БИБЛИОТЕЧКА
ПРОГРАММИСТА

Ф. П. БРУКС мл.

КАК ПРОЕКТИРУЮТСЯ И СОЗДАЮТСЯ ПРОГРАММНЫЕ КОМПЛЕКСЫ

МИФИЧЕСКИЙ ЧЕЛОВЕКО-МЕСЯЦ

очерки по системному программированию

Перевод с английского Н. А. ЧЕРЕМНЫХ

Под редакцией А. П. ЕРШОВА

МОСКВА «НАУКА»
ГЛАВНАЯ РЕДАКЦИЯ
ФИЗИКО-МАТЕМАТИЧЕСКОЙ ЛИТЕРАТУРЫ
1979

22.18
Б 89
УДК 519.6

Frederick P. Brooks, Jr.

THE MYTHICAL MAN-MONTH

(Essays on Software Engineering)

ADDISON-WESLEY PUBLISHING COMPANY
READING
1975

Б 20204 — 143
053(02)-79 30-79. 1501000000

ОГЛАВЛЕНИЕ

Предисловие редактора 4

Предисловие 7

I. Асфальтовая топь . . . 10
II. Мифический человеко-месяц 17
III. Хирургическая бригада 27
IV. Аристократия, демократия и системное проектирование 35
V. Эффект второй системы . 45
VI. Путь слова 51
VII. Почему обрушилась Вавилонская башня 60
VIII. Объявление цели 70
IX. Десять фунтов в пятифунтовом мешке 76
X. Документационная гипотеза . 83
XI. План на выброс . . . 88
XII. Острый инструмент . 96
XIII. Целое из частей . . 106
XIV. Приближение катастрофы 116
XV. Второе лицо . . . 125

Эпилог 140

Примечания и ссылки 141

Указатель 147

ПРЕДИСЛОВИЕ РЕДАКТОРА

Проблемы создания математического обеспечения сейчас волнуют и привлекают внимание многих. Трудность его качественного и своевременного создания общеизвестна и подтверждена задержками или даже срывами многих, в том числе и весьма важных проектов. Практика, однако, показывает, что одного лишь признания этой трудности и ликвидации чувства недооценки недостаточно для исправления положения. За последние годы штаты системных программистов и их оклады существенно возросли, однако опять-таки это не только не привело к большому скачку в развитии программирования, но даже кое-где усугубило положение: программисты с большим личным опытом и способностями, выйдя на руководящие должности, оказались неспособными адаптировать свой опыт к новым условиям плановой и коллективной работы, решать должным образом возникшие организационные проблемы и тем самым оправдать возложенные на коллектив надежды.

Следует признать, что сейчас узловыми проблемами являются:

а) повышение производительности труда программиста;

б) повышение надежности программного графика.

И та и другая проблемы носят комплексный характер и в этом их специфическая трудность.

Можно передать разработчикам самую совершенную технику и все же не решить проблему из-за их неспособности организовать должное взаимодействие. Можно в дополнение к технике выработать самые четкие формы документации, планы работ и методы контроля и лишь в конце работы обнаружить, что выбранная организация совершенно не соответствует потребностям проектируемой системы.

Значимость публикуемых в этой книге очерков состоит в том, что в сумме они дают возможность системному программисту подойти к проблеме разработки математического обеспечения комплексно — с позиций сути дела, технологии работы, организации взаимодействий и учета индивидуальных факторов. Другой аспект комплексности состоит в том, что сотрудник любого уровня — рядовой инженер, ведущий конструктор, руководитель работ — найдут разносторонний материал, относящийся к каждому из них.

Особое значение, однако, излагаемый материал имеет для организаторов больших программистских проектов. Организация, как это убедительно показывает содержание книги, играет роль того стержня, который во многом предопределяет «лицо» и содержание проекта.

Книга представляет собой ряд очерков по системному программированию, принадлежащих профессору университета Северной Каролины (США) Фредерику П. Бруксу. Ценность этих очерков в том, что они основаны на опыте руководства проектированием и разработкой OS/360 — одной из крупнейших систем математического обеспечения.

Очерки затрагивают практически все: эстетические основы программирования, организацию работ на высшем и низшем уровнях, сущность руководства, понятие архитектуры программного комплекса, разницу между «обычным» и системным программированием,

способы документирования программ, методы отладки и т. п. Написанные лаконично, но весьма емко, ярким и образным языком, затрагивающие самые животрепещущие вопросы, волнующие программистов и их руководителей,— эти очерки будят мысль и дают очень четкую ориентацию.

Думаю, что книга не оставит равнодушным каждого, кто ее прочитает. Ощущение приобщения к истине будет появляться у активного читателя неоднократно. Было бы, однако, большим упрощением воспринимать рекомендации авторов как абсолютные или полностью проработанные. Вычислительная техника и программирование в силу своего базисного характера связаны со сферой производственных отношений, каковая в свою очередь определяется не только своим «машинным» содержанием, но и структурой общества, которому она служит. Поэтому советскому читателю будет очень важно творчески выработать свою интерпретацию организационной гипотезы, столь ярко и убедительно развиваемой в книге, обогатив тем самым складывающийся у нас на глазах фундамент системного программирования.

А. П. Ершов

Томасу Дж. Уотсону, мл.,
чья глубокая забота о людях
по-прежнему ощущается в его
коллективе, и
Бобу О. Ивенсу,
чья непреклонная воля
сделала работу приключением.

ПРЕДИСЛОВИЕ

Руководство большим программистским проектом во многом похоже на руководство любым большим предприятием в гораздо большей степени, чем считает большинство программистов. Но столь же во многом оно и отличается от него, причем гораздо сильнее, чем ожидает большинство профессиональных руководителей.

Предмет технологии программирования уже просматривается. Ему были посвящены несколько конференций, ряд книг и статей. Но он еще не настолько устоялся, чтобы могли появиться систематические учебники. Поэтому имеет смысл предложить эту книгу очерков, отражающих в основном только точку зрения автора.

Хотя я начинал с непосредственного программирования вычислительных задач, в течение нескольких лет (1956—1963), когда разрабатывалась автономная управляющая программа и транслятор с языка высокого уровня, я занимался архитектурой машин. И когда в 1964 г. я стал руководителем проекта создания операционной системы OS/360, то обнаружил, что программирование за прошедшие годы заметно изменилось.

Руководство разработкой OS/360 было крайне полезным опытом, хотя порой и очень горьким. Коллективу, включая Ф. М. Трапнелла, который сменил меня на посту руководителя, есть чем гордиться. В системе нашли свое воплощение многие прекрасные идеи, и она очень широко используется. Некоторые новые идеи, среди которых заметно выделяются независимая от устройств организация ввода/вывода и работа с внешними библиотеками, теперь уже применяются повсеместно. Система сейчас вполне надежна, довольно эффективна и весьма гибка.

Однако проект нельзя назвать вполне успешным. Любой пользователь OS/360 моментально вам скажет, что́ именно можно было бы сделать лучше. Просчеты в проекте и его реализации особенно сказываются на управляющей программе и в гораздо меньшей степени — на трансляторах. Большинство этих просчетов относится к периоду 1964—1965 г., так что они целиком лежат на моей совести. Кроме того, проект был закончен с опозданием, система потребовала бо́льшего объема памяти, чем планировалось, затраты значительно превысили предварительные оценки, и вся система начала работать как следует только после создания нескольких вариантов, последовавших за первым.

Как и предполагалось с самого начала, в 1965 г. я оставил фирму IBM, перешел в Университет в Чепел Хилле, и уже там начал анализировать опыт OS/360, пытаясь выяснить какие технические и управленческие уроки он мне преподал. В частности, по опыту руководства я хотел понять природу различий в разработке аппаратуры и программного обеспечения OS/360. Здесь мне были очень полезны долгие беседы с Р. П. Кейзом, заместителем руководителя проекта в период 1964—1965 гг., и Ф. М. Траншеллом, руководителем проекта с 1965 по 1968 гг.

Чтобы иметь материал для сравнительных выводов, я неоднократно беседовал с руководителями других гигантских программистских проектов, в том числе с Ф. Дж. Корбато из Массачусетского технологического института, Дж. Харром и В. Выссотски из фирмы Bell Telephones, Ч. Портманом из фирмы International Computers Limited, А. П. Ершовым из Вычислительного центра СО АН СССР и А. М. Пьетрасанта из фирмы IBM.

В этих очерках воплотились мои собственные выводы, которые предназначены профессиональным программистам, профессиональным руководителям и особенно профессиональным руководителям программистов.

Книга содержит один главный тезис, хотя и рассредоточенный по отдельным очеркам, но особенно подчеркиваемый в главах II—VII. Вкратце, я уверен, что большие программистские проекты, в отличие от малых разработок, страдают от трудноразрешимой проблемы управления, вызванной разделением

труда между многими исполнителями. Я считаю наиболее существенным требованием сохранение концептуальной целостности продукта программиста. Указанные главы обсуждают как трудности в достижении этой целостности, так и методы ее обеспечения. Последующие главы касаются других аспектов управления системным программированием.

Нельзя пожаловаться на отсутствие литературы в этой области, но она очень разбросана. Поэтому я постарался дать ссылки на дополнительное освещение отдельных конкретных положений и указать заинтересованному читателю на другие полезные работы. Многие мои друзья прочитали рукопись, а некоторые сделали полезные и развернутые комментарии; те из них, которые не смогли быть указаны в связном тексте книги, были включены в примечания.

Поскольку это книга очерков, а не монография, все ссылки и примечания собраны в конце, и читатель может их пропустить при первом чтении.

Я глубоко признателен мисс Саре Элизабет Мур, г-ну Дэвиду Уагнеру и г-же Ребекке Бурис за их помощь в подготовке рукописи, а также профессору Джозефу С. Слоану за советы по подбору художественных иллюстраций *).

Чепел Хилл, Сев. Каролина
Октябрь 1974.

Ф. П. Б., мл.

---

*) К сожалению, технические условия публикации не позволяют воспроизвести художественные иллюстрации, предваряющие каждый очерк. Не являясь частью изложения, они в то же время в метафорической и зачастую неожиданной форме подчеркивают главную мысль очерка. (*Прим. ред.*)

I. АСФАЛЬТОВАЯ ТОПЬ

«Корабль на мели — моряку маяк».

(Датская пословица)

Ни одна из сцен нашей предыстории не оставляет столь яркого впечатления, как смертельная схватка огромных животных с асфальтовой топью. Перед глазами встают динозавры, мамонты, саблезубые тигры, пытающиеся выбраться из топи. Однако чем отчаяннее борьба, тем сильнее сжимаются тиски, и как ни силен, как ни хитер зверь, в конце концов он погибает.

Программирование больших систем последние десять лет и было той асфальтовой топью, в которой увязли многие огромные и сильные звери. Почти все работающие системы не соответствовали своим спецификациям, своему назначению, не укладывались в графики и бюджет. Большие и маленькие, громоздкие и гибкие коллективы разработчиков неизбежно попадали в ловушку асфальтовой топи. Ничто, казалось, не вызывало затруднений — можно вытащить любую лапу. Однако накопление одновременных и взаимодействующих факторов приводило к замедлению движения.

Неподатливость проблемы вызывает всеобщее изумление, и разобраться в ее природе непросто. Но мы должны попытаться ее понять, чтобы впоследствии решить.

Начнем поэтому с определения ремесла системного программирования и присущих ему радостей и горестей.

Комплексный программный продукт

Время от времени в газетах можно прочесть о том, как два программиста в переоборудованном гараже написали очень важную программу, превосходящую

лучшие образцы, созданные большими коллективами. И каждый программист готов поверить в эти басни, поскольку знает, что может написать любую программу с гораздо большей скоростью, чем 1000 операторов в год, составляющие официальную производительность промышленных групп.

Но почему же тогда все производственные коллективы программистов не заменить малонаселенными гаражами? Давайте посмотрим, что именно там производится.

На рис. 1.1, слева вверху изображена *программа*. Она полностью завершена, автор может ее пропустить

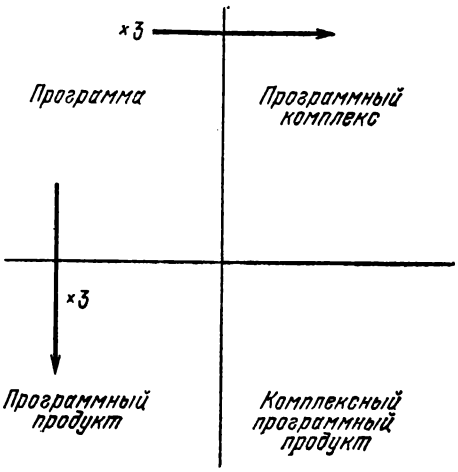


Рис. 1.1. Эволюция комплексного программного продукта.

в той системе, для которой она разработана. *Именно это* и создается обычно в гаражах, и этот объект используется для оценки производительности отдельного программиста.

Существуют два пути преобразования программы в более полезный, но и более дорогостоящий продукт. Они представлены на диаграмме вертикальной и горизонтальной стрелками.

Двигаясь через горизонтальную границу, программа превращается в *программный продукт*, т. е. в такую программу, которую любой может пропускать на ма-

шине, отлаживать, улучшать и расширять. Она используется во многих рабочих контекстах и для многих наборов данных. Чтобы превратиться в универсально используемый программный продукт, программа должна быть написана неким универсальным образом. В частности, ввод должен быть настолько обобщен, насколько это позволяет основной алгоритм. Далее, программу следует тщательно отладить, учитывая все влияющие на нее факторы, А это означает, что следует подготовить, пропустить на машине и зафиксировать значительный массив отладочных тестов, изучив область ввода и установив его границы. И, наконец, превращение программы в программный продукт сопровождается тщательной документацией с тем, чтобы любой мог ее использовать и расширять. По моим приближенным подсчетам, программный продукт по крайней мере в три раза дороже, чем отлаженная программа с той же самой функцией.

Пересекая вертикальную границу, программа превращается в компоненту *программного комплекса.* Это набор взаимодействующих программ, согласованных по функциям и по формату так, что их объединение представляет собой единое средство для решения больших задач. Чтобы стать частью программного комплекса, программа должна быть написана так, чтобы каждый вход и выход по синтаксису и семантике соответствовал точно определенным сопряжениям. Кроме того, программа должна быть организована так, чтобы она использовала только отведенные ей ресурсы: объем памяти, устройства ввода/вывода, машинное время. И, наконец, программа должна быть отлажена во всех возможных сочетаниях с другими компонентами комплекса. Эта отладка должна быть очень большой по объему, ведь число вариантов растет комбинаторно. Она требует больших затрат времени, ибо появляются очень тонкие ошибки из-за непредвиденных взаимодействий отлаживаемых частей. Компонента программного комплекса стóит, по крайней мере, в три раза больше, чем отдельная программа с той же функцией. Ее стоимость может быть выше, если система имеет много компонент.

В правом нижнем углу рисунка находится *комплексный программный продукт.* Он отличается от простой программы по всем вышеперечисленным пунктам

и стоит в девять раз больше, но это действительно полезный объект, конечный продукт всех усилий системного программиста.

Радости ремесла

Почему программирование доставляет удовольствие? Как вознаграждаются все усилия профессионала?

Первое — это абсолютная радость творчества. Как ребенок радуется, стряпая пирожки из песка, так взрослый наслаждается процессом создания вещей, особенно если он сам их придумал. Мне кажется, что прообразом этой радости творчества должно быть то удовольствие, с которым всевышний занимался сотворением мира и которое нашло свое отражение в оригинальности и красоте каждого листика, каждой снежинки.

Второе — это радость создания вещей, полезных другим людям. Где-то в глубине души мы хотим, чтобы другие использовали нашу работу и находили ее полезной. В этом смысле продукт программирования не слишком отличается от первой детской подставки для карандашей «в подарок папе».

Третье — это очарование, заключенное в самом процессе создания сложных, загадочных объектов, состоящих из взаимосвязанных, непостоянных частей, и наблюдения за тем, как они работают в запутанных циклах, сохраняя верность принципам, заложенным в них с самого начала. Вычислительная машина обладает притягательной силой биллиарда или музыкального автомата, доведенных до логической завершенности.

Четвертое — это возможность постоянно учиться, вытекающая из непрерывно меняющегося характера задачи. В том или ином отношении проблема оказывается новой, и человек, ее решающий, приобретает новые знания, иногда теоретические, иногда практические, а иногда и те, и другие вместе.

И последнее — это удовольствие работать с очень гибким материалом. Программист, как поэт, работает почти исключительно головой. Он строит свои зáмки в воздухе и из воздуха только силой своего воображения. Очень редко материал для творчества допускает такую гибкость, такую возможность столь частых улучшений и переделок и такими простыми средства-

ми позволяет осуществлять громадные замыслы. (Но, как мы увидим позднее, эта же самая гибкость порождает свои проблемы.)

Материал поэта — слова, и результат — те же слова́; в отличие от стихотворца, программист создает программный продукт, реальный в том смысле, что сам программист движется и работает, производя видимый результат, отличный от него самого. Он печатает результаты, чертит рисунки, производит звуки, управляет движением руки. Волшебство мифов и легенд стало явью в наши дни. Вы печатаете на клавиатуре заклинание, и вот экран дисплея оживает, показывая объекты, которых не было и могло не быть никогда.

Программирование доставляет нам радость, потому что позволяет удовлетворить стремление к творчеству, глубоко заложенное в каждом из нас, и разделить это чувство радости с другими.

Горести ремесла

Не все, однако, радует программиста, и знакомство с горестями, присущими нашему ремеслу, позволяет легче перенести их появление.

Во-первых, надо работать очень тщательно. В этом смысле ЭВМ тоже напоминает волшебство из сказок. Если хоть один символ, один пробел в магической формуле не находится строго на своем месте, волшебство не работает. Люди не привыкли к совершенству, и лишь немногие области человеческой деятельности требуют его. Привыкание к требованиям совершенной точности является, по моему мнению, наиболее трудным в процессе обучения программированию [1])*).

Во-вторых, задачи, стоящие перед программистом, определяют другие люди, они же отводят ему ресурсы и снабжают информацией. Один человек крайне редко сам определяет обстоятельства своей работы, не говоря уже о ее целях. В административных терминах это означает, что степень ответственности превышает объем прав. Однако, по-видимому, во всех областях созидательной деятельности формальный объем прав никогда не согласуется с ответственностью. В действительно-

*) Номера сносок указывают на примечания автора, собранные в конце книги. (*Прим. ред.*)

сти же фактический объем прав достигается как только работа завершена.

Зависимость от других проявляется здесь особо, весьма болезненно для системного программиста. Он зависит от чужих программ. В то же время эти программы зачастую неверно спроектированы, плохо реализованы, не полностью укомплектованы (например, нет программы на входном языке или нет тестов) и слабо документированы. Поэтому программисту приходится часами разбираться в таких вещах, которые в идеальном случае должны быть вполне завершены, доступны и легко используемы.

Следующее обстоятельство сводится к тому, что приятно выдавать большие идеи, но какой же поистине «адский труд» — иногда поиск самой крошечной ошибки! Любая творческая деятельность подразумевает долгие часы кропотливого и скучного труда, и программирование в этом смысле — отнюдь не исключение.

И, далее, могло бы показаться, что чем меньше ошибок в программе, тем легче их найти, т. е. скорость отладки как бы обладает квадратичной сходимостью. Совсем наоборот — сходимость оказывается линейной или хуже, т. е. в ходе отладки обнаружение последних ошибок требует гораздо больше времени, чем первых.

Последняя неприятность, а иногда и последнее разочарование заключается в том, что продукт, над которым вы трудились так долго, к моменту завершения (или даже раньше) уже устарел. Всегда коллеги и соперники гоняются за новыми и лучшими идеями. И вот уже замена выношенной вами идеи не только задумана, но и внесена в план.

Действительность, как правило, не так страшна. Новый и более совершенный продукт обычно еще не доступен к тому моменту, когда вы уже завершили свой собственный; о нем пока только говорят. И на его создание тоже потребуются долгие месяцы. Бумажный тигр не соперник реальному, если речь идет о действительном его использовании. Преимущества реальности всегда получают признание.

Конечно, техническая база, на которой все строится, постепенно идет вперед. Как только проект окончательно принят, он становится устаревшим в смысле своих концепций. Но осуществление проекта, направленного на получение реального продукта, требует

времени и труда. Степень отсталости реализованного проекта нужно измерять по сравнению с другими существующими реализациями, а не с еще нереализованными концепциями. Цель и задача состоят в том, чтобы найти реальные решения реальных проблем в соответствии с существующими планами и наличными ресурсами.

Таково программирование — одновременно и асфальтовая топь, поглощающая многие начинания, и творческая деятельность с присущими только ей радостями и горестями. В глазах многих ее радости значительно перевешивают все горести, и именно для них эта книга попытается проложить отдельные тропки через топь.

II. МИФИЧЕСКИЙ ЧЕЛОВЕКО-МЕСЯЦ

«Хорошая кухня требует времени. Если Вы готовы подождать, мы обслужим Вас гораздо лучше, и Вы получите бо́льшее удовольствие».

(Меню ресторана «Антуан», Новый Орлеан)

Почти все программистские проекты страдают скорее из-за нехватки времени, нежели из-за отсутствия каких-либо других ресурсов. Почему эта причина бедствий является столь всеобщей?

Во-первых, наши методы оценки весьма несовершенны. Строго говоря, они отражают некоторое неявно высказываемое и в корне неверное допущение, что все будет идти хорошо.

Во-вторых, наши методы оценки ошибочно путают усилия с достижениями, прячась за допущение, что человек и месяц взаимозаменяемы.

В-третьих, отсутствие уверенности в наших оценках ведет к отсутствию у руководителей программистских проектов вежливого упрямства, свойственного шеф-повару ресторана «Антуан».

В-четвертых, управление ходом разработки плохо организовано. Методы, давно опробованные и даже рутинные в других технических дисциплинах, в технологии программирования рассматриваются как радикальные новшества.

В-пятых, когда обнаруживается отставание от графика, естественная (и традиционная) реакция руководителя — добавить рабочей силы. А это, аналогично попытке заливать огонь бензином,— только ухудшает дело, причем значительно. Чем сильнее огонь, тем больше требуется бензина, круг замыкается, и последствия плачевны.

Наблюдение за выполнением графиков будет темой отдельной главы. Давайте пока подробнее рассмотрим другие аспекты проблемы.

Оптимизм

Все программисты — оптимисты. Может быть, это современное чародейство особенно привлекает тех, кто верит в добрых фей и счастливый конец. Может быть,

стократное крушение надежд способен пережить только тот, кто привык добиваться поставленной цели. Или, может быть, все дело в том, что вычислительные машины молоды, а юность всегда оптимистична. Однако как ни объясняй, но слышим мы одно и то же: «К этому сроку программа обязательно пройдет», или «Я только что нашел последнюю ошибку».

Итак, первое ложное допущение, лежащее в основе планирования деятельности системных программистов, заключается в том, что *все будет в порядке,* т. е. что *выполнение каждого задания займет ровно столько времени, сколько оно «должно» занять.*

Широкое распространение оптимизма среди программистов заслуживает более глубокого анализа. Дороти Сейерс в своей прекрасной книге «Мысль творца» («The Mind of the Maker») подразделяет творческую деятельность на три этапа: идея, реализация и взаимодействие. Книга, вычислительная машина, или программа сначала существуют как идея, вне времени и пространства, только в мозгу своего создателя, но в совершенно законченном виде. Замысел реализуется во времени и пространстве посредством пера, чернил и бумаги или же с помощью проводов, полупроводниковых схем и ферритовых сердечников. Процесс создания завершается, когда кто-то другой читает книгу, использует вычислительную машину, пропуская через нее программу, тем самым взаимодействуя с замыслом творца.

Это описание творческой деятельности человека поможет нам при решении нашей сегодняшней задачи. Для людей творческого труда неполнота и противоречивость их идей становится ясной только в процессе реализации; таким образом, описание, эксперимент, «решение» весьма важны для теоретика.

Во многих видах творческой деятельности средства реализации несовершенны. Древесина раскалывается, масляные краски засыхают, в электрических цепях происходит замыкание. Такое физическое несовершенство средств накладывает свои ограничения на предлагаемые идеи и, кроме того, может вызвать непредвиденные трудности в процессе реализации.

Реализация дается нам пóтом и кровью как из-за несовершенства физических средств, так и вследствие неадекватности наших основополагающих идей. Мы

склонны сваливать вину за большинство затруднений на средства реализации, поскольку они не «наши» в отличие от «наших» идей, которые нам трудно оценивать беспристрастно.

Программист, однако, имеет дело с очень податливым материалом: концепциями и весьма гибкими представлениями. Поскольку материал столь послушен, мы не ожидаем особых затруднений при его реализации, и отсюда наш всепроникающий оптимизм. Поскольку наши идеи неверны, мы получаем ошибки. Следовательно, наш оптимизм необоснован.

Допущение о том, что все будет в порядке, имеет вполне определенный вероятностный смысл для отдельно взятой задачи. Действительно, все может идти по плану, поскольку существует вероятностное распределение появления отставания от графика, а, стало быть, есть конечная вероятность, что отставания не будет. Большой программистский проект, однако, включает в себя много отдельных задач, каждая из которых может зависеть от окончания другой. Вероятность того, что каждая задача будет идти нормально, становится исчезающе малой.

Человеко-месяц

Вторая ложная предпосылка нашла свое отражение в самой единице, используемой при оценке производительности и составлении графиков, а именно, в человеко-месяце. Стоимость проекта действительно зависит от числа людей и от числа месяцев, но его успешность — нет. *Следовательно, человеко-месяц как единица измерения объема работы является опасным и вводящим в заблуждение мифом.* Этот миф основывается на предпосылке, что люди и месяцы взаимозаменяемы.

Человек и месяц взаимозаменяемы только в том случае, когда задание можно распределить между несколькими работниками, никак не зависящими друг от друга (рис. 2.1). Это справедливо на уборке пшеницы или сборе хлопка, но даже приблизительно неверно в системном программировании.

Когда задание нельзя распределить между несколькими работниками из-за ограничений на последова-

тельность выполняемых работ, привлечение дополнительных сил не влияет на график его выполнения (рис. 2.2). Чтобы выносить ребенка, нужно девять месяцев, независимо от того, сколько женщин будет к этому привлечено.

В заданиях, допускающих разбиение на взаимосвязанные подзадачи, следует прибавлять ко всему объе-

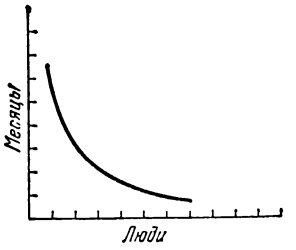


Рис. 2.1. Время и число работников — полностью распределяемое задание

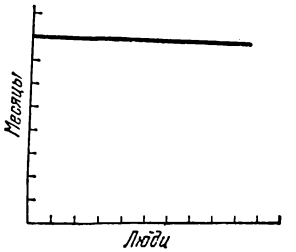


Рис. 2.2. Время и число работников — нераспределяемое задание

му предстоящей работы затраты на обеспечение связи. Поэтому здесь даже лучшие результаты всегда несколько хуже, чем в случае эквивалентного обмена человека на месяц (рис. 2.3).

Дополнительные затраты на обеспечение связи слагаются из двух частей — обучения и взаимосвязи. Каждый работник должен получить определенные технические навыки, познакомиться с целями и задачами, общей стратегией и планом работы. Такое обучение нельзя разбить на отдельные части, так что эта часть дополнительных затрат изменяется линейно в зависимости от числа работников [1]).

С установлением взаимосвязи дело обстоит хуже. Если каждая часть задачи должна отдельно координироваться с каждой другой частью, затраты возрастают как $n(n-1)/2$. Между тремя работниками в три раза больше попарных взаимосвязей, чем между двумя; между четырьмя — в шесть раз больше, Если, однако, для совместного решения вопросов нужно проводить совещания трех, четырех и более работников, ситуация становится еще хуже. Дополнительные затраты на обеспечение связи могут полностью нейтрализовать

эффект разбиения первоначальной задачи на части, что приводит нас к ситуации, показанной на рис. 2.4.

Так как разработка программного обеспечения по самой своей сути — деятельность системная, т. е. задача

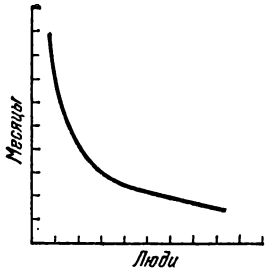


Рис. 2.3. Время и число работников — распределяемое задание, требующее связей между частями.

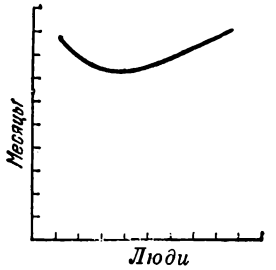


Рис. 2.4. Время и число работников — задача со сложными взаимосвязями.

на сложные взаимосвязи, то затраты на их обеспечение велики и быстро перевешивают ту экономию затрачиваемого на задачу времени, которая достигается благодаря разбиению задачи на части. Добавление людей лишь удлиняет сроки выполнения работ, а не укорачивает их.

Комплексная отладка

Ни одна часть графика работ не связана так сильно ограничениями на их последовательность, как отладка компонент и комплексная отладка. Очевидно, что требуемое время зависит от числа встречаемых ошибок и легкости их обнаружения. Будучи оптимистами, мы обычно ожидаем, что ошибок будет меньше, чем это оказывается в действительности. Именно поэтому отладка чаще всего не укладывается в график.

В течение нескольких лет я успешно применял следующее практическое правило планирования работ по созданию программного обеспечения:

— проектирование — 1/3,

— написание команд — 1/6,
— отладка компонент и отдельных подсистем — 1/4,
— системная (комплексная) отладка всех компонент — 1/4.

Это разбиение отличается от общепринятого по нескольким важным показателям:

1) доля, отведенная проектированию, больше обычной. Но даже в этом случае времени едва хватает на написание подробных и надежных спецификаций и вовсе недостает на разработку и внедрение существенно новых методов;

2) *половина* времени отводится на отладку написанной программы, что гораздо более обычного;

3) часть, которую легко оценить, т. е. собственно написание команд, занимает только одну шестую графика.

Знакомясь с проектами, планируемыми по общепринятой методике, я обнаружил, что по графику только в некоторых из них половина времени отводилась на отладку, но в действительности так получалось в большинстве проектов. Многие проекты до этапа комплексной отладки еще как-то укладывались в график [2]).

Проект оказывается в особенно бедственном положении, если отводится недостаточно времени на комплексную отладку. Так как задержка приходится на конец графика, то никто и не ожидает никаких неприятностей почти до самой даты окончания работ. Плохие новости обрушиваются на заказчиков и руководителей внезапно и чаще всего слишком поздно.

Кроме того, задержка в этот период влечет особенно суровые финансовые и психологические последствия. Штаты проекта полностью укомплектованы и затраты уже достигли предела. И что еще серьезнее, разрабатываемые программы должны обеспечивать другие виды деятельности, например, поставку вычислительных машин, ввод в действие новой системы, работу новых устройств и т. д., а так как почти всегда именно поставка программного обеспечения является последним этапом разработки, то эта задержка обходится очень дорого и вызванные ею дополнительные расходы на практике могут значительно превышать все остальные. Поэтому так важно в первоначальном графике проекта отводить достаточно времени на комплексную отладку.

Объективность оценки

Отметим, что настойчивость руководителя может определить график выполнения задания, но не в состоянии определить срок его действительного завершения. Омлет, обещанный через две минуты, может быть подан в срок, но если за две минуты он еще не готов, у заказчика два выбора — подождать или съесть его сырым. Заказчики программного обеспечения находятся перед таким же выбором.

Но у повара есть другой выход — он может прибавить огня. И зачастую омлет тогда уже ничто не может спасти — он подгорел с одной стороны и остался сырым с другой.

Я не считаю, что руководители программистских проектов обладают меньшей смелостью и настойчивостью, чем шеф-повар, или же чем руководители других технических проектов, но составление фиктивных графиков, соответствующих установкам начальства, в нашей области распространено гораздо больше, чем где-либо еще в технике. Дело в том, что энергичная и убедительная защита своих оценок, которые выводятся не на основе количественных методов, подтверждаются малым количеством данных и основываются преимущественно на интуиции руководителя,— это вещь очень трудная и сопряженная со многими неудобствами.

По-видимому, нужно систематизировать и публиковать данные о производительности, о частоте ошибок, правила выведения оценок и т. д. Профессионалы только выиграют от возможности совместного использования таких данных.

До тех пор, пока методы оценки не станут более надежными, руководителям придется проявлять твердость характера и защищать собственные оценки, следуя своей интуиции.

Нарастающие катастрофы с графиком

Что нужно предпринять, когда важный программистский проект не укладывается в график? Естественно, добавить рабочей силы. Как видно на рисунках (2.1—2.4), иногда это помогает, а иногда — нет.

Давайте рассмотрим пример[3]). Допустим, что трудоемкость задачи оценена в 12 человеко-месяцев и тро-

им сотрудникам отвели на нее 4 месяца, причем установили количественные вехи A, B, C и D, которых в соответствии с графиком нужно достичь в конце каждого месяца (рис. 2.5).

Допустим теперь, что первая отметка достигнута только через два месяца (рис. 2.6). Перед какими альтернативами оказался руководитель?

1. Допустим, что задачу нужно сделать вовремя. Допустим также, что неверно оценено только время

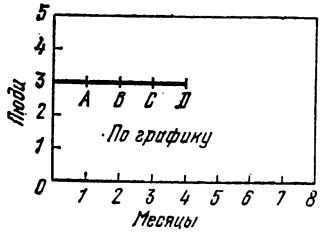


Рис. 2.5.

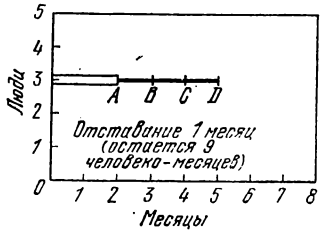


Рис. 2.6.

выполнения первой части (см. рис. 2.6). Тогда остается 9 человеко-месяцев усилий и два месяца времени, т. е. задача требует 4,5 человека. Добавим двоих людей к трем первоначальным.

2. Допустим, что задачу нужно сделать вовремя. Допустим также, что время выполнения было занижено, а реальное положение дел представлено на рис. 2.7. Тогда остается 18 человеко-месяцев работы и два месяца времени, т. е. понадобится 9 человек. Добавим шестерых к трем первоначальным работникам.

3. Составление нового графика. Мне нравится девиз П. Фагга, опытного инженера, специалиста по вычислительной технике: «Не исправляйте понемногу». Другими словами, делайте новый график достаточно свободным, чтобы работу можно было сделать тщательно и основательно без последующего перепланирования.

4. Ослабление задания. На практике это происходит довольно часто, когда рабочий коллектив вдруг обнаруживает отставание от графика. Если вторичная стоимость задержки очень высока, это является единственно приемлемым. У руководителя есть только две

возможности: или тщательно и формально сократить задание, составив новый график, или же просто наблюдать, как поспешное проектирование и незавершенная отладка мало-помалу сокращают объем задачи.

В двух первых случаях настаивать на том, чтобы задача безо всяких изменений была закончена за четы-

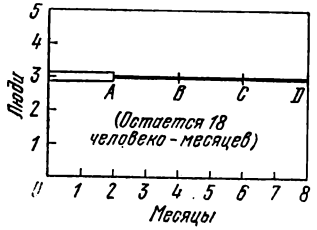


Рис. 2.7.

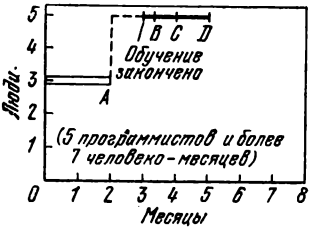


Рис. 2.8.

ре месяца, по меньшей мере ошибочно. Рассмотрим, например, какие помехи возникают в первом случае (рис. 2.8).

Для того, чтобы ввести в курс дела двух новых людей, пусть даже вполне компетентных и опытных, нужен один старый работник. Если ему на это потребуется месяц, то 3 *человеко-месяца будут отданы работе, никак не учитывающейся первоначальными планами.* Кроме того, задачу, ранее поделенную на три части, теперь придется поделить на пять частей, следовательно, часть уже проделанной работы пропадет, а комплексная отладка значительно удлинится. Значит, к концу третьего месяца останется больше 7 человеко-месяцев работы, 5 обученных людей и месяц времени. Как видно из рис. 2.8, сроки выполнения задания не сократились, несмотря на появление новых людей (см. рис. 2.6).

Чтобы выйти из положения, даже рассматривая только время на обучение и не учитывая перераспределения задачи и дополнительной отладки, потребовалось бы в конце второго месяца добавить не двоих, а четверых людей. Чтобы покрыть затраты на перераспределение и комплексную отладку, нужен еще один человек. Теперь, однако, рабочий коллектив состоит не из троих, а, по крайней мере, из семерых людей, и принципы орга-

низации работы, распределения заданий и прочее уже отличаются от прежних не только количественно, но и качественно.

Заметим, что к концу третьего месяца дела обстоят очень плохо. Отметка «1 марта» все еще не достигнута, несмотря на усилия руководителя. Очень велик соблазн повторить весь цикл и привлечь дополнительных работников. Но это безумный путь.

В данном примере предполагалось, что только первая веха была установлена неправильно. Если же 1-го марта принять более осторожное допущение, что весь график (см. рис. 2.7) слишком оптимистичен, то нужно к первоначальному коллективу добавить еще шесть человек. Предоставляем читателю в качестве упражнения вычислить, во что выльется обучение, перераспределение заданий, комплексная отладка. Но нет никаких сомнений в том, что полученный продукт будет хуже, а его осуществление потребует больше времени, чем в случае, когда исходная группа также состояла из 3 человек, но график был бы другим.

Крайне упрощая, мы сформулируем закон Брукса: «*Если программистский проект не укладывается в сроки, то добавление рабочей силы только задержит его окончание*».

Таким образом, мы развеиваем миф о человеко-месяце. Число месяцев, отводимых на проект, зависит от ограничений на его линейность. Максимальное число людей зависит от числа независимых подзадач. Исходя из этих двух величин, можно построить график, рассчитанный на меньшее число людей и большее количество месяцев. (Единственная опасность заключается в том, что конечный продукт устареет.) Нельзя, однако, составить работоспособный график, используя больше людей и меньше месяцев. В большинстве программистских проектов дела шли скверно скорее всего из-за нехватки календарного времени, нежели по всем другим причинам, вместе взятым.

III. ХИРУРГИЧЕСКАЯ БРИГАДА

«Эти исследования обнаружили большие различия в пределах индивидуальной производительности — иногда даже на целый порядок».

(Сакмен, Эриксон и Грант[1]))

На конференциях по программированию непрерывно раздаются голоса молодых руководителей, утверждающих, что они предпочитают иметь дело с небольшой боевой группой первоклассных специалистов, нежели вести проект, в котором заняты сотни программистов среднего уровня. Так хотелось бы и всем нам.

Однако эта наивная формулировка альтернативы уводит от ответа на тяжелый вопрос — как построить большую систему в разумный срок? Рассмотрим каждую сторону этого вопроса более подробно.

В чем проблема?

Руководители программистских проектов давно уже поняли, как значительны различия в производительности хороших и плохих программистов. Тем не менее результаты точных измерений этой величины просто поражают. В одной из своих работ Сакмен, Эриксон и Грант измеряли производительность группы опытных программистов. Даже внутри этой группы отношение максимальной и минимальной производительности в среднем составило 10:1, а для времени работы программы и затрат памяти — 5:1. Короче говоря, производительность труда программиста, получающего 20 тыс. долларов в год, может быть в 10 раз больше, чем у того, кто зарабатывает 10 тыс. долларов. Обратное также может быть справедливо. Статистические данные утверждают, что нет никакой зависимости между опытом и производительностью. (Сомневаюсь, впрочем, чтобы это было верно всегда.)

Выше я уже доказал, что увеличение числа людей, занятых в проекте, повышает его стоимость, ибо основные затраты приходятся на обеспечение связи и на устранение последствий плохой организации этой связи

(комплексная отладка). Это и приводит к стремлению руководителей максимально ограничить число людей, работающих над системой.

Действительно, опыт создания почти всех больших систем программирования показал, что политика грубой силы дорого обходится, работа оказывается медленной и малоэффективной и приводит к системам, в которых отсутствует концептуальное единство: OS/360, Exes-8, Scope-6600, Multics, TSS, SAGE — этот список можно существенно расширить.

Напрашивается вывод: если в проекте занято 200 человек и среди них — 25 руководителей, т. е. наиболее компетентных и опытных программистов, то следует уволить 175 исполнителей и заставить руководителей вернуться к программированию.

Давайте, однако, проверим это решение. С одной стороны, оно все же не соответствует идеальному представлению о *небольшой* боевой группе, которая, по общему мнению, не должна превышать 10 человек. Она слишком велика, так что потребует по меньшей мере двух уровней руководства, или около пяти руководителей. А это вызовет дополнительные потребности в финансах, сотрудниках, рабочих местах, секретарях и операторах ЭВМ.

С другой стороны, первоначальная группа в 200 человек не настолько велика, чтобы можно было сделать действительно большую систему методом грубой силы. Рассмотрим, например, операционную систему OS/360. В пиковые периоды над ней работало около 1000 человек — программисты, составители технической документации, операторы, лаборанты, секретари, руководители, вспомогательные службы и т. д. За период с 1963 по 1966 гг. около 5000 человеко-лет понадобилось на проектирование, реализацию этой системы и создание документации на нее. Нашей группе из 200 человек понадобилось бы 25 лет, чтобы довести систему до ее нынешнего состояния, да и то при условии, что люди и месяцы действительно взаимозаменяемы.

Вот тут-то и обнаруживается слабая сторона концепции маленькой энергичной группы: *это слишком медленно для действительно больших систем.* Посмотрим, что получилось бы, если бы создание операционной системы OS/360 поручили такой группе. Допустим, в ней 10 человек. Пусть благодаря энергии и опыту их

производительность в программировании и создании документации в 7 раз выше, чем у средних программистов. Допустим, что OS/360 создавалась только средними программистами (но это, впрочем, очень далеко от истины). И допустим, наконец, что еще один коэффициент повышения производительности, равный 7, появился из-за уменьшения затрат на обеспечение взаимосвязи в меньшем коллективе. Допустим, также, что состав группы не менялся все это время. Тогда $5000/(10\times7\times7)=10$; они сделают работу объемом 5000 человеко-лет за 10 лет. Будет ли эта система интересна через 10 лет? Или же она безнадежно устареет в результате быстрого развития идей и методов программного обеспечения ЭВМ?

Дилемма очень жесткая. Ради эффективности и концептуального единства проекта и его реализации хочется ограничиться небольшим коллективом. Однако для больших систем хочется найти такой путь использования значительной рабочей силы, чтобы продукт появился вовремя. Как совместить эти два требования?

Предложение Миллза

Харлан Миллз выдвинул оригинальное и творческое решение этой проблемы[2, 3]). Он предлагает, чтобы над каждой частью большой задачи работала отдельная группа, и считает, что группа должна быть организована по принципу хирургической бригады, где операцию делает один, а остальные ему ассистируют.

По некотором размышлении видно, что идея, если удастся ее осуществить, будет вполне соответствовать нашим желаниям. Две — три головы заняты проектированием и разработкой, для реализации же их идей имеется необходимое множество рук. Но сможет ли работать такая бригада? Кто будет в ней анестезиологом, медицинской сестрой, и как распределить работу? Позвольте мне, свободно обращаясь с метафорами, предложить возможный вариант такой организации.

Хирург. Миллз называет его *главным программистом*. Он сам, лично, определяет функциональные спецификации и показатели производительности, разрабатывает программу, пишет, отлаживает ее и готовит документацию. Он пользуется структурированным языком программирования типа PL/1 и имеет диалоговый

доступ к вычислительной системе, которая не только пропускает его тесты, но и хранит различные версии его программы, позволяет легко модифицировать файлы и обеспечивает редактирование текстов для его документации. Он должен обладать замечательным талантом, десятилетним опытом работы и значительными системными и прикладными навыками.

Второй пилот. Он — правая рука хирурга, его второе «я», и способен выполнить любую часть работы, но не столь опытен. Он принимает участие в разработке, обсуждении и оценке проекта вместе с хирургом, который проверяет на нем свои идеи, но не обязательно следует его советам. Второй пилот представляет свою бригаду на дискуссиях, взаимодействует с другими бригадами. Он до тонкостей знает всю программу. Он ищет альтернативные стратегии проектирования. Очевидно, что второй пилот страхует дело от несчастья с хирургом. Он может даже программировать, но не отвечает ни за одну часть машинной программы.

Администратор. Хирург является начальником, и за ним остается последнее слово по вопросам персонала, оплаты, помещений и т. д. Но он должен посвящать всему этому как можно меньше времени. Поэтому нужен профессиональный администратор, который бы занимался деньгами, людьми, машинами, а также входил в контакты с администрацией всей организации.

Бейкер считает, что администратор будет занят весь рабочий день, только если взаимоотношения пользователя с разработчиком предъявляют значительные правовые, отчетные или финансовые требования к проекту. В противном случае один администратор может обслуживать две бригады.

Редактор. Хирург отвечает за создание документации — для максимальной ясности он должен сам ее написать. Причем это справедливо и для внешних, и для внутренних описаний. Редактор же получает рукопись хирурга и критикует ее, перерабатывает, снабжает ссылками и библиографией, возится с различными версиями и наблюдает за ее размножением и распространением.

Два секретаря. И администратору, и редактору нужны свои секретари; секретарь администратора будет вести корреспонденцию проекта и архивы.

Архивариус. Он отвечает за ведение всей технической документации в бригаде. Архивариус знаком с обязанностями секретаря и в его ведении информация, хранящаяся как в машине, так и на столах программистов.

Выход на машину осуществляется через архивариуса, выдачи также попадают к нему. Самые последние распечатки хранятся в рабочем журнале; все предыдущие — в архиве, в хронологическом порядке.

В концепциях Миллза жизненно необходимым является превращение программирования «из индивидуального в общественное дело», для чего нужно сделать все результаты выходов на машину открытыми для всех членов бригады и осознать, что программы и данные являются собственностью бригады, а не частной собственностью.

В функции архивариуса входит освобождение программистов от канцелярской поденщины, причем он должен выполнять эти обычно столь неприятные обязанности систематически и с высокой производительностью, тем самым способствуя появлению наиболее ценного результата работы бригады — рабочего продукта. Очевидно, что вышеописанная схема предполагает работу в пакетном режиме. Когда используются диалоговые терминалы, в частности, те, что позволяют работать без распечаток, обязанности архивариуса не исчезают — они изменяются. Теперь он вносит в общие копии программ все изменения с отдельных рабочих копий, по-прежнему выходит на машину с пакетом программ и использует свой собственный диалоговый терминал для контроля за полнотой и доступностью растущего продукта.

Инструментальщик. Редактирование файлов, редактирование текстов и служба диалоговой отладки теперь имеются почти всюду, так что нет никакой необходимости каждой бригаде иметь свою собственную машину и группу ее обслуживания, но тем не менее нужен свой инструментарий, набор вспомогательных средств, преимущественно диалоговых. Их приобретение, эксплуатация и усовершенствование составят круг обязанностей опытного системного программиста — инструментальщика. Каждой бригаде понадобится такой инструментальщик, независимо от качества и надежности централизованного обслуживания. В своей работе он

должен руководствоваться нуждами или желаниями хирурга; требования других бригад его не касаются. В обязанности разработчика инструментария входит создание специализированных служебных программ, каталогизированных процедур, библиотек макрокоманд и т. д.

Контролер. Хирургу необходимо иметь набор подходящих тестов для отладки частей своей программы по мере ее написания и для отладки программы как единого целого. Контролер, таким образом, одновременно и враг, изобретающий системные тесты, руководствуясь функциональными спецификациями, и друг, предлагающий тестовые данные для повседневной отладки. Он должен также планировать последовательность тестов и подготавливать оснастку для комплексной отладки.

Языковед. Со времени появления языка алгол-60 выяснилось, что при работе с большинством вычислительных систем нужны один-два профессионала, посвященных во все тонкости языка программирования. Эти специалисты оказались очень полезными, к ним постоянно обращались за консультациями. Они должны обладать совершенно другими талантами, чем хирург, который по самой своей сути является разработчиком систем. Хирург мыслит образами, языковед же может найти красивый и эффективный способ использования языка в трудных, темных или запутанных ситуациях. Зачастую ему придется посвятить два-три дня небольшим исследованиям в поисках хорошего метода. Один языковед может обслуживать двоих или троих хирургов.

Итак, мы предложили способ организации группы программистов, состоящей из 10 человек, и распределение ролей внутри этой группы, используя в качестве модели хирургическую бригаду.

Как она работает? Многие идеи предложенного выше принципа организации коллектива соответствуют нашим требованиям. Десять человек, из них семь профессионалов, работают над проблемой, но система является продуктом одного человека, в крайнем случае — двоих, выступающих как одно целое.

Отметим, в частности, различия между обычным коллективом программистов из двух человек и группой «хирург — второй пилот». Во-первых, в обычных кол-

лективах вся работа разделена между сотрудниками, и каждый из них отвечает за разработку и реализацию своей части. В хирургической бригаде и хирург, и второй пилот в курсе всего проекта и всей программы. Это снимает проблему дележа памяти, обращений к дискам и т. п. И, кроме того, сохраняется концептуальное единство работы. Во-вторых, в обычных коллективах все сотрудники равны, и неизбежные различия в оценках требуют постоянных обсуждений и компромиссов. Поскольку работа и ресурсы разделены, различия в суждениях, конечно, подчинены общей стратегии и правилам взаимодействия, но они усугубляются противоположностями интересов,— например, чья часть памяти будет использоваться для буфера. В хирургической бригаде нет различий в интересах, а противоречия в мнениях разрешаются самим хирургом единолично. Эти два обстоятельства — единство задачи и связь только по принципу подчинения — позволяют хирургической бригаде действовать как одно целое.

Таким образом, строгое распределение функций между сотрудниками бригады является ключом к

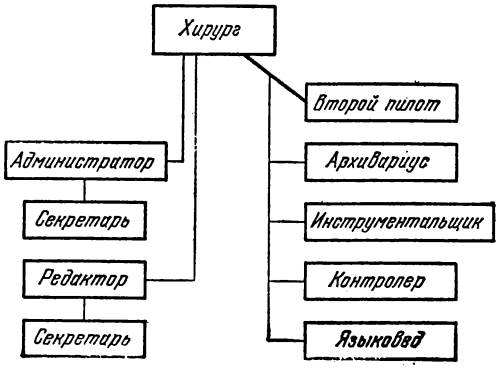


Рис. 3.1. Структура связей в коллективе программистов из 10 человек.

повышению ее производительности, поскольку оно обеспечивает гораздо более простую структуру связей между сотрудниками, как это видно на рис. 3.1.

В статье Бейкера [3]) рассказывается о проверке принципа хирургической бригады в процессе выполне-

ния одного небольшого проекта. Бригада работала так, как и предполагалось в этом случае, и с очень хорошими результатами.

«Экскалация». Пока все хорошо. Проблема, однако, заключается в том, что же делать с проектами, создание которых требует не 20 или 30, а 5000 человеколет. Группа из 10 сотрудников может работать эффективно и независимо от того, как она организована, если только *вся* задача находится в сфере ее действия. Но как использовать концепцию хирургической бригады применительно к большой задаче, в решении которой принимают участие несколько сот человек?

Залог успеха «эскалации» заключается в том, что нами уже обеспечена высокая степень концептуального единства на уровне частей — в каждой из них число умов, определяющих проект, сократилось в семь раз. Тогда можно отдать всю работу 200 сотрудникам, но проблемы координации поручить 20 хирургам.

Для проблемы координации, однако, надо использовать специальные методы, которые подробно обсуждаются в следующих разделах. Пока же достаточно сказать, что вся система тоже должна обладать концептуальным единством, а ее разработкой должен заниматься системный архитектор. Чтобы обеспечить руководство проектом, необходимо провести четкое различие между архитектурой и реализацией, и системный архитектор должен посвятить себя исключительно архитектуре. Такое распределение ролей и такая методика полностью оправдали себя и оказались очень эффективными.

IV. АРИСТОКРАТИЯ, ДЕМОКРАТИЯ И СИСТЕМНОЕ ПРОЕКТИРОВАНИЕ

«Этот великолепный храм — несравненное произведение искусства. В догмах, которые он утверждает, нет ни черствости, ни беспорядка... Это зенит стиля, работа художников, которые осознали и творчески переработали опыт своих предшественников, полностью овладели методами своего времени, но использовали их, нигде не проявляя никакой чрезмерности.

Несомненно, что именно Жан д'Орбе разработал общий план, которого придерживались, по крайней мере в важнейших чертах, его последователи. В этом заключается одна из причин полной гармонии всего сооружения».

(Путеводитель по Реймскому Собору [1]))

Концептуальное единство

Отдельные части всех европейских соборов различаются по своему стилю, потому что они создавались разными поколениями строителей. Каждое новое поколение пыталось «улучшить» старый проект в соответствии с изменениями моды или же различиями индивидуальных вкусов. Спокойный норманский трансепт примыкает к парящему готическому нефу и одновременно противоречит ему, а целое демонстрирует как славу божью, так и самоуверенную гордыню строителей.

На этом фоне архитектурное единство Реймского собора производит, по контрасту, особенно сильное впечатление. Радость, которую испытывает зритель, вызвана в равной мере как единством проекта, так и отдельными его достоинствами. Как говорит путеводитель, этого единства удалось достичь благодаря самоотречению восьми поколений строителей, каждое из которых какую-нибудь свою идею приносило в жертву чистоте замысла. Результат провозглашает не только величие бога, но и величие людей, сумевших преодолеть свою гордыню.

Хотя создание систем программирования и не занимает несколько столетий, почти все они страдают отсутствием концептуального единства, причем в гораздо большей степени, нежели соборы. Обычно это вызвано не последовательной сменой главных разработ-

чиков, а разбиением проекта на множество частей, выполняемых многими людьми.

Я продолжаю настаивать на том, что концептуальное единство является *самым* важным соображением при проектировании системы. Лучше иметь систему, не имеющую некоторых не слишком существенных свойств, но воплощающую в единое целое множество концепций проектирования, чем систему, содержащую много хороших, но независимых и нескоординированных идей.

В этой и двух последующих главах мы рассмотрим именно эту сторону создания систем программирования и попытаемся ответить на вопросы:

— Как добиться концептуального единства проекта?

— Не подразумевает ли такая постановка проблемы деления на элиту, т. е. аристократов-архитекторов, и серую массу плебеев-программистов, творческие таланты и идеи которых всячески подавляются?

— Как удержать архитектора от витания в облаках, от создания нереализуемых или просто дорогих спецификаций?

— Как добиться того, чтобы каждая, даже незначительная деталь спецификации, сделанной архитектором, дошла до исполнителя, была им правильно осмыслена и нашла свое место в конечном продукте?

Как добиться концептуального единства

Система программирования предназначена для того, чтобы облегчить пользование вычислительной машиной. С этой целью создаются машинные языки и другие различные средства, являющиеся по существу программами. Обращение к ним и управление ими тоже выполняется с помощью машинного языка. Но эти средства весьма дороги: внешнее описание системы программирования стóит в десять-двадцать раз больше, чем внешнее описание самой вычислительной системы. Пользователю гораздо легче самому определить любую требуемую функцию, чем выбрать и запомнить множество вариантов, форматов и т. п.

Пользование облегчается, если время, выигранное благодаря функциональным возможностям, больше времени, потерянного на изучение, запоминание и поиски в руководствах. В современных системах программи-

рования этот выигрыш — больше затрат, хотя за последнее время отношение выигрыша к затратам уменьшилось в связи с появлением все более сложных функций. Я все еще вспоминаю, как просто было работать на машине IBM-650, даже без ассемблера и вообще без какого бы то ни было программного обеспечения.

Поскольку простота пользования — основная цель при проектировании, то соотношение между функциональными возможностями и концептуальной сложностью является высшим критерием системного проекта. Ни функциональность, ни простота сами по себе не гарантируют его высокого качества.

Заблуждения на этот счет распространены чрезвычайно широко. Создатели операционной системы OS/360 объявили ее лучшей из всех существующих на том основании, что она выполняет больше всех функций. Функциональные возможности, а не простота, всегда считались критерием ее качества. С другой стороны, система разделения времени для PDP-10 провозглашалась ее создателями наилучшей именно из-за ее простоты и немногочисленности идей, на которых она основывается. Однако по своим функциям система PDP-10 не может быть отнесена к тому же классу, что и OS/360. Коль скоро в качестве критерия выбирается простота в пользовании, то обе эти системы соответствуют идеалу только наполовину.

На заданном уровне функциональных возможностей следует, однако, признать наилучшей ту систему, в которой различные задания выражаются с максимальной простотой и непосредственностью. Только простоты недостаточно. Язык TRAC, разработанный Муерсом, и алгол-68 достигли простоты, если ее измерять числом различных элементарных понятий. Однако они не обладают *непосредственностью*. Для выражения требуемых функций там зачастую используются весьма неожиданные и запутанные комбинации основных средств. Недостаточно выучить только элементы и правила их сочетания; необходимо знать также случаи идиоматического употребления, целый свод сведений о том, как элементы сочетаются на практике. Простота и непосредственность вытекают из концептуального единства. Каждая часть должна следовать одним и тем же принципам и одной и той же балансировке наших потребностей. Каждая часть должна использовать одну и

ту же технику синтаксиса и одинаковые понятия в семантике. Таким образом, простота в пользовании диктует требования единообразия, т. е. концептуального единства при проектировании.

Аристократия и демократия

Концептуальное единство, в свою очередь, требует, чтобы весь проект исходил из одной головы, или же из нескольких, работающих в полном согласии.

Однако график требует, чтобы система создавалась многими людьми. Существуют два метода решения этой дилеммы. Первый — это тщательное разделение труда между разработкой архитектуры и реализацией системы. Второй — это новый способ организации коллективов программистов, предложенный выше.

Отделение архитектурных проблем от реализации является весьма эффективным путем достижения концептуального единства очень больших проектов. Я убедился в этом на примере успешного создания фирмой IBM вычислительной машины Stretch и промышленной серии ЭВМ Системы 360.

Отсутствие такого подхода сказалось при разработке операционной системы OS/360.

Под *архитектурой* системы я понимаю полную и подробную спецификацию ее сопряжения с пользователем. Для вычислительной машины это руководство по программированию. Для транслятора — руководство по входному языку. Для управляющей программы — руководство по одному или нескольким языкам, используемым для обращения к ее функциям. Для системы в целом — это объединение всех тех руководств, к которым должен обращаться пользователь, чтобы решить свою задачу.

Архитектор системы, как и архитектор, проектирующий здание,— агент пользователя. Его работа заключается в том, чтобы использовать свои профессиональные и технические знания исключительно в интересах пользователя, в противоположность интересам коммивояжера, производителя и т. д.[2]).

Необходимо тщательно отделять архитектуру от реализации. Как сказал Блау, «Там, где архитектура говорит, *что* происходит, разработка говорит, *как* это должно происходить»[3]). В качестве простого примера

он приводит часы, архитектура которых — циферблат, стрелки и головка завода. Когда ребенок познакомится с этой архитектурой, он с одинаковой легкостью сможет определять время как по ручным часам, так и по башенным курантам. Разработка и реализация, однако, каждый раз описывают внутреннюю структуру — механизмы, приводящие часы в действие и обеспечивающие точность хода.

В Системе 360, например, одна машинная архитектура реализована совершенно по-разному в каждой из девяти моделей, и наоборот, одна реализация, средства обмена, память и микропрограммы модели 360/30 используются в четырех различных архитектурах: серия ЭВМ Системы 360, мультиплексный канал с 224 логическими независимыми подканалами, селекторный канал и вычислительная машина IBM-1401 [4]).

Такое разграничение в равной степени применимо и к системам программирования. Существует стандартный фортран IV. Он является архитектурой для многих трансляторов. В рамках этой архитектуры возможны самые различные реализации: программа в оперативной памяти или транслятор в оперативной памяти, быстрая трансляция или оптимизация, синтаксически ориентированный или «прямой» транслятор. Подобным образом любой язык ассемблера или язык управления задачами допускает разные реализации ассемблера или планировщика.

Теперь мы обратимся к более эмоциональной стороне проблемы: аристократия против демократии. Разве не образуют архитекторы своего рода аристократию, интеллектуальную элиту, которая указывает разработчикам, что им следует делать? Не отбирает ли эта элита всю творческую работу, отводя разработчикам роль «винтиков»? Может быть, конечный продукт улучшится, если, следуя принципам демократии, позволить всему коллективу генерировать идеи, а не ограничиваться спецификациями, разработанными всего несколькими людьми?

Последний вопрос самый легкий. Я, естественно, не собираюсь настаивать на том, что только у архитекторов могут быть хорошие архитектурные идеи. Довольно часто свежая мысль исходит от разработчика или от пользователя. Однако весь мой опыт убеждает меня в том, что именно концептуальное единство си-

стемы определяет легкость ее использования, и я попытаюсь это показать. Предлагаемые средства и идеи сами по себе могут быть очень хороши, но если они не составляют единого целого с основными концепциями системы, от них лучше отказаться. Если же таких отличных, но несовместимых идей оказалось много, остается только выбросить в мусорную корзинку всю систему и заняться созданием новой, обладающей единством и построенной на других основных идеях.

Что касается вопроса о появлении новой аристократии, то следует ответить и да, и нет.

Да — в том смысле, что архитекторов всегда немного, а конечный продукт их деятельности живет дольше, чем результаты труда разработчиков, и архитектор находится в центре всех усилий по проектированию системы, защищая интересы пользователей. Если система должна обладать концептуальным единством, то необходим кто-то, следящий за этими концепциями. Это и есть та аристократия, которая не нуждается в извинениях.

Нет — потому что разработка внешних спецификаций — ничуть не более творческая работа, чем разработка проектов. Просто это другая работа. Разработка проекта, если архитектура уже имеется, требует от исполнителей и позволяет им в той же мере проявлять творческие способности и выдавать новые идеи, как и создание внешних спецификаций. По существу отношение стоимости продукта к его производительности в основном зависит от разработчика, в то время как простота пользования зависит от архитектора.

Существует множество примеров из других областей искусства и ремесла, заставляющих поверить в то, что дисциплина полезна. Как утверждает афоризм, распространенный среди художников, «форма освобождает». Хуже всего получаются сооружения, бюджет которых больше, чем это нужно для достижения поставленной цели. Вряд ли необходимость каждую неделю писать по кантате отрицательно сказывалась на творческих возможностях Баха. Я уверен, что архитектура вычислительной машины Stretch много выиграла бы от введения более строгих ограничений; так, ограничения, накладываемые бюджетом модели 30 Системы 360, по моему мнению, положительно сказались на архитектуре модели 75.

Аналогично я подметил, что заданная архитектура повышает, а не подавляет творческие возможности группы разработчиков. Они сразу же сосредоточиваются на той части проблемы, которой еще не занимались, и в результате появляются творческие находки. Если же на работу группы не накладывается никаких ограничений, то много времени и усилий, размышлений и обсуждений уйдет на архитектурные решения, а с самой реализацией они справятся очень быстро [5]).

Существование этого эффекта, который я наблюдал неоднократно, подтверждает Р. Конвей — руководитель группы, создавшей транслятор PL/C с языка PL/1 в Корнелльском университете. Он говорит: «В конце концов, мы решили реализовать язык без всяких изменений и улучшений, потому что дебаты по этому вопросу отняли бы у нас все силы»[6]).

Чем может заполнить разработчик период ожидания?

Ошибка, стоимостью в несколько миллионов долларов, служит очень горестным уроком, но запоминается надолго. Я живо помню тот вечер, когда мы принимали решение о написании внешних спецификаций для OS/360. Руководитель группы архитекторов ЭВМ, руководитель отдела разработки управляющих программ и я бились над планом, графиком и распределением обязанностей.

В группе архитекторов было 10 отличных специалистов. Ее руководитель утверждал, что они могут написать спецификации, и сделают это хорошо. Но на это потребуется 10 месяцев, на три месяца больше, чем допускал график.

В отделе управляющих программ было 150 человек. Ее руководитель говорил, что, взаимодействуя с архитекторами, они смогут подготовить практичные спецификации высокого качества и при этом уложиться в график. Если же этим займутся архитекторы, то его 150 человек будут десять месяцев бить баклуши.

На это архитектор отвечал, что если я поручу написание спецификаций отделу управляющих программ, то мы не только не дождемся результатов вовремя, но получим их на три месяца позже и они окажутся гораздо хуже. Так и получилось. Архитектор был прав

во всех отношениях. Кроме того, отсутствие концептуального единства увеличило стоимость создания и модификации системы и, по моим оценкам, удлинило срок отладки по меньшей мере на год.

Много факторов, конечно, повлияло на принятие этого ошибочного решения, но главными были график и желание дать работу 150 разработчикам. Сирены пели мне именно эту песню, и теперь-то я вижу всю ее смертельную опасность.

Когда речь заходит о том, чтобы поручить маленькой группе архитекторов создание всех внешних спецификаций для вычислительной машины или системы программирования, разработчики выдвигают три возражения:

— спецификации будут слишком разнообразны по функциям, но не учтут практическую стоимость реализации;
— архитекторы сделают всю творческую часть работы, лишив разработчиков возможности что-либо придумать самим;
— большому числу разработчиков придется простаивать, в то время как спецификации будут проходить через игольное ушко, называемое группой архитекторов.

Первое представляет реальную опасность и подробнее будет рассматриваться в следующей главе. Два других возражения — не более, чем мираж. Как мы уже убедились, разработка — тоже творческая деятельность. Возможности для творчества и проявления изобретательности при разработке не уменьшаются сколько-нибудь значительно из-за необходимости работать с заданными внешними спецификациями, а степень проявления творческой активности может даже возрасти благодаря дисциплине. Конечный же продукт, вне всякого сомнения, от этого только выиграет.

Последнее возражение касается распределения времени и этапов работы. Первый ответ, приходящий на ум, заключается в том, что не нужно нанимать разработчиков до тех пор, пока не будут готовы спецификации. Именно так поступают при строительстве здания.

Однако в системном программировании темпы выше и график очень уплотнен. Насколько можно совместить этапы создания спецификаций и самого строительства?

Как подчеркивает Блау, в творческой деятельности выделяются три различные этапа: архитектура, разработка и реализация. На практике оказывается, что все этапы могут начинаться одновременно и проходить параллельно.

При проектировании вычислительной машины, например, разработчик уже может начинать работу, даже если он имеет довольно смутные представления о принципах действия, несколько более ясные технологические идеи и четко определенные понятия о стоимости и производительности. Он может начать проектировать информационные связи, управляющие последовательности, принципы компоновки и т. д. Он придумывает или совершенствует средства, которые ему потребуются, особенно систему документирования, включая систему автоматизации проектирования.

Одновременно может проходить разработка и устранение недостатков у интегральных схем, плат, кабелей, стоек, источников питания, запоминающих устройств и т. д., и составляться документация для них.

Эта работа выполняется параллельно с созданием архитектуры и ее реализацией.

Аналогично обстоит дело и при проектировании систем программирования. Задолго до того, как завершены спецификации, у разработчика уже много дел. Обладая некоторыми приблизительными сведениями о функциях системы, которые получат свое отражение во внешних спецификациях, он уже может действовать. Он должен точно знать поставленные ему сроки и отведенные средства. Он должен знать машину, на которой будут пропускаться его программы. Далее он может приступить к разработке сопряжения модулей, структуры таблиц, алгоритмов, распределению работы по фазам. Некоторое время следует затратить на установление контактов с архитекторами.

Кроме того, и здесь на уровне реализации также много параллельной работы. В программировании тоже есть технология. Если машина новая, то предстоит большая работа с подпрограммами, супервизором, алгоритмами поиска и сортировки [7]).

Концептуальное единство действительно требует, чтобы система отражала единую философию и чтобы спецификации в том виде, в каком они доходят до пользователя, исходили от нескольких человек. Реаль-

ное разделение труда на архитектуру, разработку и реализацию вовсе не означает, что на создание системы в целом, спроектированной таким образом, потребуется больше времени. Опыт показывает обратное, а именно, отдельные модули быстрее объединяются в систему, и на ее отладку требуется меньше времени. Таким образом, широко распространенное горизонтальное разделение труда сокращается за счет вертикального разделения труда, при этом существенно упрощается обеспечение связи и повышается концептуальное единство проекта.

V. ЭФФЕКТ ВТОРОЙ СИСТЕМЫ

«Adde parvum parvo magnus acertus erit». *«Добавляя малое к малому, получишь большую кучу».*

(Овидий)

Если ответственность за функциональные спецификации отделить от ответственности за быстрое создание дешевого конечного продукта, то как поставить пределы творческому энтузиазму архитектора?

Фундаментальное решение этой проблемы заключается в установлении тесной, последовательной и основанной на симпатиях связи между архитектором и разработчиком. Но есть и более тонкий ее аспект, обычно ускользающий от нашего внимания.

Принципы совместной работы

Архитектор сооружения занимается финансовой сметой, используя собственные методы оценки, которые позже утверждаются подрядчиком, или же последний вносит в них свои коррективы. Часто оказывается, что величина затрат, определяемая подрядчиком, не укладывается в смету. Архитектору приходится тогда пересматривать свои методы оценки с точки зрения увеличения сметы и свой проект с точки зрения его удешевления. Однако он может предложить подрядчикам способы более дешевой реализации проекта, чем разработанные ими.

В аналогичной ситуации находится архитектор вычислительной системы или системы программирования. Он обладает, однако, тем преимуществом, что подрядчики сообщают ему свои цены на гораздо более ранних этапах проекта: ахитектор может практически в любой момент потребовать от них сведения о затратах. Но он обычно испытывает неудобства из-за необходимости работать только с одним подрядчиком, который может повышать или понижать свои цены в зависимости от степени удовлетворенности проектом. На практике ранняя и постоянная связь помогает архитектору иметь нужные данные о затратах, а разработчику — осуществлять непосредственное

знакомство с проектом, при этом различия в обязанностях не стираются.

Если архитектор оказался перед проблемой завышенных оценок, у него два выхода: урезать проект или оспорить оценки, предложив более дешевую реализацию. Последний путь эмоционально очень опасен. Архитектор оспаривает метод выполнения работы строителя, предложенный самим строителем. Чтобы он привел к успеху, архитектор должен:

— помнить, что строитель несет творческую ответственность за реализацию, поэтому архитектор предлагает, но не приказывает;

— быть готовым к тому, чтобы предложить свой способ реализации продукта, спецификацию которого он написал, и к тому, чтобы принять любой другой способ, если последний отвечает поставленным задачам;

— выдвигать такие предложения спокойно, в узком кругу;

— уметь отказываться от предлагаемых улучшений.

Обычно строитель начинает противиться предлагаемым изменениям в архитектуре. И зачастую он прав — какая-нибудь незначительная деталь может оказаться неожиданно дорогой в процессе реализации.

Самодисциплина. Эффект второй системы

В своей первой работе архитектор обычно проявляет умеренность и аккуратность. Он знает, что он не знает того, что делает, а потому делает это тщательно и держит себя «в рамках».

В процессе работы над своим первым проектом ему приходят на ум всякого рода находки и украшательства. Все они откладываются «до следующего раза». Рано или поздно работа над первой системой приходит к концу, и архитектор, преисполненный уверенности и продемонстрировавший свое мастерство на системах этого класса, готов заняться второй системой.

Эта вторая система — самая опасная из всех, которые когда-либо проектирует человек. Когда он будет делать следующие, опыт прежних разработок позволит ему установить общие характеристики таких систем, а различия между ними укажут на конкретные

детали, не обобщаемые и не распространяющиеся на все системы.

Общая тенденция заключается в создании сверхпроекта второй системы путем использования всех идей и находок, от которых предусмотрительно отказались в первой. В результате, как сказал Овидий, получается «большая куча». Рассмотрим в качестве примера архитектуру ЭВМ IBM-709, позже воплотившуюся в модели IBM-7090. Это — вторая система по отношению к очень удачной IBM-704. Набор операций в модели IBM-709 настолько велик и разнообразен, что едва ли половина из них регулярно используется.

Еще более убедительный пример являет собой архитектура, разработка и даже реализация вычислительной машины Stretch, в которых нашли выход ранее сдерживаемые стремления многих людей к изобретательству, и которая была второй системой для большинства из них. Как говорит Стрейчи в своем обзоре, «*у меня создалось впечатление, что в известном смысле проект Stretch — последнее звено в цепи развития. Как и некоторые предыдущие разработки, он в высшей степени хитроумен, в высшей степени усложнен, крайне эффективен, но в то же самое время он в чем-то не продуман, расточителен и неэлегантен, и чувствуется, что существуют гораздо лучшие способы создания таких вещей*» [1]).

Операционная система OS/360 была второй системой для большинства ее разработчиков. В этот коллектив вошли создатели таких разных проектов, как операционная система DOS/1410-7010, операционная система Stretch, система реального времени в проекте Mercury и IBSYS для IBM-7090. Но почти никто из них не имел опыта создания *двух* операционных систем [2]). Таким образом, OS/360 — это чистый пример эффекта второй системы, своего рода stretch *) в искусстве системного программирования, и к ней равно относятся и похвалы, и порицания Стрейчи.

Например, в OS/360 отводится 26 байтов оперативной памяти для постоянно хранящейся в ней программы, предназначенной для обработки даты «31 декабря» в високосном году (когда 366 дней). А это можно было бы оставить оператору.

---

*) Stretche по-английски — преувеличение (*Прим. ред.*)

Эффект второй системы имеет и другое проявление, отличное от чисто функциональных украшательств. Это — тенденция к усовершенствованию методов, само существование которых становится ненужным из-за изменений в исходных посылках системы. OS/360 может дать много примеров такого рода.

Рассмотрим редактор связей, предназначенный для загрузки отдельно оттранслированных программ и обработки перекрестных ссылок. Кроме своих основных функций, он осуществляет также статистическую сегментацию программы. Это — одно из самых лучших средств сегментации, когда-либо придуманных. Оно позволяет осуществлять внешнюю сегментацию во время работы редактора связей, а не вводить ее в исходную машинную программу, что дает возможность менять структуру сегментов от прогона к прогону без повторной трансляции. Тем самым обеспечивается богатый набор полезных вариантов и возможностей. В определенном смысле это — кульминация развития метода статистической сегментации.

Но одовременно это и последний из динозавров. Дело в том, что это средство принадлежит системе, в которой мультипрограммирование является нормальным режимом работы, а динамическое распределение памяти — исходной посылкой, что находится в прямом противоречии с идеей использования статистической сегментации. Насколько лучше работала бы система, если усилия, затраченные на управление сегментацией, были бы направлены на то, чтобы осуществлять динамическое распределение памяти и динамические перекрестные ссылки по-настоящему быстро.

Далее, сам редактор связей требует такого большого объема памяти и содержит так много своих собственных сегментов, что если даже использовать его только для связи без управления сегментацией, то он работает медленнее большинства трансляторов. По иронии судьбы, сам редактор связей создавался для того, чтобы избежать ретрансляции. Как у конькобежца туловище опережает ноги, так и усовершенствования продолжаются до тех пор, пока исходные посылки системы не останутся позади.

Другим примером этой тенденции служит средство отладки Testran. Это высшая точка развития средств

пакетной отладки, обладающая действительно элегантными возможностями моментальной выдачи и распечатки памяти. Testran использует концепцию управляющей секции и простой метод избирательной трассировки и получения выдач памяти без накладных расходов или ретрансляции. Творческие концепции операционной системы Share для IBM-709 расцвели здесь пышным цветом [3]). Тем временем сама идея отладки в пакетном режиме без ретрансляции устарела. Диалоговые вычислительные системы, используя языковые интерпретаторы или интерпретирующие компиляторы, обеспечили наиболее фундаментальные преимущества. Но даже в системах пакетной обработки появление «быстро транслирующих — медленно выполняющих» трансляторов привело к тому, что предпочтение стало отдаваться методам отладки и выдачам памяти на внешнем уровне. Насколько лучше была бы система, если вместо создания Testran'а все усилия были бы направлены на разработку диалоговых средств и быстрой трансляции.

Еще одним аналогичным примером является планировщик, обеспечивающий превосходные возможности управления фиксированным потоком задач в пакетном режиме. На самом деле этот планировщик представляет собой улучшенную, усовершенствованную и приукрашенную вторую систему, созданную вслед за операционной системой DOS/1410-7010. Это — система пакетной обработки, не мультипрограммная, за исключением ввода/вывода, и предназначенная в основном для коммерческих приложений. В этом качестве планировщик OS/360 хорош. Но он почти совсем не удовлетворяет потребностям дистанционного ввода задач, мультипрограммирования и резидентных диалоговых подсистем, реализованным в OS/360.

Как архитектору избежать эффекта второй системы? Очевидно, что просто перескочить через свою вторую систему ему не удастся. Но он может помнить об опасностях этой системы и повысить самодисциплину с тем, чтобы уметь отказаться от функциональных излишеств и избежать экстраполяции тех функций, которые не сохраняются при изменении основных идей и назначения системы.

Чтобы не потерять бдительности, советуем архитектору присвоить каждой, даже самой малой, функ-

ции следующий показатель: средство x должно стоить не больше, чем m байтов памяти и n микросекунд.

Эти показатели подскажут исходные решения и в процессе реализации будут и указанием, и предостережением.

Как руководитель проекта может избежать эффекта второй системы? Во-первых, он должен проследить за тем, чтобы для главного архитектора эта система была, по меньшей мере, третьей. Кроме того, руководитель, зная о соблазнах, может вовремя проверить, насколько полно исходные концепции и поставленные задачи нашли свое отражение в подробно разработанном проекте.

VI. ПУТЬ СЛОВА

«Он сядет здесь и будет распоряжаться: Сделайте то! Сделайте это! — Но абсолютно ничто не сдвинется с места».

(Г. С. Трумен.
«О президентской власти» [1]))

Допустим, что в распоряжении руководителя есть несколько дисциплинированных и опытных архитекторов и большой коллектив разработчиков. Как руководитель сможет добиться того, чтобы все разработчики услышали, поняли и реализовали решения, предложенные архитекторами? Каким образом группа из 10 архитекторов сможет обеспечить концептуальное единство системы, которую создают 1000 человек? Целая технология, позволяющая это осуществить, была разработана для проекта Системы 360, и она равно приложима к проектам создания программного обеспечения.

Письменные спецификации — руководство

Руководство, или письменная спецификация, является необходимым инструментом, хотя и недостаточным. Руководство — это *внешняя* спецификация конечного продукта. Оно описывает каждый элемент системы с точки зрения пользователя и предписывает его поведение. И как таковое представляет собой основной результат работы архитектора.

Круг за кругом проходит процесс его подготовки, и в это время обратная связь с пользователями и разработчиками показывает, где проект неудобен для использования или реализации. Для разработчиков важно, чтобы все изменения квантовались, т. е. в графике отмечались датированные версии.

Руководство должно описывать только все то, что видит пользователь, в том числе все сопряжения; оно не должно содержать описания того, что не видно пользователю. Этим занимается разработчик, и здесь его творческая свобода не должна ничем сковываться. Архитектор всегда должен быть готов показать

пример реализации того свойства, которое он описывает, но ему не следует и пытаться навязывать *конкретную* реализацию.

Изложение должно быть точным, полным и очень подробным. Часто пользователь будет обращаться только к одному определению, поэтому каждое из них должно повторять все основные моменты, и все они должны быть согласованы между собой. Это превращает чтение руководства в весьма скучное занятие, но здесь точность важнее, чем живость изложения.

Единство «Принципов действия Системы 360» обусловлено тем, что они вышли из-под пера только двух авторов — Джерри Блау и Андриса Падегза. Идеи, в них изложенные, принадлежали примерно десятку людей, но для того, чтобы сохранить соответствие между продуктом и его описанием, превращать эти идеи в текстовые спецификации должны были один или два человека. Чтобы дать какое-нибудь определение, необходимо принять целый ряд мини-решений, которые не требуют подробного обсуждения. Примером такого рода в Системе 360 могут служить детали установления «кода условия» после каждой операции. Нетривиальным, однако, является принцип, согласно которому такие мини-решения должны быть полностью непротиворечивы.

Приложение к «Принципам действия Системы 360», написанное Джерри Блау, я считаю лучшим из всех когда-либо виденных мною руководств. В этом приложении очень точно и тщательно описываются *пределы* совместимости Системы 360, указывается, к чему следует стремиться, и перечисляются те области внешнего окружения, где архитектура умышленно хранит молчание и где результаты, полученные на разных моделях, могут отличаться друг от друга, где один экземпляр данной модели может отличаться от другого или где, наконец, система может отличаться от себя самой после каких-то технологических изменений. Авторы руководств должны стремиться именно к такому уровню точности, они обязаны определять то, чего *нельзя* делать, столь же тщательно, как и то, что можно.

Формальные описания

Английский язык, как и все другие естественные языки, нельзя считать идеальным средством для таких описаний. Поэтому создатель руководства сам должен накладывать строгие ограничения на язык с тем, чтобы достичь необходимой точности. Использование формальной системы обозначений представляется очень привлекательным выходом из этого затруднения. В конце концов точность — это основа, смысл существования формальной системы обозначений.

Давайте рассмотрим сильные и слабые стороны формальных описаний. Как уже отмечалось, формальные описания точны. Они стремятся быть максимально полными; пробелы более заметны, а потому скорее заполняются. Зато они непонятны. Когда речь идет об английской прозе, всегда можно назвать принципы ее организации, указать структуру на различных этапах или уровнях и привести примеры. Несложно перечислить исключения и выявить противоречия. И что важнее всего, можно объяснить, *почему* это так, а не иначе. Достаточно разработанные формальные описания вызывают изумление своей элегантностью и доверие — своей точностью. Но для того, чтобы их содержание можно было легко уяснить самому и объяснить другим, необходимы пояснения. По этим причинам я считаю, что будущие спецификации должны состоять как из формальных, так и текстовых описаний.

Старая пословица предупреждает: «Никогда не выходи в море с двумя хронометрами: бери один или три». Ее вполне можно отнести и к проблеме текстовых и формальных описаний. Если у вас есть и то, и другое, то одно должно быть стандартом, а второе — производным описанием, что следует указать ясно. В качестве исходного стандарта можно выбрать любое из них. Алгол-68 имеет в качестве стандарта формальное описание и, кроме того, поясняющее текстовое описание. Стандартное описание PL/1 дано текстом, а формальное описание — как производное. Система 360 также имеет текстовое описание в качестве стандарта и рядом — производное формальное описание.

Существует множество способов представления формальных описаний. Бэкусова — Наурова форма (БНФ), разработанная для описания языков, широко освещена в литературе [2]). Формальное описание PL/1 использует новую систему обозначений абстрактного синтаксиса, и она адекватно описана [3]). Язык APL, разработанный Айверсоном, применялся для описания машин, в частности, IBM-7090 [4]) и Системы 360 [5]).

Белл и Ньюэлл предложили новую систему для описания конфигураций и архитектуры машин и проиллюстрировали ее на примере нескольких машин, включая PDP-8 фирмы DEC [6]), IBM-7090 [6]) и Системы 360 [7]).

Почти все формальные описания воплощают реализацию аппаратуры или системы программного обеспечения, внешние спецификации которой они определяют. Синтаксис можно описать и без этого, но семантику обычно описывают с помощью программы, которая выполняет определяемую операцию. Это, конечно, реализация, и как таковая она диктует архитектурные решения. Необходимо указать, что формальное описание приложимо только к внешним спецификациям, и следует сказать, что они собой представляют.

Не только формальное описание является реализацией, но и реализация может служить формальным описанием. Именно этот принцип использовался при создании первых совместимых ЭВМ. Новая машина должна соответствовать старой. Какие-то места в руководстве непонятны? «Спросите машину!». Нужно было разработать тестовую программу, определяющую поведение старой машины, и добиться того, чтобы новая машина проходила через этот тест.

Программы-имитаторы аппаратуры или математического обеспечения могут использоваться точно таким же образом. Это реализация; она работает. Поэтому все вопросы описания можно разрешить путем ее проверки.

Использование реализации в качестве описания имеет некоторые преимущества. Все вопросы разрешаются однозначно посредством эксперимента. Дискуссии не нужны, потому что ответы получаются быстро. Ответы всегда точны в той степени, которая нужна, и они по определению верны. Но, кроме преимуществ, такой подход имеет и очень много недостатков. Реали-

зация может вызвать переопределение даже внешних спецификаций. Ошибка в синтаксисе в реализации приводит к любому результату; в «чистой» системе этот результат является лишь указанием на некорректность и ничем больше. В «неряшливой» системе могут появиться самые разнообразные побочные эффекты, которые могут быть использованы программистами. Когда мы предприняли эмуляцию ЭВМ IBM-1401 на Системе 360, то обнаружилось около 30 различных «курьезов», или побочных эффектов, вызываемых, предположительно, ошибочными операциями, которые, однако, стали широко использоваться и должны теперь рассматриваться как часть описания. Реализация, выступающая в качестве описания, предлагает избыточные определения; она говорит не только о том, что́ машина должна делать, но, в значительной степени, и о том, ка́к она это должна делать.

Кроме того, реализация будет иногда давать неожиданные и незапланированные ответы на трудные вопросы, это описание de facto оказывается неэлегантным в таких конкретных местах как раз потому, что они в свое время не были продуманы. Их воспроизведение в другой реализации может дорого обойтись. Например, некоторые машины не полностью очищают регистр множимого после умножения. Явление это по своей природе — часть фактического описания, однако его воспроизведение может помешать использованию более быстрого алгоритма умножения.

Наконец, использование реализации в качестве формального описания таит в себе опасную возможность перепутать, что именно является стандартом: текстовое описание или формальное. Это особенно справедливо в отношении программных имитаторов. Кроме того, нельзя вносить модификации в реализацию, пока она служит стандартом.

Прямое внесение

В распоряжении архитектора систем математического обеспечения есть превосходный метод распространения и внесения определений. Он особенно полезен для установления, если не семантики, то синтаксиса межмодульных сопряжений. Заключается этот метод в задании описания передаваемых параметров

или совместно используемой памяти, и в требовании, чтобы реализация включала данное описание через операции периода компиляции (макрокоманды или %INCLUDE в PL/1). Кроме того, если обращение ко всему сопряжению осуществляется только по символическим именам, то описание можно изменить путем добавления или введения новых переменных только ретрансляцией используемой программы без ее изменения.

Конференции и разбирательства

Нет никакой нужды говорить о том, что совещания необходимы. Кроме сотен частных консультаций, необходимы более формальные встречи. Мы считаем, что полезно их проводить на двух уровнях.

Первый — это еженедельные совещания всех архитекторов и официальных представителей разработчиков аппаратуры и математического обеспечения. Председательствует на них главный архитектор системы. Каждый может вынести на обсуждение какую-нибудь проблему или предложить изменения, но обычно все предложения распространяются в письменной форме перед совещанием. Новая проблема, как правило, какое-то время обсуждается. Причем основное внимание уделяется не принятию решений как таковых, а процессу творческого поиска. Вся группа пытается найти возможные решения проблемы, затем некоторые из этих решений передаются одному или нескольким архитекторам, которые превращают их в строго сформулированные предложения, обосновывающие изменения в документах.

Далее начинается процесс принятия решений. Предложения внимательно изучаются разработчиками и пользователями, тщательно взвешиваются все «за» и «против». Если согласие достигнуто — отлично. В противном случае решение принимает главный архитектор. Время не тратится впустую, и решения быстро и широко распространяются.

Решения, принимаемые на еженедельных совещаниях, дают быстрый результат и не тормозят работу. Если кто-либо ими *совершенно* неудовлетворен, он может немедленно апеллировать к руководителю проекта, но такое случается крайне редко.

Подобные совещания очень плодотворны по нескольким причинам:

1. Одна и та же группа архитекторов, пользователей и разработчиков встречается еженедельно в течение месяцев. Не нужно тратить время на то, чтобы ввести людей в курс дела.

2. Группа состоит из людей способных, изобретательных, хорошо разбирающихся в проблеме и глубоко заинтересованных в результатах. Ни один из них не выступает в роли «советчика». Каждый уполномочен принимать на себя обязательства.

3. Когда возникают проблемы, поиск их решения осуществляется как внутри очевидных границ, так и за их пределами.

4. Формализм письменных предложений позволяет сосредоточить внимание, стимулировать их решение и избежать противоречий.

5. Законодательная власть главного архитектора позволяет избежать компромиссов и задержек.

По прошествии времени некоторые решения устаревают. Отдельные частные способы решения не удовлетворяют того или иного участника. Другие решения вызывают непредвиденные затруднения, и иногда еженедельные совещания не в состоянии их разрешить. Так накапливается груз мелких жалоб, открытых вопросов или недовольства. Для того, чтобы справиться со всем этим, мы ежегодно осенью проводили сессии «верховного суда», которые длились обычно две недели. (Если бы мне пришлось все начинать сначала, я бы их проводил каждые шесть месяцев.)

Эти сессии проводились непосредственно перед окончательным принятием главных разделов руководств. На них присутствовала не только вся группа архитекторов, представители разработчиков и пользователей, ответственные за связь с архитекторами, но и руководители групп пользователей, сбыта и реализации. Председательствовал руководитель проекта Системы 360. Повестка дня содержала обычно около 200 пунктов, в большинстве своем мелких, которые были перечислены на плакатах, развешенных по залу. Выслушивались мнения всех сторон и принимались решения. Благодаря чудесам машинного редактирования текстов (и прекрасной организации административных служб) каждый участник находил поутру на своем

месте новый вариант руководства, уже содержащий вчерашние решения.

Эти «осенние фестивали» были ценны не только принимаемыми решениями, но и тем, что они сразу становились общим достоянием. Каждый мог высказаться, выслушать всех остальных и глубже проникнуть в суть взаимосвязей между решениями.

Совместные реализации

Архитекторы Системы 360 имели два почти беспрецедентных преимущества: достаточное время для тщательной и неторопливой работы и политическое равенство с разработчиками. Разумные сроки были обусловлены графиком выпуска новой техники; политическое равенство вытекало из факта одновременного ведения нескольких разработок. Необходимость их строгой совместимости служила наилучшей движущей силой проектирования.

В процессе создания вычислительных машин почти всегда наступает день, когда обнаруживается, что машина и документация по ее использованию не соответствуют друг другу. В последующем столкновении документ обычно терпит поражение, потому что его переделка обходится гораздо дешевле и требует меньше времени. Не так обстоит дело в случае разработки серии моделей. Тогда задержки и затраты, связанные с принятием серии плохих машин, могут перевешивать задержки и затраты на переделку машин для точного соответствия их документации.

При описании языков программирования следует иметь в виду это обстоятельство. Нужно отдавать себе отчет в том, что раньше или позже, но появятся несколько трансляторов или интерпретаторов, отвечающих различным целям. Описание будет яснее, а дисциплина — строже, если с самого начала ведутся по меньшей мере две реализации.

Журнал регистрации телефонных звонков

Как только начинается реализация, возникает бесчисленное множество вопросов, связанных с интерпретацией решений архитектора, независимо от того, насколько точны спецификации. Очевидно, что многие

такие вопросы требуют дополнений и разъяснений в самом тексте документа, другие же просто отражают недопонимание.

Необходимо всячески поощрять практику выяснения неясных мест по телефону, когда озадаченный разработчик, вместо того, чтобы действовать наугад, звонит архитектору и получает исчерпывающий ответ. Столь же необходимо осознать, что ответы архитектора на такие вопросы представляют собой вполне *официальные и авторитетные* заявления, которые следует довести до каждого.

Очень полезен *журнал регистрации телефонных звонков,* который ведет архитектор. В нем он регистрирует каждый вопрос и каждый ответ. Каждую неделю журналы нескольких архитекторов объединяются вместе, репродуцируются и распространяются среди пользователей и разработчиков. Хотя этот механизм совсем не формален, но он быстр и понятен.

Проверка конечного продукта

Лучший друг руководителя проекта — это его каждодневный противник, независимая организация, проверяющая продукт. Эта группа проверяет соответствие машин и программ их спецификациям и выступает в роли адвоката дьявола, выискивая все почти неуловимые дефекты и несоответствия. Каждой проектной организации нужна такая независимая техническая ревизионная группа, чтобы все было честно.

Независимым ревизором в последней инстанции оказывается сам заказчик. В безжалостном свете реального использования выявится каждый изъян. В таком случае группа проверки — это заменитель заказчика. Шаг за шагом опытный проверяющий найдет все места, где слово не было услышано, где проектировочные решения были неверно поняты или неаккуратно реализованы. Поэтому группа проверки является необходимым звеном в цепи, по которой проходит слово архитектора, и она должна работать одновременно и параллельно с проектом.

VII. ПОЧЕМУ ОБРУШИЛАСЬ ВАВИЛОНСКАЯ БАШНЯ

«На всей земле был один язык и одно наречие. Двинувшись с Востока, они нашли в земле Сеннар равнину и поселились там. И сказали друг другу: наделаем кирпичей и обожжем огнем. И стали у них кирпичи вместо камней, а земляная смола вместо извести. И сказали они: построим себе город и башню, высотою до небес; и сделаем себе имя, прежде нежели рассеемся по лицу всей земли. И сошел Господь посмотреть город и башню, которые строили сыны человеческие. И сказал Господь: вот, один народ, и один у всех язык, и вот что начали они делать, и не отстанут они от того, что задумали делать. Сойдем же, и смешаем там язык их так, чтобы один не понимал речи другого. И рассеял их Господь оттуда по всей земле; и они перестали строить город».

(Бытие 11.1—8)

Анализ Вавилонского проекта с точки зрения административного управления

Как считает библия, Вавилонская башня была вторым важнейшим техническим предприятием человечества после Ноева ковчега, и Вавилон был первым техническим провалом.

История о Вавилонской башне имеет глубокий смысл и очень поучительна во многих отношениях. Давайте, однако, рассмотрим ее как чисто технический проект и выясним, какие административные уроки можно из него извлечь. Были ли созданы все предпосылки для успеха? Имели ли строители:

1. *Ясную цель*? — Да, хотя достичь ее абсолютно невозможно. Однако проект провалился задолго до того, как он столкнулся с этим фундаментальным ограничением.

2. *Рабочую силу*? — Сколько угодно.

3. *Материалы*? — Глина и битум имеются в Месопотамии в избытке.

4. Достаточно *времени*? — Да, не было и речи ни о каких ограничениях на время.

5. Соответствующую *технологию*? — Да, пирамидальные или конические конструкции очень просты и хорошо выдерживают нагрузку. С каменной кладкой

строители были прекрасно знакомы. Проект потерпел крах до того, как он наткнулся на чисто технические трудности.

Итак, если у них все это было, то почему же проект провалился? Чего же не хватало строителям? У них не было, во-первых, *связи*, и, как следствие,— *организации*. Они не смогли говорить друг с другом и, следовательно, не смогли координировать свои действия. Как только прекратилась связь, застопорилась и работа. Читая между строк, мы угадываем, что отсутствие связи привело к спорам, недоброжелательству, соперничеству между группами. Короче говоря, люди начали разбредаться в разные стороны, предпочитая изоляцию ссорам и пререканиям.

Связь в больших программистских проектах

Точно то же происходит и сегодня. Неувязка с графиком, функциональные несоответствия, ошибки в системе возникают только потому, что левая рука не знает, что делает правая. В ходе работы отдельные коллективы потихоньку меняют функции, размеры и быстродействие своих собственных программ, явно и неявно пренебрегая допущениями о том, что имеется на входе, и как использовать получаемое на выходе.

Например, разработчик функции сегментации программ может, вникнув в задачу, уменьшить ее скорость, основываясь на статистических данных о том, что эта функция редко используется в прикладных программах. Тем временем, следуя за ним по пятам, его коллега может разработать основную часть супервизора так, что он будет существенно зависеть от быстродействия функции сегментации. Таким образом, изменение быстродействия само становится изменением спецификаций, о нем следует широко объявить и рассмотреть с точки зрения всей системы.

Как отдельные группы должны поддерживать связь друг с другом? — Всеми возможными путями.

— *Неформально.* При наличии хорошей организации телефонной связи и четкого определения связей внутри группы могут состояться сотни телефонных разговоров, от которых зависит общая интерпретация написанных документов.

— *Совещания.* Регулярные совещания участников проекта, где группы обмениваются технической информацией, очень важны. Благодаря им рассеиваются сотни мелких недоразумений и вопросов.

— *Рабочий документ.* Формальный рабочий документ проекта следует вести с самого начала. Однако он заслуживает отдельного раздела.

Рабочий документ проекта

Что. Рабочий документ проекта — это не столько отдельный документ, сколько структура, накладываемая на документы, выпускаемые проектом.

Все документы проекта должны входить в эту структуру. Сюда относятся цели и задачи, внешние спецификации, спецификации сопряжений, технические стандарты, внутренние спецификации и административные меморандумы.

Зачем. Техническая проза почти бессмертна. Рассматривая генеалогию технического руководства по какой-то части оборудования или математического обеспечения, можно проследить, что не только идеи, но одни и те же предложения и даже абзацы сохраняются с самого первого меморандума, предлагающего продукт или поясняющего начальный проект. Ножницы и клей нужны создателю технического документа не менее, чем авторучка.

Поскольку это так, и поскольку завтрашние руководства вырастают из сегодняшних меморандумов, очень важно сохранять правильную структуру документов. Ранняя разработка рабочего документа проекта обеспечивает продуманную, а не случайную структуру документации. Более того, установление такой структуры позволяет всему, что написано позже, придавать форму, соответствующую этой структуре.

Второй довод в пользу рабочих документов — это необходимость контроля за распространением информации. Проблема здесь заключается не в ограничении доступа к информации, а в том, чтобы обеспечить соответствующей информацией всех, кому она нужна.

Первый шаг на этом пути — перенумеровать все меморандумы с тем, чтобы каждый работник имел в своем распоряжении упорядоченный список заголовков и мог обратиться к нужному. Дальнейшая организация

рабочих документов заключается в установлении древовидной структуры меморандумов.

Механика. Как и многие другие проблемы руководства программистскими проектами, проблема технических меморандумов становится все более сложной по мере увеличения проектов. Для десяти человек документы можно просто перенумеровать. Для сотни людей будет достаточно нескольких линейных последовательностей. Для тысячи, неизбежно разбросанной по разным местам, потребность в структурированных рабочих документах возрастает, но вместе с ней растет и размер рабочих документов. Как справиться с этой проблемой?

Я думаю, что в проекте OS/360 это вполне удалось. На необходимости хорошо структурированных рабочих документов особенно настаивал О. С. Локен, который убедился в их эффективности на примере своего предыдущего проекта, операционной системы IBM 1410—7010.

Мы быстро пришли к решению, что *каждый* программист должен видеть *весь* материал, т. е. он должен иметь свою собственную копию рабочих документов.

Своевременное появление рабочих документов крайне важно. Они должны отражать ежеминутное положение дел. Но добиться этого будет очень трудно, если при внесении в документ каждого изменения его придется перепечатывать полностью. В книге со сменными листами, однако, нужно менять только отдельные страницы. В нашем распоряжении была такая система редактирования текстов на ЭВМ, значение которой для своевременного ведения документов трудно переоценить. Копии с оригинала делались тут же, на печатающем устройстве, и весь цикл подготовки новых страниц занимал меньше одного дня.

Однако человек, получивший все эти новые варианты страниц, оказывался перед проблемой сравнения. Когда ему впервые попадает страница с изменениями, он хочет знать: «Что изменилось?». Когда же он позже обращается к ней, ему нужно знать: «А как это определяется сегодня?».

Последняя потребность удовлетворяется путем непрерывного ведения документов. Для того, чтобы явно и отчетливо показать внесенные изменения, нужно предпринять другие меры. Во-первых, нужно прямо на

странице выделить текст, претерпевший изменения, например, с помощью вертикальной черты на полях против каждой измененной строки. Во-вторых, вместе с новыми страницами нужно распространять специально написанный краткий обзор, где перечисляются внесенные изменения и комментируется их значение.

Нашему проекту не исполнилось еще и полугода, как мы столкнулись с другой проблемой. Рабочие документы уже стали около 1,5 м толщиной! Если бы сложить друг на друга все 100 экземпляров, которыми пользовались наши программисты в своем здании на Манхэттене, то получилась бы башня выше самого дома. И далее, ежедневно распространялись изменения толщиной в 5 см, т. е. приблизительно 150 страниц вновь подшивалось в документ. Ведение рабочих документов стало занимать значительную часть каждого дня.

В этот момент мы переключились на микрофиши, что сэкономило нам миллион долларов, учитывая даже стоимость проекторов для чтения с микрофиш. Мы смогли прекрасно организовать производство микрофиш, объем рабочих документов сократился с 1,7 $м^3$ до 0,004 $м^3$ и, что важнее всего, изменения сразу вносились в куски по 100 страниц, тем самым в 100 раз облегчилась проблема организации документов.

Микрофиши имеют свои недостатки. С точки зрения руководителя, чем неудобнее вкладывать страницы в рабочий документ, тем больше вероятность, что изменения, в них содержащиеся, будут *прочитаны*, а именно этого он и добивается. Микрофиши могут сделать процесс ведения рабочих документов слишком легким.

Кроме того, работая с микрофишами, читатель не имеет возможности выделять в них главное, отмечать и оставлять свои комментарии. А документы, несущие на себе следы взаимодействия с читателем, более эффективны для автора и более полезны для читателя.

Тем не менее, учитывая все вышесказанное, я считаю, что микрофиши были удачной находкой, и я мог бы их рекомендовать для очень больших проектов.

Как поступать сегодня? Я считаю, что, располагая сегодняшней системной техникой, имеет смысл хранить рабочие документы в файле с непосредственным доступом и отмечать в нем все изменения и их даты.

Каждый пользователь сможет обращаться к нему с графического терминала (телетайпы слишком медленны). Ежедневно подготавливаемый обзор всех изменений будет храниться по принципу «вошедший последним выходит первым» в некоторой фиксированной точке обращения. Программист может читать его ежедневно, но если он и пропустит день, то назавтра ему просто придется прочесть больший текст. По мере чтения обзора всех изменений, он может прерываться и обращаться к самому тексту, в который они внесены.

Отметим, что сам рабочий документ при этом не изменяется. Он остается объединением всей документации по проекту и имеет тщательно разработанную структуру. Изменения претерпевают только механизм его распространения и работа с ним. Д. Энгельбарт вместе со своими коллегами из Стэнфордского исследовательского института создали такую систему и использовали ее при разработке и ведении документации по проекту создания сети ARPA.

Д. Парнас из Университета Карнеги — Меллона предложил еще более радикальное решение [1]). Он выдвинул тезис о том, что программист работает наиболее эффективно, если он не вдается в детали организации других частей системы. Это подразумевает полное и исчерпывающее определение всех сопряжений. Несмотря на то, что это звучит весьма разумно, реализация такой идеи может оказаться затруднительной. Однако хорошо организованная система информации не только выявляет ошибки в сопряжениях, но и стимулирует их исправление.

Организация в больших программистских проектах

Если в проекте занято n работников, то существует $(n^2-n)/2$ сопряжений, по которым в принципе возможна связь, и почти 2^n коллективов, внутри которых должна иметь место координация. Задача организации заключается в том, чтобы максимально уменьшить требуемый объем затрат на установление связи и координацию, следовательно, организация представляется радикальным решением вышеупомянутых проблем.

Средствами, с помощью которых уменьшаются затраты на связь, являются *разделение труда* и *специализация функций*. Древовидная структура организации

отражает все уменьшающуюся потребность в установлении контактов в условиях разделения труда и его специализации.

И действительно, дерево представляет собой структуру власти и ответственности. Принцип, что ни один человек не может служить двум хозяевам сразу, требует, чтобы структура власти имела вид дерева. Но на структуру связи не накладывается таких ограничений, и дерево является не слишком удовлетворительным приближением к структуре связи, которая представляет собой неплоскую сеть. Неадекватность чисто иерархической структуры нашла свое отражение в концепциях рабочих групп, целевых команд, комитетов и даже в организациях матричного типа, используемых во многих технических лабораториях.

Давайте рассмотрим древовидную программистскую организацию и те основные элементы, которые должны входить в каждое поддерево для обеспечения его эффективности. Сюда относятся:

1. Цели и задачи.
2. Продюсер.
3. Технический директор или архитектор.
4. График работ.
5. Разделение труда.
6. Определение сопряжений между отдельными частями.

Все это очевидно и общепринято, за исключением разницы между продюсером и техническим директором. Рассмотрим сначала две эти роли, а затем — связь между ними.

Какова роль продюсера? Он набирает бригаду, распределяет работу и устанавливает график. Он обеспечивает коллектив всеми необходимыми ресурсами и следит за их пополнением. Это означает, что его основные обязанности заключаются в установлении связей за пределами бригады, вверх и в стороны. Он устанавливает структуру связей и отчетности внутри коллектива. И, наконец, он обеспечивает выполнение графика, приводя ресурсы и организацию в соответствие с изменяющимися обстоятельствами.

А что же технический директор? Он разрабатывает проект, идентифицирует его части, определяет, как он будет выглядеть внешне и набрасывает его внутреннюю структуру. Он обеспечивает концептуальное един-

ство и целостность всего проекта и тем самым устанавливает пределы сложности системы. По мере возникновения отдельных технических проблем он обдумывает их решение или же задает проекту системы нужное направление. Он, по меткому выражению Эла Каппа *), «человек из кочегарки». Все его связи лежат в основном внутри группы. Его работа почти полностью связана с техническим содержанием проекта.

Теперь уже ясно, что эти две роли требуют совершенно разных способностей. Но эти способности проявляются в самых различных сочетаниях; конкретная комбинация, воплощенная в продюсере и директоре, должна руководить отношениями между ними. Схема организации должна выбираться в зависимости от конкретных способностей имеющихся в вашем распоряжении людей, а не наоборот; нельзя подгонять их под теоретическую схему.

Существуют три возможных типа отношений, и каждый из них вполне оправдывает себя на практике.

Одно и то же лицо может быть продюсером и техническим директором. Этот вариант прекрасно оправдывает себя в очень маленьких коллективах, от 3 до 6 программистов. Для больших проектов он малопригоден по двум причинам. Во-первых, очень трудно найти человека, обладающего одновременно талантом руководителя и незаурядными техническими способностями. Мыслители встречаются редко; деятели — реже; а мыслители-деятели — совсем редко. Во-вторых, в больших проектах каждая из этих ролей требует полной отдачи, занимая все рабочее время или даже более того. Продюсеру очень трудно снять с себя часть своих обязанностей с тем, чтобы высвободить время для чисто технических решений. А директору просто невозможно это сделать, не нарушив единства проекта.

Продюсер может быть начальником, а директор — его правой рукой. Трудность здесь заключается в представлении директору *права* принимать технические решения, не загружая его в то же время административными проблемами.

Очевидно, что продюсер должен признать права директора в принятии технических решений и поддер-

*) Э. Капп — популярный американский карикатурист. (*Прим. ред.*)

живать его власть в подавляющем большинстве случаев неизбежной проверки. Это возможно только при условии, если продюсер и директор имеют одинаковые точки зрения на основные технические проблемы. Они должны обсуждать главные вопросы еще до того, как по ним будут приняты решения. И, наконец, продюсер должен с большим уважением относиться к профессиональному мастерству директора.

Менее очевиден тот факт, что продюсер с помощью различных символов статуса (размер кабинета, ковер, мебель и т. д.) может всячески подчеркивать, что директор облечен законодательной властью, хотя формально он и не является руководителем.

Тем самым можно сделать работу очень эффективной. К сожалению, все это практикуется редко. Хуже всего, когда руководители проекта используют в качестве продюсера талантливого специалиста, который не силен в проблемах управления.

Директор может быть начальником, а продюсер — его правой рукой. В книге «Человек, продавший луну», Роберт Хейнлейн выразительно описывает такую организацию.

Ко́стер уронил голову на руки, потом вдруг поднял ее: «Я в этом разбираюсь. Я знаю, что нужно сделать — но каждый раз, когда я пытаюсь заняться технической проблемой, какой-нибудь идиот требует, чтобы я принял решение насчет грузовиков или телефонов, или другой такой же чертовщины. Простите, мистер Гарриман. Я думал, что смогу справиться со всем этим».

Гарриман сказал очень мягко: «Не принимайте все это так близко к сердцу, Боб! Вы, наверное, не высыпаетесь последнее время? Вот что я вам скажу — мы перехитрим Фергюсона. Я заберу ваш стол на пару дней и построю вам такую баррикаду, за которой никакие грузовики не страшны. Я хочу, чтобы вы могли спокойно думать о направлении реакции, КПД горячего топлива, об узких местах в проекте и не заботиться о контрактах на грузовики». Гарриман подошел к двери, выглянул в коридор и позвал какого-то человека, по виду напоминавшего старшего клерка. «Эй, Вы! Идите-ка сюда!».

Человек удивленно огляделся, подошел к двери и спросил: «Да?». «Я хочу, чтобы вот этот стол в углу, и все, что на нем, перенесли в пустой кабинет на этом же этаже, направо по коридору». Он проследил за тем, как стол и вещи Костера перенесли в новый кабинет, позаботился, чтобы там отключили телефон, и, немного подумав, велел принести туда диван. «А вечером поставим проектор, чертежный прибор, шкафы и все остальное»,—сказал он Костеру. «Вы только составьте список всего, что вам нужно, чтобы заниматься делом. Если что-нибудь еще потребуется — звоните мне». Он вернулся

в кабинет главного инженера и стал размышлять, чего же стоит его организация и что в ней не в порядке?

Часа четыре спустя он пригласил Беркли, чтобы представить его Костеру. Главный инженер спал за столом, положив голову на руки. Гарриман собирался уже уйти, но Костер проснулся. «Простите,— покраснел он,— я, наверное, задремал». «Для этого я и принес вам диван,— сказал Гарриман — на нем удобнее отдыхать. Вот, познакомьтесь с Джоком Беркли. Он ваш новый раб. Вы остаетесь главным инженером и верховным начальником, приказ которого не обсуждается. Джок — это Его Превосходительство Все Что Хотите. Отныне вам абсолютно не о чем беспокоиться — разве, что о такой малости, как создание лунного корабля».

Они пожали друг другу руки. «Я только об одном вас попрошу, г-н Костер,— сказал Беркли серьезным тоном,— переправляйте ко мне все, что хотите — в конце концов, вам командовать техническим парадом — но, ради бога, записывайте все, чтобы я был в курсе дела. У вас будет кнопка, включающая мой диктофон».

«Отлично»— Костер, как показалось Гарриману, молодел на глазах.

«А если вам попадобится что-нибудь, что не имеет отношения к проблеме, не делайте этого сами. Нажмите на кнопку и свистните — и все будет сделано». Беркли взглянул на Гарримана. «Босс говорит, что хочет обсудить с вами настоящее дело. Так что я вас покину». Он вышел.

Гарриман уселся. Костер последовал его примеру и вздохнул: «Уф!».

— Ну как, легче?

— Мне этот парень сразу понравился.

— Ну вот и хорошо, теперь он — ваша тень. Не беспокойтесь, он у меня и раньше работал. Вам покажется, что вы живете в хорошем пансионате [2]).

Этот отрывок вряд ли нуждается в комментариях. Для эффективной работы такая организация вполне приемлема.

Я считаю, что этот последний метод организации более всего пригоден для маленьких коллективов, таких, как обсуждаются в гл. III «Хирургическая бригада», в то время как продюсер в качестве верховного руководителя — это наиболее приемлемая форма организации для больших поддеревьев действительно крупного проекта.

Вавилонская башня была, возможно, первым инженерным фиаско, но не последним. Установление связи и, как ее следствие, организация наиболее важны для успеха. Методика связи и организации требуют от руководителя столько же способностей и компетенции, как и само создание математического обеспечения.

VIII. ОБЪЯВЛЕНИЕ ЦЕЛИ

«Практика — лучший учитель».
(Публилиус)

«Опыт — хороший учитель, но и он ничему не научит дурака».
(Альманах «Бедный Ричард»)

Сколько времени занимает решение задачи системного программирования? Сколько усилий потребуется? И как все это оценивать?

Выше я уже предлагал соотношения для определения затрат времени на проектирование, написание программ, отладку компонент и комплексную отладку. Во-первых, необходимо отметить, что *нельзя* оценивать задачу в целом, взяв только данные о времени, затрачиваемом на написание программ и распространив их на остальные этапы работы. Написание программ составляет примерно только одну шестую всей задачи, и ошибки в оценке этого этапа или в соотношении его с другими могут привести к смехотворным результатам.

Во-вторых, необходимо отметить, что данные, относящиеся к созданию отдельных маленьких программ, не приложимы к комплексному программному продукту. Так, например, Сакмен, Эриксон и Грант приводят среднее время на написание и отладку программы объемом около 3200 слов — 178 часов для одного программиста, что дает производительность в 35 800 команд в год. В два раза меньшая программа требует в четыре раза меньше времени, так что средняя производительность получается равной почти 80 000 команд в год [1]). Сюда следует, однако, прибавить затраты времени на проектирование, документирование, отладку, объединение в систему, обучение и всякие перерывы. Экстраполяция таких малых цифр не имеет смысла. Так, если экстраполировать время, показываемое бегуном на дистанции в 100 ярдов, то получится, что человек может пробежать милю быстрее, чем за 3 минуты.

Но прежде чем отбросить эти данные, отметим, однако, что эти числа, хотя и не годятся для строгого сравнения, показывают, что затраты растут квадратично с увеличением объема программы, *даже* когда нет

никакого взаимодействия с другими людьми, если не считать обращения человека к своей памяти.

На рис. 8.1 показаны начальные результаты исследования, проведенного Нанусом и Фарром [2]) в фирме System Development. Здесь появляется показатель степени 1,5, т. е.,

$$\text{затраты} = (\text{константа}) \times \times (\text{число команд})^{1,5}.$$

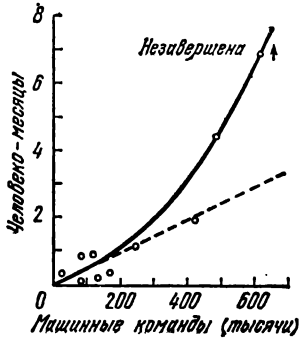


Рис. 8.1. Программирование как функция размеров программы.

В других исследованиях, проведенных этой же фирмой и опубликованных Вайнвурмом [3]), также приводится показатель степени, близкий к 1,5.

Производительность программистов неоднократно была предметом изучения, предлагались различные методы ее оценки. Моурин подготовил обзор публикаций [4]) по этой тематике. Здесь я коснусь только некоторых фактов, представляющих особый интерес.

Данные Портмана

Чарльз Портман, руководитель Северного отделения системного программирования фирмы ICL в Манчестере, высказал другую интересную точку зрения.

Он обнаружил, что его коллективы программистов не укладывались в график почти наполовину — каждая задача занимала в два раза больше времени, чем ей отводилось по плану, несмотря на то, что оценки были очень тщательны (их делали опытные люди, которые определили затраты в человеко-часах для нескольких сотен подзадач с помощью схем PERT). Когда выявилось отставание от графика, Портман попросил своих сотрудников вести подробные ежедневные записи расхода времени. Эти записи показали, что ошибочная оценка полностью объясняется тем фактом, что коллективы тратили только 50% рабочего времени собственно на программирование и отладку. Остальное время уходило на побочные, но срочные дела, совещания,

работу с бумагами, общественные дела, болезни, личные нужды, а также терялось из-за простоев машины. Короче говоря, при составлении графика было сделано нереальное предположение о том, какая часть времени затрачивается непосредственно на работу. Мой собственный опыт полностью подтверждает это заключение [6]).

Данные Арона

Джоель Арон, руководитель отдела системной технологии фирмы IBM в Гейтесбурге (штат Мэриленд), изучал производительность программистов, принимавших участие в разработке девяти больших систем (большой считалась система объемом более 30 тыс. команд, в создании которой участвовало более 25 программистов)[7]). Он классифицировал системы по числу сопряжений между программистами (и тем самым, по числу отдельных частей системы) и получил следующие значения производительности одного программиста:

Очень мало сопряжений	10 000 команд в год
Среднее число сопряжений	5000 » »
Много сопряжений	1500 » »

Но в этих данных учитывается только время на проектирование и написание программ. Если же разделить их на коэффициент два, покрывая тем самым затраты на системную отладку, то эти данные окажутся вполне сопоставимыми с данными Харра.

Данные Харра

Джон Харр, руководитель работ по программированию системы электронной коммутации фирмы Bell Telephone Laboratories обобщил свой опыт в докладе, представленном в 1969 г. на осенней объединенной конференции по вычислительной технике [8]). Эти данные приведены в табл. 8.1 и на рис. 8.2 и 8.3.

Табл. 8.1 является очень поучительной. Первые две задачи — это управляющие программы; две последние — трансляторы. Производительность определяется как число команд, отлаженных за человеко-год. Сюда включается написание программ, отладка компонент и системная отладка. Неясно, насколько здесь

учитывались затраты на проектирование, а также на обслуживание машины, документирование и т. д.

Производительность разбивается на две категории: для управляющих программ она составляет около 600

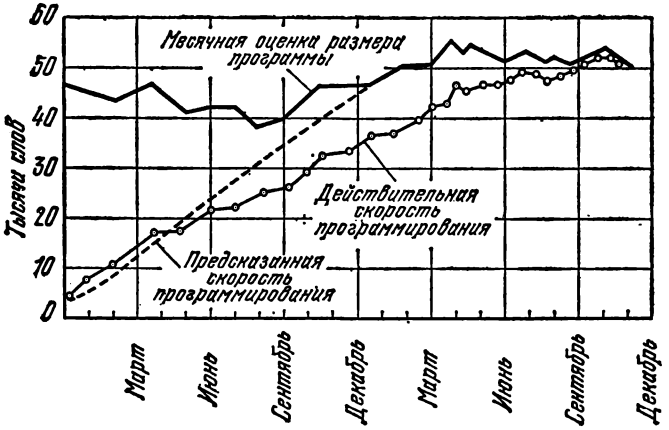


Рис. 8.2. Предсказываемая и реальная скорости программирования системы электронной коммутации.

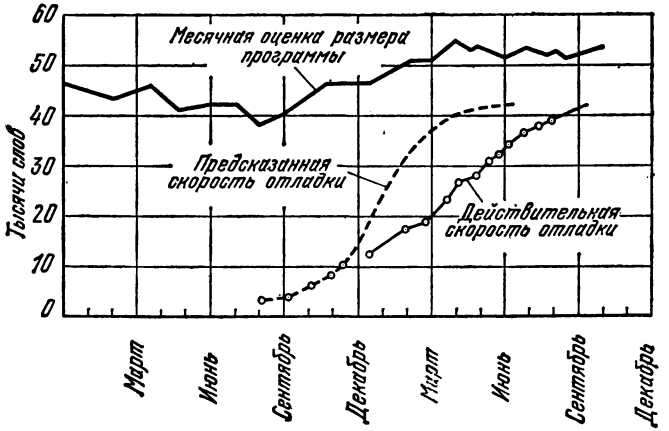


Рис. 8.3. Предсказываемая и реальная скорости отладки системы электронной коммутации.

команд на человеко-год, а для трансляторов — около 2200 команд на человеко-год. Отметим, что все четыре программы примерно одного размера — различия в ве-

личине рабочих групп, в сроках и в числе модулей. Что здесь причина и что следствие? Потому ли создание управляющих программ требует больше людей, что они сложнее? Или же требуется больше модулей и больше человеко-месяцев именно потому, что зара-

Таблица 8.1

Сводные данные по системе электронной коммутации

Тип программы	Число программ	Число программистов	Количество			Команд/человеко-год
			лет	человеко-лет	команд	
Основные	50	83	4	101	52 000	515
Сопровождение	36	60	4	81	51 000	630
Транслятор	13	9	2,25	17	38 000	2230
Ассемблер подготовки данных	15	13	2,5	11	25 000	2270

нее было больше занято людей? Потому ли требуется больше времени, что задача более сложная, или это происходит из-за большего числа занятых в ней людей? Определенный ответ дать достаточно трудно. Управляющие программы действительно более сложны. Однако если не принимать во внимание такие неопределенности, то вышеприведенные цифры описывают реальную производительность, достигнутую в большой системе с использованием современных методов программирования. И в этом аспекте они имеют достаточно важное значение.

На рис. 8.2 и 8.3 приводятся некоторые интересные данные о скорости программирования и отладки по сравнению с предсказаниями.

Данные по OS/360

Опыт разработки OS/360, не учитывавший данных Харра, тем не менее подтверждает их. Производительность групп, занимавшихся управляющими программами, составила 600 — 800 отлаженных команд на человеко-год. Производительности в 2000 — 3000 отлаженных команд на человеко-год достигли группы, разрабатывавшие языковые трансляторы. Сюда входит проектирование на уровне групп, написание программ,

отладка компонент, комплексная отладка и некоторые вспомогательные операции. Насколько я могу судить, эти данные вполне сопоставимы с данными Харра.

Данные Арона, Харра и OS/360 подтверждают существенные различия в производительности программистов в зависимости от сложности и трудности самой задачи. Чтобы не увязнуть в трясине определения этой сложности, я предлагаю придерживаться следующего правила: трансляторы в три раза сложнее обычных прикладных программ, а операционные системы в три раза сложнее трансляторов [9]).

Данные Корбато

Данные Харра и фирмы IBM относятся к программированию на языке ассемблера. Данные о производительности программирования на языках высокого уровня, кажется, еще не публиковались. Корбато из проекта MAC Массачусетского технологического института сообщает среднюю производительность в 1200 отлаженных операторов на PL/1 для системы Multics (объемом один, два миллиона команд) [10]).

Эта цифра очень показательна. Так же как и другие проекты, Multics содержит управляющие программы и трансляторы. Как и другие, он производит комплексный программный продукт, отлаженный и снабженный документацией. Поэтому данные должны быть сравнимы по затраченным усилиям. И действительно, эта производительность представляет собой хорошую среднюю величину между производительностью разработчиков управляющей программы и тех, кто писал трансляторы в других проектах.

Но цифры Корбато говорят об операторах на человеко-год, а не о командах! Каждый оператор в его системе соответствует приблизительно трем-пяти словам ручной программы. Отсюда два важных вывода:

— производительность представляется константой на уровне элементарных операторов, т. е. по отношению к затратам на осмысление операторов и к ошибкам, которые они могут содержать [11]);

— производительность программирования можно увеличить по крайней мере в пять раз путем использования подходящего языка высокого уровня [12]).

IX. ДЕСЯТЬ ФУНТОВ В ПЯТИФУНТОВОМ МЕШКЕ

«Автор должен брать пример с Ноя и научиться, как он это делал на своем ковчеге, помещать так много всего в таком малом объеме».

(Сидней Смит «Эдинбургское обозрение»)

Размер программы как стоимость

Какова стоимость программы? Если не рассматривать время выполнения программы, то именно ее объем является основным показателем ее стоимости в глазах пользователя. Это справедливо даже для арендуемых программ, где пользователь платит автору сумму, составляющую, по сути, часть стоимости разработки. Рассмотрим диалоговую систему математического обеспечения APL, разработанную фирмой IBM. Она сдается в аренду за 400 долларов в месяц и требует для своего использования по крайней мере 160 тыс. байтов памяти. В модели IBM 370/165 стоимость аренды памяти составляет 12 долларов за тысячу байтов в месяц. Если программа работает круглосуточно, то стоимость ее использования составит 400 долларов за математическое обеспечение и 1920 долларов за память. Если система APL используется только 4 часа в день, то стоимость составит 400 долларов арендной платы за математическое обеспечение и 320 долларов за память в месяц.

Нередко приходится слышать, с каким ужасом воспринимается тот факт, что в машине с памятью 2 млн. байтов операционная система может занимать 400 тыс. байтов. Ужасаться так же глупо, как критиковать Боинг 747 за то, что он стоит 27 миллионов долларов. Необходимо только задать себе вопрос: а что операционная система делает? Что можно получить за свои деньги в смысле удобства и производительности (при эффективном использовании системы)? Может быть, 4800 долларов в месяц, затрачиваемые на аренду памяти, более целесообразно истратить на другое оборудование, на программистов, на прикладные программы?

Разработчик системы превращает часть отведенных ему ресурсов оборудования в память с резидентными программами, когда он уверен, что для пользователя это лучше, чем иметь дело с сумматорами, дисками и т. д. Поступать иначе было бы в высшей степени безответственно, и результат должен оцениваться как одно целое. Нельзя критиковать систему программирования за размер *как таковой*, и в то же время постоянно выступать за все более тесное объединение проектов создания аппаратуры и математического обеспечения.

Поскольку львиная доля затрат пользователя отводится на программу, разработчик комплексного программного продукта должен установить контрольные цифры, следить за их соблюдением и придумывать методы уменьшения размеров программ точно так же, как разработчик аппаратуры устанавливает контрольные значения числа компонент, следит за их соблюдением и изобретает методы их сокращения.

Контроль за размерами программ

Управление размерами программ представляет для руководителя проекта частично техническую, а частично — административную задачу. Для установления размеров предлагаемых систем необходимо изучить потребности пользователей и те прикладные области, в которых они работают. Далее эти системы следует разбить на части и определить контрольные цифры размеров каждой компоненты. Поскольку с изменением размеров программы быстродействие меняется очень резко, установление контрольных цифр требует от руководителя профессиональной ловкости и знания допустимых соотношений в каждой части системы. Кроме того, мудрый руководитель всегда оставляет для себя лазейку, которой он сможет воспользоваться впоследствии. Хотя в OS/360 все это было проделано очень тщательно, не обошлось без болезненных уроков. Во-первых, недостаточно просто установить контрольные цифры размеров оперативной памяти, необходимо учесть все аспекты, связанные с размером программы. Большая часть предыдущих операционных систем размещалась на лентах и поскольку поиск на лентах за-

нимал много времени, это препятствовало их интенсивному вызову из сегментов программы. Система OS/360, как и ее непосредственные предшественники — операционные системы Stretch и DOS IBM 1410/7010, располагалась на дисках. Ее создатели наслаждались свободой дешевого обращения к дискам. Первоначальный результат был катастрофичным для производительности.

При установлении размеров оперативной памяти для каждой компоненты бюджет расхода памяти не устанавливался заранее. Как и предсказывали сверхпроницательные люди, программист, обнаруживая, что его программа выходит за отведенный ему участок оперативной памяти, разбивал ее на сегменты. Тем самым увеличивались общие размеры системы и уменьшалась скорость выполнения. Наша служба административного контроля не выявляла таких случаев. Каждый сообщал, какой объем оперативной памяти он использует, и если программист оставался в пределах контрольных цифр, то никто не беспокоился. К счастью, пришел день, когда начала работать программа, моделирующая производительность OS/360. Первые же результаты показали, что не все благополучно. Фортран H на машине IBM 360/65 с барабанами транслировался со скоростью пять операторов в минуту. Расследование показало, что модули управляющей программы осуществляли слишком много обращений к дискам. Даже очень часто используемые модули супервизора «ходили к колодцу с наперстками» и результат весьма напоминал непрерывную «лихорадку» памяти.

Первая мораль ясна: устанавливайте не только размеры резидентных частей, но и *общие* размеры памяти, а также вводите ограничения на число обращений к внешней памяти.

Второй урок был очень похожим. Бюджеты памяти были установлены прежде, чем было сделано точное распределение функций в каждом модуле. В результате, как только программист начинал испытывать нехватку памяти, он смотрел, нельзя ли «перелезть через забор» в память соседа. Таким образом, буферы, отведенные управляющей программе, становились частью памяти пользователя. И что еще серьезнее, такая же ситуация сложилась со всеми видами управ-

ляющих блоков, и в результате ставились под угрозу защита памяти и надежность всей системы.

Вторая мораль тоже ясна: точно определяйте функции модуля при установлении его размеров.

И третий, гораздо более глубокий, урок можно извлечь из этого опыта. Проект был достаточно велик, а административная связь достаточно плоха, так что многие члены коллектива вели себя как соперники, ломающие копья, а не как строители, совместно создающие программный продукт. Каждый старался оптимизировать свой кусок программы с тем, чтобы уложиться в контрольные цифры; однако мало кто заботился о том, во что все это выльется для заказчика. Такое нарушение ориентации и связи представляет главную опасность для больших проектов.

В течение всей реализации системные архитекторы не должны терять бдительности, чтобы сохранить концептуальное единство проекта. Кроме этих механизмов принуждения существенную роль должно сыграть и отношение самих разработчиков. Может быть, самая важная функция руководителя программистского проекта заключается в воспитании у программистов внимания к нуждам пользователя и к интересам всей системы.

Методы экономии памяти

Никакие меры по установлению бюджета расхода памяти и по контролю за его соблюдением не помогут уменьшить размера программы. Здесь нужны изобретательность и мастерство.

Очевидно, что чем больше функций, тем больший объем памяти требуется при постоянном быстродействии. Прежде всего мастерство необходимо при установлении возможных компромиссов между функциями и размерами программы. Но это вопрос очень тонкой политики. В какой мере право выбора может быть предоставлено пользователю? Можно написать программу со многими факультативными свойствами, каждое из которых требует совсем немного памяти. Можно придумать генератор, который будет располагать списком вариантов и приспосабливать программу к каждому из них. Но для каждого конкретного набора

вариантов более монолитная программа требует меньшего объема памяти. Здесь как в автомобиле: лампа, зажигалка и часы, встроенные в один блок, будут стоить дешевле, чем их отдельное исполнение. Поэтому сам разработчик должен определять степень дробления вариантов, выбираемых пользователем.

При разработке системы с широким диапазоном размеров памяти возникает другой важнейший вопрос. Существуют эффекты, ограничивающие область допустимых размеров памяти даже при очень мелком дроблении функции. Действительно, в самых малых системах большинство модулей будет в памяти перекрываться. Значительная часть резидентной памяти в таких системах должна быть выделена как буферная или страничная область, в которую попадают другие части системы. Ее величина определяет размеры всех модулей. Кроме того, распределение функций системы по мелким модулям приводит к потере и производительности, и памяти. В большой системе, где буферная область может быть в двадцать раз больше, сокращается лишь число обращений к памяти, однако остаются затруднения с быстродействием и памятью из-за малых размеров модулей. Этот эффект ограничивает максимальную эффективность системы с малыми модулями.

Другая область, требующая мастерства,— это достижение компромисса между быстродействием системы и объемом памяти. Для данной функции чем больше объем памяти, тем быстрее система. Это справедливо для поразительно широкого круга случаев. Именно этот факт делает разумным установление бюджета расхода памяти.

Руководитель, если он хочет помочь своей группе добиться хорошего соотношения между быстродействием и памятью, должен поступить следующим образом.

Во-первых, убедиться в том, что разработчики хорошо знакомы с методами программирования, а не полагаться на их природную смекалку и прежний опыт. Это особенно важно при разработке нового языка или новой машины. Следует быстро и широко распространять новые идеи или методы, всячески поощряя их появление специальными премиями или другими знаками отличия.

Во-вторых, необходимо осознать, что программирование — это техника, и компоненты нужно собирать из готовых элементов. Поэтому при работе над каждым проектом нужно иметь набор хороших подпрограмм или макрокоманд для установления очередей, поиска, расстановки и сортировки. Для каждой такой функции нужно иметь, по меньшей мере, две программы, быструю и медленную. Разработка такой техники — это важнейшая задача, которую следует осуществлять параллельно с планированием системы.

Представление данных — сущность программирования

Следом за мастерством идет изобретательность, и именно благодаря ей рождаются экономичные и быстрые программы. Почти всегда они являются результатом стратегического прорыва, а не тактической мудрости. Иногда это может быть новый алгоритм, такой как быстрое преобразование Фурье, предложенное Кули — Тюки, или замена алгоритма сортировки с n^2 сравнениями на алгоритм с $n \log n$ сравнениями.

Но гораздо чаще стратегические находки приходят в результате изменения способа представления данных или таблиц. Именно здесь лежит сердце программы. Покажите мне ваши блок-схемы, но спрячьте таблицы, и я останусь в неведении. Покажите мне ваши таблицы, и мне уже не надо смотреть блок-схемы, они и так очевидны.

Легко продолжить примеры могущества представлений. Я вспоминаю молодого человека, занимавшегося созданием сложного пультового интерпретатора для IBM 650. Он смог поместить его в неправдоподобно малый объем памяти, сделав интерпретатор для интерпретатора, поскольку контакты с человеком медленны и редки, а память была дорогой. Элегантный маленький транслятор с фортрана, созданный фирмой Digitek, использует очень сжатое специальное представление для самой программы транслятора, так что внешняя память оказывается ненужной. Время, потерянное на интерпретацию этого представления, окупается в десятикратном размере благодаря исключению ввода/вывода.

(Целый ряд таких примеров можно найти в упражнениях к шестой главе книги Брукса и Айверсона «Автоматическая обработка данных [1])», а также в упражнениях, предлагаемых Кнутом [2].)

Лучшее, что может зачастую сделать программист, оказавшийся в затруднительном положении из-за нехватки памяти,— это отвлечься от своей программы, а потом вернуться назад и пересмотреть свои данные. Представление данных — это сущность программирования.

X. ДОКУМЕНТАЦИОННАЯ ГИПОТЕЗА

Гипотеза: «В ворохе бумаги лишь небольшое количество документов становится критическими осями, вокруг которых вращается руководство каждым проектом. Они-то и являются личным инструментом руководителя».

Три фактора — технология, внешняя обстановка и традиции ремесла определяют содержание документов, которые должны появиться в проекте. Новому руководителю, не привыкшему еще к своей роли, эти документы представляются абсолютной бессмыслицей, ненужной помехой, девятым валом, грозящим его захлестнуть. И действительно, бóльшая часть их именно такова.

Однако мало-помалу он начинает осознавать, что некоторая небольшая часть этих документов воплощает в себе существенную часть его работы как руководителя. Подготовка каждого документа позволяет сосредоточить все мысли и выкристаллизовать основные идеи из обсуждений, которые в противном случае длились бы бесконечно. Их ведение становится для руководителя механизмом контроля и предупреждения. Сам по себе документ служит перечнем точек проверки, показателем положения дел и базой данных для отчетности.

Для того чтобы ознакомиться с ролью документов в проектах программного обеспечения, рассмотрим сначала конкретные документы, используемые в других областях, и выясним, возможно ли обобщение этого опыта.

Документы для разработки ЭВМ

Допустим, что создается вычислительная машина. Каковы самые важные документы?

Цели работы. Здесь определяются требования к новой машине, цели ее создания, устанавливаются ограничения и приоритеты.

Спецификации. Это система команд машины плюс спецификации производительности. С этого документа

начинается новый продукт, хотя в законченном виде он появляется последним.

График работ.

Бюджет. Один из самых полезных для руководителя документов, его функции далеко не исчерпываются установлением ограничений. Существование бюджета стимулирует технические решения, которые в противном случае могли быть и не приняты, и, что еще важнее, способствует решению вопросов общей политики.

Схема организации.

Размещение рабочих мест.

Предсказание, оценка, цены. Они находятся в циклической взаимосвязи, которая определяет успех или неудачу проекта (рис. 10.1). Чтобы сделать предсказание относительно рынка, необходимы спецификации производительности и установленные цены. Для того чтобы установить оценку производственных затрат, предсказания объединяются с расчетом компонент проекта и определяют долю труда на создание каждой единицы, а также фиксированные затраты. Эти затраты, в свою очередь, определяют цены.

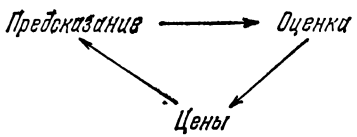


Рис. 10.1.

Если эти цены *ниже* установленных, начинает раскручиваться спираль успеха. Предсказания растут, затраты на единицу продукции падают, а цены падают еще ниже.

Если же цены оказываются *выше* установленных, то раскручивается спираль неудачи, и чтобы разорвать ее, нужно приложить немало сил. Нужно повысить производительность, разработать новые приложения, соответствующие более широким предсказаниям. Затраты следует уменьшить, чтобы дать более низкие оценки. Напряженность этого цикла создает своего рода дисциплину, зачастую вызывающую улучшение работы представителя рынка и инженера.

Это может быть также причиной смешного непостоянства. Я помню машину, в которой счетчик команд то появлялся в памяти, то исчезал, и так каждые шесть месяцев в течение трехлетнего цикла ее разра-

ботки. На одном этапе необходима была чуть более высокая производительность, тогда счетчик команд реализовывался на транзисторах. На следующем этапе вставала задача уменьшения стоимости, и этот счетчик реализовывался как ячейка памяти. В другом проекте лучший технический руководитель из всех, которых я когда-либо встречал, очень часто выступал в роли гигантского маховика: его инерция ослабляла колебания, идущие и от рынка и от руководства.

Документы для факультета университета

Несмотря на громадные различия в целях и роде деятельности, приблизительно то же число тех же документов входит в критический набор декана факультета университета. Почти каждое решение декана, ученого совета или заведующего кафедрой выражается в составлении или изменении следующих документов.

Цели работы.

Программы курсов.

Квалификационные требования.

Предложения по НИР, переходящие в планы при получении фондов.

Расписание занятий и штатное расписание.

Бюджет.

Рабочие места.

Загрузка профессорско-преподавательского состава и ассистентов.

Отметим, что вышеперечисленные пункты очень похожи на те, что уже рассматривались в проекте разработки вычислительной машины: спецификации продукта, распределение времени, распределение средств, распределение места и распределение людей. Отсутствуют только документы относительно стоимости. Такое сходство не случайно — любая задача административного руководства состоит из вопросов: что, когда, сколько, где и кто?

Документы для проекта программного обеспечения

Во многих проектах программного обеспечения люди начинают с проведения встреч для обсуждения его структуры, а затем приступают к написанию прог-

рамм. Независимо от того, каковы размеры проекта, руководитель должен немедленно начинать формализацию даже мини-документов, чтобы они могли служить ему базой данных. В конце концов оказывается, что ему нужны те же документы, что и другим руководителям.

Что: цели работы. Здесь определяются требования к новой машине, цели ее создания, устанавливаются ограничения и приоритеты.

Что: спецификации продукта. Они начинаются как предложения, а в конце уже выступают как руководство и внутренняя документация. Их важнейшей частью являются спецификации быстродействия и объема памяти.

Когда: график работ.

Сколько: бюджет.

Где: рабочие места.

Кто: схема организации. Это переплетается со спецификацией сопряжений, как предсказывает закон Конвея: «Организации, разрабатывающие системы, стремятся к созданию систем, которые являются копией структур связи в самих организациях.» Конвей [1]) придерживается точки зрения, что схема организации первоначально будет отражать первый проект системы, который почти наверняка будет плохим. Если проект системы открыт для изменений, то организация тоже должна быть к ним готова.

Зачем нужны формальные документы?

Во-первых, очень важно записывать решения. При записи выявляются всевозможные огрехи и несоответствия. Процесс записи требует принятия сотен мини-решений, а ведь именно их наличие и отличает ясную, четкую политику от смутной и неопределенной.

Во-вторых, с помощью документов решения будут сообщаться другим. Руководителя будет постоянно удивлять, что установки, предлагаемые им для общего сведения, тем не менее неизвестны некоторым членам его группы. Поскольку его основная работа сводится к тому, чтобы сохранять направление движения, его главная повседневная задача заключается не в принятии решений, а в установлении свя-

зей, и документы будут огромной поддержкой в этой работе.

И, наконец, документы дадут ему базу данных и перечень точек проверки. Периодически возвращаясь к ним, он будет видеть, в какой стадии находится проект, каким вопросам следует уделить больше внимания и в каком направлении идти дальше.

Я не разделяю столь усердно проталкиваемую коммивояжерами идею о «глобальной АСУ», когда руководитель вводит запрос в вычислительную машину, и на экране терминала загорается ответ. Существует множество очень важных причин, по которым таких систем никогда не будет. Одна из причин заключается в том, что только малая часть, примерно 20% времени руководителя уходит на задачи, для решения которых ему нужна информация не из его собственной головы. Остальное время занимают контакты: он слушает, докладывает, учит, приказывает, советует, поощряет. Для той части его работы, которая действительно базируется на внешних данных, жизненно необходимо наличие лишь небольшого количества документов, чтобы были удовлетворены почти все его потребности.

Задача руководителя — разработать план, а затем реализовать его. Но только записанный план остается точным и коммуникабельным. План состоит из документов на темы: что, когда, сколько, где и кто? Это небольшое множество основных документов воплощает в себе большую часть работы руководителя. Если их важная и ответственная роль будет осознана с самого начала, то руководитель сможет подходить к ним как к полезным инструментам, а не как к тягостным обязанностям.

XI. ПЛАН НА ВЫБРОС

«В этом мире нет ничего постояннее непостоянства».

(Свифт)

«Общепринято выбрать метод, а затем опробовать его. Если он неудачен, оставьте его и испробуйте другой. Но как бы то ни было, не оставляйте попыток».

(Ф. Д. Рузвельт[1])

Опытные установки и увеличение масштабов

Инженеры-химики давно уже знают, что процесс, который идет в лаборатории, нельзя сразу передавать на завод. Необходим промежуточный шаг, *опытная установка*, для того чтобы проверить этот процесс в больших масштабах и в условиях, приближенных к реальным. Так, например, лабораторный метод опреснения воды следует сначала проверить на опытной установке мощностью 50 тыс. л воды в день, прежде чем использовать его в городской системе опреснения мощностью 10 млн. л в день.

Создатели систем программирования тоже получали подобные уроки, но, кажется, ничего из них не извлекли. Проект за проектом разрабатывают множество алгоритмов и приступают к созданию программного обеспечения для заказчиков по планам и графикам, предусматривающим поставки первого же готового варианта.

Обычно первая система довольно мало пригодна к использованию. Она может быть слишком медленной, слишком большой, неудобной в использовании или обладать всеми этими качествами одновременно. Нет другого выхода, кроме как начать все сначала, строже подойти к проекту и создать новый вариант, в котором эти недостатки будут ликвидированы. Отказ от старого и создание нового проекта можно осуществить в один этап, можно это делать и по частям. Однако весь опыт создания больших систем говорит, что сделать это придется[2]). Каждый раз, когда используются новые концепции или новая техника, приходится создавать систему «на выброс», поскольку даже лучшие методы

планирования не настолько хороши, чтобы сразу же получалось то, что нужно.

Перед руководителем, следовательно, не стоит вопрос о том, *надо ли* создавать опытную систему, которую придется потом выбросить. Вы *должны* это сделать. Вопрос в том, нужно ли планировать с самого начала создание варианта на выброс, или же пообещать поставить этот вариант заказчику. Если есть возможность запланировать создание опытной системы, то ответ как нельзя более ясен. Поставка же заказчику варианта «на выброс» экономит время, но только ценою мук пользователя, тревог и волнений разработчиков во время создания новых проектов и ценой репутации продукта, которую трудно будет исправить даже самым лучшим новым проектом. Следовательно, *планируйте неудачу: она вас так или иначе найдет.*

Постоянны только изменения

Если вы уже осознали, что придется построить опытную систему, а затем от нее отказаться, и что повторное проектирование с изменением всех идей неизбежно, то полезно теперь без страха взглянуть на сам феномен изменений. Первый шаг заключается в том, чтобы признать факт изменения как образ жизни, а не как тягостное и досадное недоразумение. Косгроув проницательно отмечал, что программист принимает заказы скорее на удовлетворение нужд пользователя, чем на какой-либо реальный осязаемый продукт. И как только реальная потребность пользователя удовлетворена, как только программы оказались написанными, отлаженными и начали использоваться, так мы обнаруживаем, что и нужды пользователя, и его оценка этих нужд приходят в движение[3]).

Конечно, все это справедливо и в отношении потребностей, удовлетворяемых самыми разными машинами, будь то новые автомобили или новые вычислительные машины. Но само существование реального объекта сдерживает требования пользователей изменить что-либо и квантует их. А податливость программного продукта и его неосязаемость подставляют его создателей под нескончаемый поток изменений в требованиях.

Конечно, я далек от мысли, что все изменения в целях и требованиях заказчика должны, могут или могли бы быть воплощены в проекте. Следует установить какой-то порог, и чем дальше продвигаются разработки, тем выше он должен быть: в противном случае продукт так и не появится на свет.

Тем не менее, некоторые изменения в целях и задачах неизбежны, и лучше быть готовым к ним, чем считать, что их не будет. Неизбежны изменения не только в целях, но и в стратегии и методах разработки. Концепция «работы в корзину» сама по себе является принятием того факта, что, научившись чему-то, мы вносим в проект изменения [4]).

Планирование изменений в системе

Пути разработки системы, в которую легко вносить изменения, хорошо известны и широко обсуждаются в литературе — может быть даже шире, чем используются на практике. К ним относится тщательная модуляризация, широкое использование подпрограмм, точное и полное описание сопряжений между модулями и исчерпывающая документация. Менее очевидно использование стандартных вызывающих последовательностей и таблично-управляемых методов.

Для уменьшения числа ошибок, вызываемых изменениями, наиболее важным является использование языка высокого уровня и методов самодокументирования. Весьма полезным при внесении изменений в систему является использование операций периода компиляции для введения стандартных описаний.

Квантование изменений крайне важно. Каждый продукт должен иметь пронумерованные версии, и каждая версия должна иметь свой график и дату замораживания, начиная с которой изменения переходят в следующую версию.

Планирование изменений в организации

Косгроув предлагает рассматривать все планы, вехи, графики как предварительные, что облегчает их изменения. Но это, пожалуй, уж слишком — и так самая общая беда в коллективах программистов сегод-

ня заключается скорее в нехватке административного контроля, чем в его избытке.

Тем не менее Косгроув демонстрирует великолепную проницательность. Он отмечает, что отвращение к созданию проектной документации вызывается не только леностью или нехваткой времени. Оно порождается нежеланием принимать решения и отстаивать их в тех случаях, когда разработчик прекрасно знает, что они предварительны. «Подготовив документацию проекта, разработчик выставляет себя на всеобщий суд, и потому он должен быть в состоянии отстаивать все им написанное до последнего слова. Если организационная структура хоть в какой-то мере уязвима, не стоит и пытаться ничего документировать до тех пор, пока она не станет полностью обороноспособной».

Создание динамичной структуры организации гораздо труднее, чем проектирование системы, подвергаемой изменениям. Каждому исполнителю нужно отвести такие задачи, которые расширяли бы его возможности, с тем, чтобы весь коллектив был гибкой силой. В большом проекте руководитель должен иметь в своем распоряжении двух-трех лучших программистов в качестве технической кавалерии, готовой поскакать на помощь туда, где решается судьба сражения.

Структуры управления также должны меняться по мере того, как изменяется система. А это означает, что начальник должен уделять как можно больше внимания тому, чтобы его руководители и технические специалисты были взаимозаменяемыми, насколько это позволяют их способности.

Барьеры носят социологический характер, и при их преодолении следует постоянно проявлять бдительность. Во-первых, сами руководители зачастую считают ведущих специалистов «слишком ценными» для того, чтобы использовать их непосредственно в программировании. Во-вторых, работа в области административного управления имеет более высокий престиж. Чтобы справиться с этой проблемой, на некоторых предприятиях, например в фирме Bell Labs, отказались от каких бы то ни было титулов и званий. Каждый профессиональный служащий является «штатным техническим сотрудником». Другие, например, фирма IBM, имеют двойную лестницу продвижения по службе (см. рис. 11.1). Ступени теоретически эквивалентны.

Легко установить соответствия в заработной плате для каждой ступени. Гораздо труднее придать им соответствующий престиж. Кабинеты должны быть одинакового размера и одинаково обставлены. Секретариат и другие вспомогательные службы должны находиться

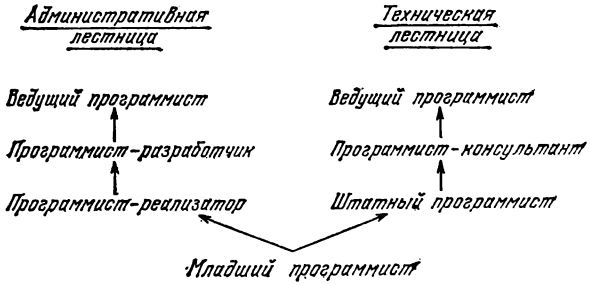


Рис. 11.1. Двойная лестница продвижения по службе в фирме IBM.

на одном уровне. Перемещение с технической лестницы на соответствующий уровень административной никогда не должно сопровождаться повышением зарплаты, и оно всегда должно объявляться именно «перемещением», а не «продвижением по службе». Обратное перемещение всегда должно вести за собой повышение зарплаты *).

Административных работников следует посылать на курсы, позволяющие им освежить технические знания, а главных технических специалистов следует обучать науке административного управления. Цели проекта, его прогресс и управленческие проблемы должны обсуждаться всеми ведущими специалистами.

Все главные специалисты должны быть профессионально и эмоционально готовы как к руководству группой, так и к тому, чтобы собственными руками написать программу. Это не так-то просто, но дело того стоит.

Сама идея организации коллективов программистов по принципу хирургической бригады является радикальной попыткой решения этой проблемы. Она позво-

*) Следует, однако, признать, что авторская оценка относительной престижности производственной и административной работы не является универсальной. (*Прим. ред.*)

ляет ведущему специалисту избавиться от ощущения, что написание программ роняет его достоинство, и пытается устранить социальные препятствия на его пути к этой творческой работе.

Кроме того, такая организация создается с целью минимизации числа сопряжений. Тем самым максимально упрощается введение изменений в систему и относительно облегчается переход всей хирургической бригады к новой программистской задаче в случае необходимости организационных изменений. Проблема гибкой организации решается, таким образом, с далеким прицелом.

Два шага вперед, шаг назад

Программа сдана заказчику, но изменения в нее продолжают вноситься. Внесение изменений в программу после ее сдачи называется сопровождением программы, однако этот процесс существенно отличается от эксплуатации оборудования.

Эксплуатация оборудования для вычислительной системы сводится к замене плохих элементов, к чистке и смазке и к внесению технических изменений, исправляющих дефекты разработки. (Не все, но большая часть технических изменений исправляет именно дефекты реализации или разработки, а не архитектуры, так что они незаметны для пользователя.)

Сопровождение программы не влечет за собой чистки, смазки или ремонта. Оно сводится в основном к внесению изменений, исправляющих дефекты разработки. Однако гораздо чаще, чем в случае оборудования, такие изменения включают в себя добавление новых функций. Обычно они прекрасно видны пользователю.

Стоимость сопровождения широко используемой программы составляет обычно 40 процентов стоимости ее разработки. Что удивительно, стоимость эта в значительной мере зависит от числа пользователей. Чем больше пользователей, тем больше они найдут ошибок.

Бетти Кэмпбелл из Лаборатории ядерной физики Массачусетского Технологического Института заметила интересный цикл жизни конкретного варианта программы. Он показан на рис. 11.2. Первоначально старые ошибки, обнаруженные в предыдущих вариантах и уже устраненные в них, проявляют тенденцию к

появлению в новом варианте. Новые функции нового варианта порождают новые ошибки. Эти ошибки вылавливаются, и несколько месяцев все идет хорошо. Но потом число ошибок опять начинает расти. Мисс Кемпбелл находит объяснение этому факту в том, что пользователи выходят на более высокий уровень и начинают полностью использовать все возможности, заложенные в новом варианте. Ценой многих усилий из этого варианта вылавливаются наиболее тонкие ошибки [5]).

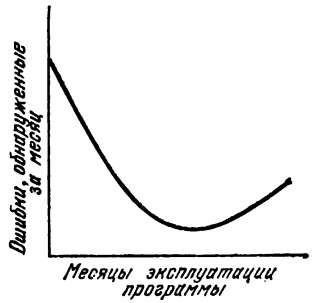


Рис. 11.2. Количество ошибок как функция времени жизни одного варианта программы.

Основная проблема сопровождения программ заключается в том, что исправление одного дефекта со значительной вероятностью (20—50%) влечет за собой появление другого. Поэтому, образно говоря, весь процесс напоминает два шага вперед и шаг назад.

Почему же дефекты не исправляются более тщательно? Во-первых, даже самый мелкий дефект проявляет себя в какой-то чисто локальной неудаче. В действительности же он зачастую распространяется на всю систему, причем неочевидным образом. Попытка исправить дефект ценой минимальных усилий затрагивает лишь только локальное и очевидное, и, несмотря на существование отличной документации, очень трудно предусмотреть все далеко идущие последствия таких исправлений. Во-вторых, исправлением обычно занимается не тот же самый человек, кто написал программу; зачастую им оказывается начинающий программист или стажер.

Вследствие появления новых ошибок сопровождение программы требует гораздо больше времени на отладку каждого написанного оператора, чем на любой другой этап программирования. Теоретически после каждого исправления нужно опять пропускать весь банк тестов, разработанных для системы, с тем чтобы убедиться, что никаким неявным образом ей не был причинен ущерб. На практике такая *регрессивная от-*

ладка должна стремиться к своему теоретическому идеалу, но это очень дорого.

Очевидно, что методы разработки программ, позволяющие устранить или по меньшей мере выявить побочные эффекты, могут оказать существенное влияние на уменьшение стоимости сопровождения, равно как и методы реализации таких разработок с меньшим числом людей, сопряжений и, следовательно, ошибок.

Шаг вперед и шаг назад

Леман и Биледи изучали историю последовательности выпуска версий большой операционной системы [6]). Они обнаружили, что общее число модулей линейно с номером версии, но что число затронутых поправками модулей растет экспоненциально с ростом номера версии. Все исправления проявляют тенденцию к разрушению структуры, увеличению энтропии и неупорядоченности системы. Все меньше и меньше усилий затрачивается на исправление первоначальных недостатков проекта, все больше и больше времени уходит на исправление ошибок, вызванных предыдущими переделками. С течением времени система становится все менее упорядоченной. Рано или поздно переделки перестанут давать какой-либо результат. За каждым шагом вперед будет следовать шаг назад. И хотя в принципе система может использоваться вечно, она устареет и не сможет больше служить основой для движения вперед. К тому же меняются машины, конфигурации, потребности пользователей, так что в действительности система не может жить вечно. Возникает необходимость качественно нового и обоснованного проекта.

И таким образом, на основе статистической механической модели Леман и Биледи пришли к выводу, подтверждаемому опытом всей жизни на земле, который также применим для систем программирования: «Вещи всегда лучше, когда они совсем еще новые»,— сказал Паскаль. Это же в более доходчивой форме сформулировал в своей книге [7]) С. Льюс.

Создание системных программ — это процесс, уменьшающий энтропию, и, следовательно, он метастабилен. Сопровождение программы — процесс, увеличивающий энтропию, и даже самое умелое осуществление его лишь отодвигает тот срок, когда система безнадежно устареет.

XII. ОСТРЫЙ ИНСТРУМЕНТ

«Хорошего работника узнают по его инструментам».

(Пословица)

Даже сейчас многие программистские проекты во всем, что касается инструментария, похожи на механические мастерские. Каждый мастер-механик имеет собственный сундучок с инструментами, вещественными доказательствами его профессионального мастерства. Он собирает такой сундучок всю свою жизнь, тщательно запирает на замок и бережет его как зеницу ока. Так и программист заводит свои маленькие редакторы, программы сортировки, распечатки, средства работы с дисками и прячет их в своем файле.

Такой подход, однако, не оправдывает себя в программистском проекте. Во-первых, основной проблемой здесь является связь, а индивидуальные сундучки с инструментами скорее препятствуют установлению связи, чем способствуют этому. Во-вторых, техника изменяется, когда меняется машина или рабочий язык, так что время жизни инструментария не велико. И, наконец, очевидно, что совместная разработка и использование универсального программистского инструментария дает гораздо больший эффект.

Однако универсальных инструментов недостаточно. Как специализированные потребности, так и персональные предпочтения обуславливают необходимость специальных инструментов, поэтому, обсуждая организацию программистской бригады, я предложил каждой бригаде иметь своего инструментальщика — человека, который разрабатывает весь общий инструментарий и может дать своему руководителю-клиенту инструкции по его использованию. Он создает также и специальные инструменты, потребовавшиеся руководителю.

Итак, руководитель проекта должен выработать установку и выделить ресурсы, необходимые для создания общего инструментария. В то же самое время он должен осознать потребность в специализированных инструментах и не возражать против их разработки в самих рабочих бригадах. Руководителю может показаться, что если всех отдельных разработчиков инструментария собрать в одну группу и поручить ей созда-

ние общих средств, выигрыш в эффективности будет очень велик. Но это не так.

О каких же инструментах должен думать руководитель? Что планировать и организовывать? Прежде всего, это *вычислительные средства:* необходимы машины и определенная целенаправленность в распределении ресурсов. Нужны *операционная система* и установленный *стиль* обслуживания. Необходим *язык* и необходимо выработать языковую политику. Сюда же относятся *вспомогательные средства, службы отладки, генераторы тестов* и *система обработки текстов* для ведения документации. Давайте последовательно все это рассмотрим [1]).

Целевые машины

Весь парк машин полезно разбить на *целевые* и *инструментальные машины.* Целевая — это та машина, для которой создается программное обеспечение и на которой оно должно в конце концов отлаживаться. Инструментальные машины — средства для создания системы. Если старая машина снабжается новой операционной системой, то она может одновременно служить как целевой, так и инструментальной.

Каковы целевые машины? Группы, разрабатывающие новые супервизоры или другое математическое обеспечение, составляющее «сердце» системы, конечно, нуждается в своей собственной машине. В таких группах потребуются операторы и один-два системных программиста, отвечающих за состояние машины.

Если нужна отдельная машина, что вообще-то бывает довольно редко, то она может не обладать предельным быстродействием, но должна иметь оперативную память объемом, как минимум, миллион байтов, еще сто миллионов байтов на дисках, и терминалы. Достаточны только алфавитно-цифровые терминалы, но они должны работать быстрее телетайпов, т. е. быстрее, чем 15 знаков в секунду.

Отладочная машина и ее программное обеспечение также должны иметь свой инструментарий с тем, чтобы можно было автоматически проводить различные измерения всевозможных параметров программы во время отладки. Контроль использования памяти, на-

пример, представляет собой эффективное средство диагностики случаев непонятного логического поведения или неожиданно низкой производительности.

График работы. Когда вы имеете дело с новой целевой машиной, для которой создается первая операционная система, то машинного времени обычно не хватает, и его распределение становится главной проблемой. Потребность во времени для целевой машины растет по своеобразному закону. При разработке OS/360 у нас были хорошие имитаторы Системы 360 и другие средства. Из опыта предыдущей работы мы представляли, сколько машинного времени нам понадобится, и потребовали от изготовителей, чтобы машины были поставлены нам заранее. Но сначала они месяц за месяцем простаивали. И вдруг сразу все шестнадцать систем оказались загружены полностью и возникла проблема распределения времени. Примерный характер изменения времени использования машин показан на рис. 12.1. Все вышли на отладку своих первых блоков одновременно, а уж затем почти каждая бригада все время что-нибудь да отлаживала.

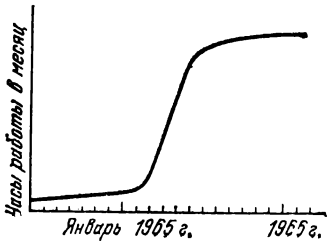


Рис. 12.1. Рост времени использования целевой машины.

Мы сначала централизовали все наши машины и библиотеку лент и поручили их обслуживание опытной профессиональной бригаде. Чтобы максимально использовать время Системы 360, мы проводили отладку в пакетном режиме на любой свободной и подходящей машине. Мы старались получать четыре выдачи в день (время оборота 2,5 ч), и настаивали на четырехчасовом обороте. Вспомогательная машина IBM 1401 с терминалами использовалась для составления графика запусков, для слежения за тысячами задач и для управления временем оборота.

Однако, от такой организации пришлось отказаться. После нескольких месяцев медленного оборота, взаимных обвинений и всеобщего недовольства мы решили распределить машинное время большими блоками. Вся группа сортировки, состоящая из 15 человек, получала,

например, систему на 4—6 ч. Они сами распределяли это время между собой. Даже если машина простаивала, никто посторонний не мог ею пользоваться.

Оказалось, что это наилучший способ распределения времени. Хотя машина использовалась несколько меньше (впрочем, не всегда), производительность возросла. Каждый член бригады может работать гораздо продуктивнее, если он получает десять выдач за шестичасовой период, чем те же десять выдач с трехчасовыми интервалами, потому что постоянная сосредоточенность почти не оставляет времени на обдумывание. После такого «рывка» бригада обычно день или два разбирает полученные результаты, а уж потом снова на шесть часов выходит на машину. Зачастую всего несколько программистов, может быть, даже только трое, распределяют между собой и плодотворно используют весь шестичасовой период. Этот способ использования целевой машины представляется наилучшим.

Так всегда было на практике, хотя никогда — в теории. Системная отладка, как астрономия, всегда была занятием для полуночников. Почти двадцать лет тому назад, занимаясь отладкой на машине IBM 701, я втянулся в работу в предутренние часы, когда в машинном зале нет начальства, спокойно спящего дома, и операторов гораздо легче склонить к нарушению правил. С тех пор сменилось три поколения вычислительных машин, техника стала совершенно другой, появились операционные системы, и только этот метод работы остался все так же предпочтительным. Он сохранился потому, что он очень продуктивен. Настало время осознать его полезность и широко распространить эту плодотворную практику.

Инструментальные машины и служба данных

Имитаторы. Если целевая машина является новой, то для нее нужен имитатор, и тогда возникает возможность приступить к отладке задолго до появления самой целевой машины. И что немаловажно, имитатор обеспечивает *достоверное* средство отладки даже после того, как появилась целевая машина.

Достоверный — это не то же самое, что *точный*. В некоторых отношениях имитатор не будет точной и

заслуживающей доверия реализацией архитектуры новой машины. Но изо дня в день это будет та же самая реализация, чего никак нельзя сказать о новом оборудовании.

Мы уже привыкли к тому, что оборудование вычислительной машины почти все время работает точно. И коль скоро прикладной программист привык к тому, что поведение системы остается неизменным при каждом запуске программы, он склонен искать ошибки в своей программе, а не в машине.

Этот опыт, однако, не применим при создании математического обеспечения для новой машины. Лабораторные варианты или первые выпуски машины *никогда* не работают так, как это заранее определяется; то, что они делают, *не всегда* надежно и, кроме того, они не остаются изо дня в день неизменяемыми. По мере того, как находятся ошибки, технические изменения вносятся во все экземпляры машины, включая и принадлежащие группе программистов. Такая нестабильная база достаточно плоха. Еще хуже периодические неполадки в оборудовании. Но хуже всего неопределенность, потому что она побуждает человека копаться в своей программе в поисках ошибки, а ее может там вовсе и не быть. Таким образом, достоверный имитатор на хорошей инструментальной машине не теряет своего значения намного дольше, чем этого можно было ожидать.

Трансляторы и ассемблеры. По тем же самым причинам нужны трансляторы и ассемблеры, работающие на достоверной инструментальной машине, но транслирующие рабочую программу для целевой системы. Затем можно переходить к отладке этой программы на имитаторе.

Программируя на языке высокого уровня, можно почти закончить отладку путем трансляции и проверки рабочей программы на инструментальной машине, еще не приступая к ее проверке в кодах целевой машины. Непосредственное выполнение как альтернатива имитации, и достоверная стабильность машины обеспечивают выигрыш в эффективности.

Библиотеки программ и учет. При создании OS/360 инструментальная машина очень успешно и с большой пользой применялась для ведения библиотек программ. Система, разработанная под руководством У. Р. Краули, состояла из двух сопряженных моделей IBM 7010,

совместно использующих большой банк данных на дисках. Машины имели также ассемблер Системы 360. Все машинные программы, как исходные, так и оттранслированные модули загрузки, уже проверенные или проходящие проверку, хранились в этой библиотеке. Библиотека делилась на подбиблиотеки с различными правилами обращения к ним.

Во-первых, каждая группа или отдельный программист имели в своем распоряжении определенную область, где хранились экземпляры его программ, варианты отладки и заготовки для комплексной отладки. На этой «*площадке для игр*» человек мог делать со своими программами все что угодно, не будучи стеснен никакими ограничениями.

Когда программист заканчивал свою часть работы и подготавливал ее к объединению с другими, он передавал экземпляры руководителю бо́льшей системы, который помещал его часть программы в *подбиблиотеку объединенной системы*. Теперь уже автор мог вносить изменения в свою программу только с разрешения руководителя объединенной системы. После того, как вся система собрана воедино, она должна была пройти через все этапы комплексной отладки.

Но вот, наконец, версия системы стала готова к более широкому использованию, после чего ее следовало перенести в *подбиблиотеку текущей версии*. Эта копия неприкосновенна, в ней можно было только фиксировать ошибки. Она доступна для использования при объединении и проверке всех новых версий модуля. Программа-справочник на машине IBM7010 хранила сведения о каждой версии каждого модуля, его статусе, местонахождении и об изменениях в них.

Здесь необходимо выделить два понятия. Первое — это *контроль*, т. е. идея принадлежности программ руководителям, которые единолично могут разрешать вносить изменения. Второе — это *формальное разделение* и *перенесение* затем с «площадки для игр» для объединения в систему.

По моему мнению, это одна из наиболее хорошо сделанных вещей в проекте OS/360. Этот метод техники административного управления был независимо выработан в нескольких больших программистских проектах, в том числе в фирме Bell Labs, ICL и в Кембриджском Университете [2]. Такая техника незаменима.

Программный инструментарий. По мере того, как появляются новые методы отладки, значение старых уменьшается, но не исчезает совсем. Поэтому нужны средства выдачи, редакторы исходных файлов, динамические выдачи памяти и даже трассировки.

Точно так же нужен полный набор средств для размещения вводимых колод на дисках, для копирования лент, распечатки файлов, внесения изменений в каталог. Если вовремя завести должность инструментальщика, то все это можно сделать заранее до того времени, когда понадобится.

Система документации. Машинная система редактирования текстов, работающая на достоверной инструментальной машине, является, может быть, самой полезной из всего инструментария и экономист больше всего труда. В нашем распоряжении была очень удобная система, разработанная Дж. Франклином. Мне кажется, что без этой системы руководства по OS/360 появились бы гораздо позже и были бы гораздо менее ясными. Говорят, что для руководств по OS/360 требуется двухметровая полка, потому что их авторы страдали недержанием речи, и что такое обилие томов придает системе своего рода непостижимость. И некоторая доля правды во всем этом есть.

Но у меня на это два возражения. Во-первых, хотя документация по OS/360 и очень велика по объему, но она снабжена продуманным планом чтения, и если использовать ее избирательно, то затраты времени сводятся к минимуму. Нужно рассматривать документацию по OS/360 как библиотеку или как энциклопедию, а не как сборник текстов для обязательного чтения.

Во-вторых, подобное изобилие гораздо предпочтительнее, чем нехватка документации, характерная для большинства систем программирования. Я охотно соглашусь, однако, с тем, что некоторые места написаны не слишком хорошо и доработка может привести к уменьшению объема. Некоторые же части написаны очень хорошо, например, «Concepts and Facilities».

Модель производительности. Лучше иметь, чем не иметь. Делайте модель по методу «снаружи — внутрь» (подробно мы рассмотрим этот метод в следующей главе). Используйте тот же самый нисходящий подход при

разработке модели производительности, логической модели и самого программного продукта. Начинайте делать модель пораньше. Прислушайтесь к тому, что она будет вам говорить.

Язык высокого уровня и диалоговое программирование

При разработке OS/360 почти десять лет тому назад не использовались два самых важных инструмента сегодняшнего системного программиста. Они и сейчас еще используются не слишком широко, хотя все указывает на их эффективность и возможность самого широкого применения. Это, во-первых, язык высокого уровня и, во-вторых, диалоговое программирование. Я убежден, что только инерция и леность мешают всеобщему распространению этих средств; технические трудности не могут более служить достаточно веским оправданием.

Язык высокого уровня. Основные доводы в пользу языка высокого уровня — производительность и быстрота отладки. Вопрос о производительности мы обсудили ранее (гл. VIII). В цифрах оценить преимущества использования такового языка очень трудно.

Убыстрение отладки связано просто с тем, что делается меньше ошибок и их легче найти. Ошибок меньше потому, что отсутствует уровень, на котором можно сделать не только синтаксические, но и семантические ошибки, например, перепутать регистры. Найти же ошибки легче потому, что этому помогают средства диагностики в трансляторе, а также потому (а это куда важнее), что легче задать отладочную выдачу.

Для меня лично соображения повышения производительности и ускорения отладки представляются непререкаемыми доводами в пользу языка высокого уровня. Трудно даже представить, что систему программирования я смог бы теперь написать на языке ассемблера.

Ну а каковы же обычные возражения против использования языка высокого уровня? Их три: я не смогу делать, что хочу; рабочая программа слишком велика; рабочая программа работает слишком медленно.

Я думаю, что первое возражение более не справедливо. Все опыты показывают, что каждый может вы-

полнить все, что ему требуется. Правда, иногда нужно крепко потрудиться, чтобы отыскать способ, как это сделать [3,4]).

Что касается размеров рабочей программы, то новые оптимизирующие трансляторы стали вполне удовлетворительными и продолжают улучшаться.

Что касается быстродействия, то оптимизирующие трансляторы ныне выдают программы, работающие быстрее, чем большинство программ, написанных в машинных кодах. Более того, проблема быстродействия обычно может быть решена следующим образом: после отладки программы, выданной транслятором, часть ее (от 1% до 5%) заменяют вручную написанной вставкой [5]).

Какой язык высокого уровня следует использовать для системного программирования? Единственным разумным кандидатом является PL/1 [6]). Он обладает очень полным набором функций и отвечает разнообразию ситуаций операционной системы. Существует множество трансляторов с этого языка, среди них есть и диалоговые, в том числе очень быстрые, обладающие также богатыми разновидностями диагностики ошибок, и, наконец, производящие хорошо оптимизированную машинную программу. Сам я считаю, что алгоритмы удобнее и быстрее разрабатывать на APL; затем я их транслирую на PL/1, чтобы обеспечить согласование с системными требованиями.

Диалоговое программирование. Одним из обоснований проекта Multics, реализованного в MIT, была его полезность при разработке систем программ. Проект Multics (и вслед за ним система TSS, созданная IBM) отличается от других диалоговых вычислительных систем как раз теми своими качествами, которые необходимы для системного программирования: несколько уровней совместного использования и защиты данных и программ, широкие возможности работы с библиотеками и средства, обеспечивающие совместную работу со многих терминалов. Я убежден, что во многих областях использования вычислительных машин системы типа Multics никогда не смогут заменить систем пакетной обработки. Но я думаю, что коллектив, создавший Multics, наиболее убедительно доказал жизнеспособность своих идей именно в приложении к системному программированию.

Пока еще существует не слишком много очевидных доказательств плодотворного использования этих, по-видимому, эффективных инструментов программиста. Широко известно, однако, что отладка — это очень трудная и медленная часть системного программирования, а медленная оборачиваемость — проклятие отладки. Поэтому логика доводов в пользу диалогового программирования представляется неоспоримой [7]).

И далее, мы слышали хорошие отзывы от многих людей, разрабатывающих таким способом небольшие системы или части систем. Единственные известные

Таблица 12.1

Сравнительная производительность программистов в пакетном и диалоговом режиме

Программы	Размер	Пакет (П) или диалог (Д)	Команды, человеко-годы
Управляющие	800 000	П	500—1000
Служебные IBM 7094	120 000	П	2100—3400
Служебные IBM 360	32 000	Д	8000
Служебные IBM 360	8 300	П	4000

мне цифры относительно программирования больших систем принадлежат Дж. Харру из фирмы Bell Labs. Они приведены в табл. 12.1. Эти цифры относятся к написанию, ассемблированию и отладке программ системы электронной коммутации. Данные Харра свидетельствуют о том, что диалоговые возможности по крайней мере удваивают производительность системного программирования [8]).

Эффективное использование большинства диалоговых средств требует, чтобы работа велась на языке высокого уровня, поскольку телетайпы и терминалы нельзя использовать для отладки путем разгрузки памяти. Используя язык высокого уровня, легко редактировать программу и получать избирательные распечатки.

Язык высокого уровня и диалоговое программирование, вместе взятые, действительно представляют собой острые инструменты.

XIII. ЦЕЛОЕ ИЗ ЧАСТЕЙ

«Я духов вызывать могу из бездны».

«И я могу и каждый это может. Вопрос лишь, явятся ль они на зов».

(Шекспир, «Генрих IV»)

Среди современных магов не перевелись хвастуны: «Я могу написать программы, которые управляют движением самолетов, перехватывают баллистические ракеты, ведут банковские счета, руководят производственными линиями». На это следует ответ: «И я могу, и каждый это может, но будут ли эти программы работать после того, как Вы их напишете?».

Как создается работоспособная программа? Как отлаживается программа? И как объединяется множество отлаженных программ-компонентов в проверенную и надежную систему? Мы эпизодически касались этих методов; теперь настала пора рассмотреть их более систематично.

Проект без ошибок

Точное задание. Наиболее коварными и зловредными оказываются системные ошибки, возникающие вследствие несогласованности основных предпосылок, выработанных авторами различных компонент. Пути достижения концептуального единства, обсуждавшиеся выше в гл. IV—VI имеют самое непосредственное отношение к решению этих проблем. Короче говоря, концептуальное единство программного продукта облегчает не только его использование, но и его создание, и обеспечивает устойчивость к ошибкам.

Подобную же роль играет подробная и тщательная разработка архитектуры, присущая этому подходу. В. А. Выссотски руководитель проекта Safeguard в фирме Bell Telephone Laboratories, утверждает: «Гвоздь проблемы состоит в том, чтобы описать продукт. Многие и многие неудачи вызваны теми его аспектами, которые не были как следует специфицированы»[1]). Аккуратное описание функций, тщательная спецификация и последовательное изгнание всяческих украшательств как в функциях, так и в методиках позволяет

сократить число системных ошибок, которые предстоит отыскать в дальнейшем.

Проверка спецификаций. Задолго до появления первых строк программы следует передать спецификации на скрупулезную проверку их полноты и ясности группе независимых специалистов. Как говорит Выссотски, сами разработчики не в состоянии этого сделать: «Они ни за что не скажут вам, что есть неясности; они будут увлеченно придумывать собственные пути через все пропасти и темные места».

Нисходящее проектирование. В 1971 г. появилась прекрасная работа Н. Вирта, в которой он формализовал процедуру проектирования, и с тех пор лучшие программисты пользуются ею. Более того, его идеи, первоначально сформулированные для разработки программ, полностью применимы к проектированию сложных систем. Разбиение процесса создания системы на архитектуру, разработку и реализацию является основополагающим в его идеях; и, далее, архитектура, разработка и реализация лучшим образом осуществляются с помощью нисходящих методов.

По Вирту проектирование рассматривается как последовательность *шагов, уточняющих проект.* Сначала грубо набрасывается постановка задачи и предлагается грубый метод ее решения, позволяющий получить принципиальный ответ. Далее описание составляется более детально, что позволяет увидеть, что результат отличается от желаемого. Большие этапы решения разбиваются на более мелкие. Каждое уточнение в постановке задачи является одновременно уточнением алгоритма ее решения, и каждое из них может сопровождаться уточнением представления данных.

В ходе этого процесса выделяются *модули* решения задачи или модули данных, и дальнейшее уточнение каждого такого модуля может производиться независимо от другой работы. Степень этой модульности определяет возможности внесения изменений в программу и подгонки ее к другим реально существующим программам.

Вирт выступает за использование на каждом этапе нотации максимально высокого уровня, позволяющей наглядно представить основные концепции, не вдаваясь в детали, пока не возникнет необходимость в дальнейшем уточнении.

Правильное применение методов нисходящего проектирования позволяет избежать появления ошибок. Во-первых, ясность, прозрачность структуры и представления облегчают точную формулировку требований и функций модулей. Во-вторых, разбиение на модули и их независимость служат гарантией от появления системных ошибок. В-третьих, из-за отказа от излишней детализации слабые места в структуре становятся более очевидными. И в-четвертых, проверка правильности проекта может осуществляться на каждом этапе его последовательного уточнения, что позволяет начать ее раньше и сосредоточить на соответствующем уровне детализации.

Процесс последовательного уточнения не гарантирует, однако, от того, что, встретив непредвиденно запутанное место, вам не придется вернуться к самому началу, отбросив верхний уровень. И такое случается часто. Но гораздо легче обнаружить точно, когда и почему пришлось отбросить проект и начать все заново. Многие плохие системы явились следствием бесплодных попыток спасти плохой проект с помощью различного рода косметических заплат. Нисходящий подход уменьшает такой соблазн.

Я убежден, что в этом десятилетии нисходящее проектирование явилось наиболее важным методом формализации программирования.

Структурированное программирование. Другой важный подход к проблеме программирования без ошибок принадлежит Дейкстре [3]) и основывается на теоретических построениях Боема и Якопини [4]).

Это метод написания программ, в которых структуры управления состоят только из циклов, определяемых оператором DO WHILE, и условных операторов, составленных из взятых в скобки групп операторов, которым предшествует условие IF... THEN... ELSE. Боем и Якопини показали теоретическую достаточность этих структур. Дейкстра убедил, что произвольное употребление перехода по оператору GO TO дает структуры, ведущие к логическим ошибкам.

Обоснованность основной идеи очевидна. Она широко обсуждалась и совершенствовалась; в частности, такие управляющие структуры, как n-значный переход (так называемый оператор CASE), и «аварийный прыжок с парашютом» (GO TO ABNORMAL END) оказа-

лись очень удобны. В то же время призыв некоторых рьяных приверженцев этого метода к отказу от всех операторов GO TO кажется чрезмерным.

Важнейшим моментом, жизненно важным для решения проблемы программирования без ошибок, является стремление рассматривать элементы управления в системе как управляющие структуры, а не как отдельные операторы перехода. Этот образ мысли является важным шагом вперед.

Автономная отладка

За последние двадцать лет процедуры отладки программ развивались по большому циклу, и в некотором смысле они сегодня вернулись туда, откуда начинали. Этот цикл прошел четыре этапа. Любопытно проследить за ними и обнаружить предпосылки каждого шага.

Машинная отладка. Первые ЭВМ имели сравнительно плохое внешнее оборудование и очень медленный ввод/вывод. Обычно машина считывала с бумажных или магнитных лент и на них же помещала результаты, а для подготовки лент и печати на них использовались автономные устройства. Они делали ввод и вывод с лент очень неудобными для отладки, поэтому вместо них стали использоваться пульты. Таким образом, отладка была организована так, что в течение одного выхода на машину имелась возможность пропустить отлаживаемую программу столько раз, сколько это удастся.

Программист тщательно разрабатывал процедуру отладки, планировал, где остановиться, какие ячейки памяти проверить, что следует ожидать там и что делать, если ожидания не оправдались. Эта неприятная необходимость программировать самого себя в качестве отладочной машины занимала почти половину времени, затрачиваемого на написание отлаживаемой машиной программы.

Самым страшным грехом считалось тогда нажатие на кнопку ПУСК без предварительного разбиения программы на части с планируемыми остановами.

Выдача памяти. Машинная отладка была очень эффективна: при выходе на машину, длившемся два часа, можно было получить до дюжины выдач. Но вычислительных машин было мало, они были очень дорогие, и

одна мысль о том, что почти все машинное время тратится впустую, приводила в ужас.

Поэтому с появлением быстродействующих печатающих устройств методика отладки изменилась. Программа пропускалась до тех пор, пока не появлялась ошибка, а затем выдавалось все содержимое памяти. Далее начиналась утомительная работа за письменным столом, проверялось содержимое каждой ячейки памяти. Времени на это уходило почти столько же, сколько при машинной отладке, но тратилось оно на расшифровку уже после прогона программы. Каждый конкретный пользователь тратил на отладку гораздо больше времени, потому что частота получения выдач зависела от оборачиваемости программы в пакете. Вся процедура, однако, организовывалась так, чтобы минимизировать количество использованного машинного времени и обслужить как можно больше программистов.

Выборочная выдача. Машины, в которых применялась выдача памяти, имели сначала 2000—4000 слов, или от 8 до 16 тыс. байтов памяти. Но размеры памяти стремительно росли, и вскоре производить выдачу всей памяти стало непрактично. Поэтому появились методы для выборочной выдачи, для избирательной трассировки и для вставки в программу команд выдачи. Testran в OS/360 может считаться последним словом в этом направлении; он позволяет вводить в программу команды выдачи без повторного ассемблирования или рекомпиляции.

Диалоговая отладка. В 1959 г. Кодд со своими коллегами [5]) и Стрейчи [6]) независимо опубликовали работы, посвященные отладке в режиме разделения времени, методу, позволяющему сочетать преимущества быстрой оборачиваемости в случае машинной отладки и эффективного использования машины при отладке в пакетном режиме.

В обеих системах в памяти вычислительной машины хранилось несколько программ, готовых к выполнению. Каждая отлаживаемая программа имела в своем распоряжении программно-управляемый терминал. Управляла отладкой программа-супервизор. Когда программист за терминалом останавливал свою программу, чтобы посмотреть результаты или внести исправления, супервизор пропускал другую программу, так что машина постоянно оставалась занятой.

Мультипрограммная система Кодда была создана, но основное внимание ее разработчики направили на увеличение производительности путем улучшения эффективности ввода/вывода, и диалоговая отладка в ней не была реализована. Идеи Стрейчи были усовершенствованы Корбато и его коллегами и реализованы в 1963 г. на экспериментальной системе для IBM 7090 в MIT[7]). Эта разработка непосредственно привела к проектам Multics, TSS и другим сегодняшним системам разделения времени.

С точки зрения пользователя основные различия между традиционной машинной отладкой и современной диалоговой отладкой сводятся к средствам, ставшим возможными благодаря появлению программы-супервизора и связанных с нею языковых интерпретаторов. Теперь можно писать программы и отлаживать их на языке высокого уровня. Эффективные средства редактирования упростили проблему внесения изменений и получения выборочных выдач памяти.

Возврат к возможностям быстрой оборачиваемости, присущим машинной отладке, не означал, однако, возврата к предварительному планированию отладки. В определенном смысле такое предварительное планирование уже не столь необходимо, как раньше, потому что машинное время не тратится впустую, пока программист сидит и размышляет.

Тем не менее, известны интересные экспериментальные данные Гоулда[8]), показывающие, что при диалоговой отладке наиболее успешным бывает первый диалог каждого сеанса. Это позволяет с уверенностью утверждать, что мы не реализуем всех потенциальных возможностей диалога именно из-за отсутствия плана отладки. Настало время вернуть к жизни старые методы машинной отладки.

Я считаю, что плодотворное использование хорошей терминальной системы требует, чтобы работе за столом посвящалось, как минимум, два часа после каждого двухчасового сеанса за терминалом. Половина этого времени обычно уходит на обработку последнего сеанса: на ведение личного журнала отладки, на занесение новых распечаток программы в личную тетрадь, на объяснение странных явлений. Вторая половина посвящается подготовке: планированию изменений и усовершенствований, разработке подробных тестов на сле-

дующий раз. Без такого предварительного планирования трудно производительно работать все два часа сеанса. А без «домашнего анализа» трудно обеспечить систематическое движение вперед от одного сеанса к другому.

Тесты. Что касается разработки реальных процедур отладки и тестов, то все это очень хорошо изложено в работе Груенбергера [9]), хотя есть и менее подробные изложения в некоторых других учебниках [10,11]).

Системная отладка

Системная отладка неожиданно оказалась очень трудной частью процесса создания системы программирования. Я уже рассматривал некоторые причины как сложности этой проблемы, так и ее непредвиденности. Из всего сказанного необходимо усвоить две вещи: отладка системы будет длиться дольше, чем это ожидается, и ее трудность можно преодолеть посредством крайне систематичного и планируемого подхода. Давайте посмотрим, в чем заключается этот подход [12]).

Использование отлаженных компонент. Если не общепринятая практика, то здравый смысл требуют, чтобы отладка системы начиналась только после того, как начали работать ее части.

Обычная практика отступает от этого принципа по двум направлениям. Первый подход гласит: «свяжите все это вместе и опробуйте». Он основывается на идее о том, что в дополнение к ошибкам в компонентах появятся ошибки в системе (т. е. в сопряжениях). И чем раньше все части будут связаны воедино, тем раньше выявятся системные ошибки. Более примитивная идея сводится к тому, что, используя отдельные части системы для проверки друг друга, можно сэкономить на подготовке тестов. Все это так, однако опыт показывает, что не совсем так,— использование готовых, отлаженных компонент экономит вполне достаточно времени на системной отладке, чтобы использовать его на подготовку тестов и на тщательную отладку компонент.

Несколько более смутная идея «известных ошибок» сводится к тому, что можно начинать комплексную отладку системы уже после того, как в компонентах *найдены* все ошибки, но прежде, чем они будут *исправ-*

лены. Тогда, согласно этой теории, при отладке системы все ожидаемые эффекты таких ошибок уже известны, так что их можно проигнорировать, сосредоточить внимание на новых явлениях.

Но это лишь иллюзия, попытка обосновать отставание от графика. Никто *не может* предвидеть всех эффектов, вызываемых уже известными ошибками. Если бы эта связь была столь непосредственной, отладка системы не была бы таким трудным делом. Более того, исправление известных ошибок обязательно приведет к появлению новых, и тогда отладка системы совершенно запутается.

Оснастка. Под «оснасткой» я понимаю все программы и данные, подготавливаемые для проведения отладки, но никогда не появляющиеся в конечном продукте. Нет ничего страшного, если объем таких программ примерно равен половине объема всей системы.

Один из таких приемов — *фиктивная компонента*, которая включает в себя только сопряжения, и, может быть, какие-то придуманные данные или несколько тестов. Например, в системе может быть программа сортировки, которая еще не закончена. Тогда соседние программы можно отлаживать с использованием фиктивной программы, которая только читает и проверяет формат входных данных и выдает набор бессмысленных, но упорядоченных данных правильного формата.

Второй прием — это *мини-файл.* Очень распространенной ошибкой является неправильное понимание форматов файлов на ленте и на диске. Поэтому имеет смысл создавать маленькие файлы, содержащие только несколько типичных записей, но зато все описания, указатели и т. д.

Предельным случаем мини-файла является *фиктивный файл*, которого в действительности вовсе нет. Язык управления задачами (ЯУЗА) в OS/360 обеспечивает такое средство, крайне полезное при отладке компонент.

Еще одна сторона оснастки — это *вспомогательные программы.* Генераторы тестовых данных, специальные анализирующие распечатки, анализаторы таблиц перекрестных ссылок — все это примеры той специальной «арматуры», которую вы можете подготавливать по своему желанию [13]).

Контроль за изменениями. Строгий контроль во время испытаний — это один из наиболее впечатляю-

щих методов отладки оборудования, и он в равной мере необходим и в системах программного обеспечения.

Прежде всего кто-то должен быть ответственным. Этот кто-то, и только он один, должен давать разрешения на внесение изменений в компоненты или на замену одного варианта другим. Затем, как уже указывалось выше, следует иметь контрольные копии системы: одну неизменяемую копию последней версии, использованной для отладки компонент; одну проверяемую копию, в которую вносят исправления; для каждого программиста — копии, с которыми он может работать в своей части системы, исправляя ошибки или осуществляя расширения.

В инженерных моделях Системы 360 среди обычной желтой проволоки вдруг оказывались ярко-красные витки. Дело в том, что когда обнаруживалась ошибка, инженеры делали две вещи. Быстро придумывался способ ее исправления, который и вносился в систему с тем, чтобы проверка могла продолжаться. Это исправление делалось красной проволокой и выделялось, как зияющая рана. Оно вносилось в журнал. Тем временем подготавливался официальный документ об изменениях и закладывался в мельницу автоматизации проектирования. Наконец, появлялись исправленные чертежи и монтажные схемы и новая панель с изменениями, реализованными в печатных схемах или желтой проволокой. Теперь физическая модель и ее описание опять соответствовали друг другу, и красная проволока убиралась.

В программировании нужен свой метод красной проволоки, крайне необходим строгий контроль и глубокое уважение к бумаге, которая в конечном счете представляет собой основной продукт нашей деятельности. К жизненно важным составляющим этого метода относится внесение всех изменений в журнал и различие, четко проводимое в исходной программе, между поспешным «приклеиванием заплат» и тщательно продуманными, проверенными и документированными исправлениями.

Не более одной компоненты за раз! Эта заповедь вполне очевидна, но оптимизм и леность заставляют нас пренебрегать ею. Чтобы ей следовать, нужны фиктивные программы и другие вспомогательные средства, а это требует дополнительного труда. И, в конце кон-

цов, может быть эта работа вовсе не нужна? Может быть, ошибок-то и нет?

Но не поддавайтесь соблазну! Тщательная отладка системы сводится именно к этой заповеди. Предполагайте, что ошибок будет очень много, и планируйте последовательную процедуру их вылавливания.

Отметим, что необходимо иметь хорошие тесты, проверяющие незаконченную систему после добавления к ней каждого нового куска, и старые, успешно работавшие в последнем неполном варианте части, которые следует перепроверять при каждом добавлении.

Квантуйте изменения! Когда система уже начнет работать, разработчики отдельных компонент время от времени будут приносить новые, свеженькие версии своих компонент — работающие быстрее, меньшие по объему, более полные или содержащие как будто меньше ошибок. Замена работающей компоненты новой версией требует той же самой систематической процедуры проверки, что и добавление новой компоненты, хотя это и займет меньше времени.

Каждая бригада, разрабатывающая новый вариант своей компоненты, использует последнюю отлаженную версию объединенной системы как основание для отладки своего куска. Их работа усложнится, если это основание вдруг окажется шатким — начнет изменяться. Конечно, изменения необходимы, но их следует квантовать. Тогда в распоряжении каждого пользователя будут периоды продуктивной стабильности, прерываемые точками — внесением изменений в систему. Но они гораздо менее разрушительны, чем постоянное встряхивание. Биледи и Леман [14]) утверждают очевидное: кванты должны быть очень большими и редкими во времени или, напротив, очень маленькими и частыми. Последняя стратегия в соответствии с их моделью ближе к неустойчивости. Мой опыт подтверждает это: я никогда не рискнул бы применить эту стратегию на практике.

Квантование изменений хорошо согласуется с методом красной проволоки. Поспешная заплата сохраняется до появления следующей по плану версии компоненты, в которой эта ошибка уже исправлена, а соответствующая документация оформлена должным образом.

XIV. ПРИБЛИЖЕНИЕ КАТАСТРОФЫ

«Никто не любит вестника, приносящего плохие новости».
(Софокл)

— Как выходит, что проект опаздывает на год?
— Постепенно.

Когда становится известно, что проект безнадежно отстает от графика, многим кажется, что этой катастрофе предшествовал целый ряд крупных неудач. Чаще всего, однако, в беде повинны термиты, а не цунами, и отставание от графика происходило медленно, но верно. И действительно, с крупными неудачами легче справиться: мобилизуются все силы, осуществляются радикальные перемены, предлагаются новые подходы. Весь коллектив оказывается на высоте.

Но ежедневное отставание от графика труднее распознать, труднее предотвратить труднее исправить. Вчера был болен нужный человек, и встреча не состоялась. Сегодня не работают машины, потому что молния повредила трансформатор и в здании нет электроэнергии. Завтра нельзя будет начать отладку программ на дисках, потому что диски прибудут с завода только через неделю. Снегопад, общественная работа, семейные неприятности, непредвиденная встреча с заказчиками, ревизия — список можно продолжать до бесконечности. Каждый откладывает часть своих дел не более чем на полдня, на день. Но день за днем отставание от графика все растет.

Вехи или помехи?

Как проследить за тем, чтобы большой проект укладывался в строгий график? Прежде всего нужно *иметь* сам график. Для каждого события, называемого вехой, устанавливается дата. Определение дат — это сложная проблема, она решающим образом зависит от опыта.

Существует только одно разумное правило устанавления вех. Вехи должны быть конкретными, специфическими, датируемыми событиями, определенными

строго и четко. Вот контрпримеры: уже по истечении половины всего времени, отведенного на программирование, видимо, можно утверждать, что оно «закончено на 90%»; почти с самого начала отладки кажется, что она «завершена на 99%»; «проектирование завершено» — это событие, о котором можно заявлять практически когда угодно [1]).

Конкретные же вехи — это стопроцентные события. «Спецификации подписаны архитекторами и разработчиками», «исходная программа написана, отперфорирована и записана на библиотечный диск», «отлаживаемая версия прошла все тесты». Эти конкретные вехи позволяют разграничивать неопределенные этапы проектирования, программирования и отладки.

Гораздо важнее, чтобы вехи были четкими и недвусмысленными, нежели удобными для проверки со стороны начальства. Вряд ли человек станет вводить в заблуждение других, *если* перед ним стоят четкие вехи и он сам не обманывается относительно их достижения. Если же вехи определены неясно и расплывчато, то руководитель, принимая желаемое за действительное, часто сам воспринимает совсем не то, что докладывает ему подчиненный. Дополняя Софокла, скажем, что и приносить плохие известия никто не любит; поэтому говорящий невольно смягчает их, не имея никакого желания солгать.

Два интересных исследования по оценке поведения исполнителей в больших проектах показали, что:

1. Оценки длительности некоей работы, сделанные до начала этой работы, тщательно пересматриваемые в процессе подготовки каждые две недели, почти не меняются по мере приближения срока ее начала, независимо от того, насколько неверными они оказываются трех недель перед завершением по графику [2]).

2. Во время работы *переоценка* ее длительности непрерывно уменьшается по мере продвижения к концу.

3. *Недооценка* не изменяется сколько-нибудь значительно во время работы за исключением последних трех недель перед завершением по графику [2]).

Четко расставленные вехи — это большая услуга бригаде, и она вправе ожидать ее от руководителя. Смутные, неясные вехи — тяжкое наказание. Это помехи, они скрывают истинное положение вещей

до тех пор, когда уже ничего нельзя исправить, они подрывают моральный дух коллектива. А хроническое отставание от графика разлагает коллектив окончательно.

«Другие все равно опаздывают» Мы на день отстали от графика, ну и что? Кто станет волноваться из-за одного дня? Мы сделаем это позже. И многое другое мы тоже откладываем на потом.

Бейсболисты знают о существовании дара, не зависящего от физической силы человека. Это — настырность. Ею непременно наделены великие игроки и великие команды. Это способность бегать быстрее, чем нужно, двигаться живее, чем нужно, быть упорнее, чем нужно. Такой дар необходим и большим коллективам программистов. Настырность означает умение найти дополнительные резервы, второе дыхание, позволяющее группе справиться с обычными неприятностями, предвидеть маленькие катастрофы и избежать их. Расчетливый ответ, размеренные усилия — это холодные компрессы, которые охлаждают пыл. Как мы уже видели, *необходимо* беспокоиться, даже если мы отстаем от графика всего на один день. Такие отставания накапливаются, приближая катастрофу.

Но далеко не каждое отставание от графика должно служить поводом для беспокойства. Какой-то расчет в действиях руководителя должен быть, пусть даже это охлаждает пыл работника. Но как определить, чем чревата та или иная задержка? В этом деле незаменима схема PERT или метод критического пути. Он показывает, кому чего не хватает, и кто оказался на критическом пути, где любые задержки отодвигают сроки окончания всей работы. Кроме того, сетевой график показывает, до каких пор могут нарастать задержки, не приводя нас на критический путь.

Метод PERT, строго говоря, является усложнением метода критического пути, где устанавливаются три отрезка времени для каждого события, соответствующие различным вероятностям выполнения их в установленные сроки. Я не считаю, что такое усовершенствование стоит дополнительных усилий, но для краткости я буду называть любой сетевой график с критическим путем схемой PERT.

Подготовка схемы PERT — наиболее трудоемкий этап ее использования. Составление сетевого графика,

установление зависимостей и определение этапов требуют очень больших затрат на планирование на самых ранних этапах проекта. Первая схема всегда ужасна, и немало изобретательности придется приложить, прежде чем получится вторая.

В ходе выполнения проекта схема PERT разоблачает деморализующую отговорку: «Другие все равно опаздывают». Она показывает, как избежать критического пути и что делать, если задержка все-таки произошла.

Сор в избе

Когда младший руководитель замечает, что его маленькая группа отстала от графика, он далек от мысли сейчас же бежать к начальству с этим плохим известием. Может быть, группа сумеет поправить дела. Или он сам придумает что-нибудь, чтоб справиться с проблемой. Так зачем же беспокоить начальника? Младший руководитель для того и существует, чтобы справиться с такими проблемами. А у старшего руководителя без того достаточно забот, требующих его внимания и участия, так что сам он не ищет новых хлопот. И сор оставляют в избе, заметают под коврик.

Но каждый старший руководитель нуждается в информации двух видов: о происшествиях, требующих его вмешательства, и о положении дел, сообщаемом для сведения[3]). Для этой цели ему нужно знать о положении во всех группах. Однако получить правдивую картину очень трудно.

Именно здесь сталкиваются интересы старшего и младшего руководителей: младший опасается, что если он доложит о своих проблемах, то «начальство примет меры». А в случае его вмешательства младший руководитель не волен поступать по-своему, его власть уменьшается, нарушаются другие планы. Поэтому пока младший руководитель считает, что он может справиться сам, он не посвящает старшего в свои затруднения.

В распоряжении старшего руководителя есть два метода, позволяющие обнаружить «сор в избе», и он должен использовать оба. Первый заключается в том,

чтобы свести к минимуму конфликт между ролями и содействовать одинаковому взгляду на положение вещей. Второй метод рекомендует «заглядывать в углы».

Сглаживание противоречий между ролями. Чтобы свести к минимуму конфликт между ролями, старший руководитель прежде всего должен различать информацию, требующую вмешательства, и информацию о положении дел. Он должен приучить себя никогда не вмешиваться в дела, с которыми могут справиться сами подчиненные, и никогда не принимать никаких мер, в то время, когда он просто знакомится с положением дел. Я знавал одного руководителя, который неизменно поднимал трубку и начинал отдавать распоряжения, не дочитав до конца даже первого абзаца доклада, информировавшего о положении дел. Очевидно, что такая реакция руководителя лишает его возможности получить полную информацию.

И, наоборот, когда младший руководитель знает, что его начальник воспримет доклад без паники или без попыток вмешательства, он склонен честно обрисовывать положение дел.

Очень полезно все собрания, совещания, конференции разделять на чисто информационные и те, на которых принимаются решения, и строго придерживаться этого разделения. Конечно, начальник может принимать решения и сразу после информационного совещания, если он считает, что дело не терпит отлагательств. Но каждый, по крайней мере, должен знать, чем это вызвано, а старший руководитель обязан дважды подумать, прежде чем сделать это.

Как заглянуть в углы. Тем не менее начальник должен иметь в своем распоряжении методы, позволяющие узнать истинное положение дел, неважно с помощью ли руководителей низшего ранга или же самостоятельно. Схема PERT и ее частые, строго расставленные вехи являются основным источником такой информации.

Отчет, показывающий как вехи, так и реальное положение дел, является ключевым документом. В табл. 14.1 представлен отрывок из такого отчета. Этот отчет указывает на некоторые затруднения. Спецификации на несколько компонент не утверждены в срок. Внешняя документация (руководства) тоже не

утверждена, и автономные испытания (альфа-тест) продукта не закончены.

Такой отчет служит темой собрания, назначенного на 1 февраля. Вопросы для обсуждения известны всем, и руководитель группы, отвечающий за данную подсистему, должен быть готов объяснить причину опоздания, сказать, когда все будет закончено, какие меры приняты, и, если нужна помощь со стороны руководства проекта или параллельных групп, указать, какая именно.

В. Выссотски из фирмы Bell Telephone Laboratories принадлежит следующее замечание: «Я обнаружил, что в качестве вех выполнения проекта полезно указывать как «назначенные», так и «ожидаемые» даты. Назначенные даты принадлежат руководителю проекта и представляют собой план последовательного выполнения проекта как единого целого, который априори рассматривается как вполне обоснованный и выполнимый. Ожидаемые даты принадлежат младшему руководителю, в чьей компетенции находится данная часть проекта, и представляют собой его точку зрения на то, когда работа будет завершена при наличии необходимых ресурсов. Ожидаемые даты не касаются руководителя проекта, он должен в основном заботиться о получении вовсе не приятных, оптимистических, пусть даже самозащитных, заниженных, а точных и беспристрастных оценок. Как только это будет осознано, руководитель проекта сможет ясно увидеть те направления, где он встретится с затруднениями, если не предпримет чего-нибудь заранее»[4]).

Подготовка схемы PERT входит в функции начальника и руководителей, отчитывающихся перед ним. Ее просмотр и проверка, подготовка отчетов составляют круг обязанностей небольшой (1—3 человека) штатной группы, работающей непосредственно под эгидой старшего руководителя. Значение такой *группы планирования и контроля* трудно переоценить. В ее полномочия входит право выяснить у всех руководителей подразделений, когда они будут устанавливать или изменять вехи, и были ли нужные вехи достигнуты. Так как группа планирования и контроля берет на себя всю бумажную работу, то деятельность руководителей замыкается на главном — принятии решений.

Резюме отчета о состоянии разработки (языковые процессоры

Проект	Место нахождения	Обязательства (объявление, поставка)	Программы (закончены, утверждены)	Характеристики (сформулированы, утверждены)
Уровень E (12К)				
Ассемблер	Сан-Хосе	04.64У 31.12.65	28.10.64У	13.10.64У 11.01.65
Фортран	Паукипси	04.64У 31.12.65	28.10.64У	11.10.64У 22.01.65
Кобол	Эндикотт	04.64У 31.12.65	28.10.64У	15.10.64У 20.01.653
RPG	Сан-Хосе	04.64У 31.12.65	28.10.64У	30.09.64У 05.01.653
Обслуживающие программы	Манхэттен	04.64У 31.12.65	24.06.64У	
Сортировка 1	Паукипси	04.64У 31.12.65	28.10.64У	19.10.64У 30.11.643
Сортировка 2	Паукипси	04.64У 30.06.66	28.10.64У	19.10.64У 11.01.65
Уровень F (44К)				
Ассемблер	Сан-Хосе	04.64У 31.12.65	28.10.64У	13.10.64У 11.01.65
Кобол	Манхэттэн	04.64У 30.06.66	28.10.64У	15.10.64У 04.01.65
NPL (PL/1)	Хэрсли	04.64У 31.03.66	28.10.64У	
2250	Кингстон	30.03.64У 31.03.66	05.11.64У	12.64У 04.01.65
2280	Кингстон	30.06.64У 30.09.66	05.11.64У	
Уровень H (200К)				
Ассемблер	Манхэттен		28.10.64У	
Фортран	Паукипси	04.64У 30.06.66	28.10.64У	16.10.64У 11.01.65
NPL	Хэрсли	04.64У 31.03.67	28.10.64У	
NPL II	Паукипси	04.64.У	30.03.64У	

Примечание: У — утверждено, З — завершено, НУ — неус-

Таблица 14.1

операционной системы OS/360 на 1 февраля 1965 г. и обслуживающие программы)

Руководства (написаны, утверждены)	Автономные испытания (начало, конец	Субсистемные испытания (начало, конец)	Системные испытания (начало, конец)	Публикация (подготовлено, утверждено	Сдача (бета-тест) (начало, конец)
13.11.64У	15.01.65У				01.09.65
18.11.64	22.02.65				30.11.65
17.11.64У	15.01.65У				01.09.65
19.12.643	22.02.65				30.11.65
17.11.64У	15.01.65У				01.09.65
08.12.643	22.02.65				30.11.65
02.12.64У	15.01.65У				01.09.65
18.01.653	22.02.65				30.11.65
20.11.64У	15.01.65У				01.09.65
30.11.643	22.03.65				30.11.65
12.11.64У	15.01.65У				01.09.65
30.11.643	22.03.65				30.11.65
12.11.64У	15.01.65У				01.03.66
30.11.643	22.03.65				30.05.66
13.11.64У	15.02.65				01.03.65
18.11.643	22.03.65				30.11.65
17.11.64У	15.02.65				01.03.66
08.12.643	22.03.65				30.05.66
12.01.65У	04.01.65У				03.01.66
29.01.65	29.01.65				НУ
	01.04.65				28.01.66
	30.04.65				НУ
11.11.64У	15.02.64				01.03.66
10.12.64	22.03.65				30.05.66
	07.65				01.67
01.02.65					15.10.65
01.04.65					15.12.65

тановленная дата.

У нас была группа планирования и контроля, проявившая большое мастерство, энтузиазм и дипломатическое искусство. Ею руководил А. М. Пьетрасанта, который проявил недюжинные способности в разработке эффективных, но ненавязчивых методов контроля. В результате отношение к его группе было в высшей степени уважительным и терпимым. Для группы, имеющей такие раздражающие обязанности, это настоящее достижение.

Скромные затраты на обеспечение функции планирования и контроля себя вполне оправдывают. Такая группа делает для выполнения проекта гораздо больше, чем если бы все эти люди были непосредственно заняты написанием программ, потому что группа планирования и контроля всегда на страже, она делает видимыми самые незначительные задержки, выявляет критические обстоятельства. Это система раннего предупреждения, предотвращающая постепенную потерю года.

XV. ВТОРОЕ ЛИЦО

«Мы не обладаем тем, чего не понимаем».

(Гете)

«О, дайте мне талант комментатора, который, не углубляясь в суть явлений, будоражит умы людей».

(Граббе)

Программа — это сообщение, передаваемое человеком машине. Строго упорядоченный синтаксис, тщательные определения существуют для того, чтобы сделать наши намерения понятными бессловесной машине.

Но написанная программа имеет другое лицо, обращенное к человеку-пользователю, и оно должно уметь говорить. Даже самые личные программы должны обладать такой способностью, потому что память может подвести автора-пользователя, и ему может понадобиться восстановить детали того, что им написано.

Как же необходима документация для производственной программы, пользователь которой удален от автора и во времени, и в пространстве! Второе лицо программного продукта, обращенное к пользователю, так же важно, как и то, что обращено к машине.

Почти всем нам не раз приходилось втихомолку проклинать далекого и анонимного автора небрежно и скудно документированной программы. И почти все мы пытались поэтому воспитать в молодых программистах соответствующее отношение к документации, которое сохранялось бы всю жизнь, невзирая на леность и нехватку времени. Вообще говоря, нам это не удалось. Я думаю, что мы пользовались неверными методами.

Томас Дж. Уотсон-старший *) рассказывал как-то о своем первом опыте в качестве коммивояжера, продающего кассовые аппараты. Полный энтузиазма, он отправился в путь, погрузив кассовые аппараты в фургон. Он прилежно изъездил всю местность, но так и не продал ни одного аппарата. Совершенно подавленный неудачей, он доложил о ней хозяину. Тот выслушал отчет, а затем сказал: «Помогите мне погрузить

*) Основатель корпорации IBM. (*Прим. ред.*)

аппараты в фургон, запрягите лошадь, и снова в путь». Так они и сделали, объехали одного покупателя за другим, и старший *показывал, как* продавать кассовые аппараты. Дальнейшее показало, что урок не пропал зря.

В течение нескольких лет на лекциях по технологии программирования я прилежно убеждал своих студентов в необходимости хорошей документации и делал это еще более пылко и красноречиво, чем начинающий коммивояжер. Но это не сработало. Я считал, что они научились подготавливать соответствующую документацию, но не делают этого из-за отсутствия энтузиазма. И тогда я решил «погрузить пресловутые кассовые аппараты в фургон» — т. е. *показать*, как нужно делать работу. Такой подход оказался гораздо успешнее. Поэтому в дальнейшем изложении я откажусь от призывов и сосредоточусь на вопросе о том, как подготовить хорошую документацию.

Какая документация нужна?

Различные уровни документации требуются для случайного пользователя программы, для пользователя, который должен постоянно полагаться на программу, и для пользователя, который должен приспосабливать программу к изменению обстоятельств или целей.

Для использования программы. Каждому пользователю нужно текстовое описание программы. Почти всегда документация не дает общего представления о программе. Описываются деревья, комментируются ветви и листья, но нет карты леса. Чтобы подготовить хороший текст, начинайте с самого начала и медленно двигайтесь вперед.

1. *Назначение.* Какова основная функция программы, для чего она?

2. *Ситуация.* На каких машинах, в какой конфигурации и на какой операционной системе она будет работать?

3. *Область и сфера действия.* Какова область входных данных? В каком диапазоне могут появиться выходные результаты?

4. *Реализуемые функции и используемые алгоритмы.* Что именно она делает?

5. *Форматы ввода/вывода.* Точные и полные.

6. *Рабочие процедуры,* включая нормальное и аварийное окончание, описывают все, что видно с пульта и будет получено на выдачах.

7. *Варианты.* Какие функции может выбирать пользователь? Как этот выбор определяется?

8. *Время исполнения.* Сколько времени требует задача указанного объема при определенной конфигурации оборудования?

9. *Точность и проверка.* Какова ожидаемая точность ответов? Каковы способы проверки точности?

Зачастую такую информацию можно изложить на трех-четырех страницах. Необходимо уделять самое пристальное внимание краткости и точности изложения. Большую часть этих документов следует подготовить еще до того, как будет написана программа, потому что они воплощают основные проектные решения.

Для доверия к программе. Кроме описания того, как использовать программу, следует сообщить некоторую информацию о том, как она работает. А это означает, что необходимы тесты.

Каждый экземпляр готовой программы должен включать небольшие тесты, которые пользователь может стандартно использовать для проверки того, что он имеет верный экземпляр, загруженный в машину.

Далее, нужны тесты, которые обычно пропускаются после того, как в программу внесены изменения. Они распределяются на три класса в соответствии с областью входных данных.

1. Главные тесты, которые проверяют основные функции программы для наиболее типичных данных.

2. Предельно допустимые тесты, которые устанавливают границы области входных данных, проверяя, работает ли программа с максимально допустимыми величинами, с минимально допустимыми величинами или с какими-либо исключениями.

3. Тесты, устанавливающие границу области входных данных извне, проверяющие, выдаются ли в случае ввода недопустимых данных соответствующие диагностические сообщения.

Для модификации программы. Для того, чтобы приспособить программу к своим нуждам, чтобы вносить в нее изменения, необходима исчерпывающая

информация. Конечно, все подробности содержатся в распечатке, снабженной хорошими комментариями. Но человеку, собирающемуся вносить в программу изменения, как и более изощренному пользователю, настоятельно необходим четкий и полный обзор ее внутренней структуры. Что должно войти в такой обзор?

1. Блок-схема или граф подчиненности подпрограмм. Последнее предпочтительней. (Подробнее обсуждается ниже.)

2. Полные описания используемых алгоритмов или ссылки на соответствующие описания в литературе.

3. Форматы всех используемых файлов.

4. Схема передачи информации — последовательность считывания данных или программ с ленты или диска — и описание того, что происходит на каждом этапе передачи.

5. Обсуждение модификаций, допускаемых первоначальным проектом, сущность и расположение мест подключения и выходов из программ, свободное обсуждение идеи первого автора о возможных модификациях и о путях их осуществления. Полезны также его замечания относительно скрытых «ловушек».

Несостоятельность блок-схем

Блок-схемы — это та часть документации к программе, которая почти всегда имеется в избытке. Между тем многие программы вообще не нуждаются в блок-схемах и лишь очень немногие из них требуют больше одного листа таковых.

Блок-схемы показывают структуру ветвления программы только в одном ее аспекте. Но даже эта структура видна достаточно четко, только если вся блок-схема помещается на одной странице, и о ней очень трудно получить хорошее представление, если блок-схема располагается на нескольких листах, связанных вместе нумерованными стрелками.

Блок-схема, помещающаяся на одной странице, для большой программы по существу превращается в общий план программы, перечень ее основных этапов или блоков, и, как таковая, она очень удобна. На рис. 15.1 показан такой граф подчиненности подпрограмм.

Конечно, такой граф и не следует стандартам блок-схем, и не нуждается в них. Все эти правила относительно вида элементов, стрелок, порядка нумерации и т. д. нужны только для того, чтобы можно было понять подробные блок-схемы.

Подробные блок-схемы, однако, устарели; они только мешают, и в лучшем случае пригодны для обучения

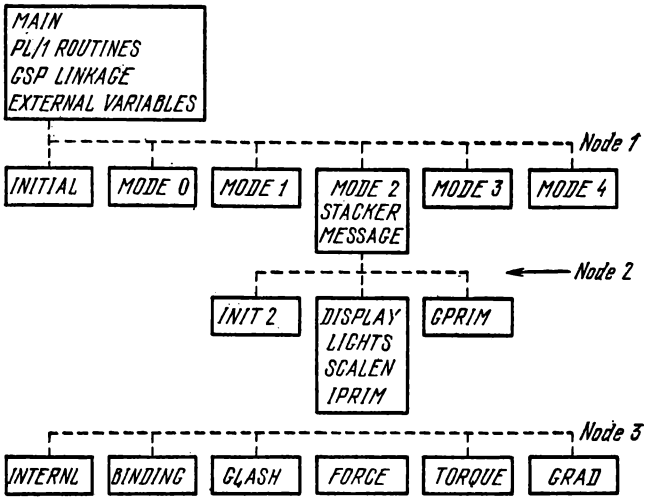
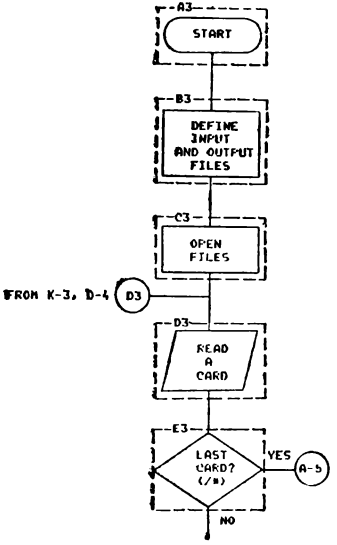


Рис. 15.1. Пример структуры программы.

новичков, еще не умеющих алгоритмически мыслить. В свое время предложенные Голдстайном и Нейманом [1]) маленькие квадратики на блок-схемах вместе со своим содержанием выступали в качестве языков высокого уровня, объединяя абсолютно непонятные операторы машинного языка в группы, имеющие определенный смысл. Как давно уже указал Айверсон [2]), в систематическом языке высокого уровня такая группировка уже осуществлена, так что каждый квадратик просто соответствует оператору (рис. 15.2). Тогда сами квадратики превращаются в случайное и ненужное упражнение по рисованию, и от них можно отказаться. Но теперь не остается ничего, кроме стрелок. Стрелки, соединяющие оператор со следующим за ним, не нужны, сотрем их. Остаются только операторы




```
PGM4: PROCEDURE OPTIONS (MAIN);

          DECLARE SALEFL FILE
          RECORD
          INPUT
          ENVIRONMENT (F(80) MEDIUM (SYSIPT, 2501));
DECLARE PRINT4 FILE
          RECORD
          OUTPUT
          ENVIRONMENT (F(132) MEDIUM (SYSLST, 1403) CTLASA);
DECLARE 01 SALESCARD,
          03 BLANK1          CHARACTER (9),
          03 SALESNUM        PICTURE '9999',
          03 NAME            CHARACTER (25),
          03 BLANK2          CHARACTER (7),
          03 CURRENT_SALES   PICTURE '9999V99'
          03 BLANK3          CHARACTER (29);
DECLARE 01 SALESLIST,
          03 CONTROL         CHARACTER (1) INITIAL (' '),
          03 SALESNUM_OUT    PICTURE 'ZZZ9',
          03 FILLER1         CHARACTER (5) INITIAL (' '),
          03 NAME_OUT        CHARACTER (25),
          03 FILLER2         CHARACTER (5) INITIAL (' '),
          03 CURRENT_OUT     PICTURE 'Z,ZZZV.99',
          03 FILLER3         CHARACTER (5) INITIAL (' '),
          03 PERCENT         PICTURE 'Z9',
          03 SIGN            CHARACTER (1) INITIAL ('%'),
          03 FILLER4         CHARACTER (5) INITIAL (' '),
          03 COMMISSION      PICTURE 'Z,ZZZV.99',
          03 FILLER5         CHARACTER (63) INITIAL (' ');


OPEN FILE (SALEFL),FILE (PRINT4);

          ON ENDFILE (SALEFL) GO TO ENDOFJOB;
```

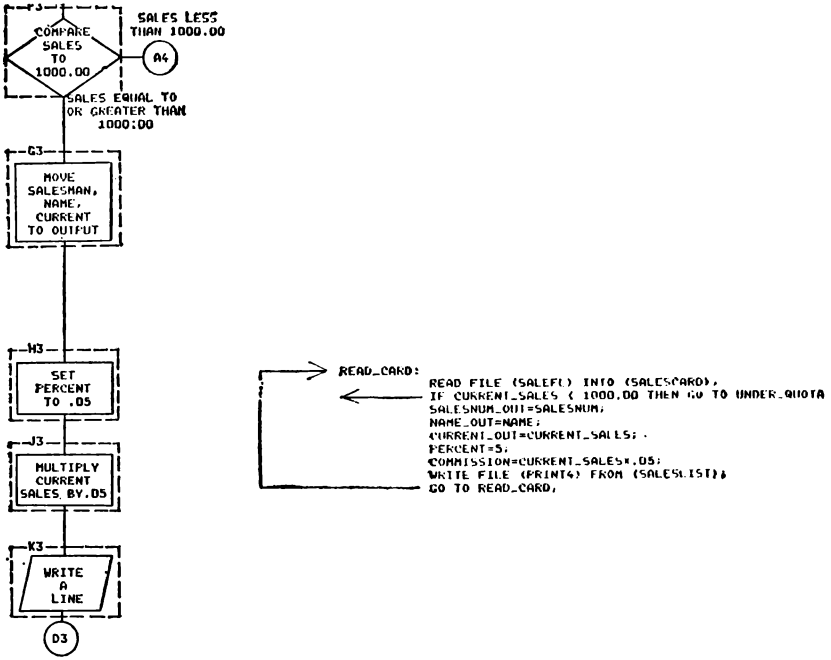


Рис. 15.2. Сравнение блок-схемы и соответствующей программы на PL/1.

перехода. Но если следовать хорошей практике и использовать блочные структуры для минимизации числа операторов перехода, то останется совсем немного стрелок, вот они-то очень сильно облегчают понимание. Эти стрелки можно перенести прямо на распечатку программы и совсем избавиться от блок-схемы.

В действительности блок-схемы гораздо больше превозносятся, чем используются на практике. Я никогда не видел, чтобы опытный программист чертил блок-схемы, прежде чем написать программу. Когда стандарты организации требуют блок-схем, то почти неизменно они рисуются после. Многие программистские организации с гордостью пользуются специальными программами для построения «этого незаменимого инструмента программиста» по готовой машинной программе. Я не считаю этот универсальный опыт прискорбным проявлением дурного тона, признание в котором сопровождается нервным смехом. Напротив, это свидетельство здравого смысла, урок, проливающий свет на истинную пользу блок-схем.

Апостол Петр так говорил о новообращенных язычниках и иудейских законах: «Что же вы желаете возложить на выи (их) иго, которого не могли понести ни отцы наши, ни мы?» (Деяние 15, 10). Я хотел бы сказать то же самое о начинающих программистах и устаревшей практике использования блок-схем.

Самодокументированные программы

Основной принцип обработки данных учит нас, что безрассудно даже пытаться вести синхронную обработку независимых файлов. Гораздо лучше объединить их в один файл, где каждая запись содержит всю информацию из обоих файлов, относящуюся к данному ключу.

Однако наша практика ведения документации по программам не следует нашим же собственным теориям. Обычно нам приходится иметь программу в форме, удобочитаемой для машины, и одновременно независимую документацию, состоящую из текстовых описаний и блок-схем, удобочитаемых для человека.

Результаты подтверждают нашу теорию о непригодности раздельных файлов. Документация крайне плоха, а методы ее ведения и того хуже. Изменения, внесенные в программу, не находят быстрого, точного и устойчивого отображения на бумаге.

Решение проблемы, я считаю, в том, чтобы слить файлы, чтобы ввести документацию в исходную программу. Это одновременно и мощный побудительный стимул к соответствующему ведению документации, и гарантия того, что эта документация всегда будет под рукой у пользователя программы. Такие программы называются *самодокументированными.*

Теперь очевидно, что введение блок-схем в такую программу — задача неприятная, хотя и возможная. Но стоит только признать, что блок-схемы устарели и нужно использовать язык высокого уровня, как появляется возможность объединения программы и документации.

Использование исходной программы в качестве носителя документации влечет за собой некоторые ограничения. С другой стороны, непосредственная, строка за строкой, доступность исходной программы читателю открывает возможности применения новых методов. Пришло время разработать радикально новые подходы и методы документирования программ.

В качестве основной задачи мы должны минимизировать затраты на документацию, тот груз, который ни мы, ни наши предшественники не могли успешно нести.

Подход. Первая идея заключается в том, чтобы те части программы, присутствие которых обусловлено самим языком программирования, играли роль документации. Так что метки, операторы и символические имена должны нести как можно больше смысла.

Вторая идея сводится к использованию пространства и форматов выдачи для повышения читабельности программы и для представления структуры подчиненности и вложенности в программе.

Третья — это введение в программу необходимой документации в качестве примечаний. Большинство программ содержит достаточно построчных примечаний; те из них, которые разрабатывались в организациях, имеющих жесткие стандарты «хорошей документации», часто содержат слишком много примечаний. Но даже в них, тем не менее, обычно не хватает

примечаний, облегчающих их понимание и дающих представление обо всей программе.

Поскольку документация вставляется в структуры, имена и форматы программы, то почти все это *следует* делать, когда программа пишется в первый раз, то есть тогда, когда она *должна* быть написана. Такой подход к документации сводит к минимуму дополнительную работу, к тому же этому почти ничто не препятствует.

Некоторые методы. Самодокументированная программа, написанная на PL/1, приведена на стр. 135—137. Цифры в кружках не относятся к ней; это — метадокументация, используемая при обсуждении примера.

1. Используйте отдельное имя задачи при каждом запуске программы и ведите журнал запусков, где указывайте, что́ было опробовано, когда и с каким результатом. Если имя состоит из мнемонической части (здесь QLT) и числа (здесь 4), то это число можно использовать как номер запуска программы, связывающий вместе распечатки и журнал. При этом методе для каждого запуска нужна новая карта задачи, но их можно делать пакетом, дублируя общую информацию.

2. Введите в мнемоническое имя программы идентификатор версии, т. е. считайте, что будет несколько версий. Здесь индекс — это цифры года 1967.

3. Используйте текстовое описание в качестве комментариев к процедуре.

4. По мере возможности отсылайте к стандартной литературе, содержащей основные алгоритмы: Это экономит место, особенно если указывается более полное описание, чем можно было бы здесь привести, и позволяет осведомленному читателю пропустить то, что он знает.

5. Укажите связь с книжным алгоритмом: а) изменения, б) специализации, в) представления.

6. Опишите все переменные. Используйте мнемонические имена. Используйте комментарии, чтобы превратить DECLARE в полную легенду. Отметьте, что такая легенда уже содержит имена и структурные описания, необходимо только описать, для чего они предназначены. Поступая таким образом, можно избежать повторения имен и структурных описаний.

```
① //QLT4 JOB ...

② QLTSRT7: PROCEDURE (V) ;

   /*********************************************************************/
③  /*A SORT SUBROUTINE FOR 2500 6-BYTE FIELDS, PASSED AS VECTOR V. A    */
   /*SEPARATELY COMPILED, NOT-MAIN PROCEDURE, WHICH MUST USE AUTOMATIC  */
   /*CORE ALLOCATION.                                                   */
   /*                                                                   */
④  /*THE SORT ALGORITHM FOLLOWS BROOKS AND IVERSON, AUTOMATIC DATA      */
   /*PROCESSING,                                                        */
   /*PROGRAM 7.23, P.350. THAT ALGORITHM IS REVISED AS FOLLOWS:         */
⑤  /*  STEPS 2-12 ARE SIMPLIFIED FOR M=2.                             */
   /*  STEP 18 IS EXPANDED TO HANDLE EXPLICIT INDEXING OF THE OUTPUT   */
   /*  VECTOR.                                                        */
   /*  THE WHOLE FIELD IS USED AS THE SORT KEY.                       */
   /*  MINUS INFINITY IS REPRESENTED BY ZEROS.                        */
   /*  PLUS INFINITY IS REPRESENTED BY ONES.                          */
   /*  THE STATEMENT NUMBERS IN PROG 7.23 ARE REFLECTED IN THE        */
   /*    STATEMENT LABELS OF THIS PROGRAM.                            */
   /*  AN IF-THEN-ELSE CONSTRUCTION REQUIRES REPETITION OF A FEW LINES. */
   /*                                                                 */
   /*TO CHANGE THE DIMENSION OF THE VECTOR TO BE SORTED, ALWAYS CHANGE */
   /*THE INITIALIZATION OF T. IF THE SIZE EXCEEDS 4096, CHANGE THE     */
   /*SIZE OF T. TOO.                                                  */
   /*A MORE GENERAL VERSION WOULD PARAMETERIZE THE DIMENSION OF V.    */
   /*                                                                 */
   /*THE PASSED INPUT VECTOR IS REPLACED BY THE REORDERED OUTPUT VECTOR */
   /*********************************************************************/
```

```
⑥ /* LEGEND (ZERO-ORIGIN INDEXING)                                    */

   DECLARE
     (H                    /*INDEX FOR INITIALIZING T                  */
      I,                   /*INDEX OF ITEM TO BE REPLACED              */
      J,                   /*INITIAL INDEX OF BRANCHES FROM NODE I     */
      K) BINARY FIXED,     /*INDEX IN OUTPUT VECTOR                    */

     (MINF,                /*MINUS INFINITY                            */
      PINF) BIT (48),      /*PLUS INFINITY                             */

      V (*)  BIT (*),      /*PASSED VECTOR TO BE SORTED AND RETURNED   */

      T (0:8190) BIT (48);/*WORKSPACE CONSISTING OF VECTOR TO BE SORTED,  */
                          /*FILLED OUT WITH INFINITIES, PRECEDED BY LOWER */
                          /*LEVELS FILLED UP WITH MINUS INFINITIES       */
   /* NOW INITIALIZATION TO FILL DUMMY LEVELS, TOP LEVELS, AND UNUSED PART*/
   /* OF TOP LEVEL AS REQUIRED                                          */

⑦ INIT: MINP= (48) '0'B;
        PINT= (48) '1'B;


        DO L=   0 TO 4094; T(L) = MINF;        END;
```

```
     DO L=   0 TO 2499; T(L+4095) = V(L);   END;
     DO L=6595 TO 8190; T(L) = PINF;        END;

⑧ K0:  K = -1;                                                          ⑩
   K1:  I = 0;                  /*          ⑪                          */
   K3:  J = 2*I+1;              /*SET J TO SCAN BRANCHES FROM NODE I    */
   K7: IF T(J) <= T(J+1)        /*PICK SMALLER BRANCH                   */
       THEN                     /*                                      */
       ⑨ DO;          ⑫        /*                                      */
   K11:      T(I) = T(J); /*REPLACE                                     */
   K13:       IF T(I) = PINF THEN GO TO K16; /*IF INFINITY, REPLACEMENT */
                               /* IS FINISHED                          */
   K12:       I = J;           /*SET INDEX FOR HIGHER LEVEL            */
            END;               /*                                      */
          ELSE                 /*                                      */
            DO;                /*                                      */
   K11A:     T(I) = T(J+1);    /*                                      */
   K13A:      IF T(I) = PINF THEN GO TO K16;        /*                 */
   K12A:      I = J+1;         /*                                      */
            END;               /*                                      */
   K14:  IF 2*I < 8191 THEN GO TO K3;  /*GO BACK IF NOT ON TOP LEVEL    */
   K15:  T(I) = PINF;          /*IF TOP LEVEL, FILL WITH INFINITY      */
   K16:  IF T(0) = PINF THEN RETURN;   /*TEST END OF SORT              */
   K17:  IF T(0) = MINF THEN GO TO K1; /*SLUSH OUT INITIAL DUMMIES     */
   K18: K = K+1;                       /*STEP STORAGE INDEX            */
        V(K) = T(0);  GO TO K1; ⑫      /*STORE OUTPUT ITEM             */
   END QLTSRT7;
```

Рис. 15.3. Пример самодокументируемой программы.

7. Выделите инициализацию с помощью метки.

8. Сгруппируйте операторы с помощью меток, чтобы показать их соответствие операторам в описании алгоритма в литературе.

9. Используйте абзацы и отступы для представления структуры и логической группировки.

10. Вручную проведите в распечатке стрелки логических переходов. Они очень полезны при отладке и внесении изменений. Стрелки можно провести справа на полях и сделать их частью текста, доступного машине.

11. Используйте построчные примечания или помечайте все, что не очевидно. Если используются способы, предлагаемые выше, то примечания будут короче и их будет меньше, чем обычно.

12. Располагайте несколько операторов на одной строке или один оператор на нескольких строчках, чтобы подчеркнуть смысловую связь или обеспечить соответствие другому описанию алгоритма.

Почему бы и нет? Каковы недостатки такого подхода к документации? Их несколько, они были вполне реальны, но с течением времени превратились в воображаемые.

Наиболее серьезное возражение заключается в том, что увеличивается размер исходной программы, которую нужно хранить в памяти. Поскольку дело идет к тому, что мы будем хранить исходную программу в памяти, вводя ее непосредственно с терминала, это соображение становится все более важным. Я сам поймал себя на том, что мои комментарии к программе, написанной на APL и хранимой на дисках, всегда короче, чем когда я пишу на PL/1 и собираюсь хранить все примечания на перфокартах.

Но одновременно мы идем к тому, чтобы вводить с терминала текстовые документы и обращаться к ним, вносить в них изменения посредством машинной системы редактирования текстов. Как видно из вышеизложенного, объединение текста и программы *уменьшает* общее число символов, хранимых в памяти машины.

А как насчет блок-схем и графов структуры программы? Если используется только граф самого высокого уровня, то он может храниться как отдельный документ, поскольку он не подвергается частым изме-

нениям. Но можно ввести этот граф в исходную программу в качестве примечания, что представляется весьма разумным.

В какой степени все вышеуказанное применимо к программам на языке ассемблера? Я считаю, что основные идеи самодокументирования вполне приемлемы и здесь. Форматы и расположения программы менее свободны, поэтому их нельзя использовать столь гибко. Однако имена и структурные описания могут использоваться аналогичным образом, а широкое применение комментариев — это хорошая практика в любом языке.

Метод самодокументирования вызван к жизни использованием языков высокого уровня, и потому он демонстрирует наибольшую эффективность и максимально оправдывает себя именно в языках высокого уровня, используемых в системах прямого доступа, как пакетных, так и диалоговых. Как я уже доказывал, такие языки и системы оказывают огромную помощь программисту. Поскольку машины созданы для людей, а не люди для машин, то использование машины имеет как экономический, так и гуманистический смысл.

ЭПИЛОГ

Асфальтовая топь технологии программирования останется непроходимой еще очень долго. Никто не сомневается в том, что человечество будет продолжать попытки ее покорения как вслед за нашими достижениями, так и независимо от них. Системы программного обеспечения представляют собой, может быть, самые запутанные и сложные творения рук человеческих. Руководство этим сложным ремеслом потребует от нас умения наилучшим образом использовать новые языки и системы, наиболее эффективно применять все известные методы технического руководства, а также здравого смысла и умения признавать наши слабости и просчеты.

ПРИМЕЧАНИЯ И ССЫЛКИ

I

1. А. П. Ершов считает, что это обстоятельство определяет не только трудные, но и радостные моменты ремесла программиста. *Ershov A.P.* Aesthetics and teh human factor in programming.— CACM, July 1972, 15, 7, 501—505. (*Ершов А. П.*. Эстетический и человеческий факторы в программировании.— Кибернетика, 1972, № 5.)

II

1. По оценкам В. А. Выссотски из фирмы Bell Telephone Laboratories, прирост рабочей силы в большом программистском проекте допускается на 30% в год. Больший рост числа сотрудников затрудняет и даже препятствует созданию важнейшей неформальной структуры и установлению связей в проекте, что обсуждается в гл. VII.

Ф. Дж. Корбато из Массачусеттского Технологического института утверждает, что в большом проекте следует предвидеть текучесть кадров в пределах 20% в год, и заранее предусмотреть необходимость подготовки новичков и ввода их в формальную структуру проекта.

2. Ч. Портман из фирмы International Computers Limited утверждает: «Когда кажется, что уже все работает, все объединено в систему — вам еще осталось работы на четыре месяца». Некоторые графики распределения работ приводятся в статье *Wolverton R. W.* The cost of developing largescale software: — IEEE Trans. on Computers, June 1974, C—23, 6, p. 615—636.

3. Рисунками 2.5—2,8 мы обязаны Джерри Огдену, который, цитируя мой пример по первой публикации этой главы, значительно улучшил его оформление. *Ogdin J. L.* The Mongolian hordes versus superprogrammer.— Infosystems, December, 1972, p. 20—23.

III

1. *Sackman H., Erikson W. J., Grant E. E.* Exploratory experimental studies comparing online and offline programming performance.— CACM, January, 1968, 11, 1, 3—11.

2. *Mills H.* Chief programmer teams, principles, and procedures.—IBM Federal Systems Division Report FSC 71—5108, Gaithersburg, Md., 1971.

3. *Baker F. T.* Chief programmer team management of production programming.— IBM Syst. J., 1972, 11, 1,

IV

1. *Eschapasse M.* Reims Cathedral, Caisse Nationale des Monuments Historiques.— Paris, 1967.

2. *Brooks F. P.*, Architectural philosophy/W. Buchholz (ed.)— A Computer System. New York: McGraw—Hill, 1962.

3. *Blaauw G. A.* Hardware requirements for the fourth generation. In: Fourth Generation Computers Englewood Cliffs. F. Gruenberger (ed.).— N. J.: Prentice—Hall, 1970.

4. *Brooks F. P., Iverson K. E.* Automatic Data Processing, Syctem/360 Edition — New York: Wiley, 1969, Chap. 5.

5. *Glegg G. L.* The Design of Design.— Cambridge: Cambridge Univ. Press, 1969. Автор утверждает: «На первый взгляд кажется, что регламентация творческого мышления любыми правилами или принципами скорее препятствует, чем помогает этому процессу, но на практике это вовсе не так. Дисциплина мышления мобилизует вдохновение, а не подавляет его».

6. *Conway R. W.* The PL/C Compiler.— Proc. of a Conf. on Definition and Implementation of Universal Programming Languages, Stuttgart, 1970.

7. Необходимость технологии программирования обсуждается в интересной работе *Reynolds C. H.* What's wrong with computer programming management? In: On the Management of Computer Programming/G. F. Weinwurm ed.—Philadelphia: Auerbach, 1971, p. 35-42

V

1. *Strachey C.* Review of Planning a Computer System.— Comp. J., July, 1962, 5, 2, p. 152—153.

2. Это относится только к управляющим программами. Некоторые группы разработчиков трансляторов в проекте OS/360 создавали свои третьи или четвертые системы, и отличное качество их программного продукта является доказательством наших утверждений.

3. *Shell D. L.* The Share 709 system: a cooperative effort; *Greenwald E. D., Kane M.* The Share 709 system: programming and modification; *Boehm E. M., Steel T. B.* The Share 709 system: machine implementation of symbolic programming.— all in JACM April 1959, 6, 2, p. 123—140.

VI

1. *Neustadt R. E.*, Presidential Power.— New York: Wiley, 1960, Chap. 2.

2. *Backus J. W.* The syntax and semantics of the proposed international algebraic language.— Proc. Intl. Conf Inf. Proc. UNESCO, Paris, 1959. Кроме того, целый ряд статей по этому вопросу можно найти в Formal Language Description Languages for Computer Programming/Steel T. B., Jr. (ed.).— Amsterdam: North Holland, 1966.

3. *Lucas P., Walk K.* On the formal description of PL/1.— In: Annual Review in Automatic Programming Language. — New York: Wiley, 1962, Ch. 2, p. 2.

4. *Iverson K. E.* A Programming Language.— New York: Wiley, 1962, Ch. 2.

5. *Falkoff A. D., Iverson K. E., Sussenguth E. H.* A formal description of System/360.— IBM System J. 1964, 3, 3, p. 198—261.
6. *Bell C. G., Newell A.* Computer Structures.— New York: McGraw—Hill, 1970, p. 120—136, 517—541.
7. Ч. Дж. Белл, частное сообщение.

VII

1. *Parnas E. L.* Information distribution aspects of design methodology.— Carnegie—Mellon Univ., Dept. of Computer Science Technical Report, February, 1971.
2. *Heinlein R. A.* The Man Who Sold the Moon.— New York: Signet, 1951, p. 103—104 (Reprinted by permission of the author, copyright 1950 by Robert A. Heinlein).

VIII

1. *Sackman H., Erikson W. J., Grant E. B.* Exploratory experimentation studies comparing online and offline programming performance.— CACM, January 1968, 11, 1, p. 3—11.
2. *Nanus B.* and *Farr L.* Some cost contributors to large—scale programs.— AFIPS Proc. SJCC, Spring 1964, 25, p. 239—248.
3. *Weinwurm G. F.* Research in the maagement of computer programming.— Report SP—2059, System Development Corp., Santa Monica, 1965.
4. *Morin L. H.* Estimation of resources for computer programming projects.— M. S. thesis, Univ. of North Carolina, Chapel Hill, 1974.
5. Ч. Портман, частное сообщение.
6. В неопубликованной работе Э. Ф. Бардена утверждается, что программисты продуктивно используют только 27% своего рабочего времени. (Цитируется по *Mayer D.B.* and *Stalnaker A.W.* Selection and evaluation of computer personnel.— Proc. 23rd ACM Conf., 1968, p. 661.
7. Дж. Арон, частное сообщение.
8. Работа докладывалась на панельном заседании и не включена в Труды АФИПС.
9. *Wolverton R. W.* The cost of developing large—scale sofftware. — IEEE Trans. on Computers, June 1974, C—23, p. 615—636. Эта важная работа рассматривает многие проблемы, упомянутые в настоящей главе, подтверждая ее основные положения.
10. *Корбато Ф. Дж.* Критические вопросы проектирования систем коллективного пользования.— Лекция на открытии Технологического центра по обработке данных фирмы Honeywell, 1968.
11. В. М. Талиафферо также сообщает данные о производительности в 2400 операторов/год на ассемблере, фортране и коболе. *Taliaffero W. M.* Modularity. The key to system growth potential.— Software, July 1971, 1, 3, p. 245—257.
12. В отчете Э. А. Нельсона из фирмы System Development Corp., Report TM—3225, Management Handbook for the Estimation of Computer Programming Costs, приводятся данные об увеличении производительности в три раза при использовании языков высокого уровня, хотя разброс довольно широк.

IX

1. *Brooks F. P.* and *Iverson K. E.* Automatic Data Processing, System/360 Edition, Chapter 6.— New York: Wiley, 1969.
2. *Knuth D. E.* The Art of Computer Programming, Reading.— Mass.: Addison—Wesley, 1968, V. 1—3.

X

1. *Conway M. E.* How do committees invent? — Datamation, April 1968, 14, 4, p. 28—31.

XI

1. Речь в Оглеторпском университете, 22 мая 1932 г.
2. Поучительный рассказ об использовании опыта Малтикса в создании двух удачных систем см. в *Corbato F. J.*, *Saltzer J. H.*, *Clingen C. T.* Multics — the first seven years.— AFIPS Proc. SJCC, 1972, 40, p. 571—583.
3. *Cosgrove J.* Needed: a new planning framework.— Datamation, December 1971, 17, 23, p. 37—39.
4. Проблема внесения изменений в проект весьма сложна, здесь я ее существенно упрощаю. См. *Saltzer J. H.* Evolutionary design of complex systems. In: Systems: Research and Design/D. Eckman ed.— New York: Wiley, 1961. И пусть все уже сказано и сделано, но я все-таки выступаю за создание пробной системы, которую планируется затем выбросить.
5. *Campbell E.* Report to the AEC Computer Information Meeting.— December, 1970. Этот феномен обсуждается также в *Ogdin J. L.* Designing reliable software.— Datamation, July 1972, 18, 7, p. 71—78. Мои опытные друзья, кажется, расходятся в мнениях относительно того, пойдет ли эта кривая снова вниз.
6. *Lehman M.*, *Belady L.* Programming system dynamics.— Given at the ACM SIGOPS Third Symposium on Operating System Principles, October, 1971.
7. *Lewis C. S.* Mere Christianity.— New York: Macmillan, 1960, p. 54.

XII

1. См. также *Pomeroy J. W.* A guide to programming tools and techniques.— IBM Sys. J. 1972, 11, 3, p. 234—254.
2. *Landy B.*, *Needham R. M.* Software engineering techniques used in the development of the Cambridge Multiple-Access System.— Software, April 1971, 1, 2, p. 167—173.
3. *Corbato F. J.* PL/1 as a tool for system programming.— Datamation, May 1969, 15, 5, p. 68—76.
4. *Hopkins M.* Problems of PL/1 for system programming.— IBM Research Report RC3489, Yorktown Heights, N. Y., August 5, 1971.
5. *Corbato F. J.*, *Saltzer J. H.*, *Clingen C. T.* Multics — the first seven years.— AFIPS Proc. SJCC, 1972, 40, p. 571—582. «Целых полдюжины систем, написанных на PL/1, были затем переписаны в кодах, с тем, чтобы добиться максимальной производительности. Несколько программ было переписано с машинного языка на PL/1, чтобы облегчить их сопровождение».

6. Приведу цитату из статьи Ф. Дж. Корбато (см. ссылку 3 к настоящей главе): «PL/1 уже есть, а альтернативы еще неапробированы». Однако противоположная точка зрения хорошо обоснована в кн. *Henricksen J. O., Merwin R. E.* Programming language efficiency in real—time software systems.— AFIPS Proc. SJCC, 1972, 40, p. 155—161.

7. Не все с этим согласны. Харлан Миллз в частном сообщении заметил: «Мой опыт начинает подсказывать мне, что в промышленном программировании за терминалом будет работать секретарь. Суть в том, чтобы сделать программирование занятием для широкой публики, а не искусством для избранных».

8. *Harr J.* Programming Experience for the Number 1 Electronic Switching System. Статья представлена на весенней объединенной вычислительной конференции АФИПС в 1969 г.

XIII

1. *Высотски В. А.* Здравый смысл при разработке программ, поддающихся проверке.—Лекция на симпозиуме по методам проверки программ.— Чапел Хилл, Северная Каролина, 1972. Эта лекция почти полностью приводится в кн : Program Test Methods/Hetzel W. C. (ed.). — Englewood Cliffs. N. J.: Prentice—Hall, 1972, p. 41—47.

2. *Wirth N.* Program development by stepwise refinement.— CACM. April 1971, 14, 4, p. 221—227. См. также *Mills H.* Top-down programming in large systems, In: Debugging Techniques in Large Systems/Rustin R. ed.— Englewood Cliffs. N. J.: Prentice—Hall, 1971, p. 41—55, а так же *Baker F. T.* System quality through structured programming.— AFIPS Proc. FJCC, 1972, 41—I, p. 339—343.

3. *Dahl O. J., Dijkstra E. W.* and *Hoare C. A. R.* Structured Programming.—London, New York: Academic Press, 1972. Дал У., Дейкстра Э., Хоор К. Структурное программирование. — М.: Мир, 1975. Здесь предмет изложен наиболее полно. См. также письмо Э. В. Дейкстры: *Dijkstra E. W.* GOTO statement consideral harmful.— CACM, March 1968, 11, 3, p. 147—148.

4. *Böhm C., Jacopini A.* Flow diagrams Turing machines, and language with only two formation rules.— CACM, May 1966, 9, 5, p. 366—371.

5. *Codd E. F., Lowry E. S., McDonough E., Scalzi C. A.*, Multiporogramming STRETCH; Feasibility considerations.— CACM, November 1959, 2, 11, p. 13—17.

6. *Strachey C.* Time sharing in large fast computers.— Proc. Int. Conf. on Info. Processing, UNESCO, June, 1959, p. 336—341. См. также замечание Кодда на стр. 000, где он сообщает о работе, аналогичной той, что предложена в статье Стрейчи.

7. *Corbato F. J., Merwin-Daggett M., Daley R. C.*— An experimental timesharing system.— AFIPS Proc. SJCC, 1962, 2, p. 335—344. Перепечатано в кн.: *Rosen S.*, Programming Systems and Languages.— New York: McGraw—Hill, 1967, p. 683—698.

8. *Gold M. M.* A methodology for evaluating time-shared computer system usage.—Ph. D.dissertation, Carnegie—Mellon University, 1967, p. 100.

9. *Gruenberger F.* Program testing and validating.— Datamation, July 1968, 14, 7, p. 39—47.

10. *Ralston A.* Introduction to Programming and Computer Science.— New York: McGraw—Hill, 1971, p. 237—244.

11. *Brooks F. P.* and *Iverson K. E.* Automatic Data Processing, System/360 Edition.— New York; Wiley, 1969, p. 296—299.

12. Проблемы разработки спецификаций, создания и тестирования системы прекрасно изложены в работе *Trapnell F. M.* A systematic approach to the development of system programs.— AFIP Proc. SJCC, 1969, 34, p. 411—418.

13. Система реального времени потребует моделирования обстановки. См., например, *Ginzberg M. G.* Notes on testing real-time system programs.— IBM Sys. J., 1965, 4, 1, p. 58—72.

14. *Lehman M., Belady L.* Programming system dynamics.— Given at the ACM SIGOPS Third Symposium on Operating System Principles, Octover, 1971.

XIV

1. *Reynolds C. H.* What's wrong with computer programming management?: In on the Management of Computer Programming/ Weinwurm G. F. ed.— Philadelphia: Auerbach, 1971, p. 35—42.

2. *King W. R., Wilson T. A.* Subjective time estimates in critical path planning — a preliminary analysis.— Mgt. Sci., January 1967, 13, 5, p. 307—320.

King W. R., Witterrongel D. M., Hezel K. On the analysis of critical path time estimating behavior.— Mgt. Sci., September 1967, 14, 1, p. 79—84.

3. Более полно это обсуждается в книге *Brooks F. P., Iverson K. E.*—Automatic Data Processing, System/360 Edition, New York: Wiley, 1969, p. 428—430.

4. Частное сообщение.

XV

1. *Goldstine H. H., von Neumann J.* Planning and coding problems for an electronic computing instrument, II, 1. Отчет, подготовленный для Отдела Армии США, 1947 г.; перепечатано в кн. *von Neumann J.* Collected Works/A. H. Taub (ed.) — Vol. V, New York: Mc Millan, p. 80—151.

2. Частное сообщение, 1957 г. Это соображение опубликовано в кн.: *Iverson K. E.* The Use of APL in Teaching. — Yorktown, N. Y.: New IBM Corp., 1969.

3. Другой перечень методик для языка PL/1 приводится в *Walter A. B., Bohl M.*— From better to best-tips for good programming.— Software Age, November 1969, 3, 11, p. 46—50. Те же самые методы можно использовать и для алгола и даже для фортрана. Д. Э. Лэнг из Колорадского университета написал программу установления форматов для фортрана,названную STYLE, которая позволяет получить такой же результат. См. также *McCracken D. D., Weinberg G. M.* How to write a readable FORTRAN program. — Datamation, October 1972, 18, 10, p. 73—77.

УКАЗАТЕЛЬ

Администратор 30
Айверсон К. Е. (Iverson K. E.) 54
Алгол 32
Алгол-68 37, 53
Аристократия 38
Арон Дж. (Aron J.) 72
Архив хронологический 31
Архивариус 31
Архитектор 34
Архитектура 38
Ассемблер 100
АСУ глобальное 87
APL (A Programming Language) 54

База данных 86
Барьер социологический 91
Бах Дж. С. (Bach J. S.) 40
Бейкер Ф. Т. (Baker F. T.) 142
Белл Ч. Дж. (Bell C. G.) 54
Библиотека макрокоманд 32
— программ 100
Биледи Л. (Beladu L.) 95
Блау Дж. (Blaauw G. A.) 43
Блок-схема 128
Боем С. (Böhm C.) 108
Брукс Ф. П. мл. (Brooks F. P., Jr.) 82
Бэкус Дж. (Backus J. W.) 54
Бэкуса — Наура форма 54
Бюджет 85
— расхода памяти 78
Bell Telephone Laboratories 8
Boehm E. M. 142
Bohl M. 145

Вайнвурм Г. Ф. (Weinwurm G. F.) 143
Вариант 79
Веха 24, 116
Взаимодействие как этап творчества 18
Вирт Н. (Wirth N.) 107
Волшебство 14
Выдача выборочная 110
— памяти 103
Выссотски В. А. 8
Вычислительная машина 38
— — IBM 650 37
— — — 701 99
— — — 704 47
— — — 709 49
— — — 1410 49
— — — 7090 47
— — Системы 360 38
— — — 360/30 39
— — — — 360/65 77
— — — — 75 40
— — — 370/165 76
Walk K. 142
Walter A. B. 146
Weinberg G. M. 146
Wilson T. A. 146
Witterrongel D. M. 146
Wolverton R. W. 141

Генератор тестов 97
Главный программист 29
Голдстайн Х. (Goldstine H. H.) 129
Гоулд М. М. (Gold M. M.) 111
Грант Е. Е. (Grant E. E.) 27
График работ 66
Груенбергер Ф. (Gruenberger F.) 112
Группа планирования и контроля 124
— небольшая, боевая 27

Ginzberg M. G. 146
Glegg G. L. 142
Greenwald E. D. 142

Даты назначенные 121
— ожидаемые 121
Дейкстра Е. (Dijkstra E. W.) 108
Демократия 38
Деятельность творческая 18, 42
Джакопини А. (Jacopini A.) 108
Дпректор технический 67
Дисковая операционная система DOS 1410-7010 49
Дистанционный ввод задач 49
Документ 83
Документация 29
Dahl O. J. 145
Daley R. C. 145
DEC PDP-8 54
DECLARE 134
Digitek 81

Единство концептуальное 35
Ершов А. П. 8
Eckman D. 144
Eshapasse M. 142

Журнал отладки 111
— рабочий 31
— регистрации телефонных звонков 58

Задачи 14, 61
— стоимости и производительности 43
Закон Брукса 26
— Конвея 86

Ивенс Б. О. (Evans B. O.) 7
Идея как этап творчества 18
Изменения 89
— в организации 90
Имитатор 99
Имя задачи 134
Инструмент 96
Инструментальщик 31
Инструментарий 97
Интерпретатор 81
Интерфейс 13, 29
IF ... THEN ... ELSE 108
International Computers Limited 8

Канал 39
Капп Э. (Capp A.) 67
Катастрофы с графиком 23
Квантование изменений 115
Кейз Р. П. (Case R. P.) 8
Кемпбелл Б. (Campbell B.) 93
Кнут Д. (Knuth D. E.) 82
Кодд Е. Ф. (Codd E. F.) 110
Команды целевые 66
Комитет 66
Компонента фиктивная 113
Конвей М. Е. (Conway M. E.) 86
Контроль за изменениями 113
— использования памяти 97
Конференция 56
Концептуальное единство 35
Корбато Ф. Дж. (Corbato F. J.) 8, 145
Косгроув Дж.(Cosgrove J.) 89, 91
Кроули У. Р. (Crowley W. R.) 100
Kane M. 142
King W. R. 146
Clingen C. T. 144

Леман М. (Lehman M.) 95
Локен О. С. (Locken O. S.) 63
Льюс С. (Lewis C. S.) 95
Landy B. 144
Lang D. E. 146
Lowry E. S. 145
Lucas P. 142

Массачусетский технологический институт 8
Мауэрс Ч. Н. (Mooers C. N.) 37

Машина целевая 97
Метод красной проволоки 114
— критического пути 118
Микрофиша 64
Миллз Х. (Mills H.) 29
Мини-решения 52
Мини-файл 113

Мнемоническое имя 134
Моделирование обстановки 146
— производительности 78
Модель производительности 102
Модуль 80
Моурин Л. (Morin L. H.) 70
Мур С. Э. (Moore S. E.) 9
Mayer D. B. 143
McCracken D. D. 146
McDonough E. 145
Merwin R. E. 145
Merwin-Daggett M. 145

Нанус Б. (Nanus B.) 71
Написание команд 22
Настырность 118
Наур П. (Naur P.) 54
Непосредственность 37
Ньюэлл А. (Newell A.) 54
Needham R. M. 144
Nelson E. A. 143

Овидий (Ovid) 47
Оператор CASE 108
— GO TO 108
Операции периода компиляции 56
Операционная система 97
— — Share 49
— — stretch 78
— — OS/360 37, 38
Оптимизм 17
Организация древовидная 66
— матричного типа 66
Оснастка 32
Отладка автономная 109
— диалоговая 110
— машинная 109
— регрессивная 94
— системная 112
Ошибка 106
— известная 113
Ogdin J. L. 141

Падегз (Padegs A.) 52
Парнас Д. (Parnas D. L.) 66
Паскаль Б. (Pascal B.) 95
Перечень изменений 64
Пилот второй 30
Планирование 100
Планировщик 49
Площадка для игр 101
Подбиблиотека объединенной системы 101
Показатель положения дел 83
— производительности 29
Портман Ч. (Portman C.) 8
Права и ответственность 14
— программиста 14
Представление данных 81
Примечания в программе 133
Принцип действия Системы/360 52
— подчинения 33
Прирост рабочей силы 141
Проверка продукта 59
— спецификаций 107
Программа 11
— самодокументированная 133
— управляющая 73
Программирование диалоговое 104
— структурированное 108
Программный продукт 11
— — комплексный 12
Продвижение по службе 92
Продюсер 66
Проектирование 21
— нисходящее 107
Производительность программиста 27
Простота 37
Процедура 134
— каталогизированная 32
Прямое внесение изменений 55
Пьетрасанта А. М. (Piestrasanta A. M.) 8
Pomeroy J. W. 144

Радости ремесла 13
Разбирательство 56
Разделение труда 66
Различия интересов 33
— суждений 33
Размер программы 76
Разработчик 41
Распределение памяти динамическое 48
—рабочих мест 85
Реализация 18
— как этап творчества 43
— совместная 58
Редактор 30
— связей 48

Рузвельт Ф. (Roosevelt F. D.) 88
Руководство 52
— Системы 360 52
Ralston A. 146
Reynolds C. H. 146
Rosen S. 145
Rustin R. 145

Сакмен Х. (Sackmen H.) 27
Свифт Дж. (Swift, J.) 88
Связь 20
Сглаживание противоречий 120
Сейерс Д. (Sayers D.) 18
Секретарь 30
Семантика 38
Сеть ARPA 65
Синтаксис 38
— абстрактный 54
Система документации 102
— программирования 36
— разделения времени PDP-10 37
— — — OS/360 102
— «чистая» 55
Скорость отладки 15
Слоан Дж. С. (Sloane J. C.) 9
Служба данных 99
— отладки 97
Смит С. (Smith S.) 76
Совещания информационные 120
Совместимость 52
Составление нового графика 24
Софокл (Sophocles) 116
Специализация функций 65
Спецификации архитекторные 36
— внешние 62
— письменные 51
— сопряжений 62
— функциональные 29
Средства 43
Средство достоверное 99
— отладки Testran 48
Сроки 43
Стандарт 62
Статистическая сегментация программ 48
Стрейчи С. (Strachey C.) 47
Стрелка логического перехода 138
Супервизор 97
Схема организации 84
— передачи информации 128
— PERT 71
Saltzer J. H. 144
Scalzi C. A. 144
Shell D. L. 142
Stalnaker A. W. 142
Steel T. B., Jr. 141
Sussengath E. H. 141
System Development Corporation 71

Творческая возможность 41
— деятельность 18, 93
— работа 40
Терминал графический 65
Тесты 15, 32
Точность совершенная 14
Транслятор 100
— PL/C 41
Трапнелл Ф. М. (Trapnell F.M.) 7, 8
Трассировка 110
Трумен Г. С. (Truman H. S.) 51
Тюки Дж. (Tukey J. W.) 81
Taliaffero W. M. 143

Университет Корнелльский 41
— Кэмбриджский 101
Уотсон Т. Дж. мл. (Watson T. J., Jr.) 7
Уотсон Т. Дж. ст. (Watson T. J., Sr.) 125
Устаревание 26
Учет 100

Фагг П. (Fagg P.) 24
Файл фиктивный 113
Фарр Л. (Farr L.) 71
Фирма IBM 8
Форма Бэкуса—Наура 54
Формализм письменных предложений 57
Формальное разделение 101
Формальные документы 86
— описания 53
Форматы ввода/вывода 127
Фортран 39, 78
Фортран Н. (Fortran H.) 78
Франклин Дж. (Franklin J. W.) 102

Фурье-преобразование быстрое 81
Falkoff A. D. 143

Харр Дж. (Harr J.) 8
Хейнлейн Р. (Heinlein R.) 68
Хирургическая бригада 27
Henricksen J. O. 145
Hetzel W. C. 145
Hezel K. D. 146
Hoare C. A. R. 145
Hopkins M. 143

Человеко-месяц 19
Шаги, уточняющие проект 107

Энгельбарт Д. (Engelbart D. C.) 65
Энтропия 95
Эриксон У. Дж. (Erikson W. J.) 27
Эскалация 34
Эффект второй системы 45
— побочный 55

Язык описаний формальный 53
— TRAC 37
— программирования 103
— высокого уровня 103
Языковед 32

Ф. П. Брукс мл.

Как проектируются
и создаются программные комплексы

(Серия: «Библиотечка программиста»)

М., 1979 г., 152 стр. с илл.

Редактор *Н. Н. Васина*

Техн. редактор *Е. В. Морозова*

Корректор *Н. Б. Румянцева*

ИБ № 2060

Сдано в набор 27.03.79. Подписано к печати 07.09.79. Бумага 84×108$^{1}/_{32}$, тип. № 1. Обыкновенная гарнитура. Высокая печать. Условн. печ. л. 7,98. Уч.-изд. л. 7,76. Тираж 20 000 экз. Заказ № 484. Цена книги 60 коп.

Издательство «Наука»
Главная редакция физико-математической литературы
117071, Москва, В-71, Ленинский проспект, 15

4-я типография изд-ва «Наука»,
630077, Новосибирск, 77, Станиславского, 25