организованным.

И все же преимущества работ в интерактивном режиме убедительно показаны экспериментально. Такая точка зрения изложена в литературе [43, 47]. Обобщая все сказанное, можно заключить, что в процессе разработки программного обеспечения больше всего не хватает думающих специалистов. Поэтому важны такие средства, как работа в интерактивном режиме. Они позволяют сделать мыслительную деятельность более продуктивной.

3.2.1.1. Средства сопровождения

Средства, о которых пойдет речь ниже, — это то, что необходимо сопровождающему программисту для работы.

Язык программирования

Выбор языка программирования является одним из наиболее важных решений, которое приходится принимать по отношению к программному обеспечению. Обычно сопровождающий программист не участвует в выборе языка, однако необходимо учитывать влияние, которое язык окажет на сопровождение. Мы настоятельно рекомендуем применять языки самого высокого уровня как при проектировании системы, так и на этапе ее реализации.

Тема описания процесса проектирования будет изложена ниже. Здесь, однако, следует заметить, что форма описания процесса проектирования (которая все чаще в наши дни является языком программирования) должна выбираться с учетом ряда ее экспрессивных возможностей:

1) декомпозиции системы на подсистемы;
2) выбора интерфейса как внешнего, так и внутреннего;
3) определения структуры данных и потока данных системы;
4) представления структуры алгоритма и управления.

Еще важнее, пожалуй, чтобы форма описания процесса проектирования системы:

5) была совместима с выбранным языком программирования;
6) позволяла включать комментарии в программу.

Два последних требования делают языковую форму описания процесса проектирования наиболее предпочтительной. Более старые способы, такие, как использование структурных схем при написании программ, неудобны для этапа сопровождения, так как с трудом поддаются модификации и поэтому быстро устаревают.

Мы считаем, что языки более высокого уровня должны использоваться не только на этапах проектирования и реализации системы, но также на протяжении всего жизненного цикла программного обеспечения, так как зачастую различные модули системы запрограммированы на разных языках, что неудобно для всей системы в целом.

Бывает так, что бо́льшая часть системы только с виду пригодна для применения единого языка на всех стадиях ее жизненного цикла, однако вскоре становится ясно, что без жесткого администрирования количество языков, применяемых в системе, быстро растет. Во многих случаях системный программист приходит к выводу, что отдельные части системы не поддаются описанию с помощью «официального» языка системы. Тогда он делает исключение и начинает применять другие языки.

И здесь первая ошибка, которую он может совершить, состоит в применении языка Ассемблера вместо языка высокого уровня (ЯВУ).

Обычной причиной такого заблуждения является добросовестность программиста. Он, например, рассуждает так: «Этот компилятор не обеспечивает возможности эффективного вычисления специальных функций; значит, необходимо составить программу их расчета на Ассемблере».

Вторым препятствием к применению единого ЯВУ для всей системы является недостаток мастерства программиста. Если в работе участвуют программисты, не имеющие опыта работы с языком, используемым в системе, то по соглашению с администрацией им разрешают для некоторых программ применить другой язык высокого уровня или Ассемблер.

Кроме того, множество применяемых вспомогательных средств не считается частью системы, поэтому их создателям разрешается выходить за рамки требований единого ЯВУ. В особенности это бывает в тех случаях, когда вспомогательные средства создаются в процессе работы и официально не считаются частью системы. Такие случаи выходят за рамки проекта, и соответствующие программы часто пишут на другом языке.

В результате нередко можно встретить «систему», состоящую из множества программ, написанных на четырех или пяти различных языках.

Рассмотрим теперь сопровождение такой «системы». Ее коррекция потребует специальных знаний (а точнее, знания различных языков). Обстоятельное обсуждение важности языков высокого уровня читатели найдут в литературе [3, 41, 42, 46, 58, 59].

Следует заметить, что сам по себе язык без компилятора не является средством сопровождения программного обеспечения. Об этом говорится в следующем разделе.

Компилятор

Самым необходимым и подчас единственным средством сопровождающего программиста является компилятор [2, 41].

Каждая часть программного обеспечения существует в виде «логического образа». Заказчик формулирует задачу с помощью спецификаций. Разработчик придает «образу» осязаемую форму, оговорив в деталях все части задачи и определив границы между ними. Вырабатываются требования к языку, который следует применить при решении задачи. Устанавливается тип машины, которая будет использоваться. Разработчик предлагает свое решение задачи, и все множество ограничений и требований к обеспечению материализуется в программе. Насколько хорошо программа отвечает своим задачам, судить приходится обычно многим людям.

Будь то программа, написанная на ЯВУ, или это — набор команд в кодах машины, в конечном счете появляется *исходная программа*, которую *компилятор* преобразует в рабочую программу, а затем машина выдает ее распечатку (*листинг программы*). Поскольку

листинг программы представляет собой последовательность команд машины, выполнение которых приводит к решению, опытные программисты считают его реальной программой. Когда в программу необходимо внести исправления, то они обычно вносятся в листинг, выведенный либо на бумагу, либо на экран дисплея.

Ввод	Обработка	Вывод
1. Прикладная программа(ы), написанная на каком-либо языке программирования	1. Перевод (трансляция) исходной программы на язык машины	1. Объектный модуль, получаемый в результате трансляций исходной программы (программ) 2. Листинг исходной программы, дополненный: а) соответствующими заголовками, включая идентификацию версии компилятора, время / дату компиляции, имя программы; б) нумерацией строк; в) диагностикой ошибок; г) таблицей перекрестных ссылок; д) распределением памяти; е) результатами компиляции, включая распределение памяти, внешние ссылки, ресурсы, используемые в процессе компиляции 3. Файлы, используемые средствами интерфейса, если они есть

Пример: Программа, написанная на языке Ада, вводится в компилятор вычислительной машины VAX. Выходом компилятора является объектная программа (объектный модуль), допускающая ее перемещение и дальнейшую обработку на микро-ЭВМ Z8000, и вспомогательный листинг.

В значительной мере компилятор используется для того, чтобы получить информацию о программе. Язык и компилятор позволяют программисту переключить внимание с процесса работы вычислительной машины непосредственно на решение задачи.

Компиляторы и языки должны не только предоставлять дополнительные логические возможности для решения задач, но также увеличивать потенциальные возможности операционной системы (ОС). Обычно ввод/вывод в более или менее развитой форме существует во всех языках, но средствами языка следует добиваться и более совершенных способов манипулирования данны-

ми. Такая информация, как время дня, ресурсы, ссылки, предоставляется почти всеми типами ОС. Однако в рамках существующих языков запросы на эту информацию осуществляются только через ОС и обычно на Ассемблере. А это значит, что программисту приходится иметь дело с излишними деталями.

Современные компиляторы хотя и обрабатывают программы, написанные на разных языках, но обеспечивают лишь самую основную информацию о транслируемой программе. Она обычно имеет вид таблиц перекрестных ссылок (содержащих перечень всех имен, использованных в программе, с указанием мест их использования) и описания (списков параметров в том порядке, в каком они хранятся в памяти). Эти списки, хотя и неполные, очень важны для работы сопровождающего программиста.

В дополнение к этой информации компилятор может также обеспечивать получение следующей:

Таблица процедурно-ориентированных перекрестных ссылок. Большинство программ, предназначенных для получения таблиц перекрестных ссылок, в настоящее время предоставляют информацию в виде имен переменных и номеров операторов в программе, где эти переменные встречаются. Такая информация важна, но не всегда достаточна. Полезнее было бы вместо номера оператора дать, например, информацию о поименованных процедурах. Имена процедур, использовавших переменную, вызывающих данную процедуру или вызванных ею, обеспечивают значительно больше информации программисту, чем номера операторов, в которых встречается какая-либо переменная или имя процедуры.

Такую информацию в настоящее время получают до и после компиляции, хотя на самом деле ее может и должен выдавать компилятор.

Справка о структуре данных. Еще одна функция, которую может выполнять компилятор и которая в ряде случаев бывает очень важна, это распечатка справки о структуре данных. Под структурой данных понимается такая их совокупность, в которой поименована не только вся эта совокупность, но и составляющие ее элементы. Очень важна информация о том, где производится обращение к элементам структуры данных или где они преобразуются, а также некоторая дополнительная информация о контексте, в котором эти данные появляются. Весьма полезно также иметь список всех этих справок для каждой структуры, в особенности при выдаче процедурно-ориентированной информации, о которой было сказано выше.

Редактор связей

Редактор связей обычно позволяет объединить все программы в единое целое. Его функция заключается в обеспечении доступа на внутрипрограммный уровень для разрешения вопросов внешних ссылок и в подготовке программы к выполнению.

Ввод	Обработка	Вывод
1. Независимо откомпилированные объектные модули	1. Распределить объектные модули в памяти вычислительной машины для загрузки 2. Решить вопросы внешних ссылок с помощью библиотеки стандартных подпрограмм или других внешних модулей	1. Общая программа, готовая для загрузки и выполнения

Пример: На вход редактора связей ЭВМ CYBER поступают объектные модули для программного обеспечения системы PAYROLL, а также информация о месте вспомогательных стандартных программ в базе данных. Редактор связей записывает всю программу в память ЭВМ CYBER и выдает модуль, готовый к загрузке и выполнению.

Однако существующие редакторы связей слишком часто налагают ограничения на возможности программиста. Например, они ограничивают пользователя шести- или восьмизначными внешними именами. Такая практика сохраняется еще с тех пор, когда Фортран и Ассемблер были единственными языками программирования. Но теперь, когда появились языки, позволяющие присваивать более длинные и легкочитаемые имена, а именно в них больше всего нуждаются сопровождающие программисты, необходимо, чтобы и редактор связей мог работать с более длинными именами. Несоответствие должно быть устранено. Например, для недавно появившегося в Министерстве обороны США языка Ада, обслуживающего три области техники, требуется редактор связей, возможности которого по обработке имен согласовывались бы с соответствующими возможностями языка [41]. Таким образом, мы находимся на пути к устранению этого недостатка.

Необходимо также иметь возможность получать таблицу перекрестных ссылок всех имен, известных системе (а не только внешних). Очевидно, что редактор связей может собрать и обработать таблицы имен, полученных от компилятора путем отдельных трансляций, с тем чтобы получить полную таблицу перекрестных ссылок. Однако это редко удается.

Существует и еще одна функция, которая может выполняться редактором связей, но редко выполняется им. Она заключается в проверке списков параметров. Использует ли вызывающая подпрограмма ту же последовательность вызовов, которая предусмотрена в вызываемой подпрограмме? Для внешних программ такую проверку может обеспечить только редактор связей.

Итак, сопровождающий программист имеет инструмент редактирования связей, который, прежде чем стать полезным, претерпел длительную эволюцию.

Операционная система/исполнительная программа

Как операционная система, так и исполнительные программы обычно создаются одновременно с техническими средствами и, взаимодействуя с ними, позволяют вычислительной машине выполнять свои функции. Они предоставляют пользователю много услуг.

Различие между ОС и исполнительными программами обычно зависит от использования системы. ОС применяются в том случае, когда вычислительная машина используется для общих целей: имеется много пользователей с различными требованиями к вычислительным возможностям машины. Услуги, предоставляемые ОС, состоят в управлении ресурсами, управлении данными и управлении задачами. Язык, используемый для связи с ОС, обычно грубый, низкого уровня и ничем не напоминает язык высокого уровня или даже Ассемблер.

Ввод	Обработка	Вывод
1. Прикладные программы и исполнительные системы	1. Технические возможности, предоставляемые машиной прикладным программам: а) плановое обслуживание; б) распределение ресурсов; в) возможность прерываемого обслуживания	1. Правильно выполненные прикладные программы

Пример: Система PAYROLL реализована на вычислительной машине CYBER. Операционная система ЭВМ CYBER устанавливает очередность выполнения программ, начинает выполнение, обеспечивает ввод/вывод, контролирует другие выполнения, пока система находится в режиме ожидания, и заканчивает выполнение.

Программы, выполняющие точно такие же или подобные функции в режиме реального времени, называются *исполнительными программами*. Исполнительная программа обычно специализируется на конкретном приложении. Хотя между операционной системой и исполнительной программой нет никаких принципиальных различий, на практике они представляют собой две отдельные части программного обеспечения, которые включаются в работу, исходя из потребностей области применения.

Для сопровождающего программиста скорее представляет интерес способ, которым операционная система либо исполнительная программа предоставляет услуги, необходимые пользователям. Возможно, между двумя рассматриваемыми типами ЭВМ и их операционными системами существуют различия в этом вопросе. Но над физической связью с операционной системой пользователь задумываться не должен, и такие детали, как драйвер, стек, очередность, буферизация и т. д., не должны быть ему известны. Все, что ему необходимо, — это лишь удобный доступ к тем функциям, которые требуются для решения его задачи. Степень достижения этой цели в значительной мере определяет, насколько удобно сопровождающему программисту осуществлять изменения программ. Это также влияет на интерфейсы операционной системы. Часто удобство сопровождения требует выделения интерфейсов в модули прикладных программ, с тем чтобы, если понадобятся изменения, их можно было бы внести только в эти модули.

Встроенные средства отладки

В любой большой программе должна быть предусмотрена возможность получения информации о ее внутреннем состоянии. С помощью большинства методов доступа общего назначения невозможно получить эту информацию в таком формате, который позволил бы немедленно оценить состояние программы. Поэтому значительная часть программы должна содержать специализированные средства доступа, позволяющие сопровождающему программисту «увидеть» данные внутри программы в ходе ее выполнения. Роль таких специализированных средств для ключевых переменных могут выполнять операторы печати, а в более общем случае — программы трассировки или операторы, которые выдают информацию о нарушении условия (например, если РАЗМЕР$\leq$100, то результат неверен).

Очевидно, что нормальное использование программы встроенных средств отладки не потребует. Более того, при работе системы в реальном масштабе времени они требуют дополнительных затрат памяти и времени счета, что нежелательно. Но сопровождающий программист должен иметь возможность провести дополнительный анализ ошибок [35, 41].

Обычно такую возможность предоставляют программы, которые печатают необходимые данные в символической форме (например, СКОРОСТЬ = 55). Этими программами управляют с помощью параметров. При обычных условиях работы программы управляющий параметр не используется (находится в состоянии «off»). Когда сопровождающему программисту информация нужна, он просто осуществляет выполнение того же самого задания, переводя управляющий параметр в состояние «on».

Необходимость физического удаления отладочной части про-

Ввод	Обработка	Вывод
1. Отладочные данные, получаемые в ходе работы программы: а) трассировка данных; б) трассировка логики; в) неправильные результаты	1. Распечатка данных в виде, удобном для чтения	1. Распечатка требуемой информации либо в составе выводимой программы, либо в виде отдельного файла: а) изменяемые имена и их величины; б) индикация выполняемых логических элементов программы; в) диагностика неправильных результатов

Пример: Программа, написанная на языке Ада и ранее скомпилированная, не проходит тест. Программист, используя трансляцию, вставил в исходный текст программы отладочные команды для трассировки переменных ALPHA и BETA и входов процедур Он также определил диапазон изменений переменных BETA и GAMMA и просит, чтобы были выданы все те их значения, которые не входят в этот диапазон. Для компиляции (с использованием средств условной компиляции), редактирования и повторного выполнения программы необходима программа отладки, которая обеспечивает трассировку и диагностическую информацию. Программист получает результаты отладки, которые помогают ему решить задачу.

граммы возникает достаточно редко. Техника условной компиляции позволяет включать или исключать фрагменты исходного текста с помощью управляющей программы. Таким образом отладочная часть может быть вставлена или изъята из всей программы.

Некоторые новые системы (например, [17]) позволяют осуществлять отладку программы, которая не требует применения специальных отладочных операторов. Это — достаточно мощные системы, которые могут обеспечить те средства отладки, в которых возникает необходимость. Однако пока эти системы еще не нашли широкого применения, поэтому предварительное планирование средств доступа играет важную роль при отладке программ.

Мы считаем, что каждая большая программа на треть или на четверть должна состоять из встроенных отладочных операторов. На практике это встречается редко, так как при разработке системы не учитываются потребности специалистов по сопровождению.

Компаратор

В работе сопровождающего программиста возникает целый ряд ситуаций, когда компаратор (или блок сравнения) может быть весьма полезен. Компаратор — это программа, которая берет два текста, сравнивает их и находит различия между ними. Самыми очевидными применениями компаратора являются сравнение двух

исходных программ, сравнение вариантов тестов, сравнение результатов тестирования.

Ввод	Обработка	Вывод
1. Два файла, содержание которых надо сравнить	1. Сравнить файлы на идентичность. Если содержание неидентичное, компенсировать различие и синхронизировать файлы	1. Распечатка, указывающая места несовпадения файлов: содержание обоих файлов, когда они не совпадают

Пример: Вариант программы PAYROLL, выполняемый сегодня, отличается по результатам от варианта прошлого месяца. Оба варианта загружаются в компаратор, и получается листинг тех мест, где имеются различия.

Необходимым условием применения компаратора является подобие двух данных, которое и обеспечивает их сравнимость. Сравнение яблок и апельсинов никогда еще не приносило пользы. (Не будет ли результатом этого сравнения лимон?)

Возможность сравнивать две различные версии одной и той же программы и получать соответствующий листинг их различий может быть непосредственно использована специалистом по сопровождению. Обычно ему задают следующий вопрос: «В чем различие между вариантами X и Y этой программы?» или «Эта программа работала, а теперь перестала. Что изменилось?» Компараторы всегда готовы помочь ответить на эти вопросы.

Анализ результатов тестирования — еще одна сфера применения компаратора. Сопровождающий программист выполняет два варианта программы с тестовыми данными и получает результаты. Он заинтересован в обнаружении всех тех мест, где имеются различия, вызванные внесенными изменениями. В результате может быть установлена причина этих различий.

В обоих описанных выше случаях на выходе компаратора должна получаться информация, соотносимая с данными ввода. Например, при сравнении двух листингов желательно получить сравнительный листинг источников с отмеченными в нем различиями или отдельно список различий. Простые различия, как, например, порядковые номера перфокарт, могут быть опущены, а вот содержание строк исходной программы является важным различием и должно быть ясно отмечено. Было бы хорошо, если бы изменения формата строки исходной программы учитывались только на уровне опций. В любом случае и те различия, которые опускаются, и те, которые регистрируются, должны легко поддаваться описанию.

При создании компаратора возникают следующие проблемы. Во-первых, мы должны создать либо очень «хитрый» компаратор на все

случаи жизни, либо целую серию их, для каждого типа сравнения свой. Во-вторых, следует уделить внимание ресинхронизации процесса компарации, с тем чтобы из-за начального несоответствия все последующие не регистрировались. В-третьих, это средство, как и все остальные, должно быть экономичным в использовании.

К счастью, уже существуют коммерческие программные средства, которые отвечают всем этим требованиям. Например, программа FILCOM для DEC 10 представляет собой универсальное средство, которое выдает выборочную информацию о различиях, одновременно выполняя ресинхронизацию. Она предназначена для сравнения исходных файлов (программ, вариантов тестов, результатов тестирования и т. д.).

Пусть, например, два файла состоят из данных

I.		и	II.
	A		*A*
	B		*B*
	C		*D*
	D		*F* 1
	E		
	F		

соответственно. В этом случае программа FILCOM выдаст на выходе

1) *C*
1) *D*
* * *
2) *D*
* * * * * * * (Эта запись означает отсутствие символа *C* в файле 2)
1) *E*
1) *F*
***
2) *F* 1
* * * * * (Эта запись сообщает, что на месте символов *E* и *F* в файле 2 стоит символ *F* 1)

Редактор текста

Редактор текста необходим, поскольку сопровождение предполагает внесение изменений в исходную программу, а с его помощью текст программы может быть дополнен, уничтожен или исправлен. Простейшими являются редакторы, обрабатывающие отдельную строку исходной программы, что позволяет вставлять или аннулировать отдельные записи. Применение подобных средств можно сравнить с работой механика, вынужденного чинить автомобиль, обходясь только отверткой. Когда требуется изменить один знак в строке, приходится заменять всю строку, при этом увеличивается вероятность возникновения других ошибок. Редакторы текста должны быть просты в обращении, обладать большими возможностями и позволять осуществлять взаимодействие с пользователем.

Перемещение сегментов программ должно легко осуществляться с помощью простых команд редактора. Замена одной последовательности операторов, многократно встречаемой в программе, на другую должна производиться одной командой. А неизбежные ошибки, связанные с побочными эффектами, необходимо свести к минимуму и, где это возможно, следует выдавать соответствующие сообщения.

Ввод	Обработка	Вывод
1. Текст 2. Команды, управляющие текстом	1. Выполнение команд управления текстом: а) стереть информацию; б) вписать информацию; в) заменить информацию; г) распечатать информацию	1. Отредактированный текст

Пример: Переменная величина BETA, определенная в одном из модулей программы, написанном на языке Ада, по ошибке используется в другом модуле, который также содержит переменную BETA. Кроме того, первоначальное значение переменной BETA присвоено неверно. Для того чтобы все BETA во втором модуле программы заменить на BETA LOCAL и исправить первоначальное значение переменной BETA, используется редактор текста.

Ниже будут перечислены возможности, которые предоставляют программистам имеющиеся в настоящее время редакторы текста.

1. Редакторы текста, ориентированные на обработку строки (выделяется строка, которую следует редактировать):
   а) стирание;
   б) вставка;
   в) замена;
   г) распечатка.
2. Редакторы текста, ориентированные на обработку цепочек (выделяется цепочка, которую следует редактировать):
   а) поиск;
   б) замещение.
3. Редакторы текста, ориентированные на обработку блоков (выделяются блоки, которые следует редактировать):
   а) копирование;
   б) замена.

Как видим, им под силу почти любая манипуляция с исходным кодом, которая может понадобиться сопровождающему программисту. Например, их можно использовать при редактировании программы для того, чтобы:

1. Заменить неправильную строку программы исправленной (замена).

2. Вставить три новые дополнительные строки (вставка).

3. Заменить все ссылки к переменной RMS на ROOT—MEAN—SQUARE (замещение).

4. Обнаружить места всех ссылок на процедуру DIAGNOSTIC (поиск).

5. Сохранить и старую, и новую версии программы (копирование).

Некоторые редакторы текста выполняют и более интеллектуальные операции. Они, например, могут писать «программы», состоящие из цепочек команд, где каждая из команд в цепочке выполняет одну из вышеназванных функций. При возрастании интеллектуального уровня редактора текста возрастает и риск. Такой редактор может внести в исходный текст коренные изменения, которые могут оказаться либо нужными, либо нежелательными. И все же, несмотря на это, хороший редактор текста может стать лучшим другом и помощником сопровождающего программиста. Известно, что программисты питают самые теплые чувства к своим любимым редакторам.

К счастью, фирмы, торгующие вычислительными машинами, предлагают вместе с аппаратурой программное обеспечение. При этом редакторы текста пользуются большим спросом, особенно для работы в режиме разделения времени.

Преобразователь формата

Ввод	Обработка	Вывод
1. Текст	1. Изменить формат текста, чтобы улучшить его читаемость. Используется набор предварительно определенных правил улучшения читаемости	1. Текст, измененный с помощью правил читаемости

Пример: Изменения, внесенные в программу PAYROLL для исправления ошибок, выявленных в прошлом месяце, были поспешными. В результате ухудшилась читаемость программы. Применяется преобразователь формата, чтобы получить более четкий, удобный для чтения вариант исходной программы.

По мере того, как улучшается структура языков, возрастает потребность в тексте, удобном для чтения. Это утверждение следует подкрепить примером. Рассмотрим читаемость следующих двух фрагментов программ:

```
a) for I in 1.. FILE_TABLE SIZE loop
      if FILE_TABLE(I) = FILE_NAME then return I; end if:
      if FILE_TABLE(I) = NULL then
        FILE_TABLE(I): = FILE_NAME; return I; end if;
   end loop;
b) for I in 1.. FILE_TABLE_SIZE loop if
   FILE_TABLE(I) = FILE_NAME then return I;
   end if; if FILE_TABLE(I) = NULL then
   FILE_TABLE(I): = FILE_NAME; return I;
   end if; end loop;
```

Содержание обоих текстов идентично, однако формат первого фрагмента читается легко, а второго — нет.

Большинство программистов придерживается первого способа написания программы. Однако для тех, кто пишет программы по второму способу, для тех, кто работает с этими программами, и в особенности для тех, кто работает в режиме разделения времени, весьма желательно иметь автоматическое устройство, облегчающее чтение текстов «b». Таким устройством и является преобразователь формата. Очевидно, что он должен располагать набором правил, определяющих, что и как следует преобразовывать.

Структурные схемы и языки проектирования программ

Структурные схемы являются традиционным способом представления проектируемой программы. С появлением мощных языков высокого уровня структурные схемы были вытеснены из обращения и применяются в качестве черновой записи при разработке программы для уяснения ее структуры. (Такая структурная схема может быть использована для дополнительной проверки надежности программы.) Во всех остальных случаях, где требуется документирование программ, структурные схемы бесполезны (кроме случая, когда она физически соответствует исходному листингу). Большинство сопровождающих программистов не пользуются структурными схемами, так как последние быстро устаревают. Всю необходимую информацию они берут из листинга и постоянно ищут новые, более удобные способы представления разрабатываемых программ. В работе [15] этот процесс эволюции рассматривается с юмором. Особенно многообещающим с этой точки зрения является использование языка проектирования программы (ЯПП). Применение ЯПП в программах в виде комментариев более эффективно по сравнению со структурными схемами и позволяет сопровождающему программисту быстро разобраться в ее сути. Как уже говорилось в разделе, где описаны языки программирования, ЯПП должен быть совместим с языком программы с точки зрения соответствия правилам комментирования.

Описание этапа проектирования программ заносится в память ЭВМ. Во-первых, это гарантирует, что описание не будет потеряно (часто оно теряется). Во-вторых, сопровождающий программист будет иметь под рукой необходимые сведения о всех стадиях жизненного цикла программного обеспечения для анализа требований, а также для дальнейшего совершенствования и реализации программ.

Часто работа средств, используемых для удобного представления программ, основана на применении одной из приведенных выше методик. Если возникла необходимость получить структурную схему программы, такое средство извлекает информацию о строении программы и выдает ее структурную схему в графическом виде. (Применение таких средств часто полезно лишь в том случае, когда комментарии к структурной схеме совпадают с комментариями к программе, что помогает описать ход ее работы.) Средства ЯПП считывают язык проектирования программы, подобно тому как компилятор считывает текст программы, и выдают на выходе: 1) результаты проверки ЯПП (нет ли лишних или недостающих элементов проектирования?), 2) таблицы перекрестных ссылок ЯПП и 3) листинг текста программы. Средство ЯПП является сравнительно новым достижением в наборе средств программиста.

Ввод	Обработка	Вывод
1. Представление текста программы в форме, читаемой вычислительной машиной: а) структурная схема — текст программы, дополненный комментариями. б) ЯПП-текст, имеющий вид программы	1. Структурная схема — подготовить графические представления программы 2. ЯПП-проверка состава языка проектирования программы	1. Структурная схема — вариант графического представления входного варианта программы 2. ЯПП-проект программы с преобразованным форматом и с результатами проверки

Примеры: а) Ряд преобразований привел к тому, что структурная схема программы PAYROLL устарела. Необходимо создать новую и верную структурную схему. б) Модификация BETA в программе, написанной на языке Ада, ее не корректирует. Становится ясно, что при проектировании программы допущена ошибка. Для того чтобы получить листинг проектируемой программы и таблицы перекрестных ссылок на BETA, используется ЯПП.

Средства верификации

Мы уже говорили о том, что мир сопровождения, как в капле воды, отражает в себе мир создания программного обеспечения. Все задачи разработчика становятся и задачами сопровождающего программиста. Верификация программного обеспечения не является

исключением. Сопровождающий программист крайне заинтересован в создании надежного обеспечения.

В последние годы появилось большое количество средств и методов проверки. Они подробно описываются и обсуждаются в «Руководстве по сопровождению программного обеспечения»; там же есть ссылки на книгу Гласса (Glass, Software Reliability Guidebook, Prentice-Hall, 1979), в которой дается исчерпывающий анализ всех средств верификации. Поэтому в данной книге мы не будем повторяться.

Другими источниками, вышедшими за последнее время и имеющими отношение к средствам верификации, являются следующие [6, 10, 12, 19, 20, 29 — 33, 37, 38, 53].

Ввод	Обработка	Вывод
1. Программа нуждается в проверке 2. Варианты тестов или критерии	1. Выполнение программы с использованием различных вариантов тестов или применением разных критериев	1. Результаты выполнения программы 2. Неправильные результаты 3. Анализ возникновения ошибочных результатов

Пример: Требуется узнать, учитывалась ли когда-нибудь в программе PAYROLL (ЗАРПЛАТА) разница в начале рабочего дня каждого сотрудника. Тестовый анализатор использует для получения результата специальную версию программы PAYROLL, которая позволяет подсчитать общее время работы по частям. Затем прогоняются все варианты тестов программы. В результате становится ясно, что разница в начале работы в программе не учитывалась.

Управление версиями

Создатель программного обеспечения системы представляет себе свое детище как сверкающий монолит, совершенный и непогрешимый. Совсем иначе представляется эта система специалисту по сопровождению, которому приходится непосредственно иметь с ней дело. Любая система программного обеспечения существует в виде серии версий. Версии появляются из-за того, что всякую систему необходимо изменять. Вызваны ли эти изменения исправлением ошибок или изменившимися техническими требованиями, они неизменно повлекут за собой новые версии.

Возникает вопрос: как уследить за этими версиями? Ответ на него непрост, потому что каждая версия состоит из большого числа модулей и каждый из модулей либо изменяется, либо остается неизменным при переходе от одной версии к другой. К тому же каждый модуль существует как минимум в двух формах — исходной и объектной; исходный модуль может меняться, в то время как

объектный модуль остается неизменным (например, когда изменяется основная база данных, внешняя справка или файл копирования).

Специалисты только начинают искать автоматические средства решения проблемы. Создана специальная дисциплина, называемая конфигурационным управлением, которая все еще находится в состоянии зарождения (разд. 4.2.4). Но уже появляется система автоматического конфигурационного управления, которую иногда называют системой управления версиями [7].

Ввод	Обработка	Вывод
1. Модули, входящие в состав системы	1. Выбрать те модули, которые составляют определенную версию, и создать ее	1. Версия системы, состоящая из набора взаимно координированных модулей
	2. Проследить взаимоотношения между модулями — родственными модулями и их производными	2. Распечатанный список межмодульных отношений
	3. Осуществить управляющий доступ к системе с помощью автоматизированного набора слов	

Пример: Модули создаваемой версии программы PAYROLL отличаются от модулей текущей версии, которая в свою очередь отличается от предыдущей версии. Предыдущая версия содержала устаревшие вычисления вычетов системы социального страхования. Надо разработать систему, в которой бы исключалось применение как экспериментальных, так и устаревших модулей.

Управление версиями должно прослеживать возможные изменения на двух уровнях: 1) на уровне модулей (какие модули взаимодействуют между собой, какие являются головными, а какие — производными) и 2) на уровне системы (какие модули следует собрать и объединить в специальную версию).

Проблема управления версиями есть частный случай более общей проблемы базы данных. Каждый модуль должен быть соединен с соседними модулями двусторонней связью, с тем чтобы было ясно, из чего состоит система, и к какой системе относится данный модуль.

Системе управления версиями может понадобиться механизм системной защиты. Обычно для большой системы программного обеспечения изменения могут вноситься лишь при определенных обстоятельствах (например, в случае одобрения специальной комиссией). Поэтому существует потребность в создании как относительно бесконтрольных экспериментальных версий (для целей проверки), так и строго контролируемых версий для производственных систем. Это наводит на мысль о необходимости выхода за

рамки средств защиты, таких, как, например, пароли и механизмы блокировки.

Проблема усложняется, если над одной системой одновременно работают несколько сопровождающих программистов. Могут ли они одновременно создавать перекрещивающиеся экспериментальные версии? Как это сделать, если они вынуждены работать отдельно друг от друга?

Круг вопросов, которые должны решаться таким средством, как управление версиями, ясен. Однако разнообразие и сложность этих вопросов приводят к тому, что число общих решений минимально. Здесь, как и в других областях сопровождения программного обеспечения, остро ощущается потребность в разработке и внедрении передовых методологий.

Суперкомпилятор

В предыдущих разделах говорилось о различных идеях совершенствования доступных сопровождающему программисту средств. В этом разделе попытаемся объединить все средства в одну систему, которую назовем *суперкомпилятором*. Насколько нам известно, такой системы пока еще нет. Существуют отдельные ее компоненты в виде трудносовместимых между собой программных средств, назначение которых до конца неясно; но некоторых частей пока нет вовсе. Возможно, суперкомпилятор будет реализован в будущем, когда сопровождению начнут уделять должное внимание.

Чтобы составить представление о том, что такое суперкомпилятор, мы сначала должны определить базу, на которой он существует. Она состоит из вычислительной машины и множества работающих стандартных подпрограмм, которые все вместе называются операционной системой. Операционная система — это *минимальное* множество стандартных подпрограмм, необходимых для того, чтобы соединить физические возможности вычислительной машины с логическими возможностями, присущими программам или задачам, которые решаются на ЭВМ. Значит, суперкомпилятор будет управлять всеми программами в рамках системы программного обеспечения. Он «знает» каждую часть системы. В него заложены данные о всех множествах, которые действуют и над которыми совершаются действия. Он создает модули исходных программ, которые являются выражениями всех функций в форме, читаемой человеком; он «знает» базу данных, построенную на этих модулях; он «понимает» взаимоотношения между исходными модулями и их объектными модулями, которые представляют собой «несистематизированные» версии программ; он же обеспечивает объединение различных частей программ в более крупные, что мы называем систематизацией. Суперкомпилятор также допускает различные версии одних и тех же исходных /объектных/описательных модулей

(дискрипторов) и позволяет производить подстановки одних элементов вместо других. Он отвечает за работу модуля. В случае прерывания суперкомпилятор сообщает о состоянии системы или о состоянии модуля в символической форме (читаемой человеком) в терминах исходной программы. О сбоях суперкомпилятор сообщает с помощью переменных, принадлежащих модулям и подпрограммам, в которых сбои были замечены. Он не указывает адрес, на котором произошел сбой, а сообщает, например, что программа XYZ дала сбой при обработке оператора 27 в процедуре ABRACADABRA, так как переменная ABC была использована в качестве индекса со значением 19, превышающим допустимую величину индекса массива SMALL ARRAY.

Кроме того, поскольку суперкомпилятор располагает сведениями о всех частях системы, он может информировать модули, которые обращаются к данным, о том, какие значения переменных уже были использованы, какие модули используют эти же переменные и т. д. Создание суперкомпилятора предполагает претворение в жизнь тех идей, о которых было сказано раньше в этой книге. К ним относятся средства, формирующие таблицы перекрестных ссылок, средства отладки, более сложные средства редактирования текстов, а также преобразователи форматов и средства верификации, о которых говорится в разд. 3.2.1.1. Суперкомпилятор способен превзойти любые наши ожидания, так как в процессе анализа программ, составляющих программное обеспечение, он собирает самую разнообразную информацию.

Ввод	Обработка	Вывод
1. Система программного обеспечения в целом	1. Скомпилировать и соединить отдельные части системы 2. Проанализировать перекрестные связи в системе 3. Произвести символическую отладку 4. Оптимизировать систему с точки зрения размера и/или времени выполнения.	1. Скомпилированный и отредактированный текст в системно-оптимизированной форме 2. Информация о связях системы 3. Символическая отладка, облегчающая выполнение программы 4. Текст программы

Пример: Необходимо перестроить программное обеспечение системы ABRACADABRA, написанное на языке Ада. Для получения объектного кода вместе с результатами символической отладки, которые облегчают проверку системы в диалоговом режиме, и с результатами проверки, которые позволяют проанализировать ее возможности, используется суперкомпилятор. Одновременно получается полный список всех перекрестных ссылок системы.

Суперкомпилятор должен иметь:

1. Возможность объединять результаты компиляции для большого числа программ. Данные, являющиеся общими для всех программ, могут в дальнейшем анализироваться с точки зрения их влияния на систему.

2. Возможность получать целые разделы программы (или системы) в символической форме (на ЯВУ), используя дамп (избирательного вывода) и модули описания.

3. Возможность оптимизировать не только отдельные программы, но и всю систему.

4. Возможность формировать системные таблицы перекрестных ссылок.

5. Возможность создать матрицу, или древовидную структуру, или сеть, описывающую структуру всей системы.

Как мог бы работать суперкомпилятор. Структура суперкомпилятора на данном этапе вполне предсказуема. Интерфейс операционной системы централизует все запросы компилятора к ресурсам и, если необходимо, управляет оверлейной структурой. На последующих этапах суперкомпиляции, часто перекрывающих друг друга, выполняются следующие функции:

1. Обработка исходного текста — программа на исходном языке считывается и подвергается грамматическому разбору, создается таблица символов, в которой хранится вся информация, касающаяся имен, и файл на промежуточном языке.

2. Оптимизация — текст на промежуточном языке просматривается с целью устранения лишних операций. В результате получается оптимизированная программа на промежуточном языке.

3. Генерация кода — исправленная программа на промежуточном языке преобразуется в объектный модуль для данной вычислительной машины, причем производится локальная оптимизация, например распределение регистров.

До настоящего времени считалось, что результат работы компилятора — это только оптимизированная объектная программа, дополненная соответствующими листингами, и данные диагностики. Сейчас становится ясно, что существует возможность получить и другие дополнительные данные.

Мы уже упоминали о различных типах таблиц перекрестных ссылок. Они могут быть получены без каких-либо изменений в структуре обычных компиляторов, описанных выше. Устройство, обрабатывающее исходный текст, располагает достаточной информацией, для того чтобы выполнить эту функцию. А вот листинг перекрестных ссылок системы требует нарушения устоявшихся правил, и компилятор должен сохранять таблицы символов всех программных модулей системы, чтобы проанализировать их вместе, когда программа уже скомпилирована.

Кроме того, сохранение таблиц символов может принести и

другую пользу, если станет реальной возможность отладки программ, написанных на ЯВУ. В этом случае таблицы символов должны быть доступны во время выполнения программы, чтобы часть суперкомпилятора, отвечающая за отладку, могла получить необходимую информацию в символьной форме и сообщить программисту о возникающих проблемах. Таким образом, мы определили уже две причины, по которым таблицы символов компилятора должны являться его стандартным выходом.

В действительности, кроме сведений такого рода, суперкомпилятор будет давать на выходе информацию, которую назовем *дескриптивным модулем*. Дескриптивный модуль будет ответственным за дополнительную (описательную) информацию, необходимую для всех последующих этапов суперкомпиляции; с помощью дескриптивных таблиц символов можно будет легко создать программу, инициирующую базу данных, которая содержала бы информацию, необходимую для проверки достоверности и проверки содержания программы. При предварительном задании большой базы данных потребность в программах такого рода часто возникает в системах реального времени при необходимости модификации базы данных без перекомпиляции всей системы.

Кроме того, суперкомпилятор может обеспечить сопровождающих программистов такими средствами сопровождения, которые «порождают» программу для определенных целей. *Порождение* — это процесс спонтанной генерации программы, которая может выполнять некоторые дополнительные функции. Дополнение исходной программы соответствующим порождением приведет к тому, что она сможет, например, подсчитать время работы каждого сегмента ее логической структуры или проконтролировать утверждения, определенные программистом. Отметим, что средства, подобные порождению, легко включаются в традиционные компиляторы и поэтому являются кандидатами в функции суперкомпилятора.

Еще одним классом средств, реализуемых на базе суперкомпилятора, являются *препроцессоры*. В то время как вышеупомянутые средства действуют во время или после традиционных стадий компиляции, препроцессор действует еще до них, переводя предложения какого-нибудь другого языка в форму, обрабатываемую компилятором. Например, «язык очень высокого уровня», применяемый лишь в узкой прикладной области, либо «структурный язык», являющийся версией базового языка, либо «язык скорописи», представляющий собой упрощенную версию базового языка, могут быть переведены на базовый язык с помощью препроцессора, входящего в состав суперкомпилятора. (Предположим, что базовый язык имеет макросредства, для того чтобы можно было обобщить функции препроцессора.)

Теперь, когда мы убедились, что компилятор способен выполнять значительно более широкие функции, чем принято

считать, можно предположить, что он может послужить источником и других очень заманчивых идей. Пожалуй, лучше всего эти новые идеи суммированы в работе [51]. Недалеко то время, когда первый суперкомпилятор начнет свою работу. Не стоит удивляться, если мы узнаем, что первый суперкомпилятор используется для поддержки нового языка программирования Ада, разработанного Министерством обороны США [60].

Интересные соображения по поводу суперкомпиляторов можно найти в работе [58] , где обсуждается понятие «суперъязык».

Проблемы суперкомпилятора. Как уже говорилось, на современном уровне состояния программного обеспечения суперкомпиляторы не используются. Частично это объясняется тем, что никто пока не попытался объединить в одном устройстве такое множество функций. Но будь эта причина единственной, можно было бы найти серьезного разработчика, который взялся бы за создание суперкомпилятора. Однако существуют и другие проблемы. Чтобы все стало ясно, уместно будет обратиться к истории создания компиляторов.

Более 20 лет назад были сделаны первые попытки разработки суперкомпилятора ограниченного применения. Тогда был создан обобщенный промежуточный язык UNCOL (Universal Computer-Oriented Language) [48], облегчающий получение группы исходных компиляторов, связанных с группой генераторов кода посредством языка UNCOL. В этом случае любой язык, для которого был создан исходный компилятор, можно было бы использовать для работы на всех ЭВМ при наличии у них соответствующих генераторов кодов. Идея была блестящая, но она все еще ждет своего воплощения. Последние годы фирма JOCIT по внедрению средств компиляции, разрабатывающая С-компилятор, и, возможно, некоторые другие фирмы вплотную приблизились к решению этого вопроса. Проблема заключается в том, что мы не располагаем достаточными знаниями о промежуточных языках, чтобы создать промежуточный язык, совершенно независимый от исходных языков. При этом он должен быть независим от генераторов кодов, а также удобен для высококачественной оптимизации.

По этой же причине мы пока не можем создать суперкомпилятор. Несмотря на то что этой проблеме уделяется серьезное внимание, много вопросов, касающихся именно промежуточного языка, остаются нерешенными. Они требуют дальнейших исследований. Кроме того, необходимо провести работу по стандартизации. По крайней мере таблица символов, являющаяся частью вывода дескриптивного модуля, должна иметь стандартизированное внешнее представление (при этом допустимо, чтобы внутреннее представление было для каждого компилятора специфическим). Тогда таблица перекрестных ссылок всей системы и (или) ее отладочный пакет оказывали бы поддержку группе суперкомпиляторов, предназна-

ченных для различных языков, или сам суперкомпилятор мог бы обрабатывать больше одного языка.

Итак, будущее принадлежит суперкомпиляторам. Частично проблему можно решить уже сейчас. Часть ее требует дальнейшего изучения. Однако в любом случае решение принесет большую пользу сопровождающим программистам.

В следующем разделе речь пойдет еще об одном подходе к проблеме создания суперкомпилятора. В его рамках набор отдельно существующих мощных средств объединяется в «оборудование». При условии что будет уделено особое внимание отдельным средствам и их интерфейсам, «оборудование» может выполнять функции суперкомпилятора. Такие системы, как мы увидим, уже существуют, хотя их возможности значительно более ограничены, чем те идеи, которые высказываются в этом разделе.

Ввод	Обработка	Вывод
1. Все материалы, необходимые программисту для создания системы	1. Все возможности, необходимые программисту: а) средства документирования; б) средства манипулирования текстом; в) средства анализа результатов проверкч; г) средства конфигурационного управления	1. Готовая для работы и проверенная система программного обеспечения

Пример: Программист системы PAYROLL хочет получить окончательный (как он надеется) вариант программы со всеми исправленными ошибками и со всей новой документацией. Он вызывает редактор текста, генератор документации, компилятор, редактор связей, набор средств тестирования и анализа. При успешном завершении работы средств тестирования и анализа программы конфигурационного управления (с санкции программиста) замещают предыдущий вариант программы новым.

Рабочее место программиста

В настоящее время все больше убеждаются в том, что для успешной работы как разработчик программного обеспечения, так и сопровождающий программист нуждаются в специальных средствах и оборудовании. В результате было создано рабочее место программиста (РМП) [21], а в Министерстве обороны США разработано вспомогательное оборудование программиста на Ада (ВОПА) [51]. В рамках ВОПА в распоряжение программиста-разработчика отдаются вычислительная машина и ряд относительно стандартизиро-

ванных средств для работы с ней. С помощью этих средств возможно осуществлять:

1) разработку документации;
2) редактирование текста программ и вариантов тестов;
3) синтаксическую проверку программ;
4) анализ результатов тестирования;
5) конфигурационное управление программами и базами данных;
6) управление данными.

Более мощные средства обеспечивают также:

7) компиляцию, редактирование связей, выполнение и отладку программ;
8) обобщенное и регрессионное тестирование систем.

Обычно вспомогательные средства поддерживаются на ведущей ЭВМ, которая может быть или не быть той машиной, на которой будет работать система. Работающий сопровождающий программист выбирает ведущую ЭВМ в соответствии с требованиями к средствам. Зная заранее возможности выбранной ЭВМ и свои нужды, он может быть уверен, что независимо от назначения его программы она будет дополнена набором общих и эффективных средств.

В концепции рабочего места программиста (РМП) основной упор делается на стандартную мини-ЭВМ (которую предполагается при необходимости подключать к большой ведущей ЭВМ для выполнения, например, компиляции). Это особенно удобно при вычислительных центрах, где создаются системы для различных мощных машин, не все из которых отвечают потребностям пользователя.

В концепции ВОПА все средства содержатся на большой ЭВМ, включая сложные средства компиляции. Программы, полученные на ведущей ЭВМ, могут использоваться и на других машинах, в частности на микро- и мини-ЭВМ, предназначенных для решения частных задач.

Очевидно, что раздел, посвященный рабочему месту программиста, обобщает содержание предыдущих разделов. Здесь представлена совокупность всех средств, необходимых сопровождающему программисту.

3.2.1.2. Методы администрирования над сопровождением программного обеспечения

Одновременно с созданием системы программного обеспечения происходит создание и другой «системы» (имеется в виду набор средств), необходимой для того, чтобы разобраться в большом количестве бумаг, неизменно сопутствующих сопровождению программного обеспечения. Поэтому мы считаем необходимым создание полностью автоматизированной информационно-административной системы. Такие системы сейчас только начинают появляться. Частично об этом говорилось в выпусках по структурному

программированию из серии «Библиотека в помощь программисту» (БПП) [1, 29], а также в литературе, посвященной рабочему месту программиста.

С точки зрения контроля над изменениями эта система должна предоставлять:

1) возможность доступа к отчетам об ошибках;

2) возможность доступа к отчетам о внесенных в систему изменениях;

3) возможность анализа взаимного влияния возникающих ошибок и связанных с ними изменений в системе, а также описание исключений;

4) возможность докладывать о состоянии разработки (например, о предполагаемой дате окончания работ или о типах вносимых изменений);

5) возможность создания инструкции пользователю.

В отчете об изменениях содержится разнообразная информация, которая может быть полезна многим. Такая информация может быть представлена в виде листинга проблем по степени важности для пользователя, в виде данных о предполагаемом сроке окончания работ, в виде рекомендаций о возможных способах устранения неполадок. Такая информация может быть извлечена из базы данных отчетов об изменениях. Вопросы ведения документации и написания отчетов обсуждаются в разд. 3.1.2. Здесь достаточно будет сказать, что средства создания отчетов являются важной частью информационно-административной системы программиста. Например, взяв за основу файл отчетов об изменениях и дополнительную информацию, программист может подготовить отчет о положении дел, который явится исправленным и дополненным вариантом отчета о текущем состоянии всех проблем.

Отчет о состоянии изменений, кроме вышеупомянутых требований контроля над изменениями в системе, включает в себя точно определенный набор данных, которые можно собрать и вручную, что, впрочем, делается очень редко. Эти данные должны включать:

1) общее количество строк программы (конкретной) в системе;

2) количество строк, измененных за отчетный период;

3) взаимосвязь между модифицированными строками и вносимыми в систему изменениями;

4) имя автора изменения;

5) задания, полученные каждым сопровождающим программистом;

6) общее ядро, использованное системой;

7) нарушение модулем процедуры использования ядра.

Такие отчеты необходимы не только пользователям и руководству, но и сопровождающему программисту, так как ему важно знать, на какой стадии работы он находится и что ему еще предстоит. Сопровождающий программист нередко задает себе вопросы:

«Существуют ли ранее составленные отчеты по этой проблеме? Удалось ли мне решить задачу по-новому? Каким образом будут развиваться сделанные мною модификации?» Быстро ответить на эти вопросы помогут автоматизированные методы администрирования.

Проблема разработки информационно-административных систем в помощь сопровождающему программисту требует дальнейшего изучения.

3.2.2. ТЕХНИЧЕСКИЕ ПРИЕМЫ СОПРОВОЖДЕНИЯ

Сопровождение программного обеспечения все еще остается работой, требующей высокой квалификации. Она не имеет четкой грани между наукой и искусством. Как мы ни стараемся формализовать ее, пока не удается обойтись без учета личных качеств, которые должны быть присущи сопровождающему программисту (имеются в виду интуиция, догадка, «искусство» программирования).

Так будет до тех пор, пока на практике не воплотят научные (или полунаучные) системные средства, описанные в разд. 3.2.1. А до этого сопровождающий программист будет вынужден изучать дампы, самостоятельно разбираться в листингах и пытаться соотнести результаты проектирования программ с требованиями и спецификациями.

В следующем разделе речь пойдет о технических приемах сопровождения, которые должны оказать сопровождающему программисту помощь в его работе. Работа [12] поможет ему оценить, насколько какая-либо часть программного обеспечения будет поддаваться сопровождению.

3.2.2.1. Профилактика сопровождения

Лучшим сопровождением программного обеспечения является его отсутствие. Это означает, что не было допущено ни одной ошибки, а все возможные изменения были заранее предусмотрены.

Затем идет сопровождение, которое легко осуществляется благодаря проведенному этапу разработки программного обеспечения. Обе вышеупомянутые разновидности настолько хороши, что хочется написать о них еще раз. (Однако мы этого делать не будем, так что, пожалуйста, вернитесь назад и перечитайте начало раздела еще раз!)

Очень важно, создавая программное обеспечение, руководствоваться идеей замены сопровождения системы ее развитием. Каждый шаг проектирования и реализации должен содержать в качестве своей неотъемлемой части цель — облегчить работу «следующему» в общей цепи. «Следующим» может оказаться проектировщик, работающий с требованиями, или программист, реализующий проект,

или контролер, работающий с программой и спецификациями, или сопровождающий программист, замыкающий эту цепь, его работа зависит от всех предыдущих. И часто «следующим» может оказаться тот, кто работал на более раннем этапе. Таким образом, ключевым вопросом всего процесса разработки является: что нужно сделать для того, чтобы облегчить работу на последующих этапах?

Ответ на этот вопрос прост: руководство должно увеличить объем затрат на сопровождение (вспомним рисунки гл. 1). Пусть проектирование и реализация программного обеспечения обойдутся несколько дороже: это окупится за счет снижения стоимости сопровождения.

Все сказанное можно свести к простой и известной истине: делай все хорошо с первого раза! Однако здесь необходимо дать точное и исчерпывающее определение тому, что значит это «хорошо». Попытаемся это сделать, но прежде кое-что уточним.

Этот раздел мы назвали «профилактика сопровождения»[1]. Мы преднамеренно не воспользовались термином «профилактическое сопровождение», так как он может ввести в заблуждение. Дело в том, что если в других областях возможен такой род деятельности, который направлен на улучшение продукции уже после ее создания, то в случае с программным обеспечением это невозможно. Материальная база ЭВМ, например, периодически проверяется с помощью раз и навсегда установленного набора тестов, чтобы выяснить, все ли функции ЭВМ исправны. Программное же обеспечение не ломается и не изнашивается, поэтому проверять, не сломалось ли оно, бессмысленно. Если в обеспечении имеется ошибка, это значит, что она была там с самого начала[2]: она не может сама появиться или сама исчезнуть. Профилактика сопровождения — это работа, проводимая еще до того, как начнется жизненный цикл программного обеспечения. Профилактическое же сопровождение подразумевает работу, которая проводится уже во время сопровождения, чтобы сделать его более эффективным. Эта работа, возможно, тоже необходима, однако о ней речь пойдет в другом разделе нашей книги, где она будет названа иначе («регрессионное тестирование»).

Теперь мы можем обратиться к более подробному обсуждению понятия «профилактика сопровождения». В последующих разделах мы расскажем о технических приемах, используя которые сопровождающий программист может облегчить задачу тем, кто придет следом, а значит, и себе самому.

---

[1] Когда речь идет о качестве сопровождения, это понятие также называют «способностью поддаваться сопровождению» («maintainability») [4].

[2] Исключение составляют лишь ошибки, появившиеся при внесении новых изменений.

Модульность

Основным условием профилактики сопровождения является создание программы, состоящей из отдельных модулей. Это ни для кого не секрет. Все согласны с этим, и каждый убежден, что он использует модульный принцип написания программы, однако по непонятной причине у одних программистов получаются программы намного лучше, чем у других. Все верят в модульность, все стараются добиться ее. Остается только выяснить: что это такое?

Дать ответ на этот вопрос не просто. Конечно, в литературе [31, 39, 40] имеются прекрасные определения этого понятия. Однако остается вопрос: как реализовать все сказанное в конкретном случае? Решение полностью зависит от индивидуальности программиста и, следовательно, от его мастерства. Это наглядно показано в литературе [34].

В этом разделе мы попытаемся объяснить, что такое «хорошая модульность», как ее добиться и распознать, а затем перейдем к рассмотрению примеров. Хорошая модульность — это в некотором роде шестое чувство программиста, которое невозможно подменить ни рассуждениями, ни советами, ни указаниями.

Однако начнем с определения. Модульное программирование — это процесс реализации программного обеспечения в маленьких функционально-ориентированных блоках. Эти блоки называются модулями и обычно находят свое применение как подпрограммы или функции, или группы функций. (В некоторых языках их называют секциями или процедурами.) Каждый модуль предназначен для решения одной или нескольких функциональных задач: модуль может быть сопряжен с одним или несколькими местами в программном обеспечении системы.

(Предыдущий абзац сам является образцом модульности. Он целиком заимствован из другой книги [14] и, таким образом, выполняет единственную функцию — дает определение модульности. В результате он является модулем информации, который может быть вызван еще несколькими другими книгами. Его полезность как модуля объясняется тем, что он рассматривает ограниченную область знаний в обобщенном виде. Если бы этот модуль содержал материал, относящийся только к конкретному тексту в другой книге, то он здесь утратил бы свое значение.)

В современной литературе можно встретить и отрицательные высказывания о модульности. Существует направление (оно является частью течения «структурного программирования»), определяющее модуль по числу содержащихся в нем строк. Правила программирования нередко включают такие ограничения, как: «Программные модули должны состоять не более чем из X строк, где X чаще всего имеет значение от 50 до 100. Нам кажется, что определять модуль по числу строк неправильно, так как это уводит внимание программиста

от функционального назначения модуля, фиксируя его внимание на размере. Несомненно, что многие удачные модули малы. Но и среди больших модулей есть много удачных. Очевидно, что маленький модуль легче понять, но несложный большой модуль тоже понять легко, в то время как маленький, но запутанный модуль понять трудно. Критерием здесь должен быть не размер, а функциональная принадлежность.

Преимущество хорошей модульности становится очевидным особенно при сопровождении. Если необходимо изменить функцию модуля, надо изменить только этот модуль. Это почти не окажет влияния на работу остальной части программы. Если возникает необходимость в новой функции, то это можно осуществить созданием соответствующего модуля. Влияние на работу всей программы будет минимальным. (Практически проверка программы, построенной по модульному принципу, будет представлять собой последовательность вызовов существующих модулей. И если будет найдена ошибка, ее устранение пройдет в рамках одного или нескольких модулей. Кроме того, если модуль уже однажды проверен, его можно использовать в дальнейшем в других местах программы. Таким образом, внесенные изменения не только обходятся дешевле, но и повышают вероятность правильного выполнения программы.)

Пожалуй, лучшим примером модульности является библиотека стандартных подпрограмм. Каждая подпрограмма такой библиотеки выполняет определенную задачу или группу задач. Программисту не приходится самому программировать функцию квадратного корня или что-либо подобное: эта работа уже сделана, и ему остается лишь воспользоваться результатом. Стандартная подпрограмма является самым быстрым средством «затыкания дыр» в программном обеспечении.

Но модульность следует понимать шире, чем просто библиотеку стандартных подпрограмм. Модули — это составляющие элементы программного обеспечения, однако модуль не обязательно должен быть универсальным. Он может быть элементом, который необходимо использовать только в нескольких местах определенной программы. Наличие нужного модуля может сделать программу легкой для сопровождения, а его отсутствие, наоборот, такой трудной, что от нее придется отказаться. Например, соображения модульности при программировании обычно требуют, чтобы внешние интерфейсы, такие, как ввод/вывод, были выделены в отдельные модули. Несоблюдение этого требования намного усложняет работу сопровождающего программиста при внесении изменений в интерфейс.

Модули могут принимать различные формы. Часто внешняя («отдельно скомпилированная») подпрограмма считается модулем. В языках типа Алгола внутренняя процедура (определенная в рамках

большей программы) также может быть модулем. В Коболе и некоторых других языках модулем может быть секция или любой другой отдельный сегмент программы. А в наиболее современной форме база данных с ее кластером для манипулирования базами данных также может считаться модулем. Подчеркнем еще раз, что суть модуля состоит в присущей ему одному функции. Важное значение имеет связь модуля с данными. Наиболее распространенной формой такой связи является список параметров; в этом случае модулю известны только местные данные и формальные параметры. Эта форма связи хороша тем, что все данные, используемые модулем, идентифицируются и выделяются; при этом модуль не может оказать неожиданного побочного воздействия на вызывающую его программу. Это могло бы произойти, если бы модуль модифицировал глобальную переменную, величина которой была важна для той части программы, где произошел вызов модуля.

Однако идея полного использования списка параметров не всегда достижима. Модулю может понадобиться доступ к большему числу данных, чем определено в списке параметров в каждой точке вызова. В таком случае связь может осуществляться через глобальные данные, а именно либо через общую область данных, либо через общий пул данных (термин «общая область» употребляется в Фортране, а термин «общий пул» — в языке JOVIAL и его производных для централизации описания данных, необходимых для всех модулей, составляющих программу. Сам по себе блок — общая область, и общий пул становится отдельным модулем или несколькими модулями программы).

Учитывая эту потребность, в современные языки программирования часто включают понятие «кластер». База данных и семейство процедур для работы с ней отделяются от остальной программы. Программа, которой нужны некоторые или все кластеры, должна обращаться точно к ним. Кластер сам принимает решение о том, какие данные и процедуры можно использовать, а какие нет. Таким образом снимается вопрос о побочных воздействиях [41, 42].

Модули, которые связываются с данными только через список параметров, называются *полностью замкнутыми*. Заметим, что полностью замкнутый модуль легко может быть перенесен в другую программу, так же как библиотечная подпрограмма. Модули, которые связаны и с глобальными данными, называются *частично замкнутыми*. Частично замкнутые модули имеют такие данные, которые связывают их с телом программы и затрудняют перенос этих модулей в другие программы. Кластер, четко определяющий, какие его части могут быть заимствованы, и поддерживающий необходимую ему базу данных, сам является закрытым модулем.

Однако о модульности, как нам кажется, сказано достаточно. Пора перейти к примерам удачной и неудачной модульности.

Пример модульности. Предположим, что вы получили задание —

определить систему управления файлом, которая будет применяться для поддержки одной или более прикладных систем доступа к файлу. После начального периода проектирования вы приходите к выводу, что решение задачи четко распадается на несколько задач по обслуживанию пользователя. Вы должны уметь: 1) создать файл; 2) уничтожить файл; 3) записать файл; 4) считать с файла; 5) найти в файле; 6) добавить к файлу и 7) удалить из файла.

Для того чтобы выполнить эти задачи, необходимо иметь базу данных, в которой отражаются все изменения информации, содержащиеся в файле.

Для удовлетворения всем этим требованиям вы создаете модули, каждый из которых будет выполнять перечисленные задачи. Например, процедура CREATE потребует на входе имя файла, который нужно создать, и даст на выходе исправленную и дополненную базу данных, включающую новый файл. Процедура SEARCH потребует на входе имя файла, в котором следует вести поиск, и ключ к той информации, которая должна быть найдена, а на выходе даст местонахождение файла и (или) позицию искомой в нем информации.

Заметим, что каждая модуль-функция — это хорошо определенный круг задач, а значит, представляет собой достаточно простой участок программы. Таким образом, можно разбить сложную большую задачу на отдельные, легко решаемые задачи.

На языке программирования Ада определение этих модулей будет выглядеть следующим образом:

```
Procedure CREATE (FILE_NAME) is
begin
   -- выполнить все необходимое для создания файла
end CREATE;
function SEARCH  (FILE_NAME; KEY_NAME) return
   FILE_INDEX is begin
   -- выполнить идентификацию файла и поиск для
   -- данного ключа
   -- затем получить индекс искомого файла
end SEARCH;
```

По мере написания функций становится ясно, что каждая из них должна обращаться к базе данных. Значит, мы имеем дело с кластер-ситуацией. База данных со своими подпрограммами образует отдельную группу информации со свойственным только ей одной и хорошо определенным набором связей.

На языке Ада определение кластера, который в Ада называется пакетным модулем, будет выглядеть примерно так:

```
package FILE_MANAGER is
   -- определение сервисных интерфейсов пользователя
```

```
    procedure CREATE (FILE_NAME);
    — — другие функции управления файлом
    function SEARCH(FILE_NAME; KEY_NAME) return
      FILE_INDEX;
  end FILE_MANAGER;
  package body FILE_MANAGER is
    — — определение реализации сервисных средств пользователя
    — — но пользователь не может видеть, как они реализованы
    — — объявление базы данных
  procedure CREATE (FILE_NAME) is
  begin
    — — программа, описанная ранее
  end CREATE;
    — — аналогично для всех других реализаций
  function SEARCH (FILE_NAME; KEY_NAME) return
    FILE_INDEX is
  begin
    — — программа, описанная ранее
  end SEARCH
  end FILE_MANAGER;
```

Здесь следует отметить два момента: 1) пакет на языке Ада, или кластер, распадается на две части: внешний интерфейс и реализация (что является само по себе хорошим примером модульности) и 2) модульность в нашем примере становится иерархической, т. е. некоторые модули включают в себя другие модули. Это наилучшее проявление модульности: каждая функция четко отделена от всех других функций, модульные взаимосвязи хорошо определены и, исключая эти связи, модули полностью изолированы друг от друга.

Теперь приведем пример, когда та же задача плохо разбита на модули. Рассмотрим несколько неудачных, но возможных вариантов:

1. Вся система управления файлом может быть встроена в прикладную программу, которая ее использует.

2. База данных может быть отделена от прикладной программы, но функции начинают выполняться без предварительной обработки информации.

3. Функциональные обслуживающие сервисные процедуры (создать, уничтожить и т. д.) могут иметь общие части, вместо того чтобы быть оформленными как различные процедуры.

Отрицательных примеров можно привести бесчисленное множество.

Структура программы

В печати за последнее десятилетие широко освещалась проблема структурного программирования. Специалисты и неспециалисты в

один голос провозглашали структурное программирование величайшим открытием с момента создания цифровых вычислительных машин. Однако это не так.

Такое категоричное заявление, конечно, потребует объяснений. Прежде всего само понятие «структурное программирование» еще недостаточно четко определено. Хотя существуют попытки дать определение [1, 29, 59], но они не всегда последовательны и совместимы. Однако суть этого понятия довольно известна. Оно включает практику программирования с применением ограниченного, но достаточного набора элементов управления, о которых мы не будем здесь говорить, так как о них столько написано, что читателям они, несомненно, хорошо известны. В понятие структурного программирования включают также множество побочных соображений, таких, как применение бригад программистов, проектирования сверху вниз и библиотеки поддержки программ. Но суть остается прежней и заключается в применении того набора элементов управления, о которых вы уже читали и слышали раньше.

Итак, что же плохого в структурном программировании? Ничего, кроме излишней рекламы, создаваемой вокруг него. Структурное программирование не является панацеей. Оно является очередным, более прогрессивным этапом развития программирования и в этом смысле достойно применения. Если расположить элементы сопровождения по степени важности, на первом месте будет стоять модульность, а почти сразу же за ней — структура программы.

В контексте нашей книги под структурой программы мы понимаем один из атрибутов программы, который позволяет лучше ее понять. Применение элементов структурного программирования делает программу легко читаемой и удовлетворяющей эстетическим требованиям. Операторы begin-end делят программу на законченные по мысли модули, блоки, предложения, что позволяет следить за логикой изложения, а значит, легче понимать ее. Отступы позволяют зрительно установить структуру программы. Это очень полезно, особенно для сложных программ. Но, пожалуй, еще важнее для структуры программы с точки зрения сопровождения последовательная система комментариев. Комментарии поясняют введение каждого нового элемента модульности и (или) структуры программы и тем самым улучшают ее форму и делают более удобной для чтения. О пользе комментариев мы скажем ниже.

Читателю может показаться неясным различие между модульностью и структурой. Разве модульность не есть подход к структуре? Несомненно. Эти два понятия рассматриваются отдельно по следующим причинам. Исторически сложилось так, что модульность появилась лет на десять раньше структурного программирования. Мы считаем необходимым выделить модульность, чтобы ее не смешивали с нечетко определенным понятием «структурное про-

граммирование». Такое разделение необходимо, так как в некоторых языках понятие «структура» относительно лишено смысла, в то время как понятие «модульность» существует во всех языках. Таким образом, граница между модульностью и структурой может показаться неявной, но она существует. Заметим, что хороший модуль не обязательно имеет хорошую структуру, а у плохого, наоборот, может быть хорошая структура. Значит, понятия «модуль» и «структура» следует различать.

Где-то на границе между понятием «модуль» и «структура» лежит понятие «межмодульный интерфейс». Сложные интерфейсы делают программу плохо понимаемой и неудовлетворяющей эстетическим требованиям; поэтому мы включаем в раздел и эту тему. Четкие интерфейсы — это еще элемент структуры программы.

Здесь также следует привести примеры. В предлагаемом отрезке программы обратите внимание на атрибут структуры программы — элементы управления, группирование операторов begin-end, отступы, комментарии, интерфейсы, — чтобы иметь возможность судить о качестве программы.

Пример структуры программы. В примере модульности мы рассматривали систему управления файлом. Функция SEARCH иллюстрирует одновременно понятие «хорошая структура программы» и взаимосвязь между «структурой» и «модульностью». Наш пример является отрезком программы, написанным на языке Ада.

```
function SEARCH (FILE_NAME; KEY_TYPE) return
FILE_INDEX is
begin
   — — поиск определяемого ключа записи в определяемом файле
   — — возврат индекса к прежнему значению
         for I in FILE_INDEX'FIRST..FILE_INDEX'LAST loop
         — — поиск всей базы данных
            if FILE_MAP(I) / = NULL
            — — проверка базы данных на нули и совпадения
                  элементов
                  and then FILE_MAP(I) = FILE_NAME
                  and then FILE_KEY(I) = KEY_TYPE then
                  return I;
            end if;
         end loop;
         raise BAD_FILE;
         — — расширить список исключении, если требуемая за-
                  пись не найдена
end SEARCH;
```

Данный пример иллюстрирует два элемента управления — цикл (for...loop...end loop) и условный переход (if... and then...then... ...end if). Здесь также показано несколько групп (begin-end SEARCH, loop-end loop и if-end if), четыре уровня отступов и небольшие комментарии. (Интерфейсы были показаны раньше, когда речь шла о модульности.)

Попытайтесь представить себе, что в этой программе отсутствуют отступы и комментарии. Теперь поставьте себя на место тех, кто обязан будет обеспечить сопровождение этой трудночитаемой программы. Впечатление от плохой структуры программы удручающее, не правда ли?

Структура данных

В литературе по вычислительной технике редко рассматривается экономическая сторона структуры данных и структуры программы. А между тем для многих программ одно хорошее описание структуры данных важнее 100 строк управляющей программы. И, что очень важно с точки зрения сопровождения, изменение одной структуры данных может заменить изменение 100 строк, разбросанных в управляющей программе.

Приведем очередной пример. В некоторых языках программирования имеется по крайней мере два способа работы с последовательностью битов или символов. Первый способ состоит в описании последовательности как части структуры:

```
STRUCTURE_DATA STRUCTURE; BEGIN
  ITEM CHARACTER_STRING 5 CHARACTERS;
  ITEM OTHER_VARIABLE 32 BITS;
END STRUCTURE;
```

а затем управлять последовательностью с помощью имени:

```
OTHER_STRING = CHARACTER_STRING;
```

Второй способ включает использование байтуправляющей функции для доступа к нужной последовательности:

```
OTHER_STRING = BYTE (CHARACTER_STRING,5);
```

Предположим теперь, что в программе происходит стократное обращение к последовательности символов, а число символов в последовательности необходимо увеличить с пяти до девяти. Как это повлияет на сопровождение? Очевидно, что в первом случае для этого надо изменить только одну легкообнаружимую строку программы, в то время как второй случай предполагает изменение уже 100 строк, которые к тому же не так легко найти. Значит, влия-

ние структуры данных на сопровождение может быть очень существенным.

(Приведенный пример выбран не случайно. Читатели помнят о вышеупомянутых трудностях, вызванных переведением Почтового ведомства США плана почтовых зон на более длинные программы. Совершенно ясно, что, если бы в прошлом уделялось большее внимание структуре данных, этот переход прошел бы почти безболезненно.)

Из данного примера видно, что структура данных важна сама по себе и имеет влияние на выполнение программы. Этот пример можно считать обобщенным: языки Кобол и ПЛ/1, например, обладают наилучшей структурой данных, так как имеют тонкую, ориентированную на редактирование структуру данных, наиболее удобную для выведения их на печать в нужном формате. (Простая команда пересылки, к примеру, может в результате устранить ведущие нули, вставить десятичную точку, добавить контрольные символы и поместить знак доллара в случае, если цель пересылки соответствует описанию.) А современные формы Фортрана и Бейсика, напротив, не представляют никакой структуры данных, кроме массива. (Это, пожалуй, самый серьезный недостаток Фортрана.)

Дихотомия структуры данных в отличие от дихотомии структуры программы, как мы уже отмечали, принимает различные формы. Она затрагивает даже область проектирования программного обеспечения. Одна из недавно появившихся методик, которую называют методикой Джексона [22], предлагает проектировать структуру программы, исходя из структуры данных. По Джексону, структура алгоритма тесно связана со структурой данных, и программы, спроектированные с учетом этого, будут наиболее удобными для сопровождения. Очевидно, что в разных областях применения потребность в структурной ориентации данных может быть различной. Не менее очевидно и то, что в тех областях, где структурная ориентация необходима, потребность в ней особенно велика.

Одной из наиболее характерных черт большой системы является алгоритмическое преобразование одной структуры данных в другую. Этот процесс иногда проистекает из так называемого «конфликта структур» (термин Джексона), а иногда является следствием неопределенного разрастания системы или погрешности проектирования. Во многих больших системах, претерпевших значительное число доработок, данные подчас с небольшими изменениями преобразуются по нескольку раз, пока наконец не получается именно то, что соответствует текущим требованиям. Чтобы остановить развитие этого процесса, необходимо заново проанализировать задачу. При проектировании промежуточных структур данных следует помнить о конечной цели использования информации. Программы, имеющие структурированные данные, требуют в

иерархии профилактики сопровождения максимального внимания к их структуре.

Не менее важно правильно группировать данные. В случае, когда описания набора данных взаимосвязаны функционально или по содержанию, для простоты восприятия и модификации их следует объединять в группы. Например, данные, называемые в Фортране «общими», а в JOVIAL — «общим пулом», следует объединить в один большой общий блок или общий пул-блок. Если потребуется изменение, его можно будет внести только в этот блок и провести повторную компиляцию только тех модулей, которые используют этот блок.

Концепция типизации абстрактных данных, так же как и метод структурирования данных, требует повышенного внимания. Идея этого понятия состоит в том, что все данные должны быть описаны как принадлежащие к определенному типу; типы определены в языке, кроме того, есть типы, дополнительно определенные программистом. Для каждого типа существует набор допустимых величин, значения которых могут принимать данные этого типа, а также набор операций, которые можно проводить с этим типом данных.

Разделение на типы имеет следующие преимущества: 1) свойства типа определены в его описании, и если они изменяются, то коррекции будет подвергаться только это описание; 2) справочные данные должны содержать только абстрактные свойства типа, а подробности реализации будут содержаться в описании; 3) строгое соответствие между типами может быть обеспечено компилятором, что исключает ошибки, вызванные их несовместимостью, облегчает сопровождение и повышает надежность.

Однако возможность полного разделения данных на абстрактные типы обеспечивают только самые современные языки, например Ада и Паскаль. Традиционные Кобол, Фортран и Алгол этого не позволяют делать. Причем в тех языках, где эта возможность отсутствует, действенного способа обеспечить ее не существует.

И наконец, важно эффективно именовать данные. Конечно, значимые имена здесь более уместны, нежели произвольные. Как вы думаете, что означает имя EMPLOYEE_NAME? А EMPNAM?

Здесь, однако, содержится противоречие. Иногда современные стандарты программного обеспечения требуют, чтобы данные были поименованы не в соответствии с их содержанием, а в соответствии с их структурой. Например, элемент данных, который определен в модуле XYZ и является частью таблицы TAB, можно было бы назвать XYZ_TAB_1. Это дало бы определенное удобство: сопровождающему программисту было бы ясно, откуда взяты данные. Однако мы считаем, что это затрудняет чтение программ. В данном случае возможен компромисс, например EMPLOYEE_NAME_XYZ_TAB включает как структуру, так и значение. Правда, такое наименование может оказаться для

программиста слишком длинным. (Здесь может помочь хороший редактор текста. Программист мог бы кодировать короткие имена, а затем с помощью команд для каждого имени производить подстановку более длинных имен.) Часто легкочитаемые (длинные) имена употребляются для глобальных данных, а короткие — для менее важных локальных данных. Таким образом, вопрос удобства наименования часто связан с вопросом его длины.

Пример структуры данных. Предположим, что в рассмотренном ранее примере управления файлами каждый файл имеет свой особый формат. Все записи в файлах имеют (возможно) один и тот же размер, но разные форматы, и, как это обычно бывает, они разнородны по составу, т. е. содержат всевозможные типы данных: последовательность знаков, целые величины, данные с плавающей точкой и т. д.

Такого рода структуры легко представить в одних языках и невозможно — в других. Ниже мы даем пример структуры данных на языке Ада. Заметим, что такой тип структуры, называемый Ада TYPE, используется в языке Ада для создания нескольких образцов этой структуры.

```
type RECORD_FORMAT_1 is
  record
    EMPLOYEE_NAME: array (1..30) of CHARACTER;
    EMPLOYEE_RATE: float;
    EMPLOYEE_HOURS: integer range 0..99;
  end record;
FILE1-RECORD· RECORD_FORMAT_1;
```

Формат записи для файла 1 определяется подобно типу записи _format_1; а он в свою очередь состоит из 30 знаков, числа с плавающей точкой и двухразрядного целого числа. Заметим, что каждое имя выбирают таким образом, чтобы оно давало ясное представление о содержании данных (это свойственно для языка Ада, так как в нем даже тип имеет описательное имя в соответствии с данными).

Понятие «группирование данных», возможно, лучше всего пояснить с помощью концепции кластера, о котором говорилось выше. Предположим, что программа управления файлом содержит таблицу описания файла, его текущий индекс, а также таблицу связи файлов с их физическим местоположением. Очевидно, что эта и вся остальная информация о базе данных файла должны быть сгруппированы вместе, так как они тесно взаимосвязаны между собой. Если в процессе сопровождения системы изменяются какие-либо данные, следует пересмотреть все сгруппированные данные с точки зрения влияния на них этого изменения; и еще более важно, чтобы это изменение не повлияло на данные других группировок.

База данных, управляющая файлом, может быть сгруппирована с дополнительными функциями, которые мы обсуждали ранее (например, CREATE и SEARCH). Это дало бы полное описание содержания базы данных и функции управления файлом.

Описание этой группы данных на языке Ада будет иметь вид

```
package FILE_MANAGER is
  — — здесь, как и раньше, определен весь сервис пользователя
end FILE_MANAGER;
package body FILE_MANAGER is
  type FILE_DESCRIPTION_TABLE_FORMAT is
    record
      FILE_NAME: array (1..10) of CHARACTER;
      FILE_KIND: integer range 0..4;
    end record;
type FILE_LOCATION is
    record
      FILE_NAME: array (1..10) of CHARACTER;
      FILE_POINTER: integer range 0. .32767;
   end record;
FILE_TABLE: FILE_DESCRIPTION_TABLE_FORMAT;
FILE_INDEX: integer rang O..NUMBER_OF_FILES;
FILE_LOC: FILE_LOCATION;
— — далее, как и раньше, следует реализация сервиса пользо-
    вателя
end FILE_MANAGER:
```

После этого структура данных группируется с другой программой, которой она требуется.

Язык высокого уровня

Мысль о том, что язык высокого уровня лежит в основе решения любой задачи программного обеспечения, почти превратилась в аксиому. Сопровождение программного обеспечения, в том числе и профилактика сопровождения, не является исключением. Что касается вопроса структуры программ, то он так хорошо освещен в литературе, что не имеет смысла повторяться [25, 59]. Говоря о перспективах данного вопроса, интересно отметить, каким образом можно поддержать все имеющиеся методы профилактики сопровождения с помощью языков высокого уровня (ЯВУ).

Структура данных, описанная в предыдущем разделе, совершенно невозможна без поддержки хорошим ЯВУ (использование массивов для моделирования древовидных структур и упаковка данных без ее описания крайне неудобны). Структура программы в меньшей степени зависит от ЯВУ, но и она на Ассемблере (даже при использовании макрокоманд) в лучшем случае неуклюжа. А вот

модульность в равной степени возможна и в ЯВУ, и в Ассемблере. Большинство программ, подвергаемых анализу с точки зрения профилактики сопровождения, не зависит от ЯВУ, в то время как документация выигрывает во многом, так как программы на ЯВУ легко читаются.

Надо также отметить, что не все ЯВУ одинаковы. Фортран в том виде, в котором он применялся до 1977 г., по числу пользователей является вторым языком после Кобола. Однако для профилактики сопровождения в качестве ЯВУ он плох. Фортран предоставляет возможность составления модулей, но ограничен с точки зрения структуры программы, структуры данных или параметризации (о которой речь пойдет ниже). В языках, созданных на базе Алгола, эти недостатки устранены. Кобол при всех достоинствах его структуры данных имеет ограниченные возможности для модульности структуры программы. Тот факт, что Кобол и Алгол по-прежнему широко применяются в наши дни, указывает на пренебрежение интересами сопровождения при выборе языка.

Примеры программ, написанных на ЯВУ, можно найти в справочниках по языкам программирования. Примеры самых лучших ЯВУ можно найти в литературе [24, 27].

Параметризация

Мелочи иногда имеют большое значение. К таким мелочам относится параметризация. Это понятие достаточно простое. Если для обозначения одного и того же множества в нескольких местах программы употребляется константа и ее, возможно, применят еще не раз, ей присваивают имя и употребляют это имя всякий раз, когда ссылаются на эту константу. Это может иметь значение только в том случае, если константа изменится. Тогда необходимо будет изменить только определение имени, не затрагивая ссылок. (Отметим сходство этого приема с подобным же приемом в структуре данных. Обобщая, можно сказать, что любой прием, позволяющий объединять переменные множества в декларации, является полезным для профилактики сопровождения.)

Не все языки предоставляют такую возможность в равной степени. Как ни странно, чаще она имеется в языках типа Ассемблера, а не в ЯВУ. Например, в ассемблерных языках почти всегда имеется макровозможность «equals» или «synonymous» (ARRAY_SIZE_EQU_10), но в языках высокого уровня она встречается редко. Например, язык военно-воздушных сил США JOVIAL предоставляет ограниченную макровозможность DEFINE: DEFINE ARRAY'SIZE «10». Как видим, и в данном случае современный язык не учитывает интересы сопровождения.

Так же совершенно недостаточна способность языка поддерживать динамические константы (которые могут быть инициированы с

помощью оператора присвоения или какого-либо другого способа). Параметризацию нельзя считать полной, если во время компиляции величина поименованной постоянной неизвестна и не используется. Заметим, что динамическая константа не обязательно должна использоваться в операторе описания, в то время как параметр может быть использован в нем, например DECLARE TABLE (TABLE' SIZE). Рассмотрим пример параметризации.

Пример параметризации. В практике применения ЭВМ существует фундаментальная задача обнаружения ситуаций, когда возможности программы исчерпаны. Слишком часто вместо решения этой проблемы ее просто игнорируют (исчерпав возможности программы, машина выбрасывает ее неожиданно для программиста).

Чаще, к счастью, получается так, что программа сама выявляет возможную аномальную ситуацию. При этом печатает предупреждение и производит регенерацию (например, деликатно прекращает работу или в случае потенциального переполнения переключает указатель области данных на возможно более безопасное положение).

Вопрос стоит так: как лучше всего с позиции сопровождения обнаружить подобные ситуации? Легко предположить, что ответ связан с параметризацией.

Рассмотрим пример на языке Ада:

```
FILE_TABLE_SIZE: constant integer: = 15;
FILE_TABLE: array (1..FILE_TABLE_SIZE) of FILE_LOCATION;
— — предположим, что FILE_LOCATION является типом запи-
    си, определенной ранее
function ADD  (FILE_NAME) return FILE_INDEX is
begin
  — — добавить файл к таблице файла
  for I in 1..FILE_TABLE_SIZE loop
    if  FILE_TABLE(I) = FILE_NAME then return I; end if;
    if FILE_TABLE(I) = NULL then
      FILE_TABLE(I): = FILE_NAME; return I; end if;
  end loop;
  raise TABLE_OVERFLOW;
end ADD;
```

В примере есть два места, совершенно различные по своей природе, из которых ясно видны возможности FILE TABLE, — это описание размера таблицы и создание цикла с меняющимся параметром цикла. Если появляется необходимость изменить размер таблицы, например увеличить его в случае обнаружения переполнения, то необходимое изменение вносится только в описание параметра таблицы. Описание таблицы и цикл не зависят от размера таблицы; известно, что таблица имеет определенный размер и что он поименован.

Проблема параметризации возникает почти всякий раз, когда в программе имеются совокупности данных. Описание совокупности должно сопровождаться решением вопроса о возможности переполнения. И если переполнение возможно, следует применить параметризацию.

Обмен данными

Программистам доступны три способа обмена данными в рамках внутрипрограммного обмена данными. (Обмен данными мы определяем как средство передачи данных от одного модуля к другим.) Первый способ заключается в использовании списков параметров. Второй — в применении глобальных или общих данных. А третий связан с только нарождающимся понятием «кластер». В этом случае «полуглобальная» база данных используется совместно с ограниченным числом программ [41, 42].

Межпрограммный обмен данных также важен. Обычно он осуществляется путем передачи флажков или блоков данных через общую область, обеспечиваемую операционной системой, или путем передачи более значительных объемов информации файлами. Идея так называемого трубопровода, по которому идет обмен файлами между программами, высказанная создателями системы UNIX, весьма привлекательна, но все же она является упрощенным решением задачи [45].

Выше мы обсуждали вопрос обмена данными в разд. «Модульность». Здесь мы снова возвращаемся к этой теме просто для того, чтобы подчеркнуть ее самостоятельное значение для профилактики сопровождения. Как уже отмечалось, обмен параметрами является, строго говоря, наиболее предпочтительным способом обмена данными. Но практика часто требует применения глобальных (общих) данных, особенно для больших программ, а кластер нередко выручает там, где появляются затруднения, связанные с неограниченным применением глобальных данных. Здесь мы не приводим соответствующей оценки достоинств межпрограммных связей.

Далее дается пример обмена данными.

Пример обмена данными. Давайте вспомним, что процедурам CREATE и SEARCH для работы требуется ряд параметров. Процедуре CREATE необходимо сообщить имя файла, который следует создать, а процедуре SEARCH — имя файла и имя ключа, который надо отыскать в файле. Эта информация передается процедурам наиболее предпочтительным способом — в виде параметров. Допустим, имеем вызовы CREATE(FILE1); SEARCH(FILE1, KEY1); которые требуют определенных услуг для FILE1 и KEY1.

Если бы в системе был только один файл (что маловероятно согласно постановке задачи, но возможно), то не было бы необходимо-

сти именовать его в списке параметров. А если бы там была глобальная переменная, названная KEY1, которая могла бы быть установлена пользователем процедур CREATE и SEARCH и была доступна этим процедурам, мы имели бы

```
GREATE;
KEY1: = SOMETHING;
SEARCH;
```

Программа теперь более лаконична, но менее описательна. И в этом случае есть вероятность того, что на переменную могут повлиять другие части программы. Например, если предположить, что программист собирается использовать значение переменной KEY1, установленное ранее, то

```
CREATE;
KEY1: = SOMETHING;
...
SEARCH;
```

В этом случае, если пассивная часть программы, обозначенная через «...», содержит инструкцию присвоения для KEY1 либо в первоначальном варианте программы, либо в ее последующей модификации, то процедура SEARCH будет ошибочна. По этой причине использование глобальных данных нежелательно.

И наконец, предположим, что имя файла и имя ключа не подлежат передаче в память в качестве параметров. В этом случае нужно осуществить блокировку памяти. Такую возможность предоставляет кластер.

```
package FILE_MANAGER is
   – – здесь определены услуги пользователю
   – – KEY1 здесь не определен и, таким образом, не доступен
        извне
end FILE_MANAGER;


package boby FILE_MANAGER is
   KEY1: integer;
   ...
   – – здесь определено применение услуг пользователю
   ...
   KEY1: = SOMETHING;
   ...
   SEARCH;
   ...
end FILE_MANAGER;
```

Вероятность того, что переменная KEY1 попадет ошибочно в память машины в промежутке между присвоением ей значения и вызовом процедуры SEARCH, теперь значительно уменьшилась, так как никакие операторы, лежащие вне кластера, на нее повлиять не смогут. Кластер позволяет воспользоваться глобальными данными, но при этом требуется блокировка памяти.

Защита программы от возможных ошибок

«Если ошибка может возникнуть, то она непременно возникнет». Эти слова должны быть написаны большими буквами над рабочим местом программиста. Если учесть, что работа программиста состоит в том, чтобы давать утомительные, подробные указания немой машине, легко понять, что здесь имеются большие возможности для совершения ошибок.

Приходилось ли вам когда-нибудь кому-нибудь объяснять дорогу к какому-нибудь месту? А теперь представьте, что этот человек слепой. Поразмыслим над разницей. А количество деталей, необходимых машине, еще весьма далеко от количества деталей, необходимых слепому. Опытный программист заранее знает, что будут ошибки, некоторые из которых проявятся только через годы. Он пишет свою программу таким образом, чтобы иметь возможность самому обнаружить ошибки сразу или же чтобы их обнаружил и устранил сопровождающий программист. Это мы назовем здесь обобщающим термином *защита программы* и проведем аналогию с «безопасным вождением» автомобиля, которое подразумевает повышенное внимание водителя к возможным опасностям.

Защита программы от потенциальных ошибок — это довольно широкое понятие, которое в значительной степени зависит от конкретного применения. Однако можно выделить несколько общих принципов. Одним из методов защиты программы является метод *утверждений* [53]. Программа должна содержать все утверждения программиста о желательном поведении программы или данных, и, более того, программа должна обладать способностью проверить правильность этих утверждений. Должно быть также особое «устройство», которое печатает диагностику ошибок, а также указывает, в каком месте они произошли, и сообщает либо о возобновлении работы программы, либо об исключении из нее ошибочного утверждения.

Сам термин «утверждение» может вводить в заблуждение. Он возник как один из терминов, употребляемых для доказательства правильности чего-либо. Здесь он употребляется совсем в другом контексте. Речь идет о расширении хорошо известных методов редактирования данных.

Утверждения могут быть использованы, например, чтобы обнаружить:

1. Ошибочные и неправдоподобные входные данные.
2. Ошибочные и неправдоподобные выходные данные.
3. Ошибочные и неправдоподобные данные вообще.
4. Ошибки в логике.
5. Нежелательные побочные эффекты вызова процедур.
6. Индексы, лежащие вне диапазона.
7. Использование неинициированных переменных.
8. Переполнение памяти.
9. Условия исключения из программы.
10. Любые другие условия, известные программисту как ошибочные.

Некоторые из этих пунктов заслуживают более пристального внимания. Основным из них является верификация входных данных. Строгая проверка достоверности входных данных, особенно если они поступают от неопытного сотрудника, может иметь решающее значение для стабильной работы программы (и программиста!). Если ошибки не замечены на входе, на выходе получаются неудовлетворительные результаты.

Переполнение памяти машины также может иметь серьезные последствия. Если программист говорит: «Откуда я мог знать, что пользователь попытается заложить 101 позицию в массив размерностью 100?» — то это значит, что он не предусмотрел защиту программы от возможных ошибок. Ему следовало ввести простые утверждения в каждой точке, в которой размерности массива присваивается конкретное значение (или в точке, где увеличивается индекс массива). Утверждение обнаружило бы переполнение, диагностировало бы его извне и приняло бы все возможные меры к восстановлению программы. Если этого не сделать, программа может повести себя самым невероятным образом.

(У авторов был случай, когда написанная ими программа вызвала переполнение таблицы символов компилятора, которое не было обнаружено сразу. Компилятор уничтожил участок памяти, напечатал 512 бессмысленных сообщений, а затем выбросил программу.)

Открытым остается вопрос о том, сколько утверждений следует оставлять активными в программе на протяжении ее эксплуатации, учитывая, что их использование влияет на время выполнения и размеры программы. Некоторые специалисты говорят, что не могут позволить себе включать утверждения в программу. Другие говорят, что не могут позволить себе этого не делать. Видимо, вопрос о том, включать или нет, и если да, то в каком объеме, должен решаться, исходя из конкретных условий.

Следует отметить, что существуют определенные способы, позволяющие легко вносить утверждения в программу или извлекать

их из нее. Некоторые компиляторы могут распознавать утверждения (при этом они должны быть заранее включены в синтаксис) таким образом, чтобы опция либо игнорировала их, либо обрабатывала во время компиляции. Другие компиляторы обеспечивают возможность «условной компиляции», при которой программа любого вида (в том числе и с утверждениями) может либо игнорироваться, либо обрабатываться с помощью опции. И если имеется намерение использовать утверждения избирательно (например, для отладки), для удобства работы необходим один из этих способов.

Защита программы от ошибок, кроме утверждений, предусматривает использование резервов. Необходимо преднамеренно сберечь некоторую часть ресурсов системы, т. е. не использовать сразу, чтобы у сопровождающего программиста была возможность выполнить свою работу должным образом. Будь то магнитная память или память на дисках, если она на 100% использована разработчиком, сопровождающему программисту вряд ли удастся что-либо исправить или дополнить. Если все резервы памяти системы использованы уже на начальной стадии, сопровождающий программист окажется в затруднительном положении.

На заре развития ЭВМ считалось нормальным, что половина времени сопровождающего программиста уходит на высвобождение ресурсов программы, а вторая половина — на то, чтобы расширить ее возможности и воспользоваться ресурсами, которые удалось высвободить. В наше время это уже непозволительно. Для нужд сопровождающего программиста необходимо иметь резервы.

Существует еще один способ защиты программы от возможных ошибок, которым является автономный блок записи, соединенный с программой. Это понятие заимствовано из области авиации и заключается в следующем: программа может производить непрерывную запись всех важных событий, которые происходили за время ее работы (например, поступление модулей, сходящиеся итерации, завершение циклов, запись утверждений и т. д.). С помощью такого устройства можно записать все ошибки и сбои программного обеспечения, что позволит определить причину аварии. (Здесь проводится полная аналогия с аварией самолета.)

Необходимая частота включения записи определяется с помощью дополнительной условной компиляции исходной программы. Например, при нормальном выполнении программы из соображений производительности могут быть записаны только начала основных фаз работы системы; но, если понадобится провести анализ причины сбоя системы, мы будем иметь возможность осуществить перекомпиляцию, увеличив частоту включений записи (например, для того чтобы проследить введение модулей в программу и изменение ключей данных). Если условия оптимизации позволяют, то частота записи может быть такой, что весь этот процесс заменит выполнение программы.

Еще одним методом профилактического программирования является практика расстановки флажков в программе и составление перечня ненадежных методов работы. Как ни странно, программисты нередко прибегают к рискованным способам программирования. Причины бывают самые разные: от стремления добиться цели («язык программирования не позволяет решить задачу без небольших уловок») до стремления к оптимизации («я не смог бы уложиться в отведенное время, если бы не схитрил»). Такие случаи не являются исключением, а скорее представляют собой повседневную реальность, с которой в большей или меньшей степени сталкиваются все программисты.

Чтобы убедиться в достоверности вышеупомянутых фактов, рассмотрим примеры, иллюстрирующие применение ненадежных методов программирования во многих языках программирования.

Пример	Ненадежные методы программирования
Хэш-алгоритмы программы	Нарушение типа — над последовательностью символов, к примеру, может быть выполнено арифметическое действие
Преобразование последовательности символов	Нарушение типа — для доступа к массиву знаков в качестве индекса использован знак
Стандартная программа вычисления	Подпрограмма, имеющая один вход. Обычно алгоритмы вычисления функций sin и cos программируются как стандартная программа с двумя входами
Исключение возвратов	Использование GOTO
Разгрузка памяти, очистка памяти и т. д.	Нарушение типа — адресация в памяти может быть представлена, например, как последовательность битов независимо от того, как описан тип в действительности
Сервисное прерывание и т. д.	Абсолютная адресация — фрагмент данных может быть получен только по абсолютному адресу

Правильный подход к ненадежным методам программирования, таким образом, предполагает необходимость: 1) признать за ним право на существование; 2) создать языки программирования и разработать стандарты, позволяющие свести к минимуму количество случаев применения ненадежных методов программирования (это даст программисту возможность выполнить свою работу надежно и четко); 3) снабдить программу средством регистрации, чтобы все случаи применения ненадежных методов заносились в листинг (какой именно ненадежный метод и почему он был использован?) и суммировались в конце программы для отчета программиста руководству.

В настоящее время такие сведения должен готовить сопровождающий программист. Однако помощь близка! В языке Ада разработана методика применения ненадежных методов программирования, а компиляторы Ада способны автоматически составлять отчеты о их применении в программах.

Для быстрого обнаружения и устранения ошибок сопровождающему программисту важно знать, какие части программы составлены с применением ненадежных методов.

Примеры защиты программ от возможных ошибок. В языке Ада заложена некоторая возможность использования утверждения. Например, программист может оговорить область определения переменной в точке ее описания:

```
ARRAY_SIZE: integer rang 1..100;
```

Затем, если переменной присваивается значение, лежащее вне области определения, компилятор или стандартная подпрограмма исчисления времени выполнения обнаружит этот случай и появится сообщение (это должно быть сделано с помощью опций, так как увеличение времени работы программы нежелательно):

```
ARRAY_SIZE: = -5; - - диагностику проводить во время
                      компиляции
or
I: = 100;
...
ARRAY_SIZE: = I + 1; - -исключения из области определяют-
                      ся в ходе выполнения программы
```

Программист может обеспечить защитную проверку даже при отсутствии утверждения для области определения:

```
ARRAY_SIZE: = XX;
if ARRAY_SIZE < 1 or ARRAY_SIZE > 100 then
   PRODUCE_DIAGNOSTIC (ARRAY_SIZE);
   ABORT_GRACEFULLY;
end if;
```

В некоторых случаях применения утверждений программист явно объявляет область определения, а затем получает диагностику выходов за рамки ее области определения за время выполнения программы:

```
!RANGE_ARRAY_SIZE (1..10);
...
ARRAY_SIZE = 101; -диагностика будет дана во время выпол-
                   нения программ
```

Как уже говорилось выше, проверка области определения не является единственным видом защитного программирования, хотя это, возможно, один из самых важных способов. Взаимоотношения между переменными или побочные эффекты вызова процедур также важны:

```
!RELATION FIRST_THINGEE<=NEW_THINGEE<=
  LAST_THINGEE AND NEW_THINGEE<0;
```

Это такие взаимоотношения, при которых процессор, обрабатывающий утверждения, мог бы обнаружить нарушение и произвести их диагностику; и

```
!CALL(FIRST_VARIABLE; SECOND_VARIABLE);
EXECUTE_PROCEDURE(PARAMETER);
```

Это запрос утверждения проверить побочные воздействия процедуры EXECUTE PROCEDURE на переменные FIRST_VARIABLE и SECOND_VARIABLE. Заметим, что, если бы не было такого утверждения, те же проверки были бы запрограммированы в менее удобной форме:

```
a) if NEW_THINGEE<FIRST_THINGEE or
      NEW_THINGEE>LAST_THINGEE or
      NEW_THINGEE = 0 then
      PRODUCE_DIAGNOSTIC(NEW_THINGEE);
   RECOVER; end if;
b) SAVE_FIRST: = FIRST_VARIABLE:
   SAVE_SECOND: = SECOND_VARIABLE;
   EXECUTE_PROCEDURE(PARAMETER);
   if FIRST_VARIABLE / = SAVE_FIRST or
      SECOND_VARIABLE / = SAVE_SECOND then
      PRODUCE_DIAGNOSTIC(EXECUTE_PROCEDURE);
      RECOVER;
   end if;
```

Стандарты программирования

Стандартами программирования называются требования, предъявляемые к программисту окружением. Они могут быть продиктованы его руководителем, условиями контракта или коллегами-программистами. Эти требования обычно высоки и трудновыполнимы. Они, как правило, бывают навязаны программисту в целях повышения качества программирования и улучшения администра-

тивного контроля. Выигрывает от соблюдения этих правил в первую очередь сам сопровождающий программист. Программы, написанные в соответствии со стандартами, обычно более понятны, и с ними легче работать. Это способствует профилактике сопровождения [57].

Стандарты программирования, однако, могут быть неверно использованы. Слишком часто они подменяются набором приемов программирования (а иногда уловок), свойственных данному руководителю, программисту или, что еще хуже, непрограммисту. Иногда они также являются отражением последних веяний в технике программного обеспечения независимо от того, приняты они или нет. В результате получается смесь хороших методов и чепухи.

Пожалуй, прежде всего следует установить перечень стандартов для лиц, составляющих их:

1. Не устанавливай новых технических условий, если не уверен, что это даст улучшение результатов.
2. Делай спецификацию небольшой по объему, чтобы она легче запоминалась.
3. Не подвергай стандартизации те вещи, которые уже определены языком или системой.
4. Выделяй основные направления, но не стандарты.
5. Выделяй содержание, обращая внимание на форму лишь по необходимости.
6. Определи, каким образом можно обойти стандарты.
7. Считай программиста профессионалом-практиком, а не дилетантом.

С этими оговорками стандарты, возможно, смогут помочь сопровождающему программисту.

Типичные современные стандарты включают в себя соглашения (перечень правил для поименованных или логических сущностей), ограничение сложности (перечень правил, ограничивающих число строк в программном модуле), конструктивные ограничения (перечень правил, оговаривающих допустимые языковые формы, такие, как GOTO или его модификации) и т. д. Как мы уже убедились, соглашения о структурированном присвоении имен и ограничения количества строк в программном модуле имеют отрицательные последствия. Таким образом, становятся очевидными накладываемые конструктивные ограничения, по крайней мере в современных языках.

Примеры стандартов программирования. На основе одного очень крупного проекта программного обеспечения (который в свою очередь был частью еще более крупного космического проекта) разработан справочник стандартов объемом около 100 страниц! Правила, которые приводятся ниже, являются наилучшими образцами из этого справочника.

1. Глобальные базы данных следует разбить на блоки, чтобы свести до минимума число модулей, требующих доступа к блоку.

2. Переменным следует присваивать однозначные имена, а не EQUIVALENCEd или OVERLAYed.

3. Запросы ввода/вывода должны обрабатываться централизованно рабочим модулем с подмодулями по одному в каждом.

4. Язык проектирования программы следует включать в программу, полученную на этапе проектирования в виде комментариев.

5. Следует для каждого модуля программы завести папку его разработки и сопровождения. Содержание папки должно включать: а) титульный лист; б) журнал измерений; в) требования к модулю; г) проектирование; д) листинг модуля, находящегося в обращении; е) план проверки; ж) результаты проверки; з) отчеты об ошибках; и) комментарии; к) замечания разработчика.

6. Названия следует выбирать так, чтобы они раскрывали суть функций и включали в себя структуру программы. Структура программы должна быть выражена путем введения в имя: а) сокращенного названия модуля; б) названия совокупности данных или названия блока, если необходимо, в сокращенном виде.

7. Следует выделять элементы структурного программирования.

8. Комментарии должны включать как минимум следующее: а) детализацию функций и интерфейсов каждой подпрограммы; б) описание применения и области существования каждой переменной; в) объяснение каждой подфункции; г) объяснение каждой ненадежной или сложной программы.

9. Глубина вложения должна быть минимальной.

10. Размеры модулей должны определяться их функцией, однако их средняя величина не должна превышать 100 строк.

Документация

Многое из того, о чем говорилось выше, имеет первостепенное значение. Самым важным средством профилактики сопровождения является модульность. Следом за ней по степени важности идет структура программы и структура данных. Стандарты программирования, хотя им уделяется большое внимание, вносят значительно меньший вклад (в основном они привносят понятия более высокого уровня).

О документации мы решили говорить в конце, но не потому, что она менее важна, а потому, что 1) она очень сильно отличается от всего остального и 2) о ней довольно подробно говорится в других разделах книги. Еще раз отметим, что документация детального уровня должна содержаться в виде комментариев в листинге программы, а не в виде отдельной книги.

В этом разделе мы не станем больше рассуждать, но приведем ряд конкретных предложений по вопросу о том, какие комментарии,

где и в каком объеме должны лечь в основу такой документации. Заранее отметим, что хорошая документация (т. е. «комментарий») *не менее важна для профилактики сопровождения, чем все остальные средства.*

По крайней мере следующие места программы должны содержать комментарии:

1. Начало каждого модуля, включая название модуля; дату назначения модуля, его входы и выходы; перечень имеющихся ограничений на работу модуля, включая основные предположения их возникновения; описание работы отрезка программы, на котором появляется ошибка и имя разработчика. Главные модули должны также включать всю хронологию их модификаций: дату внесения каждого изменения, имя сопровождающего программиста, цель изменения и его место в программе.

2. Каждая подфункция, будь то прямая последовательность операторов или логический переход, или блок begin-end, должна быть объяснена.

3. Каждый интерфейс, включая его явное определение и ссылку на информацию о других аспектах его функционирования (где возможно).

4. Каждая группа функционально или каким-либо другим способом связанных описаний, объяснений назначения и состава группы.

5. Каждое описание, включая объяснение назначения каждого пункта, его возможных величин, если такие имеются.

6. Каждая труднопонимаемая часть программы, включая объяснение работы программы и обоснование причин использования сложной программы.

Трудность, однако, состоит не в том, чтобы знать, какие комментарии и где нужно писать, а в том, чтобы описать их. Поэтому хороший руководитель должен настоять на том, чтобы необходимые комментарии включались в программу. Набор программ, данных в обзорах [1, 29], указывает сопровождающему программисту на необходимость включения в программу адекватных комментариев. Эти обзоры имеют также ряд других достоинств. Возможно, с них и начнется правильное расположение комментариев. (Если программист ввел язык проектирования в качестве комментариев, ему, возможно, не придется добавлять множество дополнительных комментариев в дальнейшем.) Вскоре должны появиться звуковые прослушивающие устройства. Либо компилятор, либо автоматизированное устройство прослушивания программы могут относительно легко проверить соответствие программы комментариям и другим стандартам.

Умение составлять хорошую документацию является своеобразным искусством, сравнимым с искусством писателя. Приведем примеры.

Пример документации. Хорошо знакомая нам функция SEARCH неоднократно рассматривалась выше, но с точки зрения структуры программы. Ниже мы вновь возвращаемся к ней для иллюстрации правильного комментирования, позволяющего сделать программу самодокументирующейся:

```
-- FUNCTION SEARCH
-- INPUT = FILE_NAME AND KEY_TYPE,
-- OUTPUT = FILE_INDEX
-- поиск файла, содержащего инструкции, затем
-- осуществляется поиск файла для данного ключа и
-- возврат указателя к файлу FILE_INDEX
-- вызов файла исключений BAD_FILE, если инструкции
-- не содержат ответа на запрос, или
-- вызов файла исключений BAD_KEY, если искомый
-- ключ не содержится в файле, написанном
-- 25 декабря 1981 г. Стадсом Аксвингером
function SEARCH (FILE_NAME; KEY_TYPE)
  return FILE_INDEX is begin
-- смотри запись данного ключа в данном файле
    for FILE_POINTER in FILE_INDEX'FIRST..
      FILE_INDEX'LAST loop
  -- первоначальное месторасположение файла
      if FILE_MAP (FILE_POINTER) = FILE_NAME
      then goto FIND_KEY;
      end if;
    end loop;
    raise BAD_FILE; return; -- если файл не найден
    «FIND KEY» RECORD_COUNT: = 0; -- файл получен
      while KEY_TYPE = FILE_KEY loop
        -- затем найден ключ
        READ (FILE_POINTER); -- читайте запись
        RECORD_COUNT: = RECORD_COUNT + 1;
        -- помните, где находитесь
      end loop;
      FILE_INDEX: = RECORD_COUNT; -- сохраните ре-
     зультат
      return;
    begin -- исключение блока
      exception
        when END_OF_FILE raise BAD_KEY; return;
       -- если ключ не найден
   end
end SEARCH;
```

3.2.2.2. Трудоемкость

Как бы хорошо ни была проведена профилактика сопровождения, мы должны признать, что бóльшая часть работ по сопровождению программного обеспечения представляет собой тяжелый труд. Некоторые проблемы могут быть разрешены только путем кропотливого вникания во все детали программы. Во многих случаях решения приходится искать, перебирая все возможные варианты, постепенно отвергая их один за другим, а то, что остается, считать решением. Иногда решение находится с помощью распечатки всех возможных вариантов. Многие задачи заставляют многократно возвращаться к этому процессу, причем при каждом новом пересмотре приходит более глубокое понимание строения программы. После долгих усилий решение вдруг может прийти само как интуитивное озарение, о котором мы уже упоминали. Но конечно, такое «озарение» невозможно, если ему не предшествуют долгие поиски.

Существует класс задач программного обеспечения, характеризующихся особой сложностью решения. В этом случае почти всегда создается впечатление, будто для их решения не хватает возможностей вычислительной машины. Программа не может работать правильно в условиях неопределенности. А воспроизвести задачу часто невозможно (или трудно). Отдельные задачи такого типа требуют больших затрат и тяжелого труда, чтобы их исправить (и к тому же немалого терпения). Для этого должны быть использованы все обычные методы коррекций и, кроме того, приходится бороться с желанием надеяться на то, что проблема решится сама собой.

3.2.2.3. Улучшение программы

Следует непрерывно улучшать качество работы программы. Очень часто под улучшением программы понимают включение в нее элементов профилактики сопровождения, которые следовало бы внести еще разработчикам. Методы, приводящие к таким улучшениям, требуют:

1. Применения модулей: там, где текст программы повторяется, необходимо сделать процедуру, с помощью которой можно вызывать модуль. Это также позволит сократить исходную программу, часто значительно. В дальнейшем необходимо пытаться выделить функциональные области программы для облегчения дальнейших изменений.

2. Исключения из программы всех тех операторов GOTO и меток, без которых можно обойтись. Хотя чрезмерное стремление к отмене операторов GOTO порождает проблем больше, чем позволяет решить, в наиболее очевидных случаях при необходимости запустить в серию несерийную программу нужно исключить их.

3. Применения взаимоконтролирующихся модулей. Пусть каждый модуль проверяет состав и соответствие сомнительных данных из других модулей. Это позволит по крайней мере обнаружить ошибки и недочеты раньше, чем они возникнут при работе системы. В тех случаях, когда известны диапазоны изменения величин переменных, необходимо выяснить, нет ли переменных, значения которых лежат вне указанных ограничений. В этом состоит методика контроля утверждений, о которой говорилось выше в этом разделе.

3.2.2.4. Большие системы. Метод подхода

Большие системы ставят целый ряд оригинальных задач как перед создателями программного обеспечения, так и перед специалистами по его сопровождению. Хотя решения этих задач по-прежнему являются предметом поиска, наиболее общая методика, использующая структурный подход к этим задачам, заключается в дополнительной формализации спецификаций, проектных представлений интерфейсов системы, предполагаемых контрольных выходов и отчетов о результатах контроля.

Подобная формализация может оказаться полезной для сопровождающего программиста. Такой подход особенно хорошо освещен в литературе [18]; неудачное сопровождение программного обеспечения улучшено с помощью полезного эксперимента, в котором система перестраивается более формально — путем использования исправленных и дополненных требований к техническим условиям программного обеспечения.

Когда в большой системе возникает ошибка, следует учесть определенные соображения. Метод поиска ошибок в больших системах должен включать, по-видимому, следующие стадии:

1. Исправление ошибок.

а) **Убедитесь в том, что ошибка существует.** Указывает ли постановка задачи и сопроводительная документация на разницу между фактическими и ожидаемыми результатами? Нередко пользователь приходит к сопровождающему программисту или консультанту с теми результатами, в которых он сомневается, но после обсуждения очень часто находит ответ на свой вопрос в справочнике пользователя. Иногда ответ содержится не в справочнике, а в требованиях. Тогда необходимо внести соответствующие изменения в справочник пользователя. Таким образом может быть исключено большое число ошибок.

б) **Выделение ошибки.** Выделение ошибки состоит в определении с помощью всей имеющейся информации наиболее вероятного ее местонахождения в программе. На этом этапе сопровождения программы или системы обязательным является тщательный пересмотр системы и всех имеющихся средств сбора информации о

ней. Должны быть определены отдельные части системы, которые могут иметь отношение к возникшей ошибке. В некоторых случаях, когда недостает необходимой информации, может оказаться полезным создать средство для ее получения. Вновь созданное средство будет использовано, когда этот тип информации потребуется снова. При наличии таких средств, возможно, потребуется одна или более прогонок программы, содержащей ошибку, для того чтобы получить всю необходимую информацию. (Были ли верны результаты работы программы в точке А? в точке Б?...) В результате такого анализа ошибка программы будет обнаружена.

в) **Воспроизведение ошибки.** Предположим, что ошибка существует и выделена. Предположим далее, что интуитивное мгновенное решение не возникло, — тогда требуется помощь. В большой системе важно создать локальный тест, позволяющий выявить признаки ошибки, т. е. воспроизвести ошибку в том виде, в котором ее легче будет устранить. Следует отбросить всю избыточную, ненужную информацию, которая окружает данные, содержащие ошибку, и выделить ее в чистом виде. Если, к примеру, потребуется трассировка или другая отладочная информация, то желательно получить распечатку только той области программы, в которой содержится ошибка, а не целый ворох бумаги, бо́льшая часть которой не имеет прямого отношения к решаемой проблеме. Часто такой локальный тест может в дальнейшем пополнить набор тестов для регрессивного тестирования системы.

3.2.2.5. Эволюция против революции

Процесс сопровождения может быть либо эволюционным, либо революционным. Если система просуществовала длительное время, отвечает своему назначению, содержит минимальное количество неполадок, используется многими людьми и достаточно доступна для понимания, то такая система может быть развита и усовершенствована. И наоборот, если работа программы непредсказуема, необъяснима, а сама программа имеет запутанную структуру, требуется немедленное, решительное, оперативное вмешательство.

Некоторые части системы могут сочетать в себе и те и другие свойства (т. е. система может иметь как части с хорошей модульностью, так и запутанные части), и таким образом в рамках одной системы можно прибегнуть и к эволюционным, и к революционным преобразованиям. Если в основе процесса проектирования лежит ошибочная идея, приходится пользоваться революционными методами (т. е. переделать модуль полностью).

В случаях, когда система создавалась в результате последовательного добавления к ней маленьких кусочков программ, возможно, следует воспользоваться эволюционными методами. В программе, написанной в старом стиле, обычно встречается много операторов

GOTO, а также имеются ссылки на глобальные метки, которые использовал предыдущий разработчик.

Например, одна большая и сложная часть программ выполняла шесть функций, имеющих высокую степень общности. Поток информации через эту часть программы имел большую интенсивность в течение всего времени ее работы. При этом, однако, значения некоторых функций были столь отличны от других, что программа во многих случаях сбивалась на обработку особых условий, а затем вновь переходила к выполнению основной программы, но уже в другом месте. Само собой разумеется, что следить за такой программой было очень трудно, а изменять ее без введения дополнительных флажков невозможно. Чтобы получить эту программу в виде отдельных модулей, ее модифицировали медленно, в течение ряда лет. Это позволило независимо усовершенствовать каждую из шести основных функций таким образом, что вспомогательные процедуры или функции вызывались по мере необходимости, причем вспомогательные функции были общими для всех основных. В результате такой эволюции стало возможным вносить изменения в каждую из шести функций, не затрагивая ни одной из остальных пяти.

3.2.3. ДОКУМЕНТАЦИЯ

Документация программного обеспечения редко содержит всю информацию, необходимую для сопровождения. Документы в том случае, если они есть, бывают написаны в соответствии с конкретными требованиями к их содержанию. К сожалению, слишком часто они содержат много излишней информации и не содержат того, что *действительно* необходимо. Таким образом, вместо того чтобы дать необходимую информацию, документы нередко скрывают ее.

Так как документация существует отдельно от программы, она часто устаревает. В идеале документ должен быть полным отражением программы. На самом деле так почти никогда не бывает. В результате документация может вводить в заблуждение. Какой здравомыслящий человек станет вносить изменения в программу, прочитав лишь документацию?

В нашей книге мы рекомендуем установить порядок, при котором документация помещалась бы в листинг. Все требования к документации, описывающей программу, практически могут быть выполнены и даже перевыполнены в листинге. (Это положение обсуждается в разд. 4.3.)

3.2.3.1. Историческая документация

Методика написания документации к программному обеспечению всегда обходила историческое документирование. Историческая документация— это неформально представленная информация, ис-

пользуемая на определенном этапе создания программы. Она включает следующие понятия:

1. **Заметки на стадии проектирования.** Это наиболее важная форма исторической документации, состоящая из записей проектировщика (они подробнее и шире официального описания проекта, представленного на языке разработки программы или в виде структурных схем). Эти записи, обычно сделанные от руки, незаменимы, если сопровождающему программисту необходимо видоизменить отрезок программы и он хочет узнать, почему имеющийся вариант проекта развивался именно этим путем, а не другим. Большинство современной документации не представляет ему таких сведений. Часть этого материала может быть оформлена в виде первоначального или обзорного документа, но мы советуем сохранять ее в самом первозданном виде и хранить в хронологическом порядке. Части материала, относящиеся к одному и тому же вопросу, могут быть помечены перекрестными ссылками путем введения нумерации страниц в файл разработки и создания списка ссылок на эти номера. Причина, по которой заметки проектировщика обычно не оформляются в виде отчета, состоит в том, что по мере применения программы они часто устаревают. Важно понять, что заметки проектировщика 1) дают полное общее представление о процессе проектирования, но не о деталях, 2) следует постоянно исправлять, чтобы они не устаревали. Первое свойство заметок обычно используется программистами. Что же касается постоянного обновления заметок, то это-дело будущего.

2. **Отчет о проблемах.** Следует вести хронологическую картотеку возникающих ошибок, пронумерованную для удобства нахождения нужного материала. Ошибки должны храниться в рабочей картотеке вплоть до окончания работы, а затем переноситься в историческую картотеку. (Частично эта работа может быть автоматизирована путем извлечения данных из индивидуальных отчетов и составления сводных списков.) Эти исторические записи также следует сохранять в первоначальном (неперепечатанном) виде, так как при этом может быть потеряна часть информации. Хотя эти картотеки применяются нечасто, но когда они все же понадобятся, от них будет зависеть многое (например, с их помощью можно выделить и исправить повторяющиеся ошибки).

3. **Предложения по усовершенствованию.** Еще одним важным аспектом исторической документации является сбор идей по новейшим усовершенствованиям программы. Идеи бывают двух видов: коренные улучшения программы и косметические усовершенствования. Последние обычно включаются в листинг или в текущий вариант программы, а затем забываются. Но коренные улучшения, которые обычно приходят в голову, когда программист занят совсем другой работой, следует хранить централизованно (в папке или подшивке) и регулярно просматривать. Тогда хорошие идеи не будут

теряться, и программисту всегда будет что показать руководителю или заказчику.

4. **Описание версий** — сопроводительная документация. Этой важной части документации не всегда уделяется должное внимание. Каждый вариант программы должен иметь полное описание всех внесенных в него изменений. Сюда входит список отчетов об ошибках, которые были устранены, словесное описание изменения и, где возможно, описание влияния этих изменений на пользователя в терминах, понятных пользователю. Здесь также следует дать ссылки на справочник пользователя, справочник документации, изменения в документации.

Заметим, что ни одна из этих форм ведения документации не является традиционно обязательной, тем не менее все они очень важны. Заметим также, что в большинстве случаев эту документацию лучше всего вести от руки, так как ясность и свежесть первоначально возникшей мысли легко может быть утеряна в процессе оформления.

ЛИТЕРАТУРА

1. См. [1] к гл. 2.
2. Bladen, Standard Compiler Workshop Final Report, Eglin Air Force Base, 1978.
   Даны результаты семинара, целью которого было определение оборудования для алгоритмического языка Ада. Четыре рабочие группы разрабатывали следующие темы: 1) план работы; 2) оценки спецификаций и верификаций; 3) оптимизация; 4) вспомогательные средства программного обеспечения. Последняя группа предложила специальный набор вспомогательных средств для поддержания языка Ада.
3. Boehm, Software Engineering, *IEEE Transactions on Computers* (December 1976).
   Даются определения техники программного обеспечения и его составных частей. Обсуждаются затраты и основные направления. В гл. 7 «Сопровождение программного обеспечения» говорится, что сопровождение поглощает около 70% средств и что ему уделяется мало внимания.
4. Boehm, Brown, Kaspar, Lipow, MacLeod, Merritt, Characteristics of Software Quality, North-Holland, 1978.
   Обсуждается сложность сопровождения как основной элемент надежности программного обеспечения. Показано разделение сопровождения по функциям в виде древовидной структуры составляющих его частей. Даны положительные и отрицательные характеристики сопровождения.
5. Carter, Donahoo, Farquhar, Hurt, Software Production Data, RADC-TR-77-177, 1977.
   Описываются исследования, проведенные корпорацией Computer Sciences с целью оценить эффект «практики современного программирования» (ПСП). Вводится понятие «нить», используемое для трассировки требований в последовательности фаз жизненного цикла программного обеспечения, и дается оценка применению этого понятия в различных проектах. Дается оценка влияния элементов ПСП на жизненный цикл и на различные типы ошибок.
6. См. [6] к гл. 2.
7. Feldman, Make – A Program for Mainteining Computer Programs, *Software Practice and Experience* (April 1979).
   Описывается средство управления версиями, использованное в операционной системе UNIX.
8. Fox, E-3A Software Maintenance, Proccedings of the AIAA Conference on Computers in Aerospace, 1977.
   Описывается работа по сопровождению, проведенная в рамках проектов командования и управления Министерства обороны США. Обсуждается проект E-3A, струк-

тура управления сопровождением программного обеспечения и связанная с этим деятельность.
9. См. [10] к гл. 2.
10. Gannon, A Verification Case Study, Proceedings of the AIAA Computers in Aerospace Conference, 1977
Описывается автоматическая верификационная система (ABC) JOVIAL и способы ее применения. Обсуждается способность ABC к самопроверке и использование ее как части правительственной приемочной проверки. Делаются выводы об эффективности тестирования.
11 Gay, Evaluation of Maintenance Software in Real-Time Systems, *IEEE Transactions on Computers* (June 1978).
Описывается программа реального времени, устанавливающая местонахождение ошибок путем «посева» (преднамеренного введения) заранее известной ошибки для установления числа ошибок в программе.
12. Gelperin, Testing Maintainability, ACM SIGSOFT Software Engineering Notes, April 1979.
Предлагаются способы оценки степени сложности сопровождения программного обеспечения методом «а что, если...», ревизия стандартов, предположительный анализ (чтобы предусмотреть предстоящие изменения), оценка структуры (чтобы лучше понять состав и сложность программы), тестирование программы.
13. Gerhart, Yelowitz, Observations of Fallibility in Applications of Modern Programming Methodologies, IEEE Transactions on Software Engineering, 1976.
Описываются ошибки, содержащиеся в программах, созданных с помощью современной техники и считающихся верными. Рекомендуется: 1) более тщательный анализ полного задания; 2) применение всех возможных средств, гарантирующих надежность; 3) отказ от укоренившихся представлений о «трудной» и «легкой» программе (в легкой может быть больше возможностей для ошибки); 4) различие понятий «хорошая структура программы» и «правильная программа»; 5) понимание того факта, что новая методика не является панацеей.
14. См. [12] к гл. 2.
15. Glass, Of Flat Earths and Flowcharts, The Power of Peonage, Computing Trends, 1979.
Юмористически описывается случай, происшедший из-за того, что закончилась эра структурных схем.
16. Goodenough, Eanes, MAIDS Study – Program Testing and Diagnosis Technology, Letter Report N / 000-6-73, 1973.
Речь идет о несоответствии тестов и о важности раннего обнаружения ошибок разработки. Включено регрессионное тестирование и общее обсуждение способов тестирования.
17 Hart, The Advanced Interactive Debugging System (AIDS), ACM SIGPLAN Notices, December 1979.
Описывается символическая, независимая от языка, не нуждающаяся в заранее запланированной программе коммерческая система отладки.
18 Heninger, Specifying Software Requirements for Complex Systems: New Techniques and Their Application, Proceedings of the IEEE Specifications of Reliable Software Conference, 1979
Описывается применение формальных требований к техническим условиям в связи с сопровождением программного обеспечения, содержащего ошибки и неполадки. Даются практические подходы к решению проблемы.
19 Howden, An Evaluation of the Effectiveness of Symbolic Testing, *Software Practice and Experience* (July 1978)
Дается оценка нескольких методик отладки и их способности обнаруживать ошибки. Автор считает символическое тестирование наиболее перспективным. Рассмотрены также анализаторы эффективности проверки на разных стадиях сбоя системы и другие способы.
20. Huang, An Approach to Program Testing, ACM Computing Surveys, September 1975.
Включает учебный материал, анализирующий трудности тщательного тестирования. Предлагается в качестве решения проблемы анализатор эффективности проверки. Обсуждаются применение, достоинства и недостатки этого средства.

21. Dolotta, Haight, Mashey, The Programmer's Workbench, *The Bell System Technical Jour.* (July – August 1978).
Описывается оборудование средств, которыми располагает операционная система UNIX. Рассматриваются применение системы, проблемы внедрения и практический опыт.
22. Jackson, Principles of Program Design, Academic Press, 1975.
Дается проблемно-ориентированный подход к идеям разработки. Каждая идея подтверждается примером решения задачи. Задачи взяты из области обработки данных с использованием языка программирования Кобол. Особое внимание уделяется разработке структуры данных.
23. Kernighan, Plauger, Software Tools, Addison Wesley, 1976.
Описываются средства, их ценность для программиста и разновидности средств, которые могут быть применены.
24. См. [14] к гл. 2.
25. Kosy, Air Force Command and Control Information Processing in the 1980s, Trends on Software Technology, R-1012-PR, The Rand Corp., 1974.
Описывается эволюция и предполагаемое будущее программного обеспечения в технике командования и управления. Речь идет о перспективных методах, включая применение языков высокого уровня и обсуждение их преимуществ.
26. Lauesen, Debugging Techniques, *Software Practice and Experience* (January 1979).
Описываются десять способов улучшения отладки: отладка снизу вверх, создание средств контроля выхода, самоконтролирующиеся программы, новый подход к отбору контрольных данных, независимые тесты, приемочная проверка, конфигурационное управление тестов, запись результатов проверок, моделирование ошибки, введение учета отказов.
27. См. [17] к гл. 2.
28. См. [19] к гл. 2.
29. Linger, Mills, On the Development of Large Reliable Programs, Current Trends in Programming Methodology, Prentice-Hall, 1977.
Авторы являются сторонниками структурного программирования, считая его средством «безошибочного» кодирования. Даны примеры.
30. Miller, Software Quality Assurance, *Computer* (August 1979).
Сборник статей на тему «Опыт работы с разнообразными средствами и методами обеспечения надежности».
31. См. [21] к гл. 2.
32. См. [22] к гл. 2.
33. См. [23] к гл. 2.
34. Myers, Composite / Structured Design, Van Nostrand Reinhold, 1978.
Продолжает тему, начатую в [31] (модульность), давая объективные критерии для ее оценки. Речь идет о функционально сильных модулях и спаривании данных.
35. Ng, Young, A 1900 Fortran Post Mortem Dump System, *Software Practice and Experience* (July 1978).
Дается описание средства отладки исходного языка, которое выдает дампы на Фортране (выборочно) по окончании программы. В работу включены символически идентифицированные данные и некоторые данные из истории работы программы, а также интерфейсы с компилятором, редактором связей и анализатором дампа.
36. См. [24] к гл. 2.
37. Paige, Software Testing: Principles and Practice Using a Testing Coverage Analyzer, Transactions of the Software 77 Conference, October 1977.
Автор считает, что проверка «является лучшим способом демонстрации правильности программного обеспечения». Определяется и иллюстрируется понятие «анализатор тестов», приводятся результаты работы анализатора. Обсуждаются способы тестирования с применением анализатора.
38. Panzl, Automatic Software Test Drivers, *Computer* (April 1978).
Обсуждается регрессионное тестирование в связи с автоматической системой, предназначенной для создания и сохранения процедур тестирования («при современных методах эффективное регрессионное тестирование редко возможно»).
39 См. [25] к гл. 2.

40. См. [26] к гл. 2.
41. Department of Defense Requirements for the Programming Environment for the Common High Order language, Junuary 1979.
См. [51], где дается более поздний вариант этого документа.
42. См. [28] к гл. 2.
43. Reaser, Priesman, Gill, A Production Environment Evaluation of Interactive Programming, U.S. Army Computer Systems Command Technical Documentary Report USA CSC-AT-74-03, 1974.
Дается сравнительная оценка создания программного обеспечения в интерактивном режиме и в режиме разделения времени. Выясняется, что при работе в режиме разделения времени эффективность возрастает, а затраты снижаются.
44. Reifer, Trattner, A Glossary of Software Tools and Techniques, *Computer* (IEEE) (July 1977).
Средства создания программного обеспечения подразделяются на шесть категорий: моделирование, разработка, тестирование и оценка, операции и сопровождение, оценка работы программы, вспомогательные средства программирования. Дается список из 70 средств и их классификация в соответствии с вышеупомянутыми категориями.
45. Ritchie, Thompson, The UNIX Time-Sharing System, The Bell System Technical Jour. (July – August 1978).
Содержит описание операционной системы UNIX интерфейса пользователя и ее использования. Раскрывается понятие «трубопровода» — механизма межпроцессной передачи данных, а также описываются другие уникальные свойства системы.
46. См. [29] к гл. 2.
47 Sackman, Timesharing vs. Batch Processing, the Experimental Evidence, Proceedings of the 1968 Spring Joint Computer Conference.
Суммированы все «за» и «против» при работе в режиме разделения времени. Приводятся результаты пяти экспериментальных разработок. Результаты показывают, что 1) уменьшаются человеческие затраты; 2) возрастают затраты ЭВМ; 3) программисты предпочитают работать в режиме разделения времени.
48. Share Ad-Hoc Committee on Universal Languages, The Problem of Programming Communication with Changing Machines – A Proposed Solution, Communications of the ACM, August 1958.
Содержит самое первое из известных определений Универсального машинно-ориентированного языка (UNCOL). Универсальный машинно-ориентированный язык должен был быть промежуточным языком, на который предполагалось переводить языки высокого уровня (они тогда назывались проблемно-ориентированными языками). На основе UNCOL собирались создать все машинные языки.
49 Sites, Programming Tools: Statement Counts and Procedure Timings, SIGPLAN Notices, December 1978.
Автор призывает использовать средство анализа программного обеспечения для достижения наглядности и повышения эффективности программ.
50. Stanfield, Skrukrud, Software Acquisition Management Cuidebook, Software Maintenance Volume, System Development Corp.TM-5772 / 004 / 02, November 1977.
Дается описание методов профилактики сопровождения на всех стадиях жизненного цикла программного обеспечения. Речь идет о конкретном обеспечении в рамках работ Министерства обороны США, но те же методы применимы для любого другого программного обеспечения. Даются идеи и набор тестов для проверки программы в процессе сопровождения. Суммированы требования, указания и спецификации Министерства обороны США к сопровождению программного обеспечения.
51. Requirements for Ada Programming Support Environments, *Stoneman* (February 1980).
Приводятся рассуждения, следствием которых является перечень общих средств для языка Ада Министерства обороны США.
52. Stucki, Automatic Generation of Self-Metric Software, IEEE Symposium on Computer Software Reliability, 1973.
Собраны методы использования программы — монитор, описано средство оценки и проверки программы (PET) для сбора операционных данных во время работы

программы, а также для подсчета частоты использования операторов, максимальных и минимальных значений данных и т. д.

53. Stucki, New Directions in Automated Tools for Improving Software Quality, Current Trends in Programming Methodology, Prentice-Hall, 1977.
Описывается проверка утверждений и методы отладки, с помощью которых программа может обнаружить собственные ошибки.

54. Swanson, The Dimensions of Maintenance, Proceedings of the 2nd International Conf. on Software Engineering, 1976.
Предлагается определить теоретическую базу сопровождения программного обеспечения. Даются определения сопровождения с целью коррекции, адаптации и улучшения программного обеспечения. Предлагается состав базы данных. Рекомендуется вести дальнейшие исследования по данному вопросу.

55. Tanenbaum, Klint, Bohm, Guidelines for Software Portability, Software *Practice and Experience* (November 1978).
Описываются трудности производства портативного программного обеспечения. Они подразделены на проблемы: 1) языков программирования, 2) плавающих точек, 3) файлов, 4) средств обмена данными, 5) интерактивного терминала, 6) операционных систем, 7) архитектуры ЭВМ, 8) документации. Обсуждаются конкретные проблемы.

56. UNIX Time-sharing System, *The Bell System Technical Jour.* (July 1978).
Дается более десятка статей, посвященных возможностям, созданию и сопровождению операционной системы UNIX, которая считается наиболее перспективным образцом операционной системы. Включается обсуждение набора средств создания и сопровождения программного обеспечения, известных под названием «рабочее место сопровождающего программиста».

57. White, Program Standards Help Software Maintainability, Proceedings of the Annual Reliability and Maintainability Symposium, 1978.
Дается оценка влияния стандартов программного обеспечения на сопровождение. Делается вывод, что модульность, структурное программирование и построчное комментирование имеют свои преимущества.

58. Winograd, Beyond Programming Languages, *Communications of the ACM* (July 1979).
Автор считает, что средства современных языков программирования не соответствуют сложным задачам. Предлагается концепция языка высокого уровня, который выделяет описание известных фактов в большей степени, чем повелительные высказывания.

59. Wulf, Languages and Structured Programs, Gurrent Trends in Programming Methodology, Prentice-Hall, 1977.
Обсуждается «кризис» программного обеспечения, необходимость структурных программ и роль языков, обеспечивающих их. Особое внимание уделяется новым идеям в области языков.

60. Wegner, Programming with Ada: An Introduction by Means of Graduated Examples, Prentice-Hall, 1980.
Дается описание языка Ада Министерства обороны США. Обсуждается его содержание с точки зрения программиста. Даются примеры.

Глава 4

АДМИНИСТРАТИВНАЯ СТОРОНА СОПРОВОЖДЕНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Руководителю работ по сопровождению программного обеспечения неизбежно приходится сталкиваться с целым рядом проблем, не имеющих под собой реальной основы.

Примером может служить существующее мнение, что сопровождение — скучная, нетворческая работа, заниматься которой должны самые неквалифицированные специалисты. В то же время неквалифицированное сопровождение отрицательно скажется даже на тщательно разработанном программном обеспечении. Мнение, что руководители обычно удовлетворены качеством сопровождения программного обеспечения [23], ошибочно (сопровождение — это только часть всего проекта программного обеспечения, и его качество зависит от того, насколько хорошо проведен процесс первоначальной разработки).

Существует ряд принципиальных заблуждений, касающихся процесса сопровождения. Если спросить у руководителей, какую часть времени отнимает сопровождение у их сотрудников, многие из них ответят: «Около 20%». Они также полагают, что 80% этого времени затрачивается сопровождающим программистом на исправление ошибок в программе. Как мы уже знаем, это далеко не так.

О сопровождении программного обеспечения написано мало. Но те работы, которые имеются, достаточно убедительны. Одно исследование за другим показывают, что сопровождение отнимает 50—80% материальных затрат и времени [3, 23]. Некоторые исследования убедительно свидетельствуют о том, что исправление ошибок составляет лишь малую долю работы сопровождающего программиста — от 15 до 40% [23]. Естественно напрашивается вопрос: чем же занимается сопровождающий программист в оставшееся рабочее время? В нашей книге мы постараемся ответить на этот вопрос, как, впрочем, и на следующий за ним: что я как руководитель должен сделать, чтобы помочь сопровождающему программисту? (В работе [21] читатель найдет подробный обзор задач руководителя работ по сопровождению программного обеспечения, накопленных на основе практического опыта.)

4.1. ПЛАНИРОВАНИЕ СОПРОВОЖДЕНИЯ

При планировании работ по сопровождению необходимо учитывать индивидуальные способности всех сотрудников. Кому лучше поручить задачу увеличения срока службы уже разработанного

обеспечения? Кто лучше других психологически подготовлен к тому, чтобы взять на себя непрестижный труд сопровождающего программиста? Кто обладает гибкостью, необходимой для того, чтобы приспособиться к стилю программ? Кто понимает важность сопровождения и сумеет создать программу и подготовить документацию, которые будут иметь долгий срок службы?

Кроме того, руководитель задает много вопросов, касающихся его самого и его работы. Как я могу контролировать качество работы сопровождающего программиста? Как сделать его работу продуктивнее? Как я могу обеспечить контроль над всем процессом сопровождения? Какие этапы процесса требуют наиболее тщательного административного контроля? Какие формы должен принимать процесс сопровождения? Как уложиться в рамки отпущенных средств? Все эти вопросы сложны. И, как и любые другие, они легче решаются упорными руководителями, обладающими интуицией, основанной на их личном опыте и соответствующем образовании. Главное качество — это «интуиция, основанная на образовании». Руководитель, который до тонкостей знает работу по сопровождению, легче справится и с ее планированием.

4.1.1. ПЛАНИРОВАНИЕ ВЫСОКОКАЧЕСТВЕННОГО СОПРОВОЖДЕНИЯ. РОЛЬ ЛИЧНОСТИ

Мы уже убедились, что к сопровождающему программисту предъявляются очень высокие профессиональные требования. Подобрать для работы и настойчивых, способных к творчеству, одаренных и умеющих составлять документацию специалистов непросто. И все же успех работ во многом зависит от того, удастся ли руководителю найти сотрудников, отвечающих всем необходимым требованиям. «Основным фактором, влияющим на надежность программного обеспечения, является подбор и руководство сотрудниками, которые разрабатывают программное обеспечение и осуществляют его сопровождение» [6]. Этому вопросу нередко уделяется слишком мало внимания, в особенности если учесть хроническую нехватку квалифицированных специалистов. Когда вычислительный центр работает при постоянной нехватке сотрудников высокой квалификации, возникает искушение несколько снизить уровень предъявляемых требований и принять на работу недостаточно квалифицированных работников. Такая необходимость может возникнуть, но делать это надо осторожно, и, конечно, ни в коем случае нельзя поручать таким работникам (все равно старым или вновь принятым) работу по сопровождению. В противном случае руководитель рискует попасть в положение путешественников на остров Небывалый[1]. В этой книге мы еще не раз вернемся к тому, каким образом можно благополучно избежать проблем, связанных с

[1] Название острова из детской английской книги «Питер Пен». — *Прим. ред.*

тонкостями техники программирования. Эти проблемы стоят особенно остро при необходимости увеличить объем и повысить качество работ по сопровождению. Сопровождающий программист проводит весь день, разбирая во всех подробностях написанную кем-то программу. Хорошо ли это? Продуктивно ли он тратит время? Об этом невозможно судить, если не знать до тонкостей, что это за программа.

Мы уже говорили о том, что сопровождающие программисты затрачивают 15—40% времени на исправление ошибок в программе. Совершенно очевидно, что от того, как они расходуют оставшееся время, зависит оценка их труда. Кроме того, оценка их труда зависит и от того, как они расходуют время, отведенное на исправление ошибок.

Одна из работ [23] по сопровождению программного обеспечения дает следующее распределение времени: 17% — исправление ошибок, 18% — адаптация, 60% — усовершенствование программы, 5% — разное.

Сопровождение, направленное на усовершенствование программного обеспечения, занимает больше всего времени и является наименее понятным этапом работы сопровождающего программиста. Что это такое? Это просто процесс «улучшения качества программного обеспечения». Под этим может подразумеваться самый широкий круг улучшений, начиная от усовершенствований для пользователя (скажем, введение вновь созданного алгоритма расчета траектории в уже существующую программу запуска космического корабля) и кончая усовершенствованиями для программиста (например, структуризация ранее заброшенного отрезка программы, считавшегося недоступным для сопровождения). Это может быть и внесение исправлений и дополнений в документацию (например, сделать справочник пользователя более удобным для чтения). Как ни странно, именно здесь и расходуются средства, отпускаемые на сопровождение, и руководителю необходимо осознавать важность этого вида сопровождения. К сожалению, многие руководители (особенно те, кто не знаком с техникой программирования стремятся сократить или даже совсем уничтожить сопровождение, направленное на усовершенствование, поскольку таким образом можно сэкономить 50—80% средств. Но это опасное заблуждение. Хотя и не очень пригляден образ ученого-фанатика, стремящегося довести свое «детище» до полного совершенства и готового бесконечно улучшать его, тем не менее определенное усовершенствование программного обеспечения крайне необходимо. При этом стоимость будущих исправлений и усовершенствований постоянно значительно снижается. И, что еще важнее, в будущем это окупится за счет сокращения затрат машинного времени. В конце концов, экономия машинного времени будет определяться уровнем усовершенствования программного обеспечения. Сам факт,что руковод-

ство в целом удовлетворено сопровождением, указывает на то, что сопровождающие программисты, как правило, проводят усовершенствование на высоком профессиональном уровне.

Суть всего сказанного состоит в том, что усовершенствующее сопровождение не предполагает сокращения программного обеспечения: оно помогает сделать обеспечение более удобным с точки зрения эксплуатации и контроля. То же самое относится к адаптирующему и тестирующему сопровождению.

Итак, контроль и отчетность совершенно необходимы для повышения качества сопровождения программного обеспечения. Вопрос здесь стоит так. Какой должна быть форма отчетности о состоянии сопровождения? Насколько строгим должен быть контроль? (Методы осуществления контроля мы обсудим в следующем разделе.) Пожалуй, самым непростым для руководителя является вопрос о том, как организовать контроль и насколько жестким он должен быть. Вопрос труден, так как на него нельзя ответить однозначно. Точного ответа нет потому, что это очень индивидуальная проблема (см., например, работу [2]). Некоторые программисты с удовольствием работают только в том вычислительном центре, где административный контроль очень жесткий, — они нуждаются в том, чтобы им постоянно напоминали об ответственности. Другие, наоборот, считают административный контроль унизительным для себя — они итак обладают обостренным чувством ответственности, что обеспечивает высокий внутренний самоконтроль. Все эти нюансы порождают необходимость решить, как осуществлять руководство и контроль, придерживаясь при этом индивидуального подхода. Как правило, руководители не в состоянии найти золотой середины: часть из них руководит слишком круто, другие, наоборот, недооценивают значения контроля, и у них он почти отсутствует. Компромиссный подход желателен, когда речь идет о разработке программного обеспечения, особенно в большом учреждении. Но, когда речь идет о сопровождении, где личные качества программиста тесно связаны с результатом его работ, компромисс неуместен.

Руководителю, который стремится учитывать индивидуальные особенности своих сотрудников, следует обратить внимание на следующие факторы:

1. Необходимо понять возможности сотрудника и меру его ответственности.

2. Следует осуществлять контроль и добиваться отчета о работе в зависимости от возможностей и степени ответственности сотрудника.

3. Необходимо убедить всех сотрудников в том, что руководитель относится к ним справедливо и беспристрастно.

Соблюдение всех этих условий — нелегкая задача для руководителя.

Каким образом способный руководитель может добиться максимальной отдачи от своих сотрудников?

1. С помощью эффективного индивидуального подхода к каждому сотруднику, о чем уже говорилось выше.

2. Необходимо быть в курсе той работы, которая ими выполняется, т. е. знать, какие вносятся изменения, почему и какое влияние они окажут на работу программного обеспечения (включая знание изменяемой программы).

3. Путем контроля решаемых ими задач. Здесь подразумевается контроль за внесением изменений, о чем будет подробно сказано ниже.

4. Путем создания им необходимых условий труда и обеспечения их оборудованием, которое повышает эффективность работы сопровождающего программиста. Об этом также будет сказано ниже.

Такой подход поможет управлять усовершенствованием, адаптацией и необходимой коррекцией программ в процессе сопровождения (хотя дальновиднее было бы сделать между ними различия). При правильном руководстве все эти разновидности сопровождения являются важными и неотъемлемыми частями единого процесса. Такие рекомендации по повышению продуктивности труда сопровождающих программистов можно, пожалуй, счесть несколько субъективными. Мы здесь как бы отвечаем вопросом на вопрос. Но факт остается фактом: более объективного ответа не существует. Сопровождение всегда было и останется искусством, основанным на интуиции. Если бы существовал объективный подход к вопросам сопровождения, то, возможно, само сопровождение могло бы стать ненужным.

4.1.2. ПЛАНИРОВАНИЕ ВЫСОКОКАЧЕСТВЕННОГО СОПРОВОЖДЕНИЯ. ОТЧЕТ И ПРОВЕРКИ

В предыдущем разделе говорилось о том, что объем работ по сопровождению программного обеспечения зависит от индивидуальных возможностей сопровождающего программиста. Речь шла о внимании к личности сотрудника и о том, что отчетность и контроль должны быть организованы так, чтобы отдача была максимальной.

Тот же принцип можно положить и в основу достижения высокого качества сопровождения. Из вышесказанного ясно, что тщательно подобранный, способный и добросовестный программист при условии грамотного руководства является основным элементом продуктивного и высококачественного сопровождения. При отсутствии этого элемента задача руководителя становится весьма сложной, так как ему самому придется восполнять все пробелы.

Но задача не сводится только к тому, чтобы подобрать подходящих специалистов. Руководителю необходимо обеспечить процесс

обзора и проверки, чтобы следить за самим процессом сопровождения.

До некоторой степени это запретная тема. Сопровождающие программисты подозрительно относятся ко всякой попытке руководителя вмешаться в их работу. А руководители раздражаются, если им приходиться судить о качестве сопровождения, особенно в тех случаях, когда они считают себя недостаточно осведомленными. Таким образом, хотя во многих источниках вы найдете положительные отзывы о влиянии внешнего контроля [5, 13, 25, 39], на самом деле такой контроль редко, а точнее, почти никогда не оказывает такого влияния на сопровождение программного обеспечения.

Как же избавиться от нежелательного влияния контроля? (Совершенно очевидно, что качество программного обеспечения на стадии сопровождения не менее важно, чем его качество на стадии разработки, а значит, всякий согласится с тем, что контроль и проверка необходимы на время сопровождения.) Чтобы не травмировать сопровождающих программистов, следует поручать контроль и проверку людям одного с ними ранга. Этот процесс должен состоять в проверке того, насколько программа отвечает своему назначению, насколько эффективна ее работа; только сами сопровождающие программисты способны должным образом оценить работу друг друга. Проверка должна проходить в атмосфере доброжелательной взаимопомощи, люди должны чувствовать, что они делают общее дело. К проверке не следует допускать ни тех, кому недостает такта, ни тех, кто склонен покрывать недостатки своих коллег. Проверка не всегда должна носить формальный характер. Простейшие изменения могут вноситься самим сопровождающим программистом. Несколько более сложные изменения могут быть обсуждены двумя специалистами. И только значительные изменения следует вносить формально, в официальной обстановке.

Вот все, что имеет отношение к сопровождающему программисту. Теперь поговорим о том, что касается руководителя. Видимо, читателю уже ясно, что часто существует несоответствие между необходимостью контроля и проверки на высоком техническом уровне и уровнем технической подготовки руководителя. Руководитель, осознающий это, обычно либо 1) запрещает отчеты и проверки, либо 2) участвует в них, ограничивая их содержание рамками своих знаний, либо 3) устраняется от них. Все три подхода ошибочны.

Варианты 1 и 2 совершенно неверны. Если мы стремимся повысить качество программного обеспечения, то, несомненно, должны оценивать его, а не избегать этого. Руководитель, который гордится тем, что «работа идет по графику и в рамках бюджета», одновременно упускает из виду качественный показатель, который, конечно, не

менее важен, чем график и бюджет. Руководитель, который принимает участие в отчетах и проверках, но ограничивается лишь общими соображениями, выполняет лишь две трети работы, сводя на нет оставшуюся треть.

Вариант 3 более привлекателен, но тоже ошибочен. Конечно, если проверки касаются чисто технических вопросов, то они могут проходить без администрации. Но проверка качества сопровождения программного обеспечения требует, чтобы соображения руководства были одним из факторов оценки.

Правильнее всего для руководителя участвовать в процессе оценки, внося вклад, когда он может быть полезен, и перенимать опыт коллег-практиков, одновременно повышая свой уровень знаний, когда не может оказаться полезным [15]. В результате руководитель будет до тонкости осведомлен о качестве сопровождения, за которое несет ответственность, и одновременно будет постоянно совершенствоваться как специалист.

Качество сопровождения программного обеспечения зависит, таким образом, от тех же факторов, от которых зависит его объем, плюс правильно организованный процесс отчетов и проверок. Хороший коллектив и тщательная оценка результатов его труда есть залог успешного руководства сопровождением программного обеспечения.

4.1.3. ДРУГИЕ АСПЕКТЫ ПЛАНИРОВАНИЯ

Было бы упрощением полагать, что планирование сопровождения программного обеспечения исчерпывается лишь качественным показателем. Руководитель вынужден выполнять еще целый ряд требований. Он имеет ограниченный бюджет, который позволяет ему воспользоваться услугами ограниченного числа служащих и машинного времени. Он имеет график (свой собственный или спущенный свыше), который определяет сроки выполнения работ. Наконец, в его распоряжении находятся ЭВМ, периферийное оборудование, вспомогательные средства и исполнители, которые могут выполнять ограниченный круг обязанностей. Ему подвластна организация, которая либо усиливает качество его руководства процессом сопровождения, либо, наоборот, сводит его на нет. И кроме того, у него есть документация, которая помогает в работе по сопровождению и которую в результате следует передать пользователю.

Все эти факторы необходимо учесть при планировании — тщательно разрабатывая детали, если программное обеспечение большое и сложное, и не вдаваясь в частности, если оно является простым. Например, решая вопрос о приобретении дополнительной ЭВМ для решения задач сопровождения, возможно, придется решать вопрос о строительстве нового здания для этой ЭВМ или об организации работы без увеличения ресурсов [15].

Главное в решении подобных вопросов — перспективное планирование. Все соображения качества, количества, ресурсов, бюджета, графика, организации и документации должны быть учтены задолго до начала работы. Конечно, такое планирование — нелегкая задача. Вспомним, что управление программным обеспечением является искусством, основанным на интуиции и глубоких знаниях руководителя. Помните также, что оценки программного обеспечения будут заведомо низкими и эффективность затрачиваемых на него усилий, включая сопровождение, будет зависеть от прозорливости руководителя, планирующего эту работу.

В этом разделе мы не пытались осветить вопросы бюджета и графика работ в основном потому, что они имеют много общего с проблемой разработки программного обеспечения. Об этом уже говорилось немало (например, [8, 11]), причем многое из сказанного там спорно (таково общественное мнение). Опыт показывает, что вопросы бюджета и графика работ должны решаться интуитивно в конкретных условиях разработки каждого проекта, причем научить человека интуитивно правильно решать такие задачи пока невозможно. В литературе описываются случаи, когда комплекс сопровождаемого программного обеспечения был первоначально укомплектован штатом в 100 человек, однако в дальнейшем потребовал дополнительно еще 41 [12]!

Вопросы организации и составления документации по сопровождению программного обеспечения решать можно и нужно. В последующих разделах эти вопросы будут изложены более подробно.

4.2. ОРГАНИЗАЦИЯ СОПРОВОЖДЕНИЯ

Мы уже говорили о том, что сопровождение программного обеспечения является составной частью процесса создания программного обеспечения в целом. Процесс исправления или внесения изменений в программное обеспечение включает этапы определения требований, проектирования, кодирования, тестирования и, конечно, следующего за ними этапа сопровождения. (Таким образом, сопровождение программного обеспечения можно рассматривать как рекурсивный процесс.) И нет ничего удивительного в том, что организация сопровождения немыслима вне связи с созданием программного обеспечения. Действительно, во многих случаях организация сопровождения ведется в тесной связи с разработкой программного обеспечения. Нередки случаи, когда сопровождение поручается программистам — разработчикам обеспечения (что бывает иногда полезно), причем они чередуют работу по сопровождению с созданием программного обеспечения (своей текущей работой). Получается что-то вроде режима разделения времени. Программист A, например, внедряет систему Z и одновременно

занимается сопровождением системы Y, которую, возможно, он же и разработал.

Существует другая организация сопровождения, которая тоже распространена и вполне приемлема. Может существовать специальное подразделение, в задачу которого входит исключительно сопровождение. Программисты *A, B,* и *C* разрабатывают программное обеспечение, проверяют его и передают сопровождающим программистам *X, Y* и *Z* для сопровождения. При этом сопровождающий программист *X* может быть ответственным за сопровождение всего программного обеспечения или какой-либо его части.

Существует и промежуточный вариант организации СПО — так называемый зонтик: подразделение, занимающееся сопровождением, находится под прикрытием подразделения, занимающегося созданием программного обеспечения. Программисты могут заниматься чистым сопровождением, или только созданием программного обеспечения, или и тем и другим, но связь между ними столь тесна, что переход от одного к другому осуществляется легко и привычно.

При выборе правильного варианта следует руководствоваться соображениями сложности программного обеспечения и степенью квалификации коллектива программистов. Если все сотрудники презрительно относятся к сопровождению, разумнее всего, пожалуй, распределять обязанности, связанные с ним, равномерно между всеми членами коллектива. Если же среди них есть люди, которым нравится заниматься сопровождением и которые обладают необходимыми для этого способностями, целесообразнее выделить их в специальное подразделение, отвечающее за сопровождение программного обеспечения. Кроме соображений, связанных с личными качествами работников, существуют соображения, которые следует учесть: сложность программного обеспечения и приоритет сопровождения. Если программное обеспечение очень сложное (например, большая операционная система или программа, управляющая работой цеха), имеет смысл привлечь к сопровождению тех специалистов, которые создали ее. При этих условиях лучше работать в режиме разделения времени. (Вполне вероятно, что программисты, создавшие данную сложную программу, уже перешли к созданию новой программы, но они должны по-прежнему нести ответственность за сопровождение предыдущего программного обеспечения.)

Но если сопровождение обладает высоким приоритетом (например, речь идет о системе расчета заработной платы или о системе, работающей в режиме реального времени), будет правильнее поручить сопровождение специальному подразделению, чтобы оно занималось только этим делом.

Совершенно ясно, что существует промежуточный вариант — большая степень сложности плюс высокий приоритет, о котором мы

ничего не сказали. Соображения сложности заставляют поручить работу по сопровождению разработчику, высокий же приоритет требует обратного — поручить сопровождение специально выделенному для этого подразделению! Как найти выход из положения? Решение этого вопроса мы оставим за руководителями, которые будут читать нашу книгу. Это и есть превосходный случай поупражняться в решении тех задач, для которых необходима интуиция, основанная на глубоких знаниях. Об этом мы говорили выше.

В нашей книге мы не будем пытаться давать определения возможных вариантов организационных структур. Вместо этого рассмотрим те характерные организационные требования, которые налагаются самой работой по сопровождению, и свяжем их, где это необходимо, с организационной структурой разработки или сопровождения. Например, мы не станем рассматривать различия между организационными структурами, в основу которых положены требования технологии (например, группы главного программиста [5, 39]), и организациями, в основу которых положены требования администрации (например, традиционная иерархическая структура [6, 12, 30, 35]). Те, кого интересуют эти вопросы, могут воспользоваться приведенной в конце главы справочной литературой.

Далее будут даны рекомендации по организации, которые следует учесть при сопровождении программного обеспечения. Мы исходим из выполняемых программным обеспечением функций. Обсуждение завершится определением нескольких возможных организационных структур.

4.2.1. СОВЕТ ПО ВНЕСЕНИЮ ИЗМЕНЕНИЙ

Самый первый вопрос сопровождения — это: Нужно ли вносить данное изменение? Решение этого вопроса может оказаться как совершенно очевидным (например, если при выплате зарплаты каждый чек содержит лишнюю тысячу долларов), так и чрезвычайно сложным (если, например, пользователь хочет увеличить число символов в строке с 30 до 35, это повлияет на файлы, форматы, описание внутренних данных и процедур и, следовательно, на затраты; поэтому необходимо рассмотреть вопрос о целесообразности такого изменения). Заметим также, что сложность вопроса не соответствует его важности. В приведенном ранее примере вопрос, решение которого очевидно, является одновременно и более важным, в то время как важность решения сложного вопроса не очевидна.

При решении вопроса о целесообразности изменения возникает извечная дилемма между срочностью принятия решения и той ответственностью, которую при этом приходится брать на себя, если решение сложно. В примере с зарплатой промедление могло бы дорого обойтись. В то же время поспешность при решении вопроса

об увеличении числа символов также может обернуться большими убытками.

Отсутствие четкого разделения функций в организационной структуре приводит к путанице при решении сложных вопросов, связанных с изменениями. Легче всего создать совет по внесению изменений, отвести ему определенное место в организационной структуре и считать вопрос решенным. Во многих сложных вопросах, для решения которых требуется большая решительность, такой вариант подходит. Однако хорошо известно, как неповоротлива наша иерархическая структура. В тех случаях, когда требуется принять срочное решение, созыв совета по внесению изменений может сильно затормозить работу. И все же, несмотря на это, совет по внесению изменений является необходимым органом, который должен иметь определенное, официально установленное место в организационной структуре. Его полномочия и мера ответственности должны быть четко определены соответствующей инструкцией, также должен быть предусмотрен образ действий в тех случаях, когда решение необходимо принять срочно.

Внесение изменений в сопровождение программного обеспечения может повлечь за собой далеко идущие последствия для всего учреждения. Поэтому совет изменений должен действовать на самом высоком уровне. В нем должны быть представлены и те отделы, которые не связаны непосредственно с программным обеспечением, либо на правах постоянных членов, либо в качестве приглашенных на заседания в тех случаях, когда обсуждается внесение изменений, затрагивающих их интересы. Например, если вносятся изменения в программное обеспечение тренажера для экипажа самолета, это может затронуть интересы организации, занимающейся конструированием и созданием самолета, а возможно, и интересы самого экипажа. Они обязательно должны принять участие в обсуждении. Обсуждение же внутренних изменений (например, устранение ошибок в программе), которые не затрагивают интересов других отделов, скорее всего не будет для них интересно. В любом случае участие в решении вопросов руководителей самого высокого уровня может оказаться полезным, обсуждается ли распределение дефицитного времени ЭВМ или речь идет о взаимодействии отделов.

Опять-таки такое участие вышестоящих руководителей не всегда необходимо и желательно. Всем хорошо знакома ситуация, когда в 2 ч дня во время работы программы, рассчитывающей зарплату, вызывают сопровождающего программиста, чтобы устранить возникшую неполадку. Высших руководителей, естественно, привлекать к решению подобных вопросов не следует. Таким образом, процесс внесения изменений следует организовать так, чтобы в аварийных ситуациях в подчиненную организацию направлялось ответственное лицо для решения вопроса об изменениях. Трудно, а может быть, и невозможно точно определить, при каких обстоятель-

ствах следует вносить такие срочные изменения. Лучше всего выработать систему отчетности перед советом по внесению незапланированных изменений.

Все сказанное выше относилось к процессу формального решения вопроса о целесообразности изменений. Существует, однако, много случаев, когда формальный подход не только не нужен, но даже вреден. Например, в очень маленькой организации или в случае, когда крупная организация ведет небольшую разработку, часто наличие способного, думающего сопровождающего программиста оказывается достаточным для того, чтобы избежать формального рассмотрения вопроса об изменениях. В этой главе мы хотим дать читателю представление о самом понятии «решение вопроса о целесообразности изменений». Степень формальности при решении этого вопроса должна быть установлена самим руководителем. Мы уже говорили об интуиции руководителя, основанной на прочных знаниях. Она заключается в способности быстро решать нелегкие задачи. Одной из таких задач и является вопрос о том, когда, как и к кому следует применить формальный подход.

4.2.2. КОНТРОЛЬ НАД РАБОТОЙ ПО ВНЕСЕНИЮ ИЗМЕНЕНИЙ

Руководители часто недостаточно следят за процессом внесения изменений в программное обеспечение. Однако им важно было бы знать следующее. Что послужило причиной изменения — ошибки в программе или усовершенствования в интересах пользователя? Какие именно изменения были сделаны верно, проверены и закончены? Какие изменения были внесены ошибочно и требуют переделки? Какие изменения явились лишь временным выходом из положения? Над какими изменениями работают сейчас? В какие сроки предполагается закончить эту работу, и если работа уже закончена, то когда? (Или, что еще важнее, какой из вариантов программного обеспечения содержит эти изменения?) Какие существуют возможные выходы для сопровождающих программистов и пользователей, зашедших в процессе внесения изменений в тупик? Увеличивается ли объем изменений или уменьшается? (Это ключ к ответу на еще более важный вопрос: улучшается или ухудшается программное обеспечение? Увеличивается или уменьшается количество неустраненных неполадок? (Это позволит судить о том, достаточна ли квалификация коллектива сопровождающих программистов, чтобы справиться с возложенной на него работой.)

Для того чтобы получить ответы на все эти вопросы, руководителю следует иметь достаточно простую и четкую систему учета изменений, вносимых в программное обеспечение. В основу такой системы следует положить два организационных соображения. Первое — применять какой-либо стандартный бланк, позволяющий отразить вносимые изменения. Примером такого бланка может

ОТЧЕТ ОБ ИЗМЕНЕНИЯХ В ПРОГРАММНОМ ОБЕСПЕЧЕНИИ

Проблемный отчет №________

Название темы ________ ЭВМ/лаб. исполнитель ________ Программа ________

Обнаружение ошибки | Кто обнаружил ошибку ________ Дата ________

Метод обнаружения
- ☐ - Выполнение программы
- ☐ - Пересмотр/анализ программы
- ☐ - Тест на этапе разработки
- ☐ - Тест на этапе интеграции
- ☐ - Тест на этапе приемки системы

Средства, используемые при обнаружении
- ☐ - Никаких
- ☐ - Проверка на этапе проектир.
- ☐ - Визуальная проверка распечатки пр-мы
- ☐ - Дамп (изменений)
- ☐ - Дамп-контроль (с помощью терминала)
- ☐ - Отладка ЯВУ
- ☐ - Анализатор
- ☐ - Моделирование
- ☐ - Утверждение
- ☐ - Доказательство
- ☐ - Другие средства

Описание признаков ________

Конфигурационный уровень ________

Важность/необходимость исправления (дата) ________

Подпись руководителя ________ Организация ________ Дата ________

Анализ ошибки | Кем проведен анализ ________ (Ф.И.О.) Дата начала ________ Дата завершения ________

Обнаружено ________

Затрачены ресурсы: Кол-во человеко-часов ________ Время счета ________

Оценка затрат на исправление: Кол-во человеко-часов ________ Время счета ________

Предлагаемый выход ________

Устранение ошибки | Кем устранена ________ (Ф.И.О.) Дата начала ________ Дата завершения ________

Описание исправления ________

Измененные компоненты программы и конфигурационный уровень ________

Затрачены ресурсы: Кол-во человеко-часов ________ Время счета ________

Характер ошибки:
- ☐ - Язык управления заданиями
- ☐ - Операционный интерфейс
- ☐ - Ошибка программирования - декларация данных
- ☐ - Ошибка программирования - инструкция выполнения
- ☐ - Ошибка проектирования - пропуск логическово оператора
- ☐ - Ошибка проектирования - нарушение логики
- ☐ - Тестирование
- ☐ - Конфигурационное управление
- ☐ - Документация
- ☐ - Разное
- ☐ - Другие ошибки

Подпись последней инстанции ________ Организация ________ Дата ________

Рис. 4.2.2-1. Предлагаемая форма бланка отчета об изменениях в программном обеспечении.

ОТЧЕТ О СОСТОЯНИИ ИЗМЕНЕНИЙ В ПРОГРАММНОМ ОБЕСПЕЧЕНИИ СИСТЕМЫ

Номер ОИПО	Первоначальный вариант	Исправленный вариант	Версия	Описание
25	9/20/80	10/20/80	4.7 ?	В момент, когда система закончила выполнение процедуры, прозвучал звуковой сигнал
26	9/20/80	10/17/80	4.7?	Нарушена непосредственная связь с библиотекой стандартных подпрограмм
27	9/21/80	10/22/80	4.6	Отсутствует запись работы библиотеки
28	9/21/80	10/20/80	4.7?	Проверка неработающей команды
29	9/23/80	9/23/80	4.6	Поступило сообщение: не работает оператор

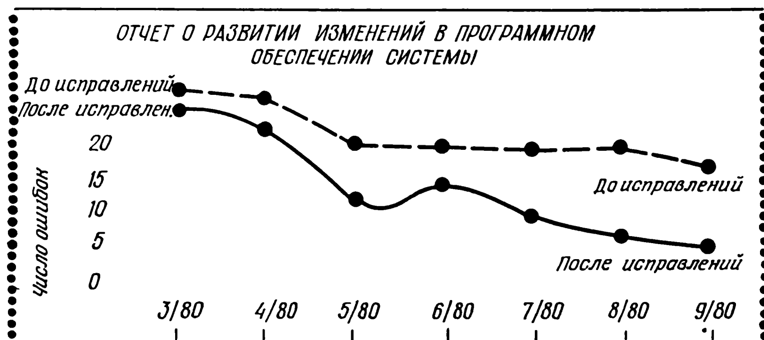

Рис. 4.2.2-2. Образец отчета о внесении изменений.

быть «Отчет об изменениях в программном обеспечении» (ОИПО), который будет содержать данные как о природе ошибок, так и о способе ее устранения. Второе — вести «Отчет о состоянии изменений в ПО системы» или «Отчет о развитии изменений в ПО системы» на основе данных файла текущих изменений. Такие отчеты дают ответы на приведенные выше вопросы. Примеры такого бланка и таких отчетов даются соответственно на рис. 4.2.2-1 и 4.2.2-2. Более подробно эта тема обсуждается в работе [7].

Важно не только вести учет и писать отчеты о возникающих неполадках. Не менее важно знать, в каких случаях и как долго следует вести этот учет.

Обычно в жизненном цикле программного обеспечения огромное множество ошибок бывает обнаружено еще до того, как возникает необходимость их формального учета. Например, ошибки, возникающие на стадиях анализа и проектирования, обычно не фиксируются, так как они устраняются на этапе определения спецификаций и реализации по мере их обнаружения. Даже на стадии проверки учет

ошибок из-за большого количества обычно не ведется и часто их можно устранить быстрее, чем зафиксировать. Формальная система учета становится необходимой лишь во время приемочной проверки, или передачи системы пользователю, или интеграции системы.

Причем не столь важно, каким образом организованы учет и контроль за внесением изменений, важно, чтобы они были организованы. Во многих случаях лучше всего этот учет осуществляет сам сопровождающий программист. Однако в больших или сложных системах, особенно там, где за сопровождение отвечает большая группа сотрудников, целесообразно иметь специальное подразделение, которое занималось бы составлением отчетов, черпая необходимые сведения из файла изменений программного обеспечения.

4.2.2.1. Отчеты об изменениях в программном обеспечении

Независимо от того, на кого возложена организация сопровождения, после запуска системы следует составить «Отчет об изменениях в программном обеспечении» (ОИПО) и присвоить ему порядковый номер. В ОИПО должны быть ясно описаны и обоснованы признаки ошибки, а также дана информация, необходимая и достаточная для устранения ошибки.

Возможны следующие типы ошибок, которые следует включать в отчет:

1. Функциональная неисправность программного обеспечения (встроенного или интерфейсного).
2. Ошибка в документации.
3. Неэффективность программного обеспечения.
4. Ошибка при выполнении теста (процедуры).

Необходимо установить приоритет устранения ошибки. При этом следует принимать во внимание различные факторы. Например, повлияет ли ошибка на работу программы? Какова возможность ее устранения? В какой степени ошибка тормозит работу пользователя? Следует указать относительную важность самой работы. Приоритет необходимо незамедлительно внести в бланк ОИПО.

Как только бланк ОИПО заполнен, его следует передать соответствующей организации, занимающейся программным обеспечением. Сопровождающий программист должен ознакомиться с ОИПО, чтобы установить, имеет ли этот отчет отношение к отрезку программы, над которым он сейчас работает, и определить приоритет устранения ошибки. Затем ОИПО нужно поместить в папку в соответствии с установленным приоритетом. Приступая к работе над ОИПО, сопровождающий программист анализирует ошибку и определяет возможный путь ее устранения. Результат анализа отмечается в бланке ОИПО. Затем характер исправлений представляется на рассмотрение совету по внесению изменений.

Как только совет выносит одобрительное решение, изменение кодируется и включается в программу. Исправленная программа, включающая исправления по одной или, возможно, нескольким ошибкам, затем прогоняется при тех же условиях, при которых она давала сбой. Если программа выдерживает эту проверку, она подвергается регрессивной проверке, имеющей целью установить, не были ли внесены новые ошибки в процессе исправления. Если любой из этих тестов дает отрицательный результат, следует вновь разработать исправление, вновь кодировать и вновь проверить его. Если же все тесты прошли успешно, программное обеспечение можно считать готовым к передаче пользователю (одобренный тест, листинг и т. п.). С точки зрения организации этот процесс графически показан на рис. 4.2.2.-3.

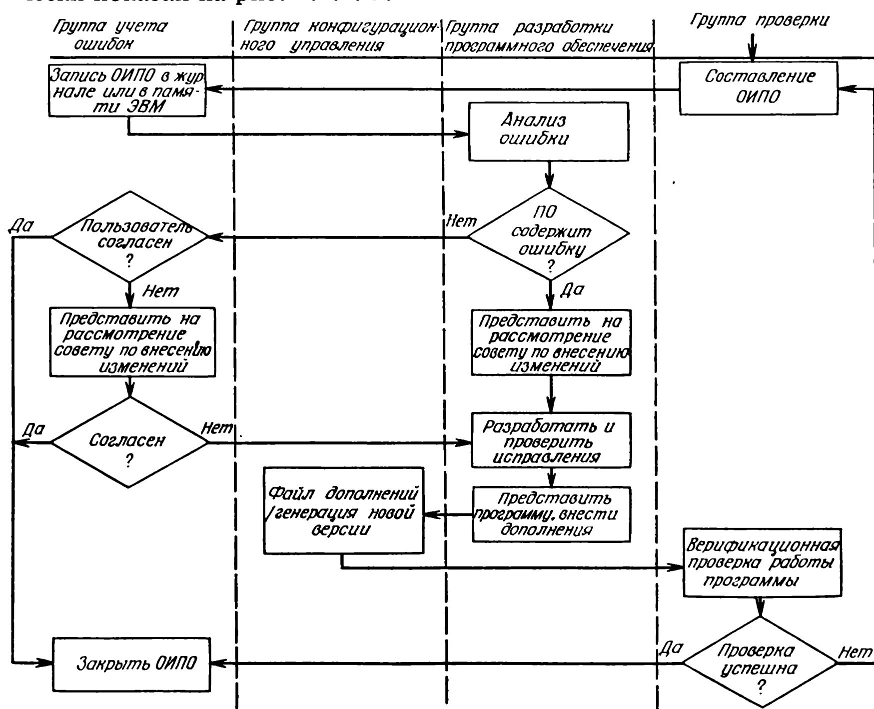

Рис. 4.2.2-3. Пример организации обработки потока ОИПО.

4.2.2.2. «Отчет о стоянии изменений в программном обеспечении системы»

Разработчики, пользователи и руководители должны постоянно быть в курсе каждого отдельного изменения и состояния программного обеспечения в целом. Для этого нужен отчет о состоянии изменений.

«Отчет о состоянии изменений» составляется на основе информации, содержащейся в файле ОИПО. В отчете необходимо в соответствии с приоритетом отразить следующее:

1. Вновь возникшие ошибки, их природу и способы устранения.

2. Старые неисправленные ошибки, их природу и возможные способы исправления.

3. Ошибки, исправленные за период, истекший с момента составления предыдущего отчета, их природу и способы устранения.

4. Данные о развитии изменений — графа, содержащая подсчет количества ответов о нерешенных проблемах за все время, а также сведения о частоте поступления таких отчетов.

«Отчет о состоянии изменений» должен давать представление о любой ошибке в программном обеспечении, а также возможность проверить, нет ли такой ошибки в списке возможных неисправностей. Отчет должен наглядно показывать, как продвигается процесс устранения ошибок.

Очень важным является вопрос распределения «Отчетов о состоянии изменений». Совершенно очевидно, что содержащаяся в них информация необходима для руководителя, чтобы он мог составить представление о положении дел, а также о профессиональном уровне и качестве работы сопровождающих программистов. Не менее важна эта информация и для пользователей, которым интересно узнать, каким образом сопровождающий программист ликвидирует возникающие неисправности, а также когда именно коррекция будет закончена. Пользователям также необходимо знать, какие изменения внесены и в каком варианте программы.

«Отчеты о вносимых изменениях» являются важным связующим звеном между руководителем и сопровождающим программистом. Они также связывают сопровождающего программиста с внешними организациями.

4.2.3. АТТЕСТАЦИЯ

В сопровождении программного обеспечения есть проблема, существование которой не всегда признают даже опытные руководители и программисты. Часто внесение даже одного изменения в программное обеспечение расстраивает ранее идеально работающую программу. Таким образом, в задачу сопровождения входит не только внесение изменений в программу, но и забота о том, чтобы оно не привело к возникновению новых ошибок. По этой причине аттестация измененного программного обеспечения особенно важна. Очевидно, что измененное программное обеспечение, прежде чем оно будет возвращено пользователю, должно пройти своеобразную проверку/аттестацию, подтверждающую его работоспособность. Назовем ее верификацией. Более того, такой же проверке следует подвергать и те части программного обеспечения, в которые не вносились изменения.

5—1387

Набор тестов должен охватывать весь процесс сопровождения программного обеспечения и осуществлять как проверку измененных частей программы, так и выявление регрессионных ошибок (т. е. ошибок в одной части программы, вызванных несогласованными изменениями в какой-либо другой ее части). Однако, поскольку такое тестирование необходимо программе уже тогда, когда она впервые написана, верификация относится не только к процессу сопровождения. По этой причине мы не станем обсуждать ее здесь подробно.

Однако у верификации есть некоторые части, относящиеся только к сопровождению. Дело в том, что к моменту начала сопровождения в программном обеспечении уже имеется набор тестов и других средств, необходимых для верификации. Они были созданы на стадиях его разработки и приемочных испытаний. Все эти средства контроля следует максимально использовать на стадии сопровождения, во-первых, для того, чтобы сократить объем работ по разработке новых тестов, а во-вторых, чтобы убедиться в том, что сопровождаемая программа подвергалась ранее не менее тщательной проверке. Средства контроля должны включать верификацию, позволяющую убедиться в том, что программа по-прежнему удовлетворяет требованиям по времени и объему занимаемой памяти, а также проверку надежности, которая показывает, выполняет ли программа свои функции или нет.

Кроме того, следует позаботиться об обнаружении регрессионных ошибок, о которых упоминалось выше. Вообще говоря, это не составляет труда, так как первоначально разработанный набор тестов достаточно велик, чтобы осуществить проверку программного обеспечения независимо от текущих изменений. Несмотря на это, существующий набор тестов в процессе сопровождения следует пополнять, включая специальные тесты, направленные на оценку самых последних изменений. Нередко обнаруживается, что $(n + 1)$-е изменение уничтожает n-е. Такие специальные тесты могут выявить эти ошибки.

Пожалуй, самым важным для процесса проверки программного обеспечения (ПО) является то, что он должен быть повторяющимся. Проверка работоспособности программы должна быть неоднократной, так как один и тот же тест может пройти на стадии разработки и давать сбои на последующих стадиях из-за внесенных в программу изменений. Поэтому важно, чтобы тесты легко прогонялись, а их результаты просто анализировались.

В дополнение к анализу результатов тестирования разработаны две основные методики контроля. Первая представляет собой самоконтроль программного обеспечения. В этом случае не только проверяется наличие ошибок в ПО, но и производится обнаружение этих ошибок и выдача сообщений о них. Такая методика широко пр ме яется для проверки компиляторов и представляет собой

программу, которая выполняется после процесса компиляции с целью проверки ее работоспособности и причины сбоя.

Вторая методика контроля основана на сравнении файлов. В том случае, если самоконтроль применить трудно или невозможно, подготавливают специальный файл, содержащий правильные результаты тестов. Сразу же после выполнения теста полученные результаты автоматически сравниваются с правильными ответами, содержащимися в файле с помощью программы сравнения файлов (разд. 3.2.1.1). Она может быть либо программой общего назначения, используемой для сравнения исходных файлов, либо программой специального назначения, созданной для выделения и объяснения не только самих сбоев, но и их возможных причин в зависимости от целей теста.

Внимательный читатель, вероятно, заметил, что в контексте существует сходство между понятиями «аттестация программного обеспечения» и «организация разработки программного обеспечения, именуемого тестированием» [29, 35]. Действительно, эти два понятия очень близки, так как их функции и назначение практически совпадают. Нередко аттестация программного обеспечения является составной частью методики проверки результата сопровождения, если таковая существует. И возможно, это наиболее правильное решение. С другой стороны, в организациях, где проверка результата сопровождения не производится, выполнение аттестации программного обеспечения может осуществляться по той же схеме, по которой она проводилась на стадии разработки. Так же как и выше, в этой главе основная цель состоит в том, чтобы обеспечить проведение аттестации, а не в определении ее места в организационной структуре.

4.2.4. КОНФИГУРАЦИОННОЕ УПРАВЛЕНИЕ

Преимущества и проблемы конфигурационного управления относятся не только к сопровождению. Однако здесь имеет место дополнительная задача, о которой следует поговорить отдельно. Это задача управления версиями.

По мере того как программное обеспечение претерпевает изменения, пользователь получает все новые и новые версии программного обеспечения. Идеальным был бы такой вариант, когда каждая новая версия полностью вытесняла бы предыдущую. Но, как все мы прекрасно знаем, жизнь далека от идеала. (Об этом говорилось в разд. 3.2.1.) На самом деле происходит примерно так:

1. Данная программа используется сразу во многих учреждениях. Не все новые версии достигают этих учреждений одновременно и не все они одновременно применяются, несмотря на настойчивые требования сопровождающего программиста. В результате случается неизбежное: сообщение об ошибке или изменении накладывается

на предыдущую версию системы и сопровождающего программиста могут попросить внести изменение в одну из устаревших версий программы. Таким образом, в рамках конфигурационного управления приходится заниматься еще одним видом работ — составлением библиотеки всех когда-либо существовавших версий с набором соответствующих документов. Совершенно ясно, что такое положение недопустимо. И здесь требуется принимать скорее административные меры, чем затрачивать усилия технического персонала. Тем не менее в тех случаях, когда выхода нет, задачу приходится решать сопровождающим программистам, и ее решение должно быть найдено.

2. Версия $n + 1$, переданная пользователю, позволила исключить многие ошибки в программном обеспечении системы, однако в свою очередь привела к возникновению дополнительной ошибки, которой не было в версии n. На время, пока пострадавший пользователь дожидается устранения ошибки в версии $n + 2$, ему нужен какой-нибудь временный выход, чтобы продолжить работу. Можно, конечно, вернуться снова к версии n, которая все же позволяла получать результаты счета. Но в этом случае получается ситуация, описанная в п. 1.

Выход один: сопровождение необходимо организовать так, чтобы постоянно работало несколько версий.

В результате можно сделать вывод, что конфигурационное управление в каждом отдельном случае должно обеспечить работу не только одной завершенной версии программного обеспечения, но и целого набора версий и каждая из них должна выдавать результаты, за достоверность которых отвечает сопровождающий программист. (Заметим, что этот набор может пополняться не только за счет самых новых версий.) И опять вопрос о том, должно ли конфигурационное управление быть выделено в отдельное функциональное звено в организационной структуре сопровождения или ответственность за него должен нести каждый сопровождающий программист, не так уж важен. Главное, чтобы конфигурационное управление осуществлялось.

Все известные методы конфигурационного управления, применяемые на стадии разработки [29, 31], применимы и для сопровождения, причем на стадии сопровождения они расширяются и углубляются. Практически для многих случаев конфигурационное управление нужно только на стадии сопровождения.

Интересное обсуждение автоматизированного подхода к данной проблеме дается в работе [10].

4.2.5. ВОЗМОЖНЫЕ ОРГАНИЗАЦИОННЫЕ СТРУКТУРЫ СОПРОВОЖДЕНИЯ

В предыдущих разделах мы преимущественно уделили внимание обеспечению функциональных возможностей сопровождения, а не

его организационной структуре. Очевидно, что разные руководители являются горячими сторонниками (или противниками) различных, часто противоположных организационных структур обеспечения. Нередко случается, что в дневнике одного руководителя содержатся самые лестные отзывы о программно-ориентированной организации сопровождения, а с другой с еще большим жаром отстаивает достоинства его функциональной организации. На самом же деле все зависит от особенностей характера и индивидуального стиля работы данного руководителя.

Задача этого раздела состоит в том, чтобы предложить читателю несколько возможных способов организации процесса сопровожде-

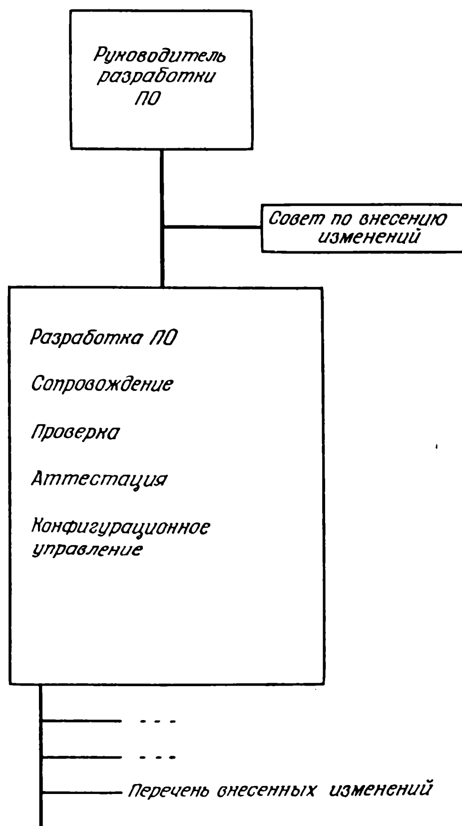

Рис. 4.2.5-1. Организационная структура работы в режиме разделения времени: СПО производится той же организацией, которая проводит разработку программного обеспечения. Нередко эта работа делается одними и теми же сотрудниками.

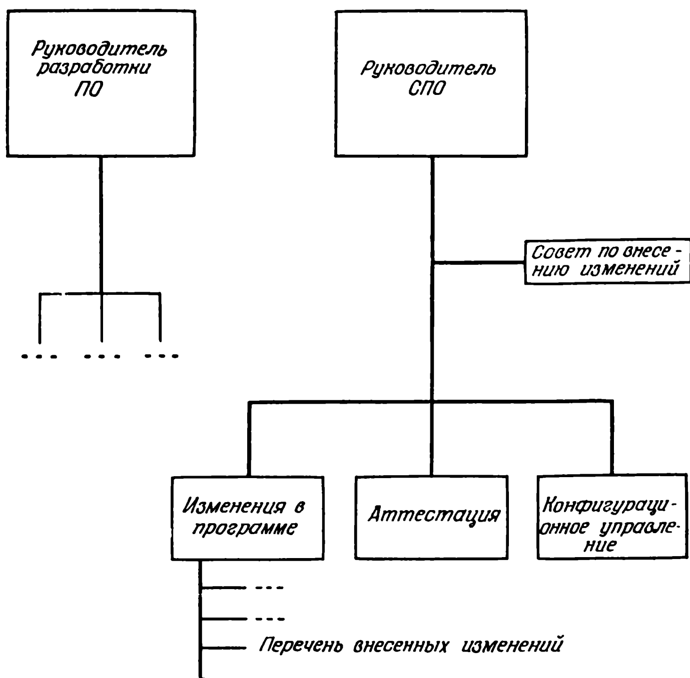

Рис. 4.2.5-2. Организационная структура «Двуглавый Дракон»: СПО производится в другой организации, независимой от той, где было разработано ПО.

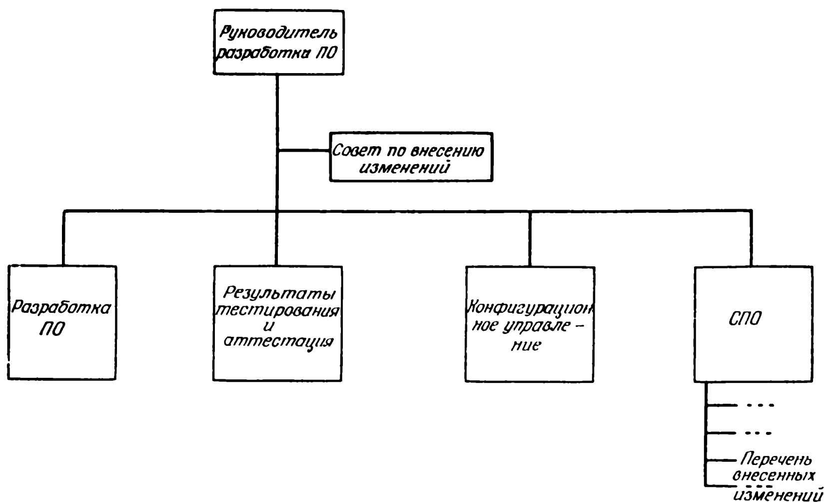

Рис. 4.2.5-3. Организационная структура «Зонтик»: СПО подчинено тому же руководителю, что и разработка ПО, но производится одной или более независимыми организациями.

ния программного обеспечения. Мы не претендуем на то, что эти способы наилучшие. И с учетом этой оговорки предлагаем вниманию читателей рис. 4.2.5-1 — 4.2.5-3.

4.3. ДОКУМЕНТАЦИЯ К СОПРОВОЖДЕНИЮ

В последние годы в области документации программного обеспечения наблюдается любопытное явление. Все инструкции по применению программного обеспечения, а также документация по сопровождению программного обеспечения представляют собой серию томов, число которых для больших систем нередко очень велико. В них дается обзор системы и объяснение каждой части ее структуры и базы данных; описываются потоки и способы управления данными, которые участвуют в работе системы; описывается каждая из ее основных функций; функции разбиваются на процедуры, каждая из которых описывается вместе с ее локальной базой данных; дается таблица перекрестных ссылок и справочник глобальных переменных.

Вся эта информация необходима. Но любопытно, что она содержится в томах документации отдельно от программы. Такой подход *совершенно неверен*! Эта информация должна содержаться в распечатке самой программы, что объясняется целым рядом причин. Хотя известны также и причины, которые привели нас к ошибочному решению.

4.3.1. ПОЧЕМУ МЫ ЗАБЛУЖДАЛИСЬ?

Программисты вообще не любят документацию. Похоже, что наличие черт характера, необходимых для того, чтобы быть программистом, исключает способность и желание составлять в словесной форме описания созданных программ.

Эта проблема, появившаяся одновременно с самой профессией программиста в 50-е годы, должна лежать в основе любого обсуждения проблемы документации. Здесь, по-видимому, от руководителя требуется «политика кнута и пряника» независимо от того, как бы глубоко программисты ни осознавали необходимость ведения документации. Если предоставить программисту возможность заняться еще девятью делами, то он обязательно отложит документацию на десятое место. А если это так, то руководитель ни в коем случае не может оставить ведение документации без контроля. Чтобы добиться высокого технического уровня документации, руководителю необходимо постоянно уделять этому вопросу внимание.

Следует рассмотреть здесь и еще один немаловажный фактор. Сами руководители обычно не любят вникать в детали программирования. Возможно, это был основной мотив, по которому человек оставил программирование и занялся административной работой.

Однако и сам характер работы руководителя часто не позволяет вникать в подробности выбора инструкций в Ассемблере или типов операторов в ЯВУ. К сожалению, необходимость заниматься общими вопросами постепенно уводит руководителя в сторону от программирования, так что в конце концов он перестает понимать смысл тех программ, за работу которых несет ответственность. И все же, какой бы критерий ни избрал руководитель при оценке «качества сопровождения», его практически невозможно оценить, не вникая во все детали. А это значит, что надо уметь читать распечатки программ.

Но чтение распечаток программ и есть то первое, от чего отказывается человек, как только его выдвинут на руководящую должность. Итак, складывается критическая ситуация: *те, кто несет наибольшую ответственность за качество сопровождения, оказываются наименее способными оценить его.*

Совершенно ясно, что эта проблема так или иначе затрагивает все области мира программного обеспечения. Но в данном разделе нас интересует, каким образом эта проблема затрагивает документацию программного обеспечения.

Во всяком случае компромисс между неприязнью программистов к документации и нежеланием руководителей читать программы неизбежно приводит к плохим результатам. Поскольку программисты не берут на себя ответственность за ведение документации, ее берут на себя руководители. А если это так, то они ни за что не согласятся поместить документацию в программу, потому что тогда они не смогут ее читать. В результате во всех организациях документация пишется в виде текста и хранится в томах, отдельно от программы.

4.3.2. ЧЕМ ПЛОХА ОТДЕЛЬНАЯ ОТ ПРОГРАММЫ ДОКУМЕНТАЦИЯ?

Существуют две основные причины, по которым необходимо вести сопроводительную документацию. Первая и главная из них — это дать возможность сопровождающим программистам, отвечающим за систему, разобраться в ней. Вторая причина, имеющая меньшее значение в данном контексте, — это обеспечить приемочные испытания.

Мы сосредоточим внимание на первой. Вторая играет роль при подписании ответственного заказа. И в этом случае обычно бывает достаточно информации этапа проектирования программного обеспечения. О влиянии такой описательной информации высокого уровня на сопроводительную (внутреннюю) документацию мы поговорим ниже.

Выше мы уже упоминали о том, что внутренняя документация ведется для того, чтобы ею пользовались сопровождающие программисты. Именно это отличает ее от любой другой

документации. Справочники пользователя, например, предназначаются для пользователей. Спецификации — для заказчиков (чтобы они удостоверились, что решаемая задача именно та, которая им необходима) и для разработчиков. Документация по тестам пишется для заказчиков (чтобы они убедились, что необходимая надежность обеспечена) и для специалистов, контролирующих работу программного обеспечения.

Верно и то, что, за исключением справочников пользователя, внутренние справочники меньше всего подвержены изменениям и поэтому являются очень важным видом документации к сопровождению. В то время как, например, спецификации и документация по тестам указывают на конкретное событие (пересмотр требований или приемочное испытание), а затем становятся достоянием истории, внутренние справочники (наряду со справочниками пользователя) остаются нужными до тех пор, пока функционирует соответствующая система.

Однако, стремясь поскорее сдать документацию, часто забывают об интересах тех, для кого она предназначена, и о ее долгосрочном использовании. И как только документация одобрена, о ней вообще забывают окончательно.

В результате складывается положение, хорошо известное большинству программистов. *Основная часть внутренней документации не отвечает своему назначению и, что еще хуже, часто оказывается устаревшей, а следовательно, ненадежной.* Такое ставшее привычным отношение к документации недопустимо.

4.3.3. КАК ПРАВИЛЬНО ВЫЙТИ ИЗ ЭТОГО ПОЛОЖЕНИЯ?

А как же выходят из положения программисты, когда им надо пересмотреть кусок программы? Они обращаются к листингу. Листинг всегда точен, поскольку это программа в полном смысле слова. По той же причине он содержит максимальный объем информации. Итак, единственно точным и полным документом на современном этапе часто является листинг. Это и есть единственно разумное решение проблемы внутренней документации к программному обеспечению. Она должна быть удобно читаемой программой с комментариями в листинге. При таком решении все пояснения к работе программы находятся в закодированном виде в самой программе. Если в программу вносятся изменения, то вероятно, что изменения будут внесены и в ее описание (хотя полной гарантии здесь нет!). Кроме того, пояснения в листинге скорее всего будут легко восприниматься именно той категорией людей, для которых они предназначались, т. е. сопровождающими программистами. Причем они будут находиться в той части программы, где программисту удобнее всего будет найти их. Полнота и точность листинга, несомненно, положительно скажутся на документации.

Некоторые новейшие течения в современной вычислительной технике отражают следующую точку зрения: структурное программирование позволяет создавать программы, которые на логическом уровне иногда бывают самодокументирующимися. В особенности это осуществимо в языках с адекватными управляющими структурами, в которых системы индентации (indentation) позволяют с помощью графических средств облегчить понимание программы. (Из этого не следует, однако, что работу и смысл этих программ можно понять без комментариев). Кроме того, языки и загрузчики, обеспечивающие использование удобных для чтения и самодокументирующихся имен, делают еще больший шаг к самодокументирующемуся программированию. Например, DOVT лишено всякого смысла и выглядит загадочно, тогда как самоопределяющееся имя процедуры DO VALIDITY TEST означает «провести проверку достоверности». (Заметим, однако, что достоинства здесь легко могут обернуться недостатками, например, если из-за невнимательности опустить самоопределяющееся имя или если нарушить правило перехода от него к мнемоническому имени. Заметим также, что большинство современных загрузчиков ограничивает возможность внешних ссылок использованием заимствованного из Фортрана стандарта шестизначных имен.)

Как бы то ни было, правильным ответом при решении вопроса о внутренней документации является переход от широко применяемого в настоящее время отдельного текста к обширным комментариям непосредственно в распечатке программы. Эти комментарии должны содержать легко читаемые имена и иметь графически ясную структуру. В результате будет получена полная и точная внутренняя документация с достаточным уровнем детализации (назовем ее документацией детального уровня).

Выше упоминалось о том, что внутренняя документация может служить описанием системы и может быть положена в основу отчета о разработке. Отметим, какой ясной становится структура документации после того, как документация детального уровня перенесена в распечатку программы, и теперь очевидно ее отличие от документации высокого уровня. Документация высокого уровня представляет собой отдельные от программы тома текста. Теперь, когда мы знаем различие между этими двумя видами документации, можем указать дополнительные функции, присущие только внутренней документации: она содержит информацию о проектном решении и основных его целях. Кроме того, мы можем связать воедино высокий уровень документации с более низкими ее уровнями. Внутренняя документация может содержать объяснения роли компонентов программного обеспечения среднего уровня и указаний к ним, которые связывают воедино документацию высокого и детального уровней.

Исходя из всего сказанного выше, мы можем теперь дать определение качественной внутренней документации.

I. Определение документации высокого уровня (документ, который, возможно, имеет небольшой объем).
а) Общий обзор структуры.
б) Общий обзор базы данных.
в) Данные о результатах проектирования.
г) Основополагающие идеи.
д) Структура(ы) среднего уровня.
ж) Указатели информации детального уровня в листинге.

II. Определение документации детального уровня (распечатки программы).
а) Комментарии, представляющие собой:
1) подробное описание структуры;
2) подробное описание базы данных;
3) подробное описание функций;
4) особенности реализации.
б) Легко читаемые имена.
в) Индентированная структура программы.

Раскроем более подробно приведенные выше критерии.

I. Определение документации высокого уровня.

а) Общий обзор структуры — рисунки с надписями, показывающие структурную схему всей системы в целом. Следует также включить функциональную блок-схему, на которой видна работа основных компонентов системы. Еще лучше будет расширить блок-схему, добавив: 1) оверлейную структуру (если таковая имеется); 2) порядок работы; 3) поток данных. Если все это невозможно уместить на одной блок-схеме, следует использовать их несколько.

б) Общий обзор базы данных — рисунки с пояснениями, показывающие роль данных в системе в целом. Каковы основные файлы, основные структуры, основные наборы данных? Как и где они используются.

в) Данные о результатах проектирования — это информация об истории проектирования (о ней говорилось в разд. 3.2.3.1). Часто это рукописные записи, снабженные краткими ссылками.

г) Основополагающие идеи — этот пункт чаще всего пропускается программистами. Почему программа построена именно таким образом? Какой стиль программирования применялся? Какие цели ставились перед программой? Многие программисты, возможно, и не подозревают о том, что существуют основополагающие идеи. И тем не менее они есть, и эта часть документации очень важна.

д) Структура среднего уровня — что должен знать читающий, если у него есть вся информация из п. «а», чтобы он мог обратиться к листингу и понять всю документацию детального уровня, приведенную в нем? Здесь должны содержаться ответы на этот вопрос, которые могут быть достаточно сложными, если система большая. После п. «г» эта информация занимает второе место в том смысле, что сопровождающие программисты часто опускают ее.

е) База данных среднего уровня — так же как п. «д», этот пункт должен стать связующим звеном между всеобщей базой данных и листингом.

ж) Указатели информации детального уровня в листинге — важно, чтобы информация детального уровня была помещена в листинге, но не менее важно иметь возможность легко находить ее там. Документация высокого уровня должна помочь сопровождающему программисту ориентироваться в листинге. Где процедура HOOPLA? Где структура данных FRAMMIS? Где кластер CLOISTER?

II. Определение документации детального уровня.

а) Комментарии — правильное составление комментария подробно обсуждалось в разд. 3.2.2.1.

б) Легко читаемые имена — условия поименования обсуждались в разд. 3.2.2.1.

в) Индентированная структура программы — структура программы обсуждалась в разд. 3.2.2.1.

4.3.4. ПРОБЛЕМЫ, КОТОРЫЕ ПОДЛЕЖАТ РЕШЕНИЮ

При всей стройности предлагаемой системы ведения документации она по-прежнему имеет слабую сторону, в основе которой лежат два факта. Вспомним о том, что программисты не любят документацию, равно как руководители — вникать в детали. Поместив детальную документацию в листинг, мы не дали окончательного ответа на вопрос. Даже, напротив, мы усугубили неприязнь руководителей к деталям. Руководитель все также несет ответственность за внутреннюю документацию, только теперь ему надо обращаться за ней именно туда, куда ему меньше всего хочется заглядывать, — в листинг. И все же есть два соображения, позволяющие примирить между собой все противоречивые интересы, решив таким образом эту проблему. Во-первых, программисты скорее станут создавать хорошо комментированный листинг, чем писать документацию. Таким образом облегчится и работа руководителей. Во-вторых, руководители обязательно должны уметь читать листинги. Руководитель программного обеспечения, избегающий чтения листингов, выглядит так же странно, как руководитель производства, изготовляющего телевизоры, который

не хочет видеть продукцию, за которую он отвечает. Руководитель должен уметь проверить продукцию, за которую он отвечает. А листинг в полном смысле слова является продуктом программного обеспечения.

В настоящее время руководители судят о создаваемых во вверенных им организациях программах по документации (которая не является собственно программой и часто не отражает точно ее смысл). В результате во многих организациях появилась система параллельного документирования, при которой в документации описывается такое программное обеспечение, которое нужно руководству, а в листинге — такое, какое есть на самом деле. При таких условиях смешно говорить о руководящем контроле. Если же принять то положение, что руководители должны читать листинг хотя бы на уровне комментариев, а по возможности и в большем объеме, исчезает последнее препятствие к ведению документации в листинге.

Остается, однако, высказать свое мнение о том способе ведения документации, который подробно обсуждался, но достижения которого пока невелики, — о самодокументирующемся программном обеспечении. Программисты могут писать его и полагаться на него, руководители (неожиданно вспомнив то, что умели в свою бытность программистами) могут контролировать и использовать его. Но не надо забывать, что обзорный документ высокого уровня остается жизненно необходимым, а самодокументирующееся программное обеспечение, хотя и становится все реальнее, по-прежнему недоступно.

4.4. ОБОРУДОВАНИЕ ДЛЯ СОПРОВОЖДЕНИЯ

Сопровождающий программист, хотя мы и назвали его невоспетым героем, такой же человек, как и разработчик программного обеспечения. Поэтому может показаться абсурдным разговор о каком-то специальном оборудовании для программного сопровождения.

В какой-то степени это так и есть. Раз сопровождение программного обеспечения является составной частью мира разработки программного обеспечения, сопровождающему программисту следует создать такие же условия, как и разработчикам программного обеспечения. Если в вашем учреждении у всех отдельные комнаты, то они должны быть и у сопровождающих программистов. Если у вас сотрудники располагаются группами в небольших помещениях, также должны располагаться и сопровождающие программисты. Если у вас огромный зал, где рабочие места тянутся рядами, насколько хватает глаз, тогда пусть также сидят и они.

Но есть одна особенность: сопровождающим программистам нужно много места для хранения материалов. Им нужны ящики для

возвращенных распечаток, книжные шкафы для документации, перфокарт, магнитных лент, а также дисков с различными версиями, к которым будут возвращаться. Сама идея возврата к версиям предполагает, что сопровождающий программист отвечает за n версий. Даже если число n невелико, оно всегда больше 1.

Эта особенность налагает дополнительные заботы на группу конфигурационного управления. Обычно эта группа сохраняет все версии программного обеспечения независимо от того, являются ли они действующими в данный момент или нет. Но даже сопровождающий программист, при условии что сопровождение и конфигурационное управление производятся разными людьми, имеет свои особые нужды. Как правило, действующих версий даже одного куска программного обеспечения вполне достаточно, чтобы сопровождающему программисту потребовалось специальное место для хранения документации. Всегда бывает легко узнать, где работают сопровождающие программисты, потому что над их рабочими местами возвышаются целые пирамиды шкафов и ящиков!

Хотя место для хранения материалов — это единственная специфическая особенность оборудования рабочего места сопровождающего программиста, есть еще одно условие их работы, о котором нельзя забывать, — тишина. Помните, что сопровождающий программист должен вносить новое, не разрушая старого. Умственное напряжение, необходимое для такой работы, требует спокойной обстановки, которую не нарушает стук пишущих машинок и болтовня сотрудников. Лучше всего позволяют обеспечить тишину специальные кабины для научной работы [14], где программист может спокойно подумать. Однако тогда затрудняется доступ к документации. По меньшей мере смешно представить себе сопровождающего программиста везущим в кабину для научной работы целую тележку листингов и документов. Как ни странно, почти нигде не думают о тишине. Руководители меньше всего заботятся о тишине как условии работы сопровождающих программистов, исключая те учреждения, где есть отдельные кабинеты. Если бы был проведен опрос мнения работников (в особенности сопровождающих программистов), важность тишины выступила бы на первый план.

ЛИТЕРАТУРА

1 Alberts, The Economics of Software Quality Assurance, Proceedings of the National Computer Conference, 1976.

Дается экономический анализ жизненного цикла программного обеспечения с целью выделить в нем те моменты, когда следует применять методику обеспечения качества программного обеспечения SWQA. Обсуждаются эффективность методики SWQA и средства, применяемые в ней (структурное программирование, направленность разработки, ведущие группы программистов, средства автоматизации). Делается вывод, что методика SWQA должна быть направлена на раннее выявление и исправление ошибок этапа проектирования.

2. Allen, Lee, Tushman, The Relation of Internal Communication to R&D Project Perfor-

mance as a Function of the Nature of the Project, M.I.T. Industrial Liaison Program WP 1016, 1977.
Изучается влияние взаимосвязей на качество выполнения задания. Отмечается, что разными фукнциональными блоками следует управлять по-разному в зависимости от их свойств, назначения и количества специалистов.

3. См. [3] к гл. 3.
4. Branning, Willson, Schaenzer, Erickson, Modern Programming Practices Study Report, RADC-TR-77-106, 1977.
Глава 6 этого отчета посвящена конфигурационному управлению, примененному фирмой Sperry Univac для создания четырех больших программ по заказу Военно-морского флота США. Дается оценка его эффективности.
5. См. [4] к гл. 2.
6. Bucher, Maintenance of the Computer Sciences Teleprocessing System, Proceedings of the International Conference on Reliable Software, 1975.
Описываются работы по сопровождению программного обеспечения для конкретного задания. Делается упор на управление сопровождением. Показана роль Совета по внесению изменений и Комиссии по оценке системы. Обсуждаются протоколы изменений и методы тестирования.
7. Cashman, Holt, A Communication-oriented Approach to Structuring the Software Maintenance Environment, Proceedings of the ADA Environment Workshop, November 1979. Дается описание системы, позволяющей вести учет рабочих отчетов об ошибках в программном обеспечении. Система MONSTR позволяет осуществлять связь между сопровождающими программистами, занимающимися учетом ошибок в программном обеспечении.
8. Dodson, Resource Analysis for Data-proceessing Software, General Research Corp., 1977
Рассматривается проблема затрат на программное обеспечение на современном этапе. Предлагаются направления для дальнейших исследований.
9. См. [8] к гл. 2.
10. См. [7] к гл. 3.
11. Finfer, Mish, Software Acquisition Management Guidebook, Cost Estimation and Measurement, System Development Corp. report SDC-TM-5772 / 007 / 02, 1978.
Раскрывается методика оценки затрат на программное обеспечение для ВВС, а также рассматривается несколько примеров.
12. См. [8] к гл. 3.
13. См. [12] к гл. 3.
14. Glass, Environment and the Computer Programmer, PGR Quarterly Newsletter, Summer 1969.
В работе речь идет об оптимальной обстановке для работы сопровождающих программистов, включая небольшие комнаты, рассчитанные на одну группу сопровождающих программистов, плюс отдельные кабины для научной работы, обеспечивающие полную тишину, и комнаты для заседаний, где проводились бы обсуждения.
15. Glass, Tales of Computing Folk: Hot Dogs and Mixed Nuts, Computing Trends, 1978.
В притче «Устарелая технология и личное забвение» описывается драма руководителей и программистов, которые потеряли квалификацию, отстали от жизни. В сказке «Небо падает» описывается трудное положение, в котором оказался руководитель, не способный идти в ногу с все возрастающими требованиями к программному обеспечению.
16. Glass, Lying to Management ... a Legitimate Problem Solution, The Power of Peonage, Computing Trends, 1979.
Хроникально изображен конфликт между знающим, способным сопровождающим программистом и несведущим в вопросах сопровождения руководителем.
17. См. [12] к гл. 2.
18. См. [16] к гл. 3.
19. Hetzel, A Perspective on Software Development, Proceedings of the 3rd International Conference on Software Engineering, 1978.
В работе приведены размышления одного руководителя по проблемам разработки

программного обеспечения. Автор считает «неэффективное» руководство основной (хотя и не единственной) причиной возникновения ошибок.
20. См. [15] к гл. 2.
21. Lehman, How Software Projects Are Really Managed, *Datamation* (January 1979).
В работе собраны результаты анализа деятельности как научного, так и административного руководства программным обеспечением, в основном в области космонавтики. Наряду с самыми неожиданными результатами (например, системы, где отсутствовал руководящий контроль, оказались в среднем более продуктивными, чем системы, где такой контроль имеется) имеются и результаты, вполне предсказуемые (например, требования должны быть четко определены до начала разработки, а разработка должна быть закончена до написания программ). Упоминаются нетипичные методы руководства (например, стимулирующая оплата или установление размеров оплаты по системе аукциона).
22. См. [18] к гл. 2.
23. См. [7] к гл. 1.
24. Lindhorst, Scheduled Maintenance of Applications Software, *Datamation* (May 1973).
Пропагандируется так называемое сопровождение по графику, при котором изменения в программном обеспечении учитываются, систематизируются и вносятся одновременно. При таком способе руководства сводятся к минимуму штат работников и количество операций, производимых с окончательным результатом программы.
25. Miller, A Service Concept for Software Auditing, Proceedings of the NSF Software Auditing Workshop, 1976.
Предлагается ревизионная служба программного обеспечения как средство достижения высокого качества программного обеспечения. Отдельно обсуждаются типы проверок и возможные затраты, связанные с ними. Обсуждается также применение вспомогательных средств проверки.
26. MIL-ST-483 (USAF), Appendix VI, 1970.
Детально описываются форма и содержание требований технических условий программного обеспечения для военных целей. В технических условиях разработки (ч. 1) описываются требования, в технических условиях на рабочие программы (ч. 2) — внутренняя структура программного обеспечения.
27. См. [24] к гл. 2.
28. См. [38] к гл. 3.
29. Perry, Willmorth, An Investigation of Programming Practices in Selected Air Force Projects, RADC-TR-7-182, 1977.
Раздел II.4.4 представляет собой открытое обсуждения организации и ошибок большого военного проекта, выполненного корпорацией системных исследований. Все выводы основаны на расчетах.
30. Pokorney, Mitchel, A Systems Approach to Computer Programs, A Management Guide to Computer Programming, American Data Processing, 1968.
Даются самые первоначальные представления о роли вычислительных программ в руководстве системами ВВС. Определены элементы процесса создания программного обеспечения и место руководства в этом процессе. Приводятся примеры, некоторые из которых удручают.
31. Prudhomme, Implementing a Software Quality Assurance Program for the Viking Lander Flight Software, Transactions of Software 77 Conference.
Речь идет о полете космического корабля «Викинг» с целью исследования поверхности Марса. Дается подробное описание средств обеспечения качества программного обеспечения системы «Викинг», включающих: 1) средство гарантии качества проектирования, 2) конфигурационное управление, 3) верификационно-оценочную проверку, 4) отчеты о неудачах. В работе говорится о влиянии системы на некоторые другие функциональные блоки «Викинга».
32. Viking Software Data, RADC-TR-77-168, Software Change Request / Impact Summary, pp. 222-228, 1977.
Открыто обсуждаются отчеты об изменениях в проекте «Викинг», в том числе и использованные в них формы и методы.
33. Sachs, Some Comments on Comments, *SIGDOC Newsletter* (December 1976).

В работе речь идет о способах комментирования документации в программе. Автор подробно высказывает свое мнение о структурном программировании и «самодокументирующейся» программе.

34. Scholten The QARole in Software Verification, Transactions of Software 77 Conference. Предлагается подход к верификации программного обеспечения на основе жизненного цикла. Описывается влияние обеспечения качества программного обеспечения (SWQA) на стадии жизненного цикла. Автор отстаивает необходимость SWQA на каждой из стадий.
35. Smith, An Organization for Successful Project Management, Proceedings of the 1972 Spring Joint Computer Conference.
    Автор предлагает проект сбалансированной организации руководства с разделением ответственности между рядом сотрудников и системой формальных проверок. Определяет главную причину неудач в разработке программного обеспечения. Предлагает иметь группы разработки, интеграции и контроля, а также определяет функции каждой из них.
36. См. [50] к гл. 3.
37. См. [54] к гл. 3.
38. Trivedi Shooman, Error Data Collection in Software Systems, Computer Software Reliability; Many State Markov Modeling Techniques, RADC-TR-169, 1975.
    Обсуждается процесс написания отчетов об ошибках, выявляемых с помощью исследования моделей надежности. Предлагаются усовершенствования современной техники написания отчетов об ошибках.
39. См. [32] к гл. 2.
40. См. [57] к гл. 3.

Глава 5

ДНЕВНИК СОПРОВОЖДАЮЩЕГО ПРОГРАММИСТА

Одна из проблем обучения по книгам состоит в том, что вообще слова позволяют получить лишь приблизительное и отрывочное представление об изучаемом предмете. Одно дело говорить о роли личности в процессе сопровождения, обсуждать методы, применяемые в сопровождении, обсуждать проблемы и выгоды от их решений. Совсем другое дело, когда на вас лично возлагают ответственность за сопровождение данного программного обеспечения, вам дают потрепанные распечатки программы, кипу сопроводительной документации, размеры которой пугают (но при ближайшем рассмотрении оказываются недостаточными), и указывают общее направление вашей работы.

Что такое сопровождение, вы начинаете осознавать постепенно, по мере того как изучаете распечатки программ, используя все свои умственные возможности для того, чтобы уловить ход мысли вашего предшественника, и обращаетесь за справками к документации, когда это необходимо. Это тяжелая, напряженная умственная работа. Даже по прошествии месяцев усердной напряженной работы она по-прежнему остается трудной — иногда изнурительной, иногда приводящей в отчаяние, а иногда (когда вы вдруг находите неполадку, которая мучила вас в течение многих недель) радостной.

В этой главе мы попытаемся оживить представление читателей о сопровождении, приблизив его к реальным обстоятельствам. Она содержит выдержки из дневника реально существующего сопровождающего программиста. Это слегка отредактированный типичный текст дневника сопровождающего программиста за месяц. Время здесь сжато, а события подобраны так, чтобы читатель получил полное представление об обстановке, в которой работает такой специалист.

Сопровождаемое программное обеспечение представляет собой компилятор языка высокого уровня (ЯВУ), применяемый для разработки и сопровождения большой системы реального времени (которая содержит более 500 программ, насчитывает более 50 000 операторов и обслуживает более 100 программистов). Все события происходят на фоне так называемой нормальной работы по сопровождению, которая заключается в исправлении различных неполадок. Те промежутки времени, которые не описаны в дневнике,

сопровождающий программист занят примерно такой же работой, как та, что описывается здесь.
Месяц начинается ...

1 августа, 8 ч 21 мин

Когда я пришел, меня уже ждал пользователь с распечаткой. Он еще не знает этого программного обеспечения, и язык для него новый. Он пытается использовать подпрограмму на Ассемблере, чтобы выполнить функцию, которая существует в языке как одно из средств. После краткого объяснения факта существования средства и способа его использования он отказывается от мысли использовать Ассемблер.

1 августа, 15 ч 00 мин

Запускаю все тест-программы на компиляцию и выполнение. Мы готовим новый компилятор для передачи пользователю, и это последняя проверка перед сдачей.

2 августа, с 8 ч 00 мин до 10 ч 00 мин

Принимаюсь за анализ результатов выполнения тест-программ. Все они в конечном итоге оказываются верными. В последние недели они выполнялись трижды. При анализе сначала были обнаружены мелкие ошибки, и для их устранения в программу были внесены изменения. Этот процесс был значительно ускорен с помощью компаратора.

2 августа, с 12 ч 00 мин до 13 ч 00 мин

Подготавливаю инструкцию пользователю, в которой сообщается о существовании новой версии компилятора и регистрируются все внесенные изменения. Задача состоит в том, чтобы описать исправления с указанием фамилии программиста, обнаружившего ошибку, описать усовершенствования и описать, как пользоваться новой версией компилятора.

2 августа, 16 ч 00 мин

Даю машине задание переписать программу компилятора на файл, который обычно используется пользователями, чтобы получить программный компилятор. С помощью инструкции пользователю должно быть совершенно ясно, как пользоваться новым компилятором.

4 августа

Новая версия компилятора готова. Высылаем инструкцию пользователю. Все изменения в новой версии внесены в документацию, документация на обслуживание и справочник пользователя дополнены.

8 августа, 10 ч 41 мин

Приходит пользователь с вопросом. Достаем справочник пользователя и ищем в нем ответ. Описание составлено расплывчато. Обращаемся к распечатке программы. В результате находим ответ на вопрос. Но после ухода пользователя приходится сесть и написать отчет о неполадке, проверить описание языка и переписать абзац в справочнике пользователя.

9 августа, 16 ч 42 мин

Появляется пользователь с большой кипой распечаток программ. Он говорит, что рекомпилировал свою программу с помощью версии компилятора и новая программа не делает всего того, что делала старая. (Сопровождающий программист говорит себе: «Спокойно. Не может быть, чтобы это была ошибка компилятора. Я очень тщательно проверял все внесенные изменения. Все тесты прошли нормально».) Пользователю: «Что именно вас беспокоит, можете ли вы выделить различия между двумя прогонами?» «Да, конечно! Похоже, что этот оператор не работает». Внимательно просмотрев программу, выданную компилятором, замечаю, что используется не тот регистр, который следовало бы. Извлекаю распечатку программы генератора кодов и нахожу ту часть, которая может относиться к ошибке. Ну, конечно, прямо в центре страницы исправление. Тщательно проверяю программу компилятора и нахожу в ней ошибку — была использована неинициализированная переменная. Первоначальная проверка не выявила этой ошибки скорее всего потому, что переменная имела неправильное, но допустимое значение, в то время как компилятор сообщает об ошибке лишь в случае, когда переменная принимает недопустимое значение. Предлагаю пользователю выход из положения и подготавливаю отчет об ошибке.

10 августа, 10 ч 25 мин

Появляется пользователь с распечаткой программы и говорит: «Моя программа выдала сбой, а я не могу понять, в чем дело». (Понимать эту фразу надо так: «Я не могу прочитать дамп».) Итак, я начинаю читать, попутно поясняя, как это делается. Наконец, мы выясняем, что была использована неправильная величина индекса, и довольный пользователь уходит исправлять свою ошибку.

11 августа, 10 ч 30 мин

Появляется пользователь с распечаткой программы. Компилятор генерирует неправильный код. Мы составляем отчет о неполадке и присваиваем ей низкий приоритет. Я присваиваю соответствующие номера распечатке и отчету о неисправности и подшиваю их в папку. Затем обдумываю и предлагаю пользователю способ устранения

этой неисправности. Документирую предложенный способ устранения неисправности и подшиваю документ в ту же папку.

16 августа, 9 ч 02 мин

Пользователь спрашивает: «Каково влияние увеличения объема памяти на время работы компилятора?» Даю объяснение: время работы компилятора уменьшается с увеличением объема памяти.

16 августа, 11 ч 14 мин

Позвонили из отдаленного центра, который также использует наш компилятор. Разговор продолжается 20 мин, в течение которых я стараюсь понять, действительно ли существует неисправность в программе. Задаю пользователю вопросы, предлагаю ему воспользоваться определенными источниками информации. Похоже, что ошибка в компиляторе действительно существует. Пишу отчет о неисправности и прошу пользователя выслать распечатку для дальнейшего анализа.

17 августа, 6 ч 15 мин

Я пришел пораньше, чтобы в тишине поработать над задачей, требующей полного внимания. Один из новых программистов спрашивает, где включается свет. Я объясняю ему, как туда пройти, и в результате все помещение просто залито светом. (На какие только вопросы не приходится отвечать сопровождающим программистам!)

18 августа, 10 ч 30 мин

Появляется пользователь с распечаткой дампа. У него нет распечатки программы объектного модуля. Ситуация обычная: выполнение программы прервано. «Объясните, в чем дело?» — спрашивает он. При внимательной проверке становится ясно, что произошло обращение к подпрограмме по неверному адресу. Мне известно, что этот адрес должен выбираться редактором связи, а вместо адреса содержится нуль. Поскольку в операционную систему было внесено изменение, в результате которого редактор связи уже давал сбои на прошлой неделе, я предлагаю пользователю провести повторную компиляцию объектного модуля и надеюсь, что ошибка больше не возникнет.

19 августа, 8 ч 21 мин

Иду в комнату, где стоит ЭВМ, чтобы взять накопившуюся с вечера работу. Там встречаю программиста, с которым вчера мы работали над распечаткой дампа. Он говорит, что провел повторную компиляцию программы и осуществил ее прогон. Похоже, что все в

порядке. Это укрепляет мою уверенность в том, что был сбой в работе редактора связей. Приглашаю его заходить еще, если вновь понадобится помощь.

22 августа, 14 ч 45 мин

Приходит один из пользователей с распечаткой и говорит, что он придумал, как усовершенствовать компилятор. Он показывает отрезок программы и предлагает свой вариант, который позволяет сберечь 30 — 40% машинного времени. Я записываю его предложение, добавив несколько своих собственных соображений, и подшиваю эту запись в специальный журнал, который веду для таких случаев. Усовершенствования весьма полезны, если учесть, что компилируются более 500 объектных программ, содержащих более 50 000 операторов. Небольшие изменения в компиляторе могут иметь большое влияние на систему в целом, особенно если она работает в реальном времени.

23 августа, с 8 ч 00 мин до 12 ч 00 мин

Я продолжаю создавать и разрабатывать новые отрезки программ в основном в ответ на поступающие отчеты о неисправностях. Когда вносятся изменения в отрезок программы, необходимо проверить все возможные последствия этого, какова бы ни была причина внесения изменения. Надо всегда стараться изменять исходную программу таким образом, чтобы это влекло за собой как можно меньше изменений или хотя бы чтобы эти изменения были неотложными. Сопровождающий программист никогда не должен забывать о тех, кто будет работать после него.

24 августа

Как обычно (когда нет другой работы), я беру отчет о неисправности, которая еще не устранена, и соответствующую распечатку. Разрабатываю методику выявления ошибки. Прогоняю старую версию кампилятора, и проверка выявляет точно такую же ошибку, как та, о которой говорится в отчете о неисправности. Допущена ошибка в модулях компилятора. Вношу исправления и рекомпилирую модуль. Разрабатываю новый тест, прогоняю программу — неисправность устранена.

29 августа

Проект, в котором занято 100 программистов, требует модификации программного обеспечения. По мере того как к нам присылают новых программистов, я даю им краткий тренировочный курс (40 ч). Показываю, как пользоваться компилятором, рассказываю о характеристиках ЭВМ и основных идеях проекта. Я также даю консультации. У программистов, применяющих ЯВУ, часто возникают трудности. Причин может быть несколько: недостаточный

опыт применения различных версий языка или использование только одного языка, отсутствие прочных основ программирования, ошибки компиляции, отсутствие опыта применения более мощных языков, недостаток знаний о программном обеспечении в целом (которые делают трудным понимание его отдельных частей), незнание различий между ЭВМ и операционными системами и т. д.

29 августа, 16 ч 00 мин

Готовлюсь к завтрашним занятиям: просматриваю материал, который надо будет дать, обдумываю ответы на вопросы, которые не были освещены в прошлых занятиях.

30 августа, с 8 ч 00 мин до 12 ч 00 мин

Аудитория. Сегодня не будет неисправностей и не нужно будет их устранять. Сегодня я преподаватель.

Месяц заканчивается ...

Вывод: сопровождающий программист является активной составной частью программного комплекса. Он выполняет самые разнообразные функции, не все из которых входят в его прямые обязанности, начиная с консультации и преподавания и кончая поисками выключателей! Но главной задачей сопровождения остается продление жизни уже существующего программного обеспечения.

Глава 6

ЗАКЛЮЧЕНИЕ

Наша книга началась простым тестом. Читателю предложили ответить на несколько точно поставленных вопросов о сопровождении программного обеспечения. Ответы на эти вопросы, как вы, возможно, помните, оказались не такими уж точными. Мы убедились, что существует целый ряд заблуждений, которые искажают представление о сопровождении программного обеспечения.

По мере того как мы изучали эти заблуждения, выявляются следующие положения:

1. СПО — это не только исправление ошибок.

2. СПО — это особый мир в жизненном цикле программного обеспечения.

3. В СПО вкладываются большие средства.

4. Для того чтобы сопровождение программного обеспечения было успешным, необходимо с самого начала понять его цели.

Исходя из этих положений, можно прийти к более глубокому пониманию СПО.

Мы, например, изучали роль личности в СПО, роль тех самых «невоспетых героев», благодаря которым и продолжают работать ЭВМ. Мы убедились, что программист, сопровождающий программное обеспечение, должен быть широко образованным, терпеливым, способным рассуждать, обладать чувством ответственности и множеством других черт, которые присущи только опытным сопровождающим программистам, а не младшим малоквалифицированным специалистам, с которыми мы обычно имеем дело. Мы убедились также, что неверное представление о понятии стиля программирования приводит к возникновению стилевых конфликтов, что отрицательно отражается на сопровождении.

Разобравшись с ролью личности в сопровождении, мы перешли к его технической стороне. Мы выяснили, что набор средств сопровождающего программиста далеко не полон. Поскольку компилятор является основным средством работы сопровождающего программиста, необходима разработка более совершенного средства — суперкомпилятора. Мы убедились, что наилучшим способом осуществления сопровождения является так называемая профилактика сопровождения (термин представляется нам неудачным), применяемое лишь к программному обеспечению, поддающемуся сопровождению.

Затем мы перешли к рассмотрению вопроса о руководстве сопровождением. Мы говорили о необходимости учета индивидуальных особенностей работников, о важности отчетов (письменных и устных) для руководства процессом сопровождения, о роли контроля за изменениями и их учете в процессе сопровождения. В несколько категоричной форме высказали свое мнение о том, что документирование ведется неверно, и предложили свои соображения по поводу того, как исправить это положение.

Наконец, мы внесли некоторую жизненную правду в сухое изложение, выдержанное в стиле учебника. Мы перелистали страницы дневника программиста, заметив при этом, что сопровождающие программисты не только исправляют ошибки и имеют дело с ЭВМ, но также дают советы пользователям, обучают заказчиков и даже помогают в поиске выключателей! Работе сопровождающего программиста, как мы уже говорили, нет конца.

Каково же положение дел в сопровождении? Об этой области слишком мало говорится в литературе. Те, кто работает в области сопровождения, так же мало знают о методах усовершенствования своей работы, как и о том, насколько она важна. К этой области руководство недостаточно внимательно.

Такое положение не должно сохраняться. Начальным стадиям жизненного цикла программного обеспечения уделялось в последнее десятилетие огромное внимание. Языки программирования высокого уровня породили языки проектирования высокого уровня, которые в свою очередь порождают языки высокого уровня описания требований. Эти языки являются признаком возникновения новой технологии для реализации начальной стадии жизненного цикла программного обеспечения.

Такое же внимание необходимо уделить и конечной стадии жизненного цикла программного обеспечения. И с практической, и с экономической точки зрения именно на конечную стадию следует обратить внимание: львиная доля денежных и людских затрат приходится именно на нее.

Эта книга и сопутствующий ей «Справочник по надежности программного обеспечения» ставят своей целью привлечь внимание к этой проблеме. Надеемся, что начало положено. Конечная стадия жизненного цикла должна быть исследована с той же теоретической и практической глубиной, что и начальная стадия. Будем надеяться, что справочники по сопровождению и по надежности сопровождения, которые будут написаны следующим поколением, окажутся полнее и подробнее!

А пока, если вам, читатель, предстоит работа по сопровождению программного обеспечения, мы надеемся, что смогли помочь вам!

Роберт Гласс
Рональд Нуазо

Глава 7

БИБЛИОГРАФИЯ

Краткий обзор литературы по сопровождению программного обеспечения выявляет некоторые интересные факты:

1. Обзор занимает не слишком много времени, поскольку написано на эту тему мало.

2. Все, что написано, относится в основном к руководству сопровождением, а не к его техническим приемам.

3. Сопровождение и сопровождающих программистов называют самыми неожиданными эпитетами — «верхушкой айсберга», обслуживаемой «невоспетыми героями». Это только для примера (из разных источников).

4. Большая часть из того, что написано, связана с программным обеспечением для военных целей, хотя попадаются и работы по коммерческому применению и по системному обеспечению.

Основные источники этой книги помещены в списках аннотированной литературы в гл. 1—4. Списки довольно обширные, но при внимательном рассмотрении можно заметить, что большинство из работ написаны не специально по сопровождению и имеют к нему лишь косвенное отношение.

Здесь мы приводим список работ, написанных специально по сопровождению программного обеспечения. Они так важны, что специалисты, работающие в области программного обеспечения, обязательно должны о них знать.

1. Brooks, The Mythical Man-Month, Addison-Wesley, 1975.
2. A Study of Fundamental Factors Underlying Software Maintenance Problems, ESD TR-72-121, Vol. 11, 1971.
3. Special Collection on Software Science, IEEE Transactions on Software Engineering, March 1979.
4. Kernighan, Plauger, The Elements of Programming Style, McGraw-Hill, 1978.
5. Lehman, How Software Projects Are Really Managed, *Datamation* (January 1979).
6. Lientz, Swanson, Tompkins, Characteristics of Application Software Maintenance, *Communications of the ACM* (June 1978).
7. Requirements for Ada Programming Support Environment, *Stoneman* (February 1980).
8. Stanfield, Skrukrud, Software Acquisition Management Guidebook, Software Maintenance Volume, System Development Corp. TM-5772 / 004 / 02, Nov. 1977.
9. Weinberg, The Psychology of Computer Programming, Van Nostrand Reinhold, 1971.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Адаптация 18
Аттестация 129

Большие системы 104

Верификация 129
Внесение изменений в систему 36

Гибкость 27, 38

Дамп 49
Дескриптивный модуль 70
Документация 37, 100, 106, 135
— историческая 106
Документирование 37
Дихотомия 85

Жизненный цикл 10, 12

Заказчик 19
Затраты 15
Защита программы 93

Исполнительная программа 56
Исправление ошибок 35

Кластер 34, 79
Компаратор 58
Компилятор 52
Контроль 124
Конфигурационное управление 131
Коррекция 18

Литература 8, 21, 38, 42, 108, 143

Методика Джексона 85
Модификация 9
Модульность 31, 77

Надежность 15, 35

Обмен данными 91
Оборудование 141
Общий блок (Фортран) 32
Общий пул (Джовиал) 86
Ограничение изменений 37
Организация 120
Операционная система 56, 67
Описание версий 108
Оптимизация 36
Организационные структуры сопровождения 132
Отчеты 117
— об изменениях в программном обеспечении 125, 127
— об ошибках 107
— о состоянии изменений в программном обеспечении 128
Ответственность 28, 41
Отладка 13
Ошибки 15

Пакетный модуль (Ада) 80
Параметризация 89
Планирование сопровождения 113
Порождение 70
Преобразователь формата 62
Препроцессоры 70
Приемочные испытания 16
Проверки 117
Программирование 13
Проектирование 12, 13
Проектное решение 13
Профилактика сопровождения 99

Рабочее место программиста 72

Реализация 13
Революция 105
Регрессивные ошибки 130
Редактор связей 54
— текста 60
Роль личности 115
Руководство сопровождающего программиста 21

Самостоятельность мышления 28
Система 12, 19
Сложность 19
Совет по внесению изменений 123
Сопровождающий программист 14
Сопровождение 8
Спецификация 12
Способы и средства 49, 51
Средства отладки 57
— верификации 64
— сопровождения 50
Стадии
— определения технических требований 12
— проектирования 13
— программирования (реализации) 13
— отладки 13
— сопровождения 14
Стандарты 98
Стиль 29
Структура данных 84
— программы 81
Структурные схемы 63
Суперкомпилятор 67

Таблица перекрестных ссылок 54
Творческий подход 29
Терпение 27
Требования 11
Трудоемкость 103

Удобство сопровождения 36
Улучшение программы 103
Управление версиями 65
Усовершенствование 18
Утверждение 93

Хорошая память 29

Эволюция 105
Эффективность 36

Язык 51
— Ада 53, 71, 80, 86
— Алгол 86
— Ассемблер 31
— Бейсик 85
— высокого уровня (ЯВУ)
— Джовиал 86
— Кобол 32, 85, 86
— Модула 29
— Паскаль 86
— ПЛ-1 85
— проектирования программы (ЯПП) 63
— Фортран 32, 85, 86
— Эвклид 29

ОГЛАВЛЕНИЕ

Предисловие редактора перевода 5

Предисловие 6

Глава 1. Введение 8

1.1. Место сопровождения в жизненном цикле программного обеспечения

1.2. Миницикл процесса сопровождения 10

1.3 Усовершенствование, адаптация и коррекция в процессе сопровождения 16

17

1.3.1. Усовершенствование в процессе сопровождения 18

1.3.2. Адаптация в процессе сопровождения 18

1.3.3. Коррекция в процессе сопровождения 18

1.4. Определения 18

Литература 21

Глава 2. Роль личности программиста в процессе сопровождения 22

2.1. Проблема подбора специалистов в перспективе 24

2.2. Личные качества сопровождающего программиста 26

2.2.1. Гибкость в работе 27

2.2.2. Широкий профессиональный кругозор 27

2.2.3. Терпение 27

2.2.4. Самостоятельность мышления 28

2.2.5. Ответственность 28

2.2.6. Скромность и самокритичность 28

2.2.7. Творческий подход к программному обеспечению 29

2.2.8. Хорошая память 29

2.3 Стили и стилевые конфликты 29

2.3.1. Стиль программирования на Ассемблере 31

2.3.2. Стиль программирования на Фортране 32

2.3.3. Стиль программирования на Коболе 33

2.3.4. Стиль программирования на Алголе 33

2.3.5. Другие стили программирования 34

2.4. Задачи сопровождения и их приоритеты 35

2.4.1. Надежность программного обеспечения 35

2.4.2. Исправление ошибок 35

2.4.3. Внесение изменений в систему 36

2.4.4. Сопровождение ради сопровождения 36

2.4.5. Эффективность программного обеспечения 36

2.4.6. Ведение документации 37

2.5. Ограничение изменений 37

2.6. Нужды заказчика 39

2.7. Личная ответственность 41

Литература 42

Глава 3. Технический аспект сопровождения 45

3.1. Чем занимается сопровождающий программист 46
3.1.1. Сопровождающий программист и пользователь 48
3.1.2. Сопровождающий программист и его журнал 48
3.2. Как работает сопровождающий программист 49
3.2.1. Средства 49
3.2.2. Технические приемы сопровождения 75
3.2.3. Документация 106
Литература 108

Глава 4. Административная сторона сопровождения программного обеспечения 113

4.1. Планирование сопровождения 113
4.1.1. Планирование высококачественного сопровождения. Роль личности 114
4.1.2. Планирование высококачественного сопровождения. Отчет и проверки 117
4.1.3. Другие аспекты планирования 119
4.2. Организация сопровождения 120
4.2.1. Совет по внесению изменений 122
4.2.2. Контроль над работой по внесению изменений 124
4.2.3. Аттестация 129
4.2.4. Конфигурационное управление 131
4.2.5. Возможные организационные структуры сопровождения . . . 132
4.3. Документация к сопровождению 135
4.3.1. Почему мы заблуждались? 135
4.3.2. Чем плоха отдельная от программы документация? 136
4.3.3. Как правильно выйти из этого положения? 137
4.3.4. Проблемы, которые подлежат решению 140
4.4. Оборудование для сопровождения 141
Литература 142

Глава 5. Дневник сопровождающего программиста . 146

Глава 6. Заключение . 152

Глава 7. Библиография . 154

Предметный указатель . 155

УВАЖАЕМЫЙ ЧИТАТЕЛЬ!

Ваши замечания о содержании книги, ее оформлении, качестве перевода и другие просим присылать по адресу: 129820, Москва, И-110, ГСП, 1 й Рижский пер., д. 2 изд-во «Мир»

Роберт Гласс, Рональд Нуазо

СОПРОВОЖДЕНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Научный редактор Т. Н. Шестакова
Младший научный редактор Ю. Л. Евдокимова
Художник И. И. Каледин
Художественный редактор И. И. Каледин
Технический редактор Т. А. Максимова
Корректор М. А. Смирнов

ИБ 3465

Сдано в набор 27.07.82. Подписано к печати 31.01.83. Формат 60 × 90¹/₁₆. Бумага типографская № 1. Гарнитура таймс. Печать высокая. Бум. л. 5,0. Усл. печ. л. 10,0. Усл. кр.-отт. 10,23. Уч.-изд. л. 10,92. Изд. № 20/2456. Тираж 20 000 экз. Зак. 1387. Цена 80 к.

ИЗДАТЕЛЬСТВО МИР»

129820, Москва, И-110, ГСП, 1-й Рижский пер., д. 2.

Ярославский полиграфкомбинат Союзполиграфпрома при Государственном комитете СССР по делам издательств, полиграфии и книжной торговли. 150014, Ярославль, ул. Свободы, 97.