Фокс Дж. Программное обеспечение и его разработка

Пер. с англ. —М.: Мир, 1985 —368 с, ил

Автор книги — американский специалист по программированию, один из руководителей фирмы IBM, в своей книге делает попытку изложить общие проблемы создания программного обеспечения, его сопровождения и использования. Особенно подробно рассматриваются все фазы разработки программ разных типов. Изложение ясное, удачно иллюстрировано примерами.

Для программистов разной квалификации и пользователей ЭВМ
ББК 32.973
Ф75
УДК 681.142.2

ПРЕДИСЛОВИЕ РЕДАКТОРА ПЕРЕВОДА

Использование ЭВМ в различных системах управления привело к тому, что в последнее десятилетие стала создаваться индустрия программного обеспечения. Программы, ранее бывшие лишь продуктом творческой деятельности программиста, стали стремительно превращаться в продукцию — плод организованной деятельности многих людей. В последние же годы программы, это специфическое творение человеческого мозга, стали даже товарной продукцией. Они уже продаются не с машиной, не как составные части сложных систем, а сами по себе. Они уже стали “программными средствами”.

Проблематика создания программного обеспечения сложных систем и товарных программ стала весьма актуальной для широкого круга создателей подобных систем и вообще людей, связанных с управлением разработками программ. Естественно, что подавляющее большинство из них не относятся к профессиональным программистам и специфика проектирования, создания и сопровождения больших программных комплексов для них представляется достаточно туманной. Не лучше обстоит дело и с узкоориентированными профессионалами. Зная хорошо свой предмет, они часто теряются, сталкиваясь с проблематикой создания комплекса программ.

В вышедшей несколько лет назад книге “Мифический человеко-месяц” [1]*{Брукс Ф.П. мл. Как проектируются и создаются программные комплексы. Мифический человеко-месяц. Очерки по системному программированию / Пер, с англ. — М: Наука, 1979}* Брукс отчасти пытался заполнить пробел в этой области. Книга Джозефа М. Фокса, в прошлом руководителя отдела программирования фирмы IBM, перевод которой вы, читатель, держите в руках, похожа по тематике, но гораздо более основательна. В ней обсуждаются весьма общие проблемы создания комплексов программ; она расчитана на самые широкие круги читателей.

На наш взгляд, работа Д.М.Фокса оказалась очень удачной. Для нее характерны широкий взгляд на предмет и обилие конкретного материала. Следует отметить очень хорошее образное изложение, великолепно подобранные цитаты (иногда ставившие в тупик и переводчика и редактора) и примеры. Весьма убедительно выглядят, например, апелляции к повседневной жизни, мгновенно разъясняющие суть дела. И конечно, самое главное то, что мы получаем сведения из первых рук.

Заметим однако, что издание книги Фокса в серии “Математическое обеспечение ЭВМ” несколько противоречит сложившимся традициям. Книги этой серии, как правило, посвящены техническим вопросам: как устроены системы пли программы, как их лучше кодировать, какие при этом используются средства, приемы и т.п. Все это можно было бы назвать “технологией кодирования”. Вопросы же общей организации производства, как правило, изданиями этой серии не освещаются. Поэтому, конечно, хорошо, что технологи и практики кодирования познакомятся с “вышестоящей” проблематикой. Однако даже автор книги говорит, что организация производства и его технология — вещи разные и специалистов в области технологии кодирования или информатики лучше использовать по их специальности, а не поручать им организацию производства программных средств. Поэтому было бы хорошо, если с данной книгой познакомятся именно организаторы этого нового для человечества производства.

Подшивалов Д.Б.

Четырем ребятишкам из Бруклина Джиму, Джоан, Джоку, Пэту.

ПРЕДИСЛОВИЕ

В 1620 г. Френсис Бэкон написал, что беседы ученых мужей часто оканчиваются разногласиями по поводу смысла слов. По-видимому, ни для какой области это не является столь верным, как для вычислительной техники. Родившись в конце сороковых годов, вычислительная машина еще не вышла из младенческого возраста, и научный мир продолжает бороться за создание достаточно объемлющей, согласованной и ясной терминологии.

В 1978г. мне посчастливилось стать председателем комиссии по обследованию вычислительных машин, действующих в ВМФ США, которая готовила доклад для заместителя военно-морского министра США по вопросам научных исследований, техники и систем. Мне сразу стало ясно, что большинство терминов вычислительной техники и программирования все члены комиссии определяли немного по-разному. Не раз нам приходилось прерывать наши заседания, чтобы найти определения, удовлетворившие бы всех.

При обсуждении вычислительных машин и программирования надо остерегаться случайной путаницы, вызванной различным пониманием даже самых распространенных терминов.

Я ввожу в этой книге несколько новых определений, поскольку во многих специальных терминологических словарях не делается различий между близкими, соседними понятиями. Такие словари приносят вред; они только усугубляют неразбериху.

Вычислительная машина может делать все, а известно, что инструмент, который может делать все, иногда может делать весьма странные вещи — и это не случайно. Почти нет ни одного утверждения, касающегося использования вычислительных машин (и, следовательно, программ), которое нельзя было бы опровергнуть специально подобранным примером. Однако наличие таких контрпримеров не означает фактического противоречия. Существование единичной надуманной ситуации, в которой нужно делать нечто противоречащее практике большинства пользователей, вовсе не означает, что это большинство заблуждается.

Вычислительные машины, несколько десятилетий назад доступные только крупным организациям, входят теперь в домашний обиход, а их цена значительно снизилась. То, что стоило двадцать лет назад 2 млн. долларов, теперь можно купить за тысячу. Производители цифровой электроники совершили экономическое чудо: в 1960 г. транзистор стоил около двадцати долларов, а в 1980 г., если бы вам удалось выделить его из большой интегральной схемы, он стоил бы всего 0,000 002 доллара!

В результате вычислительные машины вторгаются во все новые области применения, и не только в своем стандартном виде, как вычислительные машины общего назначения, но также в качестве встроенных управляющих устройств в копировальных установках, телевизорах, автомобилях, локомотивах, кухонных плитах для приготовления пищи, игрушках и т.д. и т.п. Они становятся вездесущими.

Этот замечательный прогресс в изготовлении аппаратуры предъявил дополнительные требования к программному обеспечению и сделал еще более острой нехватку разработчиков этого обеспечения.

Данная книга посвящена программному обеспечению, разработке программного обеспечения и главным образом разработке крупномасштабного обеспечения.

Многие понятия и утверждения в области разработки крупномасштабного программного обеспечения можно относить и к разработке небольших программ; многие, однако, нельзя. Вместо того чтобы прерывать изложение и пытаться разделить эти случаи, я

решил предоставить читателю возможность решать самому, в каких ситуациях справедлива та или иная идея.

Эта книга адресована всем занимающимся обработкой данных. Она посвящена программному обеспечению, т.е. важному технологическому инструменту следующих двух десятилетий. Книга будет интересна и опытным профессионалам, которые в своем большинстве очень сильны в одной или двух областях программного обеспечения и довольно плохо разбираются во многих других. Она предназначена и молодежи — недавним выпускникам-программистам, которые на самом деле знают больше, чем они думают. Книга может служить им своеобразным обзором.

Книгу могут читать и начинающие программисты или пользователи, которые знают прикладные области со всеми их нюансами, но пугаются жаргона и путаницы, царящих в мире программирования. Книга будет полезной также и для руководителей работ, не связанных с обработкой данных, но желающих получить представление о предмете. Первая часть книги вполне доступна тем, кто не является специалистом программировании. И наконец, эта книга рассчитана также и на тех, кто занимается разработкой тех или иных частей программного обеспечения.

Итак, в первой части книги возводится фундамент, на котором строятся идеи программного обеспечения. Здесь я даю определения, разъясняю их смысл и делаю различия между ними. Я вполне осознаю, что многие из них не согласуются с общепринятыми в некоторых областях, но я не знаю никакого другого способа ясно изложить материал. Останется ли пользователь со своей старой терминологией или начнет привыкать к терминологии книги — не так важно; гораздо важнее, чтобы идеи излагались ясно, не запутывались второстепенными понятиями из-за неопределенности формулировок.

Изучение процесса разработки будет продолжаться в гл.5. Основной объем текста связан с этим процессом, но первые четыре главы все же особенно важны для ясного понимания остальной части книги. Я призываю профессиональных программистов не пропускать первой части книги: хотя многое из нее всем известно, но в ней есть некоторые тонкости классификации, необходимые в повседневной деятельности.

В очень многих книгах предмет программирования рассматривается с точки зрения отдельного программиста. Детальное обсуждение рекурсивных функций, свойств языков и теорем, излагаемых техническим языком, помогает программисту справиться с его текущей задачей.

Но в очень немногих книгах ставится цель осветить проблемы, возникающие у руководителей процессом программирования и ходом работ над проектом, а также у руководителей высшего ранга.— каково общее положение дел, что наиболее важно, каковы тенденции, какие возможны ловушки, какие наиболее важные уроки извлечены за последнее время в мире программирования? В течение семи с половиной лет я имел счастье руководить большим программистским предприятием — отделением фирмы IBM. В моем подчинении было 4400 человек, в большинстве своем профессионалы, в основном создававшие программное обеспечение для больших систем реального времени. Многие уроки достались мне очень тяжело. Мы всегда шли непроторенными путями, одновременно приходилось выполнять работы по сотне и более контрактов.

Моя работа была похожа на тушение пожара в нефтяной скважине или на что-то в этом роде. Не успею я взять под контроль один большой проект, как другой внезапно, без всякого предупреждения взрывается — и горит ярким пламенем все отведенное для этого время, несмотря на все усилия овладеть ситуацией.

Мы работали над крупными проектами: 700 человек в течение 10 лет в Хьюстоне, шт. Техас, создавали систему наземного управления пилотируемыми космическими полетами; 500 человек в течение 10 лет в Атлантик-Сити, шт. Нью-Джерси, в местном отделении FAA, разрабатывали систему управления транспортным воздушным сообщением; 700 человек в течение 5 лет в Уиппани, шт. Нью-Джерси, работали над программной частью системы противоракетной обороны; 500 человек в Лос-Анджелесе в течение 5 лет делали спутниковую систему ВВС США.

Ниже я привожу примеры работ, проведенных под моим руководством за 7,5 лет, с 1969 по 1977г.:

Люди/Годы	Место	Проект
700/7	Хьюстон, шт. Техас	“Аполлон”, “Скайлэб”, “Шаттл”
700/5	Уиппани, шт. Нью-Джерси	Противоракетная система “Сэйфгард”
500/7	Атлантик-Сити, шт. Нью-Джерси	Управление воздушными транспортными линиями FAA (Federal Aviation Agency — Федеральное авиационное агентство)
200/2	Мыс Кеннеди, шт. Флорида	Система запуска ракет
60/3	Гейтсбург, шт. Мэриленд	Японская издательская система
20/3	Нью-Йорк, шт. Нью-Йорк	Диспетчерская служба полиции города
5/1	Гейтсбург, шт. Мэриленд	Автоматизация службы сбора информации газеты “Нью-Йорк таймс”
5/5	Лондон, Великобритания	Общенациональная банковская система
53/3	Гейтсбург, шт. Мэриленд	Система межиздательских связей США
200/3	Хьюстон, шт. Техас	Автоматизация нефтеочистительных заводов (Канада, Бельгия)
15/2	Сент-Луис, шт. Миссури	Автоматизация железнодорожной связи, МОРАС

Один вывод, который можно сделать из этого списка, состоит в том, что программирование по контракту имеет разбросанный, кочевой характер. Контора моя была в Гейтсбурге, шт. Мэриленд, хотя большая часть работ проводилась по месту расположения заказчиков. И для этого были причины — разработка программного обеспечения проходит достаточно сложно, даже когда разработчики и пользователи находятся в одном месте.

Нашей группе из Гейтсбурга прекрасно удалось автоматизировать выпуск двух японских газет после полосы неудач. В течение всех 7,5 лет я имел удовольствие получать отчеты от Милса, его огромный талант помогал нам неоднократно. Х.Милс и Т.Бэйкер создали архивную систему газеты “Нью-Йорк таймс” с поразительной производительностью. По окончании работ у нас получилось, по моему мнению, самое большое в мире запрограммированное собрание статистических данных. Эта система используется уже более 8 лет и стала национальным богатством. В этой книге мы неоднократно будем обращаться к проекту “Нью-Йорк таймс”.

Председатель правления фирмы IBM говорил мне, что система управления транспортными авиалиниями никогда не заработает, я же отвечал, что уверен в успехе. Наша система была раскритикована и конгрессом США за то, что в ней повторно тратились усилия на то, что уже было сделано нами в Хьюстоне для посадки человека на Луну.

Оглядываясь назад, я вижу, что короткий промежуток лет вместил много систем, кризисов, ошибок, успехов. Система управления воздушными транспортными линиями не просто работает, Великобритания купила у IBM вычислительное оборудование на сумму 10 млн. долларов, став тем самым двадцать первым участком этой системы. Для работы над этим проектом мы выделили десять человек, которые располагаются в Гейтвиче, Англия. То, что другая страна, к тому же располагающая собственным агентством по управлению воздушными перевозками (и ограниченная беспощадной политикой “покупать только британское”), готова заплатить 10 млн. долларов за “американскую систему”, заставило замолчать всех критиков

За эти 7,5 лет, что я занимался работой по созданию крупного программного обеспечения, мне еще довелось провести для высших руководящих работников серию семинаров, посвященных основным вопросам программирования. Из 7,5 лет руководства группой программистов я вынес мучительный опыт. Преподавательская деятельность научила меня преподносить идеи другим, но, что более важно, я понял, какие из концепций вычислительной техники и программирования кажутся наиболее странными непосвященным, т.е. администраторам. Этот второй мой опыт очень ценен. Вещи, которые я считал заведомо ясными и простыми, на самом деле не были таковыми.

В течение последних 4 лет я был председателем консультационной фирмы по разработке и использованию программного обеспечения. Мои коллеги и я разрабатывали планы, давали советы, проводили ревизии и вырабатывали требования и для небольших систем, и для самых крупных из когда-либо созданных. Мы проделывали это и для министерства обороны США и некоторых его подразделений, и для больших коммерческих фирм. Выводы, сделанные из подобной деятельности, собраны в тексте книги.

Только некоторые из идей, встречающихся в книге, принадлежат лично мне; я мог привлечь к работам наилучших специалистов в данной области за последние двадцать пять лет.

В частности, я хочу признать огромный вклад Эндрью Ферентино, моего коллеги по работе в IBM, а теперь также по фирме Software Architecture and Engineering, Inc. Интуиция Энди, его глубокое понимание программного обеспечения, его определения помогли сформироваться многим идеям этой книги. Многие цифры и понятия принадлежат ему

По мнению многих, выражение: “May you live in interesting times”[1] *{“Желаю вам жить в интересное время” (англ.)}* считается проклятием. Проклятие это или благое пожелание — это вопрос философский, и мы не будем его здесь рассматривать, но я уверен, что

последняя четверть века, а также последующий десяток лет человечество запомнит как время захватывающих перемен в области автоматизации интеллекта. Мы должны быть счастливы, что нам довелось жить в это время открытий и достижений.

Я хочу также поблагодарить Лауру-Анну Чарлз и Жанет Далчер за помощь, оказанную в подготовке книги к печати.

Вашингтон, Джозеф М.Фокс

ГЛАВА 1. ЧТО ТАКОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ?

Необыкновенные достижения вычислительной техники привели к тому, что термин “программное обеспечение” стал толковаться чрезвычайно широко. Столь разнообразна эта область, что утверждения, кажущиеся противоречащими другим, безусловно верным утверждениям, также могут оказаться верными.

Программное обеспечение становится все более значительным, сложным и опасным. Программное обеспечение все время упрощается, уменьшается в размерах и все легче поддается управлению. Оба утверждения верны, поскольку развитие программного обеспечения идет одновременно по двум совершенно разным направлениям. Прежде чем говорить о программном обеспечении нетривиальные вещи, следует выяснить, на какие категории оно подразделяется. Ключом к пониманию проблемы должен стать терминологический словарь.

Важные для себя вещи или абстрактные понятия люди всегда обозначали особыми словами.

Снег	У эскимосов в словаре 20 слов; у ацтеков — ни одного
Коровья шкура	У гаучо (южноамериканские ковбои) — 30 слов
То, что летает	В языке хапи одно слово служит одновременно для обозначения комара и самолета

В этом отношении область программного обеспечения находится в печальном положении. Слова, которые в нем используются, расплывчаты и неоднозначны.

Определение программы

Программа — это последовательность команд или операторов, которая после декодирования ее вычислительной машиной либо вычислительной машиной и транслирующей программой может заставить эту машину выполнить некоторую работу.

В этом определении, которое мы не раз будем исследовать и в дальнейшем, много коварства. Слова *команда* и *оператор* подразумевают наличие точного языка в отличие от *приказа пользователя.* Приказы пользователя, казалось бы, нигде не запоминаются, по крайней мере до поры до времени, а операторы и команды остаются в рабочей программе, которую можно многократно использовать.

В гл. 8 мы будем исследовать это различие между пользователем и программистом, ведь это одна из тех областей, которая может в будущем повлиять на повышение производительности труда при разработке программного обеспечения.

Рабочая программа это программа на машинном языке. *Исходная программа* может быть написана на языке ассемблера или на языке более высокого уровня (ЯВУ). С помощью программы-транслятора она переводится на машинный язык.

- Программа — это последовательность команд или операторов, которая после *декодирования* ее вычислительной машиной и транслирующей программой заставляет эту машину выполнить некоторую работу.

Программное обеспечение и программы

До сих пор мы еще не определили термины, связанные с некоторыми важнейшими понятиями, которыми пользуемся ежедневно.

Совершенно ясно, что слова “окрестность” и “город” обозначают различные понятия. Но, хотя мы понимаем их смысл и вполне различаем их, четких границ между понятиями нет.

Что делает окрестность окрестностью? Как отличить одну окрестность от другой? Когда небольшой город становится большим городом?

Точно так же можно сопоставить понятия *программа* и *программное обеспечение.* Не проводя между ними резкой границы, мы все же различаем их и, глядя на них, понимаем, что они отличаются друг от друга.

Программа имеет жесткие связи, *программное обеспечение* — это более свободно связанная *группа программ,* работающая как единое целое.

Программа “маленькая”, а программное обеспечение “большое”. Подходы к их разработке различны, это мы еще будем изучать в дальнейшем.

Программное обеспечение — это группа взаимодействующих друг с другом программ.

Это не определение, поскольку из него еще не следует различие между большой программой, состоящей из множества отдельных модулей, и группой довольно независимых программ, питающих друг друга данными и привязанных друг к другу. Несмотря на то что четкую границу между программой и программным обеспечением провести не удается, понимание различия между ними необходимо и полезно, как полезно понимать различие между небольшим городком и огромным городом. Таким образом, в этой книге я буду пользоваться термином *программное обеспечение* для обозначения большой группы взаимосвязанных и взаимодействующих программ, в то время как термин *программа* будет обозначать значительно меньший по размерам, более тесно связанный список команд, выполняющих более унифицированную функцию.

Что же не входит в программное обеспечение?

Используя слова *программное обеспечение,* многие люди включают сюда диаграммы, технические требования, графики разработки, инструкции по пользованию и другие подобные побочные продукты фазы разработки набора программ. Мы же включать все это в наше понятие программного обеспечения не будем.

Отсюда вовсе не следует, что эти понятия не очень важны. Как мы еще увидим, они чрезвычайно важны! Схема электропитания автомобиля или принципиальные схемы ЭВМ совершенно необходимы для правильного использования и ремонта дорогостоящего оборудования. Так же обстоит дело и с сопроводительной документацией программного обеспечения.

Однако принципиальные схемы *не являются* продукцией. Автомобиль это система, выполняющая некоторую работу. Он может возить меня, куда я захочу. Учебники по вождению, справочники по уходу, рисунки, необходимые для ремонта или производства, сделанные с различными уровнями детализации, важны, но они не являются частью автомобиля! Существенны и важны по многим причинам, но все же они существуют отдельно. Они *производятся* совместно с автомобилем.

Все это верно и для программного обеспечения. Программное обеспечение — это наш автомобиль: это та “вещь”, которая выполняет работу. Мы должны тратить много времени на создание сопроводительной документации на программное обеспечение и на процесс взаимодействия пользователей с системой при помощи программного обеспечения.

Одно из наиболее вредных часто встречаемых высказываний состоит в том, что некто “пишет программное обеспечение”. Получается так, будто программное обеспечение это что-то вроде письма.

Можно писать *программу,* но не программное обеспечение. И даже с программами трудности значительно возрастают с ростом их размеров.

Фраза “пишет программное обеспечение” не полностью отражает истинные усилия, затрачиваемые на разработку программного обеспечения.

Программное обеспечение мы создаем. Я намеренно выбрал слово “создаем”. Образ действий по созданию программного обеспечения подразумевает много такого, что еще не стало частью научной или инженерной деятельности. Формулировка заданий, решений, проектирование программного обеспечения требует изобретательности и творчества. Как только эти фазы преодолены, процесс становится более механическим.

Разработка программного обеспечения может быть разбита на шесть этапов:

Определение требований и заданий
Проектирование
Написание команд — программирование
Компоновка
Тестирование
Документирование

Относительно новым термином “программно-аппаратное обеспечение” часто обозначается программное обеспечение или просто программа, которая всегда находится в постоянной памяти, работающей только на чтение (ПЗУ), и выполняется непосредственно оттуда. Хотя тот факт, что запись в память исключается, и создает некоторые трудности проектирования и создания программного обеспечения, однако программно-аппаратное обеспечение не столь уж сильно отличается от программного обеспечения, работающего с обычной памятью, чтобы его нужно было выделять и рассматривать отдельно. Здесь применяются те же правила и практические приемы. Таким образом, мы будем рассматривать *программно-аппаратное обеспечение* как часть программного

обеспечения. Мы еще столкнемся с программами, располагающимися в постоянных запоминающих устройствах.

Обзор программного обеспечения

Наметившийся кризис программного обеспечения в настоящее время продолжает углубляться. Частично этот процесс происходит из-за того, что лишь немногие из людей, не занимающихся программированием профессионально, разбираются в том, что такое программное обеспечение. Программное обеспечение представляет собой обособленное, замкнутое в себе поле деятельности, семантика которого неоднозначна, а терминология бедна. Практикующиеся в этой области слишком часто концентрируют свое внимание на самом программном обеспечении, в то время как надо больше внимания уделять системной стороне дела.

Очень часто разработчики программного обеспечения не успевают выдать надежный продукт точно в назначенный срок или просто выходят из бюджета. Какая-либо помощь этим разработчикам практически не увеличивается, и разумного решения тут вообще не видно. Неудачи на поприще программного обеспечения постигают не только начинающих программистов. Случается, что даже самые крупные, самые солидные, самые технологически передовые организации страдают из-за серьезных препятствий, возникающих перед ними. За несколько недель до намеченной сдачи в эксплуатацию новой системы 138 фирма IBM объявила о годовой отсрочке, вызванной трудностями с программным обеспечением. За год до назначенного ввода в действие усовершенствованной системы связи фирма ATT объявила о 18-месячной задержке.

Пользоваться программным обеспечением легко, разрабатывать его намного сложнее. Разработка программного обеспечения — это выпущенный из “волшебной” (электронной) лампы джинн, который скитается по свету и повсюду отравляет жизнь разработчикам систем.

Одной из главных причин такого большого числа трудностей в деле разработки стал постоянный значительный рост сложности и масштабов работ, проводимых (и доводимых до успешного конца) программистами. Постоянно находятся все новые, неожиданные области применения “старого” инструмента — вычислительной машины. Прогресс электронной техники в скорости, размерах, надежности, в экономических показателях кажется просто фантастическим.

Если мы оглянемся на проделанное нами, то увидим, каких гигантских успехов мы добились. Системы резервирования авиационных билетов работают — и работают прекрасно. Американская система авиалиний к весне 1979 г. обслуживала 13500 присоединенных терминалов, через которые поступало до 10,6 млн. сообщений в день. Система управления транспортными авиалиниями работает и в США, и за их пределами. Работает спутниковая система. И эти успехи не случайны. На них было затрачено огромное количество ресурсов и усилий, но они работают.

Какие же уроки можно извлечь из прежних успехов и неудач? На какой стадии находится “техника программирования”? Следует ли нам относиться к программному обеспечению так же, как мы относимся к аппаратному?

Эта книга посвящена в первую очередь разработке больших систем программного обеспечения, которые встречаются все чаще. Мы исследуем, почему многие крупные разработки столкнулись с большими трудностями, вызвавшими незапланированные затраты бессчетных миллионов долларов, многолетние отсрочки и даже откровенные неудачи, а также немало судебных процессов.

Разработка программного обеспечения умеренных масштабов представляет собой по большей части историю успехов со счастливым исходом. Многие рассматриваемые нами в дальнейшем определения и приемы, использованные при построении больших систем, заимствованы из более ранних попыток реализации небольших систем или применимы к ним. Мы еще будем обсуждать эти вопросы.

Приведем теперь некоторые не очень очевидные факты, связанные с программным обеспечением, а затем коротко обсудим их. Оставшаяся часть книги будет посвящена более подробному их описанию.

Факты о программном обеспечении

Развитие программного обеспечения происходит одновременно в двух противоположных направлениях.

Жизнь любой программы обычно проходит три стадии, и в своей работе разработчики и проектировщики должны принимать во внимание все эти три стадии. Обычно рассматриваются только стадия разработки и стадия использования. Однако уже на ранних этапах разработки нужно иметь в виду и *стадию поддержки (сопровождения)* или *продолжающейся разработки*.

Разработка программного обеспечения проходит следящие шесть этапов: определение требований, проектирование, написание команд, компоновка, тестирование и документирование.

Разработка больших систем программного обеспечения часто зависит от наличной аппаратуры.

Любой процесс может быть выражен несколькими различными “правильными” последовательностями команд.

Программное обеспечение носит абстрактный характер, что усложняет процесс его разработки.

Основным инструментом создания нового программного обеспечения являются вычислительная машина и ее программное обеспечение.

При разработке программного обеспечения основную трудность обычно представляет собой не та функция, которую должна выполнять данная система, а методика взаимодействия с пользователем, которой должна подчиняться эта система

Некоторые виды программного обеспечения можно разрабатывать теми же методами, что и аппаратуру, другие же так разрабатывать нельзя.

Правильное программное обеспечение не подвержено никаким сбоям. Термин “поддержка” по отношению к программному обеспечению является, следовательно, неправильным.

Разработка больших программ — это весьма многогранная деятельность, отнюдь не связанная только с работой на вычислительной машине

Большая система программного обеспечения никогда не может быть отлажена до конца, даже после нескольких лет тестирования и использования.

Разработка программного обеспечения часто весьма трудоемкий и дорогостоящий процесс.

Программное обеспечение это не цель, а средство.

Развитие программного обеспечения происходит одновременно в двух противоположных направлениях. В середине 30-х гг. XX столетия английский математик Ален Тьюринг доказал, что любой процесс, который можно описать каким-либо алгоритмом, может быть реализован с помощью простейшей машины, которая выполняет всего шесть различных команд, хотя это может занимать весьма значительное время. Из этого принципа логически следует тот факт, что вычислительная машина — любая вычислительная машина общего назначения — может выполнить все, что только может быть описано с помощью алгоритма. Современное программное обеспечение становится все более сложным, находит все более широкие и сложные приложения и в то же время делается “обыкновенной” продукцией повседневного пользования обывателей

Огромный прогресс технологии производства интегральных схем драматически понизил цены на аппаратуру, и этот процесс будет еще продолжаться. То, что казалось недоступным еще несколько лет назад из-за цены или нехватки времени, вполне достижимо сегодня.

Одновременное уменьшение стоимости и увеличение мощности машин расширили область применения ЭВМ сразу на верхнем и нижнем уровнях.

На верхнем уровне, связанном с широкомасштабными усилиями, вычислительные машины получают возможность выполнять задачи за требуемое время, и эти задачи стали теперь решаться. А на нижнем уровне простые машины с невысоким быстродействием становятся такими дешевыми, что их теперь экономически выгодно использовать для автоматизации тех процессов, которые еще год или два тому назад не имело смысла автоматизировать.

Это развитие вычислительной техники в двух направлениях вроде бы не должно никого удивлять и смущать, но иногда это происходит. Программное обеспечение для верхнего уровня становится все более сложным, для нижнего же уровня программы постоянно упрощаются.

Программное обеспечение состоит из множества программ различных типов. Программы делают все, что только можно; они бывают маленькими и тривиальными, но бывают также большими и сложными, весьма дорогостоящими. Говорить о программном обеспечении можно только употребляя при этом по крайней мере одно уточняющее слово. О чем мы говорим — о “программном обеспечении проекта” или о “программном обеспечении как продукции”? О “реальном времени” или о “пакете”? А может быть о “диалоге”? Что мы имеем в виду: “помехозащищенность”, “помехобезопасность” или просто надежность? О каком программном обеспечении идет речь — о “инструментальном”, “системном” или “прикладном”? О “крупномасштабном” или “мелкомасштабном” программном обеспечении? Каждая из этих категорий обладает собственными, характерными только для нее чертами.

Жизнь программы делится на три фазы. Слишком часто разработчики программ фокусируют свое внимание только **на фазе разработки,** иногда еще и на фазе **использования** и очень редко на **фазе сопровождения** или **продолжающейся разработки** к большему ущербу для последней. Кое-что можно и нужно делать во время разработки для облегчения других фаз. Большинство

разработчиков этот аспект игнорируют. Во многих книгах, посвященных разработке и использованию программ, игнорируется тот факт, что эти три фазы существуют отдельно друг от друга, имеют различные цели и часто к тому же проводятся под разным руководством (см. рис. 1.1).

Разработка программного обеспечения состоит из шести[1] фаз: определение требований, проектирование, написание команд, компоновка, тестирование и документирование. *Простейшее в этом списке — написание команд.* Очень трудно проектировать оптимальное или хотя бы правильное взаимодействие большого числа программ. Для больших систем труднее всего сформулировать требования пользователей, а следовательно, и самих этих систем. И все же большинство учебных курсов и книг в основном делает упор именно на написание команд Как мы увидим в дальнейшем, особое внимание надо уделять определению требований и проектированию.

Разработка программного обеспечения для больших систем часто зависит от аппаратуры. Большие системы обычно требуют, чтобы аппаратура в них использовалась оптимальным образом в целях экономии и увеличения производительности.

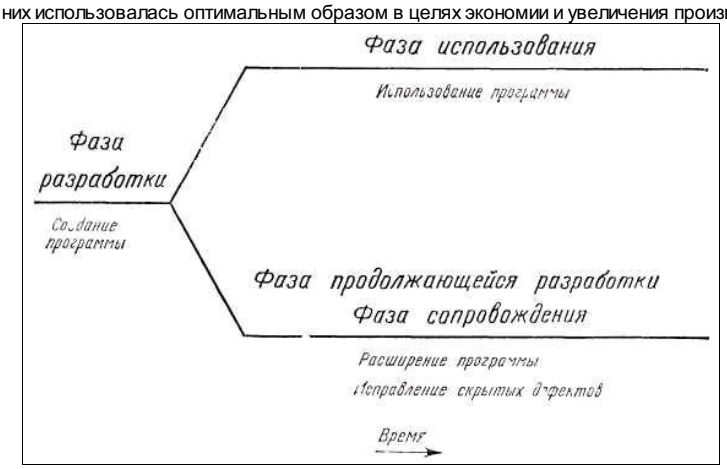


Рис 1.1 Три фазы жизненного цикла программы

Особенно важно правильно спланировать загрузку центрального процессора (ЦП) и определить все требования к памяти. Все это заставляет разработчиков при проектировании программ учитывать параметры и характеристики аппаратуры системы В случае разработки маленьких программ дело обстоит совсем иначе

Производительность системы — т.е. способность вычислительной машины выполнить задание за определенный промежуток времени — в большой степени зависит от используемой аппаратуры, ее мощности и состава. Если аппаратура плохо подходит для решения поставленной задачи, вся тяжесть достижения необходимых *пределов* производительности ложится на плечи разработчиков программного обеспечения, и в особенности на тех, кто занимается проектированием. Это основная причина того, что иногда разработчики больших систем или систем реального времени вынуждены отказываться от использования языков высокого уровня. Мы увидим еще несколько примеров того, как требование достижения высокой производительности усложняет разработку.

Любой процесс может быть описан несколькими различными "правильными" последовательностями команд. Сотня программистов смогла бы написать сто различных программ платежной ведомости, каждая из которых будет выполнять то, что от нее требуется. Небольшое количество среди них будет оптимальным. Важные характеристики каждой программы порой конфликтуют друг с другом, подобно тому как при конструировании самолетов размеры вступают в конфликт со скоростью. Прочитав гл.5, вы узнаете, что каждая программа имеет по крайней мере 12 характеристик и некоторые из них, находя воплощение в одной программе, могут вступать с другими в противоречие. Множественность возможных решений является серьезным препятствием для правильного управления процессом разработки программного обеспечения.

Абстрактный характер программного обеспечения. Ни одним из пяти человеческих чувств нельзя ощутить большую программную систему. Она абстрактна. Можно увидеть и пощупать вычислительную машину стоимостью в 20 млн. долларов или мост, стоящий 100 млн. долларов. По мере увеличения числа команд в программе ее обозрение все более затрудняется.

Посмотреть на список из миллиона строк команд отнюдь не так же легко, как посмотреть на мост. Невозможно *увидеть* ни последовательность выполнения команд, ни структуру программы. Можно *увидеть* лишь толстую пачку бумаги, строки различных символов, но никак не программное обеспечение. Это свойство программного обеспечения делает управление его разработкой очень трудным.

Основным инструментом создания нового программного обеспечения является вычислительная машина и ее программное обеспечение. Разработчики программного обеспечения в своей работе используют ЭВМ и программное обеспечение как основной инструмент для формирования и создания новых программ. Программное обеспечение является при таком использовании "инструментом для изготовления инструментов".

Рассматривать программное обеспечение можно либо в роли исполнителя задания (период использования), либо в роли помощника при создании нового дополнительного программного обеспечения (период разработки). Эти роли абсолютно различны, и тот факт, что одну и ту же аппаратуру можно использовать для выполнения обеих функций, является источником крайнего изумления для многих новичков.

Для тех же, кто постоянно работает с вычислительными машинами и программным обеспечением, этот факт настолько очевиден, что они думают, что это и для всех само собой разумеется. Их не смущает, что в одном предложении они обсуждают, скажем VAX-11 в роли исполнителя операционной системы, контролирующей полет ракеты, а в следующем предложении переходят к обсуждению работы той же физической машины, расположенной в том же самом месте, работающей в той же конфигурации, но уже в роли "инструментальной" машины, которая используется для сопровождения программ, выполняемых в тот момент, когда VAX-11 контролирует полет ракеты.

Для людей, далеких от таких способов разработки, такое изложение может показаться туманным, поскольку в тексте перемешиваются предложения, рассказывающие о двух совершенно различных способах использования одной и той же машины.

Эту двойную роль следует четко выделить. Переходить **с** описания одной роли машины **к** описанию другой профессиональные программисты должны **с** осторожностью, обязательно оговаривая такие переключения.

Трудность разработки программного обеспечения часто заключается не в тех функциях, которые оно выполняет, а в способе взаимодействия с пользователем. Требования к надежности системы часто гораздо сильнее затрудняют разработку, чем та функция, которую должно исполнять программное обеспечение. Программу, управляющую полетом самолета, сделать легко. Очень трудно добиться того, чтобы эта программа никогда не переставала работать, несмотря на аппаратные и программные сбои и ошибки летчиков. И все же факторы взаимодействия часто остаются вне поля зрения разработчиков либо рассматриваются ими неверно. Разрабатывать самовосстанавливающиеся программы это одно из самых сложных дел.

Некоторые виды программного обеспечения могут разрабатываться теми же методами, что и аппаратура, другие же так разрабатывать нельзя. Программное обеспечение состоит из отдельных команд, а мы можем написать команды, чтобы выполнить почти все что угодно, да еще массой различных способов. Команда не является физической "вещью". Аппаратура же — вещь физически реальная, и поэтому переставлять и комбинировать ее так свободно нельзя. Мы выделяем те отдельные случаи, когда программы становятся похожими на аппаратуру и должны разрабатываться теми же методами. Однако в больших системах аппаратура и программы имеют существенные различия. Аппаратуру можно здесь сравнивать с роялем, а программное обеспечение с музыкой. Аппаратура это словарь, а программное обеспечение — роман. Итак, несмотря на большое количество сходного в аппаратуре и программах, при разработке больших систем на первый план выступает их принципиальное различие. Более подробно этот вопрос будет рассмотрен далее.

Правильное программное обеспечение не подвержено никаким отказам. Оно, следовательно, не нуждается в техническом обслуживании. Подобный процесс нужно называть не обслуживанием, а продолжающейся разработкой. Если команды написаны правильно, они не могут вдруг стать неправильными. Они могут устареть, но происходит это по другим причинам. Они могут

также быть неверными с самого начала, что, однако, не было обнаружено при тестировании. Обслуживание программы, иначе говоря продолжающаяся разработка, состоит в выявлении ранее не обнаруженных ошибок, добавлении новых функций, модификации некоторых функций, повышении производительности.

Этот факт чрезвычайно важен! Многие программисты считают, что обслуживание — это дело не интересное и не такое престижное, как разработка программ. Однако продолжающаяся разработка часто оказывается делом более сложным, чем первоначальная разработка. Мы должны называть этот процесс более точно — именно продолжающейся разработкой — и тем содействовать стиранию позорного и незаслуженного пятна, лежащего на тех, кто занимается обслуживанием.

Разработка программного обеспечения — это многогранная деятельность, связанная не только с работой на вычислительной машине. При работе над проектом большой программной системы обычно необходимо быть компетентным и в общих инженерных вопросах, и в математике, нужно разбираться в различных видах вооружения, знать законы небесной механики, разбираться в военной логистике, знать методы экономического планирования, понимать, что такое резервирование; необходимо ориентироваться в теории межпланетных полетов и в аэродинамике, а также еще в огромном количестве различных областей знаний. Руководитель разработки программного обеспечения должен *в своем проекте все это учитывать.* Это означает, что этот руководитель должен обладать способностью управлять группой специалистов из различных областей.

Большая программная система никогда не может быть отлажена до конца, даже после многих лет тестирования и использовании. До конца могут быть отлажены только маленькие программы, большие же и сложные не могут. Слишком много путей предусматривает общий алгоритм программы, слишком много возможных вариантов вводимых данных или действий пользователей.

Даже сотни лет продолжающегося тестирования — если бы, конечно, это стало возможно — не хватило бы для проверки всех возможных ветвей большой, сложной программы. И несмотря на это, находятся люди, всерьез говорящие о “программном обеспечении, свободном от ошибок”!

Разработка программного обеспечения часто весьма трудоемкий и дорогостоящий процесс. В Соединенных Штатах Америки каждый год вкладывается в разработку программного обеспечения больше 20 млрд. долларов. Это очень много, а ведь имеется устойчивая тенденция роста таких вкладов. И не все капиталовложения окупаются.

В начале 1970-х гг. две основные авиакомпании возбудили судебное дело против своих подрядчиков, поскольку созданная ими система стоимостью 40 млн. долларов не только не работала, но и вообще не подавала никаких признаков жизни. Крупный европейский банк направил в суд иск о взыскании 70 млн. долларов, выплаченных за программное обеспечение. Военно-воздушные силы США затратили более 300 млн. долларов на тщетную попытку автоматизировать комплексную систему перевозок и снабжения.

Программное обеспечение — это средство, а не цель. Одна из замечательных технических групп работает в National Security Agency (NSA). Она специализируется в области защиты и разгадывания шифров и привлекает в свои ряды и воспитывает знатоков различных технологических методов.

Недавно один человек, занимающийся вычислительными машинами и средствами связи, поведал мне о событиях, происшедших несколько лет назад, когда он был председателем Комитета определения стандартов для приобретаемого программного обеспечения.

Один из членов его комиссии, страшный зануда, постоянно настаивал на том, чтобы методика “сверху — вниз”, весьма популярная среди разработчиков программного обеспечения, использовалась и в их комитете. Это означало, что им *нельзя* проводить совещания по программному обеспечению, пока не обсуждены системы, для которых необходимо это обеспечение и частью которых оно должно стать.

Наконец председатель был вынужден сходить к главе NSA и заявить ему, что указанный член комиссии был прав — *сначала* комитет должен разработать стандарты и руководства на приобретение *систем,* а уж затем переходить к программному обеспечению.

Хотя глава NSA чувствовал, что им известно, как нужно производить приобретение систем, первым документом, изданным комитетом, был документ под названием “Политика приобретения систем”. Лишь затем удалось заняться программным обеспечением!

Программное обеспечение это продукция “третьего порядка”, оно помогает работать всей системе, а *система* уже достигает нужных результатов.

Само по себе программное обеспечение никогда нельзя рассматривать как конечную продукцию. Даже как рыночный товар оно имеет вспомогательный характер и должно разрабатываться и продаваться с учетом этого.

Что мы имеем в виду под “третьим порядком”? На первом месте, безусловно, находится желаемый результат. Например, увеличение продажи билетов, происшедшее из-за введения в действие системы предварительного заказа. Результат — это то, к чему мы стремимся, он является целью первостепенной. На втором плане находится система, достигающая данного результата, а программное обеспечение, как часть этой системы, занимает лишь третье место. Важное, это верно. Может быть, самое важное, но все же лишь третье.

Зачем заострять внимание на этом очевидном факте? Уинстон Черчилль определял фанатика как человека, который не изменяет своего решения и не переводит разговор на другую тему. Нам же постоянно приходится сталкиваться с “программистами-фанатиками”, которые считают, что важнее программного обеспечения нет ничего на свете. Этим фанатикам нельзя доверять руководство системами или проектами, поскольку у них искаженное представление об окружающем мире и извращенное суждение о программном обеспечении, системе и результате.

Определение требований — это наиболее неразработанная область, связанная с созданием программного обеспечения. Эта область связана отнюдь не только с программным обеспечением; здесь поднимаются вопросы, связанные и с всевозможными типами аппаратуры, и с людьми, и с технологией! Если программное обеспечение рассматривать как вспомогательный механизм, получающаяся в результате продукция будет значительно лучше, чем в том случае, если считать его первостепенным и наиважнейшим фактором.

Две роли программного обеспечения

Очевидная роль программного обеспечения — заставить работать аппаратуру! Но давайте заглянем поглубже. Этим роль программного обеспечения не ограничивается. Оно еще должно быть легко модифицируемым.

Никто без серьезных оснований не станет ломать антенну радиолокатора или новый спутник (ценой около 50 млн. долларов). Однако руководители различных систем с легкостью идут на изменения в программном обеспечении! Они должны даже заранее планировать эти изменения. Такая роль была уготована программному обеспечению с самого его возникновения. Изучение истории этого вопроса возвращает нас к понятию запоминаемой программы.

Универсальным устройством вычислительная машина стала с появлением возможности стирать и изменять информацию в своей памяти. Но эта же возможность привела и к тому, что ЭВМ стали гибкими, легко адаптирующимися устройствами. Универсальность и гибкость одинаково связаны с возможностью изменения состояния памяти. Чтобы лучше использовать все выгоды гибкости вычислительных машин, нам необходимо очень тщательно подходить к вопросу *контроля* программного обеспечения и быть всегда готовыми внести все нужные изменения.

Почему проблема программного обеспечения еще не полностью решена?

Почему разработка программного обеспечения часто так трудна? Ниже изложены причины, всевозможные комбинации которых дают ответ на этот вопрос.

1. *Сложность.* Внутренняя сложность автоматизируемого процесса, т.е. научная сложность (лунное притяжение и т.д.), или логическая сложность (число ветвей в программе может достичь значения 10^{3000}), или и та и другая вместе.

2. *Абстрактность.* Управление и создание его происходит “вслепую”, и, для того чтобы “увидеть”, что происходит, требуются весьма сложные приспособления.

3. *Путаница в требованиях.* Руководство обычно убеждено, что вполне понимает все требования пользователей, даже в том случае, если пользователь до того ни разу не видел подобной системы. Кроме того, распространено заблуждение, что первоначально поставленная цель системы (в тех случаях, когда она известна не изменяется. Отсюда часто возникает абсурдное представление, что большая программная система может быть разработана с помощью “большого взрыва”.

4. *Социальные факторы.* Постоянно предпринимаются усилия по снижению стоимости разработки системы. Программная часть системы, как наименее понятная, подвергается наибольшим денежным ограничениям. Именно из-за мало- понятности программного обеспечения легче всего доказать необходимость ограничения капиталовложений в него. Программное обеспечение “невидимо”.

5. *Управление взаимодействием с пользователем с помощью программного обеспечения.* Создаваемое для такого взаимодействия программное обеспечение должно выполнять следующее:

а) обеспечивать, чтобы работа системы никогда не прекращалась, т.е. программы должны управлять аппаратурой, данными и другими программами так, чтобы при возникновении ошибок работа возобновлялась без вмешательства человека.

б) обеспечить, чтобы программа выполняла задание за определенный период времени: таким образом, программы должны

управлять распределением временных отрезков и состояниями составных частей, меняя, если это необходимо, последовательность выполнения работ.

в) интерфейс с пользователем ориентирован на удобство работы пользователя, а не системы, при этом пользователь может использовать систему с минимумом специальных навыков.

Разрабатывать программное обеспечение очень и очень трудно. Оно имеет весьма запутанную логическую структуру.

Глава 2. ВЫЧИСЛИТЕЛЬНАЯ МАШИНА И СПОСОБЫ ЕЕ ИСПОЛЬЗОВАНИЯ

Определение вычислительной машины

Едва ли возможно изучать законы полета, не изучая самого самолета. Обсуждение этих законов обычно проводится в предположении, что слушатели понимают основные принципы действия рулей, закрылков и т.п. К моему ужасу, я часто обнаруживал, что мое представление об аппаратуре вычислительной машины немного, но в главном отличается от представлений моих коллег. Итак, прежде чем заняться более глубоким изучением программного обеспечения, мы просто обязаны изучить ЭВМ и способы ее использования.

Одним из значений слова “определить” является “установить границы”. Предлагаемое ниже определение как раз и обрисовывает и устанавливает границы. Определение дается в два этапа, сначала определяется собственно вычислительная машина, а затем запоминаемая программа и универсальная ЭВМ.

- *Определение: Вычислительная машина* — устройство, которое автоматически и непрерывно что-то вычисляет, выбирая свои действия из множества заранее установленных.

Подробнее обсудим некоторые моменты.

Вычислить: “Выяснить или определить с помощью какого-либо процесса”. Это слово можно интерпретировать многими способами, я использую его в самом широком смысле. Сравнить — это вычислить. Измерение это также вычисление.

Автоматически и непрерывно: Пущенное в ход человеком, данное устройство будет работать “до конца” — до тех пор, пока задание не будет выполнено полностью. Теоретически оно может работать бесконечно долго, если управляемый устройством процесс не имеет логического конца. Здесь мы отмечаем *способность* проводить операции в течение длительного периода времени без вмешательства человека.

Выбор действия из множества заранее установленных: Человек вводит несколько альтернативных вариантов последовательностей операций, а вычислительная машина в соответствии со своими собственными внутренними процессами “выбирает” ту ветвь, которую надо выполнять в данный момент. Все ветви заранее определяются человеком, который программирует для этой машины.

Это определение подходит для всех вычислительных машин, а не только для универсальных. Попробуем теперь обрисовать различия между специализированными и универсальными ЭВМ. Наиболее четкое из всех когда-либо встречавшихся мне разграничений проведено в высказывании знаменитого Джона фон Неймана, написавшего в 1946 г. меморандум, навсегда связавший его имя стой архитектурой ЭВМ, которая используется нами и до сих пор. В этом меморандуме впервые появилась четкая формулировка определения универсальной вычислительной машины. Там говорилось, что во всех специализированных машинах, которые применялись уже много лет, все команды были встроены таким образом, что являлись неотъемлемой частью самого устройства. Другими словами, алгоритм, выполняемый машиной, является частью конструкции этой машины. Фон Нейман указал, что машина может и другим способом контролировать ход операций — с помощью чисел, заносимых в машину.

Очевидно, что машина должна иметь возможность запоминать некоторым образом не только цифровую информацию, необходимую для проведения данных вычислений, например граничные значения, таблицы функций и промежуточные результаты вычислений, а также и команды, которые управляют последовательностью операций, выполняемые с числовыми данными. *В специализированной машине эти команды являются неотъемлемой частью устройства и составляют часть разрабатываемой структуры.* Универсальная машина должна иметь возможность проводить любые вычисления, которые могут быть выражены как вычисления над числами).

Определение универсальной вычислительной машины

Универсальная вычислительная машина — это такая вычислительная машина, команды которой записаны в память и могут быть достаточно быстро заменены там другими командами без какого-либо инженерного вмешательства.

Даже если бы смена команд происходила медленно (скажем, за несколько часов), мы все же имели бы универсальную машину, хотя и медленную.

Это определение отделяет универсальную цифровую вычислительную машину (ЦВМ) от ручного калькулятора и от сложных автоматов, плотно заселивших наше технологическое общество.

Это определение может быть применимо к замечательным и недорогим устройствам, которые производятся на основе больших интегральных схем (БИС). Оно проводит различие между электрической схемой и вычислительной машиной, между калькулятором и ЭВМ.

Это определение охватывает все вычислительные машины, будь то микро-, мини-, миди- или макси-машины. Различие между микро- и мини-ЭВМ очень нечеткое, и постепенно сотрется по мере прогресса БИС. Через несколько лет *все* вычислительные машины станут микро-ЭВМ и будут целиком располагаться на одном полупроводниковом кристалле.

Не будем продолжать обсуждение этого определения. Это уведет нас в сторону от вопросов программного обеспечения. Всякое определение важно, а это к тому же точно и полезна.

Вычислительная машина: орудие труда нового типа

То, как будет использоваться вычислительная машина, определяет и потребность в программном обеспечении. Здесь нам полезно будет взглянуть, как человек использует орудия труда. Мак Лухан утверждает, что каждое орудие представляет собой продолжение функций человеческих органов.

Дополняемый орган	**Орудие**
Нога	Колесо, велосипед, автомобиль, самолет
Кулак	Камень, дубинка, ружье
Кожа	Одежда
Глаз	Телескоп, микроскоп, радиолокатор
Рот, ухо	Радио, микрофон
Мозг	Бумага (память), калькулятор, арифмометр
Мозг	Вычислительная машина
Воля [1)]	Вычислительная машина

[1)] *Имеется в виду воля как способность принимать решения в ходе выполнения какого-либо задания. — Прим перев.*

Какие же способности людей расширяются с помощью вычислительных машин? Оказывается, две очень разные человеческие характеристики — его ум и воля.

“Новинка” — популярная телевизионная передача рассказала однажды о вычислительной машине, названной “умной машиной”. Однако ЭВМ это больше, чем просто дополнительный мозг, и этот факт имеет глубокое влияние на программное обеспечение, которое мы должны разрабатывать с учетом того, что вычислительная машина является еще и продолжением нашей воли.

“**Продолжение” мозга.** Высокая *скорость* работы ЭВМ позволит людям решить неразрешимые до сих пор задачи. Тысячи людей, вооруженных арифмометрами, работая тысячи лет, не смогли бы проделать тех вычислений, которые проводит большая ЭВМ за считанные минуты. Такое использование позволяет вооружить наш мозг, дать возможность решить задачу, выполнить огромное множество арифметических операций.

“**Продолжение” воли.** В несколько ином смысле вычислительная машина “продолжает” человеческую волю. Она выполняет приказы и делает выбор между вариантами, невзирая на то, присутствует рядом человек или нет. Именно эта способность выбирать из разных вариантов и делает ее продолжением человеческой воли. Причем в этом смысле скорость работы ЭВМ совершенно несущественна.

Если лишить машину возможности выбора, она превратится в простое расширение человеческого мозга! Реализуется такой выбор с помощью команд условного перехода, которые являются воплощением функции выбора.

Люди и новые орудия труда. Люди всегда начинают с того, что используют новые орудия труда для решения старых задач. Только через годы или даже десятилетия продолжительного опыта новое орудие начинает использоваться для *новых* целей.

Вычислительные машины быстро проникают во все новые области применения. И именно для этих новых применений разработка программного обеспечения особенно трудна.

Телеграф заменил верховых посыльных. Радио сменило телеграф. Однако надежды на то, что радио будет использоваться вместо телефона (что приведет в исчезновению проводов), не оправдались; это, по-видимому, связано с отсутствием секретности при радиопередачах. Любой, кто имеет радиоприемник, может спокойно слушать все разговоры. Прошло 20 лет использования радио, прежде чем *совершенно случайно* не было обнаружено, что радио это великое средство для развлечения — некий дантист из Питтсбурга начал по субботним дням "трансляцию" музыки из своего гаража, и началась новая эра радиовещания и развлечений с доставкой на дом. То самое свойство, которое не позволило радио сделаться заменой телефона, и сделало его мощным орудием индустрии развлечений.

Дэвид Сарнов, предвидя такое использование радио, в своем полуфантастическом меморандуме писал за целое десятилетие до начала передач упомянутого дантиста:

Мне пришел в голову план развития, в результате выполнения которого радио станет обычным предметом домашнего обихода, каким является пианино или фонограф. Идея состоит в доставке музыки на дом без использования проводов. В прошлом уже делались такие попытки с проводами, но они не удались, так как провода в эту схему явно не вписываются. Радио — это как раз то, что нам необходимо. Можно разработать специальный приемник в виде простейшего "музыкального радиоящика", приспособленного для работы на нескольких различных длинах волн, менять которые можно будет простым поворотом выключателя и нажатием единственной кнопки...

Меморандум Сарнова в точности предсказал то, что и произошло в действительности. Однако его руководство посмеялось над ним, и передачи начала вести первой не его фирма.

Прогресс в использовании вычислительных машин шел в точности по общей схеме. Сначала они выполняли то, что делалось и раньше, без них, и было хорошо известно — платежные ведомости, инвентаризация, баллистические и инженерные расчеты. Лишь постепенно захватывались новые области применения — справочные системы, управление информационными системами и системы разделения времени.

Наконец, вычислительные машины начали использовать в той области, которую я называю *управление процессами,* и именно здесь, в этой *новой области использования* теперь уже "старого" орудия труда, нам наиболее трудно проводить разработку программного обеспечения.

Использование вычислительной машины

Одна из главнейших проблем, связанных с вычислительными машинами, состоит в их крайней разносторонности — их можно использовать буквально для всего. Как еще в 1936 г. показал Тюринг, любая функция или процесс, которые можно выразить с помощью какого-либо алгоритма, могут быть выполнены очень простой машиной, если, конечно, не принимать во внимание временные факторы.

В 1910 г. Бертран Рассел и Альфред Норт Уайтхед выпустили труд "Principia Mathematica", в котором выдвинули принцип, согласно которому основанием всех математических дисциплин является логика. Тем самым доказательство теорем, решения различных задач можно рассматривать в терминах специального вида утверждений, каждое из которых может быть истинным или ложным. Мы обнаруживаем, что вычислительная машина это гораздо более сложное устройство, нежели просто арифметический вычислитель, это еще и логическая машина.

Документы, описывающие различные использования вычислительных машин, состоят из сотен и сотен пунктов. Можно продолжать составление таких списков до бесконечности, но занятие это совершенно бесполезное. Куда больше пользы принесет попытка разделить виды использования вычислительных машин на качественные категории, отличающиеся друг от друга требованиями к аппаратуре, программному обеспечению, людям. Различия должны быть достаточно глубокими, чтобы такое деление было полезным всем тем, кто создает сложные системы, основанные на использовании вычислительных машин.

Мы будем рассматривать пять различных типов использования:

Тип 1. Обработка данных
Тип 2. Решение научных задач
Тип 3. Информационные системы
Тип 4. Диалоговые системы решения задач
Тип 5. Управление процессами

И история развития, и терминология вычислительной техники провели четкое различие между машиной, используемой для коммерческих расчетов — процесса, который отмечен как *обработка данных,* — и машиной для решения задачи, той машиной, что, собственно, и называется *вычислительной машиной.* Аппаратура, разрабатываемая для одного из этих использований, сильно отличалась от аппаратуры, применявшейся в другой области. Лишь недавно (1964) одна и та же аппаратура получила возможность работать и там и там, выполняя обе задачи.

Вычислительная машина для *решения научных задач* была устроена так, чтобы наиболее быстрым образом производить вычисления. Обрабатывающая машина — обрабатывающая данные — была машиной ввода/вывода, машиной, которая управляла внешними файлами. В течение первых 12 лет выпуска вычислительных машин фирма IBM сопровождала две различные ветви машин — машины для обработки данных и научные машины.

Серия машин IBM/360, объявленная в 1964 г., изменила положение, поскольку для IBM стало слишком дорого разрабатывать, строить и сопровождать две различные ветви.

Использование типа I. Обработка данных

Бухгалтерские расчеты проводятся постфактум. Учет векселей, ведение регистрационных журналов, составление платежных ведомостей и проведение инвентаризации есть всего лишь работа с записями о прошедших событиях. Разработанная для проведения быстрых расчетов баллистических траекторий вычислительная машина, отвергнутая в свое время фирмой IBM, была доведена до состояния экономической рентабельности корпорацией Sperry Rand в 1952 г. Пользователи заставили ее составлять платежные ведомости и инвентарные списки.

Использование типа II. Решение научных задач

Многие называют большие вычислительные машины "большими арифмометрами". На основе небольшого числа исходных данных мощная машина может работать часами, решая уравнение теплопроводности или строя прогнозы погоды.

Использование типа III. Информационные системы

После нескольких лет использования люди осознали, что все необходимые им данные находятся "внутри" и вполне доступны. Было написано специальное программное обеспечение, и пользователи получили возможность вводить запросы в вычислительную машину и получать от нее ответы на эти запросы. Родился новый вид использования вычислительных машин.

Использование типа IV. Диалоговые системы решения задач

Сели отдельный пользователь решает на машине математическую или логическую задачу, по ходу дела вмешиваясь в решение, говорят, что пользователь работает в "диалоговом", или "интерактивном", режиме.

Когда в одно и то же время одной и той же вычислительной машиной могут пользоваться сотни пользователей, то говорят, что все они работают в режиме разделения времени. Это еще один способ выполнять работы, связанный с использованием ЭВМ по типу I пли типу II. Отличие проявляется в использовании машины в целом, каждый отдельный пользователь может этих отличии и не замечать.

Разделение времени уменьшает "время ожидания решения", то есть время, проходящее от момента передачи задачи в вычислительный центр до получения результата решения. Тем самым доводится до максимума доля творчества человека в отношении: "процесс решения задачи"/"процесс созидания". В коммерческих приложениях типа III имеют дело с данными — файлами, записями, страховыми полисами, счетами, расписками, индексами клиентов. Огромные, многотомные файлы. Изнурительная работа по прочесыванию огромных файлов для нахождения *единственного* подходящего, выполнение простейших вычислений — иногда даже тривиальных,— и затем снова работа с файлами. Критическими характеристиками при этом являются скорость движения магнитной ленты, время доступа к диску, количество томов, размещаемых на диске, а также структура файла.

Пользователи типа IV занимаются именно вычислениями, а не файлами в смысле файлов со страховыми полисами, выданными. компанией миллионам различных людей. При этом обрабатываются большие количества данных, но это не файлы, а скорее массивы. Узким местом при этом являются скорость вычислительной машины и планирование процесса, необходимые для своевременного выполнения заданий.

Однако для обеспечения работы по типу IV сотен пользователей, подключаемых к ЭВМ в диалоговом, интерактивном, режиме, необходимо делать многое из того, что делается и для коммерческих приложений,— запоминать имена и местоположения пользователей, времена их работы и многие другие более простые вещи. В этом смысле между использованиями по типам III и IV есть некоторое перекрытие.

Я разделил их по тем причинам, что, во-первых, использование по типу IV создает целую индустрию, а вид использования

(разделение времени) и диалоговый режим существенно отличают этот тип от простой обработки данных и, во-вторых, для машин, используемых по типам III и IV, необходима весьма различная аппаратура, а значит, и разное программное обеспечение.

Использование типа V. Управление процессами

Если ЭВМ используется для выполнения или для помощи человеку в выполнении или слежении за некоторым процессом и мы не можем продолжать его выполнение без постоянного получения промежуточных результатов, мы используем машину по типу V. Процесс может быть простым, например управление огнями светофора, но может быть и весьма запутанным, как, например, система помощи авиадиспетчерам по предотвращению столкновений самолетов или система резервирования мест на самолетах, которая одновременно взаимодействует с тысячами служащих транспортных агентств.

Управление процессом неотделимо от выполнения процесса и обычно работает в режиме реального времени. Вычислительная машина должна успевать выполнять свою работу за время, отводимое для управления.

Для пользователей тип V выглядит так же, как и типы III и IV, но любые сбои ведут при этом к полному нарушению управляемого процесса.

Требования высокой помехозащитности и малого времени ответа приводят к необходимости разрабатывать один из самых сложных видов программного обеспечения. Программно-аппаратная система, выводящая ракету на перехват другой ракеты, построена в основном на тех же математических принципах, что и задачи расчета баллистических траекторий, необходимость решения которых привела к возникновению ЭВМ Однако теперь работы должны выполняться без каких-либо сбоев, в режиме реального времени, поскольку перехват должен выполняться именно в этот момент, когда вторая ракета подойдет к расчетной точке. И большая тяжесть работ по достижении поставленной цели падает на программное обеспечение

При использовании по типу III, когда работа людей не может быть выполнена без одновременной работы машины, мы переходим в область управления процессами, т.е. начинаем использовать машину по типу V.

Итак, тип V основывается на использованиях по типам III и IV.

Тип V появляется, когда комбинация задач типов IV и III приводит к значительным усилиям по ведению сложных *разработок*. Современная аппаратура достаточно легко справляется с задачами типа V! Узким местом таких систем является программное обеспечение.

Эволюция использования вычислительных машин шла непрерывно от типа I к типу V.

Тип I перешел в тип III. Было бы логично попытаться по запросу получить данные, находящиеся “внутри”, в ЭВМ. Каково среднее заполнение самолетов рейса Вашингтон — Париж в августе?

Использование типа III имеет значительное отличие от типа I. *Люди взаимодействуют* **с** вычислительной машиной. И опять-таки основная тяжесть этого ложится на программное обеспечение!

Пользователи типов I и II не имеют прямых контактов с машиной. Задания помещаются на перфокарты или магнитные ленты, и затем программист или ответственный по лаборатории ожидает, пока работа не будет выполнена, после чего они получают результаты, обычно отпечатанные на бумаге.

Пользователи типов III и IV непосредственно *взаимодействуют* с вычислительной машиной с помощью пишущей машинки или терминала с телевизионным экраном и клавиатурой. Они ведут диалог. Это совершенно не похоже на первые два случая, и для обеспечения взаимодействия пользователя с машиной нужно писать специальные программы (см. рис. 2.1).

Влияние пользователей на вычислительные машины и программное обеспечение

Пять видов использования по-разному влияют и на аппаратуру, и на программное обеспечение.

В больших системах типа III есть одно интересное, но очень “хитрое” место. Фактором, ограничивающим производительность аппаратуры, становится в них время доступа к диску. Большая часть действий в подобных системах связана с модификацией данных, поэтому необходимо постоянно осуществлять “поиски” этих данных. При этих “поисках” приходится простаивать в очередях, пока механизмы доступа к диску не дадут возможности осуществить обращение к нему. Программное обеспечение должно строиться так, чтобы число одновременных поисков было максимальным. Очевидно, что чем больше дисков в системе, тем большее перекрытие становится возможным. В некоторых реализациях имеются сотни отдельных дисков.

Работы по программированию системы “Скайлэб” велись в Хьюстоне как продолжение работ над проектом “Апполон XV”. Аппаратура осталась та же самая, что и для “Аполлона”; программное же обеспечение изменялось. Это было в некотором роде бедствие. Работы были закончены в положенный срок, но цена, которую пришлось заплатить группе примерно из 700 профессионалов, оказалась непомерной. Сверхурочная работа часто приводит к изнеможению и упадку сил.

Что же произошло? Почему столь опытная группа, имевшая за своими плечами 10 лет успешных разработок космических систем, т.е. всей системы Апполон и системы посадки на Луну, вдруг оказалась в таком положении? Ответ, кажущийся простым сейчас, в то время не был очевиден.

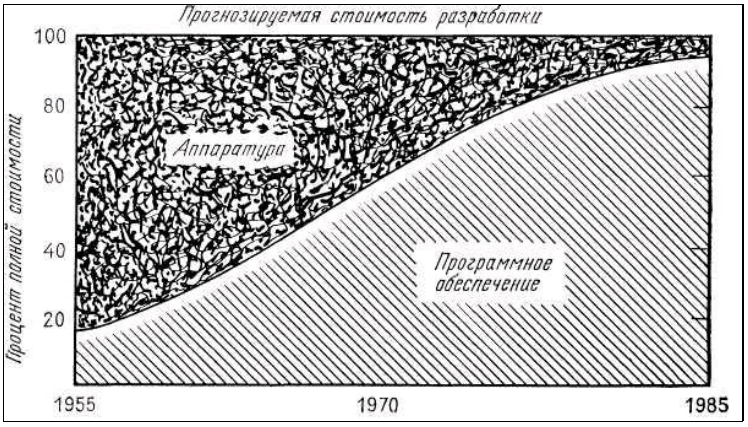


Рис. 2.2. Диаграмма, вносящая путаницу в вопрос процентного соотношения стоимости разработок.

Скайлэб была очень большая система типа III, которая постепенно превратилась в систему типа IV, и сбои в системе были исключены. Группа ожидала, что это будет система типа IV, не ожидая уклона в сторону системы типа V.

На борту космической лаборатории были проведены сотни экспериментов в реальном времени, и данные поступали от них тоже в реальном времени. Их надо было собирать, сортировать, запоминать, снабжать указателями, помечать и классифицировать — и все это в реальном времени. Ни NASA, ни фирма IBM не предвидели столь существенных отличий в применении систем, поэтому для выполнения работы пришлось создавать совершенно новое программное обеспечение.

Рис. 2.2 может быть использован как иллюстрация к большим системам типа V, а также к некоторым из систем типов III и IV. Он совершенно не отражает положение дел в системах типов I и II, хотя часто выдается за неоспоримую истину во многих книгах и докладах. Диаграмма, изображенная на нем, соответствует только той области, откуда она вышла, т.е. командным и управляющим вычислительным системам ВВС, как и было указано в докладе ВВС США.

Необходимость таксономии

Нужно постоянно помнить об одном часто не принимаемом во внимание факте, связанном с программным обеспечением. Оно слишком велико и разносторонне, чтобы его можно было обсуждать сколько-нибудь длительное время без предварительного указания того, какая область, часть или тин программного обеспечения обсуждается. Этот термин слишком широк, чтобы использовать его без определяющих прилагательных.

Разрабатывать программное обеспечение **с** каждым днем становится все труднее.

Разрабатывать программное обеспечение с каждым днем становится все легче.

Оба этих утверждения верны. Кажущееся их противоречие друг другу проистекает из широчайшего многообразия мира программного обеспечения. Давайте посмотрим, где же разработка программ становится проще.

Динамика роста мощности вычислительных машин

Очень упрощенные, но достаточно верные графики могут значительно прояснить складывающуюся ситуацию. Давайте бросим взгляд на вычислительный мир (рис. 2.3). В 1952 г. он состоял из *одной[1])* машины — UNIVAC I. *{Автор несомненно имеет в виду состояние дел в Соединенных Штатах Америки. К 1952 году в СССР тоже уже была построена первая отечественная ЭВМ. Руководил ее разработкой академик С.А.Лебедев. — Прим. перев.}*

В 1980 г. этот мир значительно расширился, вычислительные машины стали появляться как на нижних, так и на верхних частях шкалы (рис. 2.4).

После этого, в середине 1960-х гг. "миникомпьютеры" — термин относится скорее к цене, чем к размеру,— сместили положение вычислительных машин на шкале еще ниже (рис. 2.5).

В настоящее время благодаря использованию сверхбольших интегральных схем научились создавать микрокомпьютеры, вполне размещающиеся на ладони. Это снова заставляет нас перестраивать картину. Увеличивая масштаб (рис. 2.6), мы можем увидеть, что же произошло за эти годы в мире вычислительных машин — стоимость их уменьшилась, а емкость памяти и производительность увеличились.

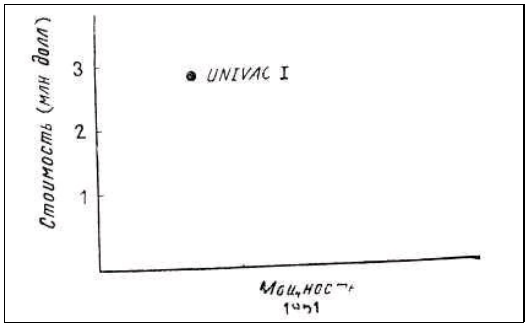


Рис. 2.3 Соотношение стоимость/мощность 1951 г.

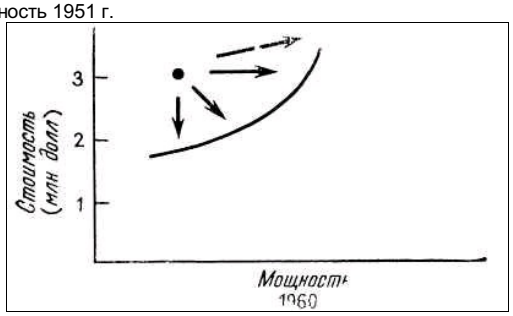


Рис. 2.4 Соотношение стоимость/мощность 1960 г.

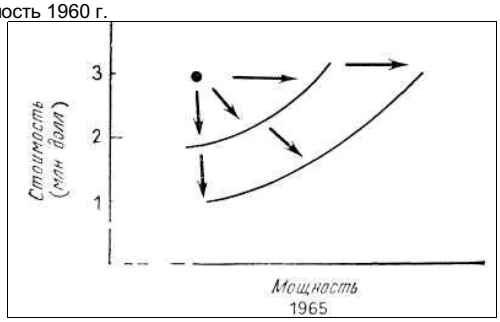


Рис. 2.5 Соотношение стоимость/мощность 1965 г.

Машины становятся все более мощными (точка *А).* Они становятся настолько дешевыми, что проникают в такие области (точка *В),* куда до сих пор вычислительные машины не допускались Программное обеспечение, конечно, следует за аппаратурой и в точках *А и В.* Оно маленькое и простое В, большое и сложное в *А.*

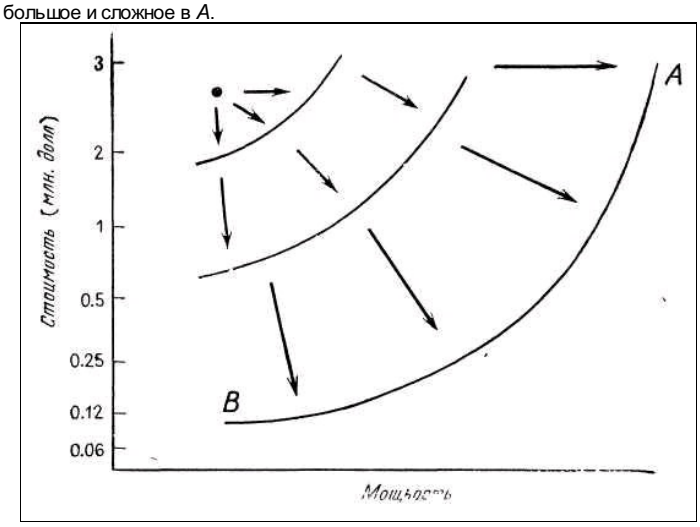


Рис. 2.6. Соотношение стоимость/мощность 1980 г.

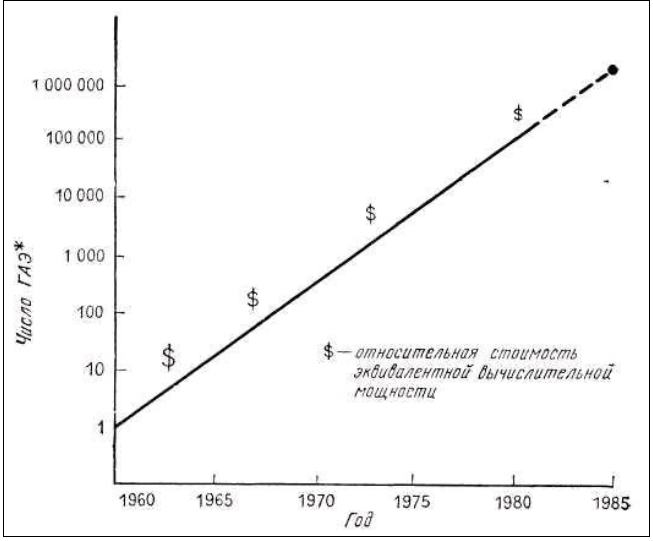


Рис. 2.7. Развитие полупроводниковой технологии

** ГАЭ — Группа Активных Элементов — примерно соответствует одному разряду или триггеру*

Цифровые схемы превратились в цифровые вычислительные машины (ЦВМ) — а это означает, что они стали программируемыми Этот великолепнейший прибор стоит теперь сущую безделицу В результате цифровые вычислительные машины проникли в такие области применения, которые ранее не могли бы оправдать таких сложных средств (см. рис 2.7)

Число элементов на кристалле *удваивается* каждый год, цена, надежность и мощность при этом остаются на прежнем уровне[1] Если бы автомобильная промышленность продвигалась вперед такими же темпами, то, по сведениям журнала "National Business Magazine", мы имели бы автомобили, которые весили бы около двухсот граммов, тратили бы около двух литров бензина на полтора миллиона километров и стоили бы примерно 2 доллара 70 центов

Возникновение технологии СБИС стирает грань между схемами и вычислительными машинами. Является ли кристалл размером 2,5 X 1,25 см с процессором и памятью в 128 слов (рис.2.8) вычислительной машиной? Конечно же, да. А ведь между тем, как следует обращаться с электрической схемой, с одной стороны, и вычислительной машиной — с другой, имеется большая разница. И этому есть причина для вычислительных машин требуются программы.

Доля программного обеспечения в стоимости изделия

Поскольку вычислительные машины становятся столь дешевыми, мы обнаруживаем их внедрение повсюду Они используются в игрушках, автомобилях, телевизорах, копировальных устройствах, ракетах, приборах, станках — всюду. И все эти использования подразумевают наличие программ

Если мы разрабатываем одну программу и выполняем ее на тысячах вычислительных машин, мы делим стоимость программы на число машин и получаем стоимость "одной" программы.

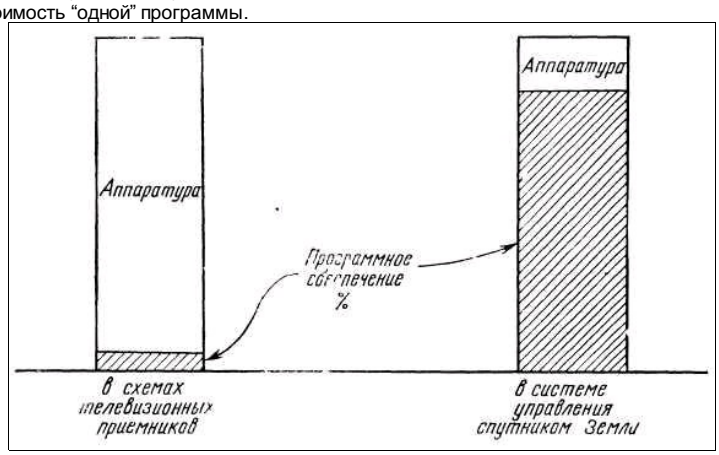


Рис. 2.9 Доля программного обеспечения в общей стоимости системы

Эта характеристика не отражает стоимости продукции, мы просто *тиражируем* программу

Таким образом, программа, заложенная в телевизионный приемник, имеет пренебрежимо малую стоимость — одна программа на 500 000 кристаллов, т. е. на 500 000 телевизоров! Программное обеспечение может составлять основную долю в стоимости системы управления спутниками, где одна вычислительная машина управляет и спутником, и линией связи (см. рис. 2.9).

ГЛАВА 3. ПОНЯТИЕ ПРОИЗВОДИТЕЛЬНОСТИ

Программное обеспечение и аппаратура

Цель программного обеспечения — заставить работать аппаратуру. Программы без аппаратуры невыполнимы, но и аппаратура без программ не работает. Ограниченность возможностей аппаратуры может удвоить затраты на разработку программного обеспечения.

Некоторые понятия и предварительные определения, связанные с аппаратурой и производительностью, могут помочь нам выделить нюансы отношений между более сложными понятиями, например между мультипроцессорностью и мульти-программностью, между сетями и распределенной обработкой. По мере развития вычислительной машины термин производительность становится более сложным. Часто для достижения максимальной производительности нашего двигателя, вычислительной машины, нам приходится привлекать программистов.

Однажды из-за того, что высшее руководство проекта не смогло понять, как оптимизировать вычислительную систему (аппаратуру и программное обеспечение), разработка большой системы зашла в тупик (стоимость оборудования около 40 млн. долларов). Были спутаны две различные меры производительности: скорость решения задачи и время ожидания решения.

Когда мы говорим о производительности вычислительных систем, то должны отдавать себе ясный отчет в том, что и как мы измеряем.

Это чрезвычайно важно. Проявлению превосходных качеств аппаратуры порой мешает плохое программное обеспечение, и наоборот. Неэффективные операционные системы (см. с.70—71) свели на нет работу многих великолепных высокопроизводительных машин. Задачей руководителей разработкой программного обеспечения является объединение аппаратуры и программ в одну эффективную систему.

Стремление к высокой производительности сильно усложняет разработку программного обеспечения. Если наша аппаратура недостаточно хорошо подходит для решения поставленной задачи, разработчикам программ придется предельно использовать все ее возможности, что приведет к значительному росту стоимости *разработки программного обеспечения*.

Различные способы измерения производительности вычислительных машин

"Что лучше, вычислительная машина А или вычислительная машина В? ...что быстрее, вычислительная машина А или

вычислительная машина В?". Это все равно что спросить: "Что лучше, модель X автомобиля Шевроле, или модель Y автомобиля форд?". Единственное, что можно на это ответить: "В каком отношении? В смысле элегантности? Стоимости? Производительности в смысле пробега в милях? В смысле удобства управления? Комфортабельности? Размера? Срока службы? Надежности?".

Автомобиль как система состоит из нескольких тысяч частей, каждая из которых выполняет определенную функцию. Автомобиль может быть сделан так, что некоторые из них будут идеально подходить для своей службы, но только за счет каких-то других. То же самое можно сказать и о вычислительных машинах.

Измерение внутренней скорости

Если мы начнем рассматривать внутреннюю скорость вычислительных машин, мы обнаружим, что время, необходимое для выполнения каждой команды, является одной из основных характеристик машины. Каждая команда выполняется в течение некоторого отрезка времени. *Умножение* длится гораздо дольше, чем *сложение.* Эти команды часто объединяются в "смеси", что дает возможность получить приближенную оценку внутренней скорости. Существуют различные смеси, обычно используются "научные смеси" и "коммерческие смеси".

Научная смесь	Коммерческая смесь
15 умножений	5 умножений
12 делений	2 деления
25 сложений	25 сложений
22 вычитания	18 вычитаний
12 записей в память	14 записей в память
2 ввода	8 вводов
2 команды печати	8 команд печати
8 условных переходов	14 условных переходов
2 безусловных перехода	6 безусловных переходов
100 команд	100 команд

Заметьте, что в моей научной смеси в три раза больше умножений и в четыре раза меньше команд ввода и печати.

Машина может иметь преимущество перед своим конкурентом по одной смеси, но не иметь преимущества по другой (см. рис. 3.1).

ТКС и МКС ТКС означает тысячу команд в секунду, МКС — это 1 млн. команд в секунду. Машина в два МКС может выполнять 2 млн. команд некоторой определенной смеси в секунду. На научных смесях коэффициенты МКС обычно оказываются более низкими, чем на коммерческих смесях.

Мы называем ТКС и МКС внутренними характеристиками, поскольку они вовсе не затрагивают возможностей ввода/вывода вычислительной машины, а также не учитывают эффектов, зависящих от объема памяти или размеров слова машины. Время выполнения команды имеет смысл измерять даже для самых параллельных машин.

	Научная смесь	Коммерческая смесь
Машина А	0.0028	0.0024
Машина В	0.0054	0.0018

Рис. 3.1. Пример разных по производительности машин — на научной и коммерческой смеси команд. Время в секундах

Когда в 1964 г. появилась серия машин IBM 360, модель 70 (она существовала короткое время, пока ее не сменила модель 75) была самой крупной (самой мощной) моделью: модель 30 была самой маленькой.

Фред Брукс, архитектор серии 360, в статье для журнала "IBM System Journal" писал:

- *Центральные процессоры различных моделей существенно разнятся по производительности. По отношению к самой маленькой модели (модель 30) внутренняя производительность самой большой (модель 70) составляет примерно 50:1 для научных расчетов и 15:1 для обработки данных в коммерческих приложениях.*

Обратите внимание на значительную разницу в сравнении при использовании коммерческих программ. Эти внутренние характеристики вычислительных машин не слишком точны, но все же широко используются как средство приблизительной оценки мощности ЭВМ.

Внешние характеристики

Понятиями более полезными, чем МКС, являются пропускная способность системы и время ожидания решения. Эти характеристики используются для оценки машины в целом, а не только ее возможностей в выполнении команд. Здесь становится существенным соотношение ввода и вывода с мощностью центрального процессора. Слишком медленный ввод/вывод поставит процессор на "голодный паек"; слишком медленный процессор заставит устройства ввода/вывода постоянно ожидать, когда можно будет заслать данные в отведенные им места.

Пропускная способность системы. Пропускная способность системы определяет то количество работы, которое может быть выполнено на машине за данное время.

Пусть, например, мы имеем 400 программ, написанных на Фортране. Мы пропускаем их на машине А, и у нас уходит на это 10 ч. На машине В то же самое заняло бы у нас всего 8 ч. Машина В на 20 % лучше машины А в смысле пропускной способности.

Пропускная способность системы зависит от многого.

1. Аппаратура. Конфигурация машины, мощность ЦП, размер и скорость работы памяти, количество каналов, магнитофонов, дисководов, система команд.

2. Программное обеспечение. Управляет ресурсами системы операционная система. Если она работает эффективно, пропускная способность системы должна быть хорошей. Неэффективная система может свести на нет все прекрасные качества даже великолепной машины.

Значительное влияние на пропускную способность оказывает размер памяти. Если память мала, то и память, и ЦП должны периодически отвлекаться от настоящей работы и перемещать данные между различными уровнями памяти (дисками и т.д.).

Без внесения каких-либо исправлений в другие части машины увеличение основной памяти уменьшает это перемещение и, следовательно, освобождает время для настоящей работы. Память всегда будет служить ключом к производительности машины. Фон Нейман установил это в своем меморандуме в 1946 г.; это верно и сейчас.

Предсказать пропускную способность очень трудно. Слишком много здесь неизвестных!

Эталонов для измерения пропускной способности не существует; можно измерить только относительную пропускную способность. Ее можно измерять только по отношению к конкретному набору задач. При отсутствии такого набора ее можно лишь приблизительно оценивать.

Пропускная способность характеризует всю систему в целом: ЦП, память, магнитофоны, программное обеспечение и операторов. Она является мерой количества работ, проходящих через вычислительную систему.

Сравнение двух характеристик: пропускной способности и МКС. МКС является мерой внутренней скорости вычислительной аппаратуры, т.е. ЦП и памяти. С помощью МКС определяется время, потребное на выполнение команд, в него не входит время работы магнитофонов, каналов, программного обеспечения.

Пропускная способность оценивает всю вычислительную систему и ее компоненты, а также все программное обеспечение, работающее на данной аппаратуре.

Профессионалы, работающие с вычислительными машинами, должны быть осторожны, чтобы не путать эти два понятия. Рассматривать МКС как время решения просто НЕВЕРНО, так как 16-разрядная машина в 1.2 МКС может обладать значительно более высокой пропускной способностью, чем 8-разрядная машина в 1.2 МКС, предполагая, конечно, что ввод/вывод и программное обеспечение остаются без изменения.

Время ожидания решения. Существует и другая характеристика вычислительных систем — время ожидания решения.

Временем ожидания решения называется время, которое проходит от того момента, когда программист отдает свою программу на счет, до момента получения им результатов этого счета. Чем меньше проходит времени, тем лучше. Время ожидания не следует смешивать с пропускной способностью.

Уменьшение времени ожидания, т.е. достижение желаемого результата, может привести к снижению пропускной способности, что нежелательно. Пакетная обработка увеличивает пропускную способность, но заодно увеличивает и время ожидания.

Уделяя внимание времени ожидания решения, мы тем самым заботимся об экономии времени инженера или программиста. Если же мы хотим снизить стоимость решения задач на данной машине, значит, нам нужно обратить особое внимание на пропускную способность.

"Отсутствие" ожидания достигается в большинстве систем разделением времени. Пользователь, сидящий за пультом устройства ввода/вывода, вводит приказы и данные. Вычислительная машина, обслуживающая сотни таких пользователей, достаточно быстро работает, чтобы при переключении с одного пользователя на другой им казалось, что в их распоряжении находится вся машина целиком. Машине приходится все время "изворачиваться": ввести данные для пользователя №10; обработать программу программиста №47; отпечатать для девятого, вывести данные для №197; №177 хочет продолжить свою работу, и т.д., и т.п. Эти увертки не являются настоящей работой и составляют только накладные расходы.

В системах без разделения времени время ожидания обычно измеряется часами. Сколько времени пройдет с тога момента, когда я "запущу" мою программу, до того момента, когда я получу мои результаты? Иногда это время может достигать 24 ч., если в машине оптимизируется *пропускная способность.*

Время ответа. Понятие времени ответа близко по смыслу понятию времени ожидания, но не эквивалентно ему. Это время, нужное вычислительной машине для ответа именно мне как пользователю терминального устройства, работающего в истинном масштабе времени. В этот момент я являюсь не программистом, а пользователем. Я использую машину для решения своей задачи. Время ответа обычно измеряется как время, проходящее от момента нажатия кнопки ввода до момента *начала* вывода на экран моего дисплея или на мое печатающее устройство.

Моей группой в отделении федеральных систем в IBM была успешно запрограммирована система диспетчерской службы нью-йоркской полиции (телефон вызова полиции 911), которая называлась "SPRINT". Мы просто измучились, так как корпорация не смогла отбиться от требования гарантировать время ответа не более 3 с. Нам надо было обслуживать 96 диспетчерских (телевизионных) пунктов, на которых находились полицейские. Модель IBM/50, которая нами использовалась, обслуживала такое множество экранов с большим трудом. Чтобы достичь этих трех секунд, нашим программистам приходилось творить чудеса. Из 2 млн. долларов, затраченных на всю разработку, на это ушло более 200 тыс. Разработчики программного обеспечения понимали, что предсказать время ответа для такой сложной системы заранее невозможно. Они отказались подписывать контракт, в котором были оговорены эти три секунды. Но руководство настояло на своем.

Такие "дополнительные пробежки" обычны для разработчиков программного обеспечения. Если системе не хватает ресурсов ЦП или памяти, разработчикам приходится работать с большей нагрузкой и намного дольше — и как следствие растет стоимость обеспечения. Им приходится отыскивать пути обхода слабых мест аппаратуры. Иногда приходится и переделывать программы.

Система команд и их влияние на производительность. Машина Тьюринга имела всего шесть команд. Она могла выполнять любые задания, но такой ограниченный набор используемых команд сильно затрудняет работу программиста. Одно из важных экономических решений, которое принимает каждый производитель электронных машин, это — сколько различных команд должна распознавать и выполнять машина, которую он строит. Множество распознаваемых вычислительной машиной команд будет служить ее "репертуаром" и называться системой команд (СК).

Современные большие машины распознают и выполняют более чем 300 разных команд. Овладеть таким "словарем" для программиста достаточно сложно. Изучение этих команд и способов их эффективного использования может потребовать от программиста месяцы, а то и годы. Только половина набора команд машины IBM 7090 использовалась регулярно.

Невозможно ответить на вопрос о том, на какой машине (большой, с богатым выбором команд, или маленькой, с ограниченным набором) легче программировать. Как и для многих других вопросов из области вычислительной техники, ответ "зависит" от множества деталей. Маленькую, простую задачу может оказаться легче программировать на маленькой машине с небольшим набором команд. Большие, сложные задачи, возможно, легче решать на машине с "богатым" словарем. К тому же все зависит от того, на машинах какого типа привыкли работать программисты. Однажды в Хьюстоне нам пришлось с помощью группы, имевшей огромный опыт работы на большой машине IBM 360 модели 75, разрабатывать обеспечение для маленькой IBM системы 7. Это было ужасно! Они не понимали ограничений этой машины, неправильно подходили к решению задачи, использовали неверные методы. Они ухлопали целый год — и чертову уйму денег,— прежде чем разобрались, в чем дело. А это были хорошие опытные разработчики.

Случается и наоборот. Перевод группы, привыкшей работать на ограниченных системах команд, на программирование с использованием "богатых" систем так же может привести к ужасающим результатам. Прежде всего, они не справляются с богатством, как и повсюду, где оно достается людям. Продуктивность их работы чрезвычайно низка.

Теоретически большие системы команд можно использовать более эффективно, это относится и к стадии разработки. Не надо только думать, что это всегда так.

Замеры времени. Наилучшим способом замерять производительность машины было пропустить на ней именно *Вашу* программу, определив время ее обработки прямо по секундомеру.

Такой способ труден и дорог. Некоторые еще не запрограммированные задачи приходится программировать специально для подобных замеров. Для замеров нужно использовать задачи, которые могут быть просчитаны на разных .машинах. Такие задачи называются *эталонными пакетами*.

Что же можно охарактеризовать с помощью эталонных пакетов и замеров времени? *Все,* что имеется в вычислительной системе — аппаратуру, отдельные программы (а следовательно, и умение тех, кто занимался их программированием), трансляторы, программное обеспечение, устройства ввода/вывода, размер машинного слова, систему команд, операционную систему, действия операторов.

Из того, что машина А выигрывает у машины В, нельзя делать вывод о том, что ее аппаратура работает быстрее. Плохое программное обеспечение могло свести на нет все хорошее, что имеется в машине В.

Балансировка системы

Чтобы вычислительная система работала эффективно, необходимо так ее сбалансировать, чтобы она подходила для решения поставленных перед ней задач, чтобы процессор был загружен в равной мере с устройствами ввода/вывода, не опережал их работу, но и не отставал от них. Кроме того, система должна включать в себя высококачественное программное обеспечение. В хорошо сбалансированной системе одновременно работают почти все устройства.

Что измеряется	
Пропускная способность характеризует систему в целом: команды + программное обеспечение + операторы + магнитофоны, а также искусство программистов и качество аппаратуры ЭВМ	МКС измеряет внутреннюю скорость центрального процессора и памяти
С помощью замеров времени	С помощью измерений на смесях команд
Как измеряется	

Рис. 3.2. Пропускная способность системы и МКС как средства оценки производительности вычислительной машины.

На рис. 3.2 наглядно подытоживаются наши рассуждения о производительности.

ГЛАВА 4. ТАКСОНОМИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Деление программного обеспечения на отдельные виды

Каким же образом мы справляемся со всем своеобразием вычислительной машины? Помогает нам распределение па категориям и классам, разделение предмета на составные части и куски. Именно это мы и собираемся проделать в этой главе с программным обеспечением.

Мы уже видели, что по типам использования все вычислительные машины могут быть разбиты на несколько категорий, мы насчитали пять таких категорий. Эти же пять категорий оказываются полезными и для понимания внешних влияний, испытываемых программным обеспечением. Как мы увидим, они могут не совпадать с обычным делением программного обеспечения на некоторые типы.

Здесь представлена таксономия программного обеспечения, которая может быть полезной для обсуждения и понимания проблем, возникающих при разработке, использовании и продолжающейся разработке программного обеспечения больших и сложных систем.

Как мы уже видели, есть пять типов использования систем с программным обеспечением:

Тип I. Использование для обработки данных.

Тип II. Использование для проведения научных расчетов.

Тип III. Использование для информационных систем.
Тип IV. Использование в диалоговых системах решения задач.
Тип V. Использование для управления процессами.

Таксономия программного обеспечения

За время своей жизни программное обеспечение проходит три фазы:
1. Фазу разработки.
2. Фазу использования
3. Фазу продолжающейся разработки (часто называемого сопровождением).

Существуют три типа программного обеспечения:
1. Прикладное обеспечение.
2. Инструментальное обеспечение.
3. Системное обеспечение.

Три отличительные черты применения программного обеспечения:
1. Масштабность программного обеспечения.
2. Сложность программного обеспечения
3. Ясность программного обеспечения.

Два класса программного обеспечения:
1. Программное обеспечение проекта.
2. Программное обеспечение как продукция.
а. Программное обеспечение как продукт.
б. Аппаратура с видоизменяемым, гибким программным обеспечением.
в. Аппаратура, сопровождаемая некоторым программным обеспечением.

Дочитав до этого места, некоторые могут воскликнуть: “Конечно, все это очевидно!”. Хорошо бы, чтобы это было так. Мне постоянно встречаются люди, совершенно не различающие все эти классы, типы и т.п. Различать же их полезно как при разговорах о программном обеспечении, так и при управлении его разработкой.

Теперь мы остановимся на каждом пункте нашей классификационной таблицы и постараемся кратко объяснить, что мы имели в виду под классификацией и чего не имели.

Жизненный цикл программы

Жизнь программ состоит из трех фаз:
1. Разработка.
2. Использование.
3. Продолжающаяся разработка (или сопровождение).

Самая трудная фаза — первая, разработка, поэтому на нее приходится тратить больше всего времени. Однако в больших программах стоимость продолжающейся разработки часто (но не всегда) превышает половину суммарных затрат на всю жизнь данной программы.

На фазе использования мы обычно бываем вознаграждены за потраченные усилия, и, если мы правильно провели разработку, использование программного обеспечения пройдет спокойно и гармонично Обратите внимание на то, что фаза продолжающейся разработки проходит параллельно фазе использования. Обратите внимание также и на то, что мы избегаем использовать термин “техническое обслуживание”. Для обозначения этой фазы в англоязычных странах в литературе часто используется неправильный термин “maintenance”, обозначающий “обслуживание (ремонт)”. Обслуживать программное обеспечение не нужно, поскольку никаких неполадок в нем быть не может, а техническое обслуживание и ремонт предполагают наличие неполадок и отказов. Правильно написанные команды не могут вдруг сами собой стать неправильными, как может порваться, отпаяться или замкнуться проводок. Команды не изнашиваются.

Необходимость продолжающейся разработки объясняется двумя причинами. Во-первых, в большой программе всегда имеется некоторое количество ошибок, которые не выявляются при тестировании. В гл.5, в разделе, посвященном тестированию, мы покажем, почему это происходит. Во-вторых, система должна развиваться. Вводится, скажем, новый налог, и приходится изменять программы составления платежных ведомостей.

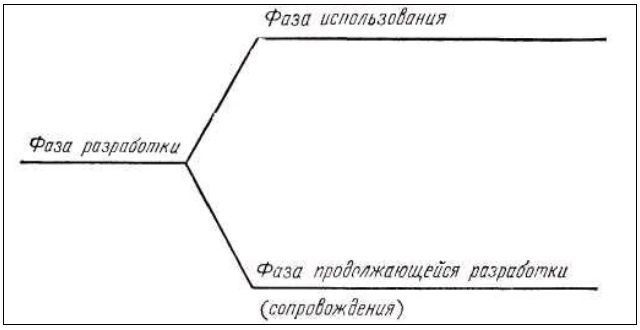


Рис. 4.1. Три фазы жизненного цикла программы.

В слове “обслуживание” есть что-то унизительное. Заниматься обслуживанием чего-то менее престижно, чем разработкой, и многие не чувствуют удовлетворения от этой работы. В то же время в деле программного обеспечения эта сторона требует к себе даже большего уважения, чем первоначальная разработка.

Разработчики, продолжающие следить за системой (а не ремонтировать ее), должны не только подключать новые функции и исправлять хитрые и коварные ошибки, которые ускользнули от внимания группы, начинавшей разработку. Они, кроме того, обычно вынуждены проводить подлинное расследование методов проектирования, которые редко бывают достаточно ясно документированы группой разработчиков.

Причиной специального выделения фазы использования является тот факт, что со многими большими системами постоянно взаимодействует множество людей. Система может быть легкой в использовании — “дружественной” пользователю — или трудной. Она может препятствовать людям совершать ошибки, т.е. быть “здравомыслящей”, а может и не делать этого. Эти качества не должны являться случайным следствием разработки. Они должны быть тщательно выверены при проектировании программы, хотя проявиться они могут только на фазе использования.

В гл.5 мы увидим, что каждой программе присущи 12 различных характеристик. Некоторые из них относятся к использованию, некоторые к разработке, другие же к продолжающейся разработке.

Казалось бы, применяемая нами для обозначения фазы использования жизненного цикла программы терминология не должна порождать никаких дополнительных проблем, но это не так. Некоторые люди называют эту фазу так, а другие иначе.

ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ФАЗЫ ИСПОЛЬЗОВАНИЯ	ОБЕСПЕЧЕНИЕ РАБОЧЕЙ ФАЗЫ ОПЕРАЦИОННОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ФАЗЫ ВЫПОЛНЕНИЯ

Рис. 4.2. Синонимы термина “программное обеспечение фазы использования”.

Некоторые называют эту фазу “операционной” или “эксплуатационной” и говорят об “эксплуатации программного обеспечения”. Обычными являются термины “фаза выполнения” пли “время выполнения”, так же как и термины “рабочая фаза” или “время работы” (рис. 4.2).

Все три фазы можно наблюдать на одной и той же вычислительной машине одновременно. В одной области памяти находится и

выполняется программа составления платежных ведомостей — это использование. В другой области памяти работает диалоговый транслятор — идет какая-то разработка. В третьей области группа сопровождения производит какую-нибудь “автоматическую перестройку”.

Все это только вносит путаницу! Чтобы добраться до истинного смысла, мы должны осознать, что рассматриваем жизненный цикл программы, а не вычислительной установки! В связи с этим для определения этой фазы совершенно неважно, что кроме нашей программы выполняется одновременно на той же машине.

Для аналогии рассмотрим вопрос разработки молотка, жизненный цикл которого показан на рис.4.3. В “жизни” молотка отсутствует фаза ремонта. Если он ломается, мы просто берем новый. Отсюда следует, что при изготовлении молотка мы должны учитывать только легкость и простоту использования.

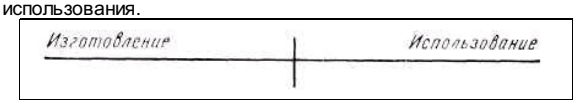


Рис. 4.3. Жизненный цикл молотка.

Если же речь заходит о велосипеде, мы сталкиваемся с тем, что в жизненном цикле возникает фаза сопровождения, или “регулировки”. Из-за износа нам придется тратить усилия на ремонт, чтобы велосипед снова мог ездить(см. рис. 4.4).

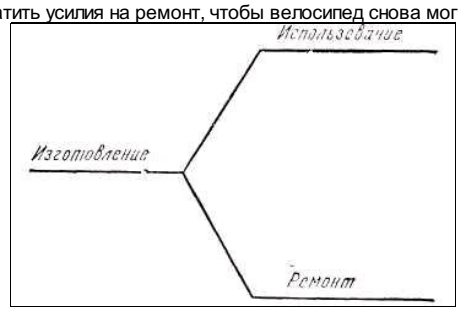


Рис. 4.4. Жизненный цикл велосипеда.

При рассмотрении жизненного цикла административного здания мы сталкиваемся с тем, что нам приходится не только заниматься ремонтом при каких-то неисправностях, но также и капитальной *перестройкой,* если организация, занимающая это здание, начинает расти или сокращаться и т.п.

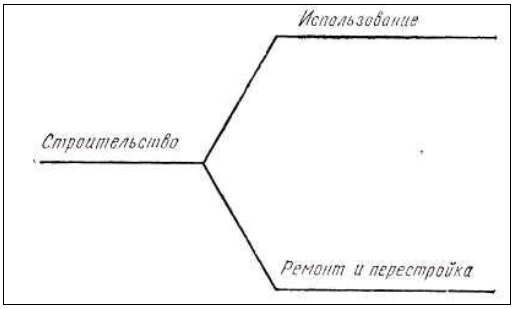


Рис. 4.5. Жизненный цикл административного здания.

Мы должны ломать стены, строить новые, переделывать отопительную систему, менять энергосеть и т.д. (см. рис. 4.5).

В связи с этим, если мы предполагаем возможность каких-либо изменений в здании в будущем, то при исходном проектировании здания (и составлении проектных документов) должны позаботиться о том, чтобы текущий и капитальный ремонт проходил без затруднений.

Обратите внимание на то, что при переходе к программному обеспечению (рис. 4.6) мы меняем слова, стоящие на правой нижней линии схемы.

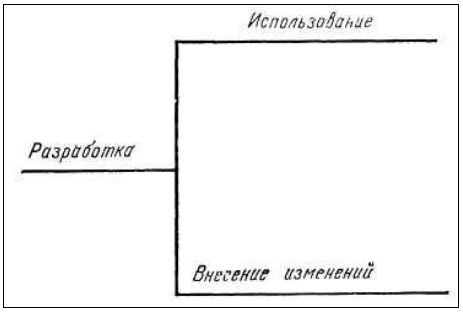


Рис. 4.6. Жизненный цикл программы

Вместо слова “ремонт” стоят слова “внесение изменений”. Слово “строительство” также заменяется словом “разработка”.

Рис. 4.7. Жизненный цикл программного обеспечения.

Однако в большей степени для программного обеспечения, особенно для больших систем, подходит схема на рис. 4.7. В исходном программном обеспечении имеются ошибки, которые прошли сквозь фазу тестирования и могут быть обнаружены только после того, как заказчик начнет пользоваться сделанной для него системой. Таким образом, теперь усилия будут тратиться на продолжение разработки, которая, однажды начавшись, практически никогда не заканчивается.

Рис. 4.8 показывает принципиальное отличие жизненного цикла аппаратуры от жизненного цикла программного обеспечения.

На этом рисунке показано, что в процессе разработки аппаратуры есть такие фазы, как фаза производства, фазы повышения технологичности и ремонтопригодности.

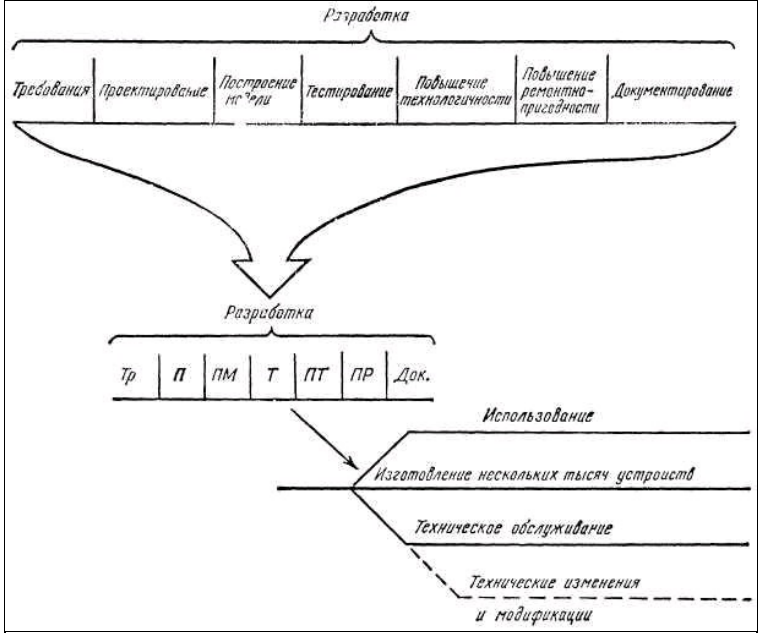


Рис 4.8 Жизненный цикл аппаратуры.

Простое перечисление этапов разработки аппаратуры, особенно указание на необходимость затрат на повышение технологичности и ремонтопригодности, говорят нам о многом. Инженеры специально анализируют прибор, который им необходимо создать, и старательно разрабатывают конвейер, так что заводские затраты в расчете на одно устройство будут минимальны. При этом всегда получается так, что серийный образец существенно отличается от опытного, построенного при разработке. Программное обеспечение не запускается в производство, и при его разработке фаза повышения технологичности отсутствует.

Рис.4.9 Одна группа руководит работами на всех стадиях жизненного цикла

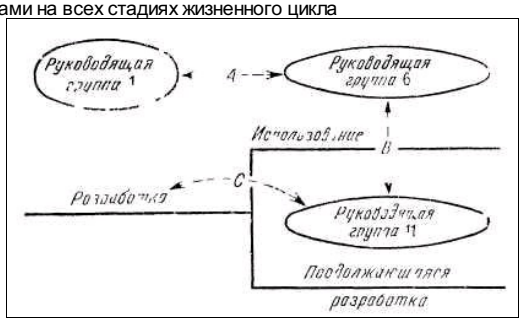


Рис. 4.10 Три руководящие группы; один жизненным цикл.

Другой функцией при разработке аппаратуры является повышение ремонтопригодности. Здесь инженеры вносят в прибор такие изменения, которые облегчат его текущий и капитальный ремонт в условиях использования. Их деятельность направлена на то, чтобы минимизировать требования к людям, производящим ремонт, к запасным частям, процедурам наладки и проверки. Процесс разработки обеспечения также включает в себя задачу снижения затрат на сопровождение, причем этой стороне должно уделять значительное внимание. Очень часто дата сдачи программы настолько связывает разработчиков, что все их усилия направляются только на то, чтобы уложиться в отведенные сроки, и никто не предпринимает никаких усилий дня облегчения последующей фазы сопровождения. Как мы увидим, правильно проводимая разработка должна включать в себя работы по обеспечению легкости сопровождения.

Другая причина того, что работы по облегчению будущего продолжения разработки зачастую не проводятся, заключается в том, что руководство различными фазами обычно проводится разными людьми. Рис. 4.9 показывает, что иногда руководство всеми тремя фазами находится в одних руках. Но это исключения.

Чаще же действует схема, изображенная на рис. 4.10. Имеются три ведущие группы, по каждой фазе своя. Руководители разработки мало заботятся о затратах на сопровождение и проблемах, возникающих па этой фазе.

При разном руководстве создаются условия для возникновения ошибок. Стрелки *A* на рис. 4.10 направлены в обе стороны, поскольку пользователи должны иметь возможность передавать свои требования разработчикам. Часто они лишены такой возможности.

Разработчики должны сформулировать, что именно может быть использовано в фазе использования и каким образом. Однако во многих случаях этого нельзя достичь без подробных инструкций для пользователей.

Все это относится и к стрелке С, с той лишь дополнительной проблемой, что очень часто группа сопровождения даже не существует в тот момент, когда разработка уже идет полным ходом. Слишком часто все озабочены только своими проблемами.

Смысл стрелки *B* вполне понятен и ясен, но она обычно игнорируется до тех пор, пока не случится какой-нибудь казус!

Патологические жизненные циклы программного обеспечения

Нарисуем простейшую схему жизненного цикла программного обеспечения двумя разными способами, что даст нам возможность выделить основные моменты.

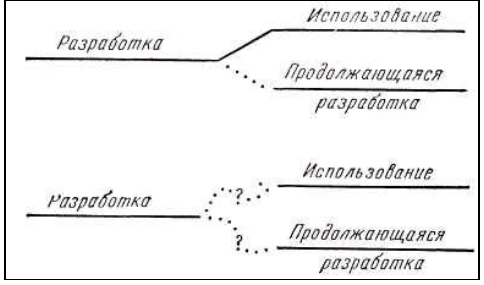


Рис. 4.11. Патологический жизненный цикл программного обеспечения, пример 1 (вверху). Рис. 4.12. Патологический жизненный цикл программного обеспечения, пример 2 (внизу).

Первая схема (рис. 4.11) иллюстрирует разрыв между разработкой и продолженной позднее разработкой. Во многих случаях за эти этапы отвечают совершенно разные организации. Заключившая контракт группа в Новой Англии разрабатывает, а где-то на юге другая группа ведет сопровождение. Пунктирная линия между разработкой и продолжающейся разработкой показывает, что этот переход не гладкий и не простой. В действительности во многих случаях мы можем нарисовать такую схему, как на рис. 4.12.

Между разработчиками и группами сопровождения часта существуют временные и географические расстояния, различия в применяемой методологии, организации, таланте и штатном расписании. Очень часто также между пользованием программами и окончанием разработки есть разрыв во времени.

Эти схемы соответствуют одноразовым разработкам, которые встречаются в случае программных обеспечений типов I и II.

Неразрывная связь разработки и продолжающейся разработки

Мы можем представить себе и такую схему, какая показана на рис. 4.13. В ней между разработчиками и группой сопровождения имеется очень крепкая связь, более того, этим может заниматься одна и та же группа.

Рис. 4.13. Жизненный цикл программного обеспечения, в котором разработчик сам сопровождает свои программы.

Это в особенности относится к программам типа V, которые разрабатываются постепенно и имеют несколько версий, каждая из которых вступает в действие в свой срок.

Имеются ли различия в задачах разработки и продолжающейся разработки? Безусловно, но они не имеют ничего общего с тем, как их многие себе представляют. Мой коллега Энди Ферентино (который привлек мое внимание к рис. 4.13) был свидетелем выступления кандидата на соискание докторской степени по программированию, темой диссертации которого было различие между разработкой и “продолжающейся разработкой” (в терминологии автора “maintenance”). Энди указал на то, что окружение, в котором создается программное обеспечение, должно быть абсолютно одинаковым на обеих фазах. Эти два рода деятельности лишь очень немногим отличаются один от другого. Давайте проследим шаг за шагом.

Разработка программного обеспечения, продлевающая все фазы жизненного цикла

Если составить диаграмму, отражающую сравнительные усилия в ходе разработки программного обеспечения, мы увидим, что затраты на определение требований и проектирование превышают затраты на использование, а затраты на сопровождение редко бывают значительными.

Таблица 4.1. Различия между разработкой и продолжающейся разработкой

Разработка	Продолжающаяся разработка
1. Определение требований для системы типа V крайне затруднено, так как до этого таких систем еще не было (Для систем типов I и II работа по определению требований практически одинакова)	С системами типа V работать легче, так как пользователь к этой фазе уже лучше знает, что ему нужно
2. Проектирование. Большие возможности. Выбираются лучшие варианты. Начинают сверху	Проще, поскольку система существует, многое из того, что спроектировано на верхних уровнях, сделано, и в большей мере определяет то, что надо делать на более низких уровнях. Труднее, если документация плохая. Напоминает археологические раскопки
3. Программирование Такое же	Такое же
4. Компоновка Такая же	Такая же
5. Тестирование Такое же	Такое же
6. Документирование Такое же	Такое же

Это проиллюстрировано на рис. 4.14, где размер букв пропорционален затратам на соответствующие фонды.

Рис. 4.14. Затраты на разработку, распределение по фазам жизненного цикла.

Такое распределение затрат абсолютно неверно. Оно, однако, повсеместно распространено, потому что разработчики программного обеспечения постоянно убеждают сами себя в том, что обеспечение должно создаваться по методу “большого взрыва”, единым усилием. Это заблуждение имеет свои причины, мы еще будем в этом разбираться.

В большинстве проектов обычно получается так, что если реализуется от половины до трех четвертей обещанных функций, то проект объявляется успешно завершенным. Продукция отправлена потребителю, спутник запущен. Все упущения будут восполняться позднее, под маркой сопровождения. Для повышения технологичности сопровождения и облегчения продолжающейся разработки делается очень мало или совсем ничего. Никаких усилий не прилагается к тому, чтобы сделать систему более подходящей для пользователя.

Фаза использования

Хотя фаза использования следует в жизни программы после фазы разработки, именно она является, или по крайней мере должна являться, *определяющей.* Ее характеристики в большой степени определяют две другие фазы. Для начала перечислим некоторые из характеристик использования, которые могут уточнить стратегию разработки.

Периодичность использования. Я могу разработать программу, которая выполняется каждый день, или раз в неделю, или раз в месяц, или раз в год. Я также могу разработать программу, которая будет работать постоянно, все время. Я могу разработать программу, которая будет исполнена только один раз, а затем выброшена. Эффективность использования аппаратуры (например, памяти) крайне важна, если программа работает постоянно, и совершенно не имеет никакого значения для программы, запускаемой однократно или только один раз в год.

Количество пользователей. Я могу разработать программу, которая будет использоваться только моей фирмой и нигде больше. Я также могу разработать программу для использования на 500 предприятиях моей отрасли. Я могу разработать программу для тысячи различных небольших фирм, каждая из которых может, а возможно, и будет использовать ее немного по-своему. И снова особенности и характеристики, на которых я настаиваю в моей программе, сильно различаются при этих различных видах использования. Пять сотен разных пользователей влияют на программу совершенно иначе, чем 500 “одинаковых” пользователей (все пользователи из

моего учреждения используют систему одинаково).

Тип использования. Существуют диалоговые и автономные системы использования вычислительных машин. Попробуем провести различия между "пользователем" и "заказчиком". Пользователь — это человек, который сидит у терминала и ведет диалог с вычислительной машиной. Пользователь управляет ею, он "приводит" ее в действие. Служащий трансагентства, занимающийся резервированием авиационных билетов, является пользователем. Заказчик же — это главный бухгалтер фирмы. Ему не нужно вести никакого диалога с вычислительной системой, когда она производит расчет зарплаты для сотрудников фирмы.

Если мы готовимся к диалоговому использованию системы, мы должны так писать программное обеспечение, чтобы пользователю было легко взаимодействовать с машиной. Это гораздо больше, чем просто справочная система; такая система должна быть настолько удобной для работы, чтобы человек, работающий с машиной, охотно ею пользовался и работал с полной отдачей сил. Программы должны быть написаны так, чтобы система подсказывала пользователю, что сделать, если полученные ею инструкции не очень ясны.

Последствия отказов при разных типах использования.

В некоторых системах многочасовой отказ вычислительной машины хотя и неприятен, но все же приемлем. В других же отказ вычислительной машины всего лишь на несколько минут чреват катастрофой. В последнем случае нам приходится ставить дополнительную аппаратуру. Но дополнительная аппаратура это еще полбеды. Нам приходится еще писать программы для сохранения критических данных и передачи их с одного процессора на другой. Часто это должно делаться не за минуты, а всего лишь за секунды. Это весьма сложное программное обеспечение. Программа, управляющая переключением светофоров в городе, не представляет собой ничего сложного; однако если эта программа должна всегда работать правильно, то сложность программного обеспечения возрастает по крайней мере втрое.

Диапазон сложного программного обеспечения простирается от научного центра (годовой) стоимостью 50 млн. долларов, обеспечивающий диалоговую работу 460 ученых, где 90% всех программ выполняется один-единственный раз, до управляющей системы, в которой программное обеспечение работает по 365 дней в году, по 24 ч. в сутки, не имеет права на малейший отказ и изменяется крайне редко — один раз в год.

Фаза разработки

Фаза разработки, входящая в жизненный цикл программного обеспечения, может быть разбита на шесть отдельных этапов:

— Определение требований
— Проектирование
— Написание команд
— Компоновка
— Тестирование или верификация
— Документирование

Поскольку разработка на обозримое будущее представляет собой основную проблему в области программного обеспечения, оставшаяся часть книги в основном будет посвящена рассмотрению этих шести этапов.

Фаза продолжающейся разработки

Как мы уже видели, эту фазу часто называют обслуживанием. Это наиболее часто игнорируемый элемент жизненного цикла, оставляемый на попечение какой-нибудь новой, часто неизвестной заранее группы. Наша ключевая идея состоит в том, что эта часть цикла должна приниматься во внимание с самого начала работ по разработке.

Задачи продолжающейся разработки. Продолжающейся разработкой занимается группа сопровождения. Ее задачи таковы:

1. Включение новых функций. В уже существующие про граммы добавляются новые функции. Например, если служащие фирмы вступили в профсоюз, мы должны начать удержания членских взносов и внести дополнение в про грамму составления платежных ведомостей.

2. Модификация функций. Существующие функции расширяются или видоизменяются. Например, принимается закон, изменяющий государственные налоги, и в ту часть программы, которая правильно работала для старого закона, надо вносить новые параметры, учитывающие новые налоговые положения. Случается также, что при первой сдаче программного обеспечения мы не успели выполнить все требования пользователей; нам приходится добавлять эти функции, выпуская новые модификации, или "версии", системы.

3. Модификация оборудования. В систему включается новая аппаратура. Например, ставятся новые терминалы с большим разрешением. Функции при этом не изменяются, но для управления новыми дисплеями нам нужны новые про граммы.

4. Исправление ошибок. Пользователь обнаруживает в программе "ошибки", и их нужно исправлять. Например, в случае когда страховые вычеты из зарплаты прекращаются в тот же день, когда начинаются выплаты за лечение в больнице, обе эти суммы оказываются напечатанными в обеих графах корешка платежной ведомости.

Усилия при продолжающейся разработке затрачиваются на:

1. Исправление программ, чтобы неправильно реализованные функции работали теперь правильно.

2. Модификацию или создание нового программного обеспечения для добавления функций, необходимость которых была заранее известна, но которые были отложены при разработке.

3. Модификацию или создание нового программного обеспечения для добавления функций, соответствующих новым требованиям, не отраженным в исходной документации.

Различие между пунктами 2 и 3 едва уловимо, но весьма важно. В пункте 2 говорится о том, что группа сопровождения должна всего лишь соблюдать первоначально документированные требования. В пункте 3 от группы сопровождения требуется фактическое определение новых требований. Это существенно отличается от простого следования требованиям. Для выполнения этой функции продолжающейся разработки требуются и другие люди, и другая их квалификация, и другая организация их труда, чем для выполнения других функций.

Три типа программного обеспечения

Все программное обеспечение может быть разделено на три всеохватывающих типа:

1. Прикладное обеспечение.
2. Системное обеспечение.
3. Инструментальное обеспечение.

Первые два типа обеспечения работают в период использования, а третье, инструментальное, используется в фазе разработки. В фазе разработки может также использоваться и системное обеспечение.

1. **Прикладное программное обеспечение.** Программы, фактически выполняющие поставленную перед ними задачу, на пример печать платежных ведомостей, инвентаризацию, коммутацию сообщений, резервирование билетов, прокладку маршрутов.

2. **Системное программное обеспечение.** Программы, которые выполняются в фазе использования наряду с прикладными программами. Системное обеспечение управляет ресурсами вычислительной машины, т.е. дисками, оперативной па мятью, лентами, центральным процессором. Программное обеспечение, известное под названием операционной системы, также попадает в эту категорию, сюда же попадают и системы управления базами данных (СУБД). И то и другое будет несколько позднее рассмотрено нами в этом же раз деле.

3. **Инструментальное программное обеспечение.** Программы, которые помогают программистам и администрации создавать программное обеспечение фазы использования. Наиболее известными представителями программ этой категории являются ассемблеры и трансляторы.

Чтобы вычислительная машина выполнила вашу работу, вам необходимо создать *прикладное программное обеспечение*.

Чтобы вычислительная машина эффективно справлялась со многими приложениями и была хорошо приспособлена к окружению, необходимо создать *системное программное обеспечение*.

Чтобы легче было разрабатывать программное обеспечение, необходимо использовать *инструментальное программное обеспечение*.

Прикладное программное обеспечение

Прикладные программы являются наиболее видимой частью программного обеспечения. Платежные ведомости, инвентарные списки, проектирование мостов, управление ракетами, расчет напряжений, вычисление траекторий, предсказание погоды, бухгалтерский учет — все это лишь несколько примеров из тысяч прикладных программ.

Этот тип обеспечения характерен тем, что: 1) его легче всего разрабатывать и 2) в этой области работает подавляющее большинство разработчиков программного обеспечения.

Большая часть прикладных программ создается служащими тех организаций, которые и будут его затем использовать. Некоторая часть прикладного обеспечения создается специальными организациями либо по контракту с конкретным пользователем, либо как продукция, предназначенная для свободной продажи пользователям.

Прикладные программы обычно составляются людьми, хорошо разбирающимися в процессах, которые они автоматизируют. Программу расчета зарплаты, например, часто составляют сотрудники бухгалтерии. По мере того как все сильнее ощущается недостаток программистов, все большее использование в различных организациях приобретают “стандартные” прикладные пакеты. Стандартный пакет — это программа, написанная таким образом, что она может применяться более чем одним пользователем. Конечно, некоторые ограничения на возможности применения пакетов существуют, но в основном эти ограничения вполне приемлемы. Эта область программирования — пакеты или программная продукция — является наиболее быстро развивающейся отраслью индустрии программирования.

Отрасли в которых применяются прикладные программы

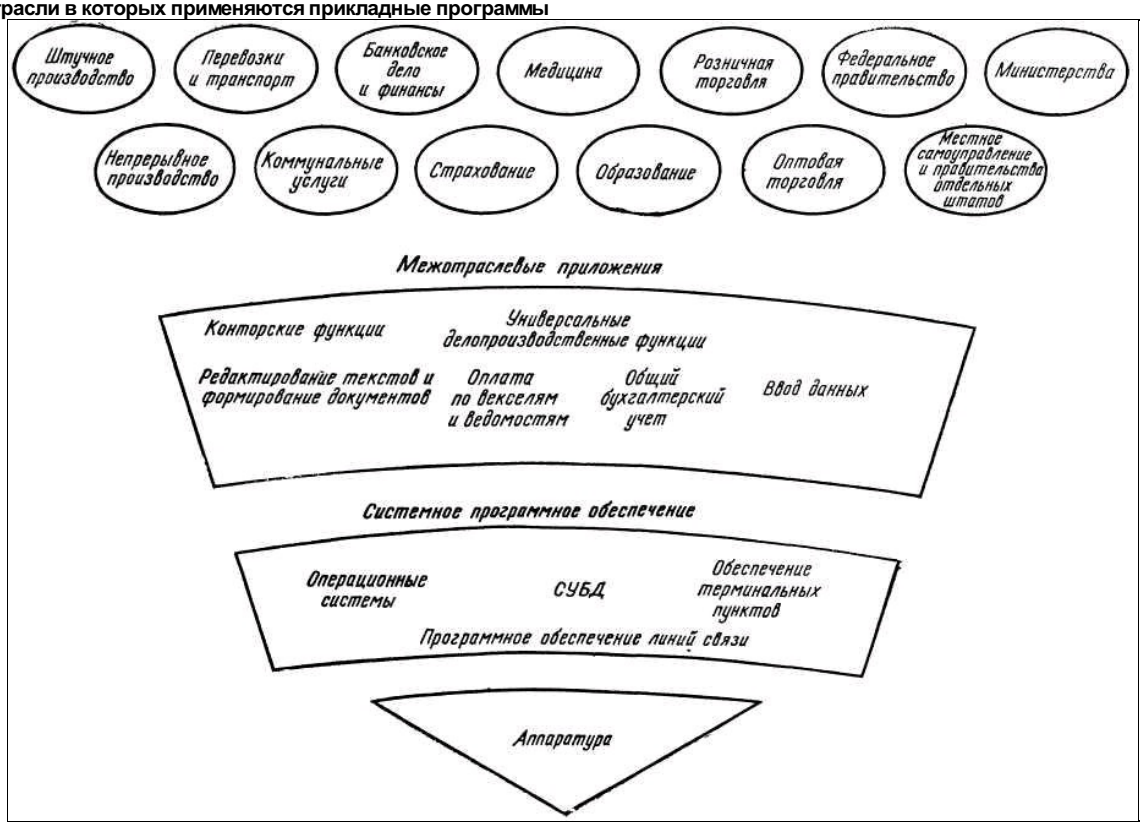


Рис. 4.15. Различные области применения прикладного программного обеспечения.

Может случиться так, что создание прикладного обеспечения станет своего рода индустрией. На рис. 4.15 показан диапазон распространения программного обеспечения. Все, кто будет пользоваться какой-либо аппаратурой, будут использовать и системное обеспечение. А наряду с аппаратурой и системными программами будут использоваться некоторые части прикладных пакетов программ, разработанных. либо непосредственно пользователем, либо группой сопровождения или закупленных в качестве стандартного обеспечения.

Системное программное обеспечение

Системное программное обеспечение используется для управления вычислительной машиной во время выполнения или разработки других программ. Использование вычислительной машины для управления ею самой и ее окружением — это логическая работа, а не работа с числами или символами. Этот тип использования вносит дополнительную сложность в программное обеспечение. Далее в этой главе мы еще обратимся к этому вопросу.

Разрабатывать системные программы труднее других, к тому же их особенно трудно представить себе во всех деталях и понять принципы их действия. Системные программы значительно сложнее прикладных и сложнее инструментальных программ.

Системное программное обеспечение служит для следующих целей.

1. Динамическое распределение устройств вычислительной машины. С каким устройством связано поступившее задание? Когда это устройство будет использовано? Каков порядок или приоритет работ? Подобные решения принимает и тем самым управляет работой машины на фазе использования большой и сложный на бор программ, называемый операционной системой.

2. Выполнение требований к окружению программ на фазе их использования. Если наша работающая система не имеет права “выключаться” более чем на 30 с, мы не можем поручить человеку привести систему в порядок — он просто не сможет так быстро отреагировать. Вычислительная машина, однако, сможет, поэтому мы пишем программы, которые будут следить за всеми устройствами системы и правильностью работы. Если они работают неверно, операционная система изменит конфигурацию аппаратуры (у нее в запасе есть дополнительные устройства) и работа будет продолжена, хотя некоторые из устройств и отключатся.

Операционная система. *Операционная система* представляет собой большой набор программ. Это наиболее распространенная форма системного программного обеспечения.

Операционные системы применяются теперь почти повсюду. Размеры денежных вложений в создание и модификацию операционных систем просто ошеломляют. Фирма IBM вложила в свою операционную систему около 3 млрд. долларов, а конца расходам еще не видно.

Однако лишь немногие представляют себе всю необъятность функций, выполняемых этими системными программами, все великое множество программ, которое они предоставляют пользователям для их работы, а также то, насколько это облегчает пользователю работу с ЭВМ.

За годы, прошедшие после своего возникновения, операционные системы превратились из относительно простого в невероятно сложное программное обеспечение, которое в настоящее время необходимо и программистам, и обслуживающему персоналу, и операторам ЭВМ. Современная операционная система:

1. “Управляет” работой аппаратуры.

а. Операционная система реагирует на все отказы, регистрирует их, распределяет работу, управляет процедурами восстановления и возобновления работ. Она обрабатывает прерывания, идущие от других машин, часов, операторов и т.д. (до появления операционных систем всем этим занимались операторы ЭВМ).

б. Операционная система составляет расписание работ на машине, “решая”, что можно начать выполнять не которую работу, так как сейчас доступны все необходимые для нее магнитные ленты, диски, свободна оперативная память, печатающие устройства или какие-нибудь иные машинные ресурсы. Она ведет списки используемых устройств и поступающих заданий. Она составляет расписание! В этой роли она управляет работой трансляторов (компиляторов) и ввода/вывода. (Раньше эту работу выполнял обслуживающий персонал машинного зала.)

в. Она приписывает поступающим заданиям приоритеты. (Это делалось раньше обслуживающим персоналом.)

2. Помогает выполнять функции, необходимые для работы прикладных программ.

а. В ней есть программы сортировок, печати и загрузки, и программистам уже нет необходимости создавать собственные их

версии. Ранее эти функции выполнялись программами, написанными прикладными программистами. Иногда эти функции считаются принадлежностью операционных систем, а иногда нет.

б. Она связывает между собой программы, тем самым множество различных частей программ может быть даже написанных разными программистами, будут работать как одно целое. (Ранее выполнялось программа ми прикладных программистов.)

3. Управляет хранением данных и их восстановлением, что совершенно необходимо для функционирования прикладных программ (так называемое Управление Данными)

а. Прикладной программист пишет команды запроса данных у операционной системы. Эти данные могут быть идентифицированы каким-либо специальным образом или вообще как-нибудь абстрактно, но в любом случае детали физического хранения не указываются. Операционная система вставляет в это места другие команды, которые приводят к запоминанию, отыскиванию, замене данных и т. д. (Ранее все это писалось прикладными программистами.) Этим достигается высокая сохранность данных и независимость программ от конкретных физических устройств. Пользователи и изготовители аппаратуры получают возможность, не внося никаких изменений в какие-либо прикладные программы, создавать новые, более совершенные, более дешевые устройства хранения файла. (Иногда эта область программного обеспечения рассматривается отдельно от операционных систем. Но она всегда является принадлежностью системного программирования.)

4. Управляет связью (посредством, например, телефонных линий) между программами, работающими на разных вычислительных машинах.

а. Обрабатывает сообщения, идущие от вычислительной машины и поступающие в нее извне, используя стандартные коммуникационные линии и сети. (Ранее писалось прикладными программистами.)

5. Управляет взаимодействием с пользователем (при помощи терминалов или телевизионных экранов).

а. Операционная система содержит программы, позволяющие пользователям работать с вычислительной машиной в диалоговом режиме с помощью стандартного дисплейного оборудования. (Ранее такие программы писали прикладные программисты.)

6. Защищает систему.

а. Она защищает свои собственные программы от “порчи” новыми, неотлаженными программами, впервые введенными в систему. (Ранее такой защиты не создавали.)

б. Операционная система выполняет восстановление функции, осуществляет дублирование, переключение, диагностическое и другое тестирование. (Ранее проделывалось вручную с помощью групп поддержки — т.е. крайне медленно.)

Операционные системы прошли длительный путь развития. В 1966 г. в журнале “IBM System Journal” была опубликована статья Мили под названием “Функциональная структура Операционной системы ОС/360” {[1] G.H.Mealy. Functional Structure of the OS/360 Operating System}. Мили отметил, что “идея операционных систем восходит по крайней мере к 1953 г., когда состоялась летняя школа по вычислительным машинам и пользовательским системам”. Перед операционными системами “тогда, как и сейчас... ставили цель добиться безостановочного выполнения сразу нескольких задач и организовать библиотеку стандартных программ”.

(Свое название операционные системы получили за то, что первоначально они помогали *операторам* поддерживать безостановочную работу машин, выполняя функции “восстановления”, проводившиеся раньше самими операторами.)

Автор статьи утверждает, что основной задачей разработки ОС/360 было получение системы “одинаково пригодной как для пакетной обработки, так и для применений в реальном времени”. (Эта вторая цель так и не была достигнута.) Были и вторичные цели:

— Повысить скорость решения задач
— Уменьшить время ответа
— Повысить производительность программиста
— Адаптируемость к новым условиям
— Расширяемость

Достижение всех этих целей, за исключением первой, помогает программистам. Что же касается производительности, то “ОС должна обеспечить качественно новый уровень гибкости путем предоставления программистам относительно большого набора входных языков”. Ставилась также и цель по достижению независимости от внешних устройств; новая аппаратура подключается автоматически без дополнительных усилий со стороны прикладных программистов! Среди многих других функций, выполняемых ОС/360 для программистов,— связывание частей больших программ, сортировки, работы по вводу/выводу. Для управления хранением и доступом к данным в операционную систему введено восемь различных вариантов программ. Теперь программисту не нужно писать самому подобные программы, в его распоряжении имеется много способов, чтобы указывать, как это должна делать операционная система.

Системы управления базами данных (СУБД). Относительно систем управления базами данных существует большая путаница. Эти системы настолько мощны и выполняют столь широкий диапазон функций, что многие путают их подлинное назначение со “случайными” проявлениями.

Самым большим достижением системы управления базой данных стало весьма значительное облегчение процесса *внесения изменений в программное обеспечение.* Благодаря СУБД облегчается модификация прикладных программ, логической и физической структур файлов данных. Во многих случаях СУБД различает, стоит ли вносить изменения или нет.

Второй причиной создания СУБД является стремление к экономии пространства для файлов. Третья причина — это необходимость повысить достоверность информации в файлах, т.е. облегчить проверку отсутствия синхронизационных сбоев. Достоверность повышается благодаря уменьшению общего числа файлов. И наконец, с появлением СУБД облегчается доступ к данным. Многие ошибочно считают эту четвертую причину возникновения СУБД самой главной

Как работает СУБД. Для понимания принципов работы системы управления базой данных полезно обратиться за иллюстрацией к организации авторемонтного дела. Начиная дело, я привлекаю всего трех механиков, причем каждый работает со своими собственными инструментами. Никаких стандартов пока не существует. Когда число механиков доходит до восьми, мы начинаем сталкиваться с проблемой несовместимости. Прибор, которым механик А1 устанавливает момент зажигания в автомобиле мистера Z, отличается от всех других аналогичных приборов — и, когда этот клиент начинает жаловаться, я проверяю все приборы и обнаруживаю, что все они работают по-разному! Различия между ними вызывают тревогу. Какой же из них “правильный”?

Шаг 1

Я ввожу стандарты на все инструменты и приборы. Они должны быть определенных марок и моделей. По мере расширения мастерской я обнаруживаю, что часть приборов для установки момента зажигания остается без дела, и вовсе не нужно иметь их столько же, сколько механиков.

Шаг 2

Я отвожу специальную кладовую для инструментов и приборов, в которой мы храним самые дорогие приборы, и выдаем, “выписываем”, их по требованию отдельным механикам, которые возвращают их после выполнения работы. Это уменьшает число используемых приборов, а также облегчает задачу их ремонта и калибровки.

Шаг 3

Я обнаруживаю, что работ по регулировке зажигания становится очень много, и создаю специальный отдел регулировки зажигания. Все работы по системе зажигания проводятся только здесь, даже в тех случаях, когда регулировка зажигания является лишь частью необходимых работ.

С чем-то подобным мы сталкиваемся и в области программного обеспечения.

Сначала у каждого программиста имеются собственные файлы, так же как у каждого механика имеется свой регулировочный прибор для установки момента зажигания. Программист может полностью распоряжаться своими файлами. Он определяет их размер, формат и содержание.

При таком порядке возникли три проблемы. Во-первых, программисту было очень трудно получить данные из чужого файла. Во-вторых, при изменении данных, скажем при переходе от чисел с 12 знаками к числам с 14 знаками, приходилось изменять *все программы.* Это было трудно, дорого, а во многих случаях просто невозможно. В-третьих, данные программиста А несколько отличались от данных программиста В. Чьи же данные были правильными?

Шаг 1

Мы утвердили стандарты на файлы — размеры, форматы, последовательности — и тем самым облегчили использование данных, подготавливаемых другими программистами.

Шаг 2

Мы создали централизованные файлы, для которых ввели правила использования, т.е. определили, какие операции можно выполнять и где эти файлы располагать. После этого мы поместили все данные в центральное хранилище и разрешили программистам

пользоваться находящимися там данными только в том случае, если они следуют установленным нами правилам и правильно оформляют свои запросы. Их программы взаимодействовали с *моими* программами, которые в свою очередь управляли работой с файлами. Мои программы были системными программами.

Это сразу избавило нас от многих неприятностей.

1. Данные хранились в меньшем числе файлов; это экономило место.

2. Стало легче отслеживать текущее состояние элементов данных.

3. Стало возможно изменять размеры данных в файлах (числа с 12 знаками на числа с 14 знаками) без изменения всех индивидуальных прикладных программ.

Все это достигалось исключительно тем, что все работы по записи и считыванию данных были сосредоточены *в одной* программе. Но программисту все же еще нужно было *знать* о файлах очень много различных подробностей — их содержимое, используемые форматы, а также точные способы организации запросов. И тут было обнаружено, что вовсе не каждому программисту нужны столь подробные сведения об обрабатываемых им данных.

Шаг 3

Так появилась система управления базой данных. Большая программа, выполнявшая все манипуляции с данными, стала еще больше. Программистам больше не нужно было знать детали структуры файлов. Им оставалось теперь только идентифицировать нужные им данные, а система управления базой данных, представляющая собой очень большой набор программ, выполняла все остальное.

СУБД обычно сопровождается другими программами, которые 1) обеспечивают работу с дисплеями и 2) позволяют формулировать запросы к содержимым файл на простом *языке.* Такой язык часто называется *языком запросов.* На шаге 3 создается *информационно-поисковая система.* Функции, определенные нами на шаге 2, уточняются таким образом, чтобы они могли *помочь* при *поиске* данных. Но это лишь некоторая дополнительная выгода, побочный эффект усилий, прилагаемых для облегчения внесения изменений в файлы. Это отнюдь *не* главная причина, приведшая к появлению СУБД.

В табл. 4.3 сведены воедино все преимущества, даваемые СУБД.

Использование системного программного обеспечения. Зададим себе два вопроса, которые нам помогут сосредоточить свое внимание на системном программном обеспечении. Зачем нам системное программное обеспечение? На *всех* ли машинах оно используется?

Главная причина возникновения системного программного обеспечения — стремление максимизировать загрузку вычислительной машины! Машина должна постоянно работать с возможно более полной нагрузкой. Чтобы этого добиться, мы и пишем программы, в помощь оператору, инженерам, программистам. Но вся эта помощь оказывается им только ради максимального использования машины, поскольку все эти программы увеличивают ее занятость. Вторая причина — облегчить возможность вносить изменения, а третья — повысить производительность труда программистов, избавив их от дублирования работ.

Таблица 4.3. Преимущества использования системы управления базой данных

Описание проблемы	**Метод, применяемый в системе управления базой данных**	**Выгода от использования СУБД**
Дублирование данных, то есть рост размеров файлов	Один файл	Экономия места на диске
Расхождение данных, неодинаковость соответствующих друг другу файлов	Один файл	Данные становятся более достоверными
Любое изменение содержимого файлов, или их структуры, или прикладных программ сразу приводит к новым разработкам	Расслоение	Облегчается внесение исправлений; исключается необходимость модификации прикладных программ

Проведенные в начале 1960-х гг. измерения фактического уровня использования большого числа различных вычислительных машин, к большому удивлению очень многих, показали, что мощность центрального процессора использовалась не более чем на 50%. Операционные системы позволили значительно повысить этот показатель!

Ответом на второй вопрос может быть только одно слово — нет! Отнюдь не все машины работают с операционными системами. Могут работать все, но не все работают. Подавляющее большинство, возможно, более 90%, но все же не все. Позже мы увидим, что для использования в системах реального времени стандартное системное обеспечение работает слишком медленно.

Теперь ясно, что системное программное обеспечение в настоящее время начинает делать то, что раньше приходилось делать прикладным программистам. При этом как системные, так и прикладные программы выполняются в фазе использования (см. табл. 4.4). Каким же образом мы сможем отличить одни программы от других?

Таблица 4.4. Функции системных программ

Системная функция	Ранее выполнялась
1 Переключение лент	Операторами
2. Распределение ресурсов, расстановка приоритетов по ресурсам	Операторами и начальником машинного зала
3. Восстановление после ошибки	Операторами; программистами
4 Работы по вводу/выводу	Прикладным программистом
5. Работы с данными и файлами	Прикладным программистом
6 Работа с линиями связи	Прикладным программистом
7 Работа с дисплеями	Прикладным программистом
8. Организация диалога	Прикладным программистом

Отнести конкретную программу к системному или прикладному обеспечению нам помогут два следующих критерия:

1. Откуда возникла данная программа? Была ли она разработана прикладным программистом или отдельной группой, созданной для сопровождения программ, а может быть, ее разработали те же, кто создал и аппаратуру? Кто сопровождает эту программу?

2. Насколько универсальна данная программа, могут ли ее использовать какие-либо другие прикладные программисты?

В фазе использования программы, написанные прикладными программистами совместно с группой системных программистов, для стороннего наблюдателя ничем не отличаются от программ, целиком созданных одними прикладными программистами. Однако в фазах разработки и сопровождения различия становятся очень и очень заметными.

Разнообразие операционных систем. Некоторую путаницу в вопросы, связанные с операционными системами, вносит и тот факт, что их развитие привело в настоящий момент к появлению специализированных систем.

Для одной и той же аппаратуры создаются операционные системы, которые рекомендуется применять в фазе использования, и системы, которые следует применять в фазе разработки. Имеются пакетные операционные системы, системы управления сетями, системы реального времени и операционные системы, ориентированные на пользователя. На некоторых машинах реализованы своеобразные смеси всех этих операционных систем.

Стоимость операционной системы. Создание и сопровождение операционных систем обходится фирмам-изготовителям в миллионы долларов, а иногда счет доходит до миллиардов. Зачастую, однако, эти расходы не выделяют из стоимости аппаратуры.

А ведь в фазе использования операционные системы тоже кое-что “стоят”: они “едят” и машинное время, и память. Тысячи людей пользуются операционными системами, это заставляет предусматривать в них широчайшее многообразие функций. Если мне какие-либо функции не нужны, я могу попробовать исключить некоторые из них из своей системы, но все исключить невозможно. Отсюда следует, что вычислительная машина делает совсем ненужные мне вещи. С этим приходится мириться, поскольку это обходится все же дешевле, чем создание собственной операционной системы или передача ее функций прикладным программам.

Источник системного программного обеспечения. В настоящее время все возрастающую долю системных программ пишут компании по производству программного обеспечения или пользователи, однако до сих пор большую их часть создают изготовители аппаратуры. Почти все системы управления базами данных отделены от операционных систем. Хотя эти системы используются совместно, но разработка их ведется по отдельности, и продают их чаще всего отдельно друг от друга.

Системные и прикладные программы в фазе использования. Итак, прикладными считаются программы, которые печатают платежные ведомости, управляют полетом ракеты, ведут самолет на посадку или выписывают чеки.

Большинство прикладных программ пишется таким образом, чтобы пользоваться ими можно было только совместно **с**

системными программами, которые управляют работой машины и ее окружением в фазе использования, а также выполняют наиболее общие функции, в частности печать. Взгляните на схему распределения памяти, изображенную на рис. 4.16. На схеме приведены те программы, которые размещаются в памяти вычислительной машины при выполнении производственного задания, ради которого эту машину и приобретали.

Из рис. 4.16 очень хорошо видно, что основную часть памяти машины занимают системные программы, управляющие работой этой машины и ее внешним окружением.

На большинстве машин программы пишутся таким образом, чтобы они соответствовали действующей операционной системе, работали именно в ней, выполнялись совместно с ней. Программисты, работающие в какой-то фирме, разрабатывают и пишут программы выдачи платежных ведомостей фирмы. Когда работает программа составления платежной ведомости, одновременно с ней работает операционное и другое системное программное обеспечение.

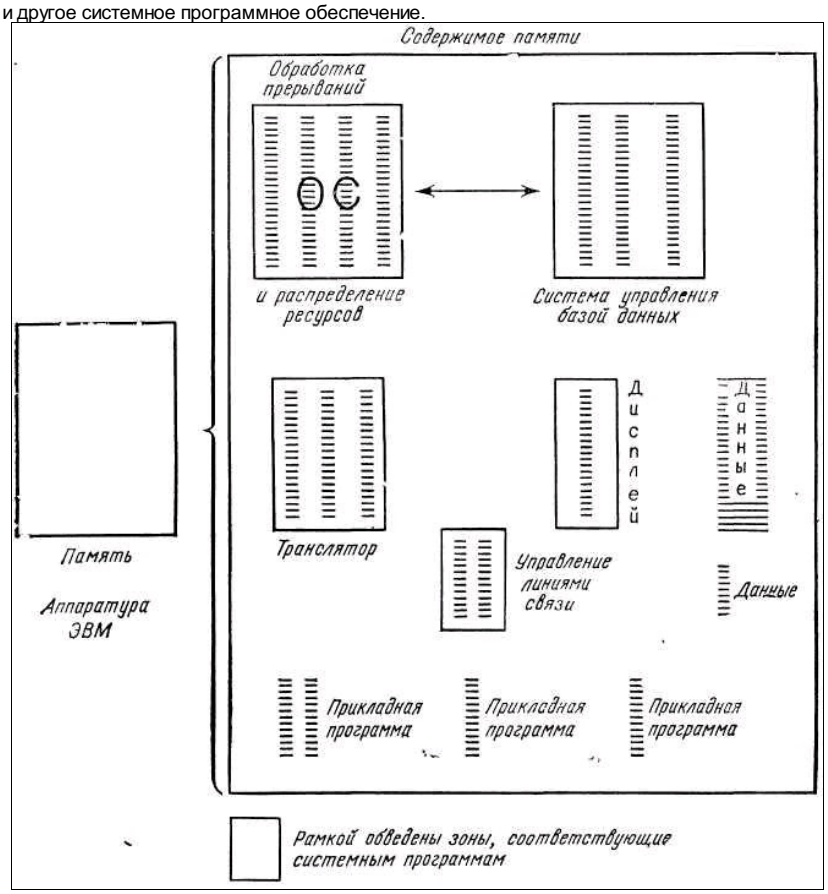


Рис. 4.16. Схема распределения памяти в фазе использования.

В самом начале процесса выполняется операционная система, которая определяет вид работы, сообщает операторам, какие ленты и куда надо поставить и т.д. В дальнейшем начинает выполняться собственно программа печати ведомости. Если возникает прерывание (например, встречается ошибка), управление опять передается операционной системе, которая обрабатывает ошибку, а затем возвращает управление программе печати.

Но этот пример слишком прост. Очень часто бывает так, что в одно и то же время в памяти машины размещается несколько, а то и несколько десятков различных прикладных заданий.

В случае мультипрограммирования операционная система управляет одновременно работой дюжины прикладных программ, распределяет между ними все ресурсы машины, доводя до максимума просто “работу” — работу, выполняемую за определенное время. Подробнее этот вопрос будет изучаться в гл.7.

Плохая операционная система может обесценить сколь угодно хорошую аппаратуру, в то время как хорошая система может спасти и плохую. Часто системные программы оказываются медленными просто из-за того, что им приходится разбираться с огромным количеством возможных пользователей, данных и т.д., а для этого нужны тысячи и даже миллионы команд. Иногда они бывают медленными, потому что плохо разработаны или плохо скомпонованы.

Стандартное и нестандартное системное обеспечение. *Существуют две группы задач — для одних используется стандартное системное обеспечение, а для других нет.* В операционные системы и системы управления базами данных уже вложено столько денег и усилий, что их нужно использовать везде, где только возможно. Писать новые программы для выполнения их функций очень разорительно.

В то же время существует часть пользователей вычислительных машин, которые обязаны создавать свои собственные системные программы, — это пользователи систем реального времени типа V. Необходимость выполнить цикл вычислений за определенное время — миллисекунды в оборонных системах, системах гражданской авиации и NASA или секунды в системах резервирования,— а также необходимость высокой надежности делают невозможным использование стандартного программного обеспечения. Поэтому многие пользователи систем реального времени должны создавать обеспечение сами.

Системные программы индивидуального пользования пишутся довольно часто. Иногда такие системные программы могут применяться более чем одним пользователем. Разработанная фирмой IBM для резервирования авиационных билетов операционная система PARS (или ACP) используется более чем двумя десятками авиакомпаний и несколькими банками. Создание системы PARS было обусловлено тем фактом, что система OC/360 оказалась слишком большой и медленной.

Система диспетчеризации воздушного транспорта, управляющая рейсовым авиационным транспортом, была написана один раз, а используется в 20 авиапортах Соединенных Штатов и в одном из авиапортов Великобритании. OC/360 не смогла обеспечить необходимую надежность и подходящую схему распределения ресурсов; пришлось FAA (Federal Aviation Agency) (с помощью отделения федеральных систем IBM) писать собственные системные программы.

Такая необходимость писать специализированное системное обеспечение является одной из причин высокой стоимости и трудоемкости больших программных систем типа V.

Не так давно сразу в двух разных книгах я встретил утверждение о том, что вычислительные машины не сбиваются при работе. Это явно абсурдное утверждение; конечно же, у них бывают сбои. Все электронные устройства подвержены сбоям.

Один из авторов пытался утверждать, что оправдание типа “произошел сбой вычислительной машины” представляет собой не более чем мошенничество; ошибка обычно заключена в процедурах или командах программ.

Хотя я согласен, что фраза “виновата машина” просто отговорка, но все же утверждение, что у машин не бывает сбоев, представляется слишком вредным, особенно в книге вводного характера.

Поскольку руководство знает, что вычислительные машины все же выходят из строя, оно должно позаботиться о том, чтобы включать процедуры проверки функционирования. Любой “сбой машины” означает недостаточно квалифицированное руководство, поскольку оно не обеспечило достаточно надежную защиту системы с помощью как программного, так и аппаратного контроля.

Для обеспечения правильного, своевременного и бесперебойного выполнения задачи руководство вольно выбирать либо

стандартное, либо изготовленное специально программное обеспечение. Сегодня такой выбор вполне возможен.

От пакетного режима к режиму реального времени. Переход от пакетной обработки к работе в режиме реального времени не требует слишком больших переделок прикладных программ и их логики. Баллистические траектории остаются баллистическими траекториями. А вот реорганизация системных программ действительно необходима.

Очень часто эта капитальная реорганизация не предусматривается заранее, что приводит к ужасным последствиям. Люди думают, что если они отладили свои программы в пакетном режиме, то сам переход к режиму реального времени не составит для них затруднения. Графики перехода не разрабатываются, сметы с затратами не составляются.

В системах реального времени основным фактором является само время. В пакетных системах данные, например данные от радиолокационных станций, собранные на магнитной ленте, сначала вводятся в машину, а затем обрабатываются в ней. Как только будут обработаны все данные, случись это через 24 ч. или даже через неделю, машина закончит свою работу. Время при этом не принимается во внимание.

В системах же реального времени данные радиолокаторов *должны* быть обязательно обработаны не более чем, скажем, за 6 с, иначе система утеряет часть жизненно необходимых ей данных. Это *обязывает* операционную систему вести распределение работ таким образом, чтобы обеспечить обработку *всех* данных именно за 6 с. И точка! И конечно же, нужно иметь программы, обеспечивающие достаточную надежность (возврат и восстановление).

Переход от хорошей пакетной системы к системе реального времени — это переход к новым концепциям.

Преимущества системного программного обеспечения

1. Системное программное обеспечение увеличивает модульность и улучшает защиту информации, значительно упрощая процесс внесения изменений в программы.
2. Системное программное обеспечение избавляет прикладных программистов от необходимости затрачивать большие усилия на сопровождение стандартных программ.
3. Уменьшая простои, системное обеспечение доводит до максимума использование аппаратуры.
4. Исключая дублирование информации во внешних файлах, системное программное обеспечение лучше использует память.

Недостатки системного программного обеспечения

1. В силу универсальности системных программ снижается скорость их выполнения по сравнению со специализированными системами.
2. Системные программы велики, сложны, их часто трудно использовать надлежащим образом.
3. Системное программное обеспечение не всегда обладает гибкостью, достаточной, чтобы удовлетворять всем индивидуальным требованиям.

Таблица 4.5. Эволюция системного программного обеспечения

Проблема	Ее решение
Машина простаивала, в то время как оператор в спешке ставил магнитные ленты и т.п.	Была написана программа, которая отслеживала список поставленных лент, переключая ленты не физически, а логически. Это привело к сильному увеличению числа магнитофонов. В одно и то же время на машине стало возможно выполнять несколько программ
Начальник вычислительного центра не успевал принимать решения по поводу того, какую задачу запускать на машине. Сколько для этой задачи потребуется памяти? Лент? И т.д.	Была написана программа, которая отслеживала списки свободных машинных ресурсов и стоящих в очереди программ. После этого машина стала сама распределять работы и устройства вычислительного комплекса
Программисты вставляли в прикладные программы детали физического расположения данных и дисков. Новые диски, более дешевые и быстрые, нельзя было внедрять до того, как будут переписаны старые прикладные программы. Это было очень трудно, поскольку программисты могли быть заняты чем-нибудь другим или вообще уволиться	Были написаны программы управления данными, которые выполняли чтение и запись данных на диске. Программисты стали теперь писать команды для программы управления данными, которая взяла на себя все заботы. Имена, использовавшиеся для идентификации этих стандартных системных программ, скорее вводили в заблуждение, чем вносили ясность. Одним из имен было “Методы доступа”. Конечно же все думали о методах и расположении данных, забывая о программах, реализующих эти методы
Разным программистам часто были нужны одни и те же данные, но в разной последовательности или в различном порядке. Поэтому им приходилось создавать свои собственные файлы из главного файла и в дальнейшем пользоваться уже своими файлами. Это было чревато двумя опасностями во-первых, память для файлов становилась все больше заполненной, но, что еще хуже, данные в одном файле не согласовывались с данными в другом файле	Были написаны программы управления базами данных, которые обеспечили сложный логический поиск файлов в тех случаях, когда файлы записывались не с теми ключами, которые использовались программистом для их поиска. Эти системы получили название систем управления базами данных (СУБД); они сняли с программистов обязанности по разработке и проведению логического проектирования методов поиска и хранения данных; все работы отныне выполнялись с помощью СУБД

Выводы о системном программном обеспечении. Мы теряем в скорости выполнения, памяти и гибкости, чтобы достичь порядка при работе, избежать создания дополнительных программ и снизить тем самым вероятность внесения ошибок в программную систему, а также чтобы облегчить процесс исправления программ. Системное программное обеспечение явилось огромным благом для пользователей, значительно увеличив коэффициент полезного действия вычислительных машин (см. табл. 4.5).

Для многих систем можно четко определить место их разработки, место работы и место сопровождения. Некоторые называют все обеспечение, посылаемое к месту работы, “операционным”. Это может вносить только путаницу, поскольку термин “операционное программное обеспечение” является синонимом термина “обеспечение времени использования”. Например, корабль, находящийся в море, является местом выполнения, но многие программы, посылаемые туда,, не работают в фазе использования. На корабль посылаются диагностические программы, помогающие инженерам-ремонтникам налаживать работу машин, но эти программы работают автономно, а не совместно с системой при ее использовании.

Таблица 4.6. Когда используется программное обеспечение разных типов

Тип программного обеспечения	Выполняются во время разработки	Выполняются во время использования	Выполняются во время сопровождения			
Инструментальное	Трансляторы	Нет	Трансляторы	Программа-библиотекарь	Программы-библиотекари	Отладочные программы	Отладочные программы
Системное	Операционные системы	Диалоговый режим	Операционные системы				
	Системы управления базами данных	Операционные системы СУБД Диагностика в диалоговом режиме Вычисления в диалоговом режиме	СУБД				
Прикладное	Нет	Ведомости (периодически) Управление или контроль (постоянно) Отслеживание даты (раз в сутки)	Нет				

Они представляют собой набор инструментальных программ, и называть их “операционными” только за то, что они находятся в одном месте со всей системой, ошибочно. В табл. 4.6 перечислено по крайней мере некоторое программное обеспечение, работающее в разных фазах независимо от места работы, а в табл. 4.7 показано, какие программы могут работать автономно в различных местах.

Таблица 4.7. Какое программное обеспечение может самостоятельно работать в различных местах

Выполняются самостоятельно по месту разработки	Выполняются самостоятельно по месту использования	Выполняются самостоятельно по месту сопровождения
	Диагностические программы	

Инструментальные	Диагностические программы	Диагностические программы Элементы калькуляции	Диагностические программы
Системные	ОС	Операционные системы	ОС
Прикладные	В целях тестирования программы калькуляции	Нет	В целях тестирования программы калькуляции

Инструментальное программное обеспечение

Инструментальное программное обеспечение — это третий раздел большой области всех программ. Применяется инструментальное обеспечение в фазе разработки. Инструментальное программное обеспечение — это совокупность программ, используемых для помощи программистам в их работе, для помощи руководителям разработки программного обеспечения в их стремлении проконтролировать процесс разработки и получаемую продукцию.

Наиболее известными представителями этой части программного обеспечения являются программы трансляторов с языков программирования, которые помогают программистам писать машинные команды. Инструментальными программами являются трансляторы с языков Фортран, Кобол, Джовиал, Бейсик, АПЛ и Паскаль. Они облегчают процесс создания новых рабочих программ.

Однако трансляторы с языков это только наиболее известная часть инструментальных программ; существует же их великое множество. Проще всего можно получить представление о количестве и разнообразии инструментальных программ, изучив список программ, используемых во время разработки большой программной системы. Взгляните на табл. 4.8.

Таблица 4.8. Некоторые инструментальные программы

Общее назначение	**Требования и спецификации**	
Текстовые редакторы	PSL/PSA	
Форматирование документации	Реляционные СУБД	
Архивные системы	Проверка непротиворечивости	
Работа с дисками и лентами	CARA/CLARA	
Модели	SADT IA	
Проектирование	**Написание**	
Графические пакеты	Транслятор	
Построители структурных блок-схем	Кросс-транслятор	Предтранслятор
Проектный анализатор APLGOL	Программа-библиотекарь	
Конструирование	**Верификация**	
Система планирования и руководства разработками PERT	Статические анализаторы	Символическое выполнение
Редактор связей	Интерпретация	
Проверка стандартов Библиотекарь	Генератор тестовых последовательностей	
	Сбор статистики	

Такое использование вычислительных машин для помощи в создании новых программ далеко не очевидно для людей, не являющихся профессиональными программистами. Часта же бывает так, что профессионалы рассказывают об инструментальном (фаза разработки) и системном (фаза использования) программном обеспечении на едином дыхании,, предполагая, что не посвященному в тайны их мастерства известно об этой роли инструментального программного обеспечения. Так же как и в фазе использования (для прикладных программ), системное обеспечение работает и в фазе разработки, но только совместно с инструментальным обеспечением.

Стоимость полного комплекта необходимых для разработки инструментальных средств может легко дойти до многих десятков и даже сотен миллионов долларов. Для выполнения инструментальных программ, кроме всего прочего, нужна также и вычислительная машина. Подробнее о средствах обеспечения разработки мы поговорим в разделе, посвященном *средствам разработки*.

Так же как и все другие программы, инструментальное программное обеспечение может содержать ошибки. Этот не очень очевидный факт вынуждает нас для придания людям уверенности в высоком качестве наших инструментальных средств, для содержания их в рабочем порядке, каталогизации, обеспечения помощи производственным программистам в обучении, для их использования, а также для исправления в случае, если они не дают верных результатов, организовывать специальные группы инструментального сопровождения. Такие группы сопровождения, состоящие из системных программистов, представляют большую ценность. (Кому хочется работать с неисправными инструментами?) Затраты на их создание оправдываются увеличением производительности труда другой части программистов, участвующих в разработке. О необходимости таких групп часто забывают, и статьи в бюджете на них не отводят.

Кроме “обычных” средств разработки мы можем упомянуть некоторые дополнительные вспомогательные области:

Разработка тестового программного обеспечения

имитаторы; моделирующие программы; генераторы.

Обслуживающее тестовое программное обеспечение

диагностика; тесты; помощь при техническом обслуживании.

Обучение с помощью программного обеспечения

помощь в изучении; программированные инструкции; программы-тренажеры.

Стоимость инструментального программного обеспечения

Инструментальное обеспечение может стоить сотни миллионов долларов. Военно-морской флот США принял решение использовать на борту всех кораблей только две различные вычислительные машины, работающие в разных диапазонах производительности. Ремонт и поставка запасных частей в море настолько сильно влияют на систему в целом, что размещение на всех кораблях одинаковых машин значительно облегчает расчеты перевозок и снабжения.

Побочным эффектом этой стандартизации является унификация системы команд, а как следствие огромные запасы инструментальных программ, разработанных за многие годы, могут использоваться при разработке всех без исключения бортовых приложений.

В середине 1979 г. в ВМФ США подсчитали, что на создание инструментального программного обеспечения для этих машин было затрачено около 300 млн. долларов. Такая сумма не является чем-то необычным. Чтобы сопровождение машин происходило хорошо, должно быть создано огромное число различных программных инструментальных средств.

ВМФ США предпринял смелое и успешное начинание, отвергнув создание целой системы нового инструментального обеспечения, когда при заказе новой самолетной бортовой машины потребовал, чтобы ее система команд совпадала с системой команд одной из судовых ЭВМ. Многие считали, что такие действия будут тормозить развитие вычислительной техники, но работы завершились успешно, налогоплательщикам сэкономлены многие миллионы долларов.

Масштаб, сложность, ясность

Все усилия по разработке программного обеспечения можно характеризовать в зависимости от масштабов и сложности этой работы, а также ясности и понятности полученных программ.

Масштаб

Термин “программное обеспечение” имеет очень широкое значение. Это общее обозначение подобно слову “животное”, которое одновременно относится и к вашей любимой кошке, и к огромному трехсоткилограммовому белому медведю. Несмотря на это, люди говорят о программном обеспечении как о некой вещи или как об однородной группе вещей. Все что угодно, только не это.

В полном одиночестве за двадцать минут я могу написать программу из 100 команд, которая будет вычислять мои ежемесячные выплаты процентов по ссуде. Это будет разработка программы. За 20 минут.

Я также могу быть одним из нескольких сотен программистов и руководителей проекта создания программы вычисления траектории и управления полетом космического корабля на Луну. Я буду создавать свою часть большой, более 1 млн. команд, программы, причем это может длиться несколько лет.

В обоих случаях я буду выступать в роли разработчика программ. Но усилия мои будут весьма различными. А быть руководителем группы, ответственным за все эти миллионы команд, это абсолютно другой вид деятельности.

Есть “программирование в малом” и “программирование в большом”. 100 команд — это просто программа, миллион команд — это уже программное обеспечение.

Давайте поищем (вокруг себя) аналогии, которые помогут нам понять наши технологические проблемы. Строителем моста может

быть человек, который строит полутораметровое сооружение из тонких досок над узким ручейком на заднем дворе, или человек, создающий подвесной мост через реку или залив. Нельзя сравнивать такие мосты, они находятся на разных концах масштабной шкалы. Но и тот и другой — мост!

Попробуем теперь разобраться в том, что такое планирование, организация и контроль работ особенно большого объема. Надо изучить требования к мосту, его несущую способность, его местоположение, подъездные пути и еще множество других совсем не очевидных вещей. Теперь пришло время сделать грубую оценку стоимости сооружения. Затем следует еще один просмотр требований и возможных вариантов моста.

Когда место расположения моста определено и его примерные очертания приняты, начинается подлинное планирование работ по его созданию. Проводятся детальные расчеты моста, его вантов, опор, пилонов, его колебательных характеристик, кабельной системы — и еще множества других технических деталей, которые составляют мост как целое. Для хранения чертежей приходится строить отдельные помещения. Только для отыскания нужных чертежей требуется специальная информационно-поисковая система, позволяющая при необходимости получить любой из них.

Создается “рабочий план”. Тщательно детализируются и документируются все сведения о том, что, когда и кем должно выполняться, какая между всем этим есть взаимосвязь. Определяется параллельность работ (какие работы могут выполняться параллельно друг другу).

Все это надо распланировать и понять заранее. Только после этого можно приступать к найму строителей, выписывать ордера на материалы, инструменты и доставлять оборудование.

Для начисления рабочим зарплаты создается бухгалтерия. Одна комната заполняется за другой, поскольку для управления созданием такого огромного сооружения требуется огромное количество документов.

Проходят годы. Мост становится реальностью, чудом, воплощенным в жизнь. Его могут видеть миллионы людей, видеть уже построенным, пользоваться им. Это триумф (см. рис. 4.17).

Но никто не видит долгих лет подготовки, тонны документов, сложную и удивительную работу по планированию. Они невидимы для тех, кто ходит по мосту, но абсолютна необходимы для достижения результата.

Урок, который можно извлечь из табл. 4.9, очевиден — для строительства моста стоимостью в 300 млн. долларов нужен тщательно разработавшей “фундамент,. Все это с полной уверенностью можно отнести и к программному обеспечению стоимостью в 100 млн. долларов, состоящему из программ размером от 500 тыс. до 2 млн. строк текста.

Эффект больших масштабов проявляется во всех отраслях Дональд Дуглас, один из пионеров авиации, говорил, что “когда вес документов достигнет веса самолета, самолет начнет летать” (См. рис. 4.18.)

Джеймс Мартин утверждает, что “документы для “Боинга-747” весили больше, чем сам самолет”. То же самое можно сказать и о большом программном обеспечении. Здесь может возникнуть вопрос, много ли в настоящее время имеется систем из программ в миллион строк, много ли их будет появляться в будущем. Некоторые я перечислил в табл. 4.10, но дело в том, что их с каждым днем становится все больше.

Таблица 4.9. Эффект влияния роста размеров на рост усилий

	Пешеходный мостик в парке	Мост в Веразано
Выработка требований	1 день	1825 дней
Разработка	1 листок бумаги	Большой склад документов
Материальный план	1 ч.	1460 дней
план осуществления	1 ч.	182 дня
материалы	1/2 дня	182 дня
заготовки	1/2 дня	182 дня
инвентарные склады	1 день	182 дня
обеспечение реализации	1 день	182 дня
использование	2 дня	36 500 дней
План занятости людей		
число людей	2	5000
занятость строителей	1 день	365 дней
Общая занятость людей	3 дня	3650 дней
занятость бухгалтеров	1 день	3650 дней
калькуляция всего этого	1 день	3650 дней
Строительство	3 дня	1825 дней
Документирование	1 день	555 дней
число листов бумаги	5 листов	500 000 листов
Полная стоимость	1000 долларов	300 000 000 долларов

Таблица 4.10. Большие программные проекты

	Сумма контракта на все время использования (млн. долл.)	Число команд (млн.)	Затраты (чел.-год)*
Хьюстон (Аполлон/Скайлэб)	209	23.00	6000 +
Управление дальней связью	30	1.25	1000 +
Управление авиатранспортом	103	1.48	5000 +
Противоракетная система	120	1.87	3500 +
Обработка данных со спутников в реальном времени	23	0.55	1300 +

*) Заметим, что получить какие-либо согласующиеся данные путем простого деления данных из одной колонки на данные из другой невозможно. К вопросу о производительности труда мы обратимся в гл.6.

Появление больших систем программного обеспечения обусловлено снижением стоимости аппаратуры вместе с одновременным увеличением его мощности. Список систем не ограничивается приведенными в таблице, этот список все пополняется. Я знаю множество программ для министерства обороны, в которых затраты на программистскую часть превысили 50 млн. долларов. Этого уровня достигает и промышленность. В этот диапазон попадают большие системы связи фирм ATT, RCA, W.U., Satellite Business Systems. Операционные системы, сделанные для крупнейших промышленников, имеют даже большие размеры и стоят дороже.

Как мы уже видели, на авиатранспорте применяются столь же большие программы. Системы подобных размеров начинают заводить себе банки, особенно иностранные.

Одной из программ, стоившей гораздо больше 100 млн. долларов, была система наземного контроля космических кораблей типа Аполлон, созданная хьюстонским Центром пилотируемых полетов. Я был в Хьюстоне в 1970 г. сразу же после вступления на пост главного управляющего федеральным системным центром с целью проинспектировать работу 700 человек, подчинявшихся лично мне.

Они показали мне, как они управляют ходом разработки программы в миллион строк. Я был просто потрясен! “Это сущая бюрократия”,— думал я, когда мне показывали планы, руководства, формы, тесты и еще множество всевозможных документов.

Вскоре после этого я посетил еще некоторые подчиненные системы. Там не было такой большой системы управления разработками — и разработки были неуправляемыми. Самым правильным был подход, применявшийся в Хьюстоне,— для управления разработкой действительно большой системы более 50% средств нужно направлять на планирование, проверку, составление графиков, руководство и управление. Это и есть та инфраструктура, которую мы видим в других отраслях. Позже мне показали график, изображенный на рис. 4.19. Он и сейчас соответствует действительности.

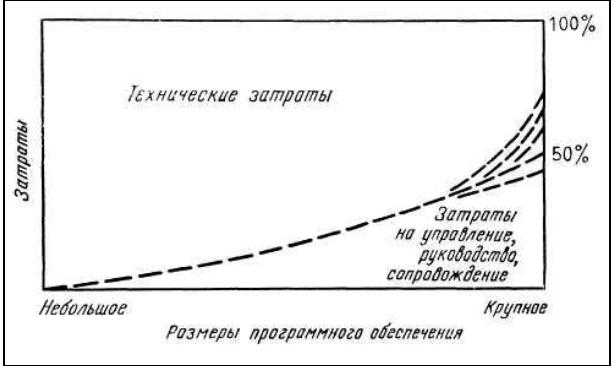


Рис. 4.19. Процентное отношение технических затрат к вспомогательным в зависимости от масштаба работ.

Наверное, самый понятный пример эффекта, проистекающего от возрастающего масштаба работ, был мною обнаружен при изучении одного интересного и удивительного факта, когда в 1968 г. я прибыл в отделение федеральных систем для того, чтобы стать помощником его президента Боба Эванса. Около десяти человек из Хьюстонской космической группы были направлены в Лондон, в один из банков. Для чего это было нужно? Специалисты по космосу помогали Лондонскому банку налаживать обработку данных?

Или, может быть, некоторые из членов Хьюстонской группы были специалистами по банковскому делу? Вовсе нет. Причиной их командировки в Лондон был тот факт, что самыми крупными заморскими партнерами фирмы IBM являются банки. А европейские, японские и другие иностранные банки в противоположность банкам США не столь сильно скованы рамками всевозможных законов и границами государств. Для связи с тысячами своих отделений они пользуются системами, стоящими более 100 млн. долларов.

Банкам понравились люди из Хьюстона. Несмотря на то что они ровным счетом ничего не понимали в банковском деле, они имели большой опыт по реализации систем *именно таких масштабов,* какие были нужны банкам, — в то время таких людей было немного. Способы управления крупномасштабными работами нельзя рассматривать как простую сумму усилий на управление мелкими проектами. Работы по созданию любого крупного объекта содержат в себе немало тонкостей.

Сложность

Одним из позднейших фундаментальных открытий математики было открытие числа нуль. Оно появилось поздно, поскольку вначале его необходимость не была очевидна.

Так же обстоит дело и с понятием сложности. Сложность легко себе представить, но трудно описать. Еще мало разработано приемов для работы с понятием сложности. У нас нет метрики для измерения того, что одна работа вдвое сложнее другой. Нет прилагательных для определения степени сложности. Мы вынуждены говорить, что это “более сложное” или “очень сложное”. Это, быть может, и верно, но не очень полезно, когда нам надо создать нечто “более сложное”, чем что-то еще. Ведь стоимость наших работ достигает миллионов долларов, а их результаты имеют очень значительные последствия. Мы, однако, можем четко различать два “вида” сложности:

1) техническая сложность конкретного приложения.

2) логическая сложность приложения и/или системы программного обеспечения.

Техническая сложность. У меня в распоряжении две программы, по 50 тысяч команд каждая. Одна “делает” платежную ведомость, а другая “делает” вычисления, связанные с безопасностью летящей ракеты. Ракетная программа будет более сложной; в ней требуется решить некоторые сложные уравнения.

Предсказание погоды, уравнения ядерной физики, орбитальные расчеты, гравитационные эффекты, баллистика — все это требует специального математического аппарата, инженерного искусства, а также высокой научной квалификации и знаний. Эти знания нужно использовать во всех фазах разработки программного обеспечения. Получение таких знаний само по себе достаточно трудный процесс, использование же их трудно вдвойне.

Логическая сложность. Существует и другой вид сложности — логическая сложность, причем управляться с логической сложностью даже труднее, чем с математической или технической. Эта сложность проявляется в многообразии различных вариантов, выбор среди которых надо производить на каждом шаге решения. Каким образом можно проследить все возможные или подходящие нам пути, по которым проходит управление при выполнении больших программ? В программах может быть лишь несколько условных переходов, в других же программах их может оказаться огромное множество! Попробуем разобраться, почему же бывает трудно справиться с программой с большим количеством условных переходов.

Сколькими различными способами можно соединить с контактами пять проводков из семижильного жгута? Таких способов 2520.

Сколько различных порядков при ударе можно выстроить для 12 — всего лишь 12 — ребят, игроков бейсбольной команды младшей лиги? Подсчитали? 79 миллионов 833 тысячи 600! Только для 12 ребят!

Одной из самых логически сложных программ является программа диспетчеризации воздушного транспорта, применяемая в США и Великобритании. Она стоила более 100 млн. долларов и используется в 20 диспетчерских центрах США, а также в Лондоне. К настоящему времени мы имеем уже пятилетний опыт ее эксплуатации, система работает вполне удовлетворительно. Великобритания, приобретая систему, выплатила за нее и вычислительную машину IBM 9020, на которой система работает, 10 млн. долларов наперекор жесточайшей политике “Покупать только британское” (Тот факт, что иностранная держава выбрала и использует американскую систему, привел к тому, что Федеральное агентство Соединенных Штатов по авиации отказалось от своих многочисленных нападок на систему и обвинений в ее бесполезности, несовременности и т.д.).

Во время переговоров при заключении контракта на изготовление диспетчерской системы мы специально изучали 600 тысяч строк программы, подготовленной для работы на центральном вычислительном комплексе, и обнаружили в ней большое число условных переходов. Мы насчитали 39 203 таких перехода, т.е. в среднем по одному на каждые 15.3 строк текста В этой программе очень много внимания уделяется принятию различных решений, что связано с запутанной логической структурой управления, предсказания возможных конфликтных ситуаций, а также множества различных вариантов, возникающих при работе с 97 устройствами для ввода данных и запросов к системе, форматной выдаче данных на дисплеи. Сколько же вариантов возникает при выполнении этой программы? Это число равно 39 203! Это просто астрономическое число, оно примерно равно 10^{11801}, или десятке, за которой следует 11800 нулей! Если бы мы могли проверять один вариант программы за одну секунду, то на тестирование всей программы в целом нам пришлось бы затратить несколько тысяч лет. Мы не можем создать специальную тестирующую систему, которая могла бы проверить нам все варианты, возникающие в окружающем нас мире.

Если кто-то начинает говорить о том, что он создал программу, в которой нет ни одной ошибки, значит, либо он говорит об *очень* маленькой программе, либо вообще не знает, о чем говорит.

Большие системы программного обеспечения логически очень сложны. Они содержат огромное число ветвлений. В своей книге “Мифический человеко-месяц” Брукс *{Frederick P. Broocks, Jr. Mythical Man-Month (Reading, Mass; Addison — Wesley, 1975) . Есть русский перевод, см. примечание на стр.5.}* утверждает, что системное программное обеспечение в девять раз труднее разрабатывать, чем прикладное обеспечение. Основной причиной этого является логическая сложность.

Если отдельно рассмотреть прикладные программы, они окажутся не такими уж сложными. Однако при первой же попытке объединить некоторое количество прикладных программ в цельную однородную систему мы сразу же столкнемся со сложными проблемами. Рис. 4.20 взят из книги Джеймса Мартина “Организация баз данных”.

К сожалению, в настоящее время вопросу изучения логической сложности уделяется недостаточное внимание. По мере внедрения вычислительной техники в различные прикладные области, связанные с проведением большого числа сложных операций и управления процессами, мы лучше сможем представить себе эту важную отрасль.

Проблемы, связанные со сложностью. Сложность всегда доставляла и до сих пор доставляет людям множество неприятностей. И никого не должно удивлять то, что она мучает нас при разработке больших программных систем,— однако это удивляет. Ведь мы склонны всегда забывать, сколько мучений мы претерпели из-за сложности, где бы она ни проявлялась.

Очень долго у людей были сложности с мостами. За одно только десятилетие с 187Х) по 1880 г. только в одних Соединенных Штатах разрушалось в среднем до 40 мостов *в год.* Граждане переходили через мост с риском для жизни. “Социология” того периода

очень напоминает положение, сложившееся в настоящее время:

Катастрофы на крупных мостах случались гораздо чаще, чем на железной дороге. Многие мосты местного значения строились окружными властями, которые соединяли в себе техническое невежество и неумение решать экономические проблемы. Подрядчики и торговцы заключали самые дешевые контракты, что фактически вело к исключению других, возможно более хороших вариантов. Безответственные исполнители продавали все, что только было можно; при первой же возможности быстро наводили мост и тут же поспешно исчезали. Самые авторитетные фирмы были поставлены конкуренцией в весьма опасное положение[1]{Gies j. Bridges and Men. New York: Grosset & Dunlap, Inc, 1963.}.

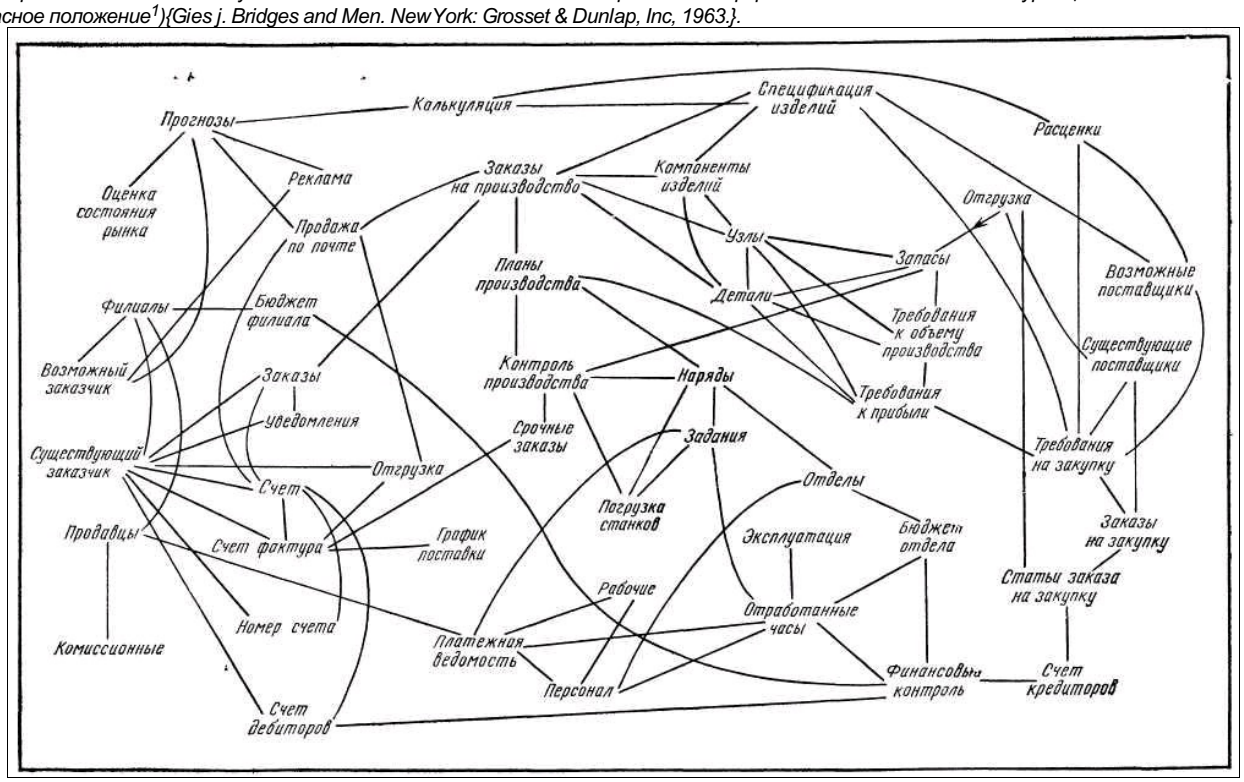


Рис. 4.20. Взаимосвязанные функции (Мартин Дж Организация баз данных в вычислительных системах. Пер. с англ. — М.. Мир, 1980) Печатается с любезного разрешения издательства Прентис Холл.

Как сильно все сказанное о том времени напоминает ситуацию, возникшую в программировании сегодня! А ведь семидесятые годы прошлого столетия не были первыми годами бедствий. Веками мосты разрушались из-за воздействия гармонических колебаний. Чего стоили только проходы через мосты солдат, марширующих *в ногу*. В 40-х гг. XX в. возникла проблема ветра. В 1940 г. упал мост через Такомский залив. В 1970-х гг. мост Бронкс Уайтстоун, "укрепленный" после Такомской катастрофы, начал раскачиваться так сильно, что люди бросали свои автомобили и в панике прыгали с пролетов моста.

Проблемы возникали не только с подвесными мостами. В 1944 г. двухфермовый мост через Миссисипи в Честере, шт. Иллинойс, был сброшен ветром со своих ненадежных опор. В 1905 г., когда был разрушен мост на кронштейнах через реку Св. Лаврентия в канадской провинции Квебек, было убито 87 рабочих. Физическая природа таких строений была тогда недостаточно понятна людям.

Новые явления приносят нам новые сложности. Моторы компании "Локхид Электра" (Lockheed Electra) разрушаются из-за дефектов проектирования. Самолеты DC-10 не летают!

Во второй половине семидесятых годов в здании Джона Хэнкока в Бостоне уже не было стеклянных окон. Были заменены 10 тыс. 400-фунтовых стекол.

Люди до сих пор не справились со сложностями, возникшими вместе с открытием ядерной энергии. Убытки от катастрофы, происшедшей на Острове третьей мили, продолжают расти и уже достигли 4 млрд. долларов!

Ясность

После того как программа уже создана и работает, ее ясность зависит только от ее разработки и реализации. На стадии разработки нам часто доставляет неприятности неясность поставленных перед нами целей. Что должна делать программа? В каком-то смысле пять различных типов использования программ отражают атрибут ясности.

Работа вычислительной машины по типу I (коммерческие расчеты) совершенно ясна. Задачи по обработке данных в большинстве случаев уже полностью решены. Все, что нам надо сделать, это разобраться в используемых процедурах и запрограммировать их. На противоположной стороне шкалы находятся задачи по управлению процессами типа V.

Чаще всего при их программировании нам приходится сталкиваться с выполнявшими до сих пор все работы "старыми мастерами", т.е. какими-то умными, искусными, все понимающими людьми, которые способны держать в голове все параметры, варианты, возможные ловушки, несовместимости, несоответствия и разрывы. Эти люди "видят", как развиваются события, могут почувствовать, когда что-то не ладится, и "знают", что надо делать в этом случае.

Попробуйте записать все правила, алгоритмы, попробуйте повторить то, что эти люди могут проделать в своем уме, и вы столкнетесь с необходимостью уметь здраво рассуждать, распознавать сложные образы, обладать изрядной долей интуиции и "чувствовать". Все это с трудом поддается определению, и получить полезную систему можно только после нескольких неудач, напрасных разработок, после потери значительных денежных сумм.

При попытке автоматизировать нефтеочистительные заводы фирмы Exxon, расположенные в Эдмонтоне, Канада, и в Антверпене, Бельгия, фирма IBM потеряла более 10 млн. долларов. Выполняли работу две сотни моих хьюстонских сотрудников. Как-то один из разработчиков спросил инженера компании Exxon, каким образом он узнает, когда надо нажать на рычаг. "Очень просто,— ответил тот.— Я опускаю палец в струю и *пробую на вкус".* Попытайтесь теперь запрограммировать это!

При построении больших систем управления процессами, взаимодействующих со многими пользователями, реальное использование вычислительной машины можно осознать только *после* того, как система попадет *в руки пользователей,* т.е. *после* того, как разработка должна быть уже теоретически закончена. Пользователи всегда находят новые способы использования своих инструментов.

Прочитав это, мой читатель может возразить, что это скорее проблемы системы, а не программного обеспечения. И читатель будет прав.

Программное обеспечение наследует проблемы системы.

При разработке большой системы типа V по мере стабилизации других частей комплекса последнее, что может быть еще модифицировано,— это именно программное обеспечение. Что мы подразумеваем под "стабилизацией"? В больших системах типа V разрабатывается сразу множество различных элементов. Коммуникационные связи/ дисплей/ радиолокатор/ сонар/ телеметрия/ ракета/ спутник/ двигатели/ управление/ еще что-нибудь — что-то из этого списка будет самым новым, самым передовым в мире, может разрабатываться также какой-нибудь необычный способ объединения этих объектов. Эта новинка может преподнести вначале своей эксплуатации любой сюрприз, и нам придется приспосабливаться к реальному положению.

Основная тяжесть адаптации падает на две части нашей системы — на программное обеспечение и обслуживающий персонал. Максимум возможного мы стремимся сделать за счет программ, оставляя нагрузку на людей минимальной. Программное обеспечение

действительно будет "гибким" *{Игра слов, заключающаяся в том, что программное обеспечение по-английски — software — "гибкая или мягкая часть, или аппаратура".— Прим ред.}*, *если* мы правильно его разработаем и будем правильно управлять им. *Если* мы документируем его и разделим на отдельные модули. В этом случае мы и вправду сможем приспособиться к любым неожиданностям. Если же мы ничего этого не сделаем, то в итоге получим всего лишь грубый бетонный блок, а не гибкую систему!

Долгосрочное планирование и правильное использование фондов в дальнейшем могут сэкономить немалые суммы денег. Мост Джорджа Вашингтона из Нью-Йорка в Нью-Джерси был построен в 1931 г с одним путепроводом, но все расчеты напряжений были сделаны так, что через 30 лет оказалось возможным добавить еще один уровень с шестирядной дорогой. И действительно, в 1962 г. этот уровень был достроен (см. рис. 4.21).

Первоначально второй уровень предназначался для железной дороги. Железная дорога построена не была, но структура моста без каких-либо изменений подошла для автомобильного путепровода.

В противоположность предыдущему примеру со строительством второго уровня рассмотрим пример строительства моста-близнеца рядом со старым мостом! Посмотрите, какой мост построен через залив Чезапик. (См. рис. 4.22.)

Хотя значительное число программ и не требует никакого развития, *большую* часть программ, предназначенных для крупных промышленных приложений или управления сложными операциями, следует либо развивать, либо выбрасывать.

При разработке больших программ, предназначенных для использования в течение 20 лет и более, подавляющая часть всех капиталовложений пришлась на этап продолжающейся разработки.

При разработке больших систем типа V единственной по-настоящему важной проблемой является отсутствие ясных требований. Руководитель проекта не может понять, что ему делать.

Таблица 4.11 Какие требования могут меняться

Продукция	Структура рынка
Размах производства	Разделение труда
Заводы, их расположение	Заказчики
Организация	Налоговое законодательство
Ведущие специалисты	Законы охраны собственности
Число заводов	Обмен продукцией между заводами
Сети	Аппаратура
Процедуры	Следовательно, прикладные программы
Необходимые данные	Следовательно, база данных

Самым первым требованием к большой программной системе должно быть обеспечение легкости внесения изменений.

Первой заботой руководителя разработки программного обеспечения должно быть резервирование средств на выпуск нескольких версий системы.

Резюме

Масштаб, сложность и ясность нужно принимать во внимание и планировать с самого начала работ. Построение системы в миллион команд — это такой же триумф, как и постройка моста, часто стоимости этих объектов тоже сравнимы. Однако программное обеспечение невидимо. Никто не видит его; только некоторые посвященные могут его оценить. В некотором смысле такое положение не очень-то радует. Когда создается что-нибудь достаточно сложное, хотелось бы, чтобы плоды труда были видны. Даже сама работа программы это как бы "верхняя часть айсберга". Нет никакого способа показать огромную сложность и достигнутые результаты. Только в процессе изучения и создания можно оценить сложность и масштаб подобных работ.

Программное обеспечение проекта и программное обеспечение как продукция

Очень полезно различать *программное обеспечение проекта* и *программное обеспечение как продукцию.* Их процессы разработки весьма отличны друг от друга. Программная продукция производится для того, чтобы ее могли использовать во многих местах многие различные и отличные друг от друга пользователи. Программное обеспечение проекта создается для одного, редко нескольких пользователей. Программная продукция в свою очередь тоже имеет три вида:

1. Пакеты программного обеспечения	Программное обеспечение, которое может использоваться на любой вычислительной машине, например транслятор с Кобола
2. Системы с программным обеспечением различных компонент	Получающиеся в результате системы, например системы обработки слов отличаются друг от друга именно своим программным обеспечением
3. Аппаратные комплексы с минимальными программными включениями	Продукция с минимумом программного обеспечения, например копировальная установка, использующая ЭВМ для управления операциями

Продукция и проекты

В разработке обеспечения как продукции есть два пункта (см. также табл. 4.12), отличающих ее от разработки программного обеспечения проектов:

1) процесс определения требований;

2) широта усилий в смысле разнообразия разрабатываемых функций.

Таблица 4.12. Различия между программным обеспечением проекта и продукцией

	Продукция	**Проект**
Число пользователей	Сотни	Один или несколько
Конкуренция	После начала использования	Только за право вести разработку
Продолжающаяся разработка	Критична	Не так критична
Контроль при использовании	Очень Незначительный	Весьма важен

Если мы заставим людей, имеющих опыт разработки программного обеспечения проектов, заняться работой по реализации программ на свободную продажу, это вряд ли приведет к успеху. Неудача нас может ожидать и в обратном случае, при привлечении к работе над проектом тех, кто раньше занимался разработкой товарных программ.

Программная продукция — это программа, которую стремятся продать на широком рынке тысячам или по крайней мере сотням пользователей. Программные проекты имеют значительно более ограниченный круг пользователей. Стандартная программа составления платежных ведомостей, стандартное обеспечение графических дисплеев, транслятор с Кобола, стандартное обеспечение бухгалтерских расчетов, пакеты генератора отчетов — все это примеры товарных программ. Примером программного проекта может служить набор программ, написанных для управления искусственным спутником Земли.

Товарное программное обеспечение разрабатывается со специальной целью удовлетворить требованиям максимально большего числа пользователей. Это оказывает влияние на весь жизненный цикл программ, но в наибольшей степени на начальный этап разработки, на определение требований к системе.

Проще всего убедиться в полезности введения такой классификации, рассмотрев крайние случаи тех и других программ, не вдаваясь в сближающие их детали. У системы "Аполлон", т. е. проекта космических исследований и полета на Луну, был всего один пользователь, при этом этот пользователь жил и работал бок о бок с разработчиком. Между ними был установлен и поддерживался тесный и эффективный канал связи. В противоположность этому обобщенный пакет составления платежных ведомостей предназначен сразу сотням пользователей, большинство из которых никогда не разговаривали и не виделись с разработчиками. Кроме того, разработчики других подобных пакетов предлагают этим потенциальным пользователям свою продукцию. Главным вопросом здесь, следовательно, является не вопрос пригодности нашей программы, а вопрос ее конкурентоспособности. Это различие является определяющим, от него зависят и удачи, и неуспех.

В табл. 4.13 представлены некоторые примеры из сферы программного обеспечения. Там есть и товарные программы, и программные проекты, там же можно найти сведения о продукции, в которой программное обеспечение является только частью общей системы. Внимательное изучение схемы не раз приводило к удивительным результатам: разработчики считали, что они заняты созданием какой-нибудь системы, а потом оказывалось, что они неправильно оценивали, точнее, неправильно понимали, какие усилия нужны будут для завершения их работ.

Таблица 4.13 Характеристики программного обеспечения проектов и программного обеспечения как продукции

Проекты - - - -

<----------- Продукция
----->

	Расчет налоговых отчислений	Хьюстон	Инструментальные программы мин. обороны	Контроль авиаперевозок FAA	Бортовые навигационные системы на кораблях	“Wizard of Avis”	Сопровождение ракет	Телевизоры	Компилятор с Кобола для IBM 370	Обработка текстов
Число пользователей	1	1	1	десятки	сотни	сотни	тысячи	тысячи	тысячи	тысячи
Число разраб. организаций	1	1	1	1	неск.	1	1	1	1	1
Посредники между пользователем и разработчиком	Нет	Нет	Да	Да	Да	Да	Да	Да	Да	Да
Параллельная разработка	Нет	Да	Да	Да	Да	Нет	Да	Да	Нет	Да, затем Нет
Требования пользователя или рынка	Пользователь	Пользователь	Пользователь	Пользователь	Пользователь	Пользователь	Пользователь	Рынок	Рынок	Рынок
Конкуренция после построения программного обеспечения	Нет	Нет	Нет	Нет	Нет	Нет	Нет	Нет	Да	Да
Финансовый риск	Нет	Нет	Нет	Нет	Нет	Нет	Нет	Нет	Да	Да
Сопровождение программного обеспечения		В	В	В	В	В	Н	Н	В	В
								Апп.-инт	Прогр.-инт.	Прогр.-Инт.
								Аппар. прод. с прогр. обесп.	Товарное прогр. обесп.	

Требования к проектам и продукции. Выработать требования к системе “Аполлон” гораздо легче, чем к системе управления процессом обработки слов. Дело заключается в том, что ошибки при определении требований исправлять в первом случае значительно легче. Руководство программой Аполлон имело тесный контакт со всеми, кто будет пользоваться программным обеспечением, и вполне способно управлять ими.

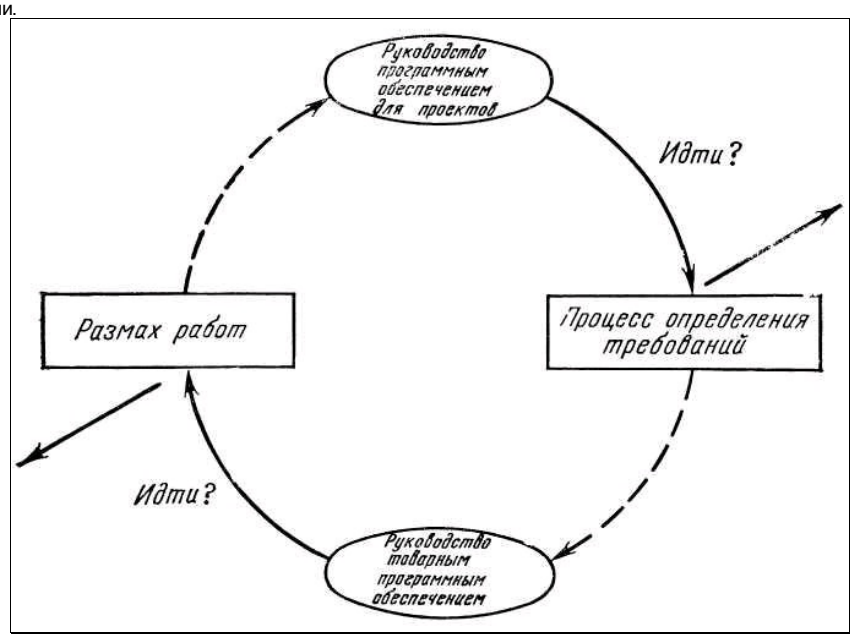


Рис. 4.23. Препятствия на пути руководства программным обеспечением.

Такая тесная связь позволяла легче справляться с некоторыми не очень грубыми ошибками. Если бы о каждой незначительной ошибке или изменении приходилось сообщать сотне или более пользователям, очевидно, что работа по их извещению отняла бы много сил.

Для выработки хорошо составленных требований к разрабатываемому программному обеспечению, предназначенному для продажи многим пользователям, необходимо проводить исчерпывающее исследование их нужд и пожеланий, а затем еще выяснять конъюнктуру рынка. Нам нужно не просто удовлетворить требованиям большого числа пользователей, нам нужно выиграть соревнование с другими системами.

В случае проектов конкурентная борьба разворачивается только в момент выбора группы разработчиков. Определение требований к товарному программному обеспечению — это сложный процесс, которым длительное время вынуждены заниматься большие коллективы людей.

Размах работ при разработке проектов и товарных программ.

В большинстве проектов для построения высоконадежных или высокоскоростных систем приходится разрабатывать уникальные программы. Для многих проектов необходимо создавать специальные средства для их разработки. С другой стороны, условия создания товарных программ носят более ограниченный характер. Продукция обычно более четко очерчена, все интерфейсы, используемые для связи с другими частями программного обеспечения, детально проработаны. Если, например, мне нужно разрабатывать транслятор с Кобола или программу печати платежных ведомостей, то совершенно необязательно начинать с разработки новой операционной системы. Писать команды, образующие интерфейс с операционной системой, безусловно, необходимо, но разрабатывать саму систему нет нужды. Напротив, шансы на разработку операционной системы резко увеличиваются, если поступает задание разработать программу для штурманского дела. (См. рис. 4.23.)

Доведение программы до товарного уровня

Существует много процессов, позволяющих некоторую часть программного обеспечения, работающую всего в одном месте, превратить в “стандартное” программное обеспечение, годное к продаже в сотни организаций. Такой процесс иногда называется доведением программы до товарного уровня. Создание продукции на основе обычной программы походит на процесс инженерного сопровождения, имеющий место при разработке аппаратуры.

Часто рассказываемая с иронией апокрифическая история о группе из трех гениев, за один вечер работы где-то в гараже написавших программу, за разработку которой некая компания обещала потом около 300 тыс. долларов, однако весьма поверхностна. Конечный результат этих двух процессов просто невозможно сравнивать. Чтобы по-настоящему понять эту разницу, давайте предположим, что некто обращается к нам — большой компании — с работающей программой и предлагает продавать эту программу тысячам пользователей и поддерживать ее. Предположим, мы действительно верим, что на этом можно что-то заработать. Составим список того, что нам необходимо будет проделать с этой программой. Список этот, далеко не исчерпывающий, приведен ниже.

— Пересмотреть существующую документацию, если она действительно существует.

Нам необходимо пересмотреть не только конечную документацию, относящуюся к рабочей программе, но и начальную — что делает эта программа, что она не делает, ограничения, инструкции пользователям.

— Создать документацию на продукцию.

Создать рекламную литературу, кратко описывающею нашу продукцию

— *Тщательно протестировать имеющуюся программу.* Проверить граничные условия. Тот факт, что программа работает в одном месте, не означает, что она годится для работы во многих местах. Может, например, так оказаться, что все данные, которые я передаю программе, находятся в некотором ограниченном диапазоне. При использовании на многих вычислительных ма шинах входные данные могут существенно различаться. Мы должны протестировать систему при всех граничных условиях, если они уже определены. Если же нет, мы должны провести тестирование с целью их определения.

— Пример граничных условий

	Входное значение	Предел
Платежная ведомость	Недельный заработок	9999.99 — нуль, отрицательных нет
Радиолокатор	Дальность	от 1000 футов до 99 432 футов

— *Сделать пользовательскую документацию ясной.* Очень маловероятно, что программа, используемая всего в одном месте, имеет достаточную документацию. Все инструкции носят неформальный характер, часто передаются от разработчика пользователю устно. Однако для передачи программы тысяче пользователей мне нужно составить четкую, ясную инструкцию, написанную понятным языком, иначе моя продукция будет отвергнута.

— Сделать рабочую программу ясной. Исследовать программу: убедиться, что в ней нет “никуда не ведущих условных переходов”, “тупиков”, мертвых концов и т.д.; добавить команды “защиты”, необходимые на случай нарушения граничных условий на параметры.

— Модуляризация/расслоение

Для обеспечения эффективной и достаточно дешевой модификации и модернизации особенно необходимо просмотреть всю программу на предмет выделения в ней отдельных модулей и не зависящих друг от друга частей.

— *Создать хорошую документацию для программистов, которые будут проводить сопровождение программы.* Четкая и понятная документация абсолютно необходима для уменьшения стоимости и повышения продуктивности работ по сопровождению программы. Для программ с одним пользователем подобная документация создается в редчайших случаях.

— Протестировать программу со всем системным обеспечением, с которым ей придется работать.

Может случиться так, что товарное программное обеспечение будет работать в различном системном окружении. Перед выпуском продукция должна быть обязательно протестирована.

— Выделить финансовые средства и определить состав группы сопровождения и необходимого для нее оборудования и т.д.

Планирование сопровождения не проводится практически никогда. Основной аргумент: “Мы еще успеем этим заняться”. Ну а пока раскачиваются, возникают невосполнимые убытки. В бюджете должны быть предусмотрены расходы на то, чтобы иметь возможность заменить или исправить программу, если в процессе эксплуатации выясняется, что это нужно сделать.

— Организовать систему “оповещения об ошибках”.

Это просто сказать, но не просто выполнить. Для организации этих работ нужно выделить людей и определить процедуры. Хорошо спланированная и хорошо выполняемая работа может значительно повлиять на мнения пользователей о покупаемом ими программном обеспечении.

— *Организовать систему оповещения об изменениях.* Система оповещения об ошибках целиком относится к работе групп сопровождения. Теперь же нам необходимо создать систему рассылки сведений об изменениях, адресованных *непосредственно* пользователям. Опять-таки весьма *непростая* задача.

При разработке программного обеспечения проектов подобные работы обычно не проводятся; для программного обеспечения как продукции они совершенно необходимы. “Гаражные программы” не могут стать товарными программами!

Программная продукция и продукция, различающаяся по программному обеспечению

Часть программного обеспечения может продаваться на рынке исключительно благодаря тому, что оно программное (т.е. может выполняться на вычислительной машине). Другие же программы неотъемлемы от аппаратуры, в комплексе с которой они продаются. На разработку этих различных видов программного обеспечения затрачиваются разные усилия. Примером системы первого типа является программа печати платежной ведомости. Второй тип программного обеспечения можно себе представить на примере программы обработки слов; эта программа в значительно большей степени, чем аппаратура, влияет на качество всей системы в целом, хотя и неотделима от нее.

Пользователь может не думать о том, на какой машине печатаются платежные ведомости, — т.е. выполняется соответствующая программа, конечно, до тех пор, пока этот процесс выполняется с достаточной степенью надежности. Если вычислительная машина находится на удаленном вычислительном центре, пользователь может даже и не знать, на какой машине и какой модели печатаются его ведомости.

В системах обработки слов аппаратура становится более существенным фактором, поскольку пользователь контактирует именно с ней. Важное значение приобретают характеристики аппаратуры, ее месторасположение, простота использования. Эти факторы отличают одну продукцию от другой. Однако большинство продающихся систем оборудовано практически одинаковой аппаратурой, и различать продукцию можно только по заложенному в них программному обеспечению, а не по этой аппаратуре.

Разработчик продукции, в которой программное обеспечение является ключевым фактором, на каждом шаге своего процесса разработки сталкивается с гораздо большей степенью интеграции аппаратуры и программ, чем руководитель разработки просто программ как продукции.

Продукция с минимальным количеством программного обеспечения

В результате появления микроэлектронных схем мы столкнулись с еще одним видом программного обеспечения, Во многих системах применение цифровых универсальных вычислительных машин оказывается более экономичным, чем использование специализированных схем. В память этих машин приходится записывать команды, а эти команды составляют целые программы.

Включение этого типа программ в нашу таксономию может оказаться весьма полезным. К тому же вопрос о том, как относиться к подобным устройствам и их программному обеспечению, терзает представителей деловых кругов.

Примером такой вычислительной машины, используемой вместо электронной схемы, может служить обычный телевизор, в котором для обработки входного сигнала применяется микропроцессор. В такую машину заносится не так уж много команд, не более двухсот, но затем программа размножается не менее чем для 500 тысяч микросхем, каждая из которых помещается в телевизор. Такое использование программ, программного обеспечения можно называть *аппаратно-интенсивным*.

Важнейшим пунктом при этом является правильность программы. Она должна быть правильной, иначе нам придется изъять у пользователей все 500 тысяч телевизоров и исправлять программу. Экономический эффект такой операции будет просто ошеломляющим. С некоторыми потребительскими товарами уже случались такие катастрофы. Что же будет, если наша микросхема будет позволять хранить все больше команд, оставаясь на прежнем уровне стоимости? Конечно же, инженеры попытаются вставить в микропроцессор более крупные программы. И если мы еще можем быть полностью уверены в правильности программы из двухсот команд, то 100%-ная уверенность в правильности программ из 1000 или 10 000 команд нам недоступна.

Люди, определяющие политику, конечно, видят эту опасность и потому стараются разделить все программное обеспечение на два больших класса — аппаратно-интенсивные программы, которые не должны подвергаться никаким изменениям и, следовательно, не

имеют права на ошибку, и программно-интенсивное программное обеспечение, исправление и модификацию которого мы всегда должны предусматривать.

Для обеспечения второго класса необходимо вводить различные стандарты — стандарты на языки программирования, стандарты на процесс разработки, стандарты на тестирование, на документацию и другие. Подробнее о стандартизации речь пойдет позднее. В аппаратно-интенсивных приложениях все эти стандарты не имеют экономического обоснования. Проведение разработки и использование в рамках жестких стандартных ограничений только увеличат стоимость разработки, причем довольно значительно — в 5—10 раз.

Поэтому нам вполне понятно желание разделить эти два класса. Если хорошо осведомленный руководитель видит и представляет себе все предстоящие затраты на разработку, управление разработкой пройдет четко. Но каким образом руководители предприятий *составляют руководящие технические материалы* для сотен или даже тысяч разработчиков, чтобы добиться обеспечения минимального риска в процессе самой разработки? Вычислительные машины, являющиеся составной частью многих предметов потребления, имеют программы с ошибками, но для их исправления и модификации не создаются никакие планы, не отводятся никакие средства.

Легче всего провести грань между этими столь различными классами программного обеспечения, произвольно установив, что все разработки программ должны вестись с применением методов программно-интенсивных разработок в тех случаях, когда вычислительная машина, на которой будет использоваться данное обеспечение, имеет память размером от 2000 команд и более!

Конечно, это произвольное решение. Существуют примеры достаточно простых программ в 3000 команд, поскольку в них мало условных переходов. Но дело тут в том, что мы пытаемся уберечь ничего не подозревающих создателей аппаратуры от попадания в *ловушку взаимосвязи.* Замечательные новые кристаллы с интегральными схемами действуют на инженеров как призывные песни сирен. Мощность микропроцессоров так сильно возросла, они стали такими дешевыми. Но столь же возросла и цена ошибки.

Мы не знаем, каким должен быть предел емкости памяти —- 2000, или 1000, или 4000. Эти пределы должны определяться руководством конкретных предприятий. Конечно, иногда можно требовать права на исключение. Имеются в виду случаи, когда создатели аппаратуры абсолютно уверены в ясности и стабильности сферы приложения их усилий, а также в своей способности писать безошибочные программы.

Если мы решим, что наша программа не имеет ни одной ошибки и выпустим нашу продукцию в свет, а потом вдруг обнаружим в небольшой программе, всего в 1000 команд, одну-единственную ошибку, мы будем принуждены возвращать себе и вскрывать тысячи микроволновых печей или посылать тысячи техников для замены микросхем в копировальных установках или телевизорах. Стабильность программы крайне важна для всего экономического успеха продукции. Любая ошибка как в определении требований, так и в реализации может оказаться *фатальной для всего дела,* несмотря на то что речь идет всего лишь о “маленькой” программе!

Человек, ответственный за программное обеспечение, прежде всего, конечно же, захочет выяснить, является ли данное приложение аппаратно-интенсивным. К разработке аппаратно-интенсивного программного обеспечения должны применяться иные, менее строгие стандарты.

Использование таксономии — некоторые примеры

Однажды меня попросили перечислить десять ведущих специалистов Соединенных Штатов по программному обеспечению. Я разделил свой список на три части. В первый список попали люди, действительно знающие, как надо строить, и построившие большие системы программ.

Во второй список я поместил имена тех, кто обладает обширными знаниями в области языков программирования и библиотек программ, т.е. в области инструментального обеспечения. Сюда попали многие деятели из различных университетов страны.

Третий список, однако, относился не к специалистам в области прикладных программ, а к тем, кто занимается построением обеспечения проектов. Прикладная область не представляет собой ничего сложного. Те же, кто разрабатывает программное обеспечение проектов, используют продукцию первых двух групп, соединяют их с прикладной областью и создают окончательную работающую систему. Они в прямом смысле интегрируют программное обеспечение, создавая самые большие, самые быстрые, самые сложные системы. Они построили систему управления воздушным транспортом, спутниками Земли, системы резервирования авиабилетов, военные командные и управляющие системы. Они создавали системные программы, объединяли их с прикладными, основывая свой труд на множестве инструментальных программ. Они руководили разработкой 100 млн. систем.

Наиболее важный вывод, следующий из этой классификации, состоит в том, что для каждой из этих областей необходимы различные таланты и мастерство! *Никогда* не придет мне в голову назначить специалиста по языкам программирования ответственным за разработку большой системы реального времени; он не компетентен в этой области. И наоборот, нельзя сделать ответственными за создание системы противовоздушной обороны даже высококлассных системных программистов! При выполнении же работ, относящихся к первым двум классам, нельзя использовать специалистов по сложным системам реального времени. *Все это совершенно разные работы.*

Проект с товарными программами

Система общенациональной связи, предусматривающая также обслуживание, является огромным программным обеспечением проекта, но кроме того, требует создания также еще и многих товарных программ Система передачи сообщений может соединять конторы клиентов и передавать данные с некоторого устройства (например, копировальной установки) одной конторы на другое устройство, находящееся в другом месте.

Для обеспечения работоспособности такой системы необходимо создать и заставить работать сеть средств передачи информации (микроволновые передатчики, спутники и т.д.), и средства хранения информации, и коммутационные устройства Обслуживать систему должна небольшая группа людей, которые будут управлять средствами поддержки отдельных узлов сети и регулировать работу системы при возникновении переполнений.

Большая часть сети должна управляться с помощью вычислительных машин. Программы, которые нужно для этого написать и связать между собой, относятся к классу *программного обеспечения проектов.* Однако очевидно, что средства, предлагаемые пользователям, будут включать также средства радиосвязи. Оператор копировальной установки, включенной в состав сети, может посылать копии своих документов в десятки различных мест. Если имеемся некий постоянный набор клиентов, часто получающих одни и те же копии, оператор может не перечислять их, а адресоваться к группе. Сеть — с помощью соответствующих программ — будет переводить введенный оператором *идентификатор группы* в фактические адреса. Эта программа — просто автономная программа, продукция, а не часть обеспечения проекта.

Такая продукция

— должна быть рассчитана на огромное число пользователей; тысячи этих пользователей могут работать в различных областях и отраслях;

— должна быть легкой в использовании, хорошо защищенной от случайных воздействий, хорошо приспособленной к возможным добавкам или исключениям;

— должна быть очень хорошо документирована;

— должна выполнять большое число функций, что поможет ей удовлетворить большое число пользователей;

— должна сопровождаться людьми, которые *готовы* исправить ее и внести в нее дополнения;

— богатый набор ее функций должен быть перед реализацией тщательно выверен и определен;

— соревнование за пользователей начнется для нее после введения в эксплуатацию; конкурентами будут другие сети.

Таблица 4.14. Примеры программ и их соответствие классификации

	Ведомость	Расчеты орбит	Управление информационной системой	Система резервирования мест при авиаперевозках	Транслятор с Кобола	Программы сортировки и квадратного корня	Автоматизация конторского дела	Система ведения конторской документации	Телевизионный приемник	Система управления базой данных	Управление воздушным транспортом
Время разработки	Н	Н	В	В	В	С	В	С	С	В	ОВ
Время использования или выполнения	Н	Н	В	В	Н	Н	В	Н	Н	В	ОВ
Время	Н	Н	В	В	Н	Н	В	Н	Н	В	ОВ

сопровождения	Н	Н	В	В	Н	Н	В	Н	Н	В	ОВ
Прикладное	V	V	V	V		*	V	V	V		V
Вспомогательное					V						
Системное						*				V	
Масштаб	М	М	С	В	М	М	С	М	М	С	В
Сложность											
Научная	Н	В	Н	Н	Н	С	Н	Н	Н	Н	Н
Логическая	Н-С	С	В	ОВ	В	С	В	С	Н	В	В
Обеспечение проекта	*	V	V	V		*					V
Программная продукция					V	*	V	V		V	
Продукция с аппаратно-интенсивным обеспечением								V	V		
Продукция с программно-интенсивным обеспечением					V		V			V	

V — обычно присутствует, * — может быть несколько вариантов, Н — низкий, С — средний, В — высокий, ОВ — очень высокий.

Таким образом, для разработки такой сети по обслуживанию деловых операций необходимо разрабатывать как обеспечение для проектов, так и отдельные программы как продукцию.

В табл. 4.14 приведены примеры различных программ и их распределение по категориям в соответствии с таксономией. В дополнение к этому дается грубая оценка трудности работы с данной программой на каждом из трех этапов жизненного цикла программы.

Стоимость программного обеспечения

В гл.2 мы уже отмечали, что стоимость больших программ может достигать 90% общей стоимости проекта.

Существуют и такие применения программного обеспечения, в которых стоимость программ неизмеримо мала по сравнению со стоимостью аппаратуры. Разработчики таких программ могут совершенно не замечать кризиса программного обеспечения.

Если от нас требуют разработать основную программу, используемую на одной и только одной вычислительной машине, например программу управления информационной системой для компании XYZ мы будем не вправе делить стоимость программы на число пользователей. К тому же число это равно 1, и вся стоимость накладывается на одного пользователя. Если же нам нужно вставить нашу программ) в маленькую вычислительную машину внутри телевизора, мы разделим стоимость программ, написанных для этой машины, на общее число телевизоров, которые будут выпущены в эксплуатацию. Если разработка программ обошлась в 200 тыс. долларов, то после деления на 500 тысяч телевизоров получим удельную стоимость всего 40 центов. Все очень просто! О чем же волноваться? Пока программа правильна, никаких проблем и не существует. В табл. 4.15 представлено все многообразие “тиражей” различных вычислительных машин.

При разработке системного и прикладного обеспечения в рамках некоторого проекта, скажем для управляющего центра общенациональной системы связи, стоимость этого обеспечения может легко достичь 30 или даже 60 млн. долларов в расчете на одну станцию связи. Стоимость же программ для одного микропроцессора, встроенного в телевизор, как мы уже подсчитали, может оказаться в диапазоне 40 центов на штуку.

Словарь программного обеспечения

Если мы говорим о программном обеспечении и не употребляем рядом с этим термином никаких добавочных определений, то должны четко понимать, что при этом мы все очень сильно упрощаем. Лишь самые общие замечания, сделанные по поводу аппаратно-интенсивных приложений программного обеспечения, можно отнести также и на счет программно-интенсивных приложений, и наоборот.

Программное обеспечение	Системное обеспечение	Обеспечение как продукция	Обеспечение проектов	Инструментальное обеспечение	Тестирующее обеспечение	Большое обеспечение	Обеспечение для реального времени	Пакетное обеспечение	Прикладное обеспечение	Диалоговое обеспечение	и т.д.

Мы не должны допускать предложений типа “70% стоимости программного обеспечения приходится на этап продолжающейся разработки”. Во многих случаях это правда; во многих — нет. Это слишком общее высказывание. Мы должны добиваться большей точности, использовать уточняющие определения. Люди, не являющиеся специалистами в области вычислительной техники, не только введены в заблуждение относительно вычислительных машин и их программного обеспечения, но также возмущены жаргоном и небрежностью, царящими в нашей отрасли[1]). **Если профессионал не настаивает на точности высказываний, является ли он подлинным профессионалом?**

[1]) Следует отметить, что еще с большим основанием это утверждение относится и к положению с соответствующей терминологией у нас. — Прим. ред.

ГЛАВА 5. РАЗРАБОТКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Разделив все программное обеспечение на категории,, изучим теперь подробнее процесс его разработки. Именно эта область доставляет в настоящее время наибольшие неприятности.

Свойства, присущие всем программам, дают нам возможность глубже проникнуть в более тонкие и более важные характеристики программ. Имеется по крайней мере 12 таких свойств или характеристик. Взгляните на табл. 5.1.

Таблица 5.1

Характеристики программ	Обычно носят название
1. Заставляет вычислительную машину выполнить задание	Функция
2. Занимает память машины	Размер
3. Тратит ресурсы ЦП	Эффективность
4. Легкость использования	Практичность
5. Легкость восстановления после сбоя	Устойчивость; восстанавливаемость
6. Содержит ошибки	Правильность
7. Требует времени для создания	График разработки
8. Требует людей для создания	Людские ресурсы на разработку
9. Требует специального инструментария для разработки	Материальные ресурсы на разработку
10. Требует денег для создания	Стоимость
11. Модифицируема	Архитектура, структура
12. Существует по крайней мере в одной форме, а нужны две	Документация

Характеристики программ

Стремление воплотить эти характеристики в программу приводит к конфликтам. В программах, написанных очень быстро (характеристика 7), обычно крайне неэкономно используется память и машинное время (характеристики 2 и 3) по сравнению с программами, которые писались медленнее. Быстро написанные программы часто не выполняют на все 100% функции, которые они, по предположению, должны были выполнять (1). Программа может печатать почти любую ведомость, однако нам все же приходится держать двух сотрудников для выполнения функций, которые должна была бы (и могла бы) выполнять машина, но которые не были вовремя запрограммированы. Такое случается очень часто.

Если программа печати ведомости нужна нам к 1 июня будущего года, мы можем на этот срок включить дополнительно в группу разработки нескольких программистов, чтобы они помогли сократить время работы. Очевидно, что если на создание программы требуется 100 чел.-мес., то мы могли бы выделить на ее создание 10 программистов, которые будут работать в течение 10 мес. Но сделать эту же работу за один месяц, бросив на нее 100 человек, невозможно

Укороченные графики могут совершенно разрушить попытки построить стройную, легко модифицируемую структуру (11). Малая стоимость разработки (10) находится в явном противоречии со всеми другими аспектами. (См. табл. 5.2.)

Таблица 5.2. Двенадцать характеристик программы

Фаза разработки	Требует времени для создания	График разработки	Требует людей для создания	Группа разработки	Требует специального инструментария для создания	Средства разработки	Требует денег для создания	Стоимость			
Фаза использования	Заставляет машину выполнить задание	Функция	Занимает память машины	Размер	Использует ресурсы ЦП	Эффективность	Легка для использования	Практичность	Легко восстанавливается до рабочего состояния	Восстанавливаемость	
	Содержит ошибки	Правильность									
Фаза сопровождения	Модифицируема	Архитектура	Существует по крайней мере водной форме, а нужны две	Документация							

После того как все разработчики представят свои оценки еще не испробованных возможных вариантов, руководство должно принять компромиссное решение.

Давайте посмотрим, какие характеристики существуют в фазе использования.

1. Каждая программа выполняет некоторую функцию, например, она может составлять платежную ведомость.

Здесь мы еще не можем давать численных оценок, но после некоторого наблюдения мы можем определить, действительно ли программа выполняет ту функцию, которая нам нужна.

2. Каждая программа при выполнении занимает некоторое место в памяти. Если памяти у машины мало, нам придется затратить время и людские ресурсы на то, чтобы сжать программу до нужных размеров.

3. При выполнении каждая программа использует некоторые ресурсы машины. Например, сколь быстро напечатает машина нужную нам ведомость по данной программе? Если наш центральный процессор работает медленно, разработчикам придется потрудиться над программой, чтобы выжать из процессора все, что только воз можно.

4. Легкость использования. Некоторые программы весьма успешно отражают всякие попытки воспользоваться ими. Они не “дружественны пользователю”. Легкость использования — это не случайно возникающее побочное следствие разработки, а качество, требующее тщательного проектирования и разработки специальных требований.

5. Чем больше программа, тем больше вероятность, что в ней имеются ошибки и что самое строгое тестирование не обнаружит их полностью. Большинство этих ошибок может обнаружиться только в необычных комбинациях данных и пользователей.

6. Каждая программа должна сохранять работоспособность. Некоторые из них перезапустить легко, другие сложнее. И опять эта характеристика является объектом тщательного проектирования.

Теперь рассмотрим характеристики фазы разработки.

7. Каждая программа требует на разработку некоторого времени. При разработке больших систем этого времени часто не хватает. И часто *именно* время является определяющим критерием, доминирующим над всеми другими характеристиками.

8 Каждая программа создается некоторым числом программистов, работающих в течение некоторого времени (чел.-мес.).

9. Для создания любой программы нужны какие-нибудь инструменты. Ими могут быть вычислительные машины, производящие трансляцию с языков высокого уровня и тестирование, перфокарты, магнитные ленты, “пространство”, программы для трансляции, тестирования и еще многое другое.

10. Для создания каждой программы необходимы денежные средства.

Теперь обратимся к характеристикам фазы сопровождения.

11. При возникновении новых требований их приходится отражать в программе. Программа, сработанная на скорую руку, труднее поддается модификации, чем программа, проектирование которой велось аккуратно и без спешки. Хорошо ли спроектировано, насколько модульно построено программное обеспечение?

12. Программы так часто хранятся только на магнитных лентах, что упоминание об этом стало уже избитой фразой. Бедняги, которых заставили исправлять в таких про граммах ошибки и расширять их возможности, должны проводить подлинные археологические раскопки, чтобы понять, что же *пытался* сделать основной разработчик. Нам необходима четкая документация!

Теперь попытаемся представить все эти характеристики программ на диаграмме, иллюстрирующей жизненный цикл программы (см. рис. 5.1).

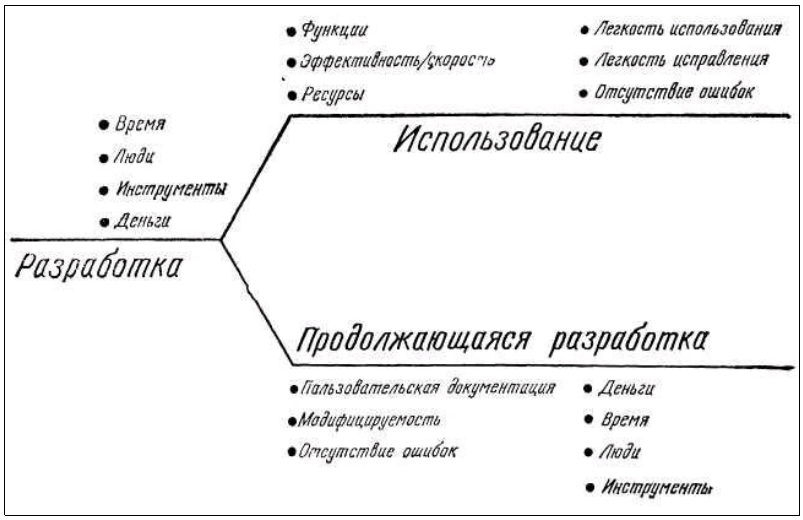


Рис. 5.1. Фазы жизненного цикла и 12 характеристик программы.

Заметьте, что четыре аспекта разработки относятся также и к продолжающейся разработке!

Модифицируемость *практически никогда не принимается во внимание,* в результате группы сопровождения вынуждены модифицировать “бетонные блоки”! И часто без всякой документации.

Многие программы и не требуется развивать. Многие программы не нужно делать дружественными пользователю. И нам не нужно усложнять разработку таких программ требованиями, выполнение которых необязательно. Но от нашего руководства мы вправе ожидать сознательного отказа от этих требований и четкого обоснования принятого им *решения*.

Жизнь каждого изделия или инструмента проходит несколько этапов. Все инструменты проходят по крайней мере два этапа — разработку и использование. Некоторые, большая часть, имеет и третью — техническое обслуживание и совершенствование.

Простая деревянная *зубочистка,* как и молоток, упомянутый в гл.4, проходит всего два этапа. Вы ее делаете, используете и выбрасываете. Никакого ремонта не производится.

Существуют и программы типа “зубочистки”, которые я пишу, выполняю, получаю удовлетворяющий меня результат и выбрасываю.

Молотки делают из металла. Мы делаем их и используем. Ремонт им не нужен, но пользуемся мы ими неоднократно.

Существуют программы типа “молотка”. Их пишут, их доводят до работающего состояния и исполняют очень много раз. Как и молотки, они не требуют никаких модификаций, изменений или исправлений.

Существуют программы типа “небоскреба”, которые, как и те здания, в которых мы работаем, требуют некоторого дополнительного сопровождения. Однажды мы обнаруживаем, что в нашем здании забиты не все гвозди и завернуты не все болты, поэтому нам надо внести некоторые коррективы. Большинство программ относится к типу “небоскребов”, так как мы часто добавляем к нашим “зданиям” новые “пристройки”.

Мы обратились к этим трем примерам постольку, поскольку они дают нам повод поговорить о качестве программного обеспечения Качество зубочисток необходимо только при их использовании. Для программ типа “зубочисток” нам не приходится заботиться о компоновке — до тех пор пока они работают на нас,— нам не нужно думать об их документировании, раз мы собираемся использовать их только однажды, а затем выбросить.

Качество молотка нас заботит больше, поскольку нам приходится использовать его продолжительное время. Не так уж много молотков ломается при первом же ударе, но плохо изготовленный молоток сломается гораздо быстрее, чем молоток, изготовленный из более хорошего материала и более старательно.

Так же обстоит дело и с *программами типа “молотка”.* Чем дольше мы собираемся ими пользоваться, тем больше старания и затрат (имеются в виду прежде всего деньги) мы вкладываем в их изготовление, надеясь, что эти вложения оправдаются на этапе использования программ типа “молотка”.

Заметьте, однако, что для этих программ мы не предусматриваем этапа модификаций (обслуживания). Поэтому *документация* необходима только для использования программ. До тех пор пока у нас есть хотя бы одна копия программы, написанная в форме, подходящей для ввода в машину, никакая документация на эту программу нам не нужна. Если программа правильна и не требует никаких изменений, все, что нам будет нужно, это инструкции, обращенные к пользователям.

Обращаясь теперь к программам типа “небоскребов”, сразу же заметим, что если нам нужно добавить к небоскребу дополнительные помещения, то для оценки, планирования и выполнения всех работ пригодятся любые сведения о проложенных в здании водопроводных трубах, балках, электрических соединениях, проводах и еще о массе различных вещей. Нам будет нужна не пользовательская документация, а сведения о деталях конструкции, которые могут облегчить нам процесс перестройки.

Единственной причиной, по которой нам необходима документация на нашу программу типа “небоскреба”, является то, что нам известно о предстоящих в будущем изменениях, а документация сильно облегчает этот процесс.

Это наиболее интересное утверждение; давайте обсудим его подробнее.

Если сформулировать это утверждение в негативной форме, то оно будет выглядеть так: если мы абсолютно уверены в том, что нам никогда не потребуется ни изменять, ни исправлять работающую для нас программу, то в виде, пригодном для чтения ее человеком, она нам никогда не понадобится.

Если бы мы были обладателями такой программы (а в настоящее время в мире достаточное число таких обладателей), мы бы просто загрузили эту программу в память машины, выполнили бы ее, получили бы результаты и были бы ими довольны.

Конечно, временами мы бы волновались из-за того, что в программе могут быть скрытые дефекты или ошибки, или из-за того, что окружение программы меняется, и нам необходимо либо разбираться в структуре программы, либо писать полностью новую программу. Но, если все эти ужасы минуют нас стороной, пока мы находимся “у руля”, мы можем сказать, что абсолютно удовлетворены работой нашей программы. Она выполнила свое дело, а мы этим остались довольны.

В настоящее время большинство программ в мире являются программами типа “небоскреба”; в них есть ошибки и недоработки, они нуждаются в модификации, чтобы не отставать от постоянно меняющегося окружения. Следовательно, как и проектировщикам, людям, проводящим модернизацию, необходимы синьки со схемами и документы, помогающие разобраться в причинах ошибок и исправлять их.

В здании, построенном согласно утвердившейся практике, обязательно должны быть предусмотрены вполне определенные коммуникации. Мы можем на это рассчитывать, а это облегчит нашу задачу перестройки.

Так же обстоит дело и с программным обеспечением. Если программисты и проектировщики исходной версии программы следовали “отработанным правилам”, наша задача весьма и весьма облегчается. Даже сильнее, чем в строительстве, поскольку мы имеем дело с неосязаемыми вещами.

Процесс разработки программного обеспечения

Существуют различные способы разделения процесса разработки программного обеспечения на этапы. При некоторых из них выделяют большее количество этапов, при других — меньшее. По-видимому, неизбежными являются шесть этапов. Разработка программного обеспечения: Определение требований Проектирование

Написание команд — программы Компоновка Тестирование Документирование

Первый род деятельности, определение требований, представляет особую сложность для больших систем типа V, и вскоре мы его рассмотрим весьма подробно.

Проектирование, имеющееся в виду здесь, есть проектирование именно программ, а не той системы в целом, частью которой являются эти программы. Более глубокое изучение этого процесса мы проведем после того, как рассмотрим определение требований.

Третьим пунктом является написание команд, сведение проекта программного обеспечения или просто программы к последовательности машинных команд. Мы будем называть этот процесс программированием.

Рис. 5.2. Идеальный случай разработки программного обеспечения.

Компоновка представляет собой комбинирование, связывание отдельных частей программы, написанных разными людьми или группами, в одну большую систему программного обеспечения. Если в нашем распоряжении находятся 700 программистов, создающие отдельные программы, то задача “объединения” этих программ в одну систему может стать просто огромной, требующей много времени, дорогой и чреватой многими ошибками. Мы в дальнейшем увидим, что для компоновки систем из отдельных заранее написанных программ нужно создавать специальные инструментальные средства.

Тестирование, или, как его сейчас часто называют, верификация, — это весьма важный и сложный род деятельности.

Как мы уже знаем, документация особенно важна на этапе продолжающейся разработки.

Полный цикл

Давая поэтапный список процессов как это сделано на рис. 5.2, мы должны понимать, что реально он никогда не бывает таким простым и понятным.

Использование

Очевидно, что только простейшие задачи проходят все шаги без каких-либо итераций, т.е. постоянных возвратов на более ранние этапы процесса. При проектировании кто-то может обнаружить, что следование какому-нибудь требованию может привести к двукратному увеличению стоимости разработки подсистемы. Разработчик должен пересмотреть и переоценить требования. Этот процесс продолжается непрерывно. Группе проектировщиков передается новая информация, проект должен быть пересмотрен. (См. рис. 5.3.)

В конце 60-х гг. в Гейтсбурге в отделении фирмы IBM мы создали специальный курс лекций под названием КУПП — курс управления программным проектом. Этот курс был предназначен для того, чтобы молодые руководители работ лучше понимали, на что обращать особое внимание, как привести проект к оптимальному результату (см. рис. 5.4.) Материал мало менялся с годами, я здесь привожу две диаграммы распределения людских ресурсов, использовавшихся на протяжении 5 лет.

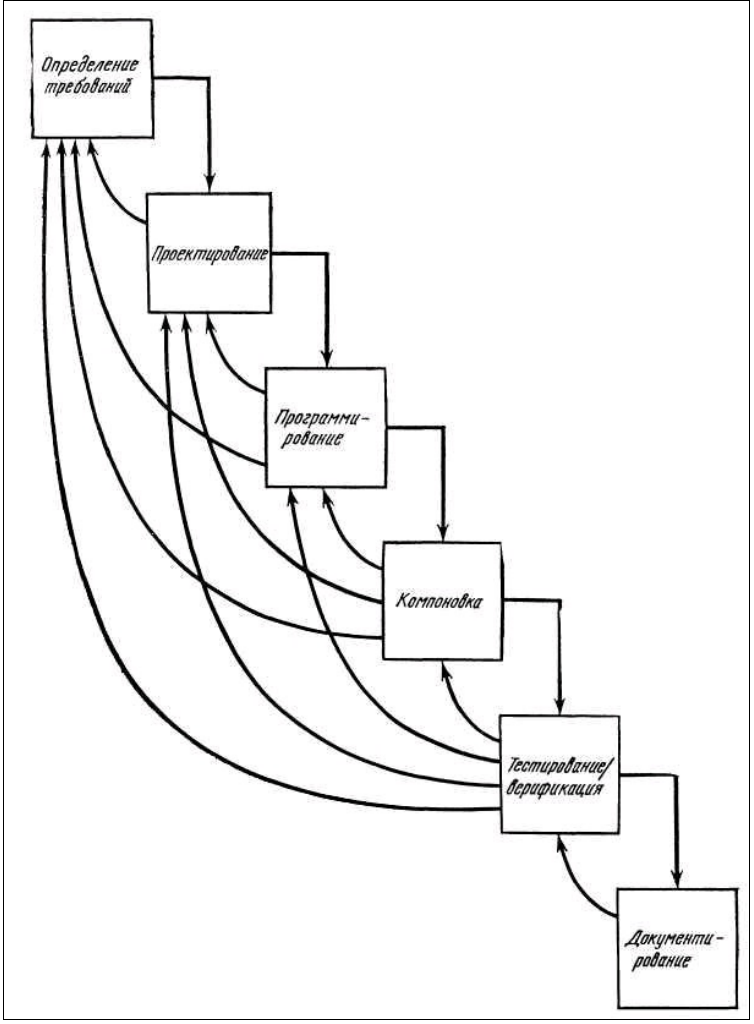


Рис. 5.3. Реальный ход разработки программного обеспечения.

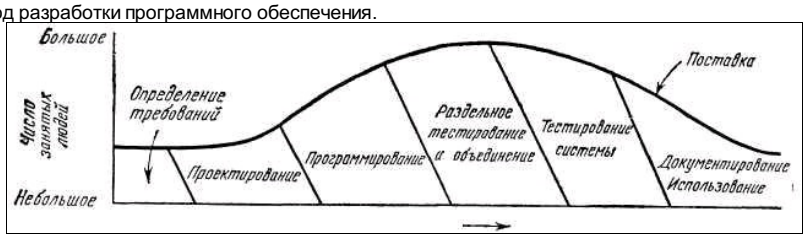


Рис. 5.4. Диаграмма распределения людских ресурсов по данным 1970 г.

Рис. 5.4 был в конце концов признан неправильным. Для больших проектов проектирование не кончается никогда. Диаграмма была изменена таким образом, чтобы отразить продолжающееся в течение всего этапа разработки проектирование (см. рис. 5.5).

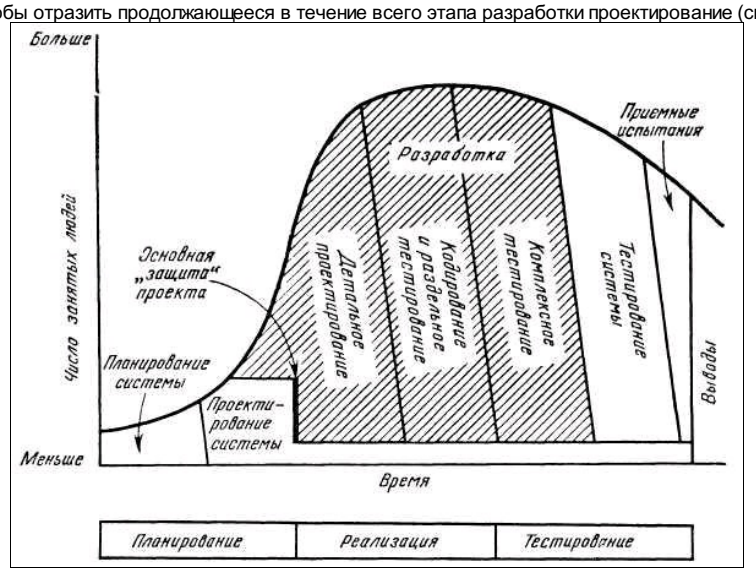


Рис. 5.5. Диаграмма распределения людских ресурсов (по данным 1975 г)

Нам никогда бы не пришло в голову, что деятельность по определению требований, как и проектирование, могла бы продолжаться в течение всего хода работ по разработке. Как показано на рисунках, число занятых людей резко уменьшается после сдачи системы. Такого не случается только с системами типа V. Причина заключается в том, что “закон обязательной даты” — необходимость сдать работу в срок, приводит к временному откладыванию реализации многих обещанных и запланированных функций. Их приходится вводить в действие после “сдачи”.

Как мы увидим, распределение времени по различным этапам зависит от множества факторов. Однако размер программной системы, зависящий только от того, какие функции она должна выполнять, является одним из основных определяющих факторов. Время, затрачиваемое на написание программ, сокращается по отношению ко времени и усилиям, затраченным на весь проект в целом, по мере увеличения проекта.

Эти ошибочные диаграммы укомплектования персоналом очень живучи. Диаграмма, изображенная на рис. 5.6, появилась в ведущем журнале отрасли в 1979 г.

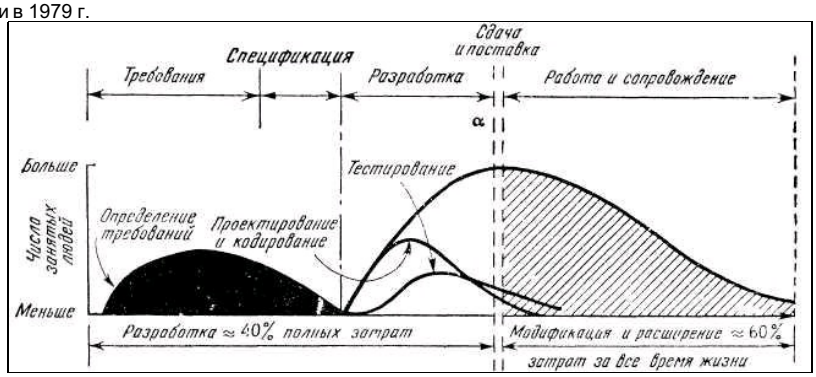


Рис. 5.6. Распределение людских ресурсов для программ типов I и II.

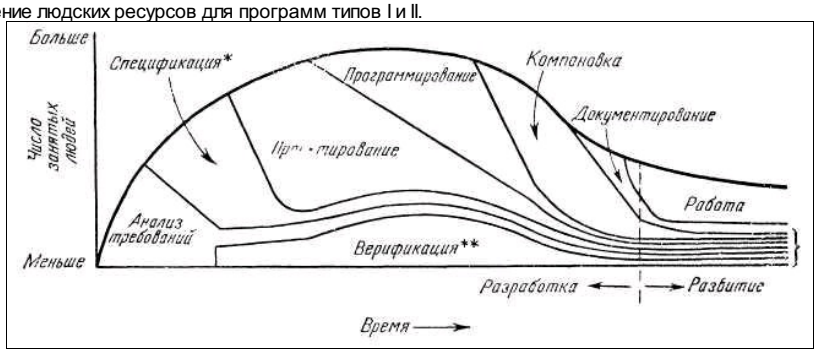


Рис. 5.7. Распределение людских ресурсов при разработке программного обеспечения.

Для маленьких программ в хорошо известных уже полностью автоматизированных областях эта диаграмма вполне пригодна. Но она совершенно *не подходит* большим программным системам и даже системам небольшого размера, относящимся к области управления процессами! Определение требований и проектирование продолжаются гораздо дольше.

Рис. 5.7, созданный Э.Ферентино, полнее соответствует реальной ситуации, или, лучше сказать, ситуации, которая должна возникать при разработке крупномасштабного программного обеспечения.

Еще раз взгляните на модель, использованную нами в отделении федеральных систем фирмы IBM в начале 70-х гг. (рис. 5.5). Процесс проектирования, как мы уже видели, отражен вполне удовлетворительно, но совершенно неправильно представлена роль тестирования. Показано, что тестирование начинается только вместе с кодированием.

Как можно видеть из правильной модели, тестирование должно начинаться вскоре после "первого прохода" процесса определения требований.

"Большой взрыв" и эволюция

Первые из приведенных выше диаграмм отражают подход, называемый теорией "большого взрыва". Этот подход строится на предположениях, что разработчики хорошо знакомы со всеми требованиями, что требования не меняются и что можно спроектировать достаточно эффективную систему, удовлетворяющую этим требованиям. Такое, однако, случается только для простых программ типов I и II, причем только небольших и средних размеров.

Для программ типов III, IV и V дело осложняется тем, что разработчик не вполне понимает все требования. Когда *пользователь* получает *первую версию* системы, он говорит: "Ага, посмотрим, что с этим можно сделать", после чего возникает целый поток новых требований! Никто не в силах предугадать, каким образом пользователь будет *в действительности* применять даваемый ему новый инструмент. Та скорость, с которой разработчик ответит второй версией системы на новые требования, зависит от огромного множества различных обстоятельств, среди которых немаловажную роль играет то, насколько разработчик может контролировать пользователя, иными словами, что же мы создаем — программу как продукцию или программное обеспечение проекта.

Мы уже отмечали, что к моменту первичной сдачи редко завершаются работы по реализации всех необходимых функций. Недоделки устраняются *после* первого предъявления системы к сдаче.

Необходимо при этом отметить, что кривая, отмечающая количество используемого персонала, после первичной сдачи не должна сразу же резко снижаться! Это и не происходит в действительности! На самом деле строится несколько версий системы, каждая из которых содержит все больше функций, все меньше ошибок, в более поздних версиях все с большей полнотой отражается опыт пользователя.

Такая практика долгосрочного планирования и финансирования разработок редко применяется в больших организациях. "Социология" приемки новых систем, возможно, в некоторой степени объясняет такое положение, но чаще причиной ошибочного выбора методики большого взрыва является полное отсутствие опыта.

"Социология"? Ну конечно, ведь поддержание на высоком уровне количества занятых людей в течение многих лет ведет к таким затратам, что проект может быть отвергнут. Поэтому и выбирают график вида большого взрыва. Для небольших приложений типов I и II такая схема привлечения персонала работает вполне удовлетворительно. В других условиях использование большого взрыва — это чистейшей воды афера, причем совершенно бессмысленная! (См. рис. 5.8.)

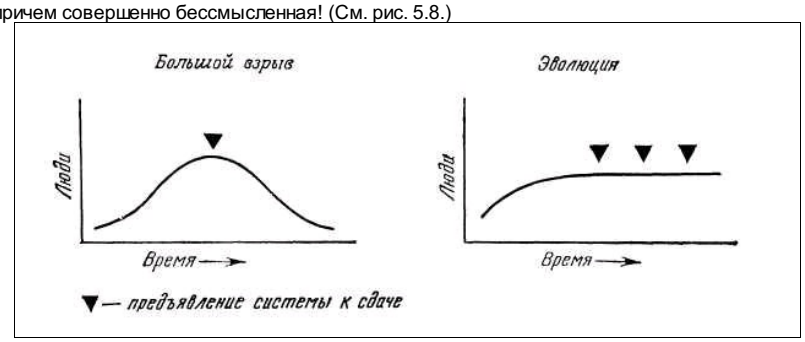


Рис. 5.8. Планирование занятости при методике большого взрыва и при эволюционном подходе к разработке.

Все, что мы говорили до сих пор, ни в малейшей степени не объясняет те огромные затраты людских ресурсов, которые необходимы для *создания* больших последовательностей команд — программного обеспечения. Теперь подробнее изучим процесс создания и разработки системы из миллиона команд.

Для начала обратимся к конечному результату — что представляет собой программа из миллиона команд? Никто никогда такой программы не видел. Если мы напишем на бумаге миллион строк по 4 команды на 1 см, мы получим рулон бумаги длиной в 250 000 см. Это составляет 2,5 км. бумаги.

Чтобы представить себе миллион команд, представим себе лишь отдельные части программы и их взаимоотношения. Способы представления таких программ будут вкратце описаны на с.155.

Для того чтобы поверить в истинные масштабы затрат на разработку, их иногда необходимо просто увидеть. Десяток, а иногда и несколько десятков людей работают, чтобы выработать требования, установить, что же должна делать система!

Первичное проектирование ведется меньшим количеством людей, но зато написание программы, программирование должно выполняться сотнями, а иногда тысячами программистов. И опять не меньше десятка людей требуется для компоновки, а для тестирования необходимо привлечь несколько десятков, а то и сотен людей. Документирование работ также требует не одного десятка людей. (См. рис. 5.9.)

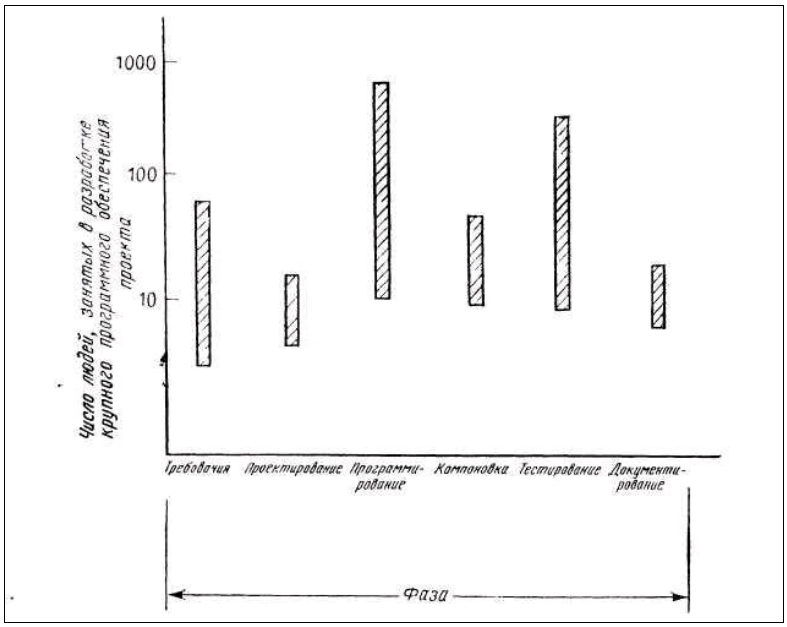


Рис. 5.9. Занятость в различных фазах разработки.

Определение требований

Этот шаг является важнейшим среди всех шести этапов процесса разработки. Он влияет на все остальные этапы. Увы, это наименее изученный и наименее понятный процесс.

Процесс определения требований разбивается по крайней мере на две части. Во-первых, нужно понять, что нужно сделать, а во-вторых, нужно документировать это. Без этой второй части большой проект останется неуправляемым. *Гели требования не записаны и не сделаны доступными разработчикам, они вроде бы и не существуют.* То, что они существуют в голове какого-нибудь гении, все равно не спасает проект от неожиданной катастрофы.

Требования системного уровня

Определение требований это проблема системного уровня, которая постепенно спускается на уровень программного обеспечения. Если взглянуть на типичную картину большого проекта по управлению процессами типа V, мы увидим что-то похожее на рис. 5.10.

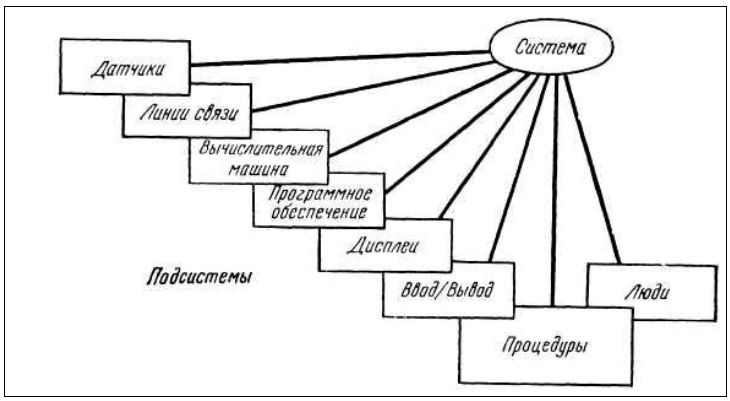


Рис. 5.10 Система и ее подсистемы.

Изменения неизбежны

Требования системного уровня определяют требования для отдельных подсистем. По мере развития работ затраты, возникающие изменения, ограничения, возможные прорывы в подсистемах будут оказывать обратное воздействие на требования системного уровня, делая весь процесс итеративным.

Такие итерации в наибольшей степени отражаются на требованиях к программному обеспечению, поскольку именно оно является последней регулируемой инстанцией, с помощью которой руководство проектом может исправлять всякие несогласованности. Аппаратура — вычислительные машины, радиолокаторы, дисплеи, средства хранения информации, модемы — могут быть исправлены только после нескольких лет напряженного труда. К счастью, программное обеспечение может быть модернизировано в соответствии с новой ситуацией всего за несколько месяцев!

Лос-Анжелесская группа из 500 разработчиков под моим руководством делала систему управления искусственным спутником. Спутник вышел на неправильную, нерегулярную траекторию. Некоторые приборы выполняли не те функции, для которых они были спроектированы, другие же не работали совсем. Ценой огромных усилий мои сотрудники сумели модифицировать программное обеспечение таким образом, чтобы задание все же было выполнено.

И вдруг мы обнаруживаем, что наш заказчик крайне расстроен тем обстоятельством, что нам пришлось выйти из бюджета! Мы терпеливо объясняли ему, что наш перерасход в 3 млн. долларов сэкономил ему несколько лет труда. Кроме того, мы спасли его от необходимости запускать новый спутник, что обошлось бы ему в 50 млн. Нас оправдали, но неохотно!

Программное обеспечение всегда становится последней инстанцией; за его счет исправляют недостатки всех остальных подсистем. Оно получает в наследство чужие промахи и ошибки.

Определение требований для систем типов I и II — достаточно простой процесс, практически всегда он бывает сделан достаточно хорошо. Этого нельзя сказать о больших системах типов III, IV и V. Новизна вычислительной техники комбинируется с трудностью определения требований для больших систем, которые прежде никогда не строились. Люди никогда раньше не применяли автоматизированные системы в этих областях, поэтому они не сразу могут понять все возможности этих новых для них систем.

В дополнение к этой неизвестности действуют еще два фактора. Во-первых, окружение, в котором будет работать система, может измениться еще до того, как система будет введена в действие. Во-вторых, может измениться сама система. Различные ее подсистемы могут оказаться совсем другими, чем предполагалось до ее сдачи.

Таблица 5.3. Изменения в системах реального времени

Не влияют на системы	Влияют на системы

Число пользователей	Спутник
Стратегия	Радиолокатор
Тактика	Датчики
Структура организации пользователей	Дисплеи
	Навигационное оборудование
Режим работы пользователя	Стартовая площадка
Приоритеты	Ракета
Сети связи	Интеллектуальные терминалы
Стратегия принятия решения	
Угроза	

В действительности лучше было бы задать такой вопрос: “Что же не будет меняться?”. Ответ поможет нам определить самое первое требование для больших систем программного обеспечения, которое мы уже проводили на с.103.

Самое первое требование к проектированию больших систем программного обеспечения — предусмотреть возможность будущих изменений!

О том, как этого добиться, мы поговорим в разделе книги, посвященном проектированию программ.

Кто формулирует требования к программному обеспечению?

Каким же образом мы приходим к первой формулировке требований? Я говорю “первой”, поскольку мы знаем, что этот процесс будет повторен несколько раз по мере перехода от разработки требований к проектированию и обратно к выработке новых требований.

Кому пристало формулировать требования? Конечно, пользователю, а также проектировщику системы. Формулирование требований следует поручать представителям *обеих* этих групп. Поодиночке никто из них не выполнит эту работу.

Пользователь знаком с приложениями и большинством нюансов любого сложного проекта. Пользователь должен быть абсолютно уверен, что требования точно и полно отражают стоящую перед ним проблему.

Пользователь не знаком с состоянием технологии и не понимает, что сделать легко, а что сложно. Когда пользователь один формулирует требования, часто оказывается, что возникают технологически наивные формулировки требований, запрашивающие либо слишком много, либо очень мало.

Таблица 5.4. Определение требований пользователем и проектировщиком

	Формулирование требований	
	Может сделать	Пропустит
Пользователь	Ясно выразить важные потребности	Требования к технологии
	Правильно расставить приоритеты	Потребности инфраструктуры
Проектировщик	Определить состояние дел в технологии	Сортировку интересов пользователя
	Определить полноту сформулированных требований	Тонкости прикладной области

Проектировщик обычно не знаком со всеми тонкостями приложений. Если проектировщику позволить формулировать требования самостоятельно, он, вероятнее всего, *пропустит* некоторые тонкие, но весьма важные функции. Получив такую систему, пользователь может подумать, что ее проектировщики живут в башне из слоновой кости. Разработчик должен сформулировать требования к инфраструктуре, которые обычно непонятны пользователю. Чтобы адекватно отразить все потребности проекта, проектировщик и пользователь должны вырабатывать требования совместно. (См. табл. 5.4.)

Язык документирования требований

Язык, на котором формулируются требования, должен быть понятен пользователю. Пользователь должен участвовать в развитии формулировки требований. Эта формулировка должна стать элементом словаря технического проектирования. Она связана скорее с документами проектирования, а не с документами на требования.

Особая важность требований

Тот факт, что наша работа над программами начинается с установления требований, сильно отражается на всем дальнейшем ходе работ. Если этот процесс выполнен недостаточно аккуратно, недостаточно точно, недостаточно адекватно, то от этого пострадают все остальные части процесса разработки. Последующие усилия могут быть просто героическими, но нужная система уже не будет создана.

Персонал, работающий с созданной системой, будет трактовать ошибки, искажения и неоднозначности как правильные требования, понапрасну тратя усилия не только на то, чтобы их внедрить, но и на то, чтобы затем от них избавиться. Плохие требования подобны кривым зеркалам — они все искажают.

В августе 1977 г. министерство обороны США провело изучение основных автоматизированных систем. Большинство из них были системами связи. Это были системы:

1) TRI —TAC;
2) LDMX/NAVCOMPARS и AMMC/SRT;
3) SATIN IV;
4) WMMCCS NORAD/COC;
5) WWMCCS ADP — LANTCOM;
6) Телекоммуникационный центр Пентагона;
7) WWMCCS ADP —CCTC;
8) Автоматизированный технологический контроль;
9) CUDIX/NAVMASC.

Изучение показало:

— Во всех системах требования были неустойчивыми и подвергались пересмотру; чем больше была система, тем большие изменения вносились в них.

— В большинстве систем отсутствовал формальный механизм отслеживания и управления процессом выработки требований.

— Некоторые разработчики даже не смогли осознать необходимость обоснования требований.

— В большинстве систем не было отбоя от “списков пожеланий”.

Результаты исследования в точности соответствуют моему личному опыту деятельности в коммерческой сфере, охватывающей вычислительную технику.

Опытный строитель, работая на богатого заказчика, может столкнуться с большими затруднениями. У такого заказчика может оказаться “мания” внесения постоянных изменении в проект рабочего кабинета, гостиной, веранды и вообще всех комнат дома. Строитель может быть очень опытным специалистом, знающим, умелым, изобретательным и компетентным в своей области — и все же он не сможет закончить строительство дома в отведенный для этого срок и уложиться в смету. Да и более или менее приличный дом он не построит. Любые изменения, вносимые в какой-либо проект в середине работ, по окончании этих работ обычно выглядят неуместными.

Даже тому, кто остро ощущает их необходимость и вкладывает в свою работу всю душу, с трудом удается четко сформулировать требования. Напомним еще раз:

1) формулируйте требования с максимально возможной строгостью;
2) заранее планируйте изменчивость системы.

Кто является действительным пользователем в любом проекте?

Очень часто разработка систем ведется научно-исследовательскими отделами какой-нибудь организации. В их задачу входит и выяснение того, что необходимо потенциальным пользователям.

Когда в начале 1970-х гг. мы успешно завершили первую рабочую версию новой системы управления воздушными перевозками, мы испытали чувство огромного удовлетворения. Группа определения требований FAA подписала свидетельство о завершении работ. Это была настоящая волокита.

Система была отправлена в Джексонвилл, шт.Флорида, для испытаний в рабочих условиях **в** ночную смену в Центре управления авиаперевозками. Однако диспетчеры из Джексонвилла отказались пользоваться ею — они заявили, что она “ненадежна”.

“Ненадежность” диспетчерской авиаслужбы требует особого внимания. Нам удалось решить проблему исключением из состава

системы значительной доли функций, которые ранее были туда включены. Следующий вариант программы, поставленный в Джексонвилле, содержал гораздо меньше системных возможностей. И диспетчерам он понравился. А потом постепенно, очень медленно мы стали добавлять функции, уже запрограммированные и оттестированные нами ранее.

Это служит подтверждением истории, которую я слышал в Хьюстоне. Если вам предложат пуд мороженого, трудно ожидать, что вы сможете проглотить его за один прием. Это не получится.

Такая методика обходится значительно дороже! Последовательное введение функций в действие занимает больше времени. Но у нас нет выбора.

Это другая причина, по которой необходимо отдавать предпочтение эволюционному подходу к разработке систем. Пользователи не могут переварить сразу все функциональные возможности, даже если разработчики и в состоянии их предоставить.

Полученный урок пошел нам впрок. Для дальнейшей работы FAA прикрепило профессиональных диспетчеров к. группе определения требований. Несколько лет эти поистине бесценные специалисты делили свое время между основной работой и работами по проекту. Таким образом нам удалось добиться, что группа определения требований работала именно над тем, что было действительно необходимо реальным пользователям.

Чем более крупную систему создают разработчики, тем больше она подвержена опасности стать излишне сложной в использовании. Может получиться и так, что система будет решать давно решенную проблему.

Я принимал личное участие в создании двух крупных систем, на разработку которых были потрачены миллионы долларов и пользователь которых выбрасывал половину предложенных ему функций после того, как они были полностью разработаны. Операторы не могли с ними работать.

Противоречивые требования разных пользователей

Моя группа разработчиков потеряла 2 млн. долларов при разработке автоматизированной системы выпуска двух крупнейших японских газет; позднее за эту работу мы получили японскую медаль Исикавы за достижения в области технологии и промышленности. С одной из газет мы подписали соглашение о том, что беремся автоматизировать весь процесс выпуска от ввода японских иероглифов, редактирования их и разметки страниц с помощью телевизионных дисплеев и до передачи полученных фотоформ прямо на печатные станки. Мы могли бы выполнить эту работу, полностью уложившись в отведенный нам бюджет. Однако японское отделение IBM предложило пользоваться этой системой и другой газете, которая была конкурентом первой. Между этими двумя газетами постоянно возникали разногласия по вопросам, какие функции и как надо исполнять.

Такой противоречивый поток требований сильно повлиял на процесс разработки и проектирования системы. Необходимо было иметь “хребет”, фундамент, на котором держались бы различные части программ, выполняющие различные функции, заказанные разными газетами. Такая схема построения и постоянно меняющиеся требования привели к значительному превышению затрат по сравнению с простой автоматизацией одной газеты.

Через несколько лет после описанного случая мы понесли убыток еще на 10 млн. долларов. В это время мы были заняты автоматизацией двух новых нефтеочистительных заводов фирмы “Эксон” —в Эдмонтоне, Канада, и в Антверпене, Бельгия. И опять возникли те же проблемы, сначала мы пытались понять, какие функции нужно вносить в систему, а затем пытались удовлетворить сразу двух разных пользователей. То, что оба этих завода входили в одну компанию “Эксон”, еще не означало, что среди них имелось согласие по поводу способов управления процессом очистки. Но опять-таки система работала! И работала хорошо. Просто заранее оказалось сложным оценить весь объем работ с достаточной точностью.

Другой важной причиной, по которой нельзя “замораживать” требования, является то, что пользователи таких систем получают в свое распоряжение совершенно новый инструментарий. На него распространяются те же законы, что и на другие новые орудия труда. По мере роста понимания возможностей нового инструмента пользователи начинают лучше осознавать те функции, которые можно было бы выполнять значительно эффективнее, если бы они были встроены в систему. Предвидеть заранее все функции, а иногда даже хотя бы большинство их не удается.

Один урок, который удается извлечь из всех этих наших усилий, — это разница между “товарными” программами и обеспечением проектов. Проект создается для одного заказчика; товарная продукция должна удовлетворять требованиям многих различных заказчиков. Снова и снова — при работе и на очистительных заводах, и в газетах — мы вынуждены были возвращаться в исходное положение и начинать все сначала, поскольку, когда один заказчик соглашался с предложенными ему методами автоматизации, другой начинал требовать чего-нибудь другого. “Чуть-чуть иначе”.

Требования к товарным программам и программному обеспечению проектов

Основным различием между созданием обеспечения как продукции и обеспечения проекта является различие в *процессах* выработки требований, которым должно удовлетворять программное обеспечение.

Создавая продукцию для продажи, мы должны удовлетворить множество пользователей, поэтому в процессе выработки требований определяется некий *компромисс,* необходимый для того, чтобы общие затраты не стали чрезмерными. К тому же продукция встречает конкуренцию уже с того момента, как объявляют о ее создании. А если она будет прекрасно работать, но не будет продаваться, какой же от нее будет прок?

Для товарных программ, так же так и для аппаратуры широкого потребления, все время вплоть до того, как разработка перейдет в свои последующие фазы, необходимо иметь значительную по размерам группу широко образованных специалистов по выработке требований.

Изменения, вносимые пользователем

Опытные руководители разработок программ знают, что проектировать систему на основе навязчивых идей одного или целой группы влиятельных пользователей — значит, заранее обрекать систему на недолговечность. Системы, “персонализированные” таким образом, никогда не переживают тех пользователей, для которых они создавались. Эта проблема чаще, чем в других областях, возникает при проектировании военных систем.

Адаптируемость способствует непрекращающимся изменениям

Никто всерьез не осмелится предложить внести крупные изменения в конструкцию моста у Золотых Ворот. И нижние пути на мосту Джорджа Вашингтона появились только благодаря тому, что возможность их появления проектировалась с самого начала.

Конструкции из таких материалов, как кирпич, железо, сталь, строятся в течение многих лет, и перестраивать их не так-то просто. Стальную балку не просто *изогнуть,* чтобы придать ей новую форму. Никто не перестраивает вычислительную машину, перегруппируя десятки тысяч транзисторов заново и получая тем самым новую архитектуру. Не так обстоит дело с программным обеспечением!

Программное обеспечение существует *на бумаге,* в памяти машины и нигде больше! Если оно правильно построено, изменять его легко. А нам и нужно, чтобы его можно было легко изменять!

Определение требований — это длительный процесс

Если не поручить определение требований специально выделенным для этого сотрудникам, группа проектирования и реализации создаст “неправильную” систему, которой скорее всего вообще нельзя будет пользоваться.

При обычном ходе событий создается документ, который подписывают представители всех заинтересованных сторон. На него приклеивается этикетка “Требования”, а затем он быстро забывается и выходит из употребления. Это архивный хлам, годный только на то, чтобы выискивать в нем ошибки. В одной системе министерства обороны США, состоявшей из программ в 1 млн. строк, в подобный документ в течение 8 лет не вносили изменений. А ведь программы изменялись! Документы с требованиями абсолютно необходимы для правильной организации работ по разработке, в них должны постоянно учитываться происходящие изменения. Для этого нужны деньги, и в течение всего времени жизни проекта им должны заниматься квалифицированные люди.

При работе над многими проектами их руководители даже не видели необходимости в создании групп определения требований. Обычно считалось, что этим могут заниматься группы разработки *и/или* проектирования. Но они не занимались этим. У них вполне хватало своих дел; они слишком заняты проектированием и разработкой, чтобы заботиться еще и об определении требований; некоторые нужды они просто игнорируют, если последние мешают им уложиться в отведенные сроки и сметы.

Если спросить их, то они конфиденциально ответят, что “знают все требования”. В девяти случаях из десяти они не знают ни одного!

Спецификация требований

Английский язык (как, впрочем, и любой другой.— *Ред.)* полон двусмысленностей и часто приводит к путанице. Мы обнаружили, что он не соответствует задаче создания спецификаций. Прибегнув к помощи американской группы разработки газетных систем, девять американских газет решили автоматизировать свое производство даже раньше, чем это успеют сделать японцы.

Предвидя проблему спецификации требований пользователей, мы потратили значительные усилия и время на создание точного

документа размером в 2400 страниц, в котором постарались учесть *все*. За исключением того, что читали этот документ мы и те, с кем эти требования нам нужно было согласовать, совершенно по-разному. Впрочем с лучшими намерениями. В самом деле, какой редактор может согласиться с редактором другой газеты по вопросу, что считать спецификацией издательского дела? Язык наш недостаточно точен для этого.

С аналогичной проблемой столкнулись мы при создании системы реального времени для военно-морского флота. Когда до срока сдачи оставалось всего несколько месяцев и спецификации требований пользователя уже разрослись в том в полторы тысячи страниц, руководитель разработки сообщил мне, что директивные сроки и финансовые планы будут выполнены, а также, что требования постоянно изменяются. "А что происходит со спецификациями?" — спросил я. "А-а, на них никто не обращает ни малейшего внимания. Они уже устарели на несколько месяцев". И в то же время он старался уложиться в сроки и в смету. Зачем?

Определение требований к окружающей обстановке в фазе использования

До сих пор мы говорили о необходимости определять требования пользователей. Но, кроме того, следует ясно определять требования, относящиеся к окружению в фазе использования. *Необходимость удовлетворять этим требованиям может на порядок увеличить затраты на создание программного обеспечения*.

Главными факторами фазы использования являются:

1) требования надежности;

2) требования скорости реакции системы;

3) требования реального времени;

4) требования диалогового режима.

Этим требованиям должно удовлетворять как программное, так и аппаратное обеспечение, однако пользователь их замечает, только если они не удовлетворяются. Если все сделано правильно, пользователь никогда этого не замечает.

В этой главе мы обсудим все эти и еще многие другие факторы, из которых складывается обстановка фазы использования и фазы разработки. Мы увидим, почему они имеют столь всеобъемлющий характер.

У разработчика часто возникает возможность навязать пользователю некоторый режим работы! Но эту возможность надо еще заметить и уметь ею воспользоваться! Однажды на моих глазах одна большая система типа III перерастала в систему типа V, потому что никто не сказал пользователям — руководству системы — о том, что систему можно выключать на несколько часов. Это было бы приемлемым условием. Пользователь не простаивал бы без дела, данных для него было вполне достаточно. Система могла не быть безотказной! Но кто-то, опережая требования пользователя — и весьма опрометчиво,— потребовал, чтобы система не имела отказов.

Получить безотказно работающую систему стоит очень дорого, для этого нужно много избыточной аппаратуры и множество логически очень сложных программ.

Управление процессом выработки требований

Требования нельзя просто "собрать" и документировать. Ими нужно управлять! Особенно это необходимо при создании товарных программ. Очень многие товарные программы пропадали — а в них были вложены десятки миллионов долларов — только потому, что руководитель разработки согласился на слишком широкий набор требований. Некоторые называют избыток требований "потерей элегантности". Лучшее — враг хорошего. Обычно оно бывает пагубным. Торговцы всегда будут настаивать на включении *n-й* функции; руководитель разработки должен отказываться от внесения добавлений к требованиям.

Это же относится и к проектам типов III, IV и V. Все пожелания пользователя даже просто *перечислить* невозможно (мы предполагаем, что нам удалось найти подлинного пользователя). Это слишком перегрузило бы систему. К тому же пользователь часто и не знает, что необходимо и сколько стоит то, что желательно.

Руководитель разработки программного обеспечения должен назначить кого-нибудь на пост руководителя процессом определения требований, и работа должна идти одновременно в двух направлениях: руководитель определением требований должен свести требования пользователя к действительно необходимым и должен держать пользователя под контролем, не позволяя тому выдвигать нелепые требования.

При создании программного обеспечения проекта пользователь должен сотрудничать с разработчиком. Все связи между разработчиками и пользователями должны быть длительными и интенсивными. Этому способствует их расположение в одном месте! Обязанностью разработчика программного обеспечения является удовлетворение пользователя. Если этого удается добиться, проект будет успешным.

Новые методы определения требований

Для того чтобы сделать первую фазу разработки программного обеспечения более управляемой, разработаны новые методы, хотя их не так уж и много. Аналогично тому, что происходит в области языков программирования и трансляторов, важнейшим принципом, лежащим в основе этих новых методов, является понятие абстракции. Эти методы представляют собой комбинации правил записи и правил ведения работ.

Правила ведения работ помогают достичь нужного уровня абстракции, правила записи необходимы для облегчения взаимодействия и проверок. Эти правила приводят к тому, что люди заняты определением требований в течение длительного периода времени — иногда это даже пугает руководство. Но все окупается.

Для всех методологий характерно следующее:

— Разбиение на уровни абстракции. (Особенно важно для анализа сложных систем.)

— Ограниченное число элементов, приходящихся на один уровень абстракции — очень существенно для успеха, — обычно 7 или меньше.

— Ограниченный контекст: в него включается лишь то, что входит в процесс, все, что в процесс не входит, исключается.

— Включаться должны как данные, так и действия над ними.

Новые методы улучшения процесса определения требований многообещающи и сложны. Обычно они представляют собой комбинации методов документирования и правил ведения работ.

Вот некоторые из этих методов:

SADT	Технология структурного анализа и проектирования
IA	Информационный анализ
HIPO	Иерархические схемы ввода, обработки, вывода
PSL/PSA	Язык постановки задачи/анализатор постановки задачи
RSL/REVS	Язык формулировок требований/системы вычислений и оценок требований
CARA	Анализ требований с помощью вычислительных машин

Большинство методов основано на строгих формальных правилах записи и последовательном приближении к конечному результату.

Эти методы обещают очень многое. Когда-нибудь в будущем мы удивимся, почему же мы раньше обходились без них. Для подробного их описания мало одной странички, мало даже отдельной главы. Для этого нужно писать целые книги, для их понимания необходимо прослушать двухнедельный курс лекций.

Личный опыт

Нам не удалось с первой попытки понять нужды диспетчеров FAA. Группе определения требований нужна была информация, постоянно поступающая от диспетчера, находящегося "на линии". Тогда в Хьюстоне мы правильно поступили, послав наших разработчиков поработать бок о бок с нашими пользователями.

Нашей же хьюстонской группе не сразу удалось понять нужды энергетических компаний. Нам не сразу стали ясны требования к системе автоматической очистки нефти. Первая попытка понять требования японских газетчиков тоже не удалась. Процесс определения требований все еще является камнем преткновения при разработке новых больших программных систем.

Резюме: требования к большим системам программного обеспечения

1. Первым требованием к системе является легкость ее модификации, поскольку требования будут постоянно изменяться. Программы надо заранее делать модифицируемыми; для этого нужно заранее планировать денежные средства.

2. В течение всего времени развития системы за выработку требований должен отвечать один и тот же человек.

3. Требованиями необходимо управлять.

4. Если требования не записаны и не доступны каждому, можно считать, что они не существуют. Пока требования находятся только в голове, они не приносят пользы и не долговечны.

5. Должен быть найден и вовлечен в дело действительный пользователь, а не его заменитель.
6. Из требований нужно удалить все относящееся к личности заказчика.
7. Необходимо действовать по методологии определения требований.

Проектирование

Что такое проектирование?

Это слово одновременно означает и некоторое понятие, и самый процесс. Переходный глагол “проектировать” означает 1) “готовить предварительный набросок или план (для работ, которые нужно в дальнейшем выполнить)”, в особенности это относится к составлению схем некоторых структур и форм, например, можно проектировать новый мост; 2) в последнее время этот глагол стали использовать в значении “планировать и моделировать с особым искусством и художественным вкусом” *{В СССР для 2) используется слово “дизайн” и его производные. Так что термин “проектирование” не столь многозначен. — Прим. ред.}*.

Область эта трудна именно потому, что она в большой степени связана с искусством. Проектирование подразумевает возможность выбора. Выбирается форма, очертания, последовательность, схема цветовых сочетаний или другие компоненты, и из них составляется некоторое единство.

Проектирование может быть очень хорошим и очень плохим. В архитектуре проектирование имеет функциональное и эстетическое назначение. В смысле функциональности мы можем говорить о том, “хорошо используется” пространство или нет. С точки зрения эстетики речь идет о том, “хорошо ли выглядит” объект или нет. Для оценки функционального проектирования существуют специальные меры или критерии. Во сколько обходится застройка 1 м^2 площади, или обогрев, или охлаждение? Сколько уже построено “работающих устройств”? Сколько времени потребуется для конструирования? Вообще может использоваться огромное множество разных критериев.

В искусстве же то, что кажется прекрасным одному, может не нравиться другому. Только в тех случаях, когда что-то уж очень хорошо или безобразно, мы можем встретить некоторое подобие единодушия.

Ну а что же мы проектируем при разработке программного обеспечения? Это весьма нетривиальный вопрос. Какие части или куски, формы или цвета, связи и образы можем мы выбирать?

В своих “Заметках о синтезе форм” Кристофер Александер *{Alexander Chr., Notes on the Synthesis of Form (Cambridge Harvard University Press, 1964).}* указывает несколько наиболее важных моментов проектирования. Самым интересным является то, что измерять качество проектирования с позитивной точки зрения невозможно, но следует убедиться в отсутствии отрицательных черт, и это дает представление о том, насколько хорошо проект соответствует (его собственные слова) данной ситуации. На идее соответствия автор подробно останавливается, подразумевая под этим применимость проекта к данной ситуации, к данным обстоятельствам.

Александер различает “сознательное” и “бессознательное” проектирование. Бессознательное проектирование длится целые столетия, так что дома и хижины всех местных жителей полностью соответствуют окружению, в котором они используются. Изменения вносятся медленно и постепенно, каждое всесторонне изучается в течение нескольких десятилетий и лишь затем принимается. Проектирование превращается в адаптацию жилища к нуждам жителей.

Сознательное проектирование — это то, чем мы сейчас занимаемся. Со всех сторон на нас обрушиваются различные изменения, мы используем новые инструменты и методы, у нас появляются все новые и новые обязанности. Проектирование становится безотлагательным и специфическим. Качество проектирования оценивается по его соответствию условиям, сравнением его с конкурентными предложениями, если таковые имеются, по его работоспособности.

Программное обеспечение — это подсистема

Чтобы получить необходимую нам систему, мы должны в определенном порядке пройти по разным уровням проектирования: система, подсистема, программное обеспечение, программа и, наконец, рабочая программа — код.

Система как целое состоит из подсистем, и самый выбор этих подсистем уже является проектированием. К примеру, для общенациональной спутниковой системы мы можем остановиться на нескольких различных наборах подсистем.

План № 1 Спутник; система связи космос — Земля; наземная система и т.д.

План М 2 Спутник; микроволновая система; вычислительная система и т.д.

План № 3 Спутник; система связи; система сообщений; управляющая система и т.д.

Этот самый верхний уровень проектирования является самым критическим. Именно он определяет главные черты построения системы, ее работу и обслуживании.

Подсистема программного обеспечения входит в состав подсистемы обработки данных, которая в свою очередь может являться подсистемой одной, двух или даже всех основных подсистем. Конечно, вычислительные машины есть и на спутнике, следовательно, и на спутнике есть программное обеспечение. Вероятнее всего, программное обеспечение есть в каждой большой подсистеме.

Рис.5.11 Требования.

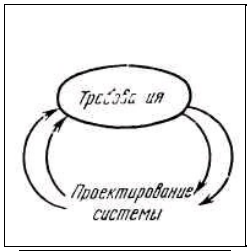


Рис.5.13 Переход к подсистемам.

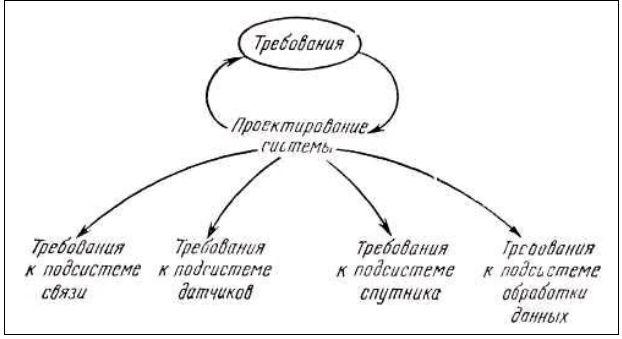


Рис.5.14 Подсистемы определены

Подсистемы обработки данных (ОД) состоят из вычислительных машин и программного обеспечения. Программное обеспечение должно “соответствовать” аппаратуре, оно должно быть спроектировано так, чтобы увеличивать производительность как этой аппаратуры, так и всей системы в целом.

Звучит это просто; однако чаще всего ответственный за аппаратуру ОД мало знает о программном обеспечении, он выбирает машину, только исходя из ее стоимости и скорости работы. Разработка программного обеспечения игнорируется. Я видел, как подобные ошибки допускают даже в тех случаях, когда всеми признавалось, что разработка программного обеспечения будет ключевой во всей схеме разработки системы.

При проектировании систем нужно делать обоснования выбора как аппаратуры, так и программного обеспечения. Проектировщики программного обеспечения должны обязательно поставлять информацию, необходимую для правильного выбора.

Многократное и параллельное проектирование

Существуют десятки и даже сотни методов, которые позволяют производить проектирование параллельно. Иллюстрации этого утверждения приведены на рис. 5.11—5.17.

По мере хода работ по проектированию разработчики сноса и снова возвращаются к выработанным ранее требованиям, переоценивают их или даже модифицируют в соответствии с тем, как они влияют на усилия проектировщиков.

Одним из наиболее важных результатов проектирования систем является расслоение целой системы на отдельные подсистемы. Выбор подсистем в наибольшей степени определяет природу системы и влияет на ее разработку. Необходимо выработать требования ко всем подсистемам, причем эти требования должны быть согласованы с требованиями верхнего уровня.

Параллельная разработка

На рис. 5.15 показано, что все подсистемы можно разрабатывать параллельно. Очевидно, что разные люди проводят свое собственное проектирование, результаты их труда должны быть проверены и проконтролированы проектировщиком системы (или группой системных инженеров).

Такой процесс все большей и большей детализации верхнего уровня продолжается непрерывно. Начиная с этого места мы будем изучать только одну подсистему. Так как в наибольшей степени нас интересует программное обеспечение, мы будем исследовать подсистему обработки данных (рис. 5.16), в которую входит большая часть всего программного обеспечения. Отметим тут же, что программное обеспечение может в той или иной степени входить в любую подсистему. Итак, подсистема обработки данных должна быть в свою очередь разделена на составные части.

И конечно, каждая из этих подсистем тоже может подразделяться. Чтобы изобразить “окончательное” разбиение на отдельные части, нам нужно построить целое “деревом проектирования, что и сделано на рис. 5.17.

Рис. 5 15. Проектирование подсистем — дополнительные итерации.

Рис. 5.16. Подсистема обработки данных и ее составные части.

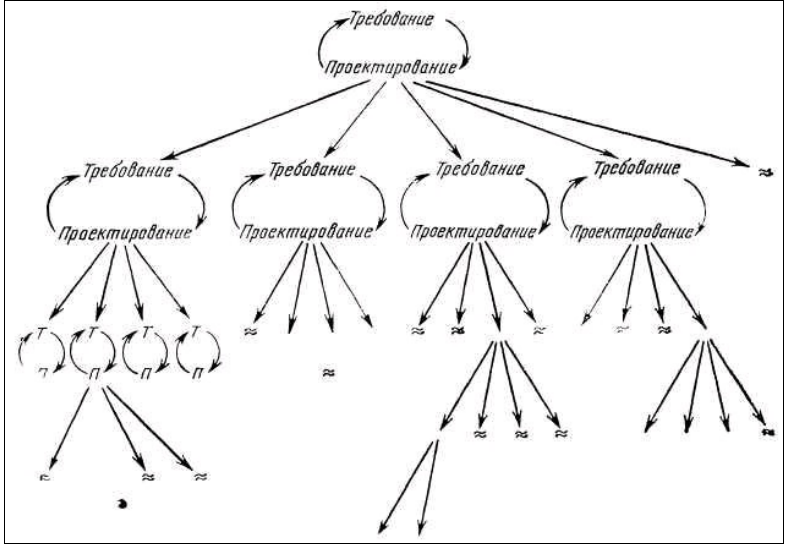


Рис. 5.17. “Дерево” проектирования.

Очень быстро мы исчерпали все средства и не можем изобразить всю систему на одной диаграмме. Одновременно проводятся работы по проектированию десятков частей! А отслеживание взаимосвязи тех отдельных частей, которые могут взаимодействовать друг с другом, входит в круг обязанностей проектировщика системы. Выбор подсистем во многом базируется на минимизации связей между подсистемами — и проектирование каждой подсистемы должно заключаться в самостоятельном определении средств, при помощи которых можно удовлетворить требованиям к входной информации, поступающей от других подсистем, и к информации, передаваемой из данной подсистемы на вход последующих.

Итерации при проектировании и выработке требований

Итерационный процесс можно проиллюстрировать и так, как это сделано на рис. 5.18. Пунктирными линиями обозначены процессы, которые еще предстоит спроектировать, они основываются на уже определенных областях, отмеченных сплошными линиями. Названия месяцев представляют собой лишь общее описание отрезка времени. При проектировании систем меньшего размера месяцы можно заменять пронумерованными неделями, а для совсем небольших систем можно даже указывать и отдельные дни.

Обратите внимание на полное возобновление работ в марте. По мере более подробной детализации, проводившейся в феврале, проектировщики обнаружили важное упущение или нашли более оптимальное строение системы. Это привело к тому, что им пришлось вернуть процесс в исходное положение и заменить начальную точку *А* новой точкой *В*. Очевидно, что такое может произойти со всеми или с большинством подразделов системы. Части 1, 2 или 3, определенные на ранних этапах проектирования, могут использоваться в новых подсистемах В, Р и Л.

Итерации при проектировании идут постоянно. По самому своему характеру работа проектировщика полна сюрпризов. Если бы он мог представить *себе весь ход работ в деталях,* то это значит, что у него имеется не готовый проект, а лишь иллюстрация к нему. Итерации случаются на всех уровнях, даже на уровне микропроектирования. События и открытия, приводящие к изменениям, не обязательно должны подчиняться общей иерархии, некоторые могут влиять на последующие стадии, а некоторые на стадии, проходящие параллельно. Схема февральских работ показывает, что одновременно происходит детализация и определение четырех различных процессов, значит, *эти процессы не существует* независимо друг от друга, они чаще всего взаимодействуют между собой Эти взаимодействия нужно исследовать, детализировать и иногда специально проектировать Взаимодействия и детали одной или нескольких работ или функций, возможно, придется видоизменить в соответствии с тем, как проектировщик решил выполнять некоторую другую работу. Может случиться, что после определения работы Ф, нам придется модифицировать П.

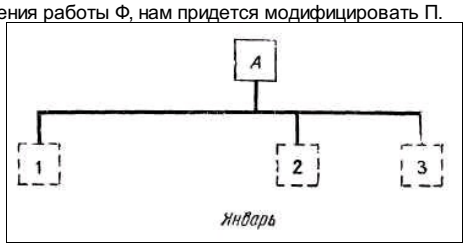


Рис.5.18-а Изменения при проектировании. (январь)

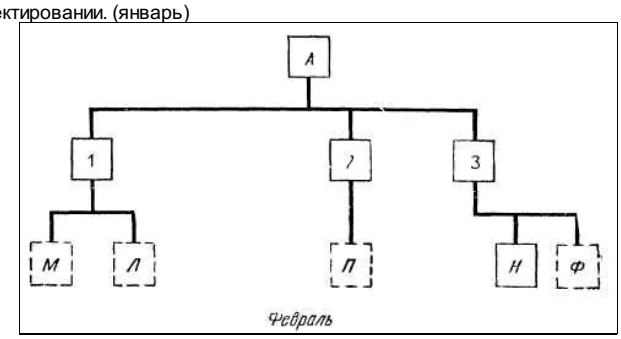


Рис.5.18-б Изменения при проектировании. (февраль)

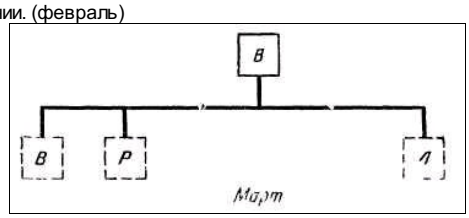


Рис.5.18-в Изменения при проектировании. (март)

Конечно, этот пример не очень точно описывает то, что происходит на самом деле при разработке больших систем. Десятки, сотни работ выполняются одновременно. Вся эта деятельность напоминает работу пчелиного роя, она обязательно должна кем-то координироваться.

Из всего сказанного можно логически вывести такое заключение: утверждение “проектирование завершено на 90%” — это не факт, а лишь его оценка, поскольку никто в точности не знает, не встретится ли нам какой-нибудь сюрприз, который заставит нас вернуться несколько назад и начать переделку проекта, который мы считали завершенным.

И все же нам не устают повторять, что “проектирование завершено на 90%”.

Проектирование программного обеспечения фазы использования

Давайте на некоторое время предположим, что аппаратура настолько соответствует поставленной перед нами задаче, что меняться больше не будет. Что же в таком случае мы будем проектировать в программном обеспечении?

В первую очередь нам придется спроектировать, т.е. выбрать, главные составные части нашей программы. Нужно ли нам иметь отдельные системные программы? Нужна ли специальная операционная система? Система управления базой данных?

По каким признакам мы можем определить, стоит ли выделять эти подсистемы или включать их в состав прикладных программ? Мы судим об этом по двум вещам: по требованиям, определенным в формулировке требований (имеются в виду требования как к функциональным возможностям, так и к окружению), и по всем 12 характеристикам программ, распределенным по всем фазам жизни программ.

Проектирование должно учитывать все 12 характеристик программ и программного обеспечения и все три фазы полного цикла жизни программы, даже если некоторые характеристики при проектировании придется отбросить!

Для группы десантников, которую ночью направили через болото, любой путь, приводящий к цели, хорош. Это дорога? Она приведет нас туда, куда надо? Да? Тогда это то, что надо! Времени на изучение и поиски более коротких дорог нет. Надо бежать по этой!

Такие методы хороши только для программ типа “зубочисток” или в тех случаях, когда через это болото больше никому не придется переходить.

Если же проектируется долгоживущее программное обеспечение, то на фазе проектирования необходимо затрачивать некоторые усилия на то, чтобы максимально облегчить сопровождение программ в дальнейшем.

Иногда проблемы использования можно не принимать во внимание. Ведь программа может использоваться крайне редко. В некоторых случаях упор приходится делать на фазу разработки, в других случаях особую важность приобретает фаза сопровождения. Более подробно об этом будет сказано чуть позже.

Но *если только* та машина, которая будет применяться на фазе использования, обладает достаточной мощностью и памятью и мы можем себе позволить некоторые дополнительные расходы, обычно связанные с универсальностью уже готовых системных программ, нам обязательно надо выбирать готовые программы. Следующим нашим шагом должно быть определение наилучших среди всех доступных нам программ. В этом состоит проектирование, это выбор того, какие конкретные программы взять из имеющихся, и это действительно очень ответственное проектирование. При этом формируется система в полном смысле слова. Это расслоение на отдельные части сродни выделению сборочных узлов при проектировании аппаратуры.

Проектировщик соглашается на компромисс между стоимостью и скоростью, между простотой запросов данных и скоростью их поиска, между объемом занимаемой памяти, схемой упаковки и временем ответа, между средствами распределения ресурсов и богатством функций, между средствами управления окружением и выполняемыми функциями, а также между многими другими характеристиками системы.

Кто должен вести проектирование

Заниматься проектированием может не каждый. Дейкстра в статье “Структура мультипрограммной системы „THE" {Dijkstia E W, The Structure of “THE” Multiprogramming System: Communication of the ACM, vol.11, 5 (1968). Выдержки публикуются с разрешения Ассоциации по вычислительной технике.} отмечает два момента:

- *Попытки избежать разного рода “патологий” отняли гораздо больше сил, чем мы ожидали, причем некоторые наши беды были прямо связаны с нашими изобретениями на первых этапах работ... Если бы мы больше уделяли внимания патологии на ранних стадиях проектирования, мы бы выработали гораздо менее утонченные правила управления работой*

И

... по крайней мере в моей стране (имеются в виду Нидерланды) интеллектуальный уровень, необходимый для проектирования систем, чаще всего сильно недооценивается этот вид работ чрезвычайно труден, и любые попытки вести проектирование иначе как с помощью наилучших специалистов, обречены либо на провал, либо на весьма умеренный успех за огромную плату

Обратите внимание на слова: “наилучшие специалисты”.

Выбирая проектировщика, прежде всего обращайтесь к тем, кто уже ранее выполнял такую работу! К тому, кто уже вел систему от самого начала до самого конца! Постарайтесь договориться с ним. Если в вашем распоряжении нет людей, проектировавших ранее программное обеспечение *данного* типа, нужно создать “группу проектирования”. Последите за ее работой. Из этой группы выделятся один или два человека, которые и будут заниматься проектированием программного обеспечения.

Тот *единственный* человек, который является подлинным проектировщиком большой системы, обычно весьма умен, спокоен и упорен. Он достаточно спокоен, чтобы быть хорошим руководителем, достаточно умен, чтобы уметь усвоить множество конфликтующих целей, влияний, методов и противоречий между людьми и разобраться в них.

Если проектированием занимается слишком много людей, если при проектировании слишком увлеклись демократией или если *проектировщики* начинают сменять друг друга — все это очень сильно затрудняет разработку.

Конечная продукция — что же мы создаем?

Зрительно представить себе программное обеспечение очень трудно. Статические представления его весьма бледны, но зачастую это лучшее, что нам удается сделать. Рассмотрим пример программного обеспечения системы наземного контроля проекта “Аполлон”, это обеспечение работало в Хьюстоне много лет, и оно поможет нам увидеть все, что мы намеревались создать.

У нас была большая (более 1 млн. команд) программная система, хорошо документированная и оттестированная. Все эти команды были распределены по программам, подпрограммам и модулям. Подразделы строились отдельно и включались во внешние структуры.

Наше программное обеспечение “приняло” несколько форм. У нас имелась распечатка (напечатанный на машине текст) всех команд по порядку. Имелись колода перфокарт и магнитная лента, на которой была записана программа. Ни одно из этих воплощений программы человеку увидеть не легко — даже распечатку. Ни одно из них не давало возможность увидеть структуру или последовательность выполнения команд программы. Чтобы понять идею системы, нам требовались такие схемы, как изображенные на рис. 5.19 и 5.20. Вся программная система была разделена на восемь основных частей, что и показано на рис. 5.19.

Назначение каждой подсистемы совершенно ясно из их наименований. Общее управление системой осуществлялось операционной системой реального времени, которая определила, какую программу надо исполнять в **данное время**.

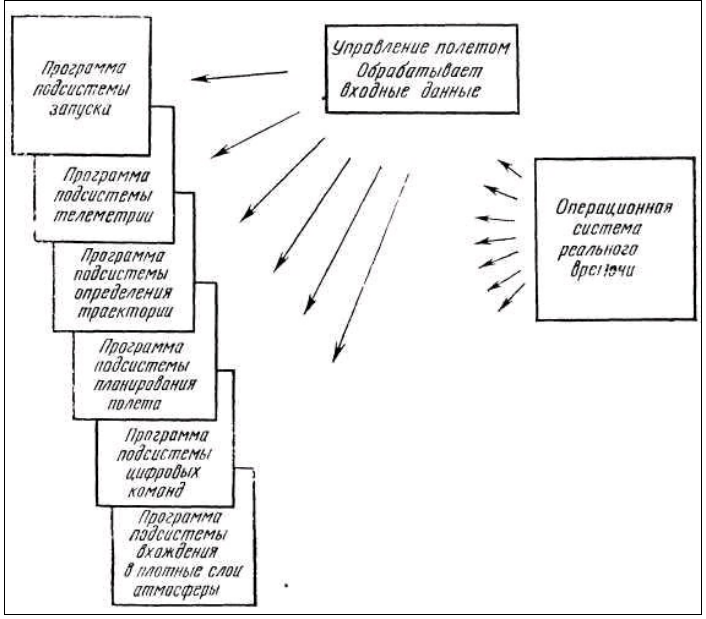


Рис. 5.19. Структура программного обеспечения системы наземного контроля „Аполлон"

Рис. 5.20. Структура программного обеспечения подсистемы запуска "Аполлон".

Теперь перейдем к более подробному изучению подсистемы запуска (рис. 5.20). Каждый квадратик обозначает очень большую программу, которая при необходимости будет выполнена. Операционная система определяет, *когда* наступает такая необходимость. Если мы продолжим разбиение этих блоков программного обеспечения на более мелкие детали, мы получим разветвленное дерево, подобное представленному на рис. 5.21. Каждому квадрату соответствует некоторый акт проектирования. Проектирование здесь заключается в выборе конкретной комбинации функций и распределении их между отдельными вполне определенными программами.

Когда мы смотрим на статическое представление программы, подобное дереву на рис. 5.21, мы должны помнить, что каждый квадрат — это программа, которая может быть выполнена в любое время и в самых разных комбинациях **с** другими программами.

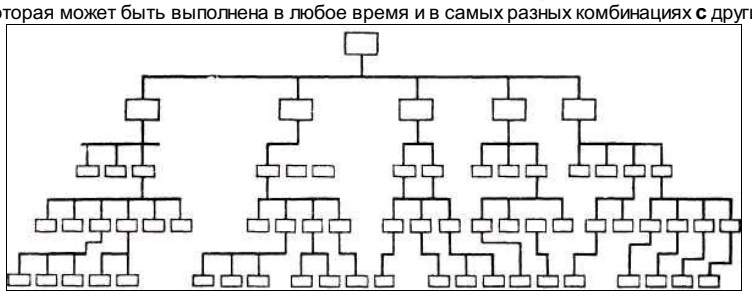


Рис 5.21. Типичная структура большой системы программного обеспечения.

Для наземного контроля системы “Аполлон” было создано более 400 программных модулей. Но все же “познать” программу мы не могли. Диаграмма давала нам лишь безжизненную иллюстрацию. Работающая программа — это нечто совсем другое.

Это все равно что сравнить художественный фильм с пленкой, на которой снят этот фильм. Увидеть пленку без использования кинопроектора не означает еще увидеть фильм! Мы имеем здесь два совершенно разных явления! Слово “пленка” обозначает понятие носителя, на котором заснят фильм; а слово “фильм” связано с самим результатом— объектами, о которых идет речь в фильме. К счастью, в области программного обеспечения такого рода путаница не возникает.

Составные части и процесс проектирования

В проектировании можно выделить три части. Принятие решения, структуризацию и представление (описание) (см. рис. 5.22).

ПРИНЯТИЕ РЕШЕНИЯ

Собираемся ли мы организовывать поиск последовательным просмотра файла или хотим применить метод двоичного поиска — настраиваясь сразу на середину файла, определяя нужную нам половину, верхнюю или нижнюю, и затем переходя в середину выбранной части, многократно повторяя данный процесс?

Это и есть принятие решения, выбор алгоритма, процесса. Обычно это творческий акт, в результате которого возникает новый метод, особенно в новых, сложных или очень крупномасштабных разработках.

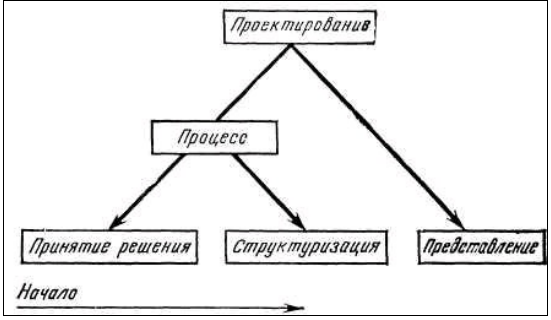


Рис. 5.22. Составные части проектирования.

Структура

Выбрав некоторый метод, мы “детализируем” его, превращая в некоторую “структуру”. В любой значительной работе к одному и тому же результату могут привести десятки различных структур. Различия в структурах могут быть значительными, результат же будет одинаковым.

Например, выбрав метод двоичного поиска, я могу писать программу либо на Фортране, либо на Коболе. В Коболе затруднено индексирование, а в Фортране очень неудобны манипуляции с данными. Программы на этих языках будут сильно отличаться друг от друга; в результате две разные программы будут осуществлять один и тот же двоичный поиск.

Оценивать структуру можно разными способами. Двумя мерами оценки могут служить модульность и четкость интерфейса.

Описание

Описание, или представление, программы может принимать различные формы. Это могут быть блок-схемы, снабженные некоторыми примечаниями; это могут быть иерархические схемы ввода, обработки, вывода (HIPO), схемы Вернера — Орра и т.д.

Целью описания является объяснение новичкам и продолжателям разработки назначения программы и методов, которыми она решает задачу. Следовательно, оценивать описание нужно по тому, насколько хорошо оно отражает эти сведения. “Хорошо” означает понятно, однозначно и немногословно.

В книге Лоуренса Хэлприна “Творческие процессы в жизни людей”[1]{*Halprin L. The RSVP Cyeles New York: George Braziller, Inc. 1969.*} исследуются процессы, с помощью которых люди пытаются проиллюстрировать, документировать в удобочитаемом виде формы в различных областях искусства или связанную с ними деятельность. Слово “партитура” используется автором для обозначения представления, “синтезирования процесса, его развития во времени. Наиболее известны музыкальные партитуры, но я расширил это понятие и включил в него все области человеческой деятельности.”

Далее Хэлприн переходит к обсуждению танцев и партитур для них. Вслед за этим он исследует городское планирование, астрономию, архитектуру, игру, посадку на Луну — и десятки других видов деятельности. Он показывает, какой замечательной изобретательностью отличаются люди при создании методов изображения развивающихся процессов.

Он также указывает, что, хотя партитура и сама деятельность неразрывно связаны, они сильно отличаются друг от друга. Скрипач-виртуоз исполняет Брамса намного лучше, чем студент, пользующийся теми же нотами. Разработчики программного обеспечения знают, что всякое описание большой программы статично и что в процессе ее работы может случиться множество вещей, не отраженных в описании! Мы не можем описывать выполняющуюся программу. На с.96 мы уже отмечали тот факт, что при выполнении различных вариантов последовательностей команд их даже невозможно оттестировать. Когда начинает выполняться наша большая система программ, реализуется какой-нибудь один из вариантов.

Многие из новейших “методов проектирования”, предлагаемых в настоящее время, имеют чисто описательный характер. Однако это не должно их опорочить. Мы отчаянно нуждаемся в лучших способах обмена сложными идеями.

В большинстве книг, в том числе и в этой, мало пишут о первой части проектирования или создания. Мы просто еще не готовы много говорить об этом. Книги, посвященные подобным вопросам, имеют гораздо более широкую область приложения, чем вычислительная техника и программное обеспечение. Обычно в них идет речь о свободных ассоциациях, или о “мозговом штурме”, или об изучении процесса решения реальных проблем.

Последовательный ход процесса проектирования программной системы

Процесс, ход проектирования понимается недостаточно хорошо даже самыми опытными программистами. Многие выполняют его инстинктивно, пропуская некоторые шаги, необходимые для хорошей разработки. Многие выполняют этот процесс очень плохо.

Показанный на рис. 5.23 процесс проектирования начинается после определения требований к системе и развивается, охватывая все более мелкие его этапы.

Функции процесса определения требований рассматривались нами в предыдущем разделе книги. На этой схеме мы снова показываем этот процесс, причем документы, получающиеся в результате, являются первой, исходной точкой проектирования. Заметьте, что мы отделяем действие от продукта, получающегося в результате, помечая и то и другое. Делаем мы это потому, что при разработке программного обеспечения и систем часто путают некоторые из этих терминов.

Второй шаг, проектирование, связан с рассмотрением различных подходов к решению проблемы и с выяснением, могут ли они привести нас к этому решению. Этот шаг часто связан с применением методов проб и ошибок, нередки возвраты к предыдущему шагу. По мере продвижения вперед, нам часто бывает необходимо развить наш проект еще дальше, чтобы проверить, действительно ли мы можем завершить его так, как нам бы этого хотелось. После того как это будет проделано достаточное число раз и с достаточной степенью детализации, мы получим некий подход — и очень часто он оказывается искомым подходом, в большинстве случаев он более или менее работоспособен.

К сожалению, в подавляющем большинстве случаев проектирование на этом и заканчивается, а все дальнейшие работы по созданию системы связаны с воплощением в жизнь только что выбранных вариантов.

Заметим, однако, что в результате второго шага создаются многие тома документов и отчетов. Ведь мы получили какое-то решение. В некоторых случаях кипа документов достигает двух и более метров в высоту.

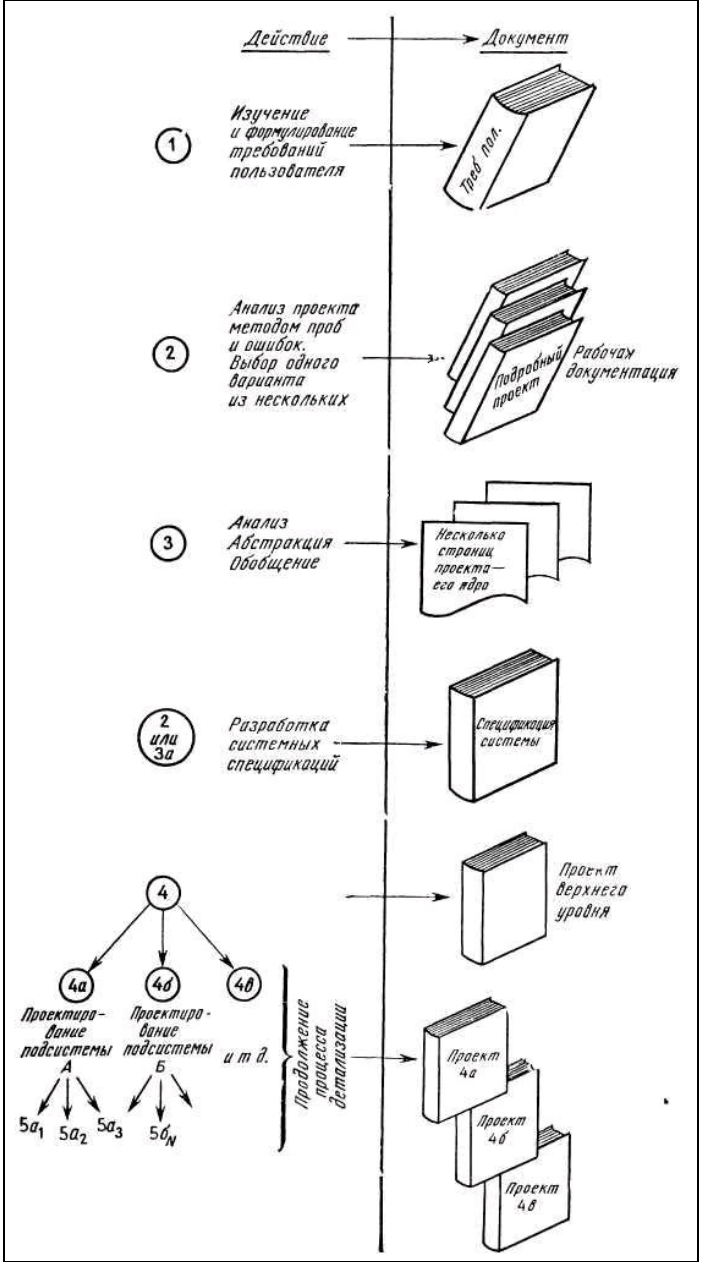


Рис.5.23. Основные шаги процесса проектирования.

Может ли это называться проектом? Конечно, но только не вполне законченным, причем, по всей вероятности, он не очень-то нам понятен и его трудно кому-нибудь объяснить! Даже если он правилен!

И надо проделать еще третий шаг, чтобы сделать полученный проект более компактным, четким, легким для понимания и объяснения и, следовательно, для реализации.

Чаще всего третий шаг, связанный с синтезацией и уменьшением объема проекта, *пропускается,* многие сразу переходят к четвертому шагу — подробному проектированию после завершения шага 2 — нахождения работоспособного решения. *Выполнение шага 3 приносит существенную пользу, потеря преимуществ при отказе от него наносит серьезный удар по всей разработке большой программной системы.*

Преимущества этого шага, вносящего в проект ясность и краткость, сразу бросятся в глаза после его выполнения. Проект станет легко описываемым и вполне понятным. Лежащие в основе принципы понять просто. Результаты проведенного ранее эвристического (методом проб и ошибок) проектирования (шаг 2) слишком объемисты и неясны. Проект может быть правильным, но пользование им будет связано с большими затратами труда и времени. Обычно он бывает понятен одному или, в более счастливом случае, нескольким лицам. Все остальные участники работ понимают до конца лишь части системы, но не систему как целое.

Переработанный после шага 3 проект имеет то преимущество, что его можно показать всем, тогда проектировщики и программисты подсистем смогут работать над своими частями с большей пользой для общего дела.

На четвертом шаге создается проект верхнего уровня. Положения, сформулированные в нем, служат исходными требованиями для работ по созданию других проектов, более низких уровней.

Повторным применением шага 4 ко всем более низким уровням проекта мы создаем целую серию проектов. Это длится до тех пор, пока не будет создан проект самого нижнего уровня. На каждом уровне создаются как описания требований к более низкому уровню, так и проект модуля; этот процесс продолжается до тех пор, пока не будет достигнут уровень реализации (написания) или уровень тривиальности. Чаще возникает эта последняя ситуация.

Рассмотрим различные варианты третьего шага: шага критической оценки и очистки. На рис. 5.24 направление процесса проектирования указывается стрелкой — слева направо. С левой стороны, в начале проектирования, обозначен процесс *создания.* Формирующийся при этом проект еще очень расплывчат и находится пока только в голове проектировщика. Небольшой прямоугольник в центре рисунка обозначает очистку, которая следует дальше. Очистка крайне важна. Из проекта исключаются повторы, неоднозначности и тривиальные положения. Внимание сосредоточивается на самых существенных частях решения. В процессе очистки исключаются все случайные, несущественные элементы, выделяются основные принципы, которым надо следовать. Краткое, сжатое определение метода решения облегчает управление всей оставшейся частью работ по разработке программного обеспечения.

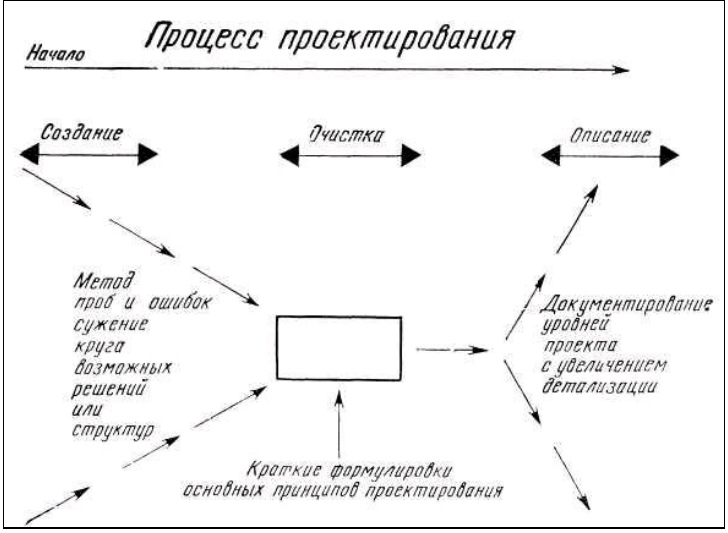


Рис. 5.24. Процесс проектирования.

Повторяю, что подобная очистка часто не производится. Производить ее трудно, для этого требуется особый талант и очень много труда. Но проводить ее все же надо обязательно.

Расширяющееся пространство на рисунке справа от шага очистки служит для обозначения всевозрастающего объема проектирования и документирования по мере продолжения работ. На базе сформулированных нами фундаментальных основ проектирования мы приступаем к наращиванию уровней детализации, двигаясь прямо к реализации проекта.

Джон Ливингстон Лоувс (1867—1945) в книге “The Road to Xanadu” (1927 г.) пишет о творчестве и приводит примеры великих открытий. Он устанавливает в процессе творчества, открытия, три части, которые называет: ИСТОЧНИК, ВИДЕНИЕ и ВОЛЯ. В источнике находятся все факты, которые ожидают, чтобы их использовали...

Где, действительно, в каждый момент находятся все несчетные известные нам факты и миллионы пережитых нами эпизодов? Где бы это таинственное место ни было... ИСТОЧНИК — это только подходящее обозначение для тайны. И там, поглощенные некими неведомыми глубинами, для всех намерений и целей как бы не существовавшие — они покоились в глубоком сне. Некоторым было суждено дремать неделями, некоторым месяцами, а некоторым целые годы. И вдруг все сразу они пробудились..

Картина пришла в движение и начала разворачиваться помимо моей воли. На мгновение я просто позволил течь потоку образов. Затем я осторожно стал управлять им...

Без ВИДЕНИЯ хаос элементов остается хаосом. А ФОРМА спит вечным сном в просторном хранилище не родившихся замыслов. И в том хаосе лишь творческое ВИДЕНИЕ может различить эту ФОРМУ. Точно так же без содействия ВОЛИ, подчиненной этому ВИДЕНИЮ, воспринятая в сонме других картина не могла бы стать объективной реальностью.

То, что Лоувс называет тремя основными составляющими, очень напоминает описанные нами три части процесса проектирования. Его “воля” хотя и не является полным эквивалентом очистки, но близка ей.

Уровни проектирования

Проектирование автомобилей связано с взаимодействием многих процессов. На рис. 5.25 изображен нисходящий поуровневый метод проектирования автомобиля. На самом верхнем уровне мы пытаемся представить себе автомобиль как целое. На последующих уровнях составляются требования на двигатель, рулевое управление, приборную доску, и, конечно же, на этом уровне может быть еще очень много других составных частей.

В последующем мы начинаем делить на части все эти подсистемы. Перед нами начинает вырисовываться древовидная структура, в которую включается все больше подробностей. Это длится до тех пор, пока мы наконец не достигнем уровня тривиальности, на котором добавление дополнительных подробностей ничего нового не дает. Когда мы начинаем проектировать оконное стекло, или фары, или дверную ручку, мы очень скоро достигаем уровня, на котором все, что мы можем сказать, это — “мы получаем то, что нам было нужно”, будучи полностью *уверены* в том, что все будет сделано так, как нами задумано.

Выбор подсистем, на которые надо делить наш объект, это и есть акт проектирования, он оказывает глубокое влияние на нашу работу по созданию системы и на ее дальнейшее функционирование.

Такое поуровневое проектирование может быть обнаружено практически во всех областях, причем чем более отработана технология, тем более стабильна и надежна та информация, которая передается с одного уровня на другой. *Информация,* необходимая на любом уровне, обычно весьма понятна, также вполне понятен и отработан *механизм представления* этой информации (формат, язык и т.д.).

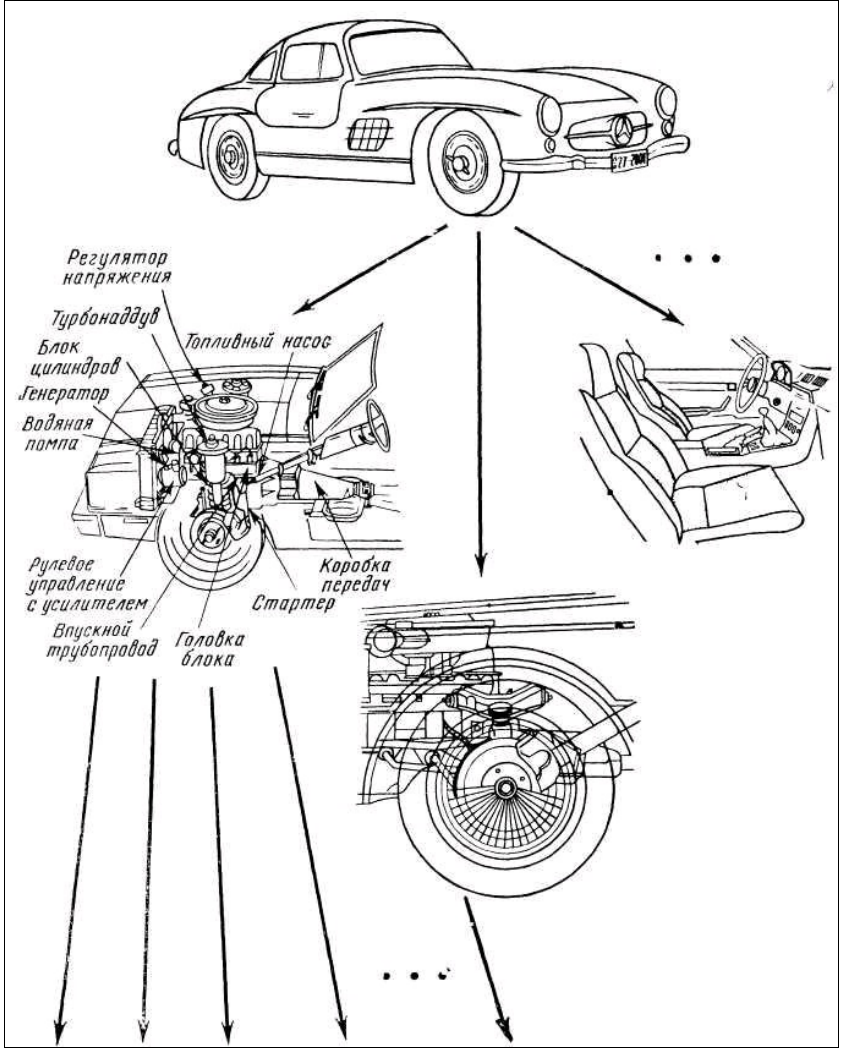


Рис.5.25 Переходы с уровня на уровень при проектировании.

В программном обеспечении ни один из уровневых механизмов не достиг нужной стабильности. Содержимое каждого уровня запутанно, а механизмы передачи информации с уровня на уровень ненадежны, или их вовсе нет. Тем не менее проектировщики нижних уровней часто соглашаются на фрагментарные или невразумительные требования/проекты, потому что *для спецификаций не существует никаких стандартов!* Более низкие уровни могут легко отойти от нужного подхода, и это с самыми лучшими намерениями.

Отсутствие "технологической инфраструктуры" приводит к тому, что процесс (или система) производства программного обеспечения не застрахован от ошибок. *Именно поэтому управление им должно быть гораздо более твердым, чем в отраслях с более отработанной технологией*.

Проектирование верхнего уровня

Программа управления, созданная для посадки на Луну, состоит примерно из 400 модулей. В операционной системе IBM — ОС для системы 370 в 1975 г. было более 3000 модулей.

Очевидно, что число различных способов их взаимодействия очень велико. В самом начале процесса отработки взаимосвязей наши возможности имеют поистине астрономические масштабы.

Мы делим функции больших программных систем на все меньшие единицы до тех пор, пока размеры модулей не достигнут порядка 30—50 операторов на языке высокого уровня. После этого начинаем связывать модули друг с другом. Затем их объединяют в подсистемы.

Дойдя до этого места, разработчики вычислительной аппаратуры могут сообразить, что все, о чем мы ведем речь, уже сформулировано в одном из принципов аппаратных систем, который гласит, что чем чище и определеннее связи между различными частями системы, тем легче будет в дальнейшем произвести разделение работ и получить работающие и проверочные подсистемы. Чем более замкнутой является каждая отдельная часть, тем легче будет выполнить всю работу. То же самое мы делаем и с программами, называя результат этого процесса "упрятыванием информации". Скоро мы расскажем об этом подробнее.

При разработке программного обеспечения замкнутость приобретает еще большую важность, поскольку программисту бывает очень легко перешагнуть "границы" программных модулей и забраться в модули, принадлежащие кому-то другому. Нам нужно планировать взаимодействие между компонентами и определять то, что нужно выполнять с помощью подпрограмм.

Связывание и разделение мы проводим на основе расслоения. Все связи между подсистемами должны быть точными и ясными. Это значительно облегчит тестирование, поскольку мы получим возможность сначала протестировать отдельные подсистемы и лишь затем, приступив к их совместному тестированию, сосредоточить внимание на взаимосвязях.

Расслоение

Одной из основ всех сложных предприятий является принцип "разделяй и властвуй", это оказывается верным и для программного обеспечения Расслоение облегчает задачу повторной сборки построенных блоков после внесения в некоторые из них исправлений Этот принцип очень стар. Он применяется во многих областях человеческой деятельности, в том числе — в технике. Это главнейший принцип, используемый при настоящем техническом подходе к делу.

Выделение макрослоев

Хотя обычно мы не рассматриваем операционную систему или систему управления базой данных в качестве отдельного слоя общей системы, по сути это на самом деле так. С каждой из этих подсистем мы связываем определенные функции, причем эти функции сосредоточиваются в ограниченной части системы.

Временные ограничения

Системы следует проектировать в расчете па пиковые ситуации — ситуации, которые обычно не возникают Если же возможность их возникновения реальна, то именно в этих случаях система особенно необходима (Очень интересно исследовать чистые затраты груда на имитацию пиковых ситуаций при тестировании системы. Мы обратимся к этой теме в разделе, посвященном тестированию). Необходимо идентифицировать, анализировать и изучать пиковые входные сигналы, выходные сообщения пользователям, "поведение" во всех взаимосвязях. Систему надо проектировать так, чтобы она могла управлять именно такой ситуацией.

Проектирование среднего уровня

Результаты проектирования программного обеспечения на среднем уровне могут быть проиллюстрированы на таком примере:

как нам разделить на части программу *обработки запросов,* чтобы работа была максимально эффективной?

В первом варианте решения (см. рис. 5.26) есть модуль "**одиночного обслуживания", который строит из всех** нуждающихся в нем очередь и "обслуживает" их по одному за раз.

Если очереди станут длинными, то такое решение приведет к задержкам, т.е. одновременно будет поступать много запросов. Второй вариант решения заключается в том, что в памяти машины одновременно находится несколько копий одного модуля, так что одновременно можно обрабатывать несколько очередей. Однако на это затрачивается много памяти.

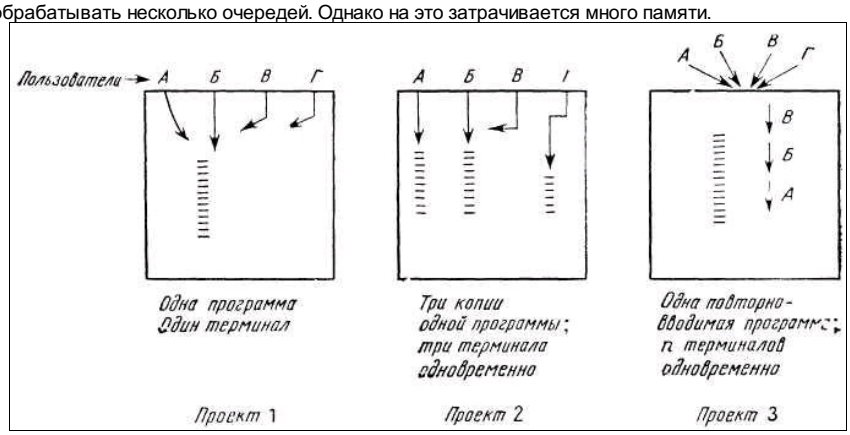


Рис. 5.26. Три разных проекта обслуживания запросов.

Третий вариант предлагает одну программу обработки очередей, которая одновременно может работать сразу с несколькими очередями. Эта программа является повторно входимой. Она может одновременно выполняться для нескольких пользователей. Вариант, который будет выбран группой проектировщиков, окажет значительное влияние на производительность системы.

Выделение микрослоев

Внутри модуля, служащего для управления линиями связи, можно иметь около 100 микромодулей размером около 30 операторов на языке высокого уровня или 30 модулей размером в 100 операторов. Каждый микромодуль должен быть максимально изолирован от других. Под этим мы, в частности, подразумеваем следующее:

1. Ни один модуль не зависит от других, кроме как в смысле последовательности исполнения; каждый модуль должен быть обеспечен всеми данными, кроме тех, которые нужно вводить с внешних устройств.

2. Все модули должны иметь минимальные размеры, лучше всего около 30 операторов на языке высокого уровня. Этим достигается микромодулярность.

3. Каждый модуль должен выполнять одну и только одну функцию.

Ключ к созданию легко изменяемого программного обеспечения заключается в том, что создавать его надо компактными сегментированными разделами. К счастью, для этого у нас теперь есть достаточно стандартные способы. Технические методы описаны в многочисленных публикациях, здесь мы ограничимся перечислением основных моментов.

Проектирование модуля

Мы продолжаем делить программные модули на все меньшие части до тех пор, пока не дойдем до таких частей, которые могут быть "записаны" с помощью команд. Каждую группу команд, которую можно "вызвать" для выполнения, мы называем *модулем*.

Проект модуля включает в себя:

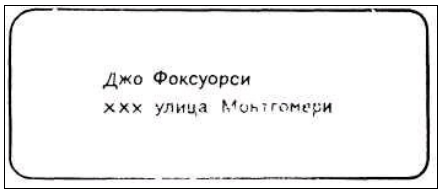
Размер модуля
Внешние данные
Содержание
Внутреннее строение
Внешние связи

Желательно иметь минимум внешних данных; этим определяется независимость модулей. Модуль, выполняющий единственную, хорошо определенную функцию, *строго функционален*.

Строго функциональные модули могут быть реализованы методом, позволяющим организовать упрятывание информации. Различные состояния данных, возникающие при выполнении функций модуля, известны только этому модулю, поэтому проектировщики и программисты, составляющие другие модули, не должны знать подробности работы данного модуля. Они будут лишены возможности пользоваться сведениями о том, как мы собирались написать данный модуль. Это удержит их от поспешных решений. Вот так-то!

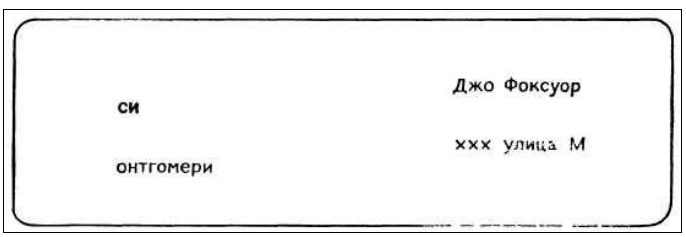
Упрятывание информации

Принцип *упрятывания информации* мы проиллюстрируем следующим примером. На языке Бейсик команда печати, в которой за символом @ следует число, например PRINT@ 482, отсчитывает заданное число позиций строка за строкой начиная с самой левой позиции верхней строки и печатает некоторое значение в 482-й позиции. Если я программировал в расчете на использование экрана шириной в 32 позиции, то команда PRINT® 69 вызовет пропуск двух пустых 32-символьных строк и выдачу на экран с отступом на 4 шага вправо. Чтобы напечатать



я должен написать: PRINT@69, N (имя) PRINT® 69, A (адрес)

Но, если используемые нами экраны сменились на 79-символьные, программа станет работать неверно. Я получу в этом случае что-нибудь вроде этого:



Чтобы работать с новыми устройствами, надо менять программу.

Приведенный пример тривиален, но представьте, что нам нужно менять сотни программ, работавших в течение многих лет. Это уже проблема! Многих программистов уже нет; другие заняты в других важнейших делах. Ответ напрашивается сам собой — не будем ставить никаких новых дисплеев! Даже если они более дешевы, более надежны и т.д.!

Обходным путем решения подобной проблемы может быть написание программ, *не зависящих* от числа позиций на экране. Программист просто следует соглашению, по которому текст начинается: 1) с самого начала или 2) отступя 1/8 часть строки, и т.д.

Системные программы берут указания о том, где надо начать текст, преобразуют их к фактическим значениям в соответствии с фактически подключенным дисплеем, и лишь затем будет выполняться наша программа. Теперь при смене физического дисплея нужно

будет изменить только системные программы. Если у нас сотни прикладных программ, выгоды налицо. Возможность внесения изменений планируется заранее, (см. рис. 5.27.)

Почему пользуются упрятыванием информации? Во время разработки на него приходится тратить дополнительные средства— оно удлиняет время проектирования и реализации, оно требует затрат и во время использования,— тратится дополнительная память, не очень много, но все же тратится.

Где же оно помогает? Во время сопровождения модификация таких программ будет проще. Мы построили систему, состоящую из блоков, которые легко вытаскивать и исправлять, причем любое изменение оказывает на другие блоки минимальное воздействие.

Рис. 5.27 Зависимость от внешних устройств и упрятывание информации в программах.

Производственные здания можно строить с капитальными внутренними перегородками или с раздвижными стенами. Наши действия при этом будут различными. Различными на первом этапе строительства, различными и в то время, когда нужно будет здания перестраивать.

Если мы *знаем,* что в программы придется вносить очень небольшое число изменений и что ошибок в программе осталось мало, в этом случае во время разработки, возможно, и не будет особых причин связывать себя дополнительными ограничениями для упрятывания информации.

Приведенный пример упрятывания информации можно относить и к проектированию среднего уровня, и к проектированию нижнего уровня в зависимости от того, сколько информации высвечивается на дисплее.

Если “выдачей” имени и адреса все и ограничивается, я склонен называть это нижним уровнем. Если же нам нужно выдавать на дисплей 40 переменных да еще в восьми различных форматах, это будет уже средний уровень. Опять мы сталкиваемся с положением, когда отсутствуют какие-либо метрики.

Проектирование нижнего уровня

Сам акт проектирования — выбор частей, на которые надо делить программу, и их расположение — продолжается вплоть до самых маленьких программ. И для самых маленьких программ необходимо определить требования, провести проектирование, написать команды, тестировать их и, наконец, выполнить всю программу. Мало, очень мало организаций, практикующих на самом нижнем уровне разделение процесса написания команд и проектирования. Очень редко группы проектирования *полностью* доводят проекты до такой точки, что людям, пишущим команды, не остается делать ничего, кроме тривиальнейшего преобразования. Когда такая точка достигается, человек, пишущий команды, уже не выполняет никакого проектирования и становится кодировщиком. Это определение имеет экономический подтекст.

Программист — это профессионал; кодировщик — это обыкновенный конторский служащий. Такова моя точка зрения; этот вопрос, однако, все еще обсуждается. Министерство труда имеет промежуточное мнение. При чтении циркуляра министерства может создаться впечатление, что все программисты считаются конторскими служащими, которые не обязаны в своей работе проявлять творческую активность. Им, следовательно, *нужно* платить сверхурочные!

Решением флоридского суда было признано, что три программиста, требовавшие от своего бывшего работодателя сверхурочных выплат за несколько лет, имеют право не только на возмещение убытков, но даже на их возмещение в тройном размере!

Можно себе представить тот хаос, который начнется, если 10 тыс. программистов фирмы IBM вдруг потребуют оплаты нескольких лет сверхурочных работ в трехкратном размере!

Циркуляр министерства труда только на первый взгляд считает программистов не творческими работниками; циркуляр написан весьма неопределенно, он, очевидно, написан человеком, который не понимает, что такое вычислительная машина и что делают программисты!

Ключевое различие между конторскими служащими и творческими работниками, применяемое в министерстве труда, основано на требовании принимать в своей деятельности ответственные решения. Проектирование и состоит в принятии ответственных решений.

Чтобы стать инженером, нужно окончить университет и пройти специальные тесты. Не так обстоит дело в мире программирования. Поскольку акт проектирования очень тяжело поддается описанию, обучать этому делу так же нелегко. Чтобы стать программистом, не обязательно быть выпускником факультета прикладной математики. Но принимать ответственные решения программисты должны, поскольку они должны вести проектирование. Кодировщики этого не делают.

Процесс проектирования даже на самом нижнем уровне, на уровне кодирования, проектирования алгоритма и структуры данных, остается творческим процессом. Если кому-то нужно написать программу в 100 команд, он должен проектировать и всю программу, и ее отдельные части.

Как организовать программу, где расположить константы, где организовывать проверки — после тела функции или перед ним,— все это принимается во внимание на нижнем уровне. Каким образом должен я разделить 3000 операторов языка высокого уровня на отдельные модули? Это и есть принятие ответственного решения.

Межуровневые механизмы и уровень тривиальности

Рассмотренные нами этапы процесса проектирования можно обнаружить во всех отраслях человеческой деятельности — в медицине, авиационной промышленности, мостостроении, строительстве и т.д. И в каждой отрасли имеется свой, специфический для нее момент тривиальности.

Это такой момент, когда становится бесполезным дальнейшее уточнение деталей, поскольку все уже знают, как это делается, и дополнительные подробности лишь вносят путаницу при реализации.

Момент тривиальности обычно зависит от сочетания двух факторов — развитости данной отрасли (какое число людей и в течение какого времени работают в ней) и сложности (сколько сил потребуется, чтобы четко описать все, что нужно описывать). Очевидно, что эти факторы воздействуют друг на друга, причем сложность тормозит развитие отрасли.

Разработка программного обеспечения находится в самом начале пути. Мы уже видели всю ее сложность, к тому же это очень молодая отрасль. Возникла она недавно, занято в ней не так уж много людей.

По мере совершенствования отрасли практические методы и конкретные сведения, передаваемые с одного уровня на другой, изучаются и проверяются опытным путем. Со временем они становятся четко определенными. Если накопленный опыт игнорируется или просто плохо используется, проектировщик нижнего уровня сразу заметит, что спецификация имеет серьезные дефекты. Его реакцией на плохой проект будет комментарий: “Это несерьезно!”.

В программировании опыт и практические навыки еще не накоплены. Сталкиваясь с недостаточно подробными, неясными или неточными описаниями, проектировщик нижнего уровня не отвергает их, а с большей энергией принимается за реализацию. Не удивительно, что в результате могут возникать и задержки, и отклонения от целей, и путаница, и неудачи.

Из-за отсутствия стандартов, регламентирующих взаимодействие между уровнями, руководить разработкой программного обеспечения приходится с мучительной точностью. Каждая межуровневая связь, каждый уровень имеют слишком много возможностей для развития в самых случайных направлениях.

Когда мы ведем проектирование в области хорошо нам знакомой, когда мы уверены в успехе, мы доводим наш проект до

определенного уровня и останавливаемся, оставляя дальнейшее проектирование менее квалифицированным специалистам. В случае достижения уровня, на котором все оставшиеся работы настолько просты, что ничего плохого уже не может произойти, мы говорим, что достигли уровня тривиальности.

Различия в уровнях тривиальности, которые мы можем наблюдать в разных областях человеческой деятельности, связаны с тем, что некоторые из этих областей достигли большего развития, чем другие, а также с разной степенью подготовленности и компетентности проектировщиков и технического персонала, их способности завершить проект после выхода на тривиальный уровень.

Так, например, при проектировании нового спортивного автомобиля руководитель группы проектировщиков может без всякого сомнения передать работу над структурой двери "деталировщику" Он может быть абсолютно уверен в том, что полученная в результате дверь будет вполне приемлема и работоспособна.

Уверенность руководителя в таком исходе основана не только на том, что в его распоряжении имеются достаточно компетентные "деталировщики". Много лет руководитель занимался тем, что разрабатывал стабильный и развитый механизм определения требований к двери, это позволяет ему не сомневаться в том, что "деталировщик" сможет обеспечить выполнение всех условий и достижение всех целей, указанных в спецификации.

В области разработки программного обеспечения деталировщиков еще мало. Огромная потребность в проектировщиках приводит к тому, что всякий, кто может хоть как-нибудь вести проектирование, быстро переводится на роль главного проектировщика в каком-либо другом проекте, часто это случается чересчур быстро, что приводит в действие принцип Питера *{О принципе Питера см. Иностранная литература, № 8, 1971 год.— Прим. перев.}*. Кроме того, недостаточно разработаны процедуры и правила описания условий и целей. Этот двойной недостаток создает большую путаницу, поскольку проектировщики не осознают, что они не связаны между собой, и бывают очень удивлены и даже взбешены, когда выясняется, что программы, считавшиеся ясными и полностью определенными, возвращаются к ним неузнаваемыми.

Обнаружение только что описанных нами недостатков есть первый шаг, помогающий избежать возникновения таких проблем. Кто предостережен, тот вооружен; он должен проявлять гораздо больше осторожности, вести более тщательный надзор, чем это делалось им ранее.

Связующих механизмов, работающих между разными уровнями проектирования в других отраслях, в программировании не существует. Тем самым проект большой системы программ должен быть значительно более детальным, чем мы обычно считаем необходимым. Когда кто-нибудь просит дать ему взглянуть на проект системы, его надо обязательно спросить: "На каком уровне?".

Однако и в программировании тоже имеется свой уровень тривиальности. Но он лежит значительно ниже, чем в большинстве других технических областей. Тот факт, что связующие механизмы несовершенны, вынуждает проектировщиков на смежных уровнях постоянно следить за тем, насколько точно совпадают их усилия. Если это не так, мы часто становимся свидетелями того, что возникают неработоспособные проекты либо проекты с многочисленными проблемами.

Вариации уровня тривиальности и межуровневых механизмов

Одна и та же работа может иметь разные уровни тривиальности в зависимости от того, кто будет вести ее реализацию — новички или опытные эксперты. То, что приемлемо в одной ситуации, в другой приведет просто к катастрофе. Аналогично, если моя команда уже провела в прошлом достаточное число разработок, я могу чувствовать себя более уверенно и меньше обращать внимание на межуровневые механизмы.

Структурное программирование

Структурное программирование является одним из элементов, относящихся к проектированию. Оно используется для управления связями, определения структур как больших, так и маленьких программ. Существует очень большая путаница по поводу того, что является, а что не является структурным программированием. Многие путают реализацию сверху вниз (см. с.220—221) со структурным программированием. Некоторые смешивают со структурным программированием концепции главного программиста и сквозной контроль.

Х.Милс, Р.Лингер и Б.Уитт написали книгу под названием "*Структурное программирование" {Richard C. Linger, Hanan D. Mills, and Bernard I. Witt "Structured Programming: Theory and Ptactice (Reading, Mass.: Addison — Wesley, 1979).},* в которой использовали такое определение:

Структуризованная программа — это программа, составленная из фиксированного базового множества первичных программ.

Первичная программа — это простая программа, не имеющая простых подпрограмм, состоящих более чем из одного узла.

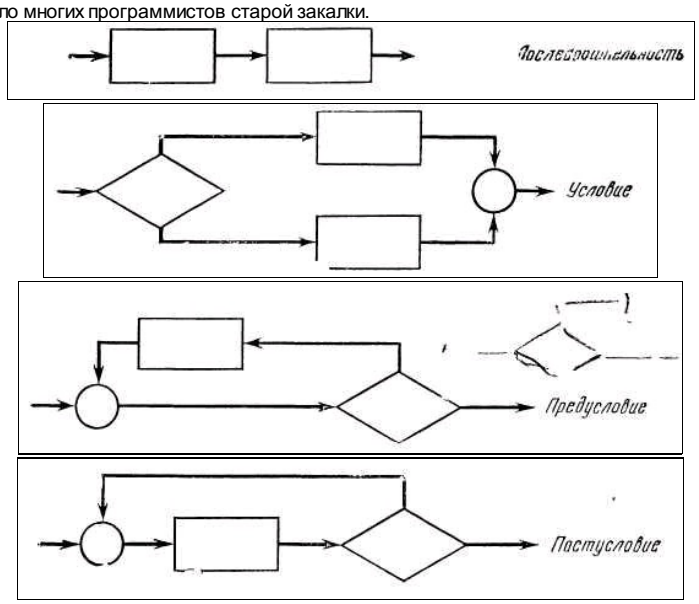
Простая программа — это программа, которая:

1) имеет один вход и один выход, и

2) для каждого узла существует путь от входа до выхода, проходящий через этот узел

Это хорошее определение, поскольку оно позволяет определить базовое множество структур в отдельности от программ. (См. рис. 5.28.) Суть дела здесь заключается в том, что если программное обеспечение строится только из первичных и простых программ, то логика и сам ход процесса ее выполнения значительно проясняются благодаря структуризации. Использование таких (готовых) структур дисциплинирует разработчика программ, что в результате приводит к появлению более понятных программ, в которых, следовательно, имеется меньшее число ошибок. К тому же структуризованные программы гораздо легче читать. Сложность программы также "структуризуется".

Полная свобода, которой обладали программисты, подверглась ограничению — это привело к повышению ясности, четкости, к выработке стандартов и напугало многих программистов старой закалки.



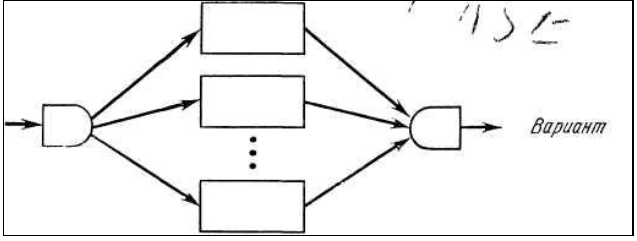


Рис. 5.28. Основные конструкции структурного программирования.

То, что структурное программирование идет сверху вниз, это случайность. То, что структурное программирование легко приспосабливается к методам сверху вниз и пошагового уточнения, можно отнести к его дополнительным достоинствам, однако существо структурного программирования не в этом.

Мы включили обсуждения структурного программирования в раздел, посвященный проектированию, а не написанию программ потому, что представление хода выполнения программы в структуризованном виде и передача людям, пишущим программу, информации тоже в структуризованном виде есть не программирование, а проектирование. При написании программы тоже проводится некоторое проектирование, при этом программы нижнего уровня строятся на основе уже знакомых нам базовых структур.

Если же проекты более высоких уровней выполняются раньше проектов более низких уровней и структуризация их не проводилась, то структуризация нижних уровней не приведет ко всем возможным выгодам. Общий ход выполнения программы будет по структуре напоминать блюдо спагетти.

Это может прозвучать здесь, на страницах книги, риторически, но структуризация делает программы гораздо более наглядными и понятными по сравнению с программами, не подвергавшимися этому процессу.

Структурное программирование придало программному обеспечению совершенно новый вид.

Введение в структурное программирование

Автоматизация хранения справочного материала (газетных вырезок) газеты “Нью-Йорк таймс” стала заметной вехой на пути развития программирования, и по праву. Выполненная Т.Бейкером из отделения федеральных систем фирмы IBM под руководством Х.Милса, она изменила направление развития программирования. До сих пор она часто упоминается в литературе. Было установлено несколько рекордов производительности. Были доказаны преимущества структурного программирования. Мы настаивали на использовании методов структурного программирования, и последующее обучение им 2600 наших кадровых работников основывалось на эффектных результатах работы с “Нью-Йорк таймс”.

События, происшедшие за кулисами этой работы, известны не так широко. Их стоит описать, поскольку на этом примере можно увидеть много интересного для руководителей подобными работами. Контракт с “Нью-Йорк тайме” с заранее установленной суммой достался фирме IBM в 1969 г. в результате соревнования с другими фирмами. Наши люди из Гейтсбурга были встревожены тем, что плата, запрошенная фирмой IBM, была на 800 000 долларов меньше того 1 млн., который запросили другие фирмы. В то время в Центр федеральных систем прибыл мой предшественник. Это был Генри Уайт, прекрасный, очень способный руководитель. Он организовал проверку, которая показала, что фирма не уложится в предполагаемую цену. Рассчитывали, что после исправления можно будет объединить некоторые уже существующие программы поиска с уже существующими системами, а они оказались несовместимыми! Было похоже на то, что потери составят около 800 000 долларов.

Уайт обратился к Х.Милсу. Милс только за год до этого поступил в фирму **IBM** после выдающейся карьеры в качестве математика и администратора. Он работал в “Дженерал электрик”, в Американской радиокорпорации, в Математическом институте. Он был среди других технических работников в распоряжении Уайта. Именно ему Уайт и поручил работу по контракту с “Нью-Йорк таймс”.

Милс сразу согласился, выбрал главным программистом Бейкера, а его помощником Мейера. Для ведения всех необходимых записей и канцелярских дел им в помощь был выделен библиотекарь.

Группа работала в Гейтсбурге, очень часто наведываясь в Нью-Йорк. Они выполнили свою работу — 83 тыс. операторов на языке высокого уровня — превосходным образом за 22 месяца, затратив около 300 000 долларов. Ошибок в системе практически не было, она была готова в срок, система была создана и заработала! Вот так-то! Мы сэкономили около 700 000 долларов!

Когда я (я вступил в свою должность в декабре 1969 г.) выдавал Милсу, Бейкеру и другим крупную денежную премию, я обнаружил, что во всей фирме IBM никто не имеет никаких статистических данных по производительности труда! Сколько строк программы считается нормальным написать за один человеко-месяц? Никто этого не знал. Ни у кого даже не было точного определения “строки программы”. Все были согласны с тем, что они заслужили премию, но во сколько раз они повысили производительность труда — в 5 раз (как считал я) или всего в 3? В результате мы решили внедрить в Центре федеральных систем систему измерений программ, которой постоянно пользовались в течение более чем 8 лет. Собирались статистики и данные по всем аспектам проектов; база данных содержит ныне колоссальное число данных по 100 проектам. Это единственное в своем роде собрание из всех, с которыми мне приходилось сталкиваться! Мы обратимся к нему позднее в этой же главе.

Мы постановили, что при разработке всех новых проектов надо пользоваться методами структурного программирования. К моему изумлению, реакция опытных программистов была крайне отрицательной! “Что понимает в программировании этот Милс?” “Работа для “Таймс” это счастливая случайность!” “Все сделал Бейкер”. “Милс и Бейкер — это не средние люди”. С этим-то я согласен! Но мы настояли на своем, и это было правильное решение. Мы истратили уйму денег и сил на обучение более чем 2600 человек. Две недели занятий с 2600 людьми составляют 100 человеко-лет! **Нет никаких** сомнений в том, что это было **правильное решение**

Причиной того, что работа для “Нью-Йорк тайме” была выполнена с такой высокой производительностью, было то,, что Милс и Бейкер еще в 1970—71 г. использовали множество самых передовых методов разработки программного обеспечения. Среди них:

структурное программирование,
метод главного программиста,
метод реализации сверху-вниз,
использование библиотекаря,
сквозной контроль,
формальные грамматики.

Именно успешный исход работы для “Таймс” породил путаницу, связанную с тем, что многие авторы стали включать все эти методы в одну категорию под общим названием структурного программирования.

Преимущества структурного программирования

Структурное программирование внесло порядок в царство анархии. Руководителю работ и инспектору оно подарило надежность! Наглядность. Наглядность стала делать программное обеспечение более управляемым! Обозримым! Стал возможным сквозной контроль! Появились стандарты и возможности обмена программами. Были прояснены сложнейшие проблемы. Разработка программного обеспечения вступила в новую эру.

Приемлемость структурного программирования

Многие программистские организации публично объявляют свою верность структурному программированию — и не пользуются его методологией. Некоторые до сих пор путают со структурным программированием другие хорошие методы. Большинство не знает точно, что это такое.

Любая достаточно крупная команда разработчиков программного обеспечения, не пользующаяся четко определенными методами структурного программирования, не может считаться достаточно компетентной. Мы вернемся к этому вопросу и продолжим обсуждение причин сопротивления структурному программированию в гл.6.

Хорошее проектирование

Хорошее проектирование приводит к ясному проекту; в таком проекте имеется внутреннее единство. В своей книге “Мифический человеко-месяц” Брукс *{Brooks F.P. Jr., The Mythical Man-Month (Reading, Mass.; Addison-Wesley, 1975). Русский перёвод см. примечание на стр. 5.}* упоминает европейские соборы, строившиеся по нескольку веков. Наиболее эффектны из них те, в которых видна идея самого первого архитектора. К “плохим” относятся те соборы, в которые каждое столетие вносилось что-нибудь от очередного строителя. Многие языки программирования не прижились потому, что их создатели попытались объединить в них слишком много разноречивых свойств. В связи с этим Брукс упоминает язык PL/1.

С концепцией единства проектирования согласны и Миле, и Уитт, и Лингер, с ней согласны также Йенсен и Тонис *{Randall W. Лельеп and Charles C. Tonies, Software Engineering (Englewood Cliffs, N. J.: Prentice Hall, Inc, 1979).}*. Мой опыт говорит, что автором проекта больших систем является обычно один человек. Этот человек определяет направление работ.

Как говорится, о вкусах спорить не приходится. Система диспетчеризации воздушных перевозок доводилась до рабочего состояния в течение 10 лет (совершенно завершена эта работа не будет никогда), за этот период времени заказчик провел немало разных ревизий. Одна ревизия, которую проводила группа независимых экспертов по программному обеспечению, констатировала, что проект управляющей (системной) программы выполнен "архаичными методами и неправильно". Несмотря на это, после завершения работ программа прекрасно работала. (Но скольким толкам дал пищу этот отзыв!)

Некоторые руководители разработок программного обеспечения толкают своих программистов на постоянное использование новейших, самых "привлекательных" методов. Они пытаются привычные вещи делать новыми способами или находить новые работы. Этих программистов снабжают инструментарием, специально проектировавшимся с расчетом на творческое использование. Мы должны называть такие работы не разработками программного обеспечения, а научными исследованиями.

С другой стороны, находятся такие руководители, которые уже закончили проект и теперь хотят, чтобы были написаны программы размером в 400 000 строк или около того. А после того как программы будут написаны, их надо будет отправить пользователям и заняться их сопровождением. Этим руководителям не нужна ни привлекательность, ни изобретательская деятельность. Им требуется надежность, ясность, воспроизводимость. Им нужны простые, незатейливые программы.

Проектирование это...

Проектирование — это все же искусство и что-то еще, что мы еще недостаточно понимаем. Стравинский называл искусство "творчеством, требующим скачка воображения". Может быть, и существуют проекты, которые не являются произведениями искусства, а возникают в результате простого следования образцам — проекты мостов, зданий, одежды и т.д. — но нужно ли нам называть их проектами?

Итеративность

Мы обнаруживаем то, что не работает, и исправляем эго место. Когда мы первый раз выполнили мультипроцессорную обработку в системе диспетчеризации воздушных перевозок, то обнаружили, что процедура блокировки (дающая гарантию того, что ни второй, ни третий, ни четвертый центральные процессоры не смогут работать с той записью, которая обрабатывается в данный момент, и не нарушат достоверность данных) работала слишком уж хорошо. Нашим центральным процессорам приходилось лишь лениво дожидаться, когда будет снята блокировка. На втором проходе мы укоротили все процедуры, выполнявшиеся в режиме "блокировки".

Компромиссы

При проектировании возникает много компромиссных решений, одно за счет другого — например увеличение занимаемой памяти и за счет этого повышение скорости выполнения программы. Так же как авиаконструктор постоянно жертвует ради увеличения веса и скоростью, и экономичностью двигателя, так и проектировщик программного обеспечения должен принимать во внимание все 12 свойств программ во всех их взаимоотношениях и еще множество других характеристик, которые связаны и с программой, и с теми функциями, которые ею автоматизируются. Что произойдет с остальной частью системы? Что изменится в радиолокаторе? В окружении? Не поглотят ли наш банк другие, более крупные банки? Или его просто вытеснят из данной сферы деятельности?

Проектирование больших программ типа V — это многогранная деятельность

Мы уже видели, что в работе над созданием программы для приложений типа V гораздо меньше требуется усилий, чтобы написать, чем для того, чтобы понять поставленную задачу и свои возможности, сформулировать требования, понять их и объяснить проектировщикам и кодировщикам. Для понимания требований обычно нужны опытные, профессионально подготовленные специалисты, которые либо ранее занимались автоматизацией этой же области, либо имеют достаточную квалификацию, чтобы воспринимать информацию от тех, кто выполняет работу в настоящее время.

Если мы, например, заняты автоматизацией управления посадкой на Луну, то в нашу группу необходимо включить тех, кто разбирается в физических процессах прилунения, кто понимает гравитационные явления, происходящие на Луне, знает, как подействует удар при посадке на двигатель и т.д.

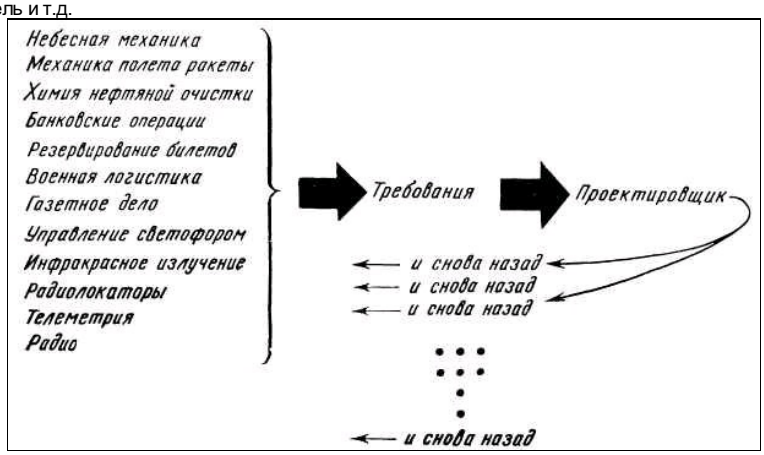


Рис. 5.29. Специальные знания, необходимые при проектировании.

Эти знания требуются постоянно. Неверно представлять себе, что нам удастся в самом начале выяснить все, что нужно, и затем пользоваться только этим. Эти знания являются зеркалом, в котором отражается все, что мы делаем. Нам не удастся просто *написать* требования раз и навсегда. Мы будем постоянно возвращаться от требований к проектированию, от проектирования к программированию и назад к требованиям. Специальные эксперты должны контролировать проектирование, его правильность надо проверить до того, как начнется собственно программирование.

Процесс, изображенный на рис. 5.29, выглядит красивым и аккуратным, однако вряд ли возможно с помощью стрелок изобразить умственные усилия, необходимые для успешного его завершения. Нельзя также изобразить тот горячий спор, эмоциональный всплеск, который возникает, когда эксперты (точнее, выдвигаемые ими требования) отвергают некоторый подход (проект) как "рискованный", или "запутанный", или "неверный".

Деталировка — это примитивное проектирование

У нас был один проект, который принес нам убытков на 3 млн. долларов. Это была система редактирования данных в реальном времени, которую мы делали по заказу Национального управления по аэронавтике и космическим исследованиям, причем цена была заранее установлена. Х.Милс постоянно говорил мне, что проект системы отсутствует; исполнители же говорили, что работают по проекту.

Я назначил на один из дней проверку, и исполнители работ докладывали в течение 10 ч.! У них было огромное количество графиков, расчетов, бумаг и данных.

Через 10 ч. ответ был ясен! У них был проект, метод, казавшийся вполне осуществимым. Все это было сосредоточено в двухметровой куче графиков и в головах двух ведущих сотрудников. Для нас, слушателей, проект оставался недоступным. Некоторые проблески его мелькали перед нами в течение этих 10 ч. Мы убедились, что проект существует, но увидеть его мы не могли, никто из нас не мог его отчетливо сформулировать.

Милс, Лингер и Уитт назвали это "деталировкой". Она может служить в качестве проекта, но на самом деле является лишь его бледной тенью. Деталировка часто "удовлетворительна" — однако выбирается всегда первое попавшееся решение, поиски более хороших решений не производятся.

Устойчивые программы, или программы, дружественные пользователю

Очень многие программы после работы групп определения требований и разработки передавались заказчику ("пользователю"), а затем никогда не использовались. Они были слишком сложны в обращении либо неверно выполняли требуемые функции. Мы уже рассказывали об инциденте в Джексонвилле, когда центром диспетчеризации была отвергнута программная система, выполнявшая

слишком много функций. Здесь мне хотелось бы ненадолго остановиться на той части программы, которая управляла взаимодействием с человеком.

В тот момент, когда мы подключили людей к диалогу с вычислительной машиной (типы III, IV и V), задачи, стоявшие перед нами, значительно возросли. Мы должны писать теперь программы, которые могут давать людям возможность взаимодействовать — вести диалог с системой.

Я однажды работал с небольшой группой программистов — всего 7 человек,— занимавшейся созданием диалоговой программы по анализу многозначных решений. Программы писались на языке APL; их размер составлял 10 тыс. операторов языка APL — из них 9 тыс. операторов было написано для управления интерфейсом с человеком. Девять из десяти. Нам необходимо было сделать так, чтобы надежды пользователей оправдались, иначе нашей системой не стали бы пользоваться.

Если необходимо было ввести букву О, а пользователь набирал цифру 0 либо наоборот — мы не ограничивались тем, что сообщали "неверные данные на входе". Мы давали подсказки, чтобы помочь поддержать процесс активным. "Вы хотели ввести символ нуль или О?" — выдавали мы на экран.

Проектирование этой части программы имеет решающее значение и очень сложно. Обычно после того, как пользователь получит возможность опробовать систему, эта часть подвергается значительным переделкам. Она имеет наибольшее значение для того, чтобы пользователь принял систему.

Документирование проекта

Если отсутствует документация, доступная для всех, имеющая несколько уровней детализации, проекту грозит беда. Под несколькими уровнями детализации я подразумеваю то, что проект может быть описан за 30 мин, за 2 ч, за 8 ч, или за 16 ч. Если имеется только 30-мин. обзор или 16-ч. версия и ничего между ними, остерегайтесь. Почему?

Потому что проект начинает существовать только в голове одного, в крайнем случае двух проектировщиков. Мы лишены возможности ухватить идею и следить за развитием проекта на уровне подробностей 16-ч. версии, если только мы не станем заниматься этим проектом все свое время и не познакомимся с ним достаточно близко.

Наша группа по разработке системы выпуска газет США имела два уровня проекта — 30-мин. и 16-ч. Я знал и руководителя работ, и проектировщика — соответственно Б. Уитта и Т. Бейкера. Они не знали неудач и в прошлом уже выполнили просто фантастические работы. И я доверял им! Но что случилось бы, если бы кто-нибудь из них заболел или уволился? На карту была поставлена судьба целого проекта — миллионы долларов.

Проекты таких больших систем должны быть документированы, чтобы их можно было контролировать на всех уровнях. На это нужны и деньги и таланты, но не делать этого — значит, играть с огнем.

Спецификация — это проектирование и одновременно выработка требований

Проект отвечает на вопрос КАК; требования отвечают на вопрос ЧТО.

Если на листке бумаги я напишу одно лишь слово "автомобиль" и попрошу вас достать мне такую штуку, содержанием надписи будет *и* "что", *и* "как". Для вас это будет "что"; это автомобиль, вам надо пойти и достать его Для меня это будет "как", некоторая определенная конструкция, отличная от мотоцикла, грузовика, велосипеда и всех других механических транспортных средств. Я буду путешествовать "как"? На автомобиле.

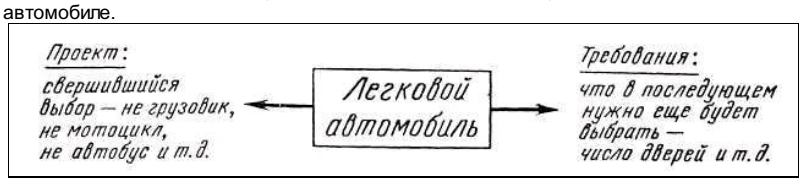


Рис. 5.30. Спецификация — это требование и проект.

С одной точки зрения некоторая вещь кажется проектом; с другой — она же кажется требованием, (см. рис. 5.30.)

Если уж слово "автомобиль" может быть и требованием, и проектом, то и подавно документ в 100 страниц может быть и проектом и требованием.

Я не могу описать все часы интенсивной борьбы на самых высоких уровнях, которые пришлось мне испытать, решая важную проблему, чем является некий документ — проектом или требованием. Он выполняет обе функции!

Новые методы проектирования

В нашем распоряжении имеется чересчур много новых методов проектирования. Многие из этих методов являются в сущности методами записи, а не ведения процесса. Они достаточно ценны, поскольку любая помощь при проектировании только приветствуется. У них множество имен: метод Вернера — Орра и Джексона, метод HIPO, SADT (технология структурного анализа и проектирования), IA (информационный анализ), структурное проектирование — все они лучше, чем отсутствие какого-либо формализма при проектировании. Все они большей частью связаны с документированием и демонстрацией проекта, чем с его созданием, причем в значительной степени это относится к последним двум стадиям — структуризации и представлению, чем к первой — созданию.

Данные и процесс

В большинстве методов делается упор на процесс и лишь затем переходят к данным. Немного найдется сторонников обратного перехода — от данных к процессу.

С чего начинать — с данных или с процесса,— зависит от прикладной области или от проектируемой системы: что в данном случае имеет превалирующее значение. В больших системах типа III это данные, поэтому проектирование нужно начинать с них. Большие системы типов IV и V, вероятнее всего, следует начинать с проектирования процесса.

Структурное проектирование

Этим словосочетанием определяют столько различных методов документированного проектирования, что всякие слова тут практически бесполезны. Что значит структурное? Этот термин не имеет совершенно никакой связи с термином программирование!

Тот факт, что такое множество новых методов возникает именно в области проектирования, хотя бы только в его документировании, очень обнадеживает. Почти во всех методах имеются довольно строгие правила, говорящие, как надо отмечать *что* и *как,* а также определяющие последовательность переходов. Почти во всех имеются правила управления всеми уровнями детализации, которые могут потребоваться, и правила, помогающие пользователю избежать ненужных ему подробностей.

Известно, что обмен сложными идеями и концепциями является основной трудностью при управлении сложными системами, поэтому мы должны быть благодарны за любую помощь, которая оказывается нам в этой области. Большинство новых методов очень полезно именно в данном отношении, поэтому мы рекомендуем всем их использовать.

Трудности нововведений

Как и всякая новая человеческая деятельность, новые методы проектирования оказываются тоже очень трудными для введения. Изменениям сопротивляются; правила раздражают; строгости кажутся бесполезными. Но это бывает только в начале. Как и многие другие полезные новые методы, эти методы в конце концов признаются "единственно возможными", а на все старые при этом начинают смотреть с изумлением и весьма критически.

Это происходит со всеми новыми методами — как мы уже видели, не было исключением и структурное программирование. Человеку нужна по крайней мере неделя, а обычно даже две недели формального изучения *и* затем несколько месяцев тренировок, чтобы окончательно признать новый метод. Тут нужны контроль и принуждение. Весь наш персонал, занятый в информационных системах — программисты, аналитики и другие,— нуждается в таком "вдалбливании".

Если группа разработки и внедрения информационной системы состоит из нескольких сотен человек, цикл обучения получится весьма дорогостоящим. Его, однако, *необходимо* провести. Результаты настолько значительны, что пренебрегать новыми методами нельзя.

Написание программ — программирование

Мы приступаем к изучению деятельности, которую многие ошибочно считают разработкой программного обеспечения, а именно к изучению процесса написания программ. Это всего лишь часть общего процесса разработки программ. В больших системах эта часть является самой простой из всех шести частей разработки. В небольших проектах эта часть требует наибольшей доли времени.

В какой-то момент проектирование доходит до такого низкого уровня, что для того, чтобы заставить вычислительную машину выполнить то, что задумал проектировщик, достаточно написать соответствующие команды.

Для очень маленьких программ, которые пишутся одним, двумя программистами, единственной загвоздкой тут является выбор инструмента, т.е. языка, на котором следует писать. А при программировании больших программ этот вопрос также является одним из

ключевых.

Наше изучение процесса написания команд мы разделим на три части — выбор языка программирования, процесс доведения программы до выполнения и управление этими двумя процессами. Мы ограничимся *только* изучением деятельности по написанию маленьких программ. Написание небольшой программы (до 10 тыс. строк текста программы) не так уж сильно отличается от написания программы в 1 млн. строк; различия скорее находятся в других частях процесса их разработки. Поскольку различия в написании невелики, мы будем обсуждать проблемы, не уделяя внимания дополнительным обязанностям, имеющимся у программистов при программировании программ в 1 млн. строк. Конечно, различия все-таки есть, все дополнительные обязанности мы обсудим в следующем разделе, посвященном компоновке. Дополнительная нагрузка на программиста обычно заключается в некоторых ограничениях и устанавливает определения данных, констант и интерфейсов.

После того как программист выяснил алгоритм и структуру программы, он начинает писать команды.

Написание программы обычно выглядит как акт записи команд на листке бумаги, которые, возможно, будут переданы на машину. После того как команды будут записаны, их надо перенести на какой-нибудь носитель информации, с которого их сможет считать машина и с которого они могут быть загружены в вычислительную машину, чтобы проверить, выполняет ли программа то, что от нее требовали. Сотни тысяч программистов водят карандашом по бумаге и *пишут* команды. Они занимаются "кодированием". В настоящее время все большее число программистов вводят в машину команды, пользуясь терминалами, но большинстве делают это, уже имея готовый текст.

Важнейшие моменты, связанные с написанием текста программ, таковы:

1. Ясность того, что нужно сделать — это мы уже обсуждали.
2. Язык, который будет использоваться, и стандарты, которым необходимо следовать.
3. Процесс управления *преобразованием* программ с языка программирования на машинный язык.
4. Инструментарий для реализации пп. 2 и 3.

Языки

Лингвисты, занимающиеся семантикой языков, до сих пор продолжают открывать в языках новые свойства. Все еще продолжаются дебаты по вопросу, что важнее — разговорный язык или язык письменный. В книге "*Введение в теоретическую лингвистику"*Джон Лайонс *{J.Lyons. Introduction to Theoretical Linguistics. (London: Cambridge University Press, 1972).}* отмечает, что язык — это не просто свод правил, *лингв,* это и не собрание всех возможных устных и письменных форм *слов,* но и то и другое вместе. Даже неправильно произнесенное, синтаксически неправильное выражение может быть понято. Это происходит потому, что слушатель заполняет пропуски и исправляет ошибки. К сожалению, вычислительная машина и ее транслирующая программа (ассемблер или транслятор) делают *только* то, что им приказывают. Даже самые хорошие трансляторы имеют весьма ограниченный интеллект. Для языков вычислительных машин синтаксические правила приобретают решающее значение.

Наверное, не очень очевиден тот факт, что процесс получения команд основывается на трех важнейших положениях:

1. Вы должны понимать процесс, для выполнения которого пишутся команды.
2. Вы должны уметь сформулировать (выразить словами) последовательность команд.
3. Вы должны выбирать такие слова, которые будут поняты вашими собеседниками.

Мы часто предполагаем, что п.1 выполняется; что это не так, мы видели в разделе, посвященном определению требований. Мы также предполагаем, что человек, который может "выполнить процесс", может и облечь его в словесную форму, — это тоже не всегда так.

Теперь обратимся к третьему пункту — написание команд для конкретного исполнителя. В нашем случае исполнителем .оказывается вычислительная машина, но, прежде чем переходить к командам для вычислительных машин, я хотел бы обсудить некоторую аналогию.

Если бы я был великим нейрохирургом, я мог бы сформулировать инструкции для разрабатываемой новейшей хирургической техники, у меня не было бы трудностей по выдаче таких инструкций другим нейрохирургам. Введите в мою аудиторию обычных хирургов, и моя задача сразу усложнится. Добавьте туда студентов-первокурсников медицинского института, и я не уверен, что вообще смогу выполнить свою задачу. И так далее.

Так же и с вычислительной машиной. Словарный запас ее крайне примитивен. Заставить ее что-то сделать, сказать это на ее родном языке, называемом машинным языком, исключительно трудно. И все же мы должны осуществлять сложные логические построения именно *с таким словарным запасом,* поскольку только на такие слова машина и может реагировать.

Самые первые пользователи вычислительных машин сразу поняли, что вести дела таким образом невозможно, поэтому и "наняли переводчиков".

Возвращаясь к аналогии с нейрохирургией, заметим, что этот процесс выглядит так — выдающийся хирург будет теперь передавать инструкции посреднику, который переведет их на язык, *доступный* студентам-первокурсникам медицинских институтов.

Так же обстоит дело и с вычислительными машинами. Мы больше не заставляем программистов пользоваться машинным языком, а переводим их на работу с языком высокого уровня. Мы нанимаем в качестве переводчика *ту же самую* вычислительную машину, которая будет *выполнять* процесс *после* его перевода (или трансляции), и помещаем в нее программу, которая будет осуществлять перевод: либо ассемблер, либо транслятор, либо и то и другое вместе (рис. 5.31).

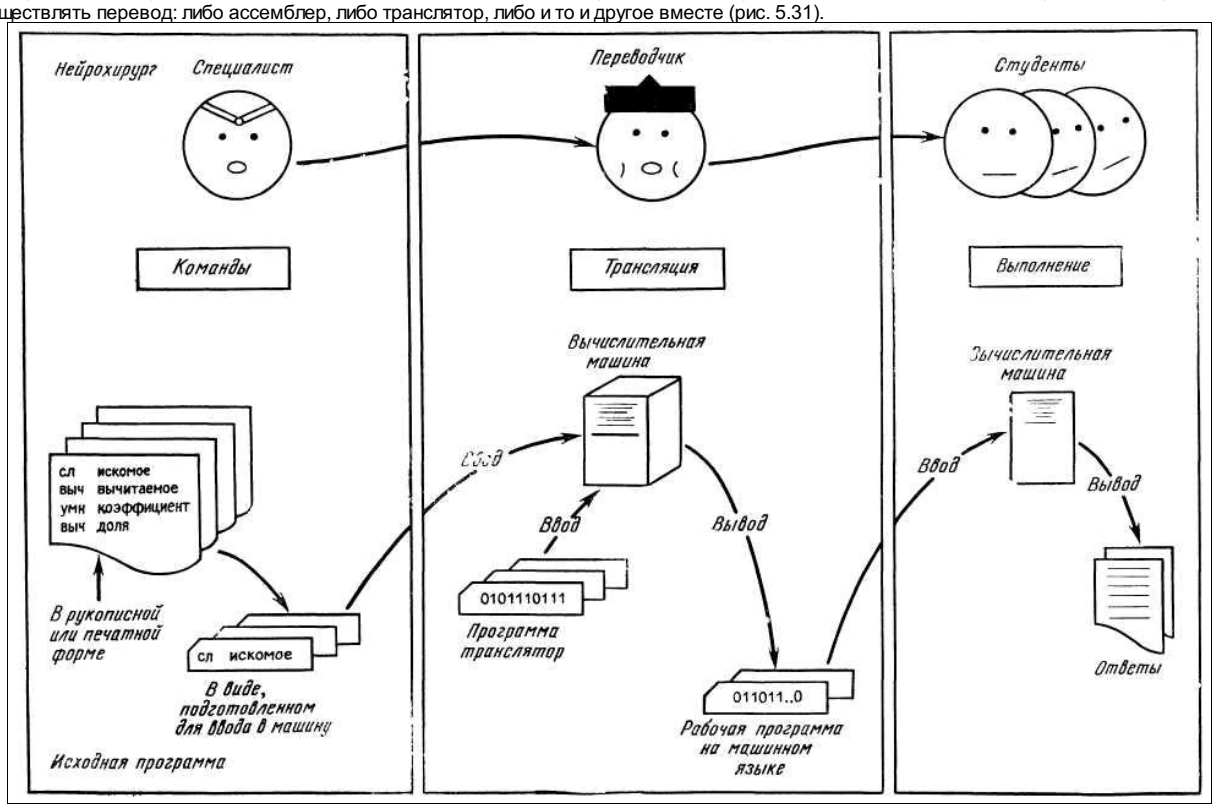


Рис. 5.31. Процесс перевода (трансляции).

Для того чтобы машина могла выполнить перевод, в нее должны быть введены программы *ассемблера* или *транслятора*.

Транслятор воспринимает операторы языков высокого уровня, т.е. Фортрана, Кобола, Паскаля, Бейсика, и "заставляет" вычислительную машину преобразовать эти операторы в команды на машинном языке, которые затем можно будет в свою очередь поместить в машину и выполнить с целью решить реальную задачу — например напечатать платежную ведомость.

Такой процесс выполняется по крайней мере в два этапа. В нем присутствуют этап трансляции и этап использования. *Пока* никто даже не приступил к созданию машины, которая могла бы сразу понимать языки высокого уровня *{Это утверждение не совсем верно. Коммерческих машин такого типа еще нет, но экспериментальные уже есть. — Прим. ред.}*.

Можно отметить, что в аналогии с выдающимся нейрохирургом он теперь не должен мучиться, объясняя свои операции на языке новичков. Так же и программисты. Языки высокого уровня позволяют им писать ближе к собственному уровню.

Мощность языка и связанные с ней трудности

Чем мощнее язык, тем труднее его изучить, так же как трудно выучить медицинскую терминологию и трудно заучивать слова с многозначной семантикой.

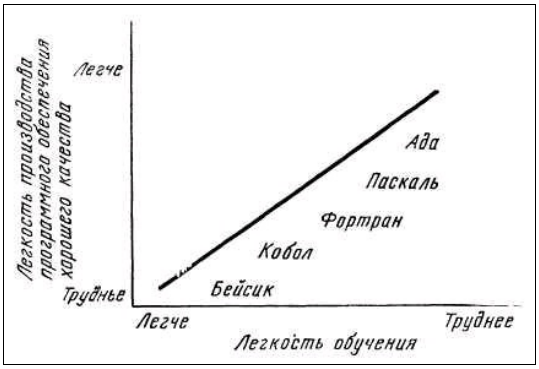


Рис. 5.32. Зависимость легкости обучения от мощности языка.

Если пытаться построить график зависимости легкости производства хорошего программного обеспечения от легкости обучения языку, получится нечто вроде того, что представлено на рис. 5.32.

Под хорошим программным обеспечением я подразумеваю хорошо скомпонованные, легко просматриваемые, хорошо документированные программы. Значит ли это, что нам никогда не следует пользоваться Бейсиком? Конечно, нет. Мы иногда пишем и такие программы, для которых не нужна ни легкость модификации, ни легкость расчленения.

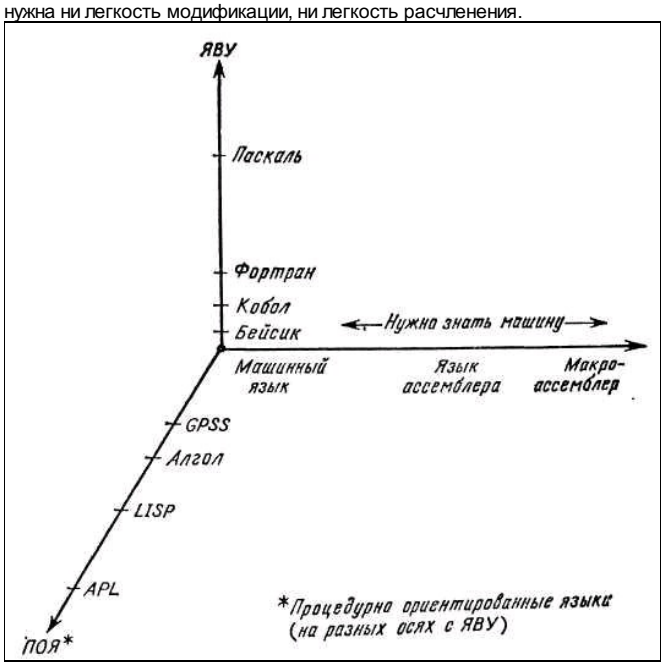


Рис. 5.33. Третье измерение — процедурно-ориентированные языки.

Итак, выбор языка зависит от того, что мы собираемся писать. На графике не отмечены языки APL и LISP. Эти языки применяются не для построения сложного программного обеспечения, а для решения задач. Программы на машинных языках, получающиеся после трансляции с данных языков, не имеют большого значения *{Более того, часто такие программы вообще отсутствуют, ибо эти языки, как правило, интерпретируются. — Прим. ред.}*; в большинстве случаев их даже не видят и не сохраняют. Я просто не могу себе представить, чтобы на APL кто-нибудь решился строить большую систему программного обеспечения. На рис. 5.33 изображены процедурно-ориентированные языки.

ПОЯ — другой тип языков. ПОЯ означает проблемно-ориентированный язык, или, иначе, процедурно-ориентированный язык *{В традиционной литературе принято различать эти два понятия. На процедурно-ориентированном языке описывается процедура получения решения любой задачи. Это универсальные языки высокого уровня. На проблемно-ориентированном языке формулируется сама задача. Естественно, что такой язык универсальным быть не может. — Прим. ред.}*. Эти языки также находятся на более высоком уровне абстракции, чем машинные языки. Их проектируют специально для решения какой-нибудь задачи или выполнения конкретной процедуры. Эти языки, следовательно, хорошо подходят именно для такого использования и не очень хорошо для других целей.

Среди этих языков заслуживают упоминания:

APL — решение задач

LISP — обработка списков

GPSS — универсальное моделирование

SIMSCRIPT — обработка текстов

ATLAS — язык министерства обороны США, применяемый в тестовом оборудовании

Целью этих языков является сокращение времени разработки программ. Делается это путем облегчения средств, с помощью которых создатель программы доводит программу до выполнения на машине. В этом смысле эти языки нельзя называть универсальными. Они реже используются при разработке регулярно используемых программ, чем при разработке программ, выполняемых либо единожды, либо очень редко.

Именно поэтому мы не будем уделять этим языкам чересчур много внимания.

Каждый язык представляет собой объединение словаря и синтаксиса. Словарь — это список слов, которые понятны транслятору. Синтаксис — это список правил расположения элементов словаря — порядок следования, знаки препинания, допустимые комбинации.

Когда вычислительная машина совместно с транслирующей программой воспринимает оператор языка высокого уровня, происходит сразу много разных действий. Начинается трансляция, а вместе с ней *расширение*. Когда я говорю своему помощнику:

“Пойдите в библиотеку и принесите мне копию X”, остается много не уточненных подробностей.

Помощник сам восполняет недостающие детали: какая библиотека, когда, как туда добраться, где находится X, как получить его и т.д.

Человек способен сам провести подобное расширение команды, поскольку он достаточно умен и хорошо осведомлен. Даже если он никогда не бывал в этом городе, он легко сможет выяснить, куда надо пойти, но *мне* говорить об этом все равно не было бы необходимости. Это же относится и к языкам машины и трансляторам, которые с ними работают. Многие трансляторы допускают использование неописанных переменных.

Многие языки программирования пытаются добиться эффективности в одной или в нескольких областях применения. Они могут облегчить человеку выражение глобальной идеи или сделать так, чтобы окончательный вариант программы исполнялся на машине как можно быстрее. Приспособлен ли язык к данной проблеме? К общению с человеком? С машиной? Или какой-нибудь комбинации этих свойств? Что нужно знать человеку для постановки задачи, ее решения, выполнения искомого процесса? Как думают люди? Как они общаются? Работают? Пишут? Ошибаются? Что может заставить машину работать более продуктивно? Всякую машину? А эту машину?

Вот потому-то десятки миллионов тратятся каждый год на проектирование и разработку новых языков программирования. Вот потому-то люди занимаются всю свою жизнь исследованиями в области языков для вычислительных машин.

Рост числа языков

К счастью для программистов, разработка языков программирования началась практически одновременно с появлением вычислительных машин. К сожалению, языков для них образовалось великое множество. В министерстве обороны США было подсчитано, что в его организациях используется около тысячи разных языков! Почему? Потому что программисты создают себе собственные языки программирования! И еще потому, что нет хорошего способа определить, какая версия лучше, а какая хуже, даже при использовании. И еще потому, что до сих пор языки оставляют желать лучшего.

В то же время наметилась некоторая стабилизация. Министерство обороны выбрало семь языков (инструкция 5000.31), на которых будут писаться все новые “*встроенные”* системы реального времени (Имеются в виду языки: Фортран, Кобол, Джовиал, CMS-2, SPS, TACPOL и TOS).

После длительного и глубокого изучения министерство обороны представило новый язык Ада, основанный на некотором расширении Паскаля, разработанного Н.Виртом. Если Ада докажет свою полезность, она будет включена в список инструкции 5000.31. Это очень мощный, очень сложный язык.

Однако программисты предпочитают старые языки. Сообщество программистов непоколебимо отказывается переходить на работу с новыми языками программирования. Подавляющее большинство программистов продолжают писать на Коболе и Фортране, несмотря на то что оба этих языка в настоящее время считаются устаревшими.

Почему же все старые языки продолжают существовать и остаются столь популярными? Они были изучены первыми! Тони Хоар в своем письменном докладе военно-морскому флоту констатировал следующее:

- *Привязанность программистов к существующим языкам сильнейшим образом контрастирует с энтузиазмом различных комитетов, утверждающих, что языки не отвечают поставленным требованиям. Причина? ...затрачено огромное количество времени на овладение языком, новый же язык представляет в этом отношении серьезную угрозу . . или, что еще хуже, новый язык может оказаться даже более простым для изучения и более эффективен в использовании, что может совершенно подорвать с огромным трудом завоеванное положение эксперта по старому языку...*

Таким образом, может получиться и так, что более легкий язык будет побежден сложным. Программисты из Советского Союза рассказывают аналогичные истории. По-видимому, одним из первых полученных ими языков был Кобол — *англоязычный* COBOL. Видимо, первые вычислительные машины попали туда из Соединенных Штатов *{Это утверждение не соответствует реальному положению дел и лишний раз доказывает, сколь мало знают в США о нашей стране. — Прим. ред.}* и программисты должны были учить английский язык, для того чтобы воспользоваться оператором PRINT. Англоязычный COBOL выжил! Потому что старики были настроены против русскоязычного Кобола. “Пусть новички помучаются”, Англоязычной версией Кобола пользуются до сих пор и в Германии.

После того как уже написано большое количество программ, перейти на новый язык трудно. Как говорят юристы, владение имуществом почти равносильно праву на него. Желание не расставаться с накопленным багажом неистребимо.

Язык и мышление

Язык, которым мы пользуемся, очень сильно воздействует на методику нашей работы. Многие утверждают, что язык формирует мышление. В книге, посвященной стилю письма и коротко названной “Стиль” Ф.Л.Лукас *{Lucas F.L. Style, (New York: Collier Books, 1962.)}* пишет:

Вот прекрасная история о человеке, зашедшем в кафе для иностранцев, в котором уже сидели группы англичан, французов и немцев. Человек этот обратил внимание на то, что англичане в полном молчании сидели вокруг своего столика, французы говорили все разом, а немцы слушали друг друга по очереди с таким напряженным вниманием, что на мгновение поразили нашего героя. Потом он понял — они ждали конца фразы, чтобы услышать глагол (или его отрицание).

Это же можно отнести и на счет языков программирования. Выбор языка оказывает на весь дальнейший процесс глубокое воздействие. Почему Вирт предложил Паскаль? Потому что ему надоело учить студентов языкам, которые “слишком часто нельзя было логически объяснить”.

Нотация, или система обозначений, помогает процессу мышления. Уайтхед и Рассел (“Principia Mathematica”, 1910) [1] объяснили, что хорошая система обозначений способна освобождать мозг от излишних деталей и переключать его на обдумывание других вещей. Мощные языки типа APL дают возможность программисту заставить машину выполнить огромную работу с помощью весьма немногих команд.

Конструкция языка ограничивает возможности мышления. В лекции, прочитанной при вручении ему премии Тьюринга, Э.Дейкстра рассказал историю одного проведенного им эксперимента. Он разослал нескольким своим корреспондентам условие одной задачи и попросил их прислать ему алгоритм ее решения. Все они прошли мимо самого очевидного и элегантного решения, сказал Дейкстра, поскольку их понятие о повторных вычислениях было слишком тесно связано с представлением об управляющей переменной цикла и ее пошаговом изменении.

Ограничения, накладываемые языками

Уже с давних пор мы тщетно бьемся над созданием современного языка. Приведу обширную цитату из статьи Якоба Броновски “Логика ума” в сборнике “Чувство будущего” *{Bronowski J. A. Sense of the Future: Essays in Natural Philosophy (Cambridge, Mass, and London, England, The MIT Press, 1977). Selected .and edited by Pierro E. Ariotto in collaboration with Rita Bronowski.}*.

Проблема принятия решения ... такой поразительный вопрос сформулировал Давид Гильберт: очевидно ли, что ... все математические утверждения, имеющие смысл, будут рано или поздно отнесены либо к истинным, либо к ложным...

В 1931 г. ... Курт Гёдель доказал две замечательные и встреченные крайне недоброжелательно теоремы. В первой говорилось, что ... любая “е слишком простая логическая система может заключать в себе истинные утверждения, которые тем не менее не могут быть выведены из ее аксиом. Вторая теорема утверждала, что ... про аксиомы такой системы ... нельзя заранее сказать, что они свободны от внутренних противоречий. Коротко это можно сформулировать так — более или менее богатая логическая система никогда не может быть полной, и в то же время нельзя гарантировать ее непротиворечивость...

А.М.Тьюринг в Англии, а Алонсо Чёрч в Америке показали, что нельзя построить механическую процедуру, позволяющую проверить все утверждения логической системы и за конечное число шагов установить их истинность или ложность...

Тарский показал, что нельзя построить точный язык с универсальными свойствами; всякий формальный язык, который по крайней мере так же богат, как арифметика, содержит правильно построенные предложения, про которые нельзя сказать, истинны они или ложны...

Нельзя построить аксиоматическую систему, которая бы производила описание окружающего нас мира в полном соответствии с действительностью, причем каждой точке мира соответствует точка описания; в некоторых точках будут оставаться пробелы, которые невозможно заполнить с помощью дедукции; в других же точках может оказаться возможным прийти к противоположным выводам...

И наконец, теория Тарского продемонстрировала, и я думаю, что окончательно, что универсальное описание природы на едином замкнутом, непротиворечивом языке построить нельзя...

Это основной момент: язык, которым мы пользуемся для описания природы, предписывает (классификацией своих определений и аксиом) как форму, так и ограничения на открываемые нами законы...

Особенностью человеческих языков является то, что они построены на основе старых метафор и аналогий, которые служат

благодатной почвой для изучения двусмысленностей и обнаружения скрытого сходства..

Многие логические проблемы вырастают из этого общего корня, а именно система значений для любого разумно богатого языка непременно включает в себя объекты самого языка. Это создает бесконечный спуск, бесконечный зал с отражающимися друг в друге зеркалами. Этот спуск попадает в фокус всех парадоксов логики, которые в каком-то смысле сродни классическому противоречию, известному еще грекам: они называли его критским парадоксом (или "парадоксом лжеца")...

Бертран Рассел (совместно с Альфредом Нортом Уайтхедом в их общей книге "Principia Mathematica") попытался развязать узел парадоксов этого типа и положить конец нескончаемому потоку утверждений об утверждениях, построив теорию типов. Она предназначалась для того, чтобы удержать нас от использования того же самого языка для обсуждения вещей, которые этот язык обозначает, и для обсуждения фактов самого языка. Человеческий язык богат потому, что мы думаем о самих себе. Мы не можем исключить из человеческих языков ссылки на самого себя и при этом не превратить его из подлинно информационного языка в язык машинных команд...

Всякое размышление о мышлении обязательно включает ссылки на самого себя: первое же положение в философии Декарта, "Cogito, ergo sum", содержит такую ссылку. ...Никакая логическая машина не может разрешить все трудности и парадоксы, создаваемые ссылками на себя...

Машина не является природным объектом, это человеческий артефакт, который подражает нам и пользуется нашим представлением о природе.

"СТРАННЫЕ ПЕТЛИ", о которых пишет Дуглас Хофстедтер в книге "Гёдель, Эшер, Бах: Вечная золотая лента" *{Hofstadter D. "Godel, Escher, Bach: An Eternal Golden Braid" (New York: Basic Books, Inc., 1979).}*, являются расширением этих идей.

А в своем письме-докладе ВМФ США Хоар писал:

- *...а те, кто должен внедрять только что спроектированные языки высокого уровня, сталкиваются с множеством невероятно сложных частных случаев, неоднозначностями и противоречиями. Нужно создавать комитеты по стандартизации ... на это уходит не меньше 6 лет. Такая работа "разбивает души", ведь очевидно, что "правильного" разрешения всех неточностей начального проекта не существует...*
- *От возникновения языка до его "успешной стандартизации" проходит 10 лет...*
- *Это относится к Фортрану, Алголу и PL/1. Паскаль не стандартизован до сих пор.*
- *Почему? Из-за необычных, сложных и неожиданных взаимовлияний одних разделов языка на другие.*
- *Паскаль имеет слишком высокий уровень, чтобы можно было с удобствами проводить (например) двоичный ввод/вывод; этот язык имеет недостаточно высокий уровень для того, чтобы обеспечить полную гарантию безопасности при работе с записями. Включение в язык элементов как высокого, так и низкого уровня является очень сложной проблемой.*
- *Поскольку вред наносят именно эффекты взаимодействия, нужно изучать весь язык, чтобы понять, что следует исключить из него.*
- *Основная заслуга языка высокого уровня в том, что он помогает разуму при проектировании и документировании программ для вычислительных машин; помощь, оказываемая им при непосредственном кодировании, может иметь второстепенное значение {Отнесение помощи при кодировании на второй план несколько симптоматично; может, языки высокого уровня и не так уж ускоряют этот процесс? — Прим. ред.}.*

В недавнем докладе (1978) военно-морскому флоту исследователи насчитали более 2570 различных возможностей или свойств, которые можно выделить в языках программирования. 2570!!

Какими же богатыми оказываются наш мозг и наши языки!!

Кроме сформулированных Броновски естественных сложностей богатых языков, существуют еще и проблемы, связанные с оптимизацией, возникающие при построении и языков, и трансляторов с них. При оптимизации одной из 12 характеристик невольно ущемляются интересы остальных.

Написав транслятор, который проводит трансляцию очень быстро, мы плохо повлияем на скорость выполнения рабочей программы. И наоборот.

Процесс написания программы

Процесс написания программы представляет собой итеративный процесс, состоящий из собственно написания, исполнения и исправления. Исправления потому, что редко какая программа с самого начала не имеет ошибок. В зависимости от задачи и от самого программиста эти три шага могут повторяться несколько, иногда очень много, раз. Программист пишет команды *транслятору,* который в свою очередь переписывает их на язык — более детальный — вычислительной машины. Но транслятор это не человек. Он не понимает полутонов, жестов, тонких намеков, интонаций, выражений лица и еще множества символов, присутствующих в общении человека-переводчика и говорящего или пишущего. *Вычислительная машина плюс транслятор* абсолютно педантичны. Необходимо, чтобы были правильными все запятые, пробелы,, тире, а также их места — все, как определяется языком.

Когда программист представляет свою программу на трансляцию, первое, что делает транслирующая программа,— это проверяет все синтаксические ошибки, а если таковые находятся, исходная программа отвергается; это сопровождается напечатанными сообщениями о том, что в такой-то и такой-то точке программы найдены ошибки такого-то и такого-то типов. Пропущена скобка. Целое число умножается на число с плавающей точкой (в некоторых языках это запрещено).

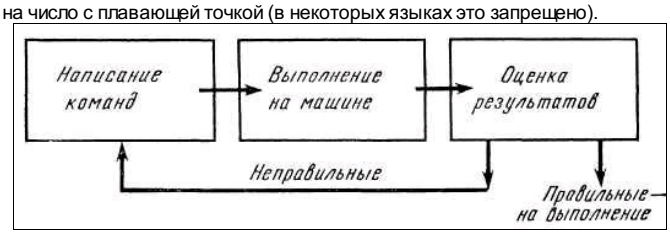


Рис. 5.34. Итеративный процесс отладки программы.

Дж.Вейнберг *{Weinberg G. Psychology of Computer Programming (New York: Van Nostrand Reinhold Company, 1971).}* указывает, что "типичный программист" а своей деятельности по доведению программы до выполнения на машине, проходит два этапа. Сначала он стремится добиться первой безошибочной компиляции. Компиляция — эта трансляция и расширение (добавление нужных программ). "Безошибочность" заключается в отсутствии синтаксических и очевидных логических ошибок, поскольку первый шаг состоит в получении "семантически правильной" программы. Для получения такой программы программист всякий раз при перезапуске программы на машине изменяет в ней несколько операторов.

Через некоторое время предпринимается и второй шаг. Команды на машинном языке загружаются в вычислительную машину *и исполняются* обычно с использованием тех значений данных, которые программист задал наряду с командами. Результаты выводятся на печать и возвращаются программисту. Программист изучает их с целью определить, правильна ли логика работы программы, причем обычно оказывается, что она неправильна. Он исправляет логику процесса и снова перезапускает программу. Когда выясняется, что программа работает правильно, она либо наконец вычисляет то, что нужно, если это отдельная программа, либо передается кому-то другому, кто будет включать ее в более крупную программу. Процесс этот выглядит так, как это изображено на рис. 5.34. Проверка синтаксиса трансляторами — это весьма полезный процесс, помогающий программистам находить не только ошибки в пунктуации, но также и некоторые логические ошибки.

Если, например, программист написал программу, в которой управление передается подпрограмме по имени СЛЕД, но ни одну из особых подпрограмм этим именем не назвал, транслятор отметит этот логический промах.

Кросс-транслятор

Язык высокого уровня типа Фортрана может использоваться в качестве входного языка на самых разных вычислительных машинах, имеющих свои собственные транслирующие (компилирующие) программы. Язык стандартизован, каждая машина имеет свою транслирующую программу и создает рабочую программу именно для себя.

Логическим шагом в использовании этих транслирующих программ является использование машины и некоторого транслятора для перевода с исходного языка *на язык другой машины*.

Это называется кросс-трансляцией. Ее обычно применяют в тех случаях, когда машина, на которой будет выполняться рабочая программа, очень мала (по памяти, устройствам ввода/вывода или возможностям центрального процессора). Пускать трансляторы на

маленьких, медленных машинах не очень эффективно.

Использование кросс-трансляции машины выглядит так. Программа пишется на входном языке; она переводится на большой машине транслятором, который выдает программу на языке второй машины. Затем на этой второй машине выполняется полученная рабочая программа. Вторая, маленькая машина обычно имеет систему команд, совершенно отличную от той, которая имеется на большой машине.

Этот процесс является логическим следствием преимуществ машинной независимости языков высокого уровня. Машинная независимость позволяет нам передавать программу, написанную на стандартном языке, на машину *любого* типа и модели, если только на ней имеется транслятор, который может переводить программу с языка высокого уровня на машинный язык. Схема процесса приведена на рис. 5.35.

Изменения и исправления в программу обычно вносятся на входном языке, с тем чтобы после этого вновь ретранслировать ее кросс-транслятором. На маленькой машине может транслятора вообще не быть.

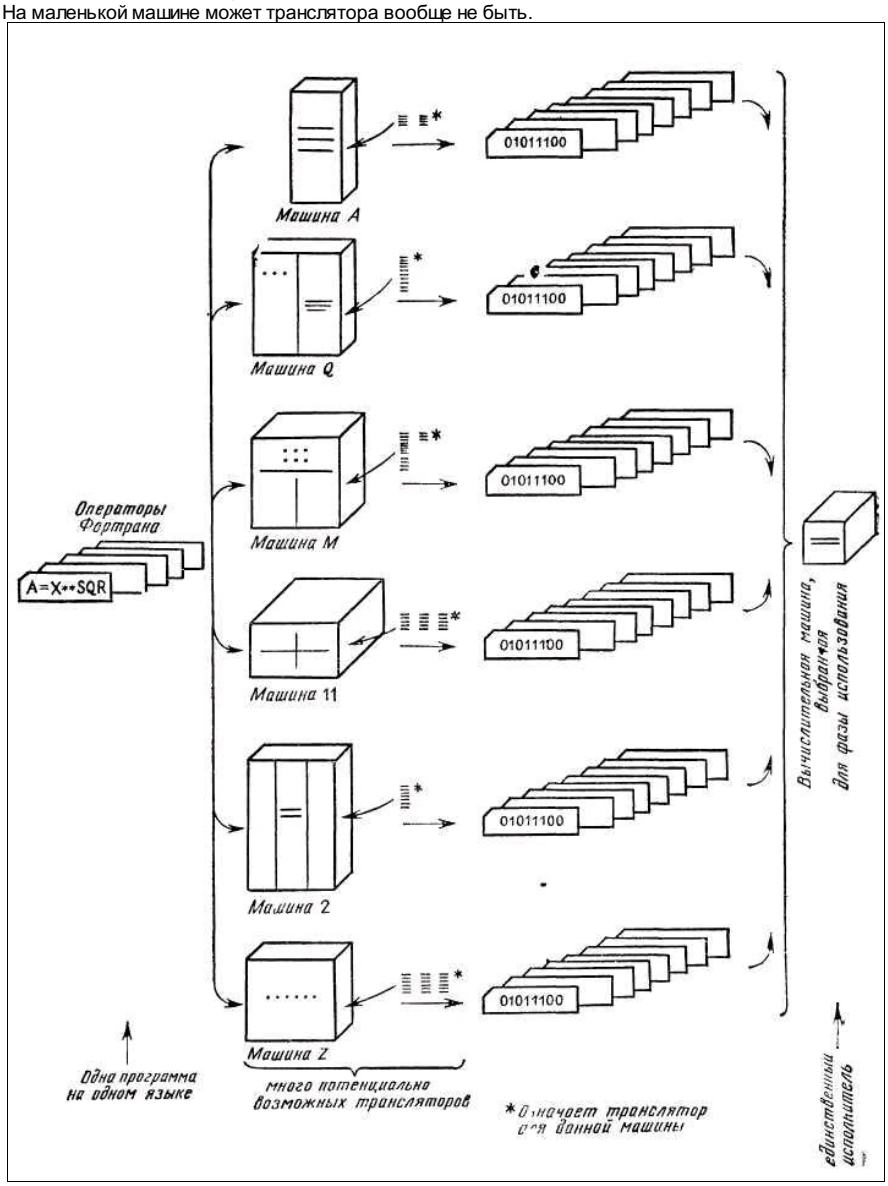


Рис.5.35. Кросс-трансляция.

Этот процесс значительно ускоряет программирование. Очевидно, что его недостатком является необходимость иметь большую машину.

Не так очевидно то, что указанный транслятор должен быть доступен группе, фактически продолжающей разработку. Случается, что разработчики программ при создании программы пользуются частными трансляторами, а затем возвращают их владельцам. В результате группа сопровождения лишается возможности транслировать исправляемые программы. Отвратительная ситуация. На создание "новых" трансляторов были затрачены миллионы!

Множество форм одной программы

Частую путаницу вносит тот факт, что программа одновременно может существовать во многих формах сразу. Причем именно этого мы и хотим. Нам нужна "версия" программы на языке рабочей машины (для исполнения) и версия на входном языке (для модификации). Подробнее мы обсудим это позднее, а сейчас давайте посмотрим, какие формы может принимать программа. Сначала посмотрим на простое деловое письмо.

"Где мое письмо?"

"Ваша рукописная копия лежит на вашем письменном столе; отпечатанная копия находится на столе у начальника; имеются версия, хранимая в системе обработки текстов, и версия, передаваемая по телефонным проводам во Флориду, кроме того, одна копия передается через спутник связи в ФРГ". (См. рис. 5.36.) Некоторые из этих форм писем могут быть прочитаны людьми: некоторые могут читаться только вычислительными машинами.

Такое же бывает и с программами. На рис. 5.37 изображено всего пять различных форм, в которых может существовать программа печати платежной ведомости.

Вычислительные машины для трансляции

Часто забывают о том, что для перевода нам нужен не только транслятор, но и вычислительная машина. Это кажется очевидным, но при моей работе над тремя проектами больших программных систем, в которых именно для трансляции не хватало вычислительной мощности, это привело к потрясающему росту затрат на разработку систем.

При разработке обеспечения для системы "Скайлэб" те вычислительные машины, которые использовались для проведения трансляции и построения системы, были переданы группе сопровождения для окончательного завершения работ по программе "Аполлон". Программисты писали программы, но трансляций было проведено не так уж много, а тестирование было просто невозможно. В конечном итоге был затрачен целый миллион долларов на аренду мощной машины IBM 360 модели 65, которая была необходима единственно только для сопровождения системы.

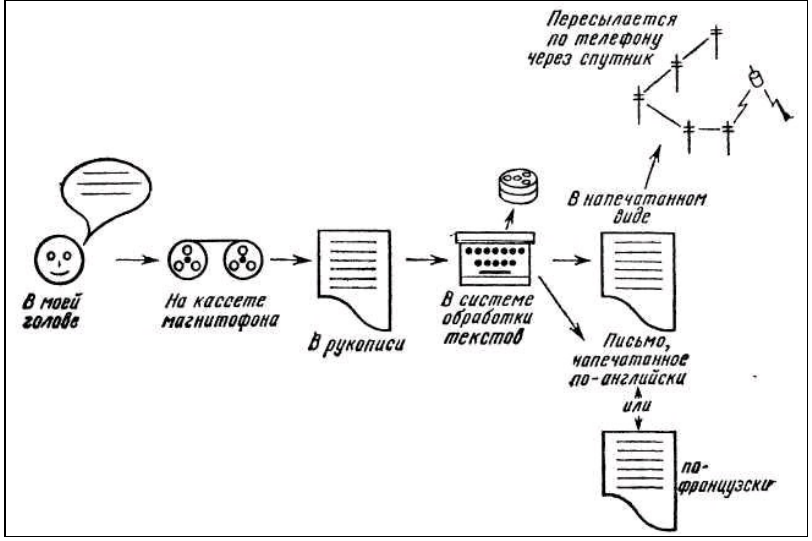


Рис. 5.36. Разные формы существования письма.

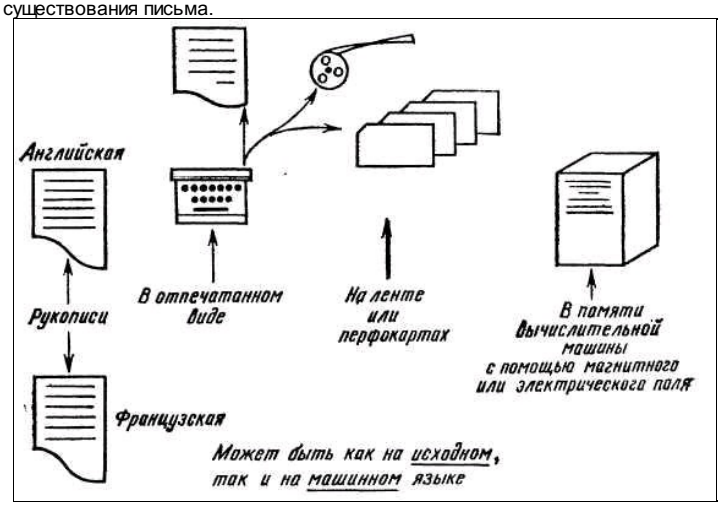


Рис. 5.37. Разные способы существования программы.

Во время работы по заказу военно-морского флота, проводившейся на Западном побережье, мои 40 программистов сделали очень мало, основной причиной чего была недостаточно большая и недостаточно комплектная инструментальная машина IBM 370. Программы писались для выполнения на машине фирмы Sperry UNIVAC UYK20, а трансляция проводилась на IBM 370/138. Дело оказалось не только в том, что эта машина слишком мала (ее мощности явно не хватало), но также и в том, что на ней имелся всего один канал, а, значит, возможности ввода/вывода были сильно ограничены. Нам пришлось переходить на машину модели 148, на которой было больше каналов. Наша группа плохо произвела выбор инструментальной ЭВМ.

Однажды в Европе меня познакомили с разработкой системы, состоявшей из программ в 1 млн. операторов на Коболе. Программы, входившие в систему, были слишком велики, чтобы из них можно было составить хорошую модульную систему. “Почему так?” — спросил я. По той причине, что в вычислительную машину, использовавшуюся для компиляции, было слишком тяжело вводить отдельные задачи. Это привело к тому, что проект обеспечения времени использования был изменен таким образом, чтобы уменьшилось время, необходимое на трансляцию. Трансляция одного большого модуля занимала меньше времени, чем трансляция нескольких модулей меньших размеров, выполнявших все те же функции.

Программирование в диалоговом режиме

Появление возможностей разделения времени обещало целую революцию в программировании. Казалось, время ожидания решения больше не будет самым узким местом, оно уменьшится до нуля.

Но это далеко не так. Логично предположить, что предоставление терминалов всем программистам увеличит их производительность, но доказательств этому пока нет. К тому же терминалы стоят денег. На основе базы данных по производительности, созданной в отделении федеральных систем IBM, “кажется”, можно прийти именно к такому выводу, но люди, изучавшие этот вопрос, не хотят делать такого заключения.

В больших проектах, где заняты сотни программистов, за огромное число терминалов приходится платить очень дорого; эта тенденция катастрофически влияет на всю работу в целом, вся работа может превратиться в халтуру. Когда человек садится за терминал, у него возникает ощущение, что он должен его использовать во что бы то ни стало. Это совершенно противоречит стремлению к разумной, спокойной деятельности. Мы вернемся к этому вопросу, изучая проблемы производительности.

С чем же мы ведем диалог?

Давайте предположим, что я вошел в машинный зал и подошел к вычислительной машине. Никто мне ничего о машине не рассказывал, но я с радостью увидел, что это DEC 11/70, с которой я знаком, поскольку уже работал с ней. Я вижу, что машина включена и готова к использованию.

Возникает вопрос: пуста ли машина, ждет ли она самой первой команды? А есть ли в ней транслятор? Или операционная система?

В зависимости от ответов на эти вопросы я буду выдавать машине совершенно разные команды.

Если в машине ничего нет, я начну вводить команды с помощью тумблеров на внешней панели.

Если в машине есть ассемблер, а я хочу начать программировать, мне надо вводить команды на языке ассемблера.

Если в машине находится транслятор, я должен вводить операторы языка высокого уровня.

Если есть операционная система, мне необходимо знать, как этой операционной системе давать приказы, поскольку всей работой с машиной управляет именно она и мне нужно иметь дело именно с ней.

Никаких внешних признаков того, что там есть, не существует; поэтому если никто не подскажет мне, то мне придется применить метод проб и ошибок. Я попробую один из вариантов, который должен работать, если в машине находится то-то и то-то, потом посмотрю, работает ли этот вариант, и т.д.

В памяти вычислительной машины могут находиться или отсутствовать такие системы программ:

1) операционная система;
2) система управления базой данных;
3) система связи;
4) система управления дисплеями;
5) ассемблер (транслятор);

6) транслятор с ЯВУ;

7) прикладные программы.

Когда человек садится и пишет программу либо вводит ее через терминал, присоединенный к вычислительной машине, он может связываться либо непосредственно с аппаратурой (линия *C* на рис. 5.38), либо с целой цепочкой программ, написанных специально для программистов или пользователей. Человек на линии *A* ведет диалог с шестью программами и аппаратурой. Человек на линии *B* ведет диалог с одной программой и аппаратурой.

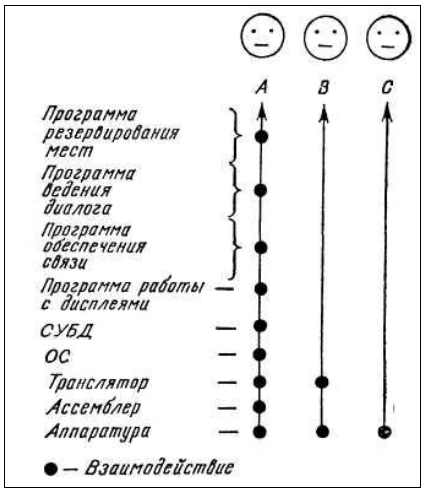


Рис. 5.38. Взаимодействие с программой для вычислительной машины.

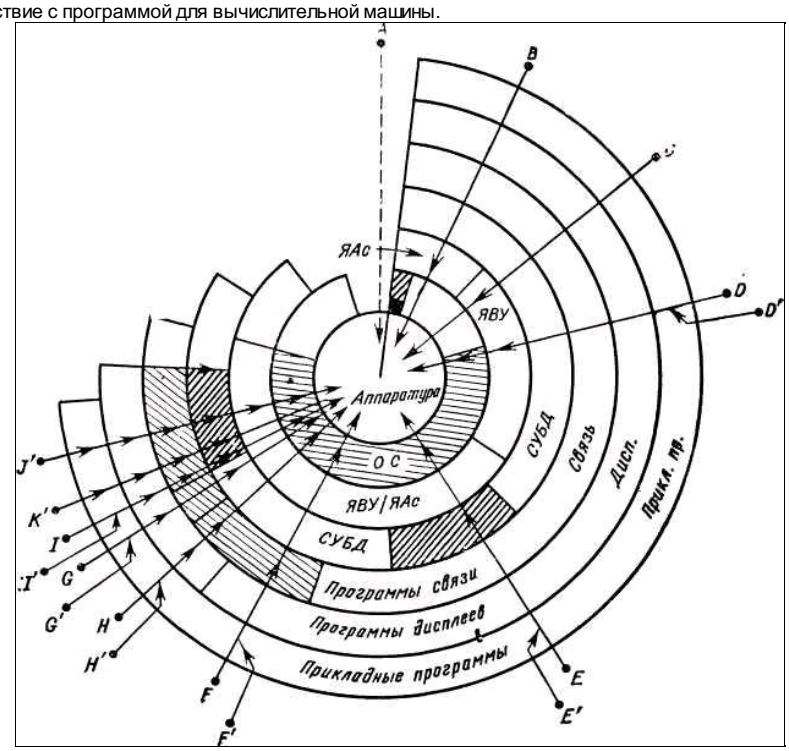


Рис. 5.39. Возможности ведения диалога с вычислительной машиной и программным обеспечением.

Очевидно, что по мере добавления различных программ и установления связи с ними мы получаем огромное число разных возможностей для ведения диалога. На рис. 5.39 изображены только некоторые способы связи программиста или пользователя с аппаратурой!

Если я нахожусь в точке A, я пишу на языке машины.

В точке *B* я использую язык ассемблера.

В точке *C* я использую язык транслятора.

В точке *D* я использую приказы операционной системы.

В точке *E* я использую язык базы данных.

В точке *F* я использую язык системы связи.

Человек, программирующий на уровне E, *не обязан ничего* знать об уровнях, расположенных ближе к машине. Он может сколь угодно долго работать, не обращаясь к более глубоким уровням. Он имеет дело с *абстрактной машиной,* которая состоит из аппаратуры и определенных слоев программного обеспечения.

Все эти люди заняты программированием. Все они пишут команды, которые должны выполняться машиной. Команды проходят через транслятор или даже через целую серию трансляторов, а в результате получается группа машинных команд, которые потом исполняются аппаратурой.

Буквы со штрихами (E', *F'* и т.д.) обозначают не программистов, а *пользователей.* Обратите внимание на то, что их первое вхождение в вычислительную систему (аппаратура плюс программное обеспечение) осуществляется через прикладные программы. *Пользователь* вводит *данные* для прикладных программ, а результатом при этом будет не программа на машинном языке, а ответ ему либо несколько ответов.

Такое разнообразие способов ведения диалога с машиной вместе с ее программным обеспечением может вносить большую путаницу, поскольку все эти способы сильно отличаются друг от друга. Число способов, которыми можно взаимодействовать с вычислительными системами, поистине неограниченно. На нашем рис. 5.39 мы ограничились только одним прикладным уровнем, в то время как их могло бы быть и больше, в этом случае сообщения переходили бы с одного прикладного уровня на другой. Но и с одним показанным на рисунке уровнем число комбинаций доходит до 720 способов ведения диалога с аппаратурой и программным обеспечением с помощью терминалов. Это разнообразие вариантов является *и* благом *и* проклятием для вычислительных машин.

Точка / является хотя и необычной, но не такой уж редкой ситуацией. Используются сразу все слои программного обеспечения. Такой случай возникает, когда мне приходится из одного города вести диалог через диалоговый транслятор **с** вычислительной машиной, находящейся в другом городе. Вариант *K* (без использования уровня ЯВУ) встречается очень часто.

Управление написанием программ

Из-за чрезвычайной гибкости запоминаемых программ программирование стало очень трудным делом; одна и та же задача

может быть запрограммирована сотней разных способов, ответ будет один и тот же. Все варианты будут работать, все дают одинаковые ответы! Может ли один быть лучше другого? Может, причем по-разному. Один быстрее выполняется. Другой требует меньше памяти. Мой вариант, возможно, был запрограммирован быстрее, чем ваш. То, что ваше решение "работает быстрее", компенсируется тем, что вы дольше писали вашу программу.

Число различных последовательностей команд почти беспредельно, ведь в своей работе программист связывает воедино множество шагов, которые он предпринимает, чтобы решить задачу. Кто может сказать что-нибудь о правильности или хотя бы о работоспособности программы до тех пор, пока она не будет опробована? До тех пор, пока программа не будет выполнена на вычислительной машине?

Очень трудно управлять чьей-нибудь работой и быть уверенным, что у него все в порядке, если число вариантов, которые тот может предложить, практически неограниченно.

Управление процессом написания крайне затруднено. Делают программное обеспечение трудным такие его свойства, как абстрактность и неосязаемость, необозримость и изменчивость.

Для помощи при управлении этим процессом предлагается использовать множество новых прекрасных методов.

Структурное программирование не только помогает процессу проектирования, но также делает окончательный вид программ более наглядным. Для отслеживания каждой команды (строки программы) в разрабатываемой программе создаются специальные программы, называемые *автоматическими библиотекарями.* Эти программы тоже сделали практическое программирование более наглядным. Они сделали программирование из частного дела общим достоянием.

Различия уровня квалификации программистов

Изучение программирования "в малом", т.е. небольших программ, показывает просто фантастическое различие в производительности и конечных результатах работы среди программистов. Различие в производительности, выраженной в строках программы в единицу времени, достигает 25 раз. Это означает, что некоторые люди пишут в день в 25 раз больше строк программ, чем другие. Размеры получающихся программ также отличаются, коэффициент доходит до 20; программы одних программистов могут быть в 20 раз больше программ других. Эффективность программ изменяется в диапазоне от 1 до 10; одни программы работают в 10 раз быстрее, чем другие, выполняющие ту же работу.

Однако совершенно не обязательно будет так, что программист, пишущий в 25 раз быстрее другого, будет хорошим программистом. Строки программы — это обманчивое, несовершенное средство измерения качества. Но это все, что у нас есть. Более или менее подробно мы будем изучать эту меру в гл.6. Здесь мы хотели только указать, что среди программистов имеются значительные различия.

Из всего сказанного и из приведенных цифровых данных надо сделать вывод, что необходимо проводить тщательный отбор сотрудников, набирать и выбирать хороших программистов, поручать людям работу, соответствующую их наклонностям.

Если ваше задание достаточно велико, будет крайне глупо оценивать результаты сотен программистов, основываясь на производительности лучшего из них.

Главный программист

Предположим, мы хотим задать вопрос: в какой момент следует разделить программу на две части и поручить программирование двум программистам? На этот вопрос нет полного всеобъемлющего ответа. Только в идеальном случае можно избежать разделения программы на отдельные части. С той минуты, как вы поделили работу и поручили ее выполнение двум работникам, они должны начать контактировать друг с другом, и тут же возникнут излишняя трата времени, путаница, появятся ошибки.

Очень многое зависит от работы, которую надо выполнить, от программиста, которому она поручена, времени, за которое нужно ее сделать, а также от тех средств, которые можно использовать при написании программ. В проекте системы для газеты "*Нью-Йорк таймс" один* программист, Т.Бейкер, за 22 мес. создал работающую программную систему, состоящую из 83 тыс. операторов на языке высокого уровня. Этим успехом мы обязаны тому, что в работе был применен метод Х.Милса под названием "Группа главного программиста". Бейкер работал так, как работает хирург вовремя операции. Ему помогал библиотекарь, выполняющий все канцелярские дела, включая черную работу по отправке программ на вычислительную машину для проведения трансляции. Инструментальные средства, необходимые Бейкеру, поддерживались в нужном порядке кем-то еще. Один помощник проверял программы Бейкера и давал ему советы.

Ясно, что, поскольку работа выполнялась *одним* человеком, производительность сильно возросла.

Метод главного программиста прекрасно проявил себя в проекте для "Нью-Йорк таймс" и еще в нескольких проектах после него. Но у этого метода есть один недостаток. Главный программист может проделать такой объем работы, только если он в шесть — десять раз превышает средний объем. Этот метод тем не менее не подходит для больших проектов, программы которых составляют сотни тысяч строк программного текста.

Если объем программ меньше 100 тыс. строк, то метод главного программиста работает хорошо. Не так уж часто нам приходится работать над такими "небольшими" проектами. Ни разу нам не удалось добиться хорошей работы по методу "группы групп".

Из-за того что метод главного программиста был впервые применен в проекте "Нью-Йорк тайме", многие решили, что он является частью структурного программирования. Это не так!

Некоторые утверждали, что работы для "Нью-Йорк таймс" кончились неудачно. Но это не так. Мы (фирма IBM) сэкономили около 700 тыс. долларов, а система работала. Первое время в редакции "Нью-Йорк таймс" было слишком мало людей, которые имели опыт работы с системой. По мере подключения сотрудников газеты к работе в системе начались обычные проблемы перехода, которые всегда возникают при перекладывании ответственности за систему с группы разработчиков на группу сопровождения. Эти нормальные трудности были кое-кем неправильно истолкованы с целью бросить тень на великолепные результаты, полученные благодаря структурному программированию и использованию метода главного программиста.

Мне до сих пор досаждают вопросами вроде того, сколько же надо платить программистам, а я отвергаю эти вопросы с самого начала. Для оценки качества программ, создаваемых разными людьми в единицу времени, нужно учитывать по крайней мере 27 разных параметров. И прежде всего нужно определить, что представляет собой эта самая строка текста. Об этом мы поговорим в гл.6, посвященной производительности и ее оценкам.

Библиотекарь

Сравнительно новым методом, применяемым при разработке программного обеспечения, является использование библиотекаря. В прошлом программисты тратили очень много времени на то, чтобы перебирать колоды перфокарт, подготавливать распечатки программ, и на другие подобные мелочи. Вся эта канцелярщина теперь поручена библиотекарям.

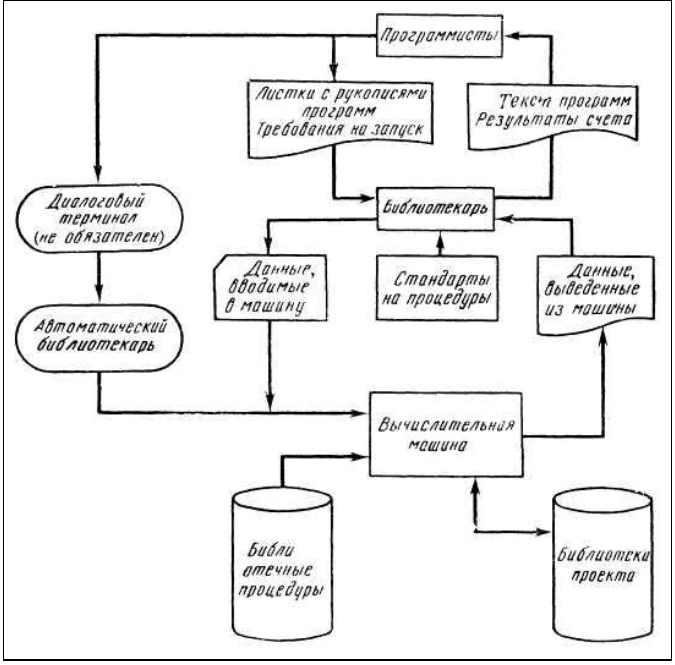


Рис.5.40 Работа над проектом с помощью библиотеки и библиотекарей.

Но дополнительным преимуществом их привлечения оказалось то, что все программы, написанные программистами, теперь бывают собраны в некотором централизованном доступном хранилище.

Почти неуловимое, но весьма важное изменение произошло в процессе разработки программного обеспечения. Программы стали собственностью организации, в то время как раньше они были собственностью программистов вплоть до тех пор, пока работа над ними не бывала закончена. Редко какой программист показывал своим товарищам все свои неудачные попытки заставить программу работать правильно. Если на это ушло девять пусков, кто об этом будет знать? Такая свобода порождала неопределенность в проведении процесса программирования. Она порождала — или па крайней мере сильно облегчала — поспешные решения, сильно ударяла по всей методологии выполнения работ.

Теперь руководитель работ может просмотреть легко читаемые, структурированные программы, которые вводятся в машину, и увидеть, сколько попыток сделал программист Джонс, чтобы получить работающую программу. Из частного дела программирование превращается в общественное.

“Библиотека” не имеет никакого отношения к библиотекарю. Библиотека — это собрание программ, обычно вспомогательных, которые могут найти применение в любое время.

Некоторые неправильно считают использование библиотекаря частью методики структурного программирования. Это разные понятия и методы.

На рис. 5.40 показаны и автоматический библиотекарь, работающий совместно с программами ведения диалога, и люди, которые были назначены библиотекарями для данного проекта.

Компоновка программ

Сборка большого числа модулей в одну работающую систему программного обеспечения

На этой фазе разработчики получают уже написанные программистом или группой программистов и даже до некоторой степени уже оттестированные программы и соединяют их в единое целое, таким образом, что вычислительная машина сможет выполнить сразу десятки или даже сотни программ, от самого начала до самого конца, получив в конце концов желаемый результат или выполнив желаемый процесс.

Я выбрал слово “компоновка” потому, что оно и в окружающем нас мире означает соединение заранее изготовленных частей в целое изделие. Именно этим мы и занимаемся при создании большой системы программного обеспечения.

Шесть фаз, выделенные нами в разработке программ, таковы:

Определение требований
Проектирование
Написание команд, программирование
Компоновка
Тестирование, или верификация
Документирование

Если нам нужна единственная, выполняемая сама по себе программа, фаза компоновки будет *отсутствовать*.

Причина, вынуждающая нас выделить компоновку в качестве отдельной фазы разработки программного обеспечения, состоит в том, что для такого рода деятельности в изобилии имеются новые автоматические средства. В прошлом подобная работа выполнялась вручную и, следовательно, очень медленно. В настоящее время в нашем распоряжении оказалось значительное количество программ, автоматически выполняющих компоновку систем программ из сотен подпрограмм.

Эти средства жестко связаны с выбранным ранее языком программирования, а иногда и с операционной системой, но все же стоят особняком от них.

Например, с языком программирования Ада, который, видимо, будет стандартизован министерством обороны, связан большой набор программ сопровождения, составляющих некоторое “окружение”, которое могут использовать разработчики программного обеспечения для компоновки длинных цепочек программ *{Мягко выражаясь, это утверждение не совсем верно (а точнее, совсем неверно). — Прим. ред.}*. Это окружение носит название APSE (Ada Programming Support Environment — окружение сопровождения программирования на языке Ада), в сам язык встроены специфические конструкции и описатели, позволяющие обращаться к элементам окружения.

Между языком и его окружением в этом случае есть тонкое различие. Это различие весьма существенно. Если мне не надо вставлять программу в большую систему, я могу пользоваться только языком, не обращаясь к его окружению. Мне, следовательно, может понадобиться только транслятор, и я потребую, чтобы этот транслятор мог работать отдельно от всего окружения.

Следует ли проводить компоновку перед тестированием?

В нашем списке тестирование стоит на пятом месте, вслед за компоновкой. Но тестирование отдельных программ проводится часто и перед компоновкой. Под тестированием мы здесь понимаем тестирование системы, которое нужно выполнять только после компоновки.

Средства обслуживания руководства

Операционные системы помогают и программистам, и их руководству, и точно так же автоматизированные средства обеспечения разработки программ имеют многофункциональное назначение. Они также нужны и программистам, и руководителям разработки.

Мы уже видели, как они оказывают помощь при программировании. Теперь посмотрим на то, как они могут помочь руководству.

Когда под нашим руководством над созданием большого числа программ работают большие группы программистов, нам необходимо принимать во внимание несколько моментов:

1) взаимодействие;
2) распределение данных;

3) объединение;
4) выполнение собранных “кусков”;
5) отслеживание номеров версий и модулей.

Взаимодействие. Слово “заменимый” обозначает всем известное явление. Если я покупаю скобы к сшивателю бумаг, то, естественно, ожидаю, что смогу ими пользоваться. Это удается всегда. Удается потому, что их размеры, конструкция, форма, материал, вес и упаковка удовлетворяют некоторым стандартам. Такие скобы производят многие фирмы. Полная взаимозаменяемость скоб констатируется словами: “Скобы заменяемы”. Взаимозаменяемы.

Этот же принцип имеет значение и в области программного обеспечения. Нам нужно установить образ действий, форму, синтаксические правила, словарь и необходимое содержание программы, которая будет работать совместно с другой программой. Если начать разговор о взаимодействии **с** определения всего этого, то программы можно будет объединить друг с другом со значительно меньшими затратами.

Распределение данных. Разные программы часто работают с одними и теми же данными. Все данные мы можем разделить на две очень разные группы — общие данные и данные приватные. Приватные данные принадлежат только одному модулю. Общие данные передаются в другие модули. Работа с данными по тому или иному способу имеет и свои плюсы, и свои минусы. Приватные данные делают модули более независимыми друг от друга, но такие программы труднее разрабатывать. Программы с общими данными разрабатывать легче, но такая методика может привести к тому, что один модуль повлияет на другой или уничтожит результаты его работы.

Автоматическая компоновка — объединение. Мы проектируем модули таким образом, чтобы их объединение проходило как можно легче и понятнее. Это дает нам возможность заменять ручной труд трудом автоматизированным с использованием программ, которые будут объединять модули автоматически. Раз наши правила ясны, объединение будет несложным делом. Процесс объединения имеет множество наименований:

Редактирование связей
Автоматическое построение
Интеграция
Объединение
Связывание

Между этими процессами есть некоторая разница, но все они очень похожи.

Выполнение собранных частей. Здесь мы снова сталкиваемся с мощными новыми средствами. В прошлом на многих существующих в настоящее время вычислительных машинах группы программного обеспечения вручную управляли системой программирования. Если у нас сотни модулей, такая работа отнимает больше всего времени.

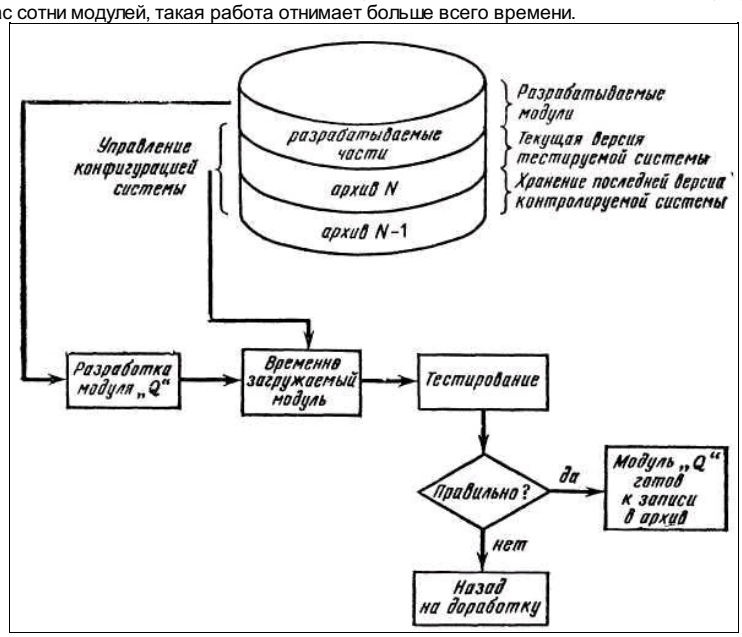


Рис.5.41. Автоматическое управление версиями системы.

К счастью, существует и продается множество программных пакетов, которые могут выполнять подобную работу автоматически. Они помогают пропустить на машине систему, которая получилась после выполнения процесса объединения.

Контроль версий системы. Для сборки, составления каталогов и хранения всех частей программной системы нами используются и вычислительная машина, и программы. Все модули имеют имена. Мы отдельно храним проверенные и готовые к работе системы, отдельно системы, проходящие тестирование, модули, прошедшие тестирование, и модули, не подвергавшиеся трансляции. Для автоматизации такого хранения, облегчения, ускорения и уменьшения вероятности возникновения ошибок мы опять прибегаем к помощи вычислительной машины. На рис. 5.41 изображен диск, на котором организовано хранение нескольких версий системы.

Автоматическое использование инструментальных средств

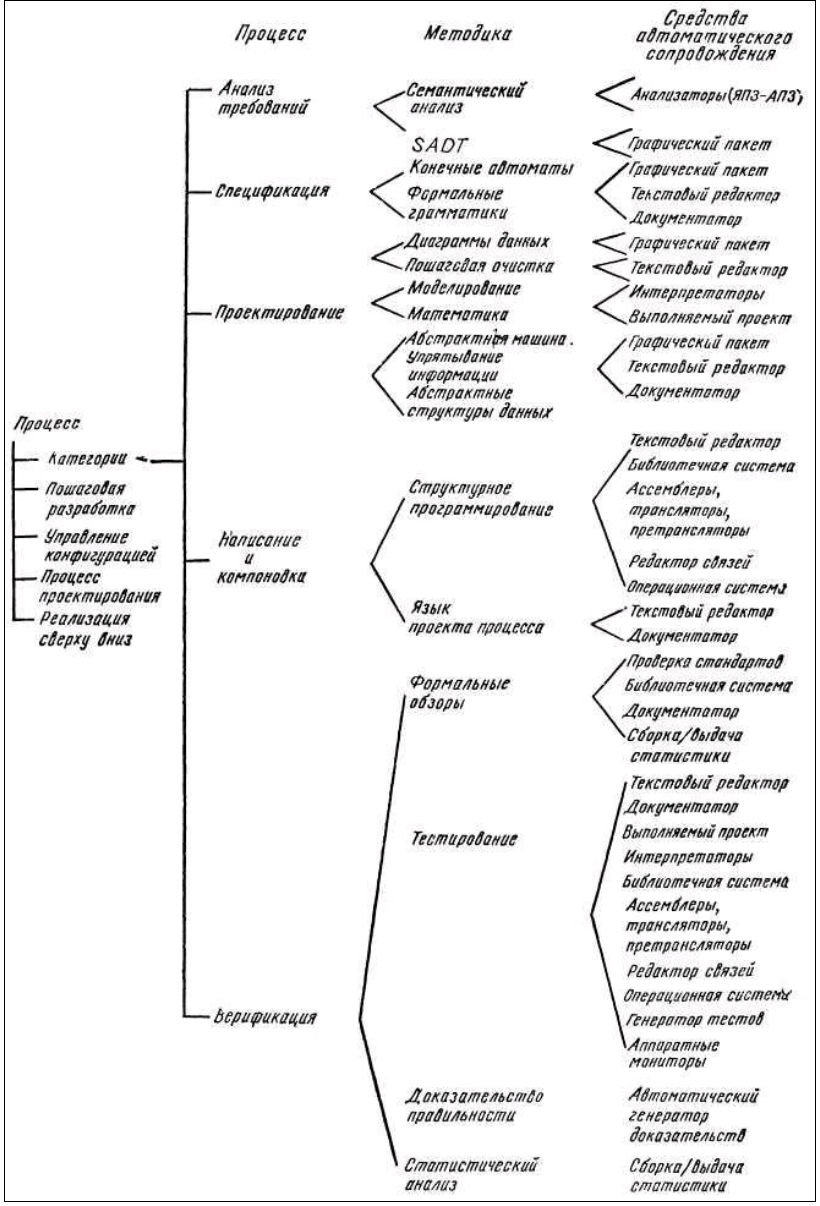
Собрание инструментальных средств это одно. Взаимосвязанный, автоматизированный набор инструментальных средств это совершенно другое. Еще в очень многих современных разработках программисты продолжают применять инструментальные средства вручную. Они вставляют в свою программу команды для вызова вспомогательных программ или физически вкладывают свои программы в тестирующие колоды и передают их на выполнение.

Сегодня программист уже имеет возможность оформлять обращение к этим инструментам отладки как вызовы процедур, а программное окружение, работающее под управлением операционной системы, выполнит задание, которое ранее приходилось выполнять вручную.

Недостаточно хорошие инструментальные средства значительно замедляют процесс разработки. Чем новее вычислительная машина, тем более непривычные средства предлагаются на ней, тем меньше средств предлагается вообще.

Выбор инструментальных средств должен следовать за выбором методики. Если сначала вы выберете язык, то это может помешать вам применять методы структурного программирования. Некоторые языки не отвечают методам структурного программирования.

Некоторые языки не дают вам возможности контролировать получающиеся модули. Некоторые лишают возможности упрятывания информации. Переходить к выбору инструментов нужно после выбора методик. А методика должна выбираться после того, как произведен выбор процесса. (См. рис. 5.42.)



Рис, 5.42. Процесс, методика, инструментальные средства.

Выбор трансляторов и языков

В основании выбора транслятора лежит выбор двух типов проводимой нами оптимизации, мы оптимизируем:

1) либо использование транслятора (время трансляции),

2) либо программу, получающуюся в результате трансляции (время использования). Техническая программа, которая будет выполнена только один раз, совершенно не нуждается в том, чтобы оптимизировать параметры ее использования.

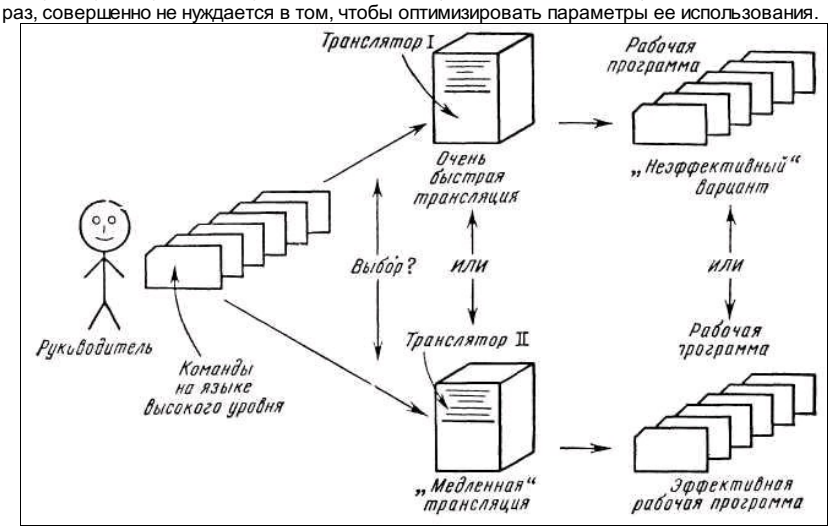


Рис 5.43. Что оптимизировать при трансляции — использование или разработку?

Пользователь, который транслирует свою программу один раз, а затем ежедневно выполняет полученную в результате рабочую программу, совершенно не интересуется тем насколько быстро проходит эта трансляция; этот пользователь хочет иметь “эффективную рабочую программу”. (См. рис. 5.43.)

Выбор языка. С такими же компромиссами приходится иметь дело и при выборе языка программирования. Какой язык вы будете использовать, зависит от задания, которое вам поручено, контингента ваших сотрудников и сроков, которые вам отведены. За выбором языка следует выбор транслятора, и язык должно выбирать руководство, а не программист (см. рис. 5.44).

Язык I (на рис. 5.44) оптимален в фазе разработки; язык II — в фазе сопровождения, но за счет времени первичной разработки.

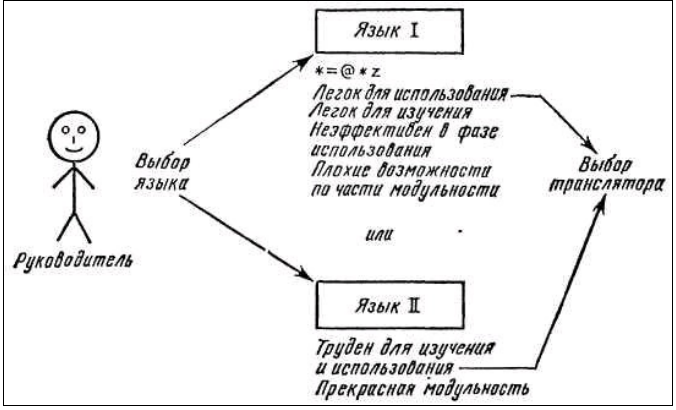


Рис. 5.44. Выбор языка программирования

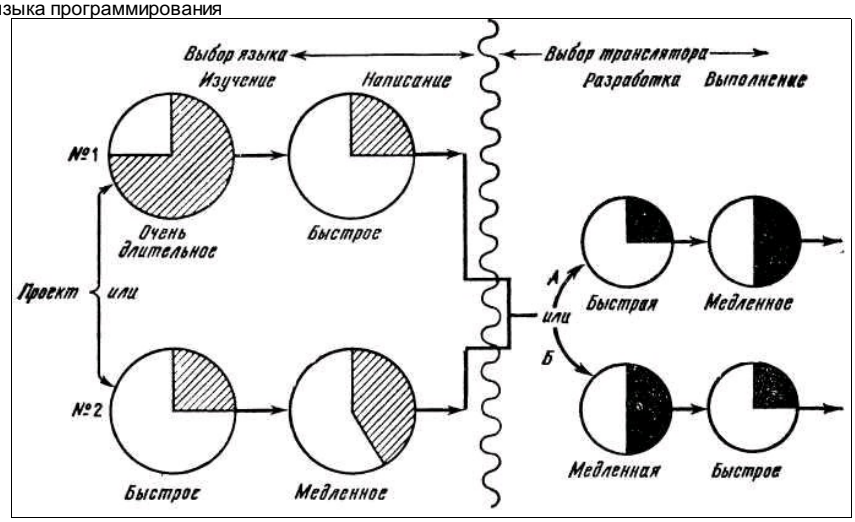


Рис. 5.45. Выбор языка программирования и последующий выбор транслятора.

Заметьте, что выбор языка не связан с выбором транслятора. А теперь посмотрите на схему последовательного выбора языка и транслирующей программы (рис. 5.45).

Реализация сверху вниз

Мы обращаемся к методу сверху вниз именно сейчас, а не при обсуждении процесса проектирования. Реализация сверху вниз смысл имеет, а вот проектирование сверху вниз совершенно бессмысленно.

В разделе, посвященном проектированию, мы увидели, что первой частью этого процесса является этап создания, творчества. По своей природе этот процесс связан с переходами взад и вперед. У этого процесса может быть верхний уровень, а может и не быть.

Но если мы можем найти этот верхний уровень в фазе проектирования, то реализовать программное обеспечение мы сможем, двигаясь сверху вниз. Мы можем создать программы верхнего уровня и, “отталкиваясь” от них, добавлять подчиненные программы, продвигаясь вглубь, уровень за уровнем, создавая нужное нам программное обеспечение.

Такой подход заметно отличается от метода реализации снизу вверх, когда первыми создаются программы самого нижнего уровня. Можно ли говорить в этом случае о проектировании сверху вниз? Многие считают, что это не проектирование, а реализация сверху вниз. Во многих сложных системах во многих случаях нужно проводить проектирование снизу вверх, поскольку система часто зависит от решений, принятых при проектировании одной-двух подсистем.

Окружение разработки программного обеспечения

При создании новой группы программ существенную помощь программистам и руководителям процесса разработки оказывают вычислительные машины и множество различных программ. В систему эффективно проводящейся разработки входят:

1) мощная вычислительная машина, на которой проходит разработка;
2) мощный и стабильный комплект программ, помогающий при программировании;
3) хорошая группа людей, составляющая и сопровождающая программы;
4) мощный набор программ, помогающих управлять разработкой.

Раньше для разработки нужны были помещения, люди, вычислительные машины и программы. Теперь разработка стала как никогда ранее автоматизированной и управляемой, и все это благодаря системе разработки — множеству программ, называемому окружением. Такое окружение — как написано применительно к языку Ада министерством обороны — состоит из: правил взаимодействия (интерфейса) с пользователем, базы данных и множества инструментальных средств.

Интерфейс с пользователем должен позволять разработчику программ управлять любыми инструментальными программами, а также вводить и выводить из системы информацию.

Базы данных могут быть разными — от простых справочников для пользователей до поддерживаемых с помощью вычислительной техники библиотечных систем. Инструментальные средства могут также быть разными — от листа бумаги и карандаша до автоматизированной системы тестирования, формирующей проблемно-зависимые тестовые данные.

Пользователь должен иметь возможность извлекать из базы данных сведения:

о доступных инструментальных средствах, о способах работы,

о доступных прикладных библиотеках, о том, чем различаются между собой версии инструментальных средств и программ.

Когда транслятор обнаруживает синтаксическую или логическую ошибку, он печатает сообщение об этом. Сообщение об ошибке, выдаваемое для разработчика программы, должно:

выдаваться развернутым текстом, а не каким-нибудь кодом,

указывать неправильную лексему, объект, действие пользователя,

указывать контекст, в котором обнаружена ошибка. База данных должна удовлетворять современным стандартам на легкость использования и содержать:

информацию о самой себе, своей структуре и связях, все версии разрабатываемых программ, учебный материал по языку программирования и инструментальным средствам. В системе должны быть собраны сведения: об ошибках: их частоте и типах, о простоях, о денежных затратах, о графиках работ, о результатах тестирования,

об отчетах по использованию машин и программ. В систему должны быть включены модули (программы), работающие совместно с выполняемыми программами и похожие на модули обычной операционной системы. Примером могут служить программы стандартного ввода/вывода. Инструменты, используемые при проектировании, это: справочники, стандарты разработки, руководства по разработке (“рецепты”),

анализ конкретных ситуаций проектирования с помощью вычислительных машин как в пакетном режиме, так и в диалоге. Инструментальные средства должны:

помогать при интегрированном проектировании аппаратного и программного обеспечений, поддерживать структурную

декомпозицию, включать графические средства,

быть интерактивными (работать в диалоговом режиме), по возможности использовать базу данных, включать методы моделирования и тестирования. Интерфейсы должны обладать следующими свойствами: быть стандартизованными, использовать соглашения о связях и загрузке, их трансляция должна обеспечивать взаимодействие с модулями непосредственной поддержки, в них должны учитываться средства тестирования и отладки.

Любой начальный набор инструментов и библиотек программ должен включать: Транслятор.

Пакет программ непосредственной поддержки, в том числе пакет программ ввода/вывода. Пакет программ работы с файлами.

Библиотечную систему для работы с модулями, написанными на языке Ада. Редактор связей/загрузчик.

Набор средств отладки, работающих на символьном уровне *{Т.е. имеющий дело с именами ("символами") объектов в программе, а не с их машинными эквивалентами. — Прим. ред.}*. Оптимизатор.

Вспомогательные средства для составления толковых словарей, указателей и перекрестных ссылок. Редактор текстов, ориентированный на входной язык. Анализатор структур программ. Средства преобразования.

В фирме IBM есть некоторый документ, описывающий подмножество такого окружения в несколько других терминах, но настолько похоже по существу, что о нем тоже стоит рассказать. В IBM это называется *библиотеками поддержки разработок* (Development Support Libraries).

Любая библиотека поддержки разработок представляет собой множество программ и процедур, которые обеспечивают

1) постоянное отслеживание состояния дел, связанных с разработкой продукции;

2) руководство возможностью управления ходом работ по разработке этой продукции.

Библиотека поддержки разработки состоит из четырех элементов:

Внутренней библиотеки в виде, воспринимаемом машиной *{Не обязательно электронной вычислительной. — Прим. ред.}*.

Внешней библиотеки в виде, воспринимаемом человеком.

Механизированных процедур и программ.

Канцелярских процедур.

Механизированная библиотека содержит все текущие сведения о программной части проекта, включая программы на входном языке, перемещаемые модули, операторы редактора связей, рабочие программы, операторы языка управления заданием, тестовые данные и связанную со всем этим информацию. Вся внутренняя библиотека печатается и хранится в удобном для человека виде во внешней библиотеке. Для ссылок хранится вся история предыдущих версий. Машинные процедуры состоят из стандартизованных управляющих операторов, которые выполняют следующие действия:

Редактирование библиотек.

Поиск модулей для трансляции и запоминания результатов.

Редактирование связей задач и организацию тестовых пусков.

Восстановление библиотек и возврат к предыдущим версиям.

Выдачу распечаток состояния библиотеки. Канцелярские процедуры используются библиотекарями для:

регистрации изменяющихся состояний текстов внешней библиотеки.

Регистрации и замены листов в архивах. Программист работает только с внешней библиотекой.

Стыковка программ. Если наш процесс продолжается, переходя от человека к человеку, от машины к машине, от программы к программе, нам необходимо сопровождать его сведениями двух следующих типов.

1. Данные, или сырье, "полуфабрикаты". "Это материал, с которым нужно произвести известные вам действия. Приступайте к работе". *(Данные.)*

2. Информация о процессе. "Все надо делать вдвое быстрее". *(Управляющая информация.)*

Некоторые из таких пояснений могут достигать весьма значительных объемов. Пояснение данных выглядит просто, но оно может разрастаться, становясь тяжеловесным и слишком подробным. Управляющую информацию, возможно, и не удастся передать, придется просто *начать* с нее работу по следующему процессу, но в таком случае будет потеряна гибкость системы.

Есть и еще один весьма важный вопрос, который редко бывает поставлен с достаточной четкостью. "Когда нужно включать управляющую информацию?" При *стыковке* двух программ это может произойти на четырех разных этапах.

1. Во время трансляции. Если управляющая информация вводится на этом этапе, рабочая программа становится не соответствующей действительности. Для ее исправления надо провести перетрансляцию.

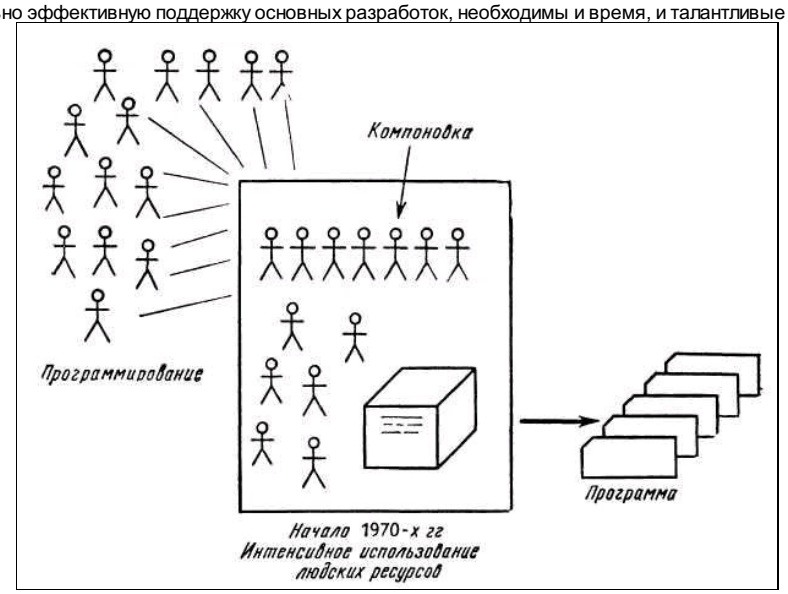
2. Во время редактирования связей. Эта часть этапа компоновки разрабатываемых программ влияет сразу на несколько раздельно транслированных программ. При этом достигается большая гибкость, поскольку пере трансляцию проводить не нужно, достаточно провести повторное редактирование связей.

3. Во время выполнения. Такое управление должно осуществляться непосредственно в вычислительной машине всякий раз, когда программы передаются на выполнение. Чтобы таким образом состыковать программы, нужно затрачивать и время процессора, и память, но гибкость системы значительно возрастает. Мы можем менять, модифицировать, приспосабливать программы во время каждого их выполнения. Для этого не требуется ни перетрансляции, ни повторного проведения редактирования связей. Такой метод позволяет достичь наибольшей гибкости.

4. Во время написания. Пишется программа, которая *всего лишь* транслирует другую программу, больших размеров. Это наименее гибкий метод; для внесения исправлений нужно переписывать программу.

Выводы

Все это говорилось вовсе не для того, чтобы навязать кому-нибудь систему, которую ему следует применять во время разработки. Просто хотелось подчеркнуть, что в настоящее время можно проводить разработку программного обеспечения, используя хорошо обеспеченные всем необходимым, обладающие хорошими эксплуатационными качествами системы разработки. Системы эти достаточно сложны, случайно возникнуть они не могут. Для того чтобы спроектировать и реализовать на практике систему, которая будет осуществлять действительно эффективную поддержку основных разработок, необходимы и время, и талантливые работники.



Разбитие методов компоновки

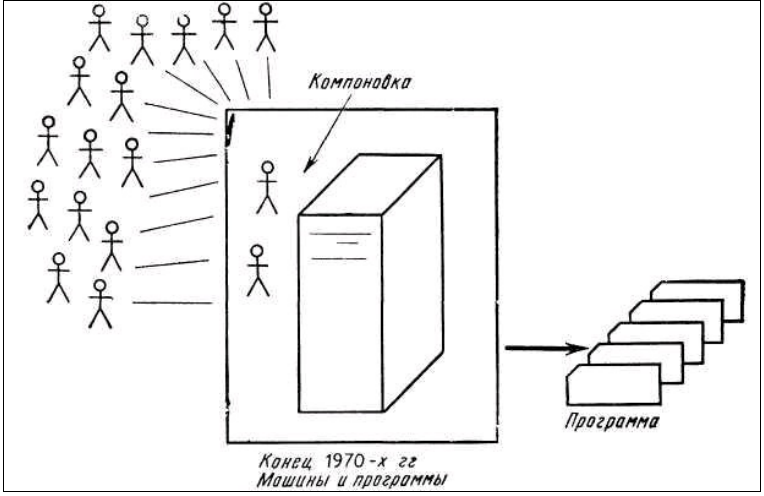


Рис. 5.46. Рост средств автоматизации разработки программного обеспечения.

Строительство моста является результатом работы множества коллективов. Без организационной деятельности по управлению этими коллективами большой мост построить нельзя. Управление распределяет фонды, планирует, проектирует, нанимает, платит, строит графики работ, выдает директивы. Непосредственное руководство трудом рядовых сотрудников является частью этой многогранной деятельности.

Аналогичное положение имеет место и при разработке программного обеспечения, особенно при разработке крупномасштабного обеспечения. Руководство должно осуществлять все те же самые функции; исполнителями большей частью являются программисты, те, которые пишут команды. Перед руководством стоит двойная задача: ему надо не только воздействовать на увеличение отдачи от программистов, но также объединять результаты их труда в единое целое. Эта вторая часть сильно зависит от результатов проектирования, за которым она следует, и от методов, которые применяются при соединении частей системы. Именно это объединение составных частей мы и называем компоновкой.

На рис. 5.46 представлена роль автоматических средств обеспечения разработок, сильно возросшая в 1970-е гг., а также роль аналогичных работ, проводимых вручную, которая все это время неудержимо ослабевала. Этот процесс происходит только при хорошей организации работ по созданию программного обеспечения.

Верификация и тестирование

Верификация

Тестирование позволяет нам определить состояние того, что уже сделано, и показывает, на какое место надо обратить внимание и куда внести исправления. Верификация, прежде всего, призвана помочь нам избежать ошибок. Ошибки могут быть самыми разными — от неправильно закодированных операторов, которые могут приводить к остановке всей системы, и до чего-нибудь совершенно тривиального вроде ошибок при разбиении печатного текста на столбцы, из-за которых распечатка выглядит не так красиво, а ничто больше не затрагивается. Ошибки могут возникать из-за неправильной формулировки требований, неправильно проведенного проектирования, неправильного кодирования. В программах могут быть пропуски, или же туда может попасть лишнее. Причиной ошибок могут быть недоразумения, неоднозначные толкования, недоговоренности или путаница.

Верификация — это новый термин, он стал применяться около 5 лет назад. Обозначает он процесс проверки требований и проекта после того, как они документированы, а также разных частей программы по мере их появления. До того как возникла идея проводить верификацию, весь процесс выяснения того, отвечает ли программа сформулированным требованиям, откладывался до самого конца, до тестирования системы. Верификация передвинула процесс проверки ближе к началу, и ошибки и оплошности стали вылавливаться раньше, что дало возможность программистам избежать реализации неправильно сформулированных требований. Тестирование же обычно заключается в выполнении уже завершенных программ, чтобы убедиться, что они отвечают поставленным требованиям. Верификация выполняется с помощью инспекторских проверок или сквозного контроля. Коллеги, руководители, посторонние программисты, пользователи, если это возможно, собираются и просматривают результаты работы группы по определению требований (либо группы проектировщиков), пытаясь обнаружить в них пробелы, ошибки, неоднозначные высказывания и т.д. С каждым годом проведение верификации облегчается, поскольку методы формулирования требований и проектных решений становятся более четкими и легкими.

Раннее выявление “пробелов”

Сама идея тестирования предполагает некий “конечный продукт”, который теперь надо протестировать и убедиться, что он работоспособен. Мы можем проверить, как работает автомобиль только после того, как этот автомобиль изготовлен.

Но очень хорошо известно, что если можно проверить отдельные части автомобиля еще до сборки, то машина более высокого качества будет получена с меньшими затратами. Своевременное выявление ошибок избавляет от них конечную продукцию или программу. Как же мы проверяем не готовую еще, не материальную продукцию? С помощью нового метода, который называется верификацией. Верификация устанавливает точность и правильность промежуточных результатов, документов, различных этапов процесса разработки. Сквозной контроль, который мы будем изучать позднее, тоже относится к процессу верификации. Частью верификации является и тестирование.

Тестирование есть деятельность, целью которой является установление того, что завершенная программа делает то, что ей надо делать. Лучшим тестированием любой системы является ее эксплуатация. В особенности это относится к системам реального времени. Однако слишком дорого обходятся ошибки и промахи, найденные слишком поздно. Мы желаем находить ошибки как можно раньше.

Вот в чем заключается деятельность группы тестирования:

понимании функций, которые должны выполняться системой,

обнаружении узких мест системы — временных ограничений и неустойчивости, вызванной входными параметрами, проектировании серии упражнений, приводящих систему в действие,

определении в тестирующей системе как аппаратных, так и программных компонент, построении тестирующей системы,

планировании фактического проведения тестирования и дирижирования им.

Для всего этого требуется и изобретательность, и широкие познания.

Тестирование есть сопоставление результатов разработки с процессом с целью выяснения, действительно ли разработанный процесс выполняет то, что предполагалось. Хотя речь о тестировании и верификации зашла в этой книге только сейчас, *после* разговора о компоновке, это связано лишь с необходимостью вести последовательное изложение. На практике правильнее будет провести большую часть действий по тестированию и верификации еще *до* компоновки. Мы включаем в систему тестирования все части программного обеспечения, по мере того как отдельные программисты и целые группы, работающие над нижними уровнями, объявляют об окончании своей деятельности.

Полезно будет еще раз взглянуть на рис. 5.7, иллюстрирующий процесс разработки. Заметим, что верификация и тестирование появляются уже на самых ранних стадиях проекта.

Часть усилий при проектировании следует направлять на то, чтобы сделать программное обеспечение легко тестируемым.

Инспектирование

Принятие методов структурного программирования привело к тому, что программы стали читаемыми. Сложившееся положение являет собой разительный контраст с прежними временами, когда неструктурированные программы больше напоминали бетонные монолиты или мешанину из спагетти.

Структурированные программы делятся на абзацы, при их написании следуют строгим правилам, в них включаются специальные примечания. Если к тому же размеры модулей невелики, не больше 30—50 операторов на модуль, мы получаем очень хорошо читаемые

программы. Прекрасным способом проведения проверок на протяжении всего процесса разработки является инспектирование программ, называемое сквозным контролем.

Несколько коллег, а возможно, и руководителей, знакомых с проектом и применяемой терминологией, садятся рядом с автором программы, который показывает им все, что он написал, и описывает все, что он сделал: алгоритм своей программы, оператор за оператором, результаты всех тестов и т.д. Это новый метод; он стал возможен благодаря применению структурного программирования.

Если подобные инспекции проводятся регулярно, то в течение всего *запланированного времени* можно избежать любых, самых неприятных сюрпризов. Руководство становится более наглядным и простым.

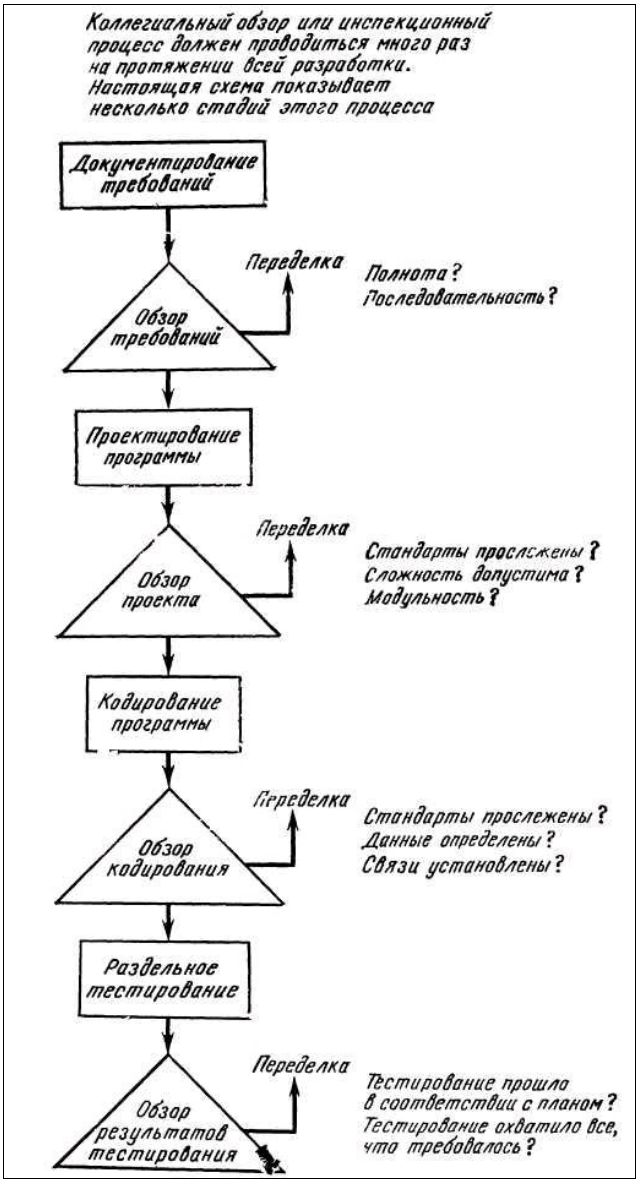
Преимущества этого метода велики. Свежий взгляд позволяет избежать положения, когда "из-за деревьев не видно леса". Появились новые подходы и предостережения, позволяющие избегать ошибок.

Важным фактором становится "рабочая гордость", так как теперь видны не только результаты работы программы, но и каждая команда программы. Многие программисты сообщают, что там, где раньше они позволяли себе проявлять беспечность и небрежность при программировании (хотя никто из них этого раньше не замечал), теперь изо всех сил стараются делать все наилучшим образом. Повышенная аккуратность в работе приводит к повышению качества программ.

Член группы программистов учится у других членов своей группы, изучая их работу; он перенимает методы программирования, возможности языка, тонкости прикладной тематики. Постепенно пробелов в знаниях не остается. Инспекция программ должна стать неотъемлемой частью каждого более или менее крупного программного проекта. Подобные обзоры надо проводить не только для уже написанных программ, их следует регулярно осуществлять на всех стадиях процесса разработки. (См. рис. 5.47.)

Если разрабатывается особенно большая программная система и в производстве находятся сразу несколько модулей, следует одновременно проводить несколько сквозных контролей.

Если бы мы проводили сквозной контроль раньше, нам удалось бы избежать ошибки, которую мы допустили в одном космическом проекте.



Рис, 5.47. Коллегиальные обзоры — сквозной контроль,

Пример ошибки

Когда в 1965 г. приводнился корабль "Джемини V", мир затаил дыхание, ведь произошло это в *100 милях* от заранее рассчитанной точки приводнения. Телевизионные комментаторы заставили весь мир поверить, что капсула может утонуть. К капсуле устремились авианосцы; через час после приводнения капсула была обнаружена вертолетами.

Из-за чего же могла возникнуть такая ошибка? Радиолокаторы, телеметрия, вычислительные машины работали отлично. Программы работали правильно и точно. Все дело оказалось в константе, необходимой для вычисления точки приводнения. Так как Земля вращается не только вокруг собственной оси, но еще и вокруг Солнца, то в одном из уравнений нужно подставлять полное время прохождения по орбите вокруг Солнца.

Спецификации для вычисления были правильные, однако программист решил пойти более короткой дорогой. Для вычислений следовало определить количество дней, часов, минут и секунд, прошедших со времени запуска, и, принимая Солнце в качестве неподвижного центра, рассчитать положение Земли в этой инерциальной системе. И все будет прекрасно. Программист выбрал более короткое решение и просто исключил из рассмотрения число дней полета (он разделил число часов на 24), "логично" рассудив, что по отношению к Солнцу Земля через каждые 24 ч оказывается в одинаковом положении. Это неправильно. Это было бы правильно, если Земля вращалась бы только вокруг собственной оси, а ведь она вращается и вокруг Солнца. Орбитальное движение вносит небольшие изменения в период возвращения Земли в ту же точку по отношению к Солнцу. Этот период не точно равен 24 ч. Этот "просчет" и привел к стомильной ошибке в определении места приземления.

Чья же это была ошибка? Конечно, программиста — он отклонился от проекта. Короткий путь оказался "заминированным". Но в самой программе ошибки не было; она была в формулировке решения. Сквозной контроль мог и, наверное, должен был привести к

обнаружению ошибки. По крайней мере было бы обнаружено, что реализация не совсем точно следовала за проектом.

Тестирование и качество

Тестированию подвергаются все новые разработки. В Детройте имеются испытательные полигоны, на которых можно либо испытать новые автомобили, либо обнаружить скрытые пороки в его конструкции или проекте. Тестирование позволяет точно указать место ошибки, но повысить качество тестируемой системы оно не может. Качество должно повышаться процессом разработки и руководством.

Термин качество" означает разные вещи для разных людей. Я понимаю под высококачественным обеспечением такое обеспечение, которое построено так, что в окончательной продукции находятся характеристики и фазы использования и фазы продолжающейся разработки. Напомним эти характеристики.

Заставляет машину выполнить действие	Функция
Занимает память машины	Размер
Тратит ресурсы центрального процессора	Эффективность
Легкость использования	Практичность
Легкость восстановления	Восстанавливаемость/ Устойчивость
Содержит ошибки	Правильность
Модифицируема	Архитектура
Существует по крайней мере в одной форме, а должна быть в двух	Документация

Тестирование помогает найти ошибку, но не избежать ее. Только хороший процесс разработки в сочетании с различными действиями по верификации и хорошо выполненный проект могут с самого начала воспрепятствовать возникновению ошибок. Система, состоящая из "заплаток", возникших при исправлении ошибок, редко оказывается понятнее системы, которая с самого начала не имела ошибок.

Если разрабатывается крупное программное обеспечение, мы сталкиваемся с проблемой многофункционального тестирования. Иначе говоря, если программы могут выполнять большое число функций да еще в большом числе разных последовательностей, мы должны попытаться оттестировать те из них, которые нам кажутся наиболее вероятными.

Важно, чтобы тестирование проводилось в широких масштабах, было интенсивным, ведь в большой, сложной программе может быть множество скрытых проблем и поворотов. Абстрактный характер программного обеспечения значительно затрудняет возможности увидеть эти проблемы до момента выполнения программы.

Тестирование систем типа V обходится очень дорого. Для тестирования системы диспетчеризации авиаперевозок в месте проведения тестирования в Атлантик-Сити, шт. Нью-Джерси, были собраны:

1) более 90 диспетчеров для работы с системой;

2) более 50 "пилотов", частично занятых местных сотрудников, имитирующих "полет" с помощью специальной аппаратуры, отсюда данные передавались на имитаторы радио локаторов;

3) более 50 наблюдателей и советников;

4) более 50 работников вычислительного центра, обеспечивающих постоянную готовность программ и проч.

В распоряжении советников и диспетчеров имелись рукописи размером с большие телефонные справочники, в которых для каждого пульта (дисплея), каждой клавиатуры, печатающего и коммуникационного устройства секунда за секундой расписывалось все, что должно быть выполнено, показано на экране и т.д.

Тестирование длилось несколько недель со средней занятостью 250 человек в день. Но — и это "но" существенно — тестирование не закончено и до сих пор. Моделирующая система никоим образом не может дать той интенсивности, какую дает использование в реальной ситуации.

Число различных ветвей в этих больших программах и число возможных комбинаций состояний входных сигналов, данных, вычислений и взаимодействий настолько велико, что даже за 100 лет использования мы сможем начать выполнять лишь несколько процентов из всех возможных ветвей. Даже после нескольких лет реальной работы в программах могут сохраняться ошибки. Было известно, что в программах управления посадкой на Луну, написанных в Хьюстоне, есть почти 100 ошибок, но ни одна из них не была существенной.

Один высокопоставленный чиновник недавно даже приказал применить против некоторых системных программистов меры дисциплинарного взыскания, поскольку они твердили, что им никогда не удастся выловить все ошибки в большой системе программ. Конечно же, они не смогут этого сделать. Тот, кто продолжает вести разговоры о больших системах программ, свободных от ошибок, не понимает в программном обеспечении ничего.

После каждого теста системы диспетчерского контроля все расхождения, отмеченные в регистрационных книгах — "признак АА 222 не зажегся, хотя шум был слышен в 8:14:03 9 июля 1972 года",— проверялись, изучались, исследовались, создавались группы для определения причин. Относилось ли это к программированию? К аппаратуре? Методике? Может быть, это ошибка пользователя — может быть, диспетчер №48 ввел неправильное число? Может быть, нарушена связь между машиной и системой отображения?

В течение нескольких месяцев в Атлантик-Сити была проведена целая серия проверок системы воздушного контроля. После завершения тестирования программы были поставлены в один из двадцати диспетчерских центров страны, и система работала там более восьми месяцев с полуночи до 8 ч. утра.

Только после этого, после исправления многих ошибок, после продолжительного тестирования, система стала использоваться во время тяжелых многочасовых дежурств в одном из центров.

Такой медленный изнурительный процесс необходим. Раньше мы уже говорили, что логическая сложность 600 тыс. строк программы была очень высока, а последствия ошибок были весьма тяжелыми.

Надежное программное обеспечение — неверный термин

Программное обеспечение не может ломаться! Оно может быть неправильным с самого начала, каким-то образом пройдя через тестирование, но сломаться, выйти из строя так, как это бывает с физическими устройствами, оно не может. Мир, для которого создавались программы, может измениться и потребовать чего-нибудь нового или другого (введен новый налог), но сломаться программа не может!

И все же люди говорят о "надежном программном обеспечении". Термин этот только наносит вред, он способствует неправильному пониманию природы программного обеспечения. Нам следует пользоваться выражениями "правильное обеспечение" или "высококачественное обеспечение". Это такое обеспечение, которое хорошо спроектировано, хорошо реализовано, хорошо оттестировано.

Устойчивым (робастным) может быть названо обеспечение, построенное так, что оно продолжает работать даже тогда, когда остальные части системы начинают действовать не так, как это планировалось. Когда вводится буква L там, где ожидалась цифра 1, программы не останавливаются, они начинают выяснять: "Вы ввели букву L, не хотели ли вы нажать клавишу 1? Эта процедура не рассчитана на ввод букв". Можно употреблять и термин "безошибочное обеспечение", но он имеет смысл только по отношению к небольшим программам.

Тестирование с возвратом

Как и все, что связано с вычислительной техникой, алгоритм тестирования целиком и полностью зависит от того задания, которое нужно выполнить, будь оно тривиальным или до крайности сложным.

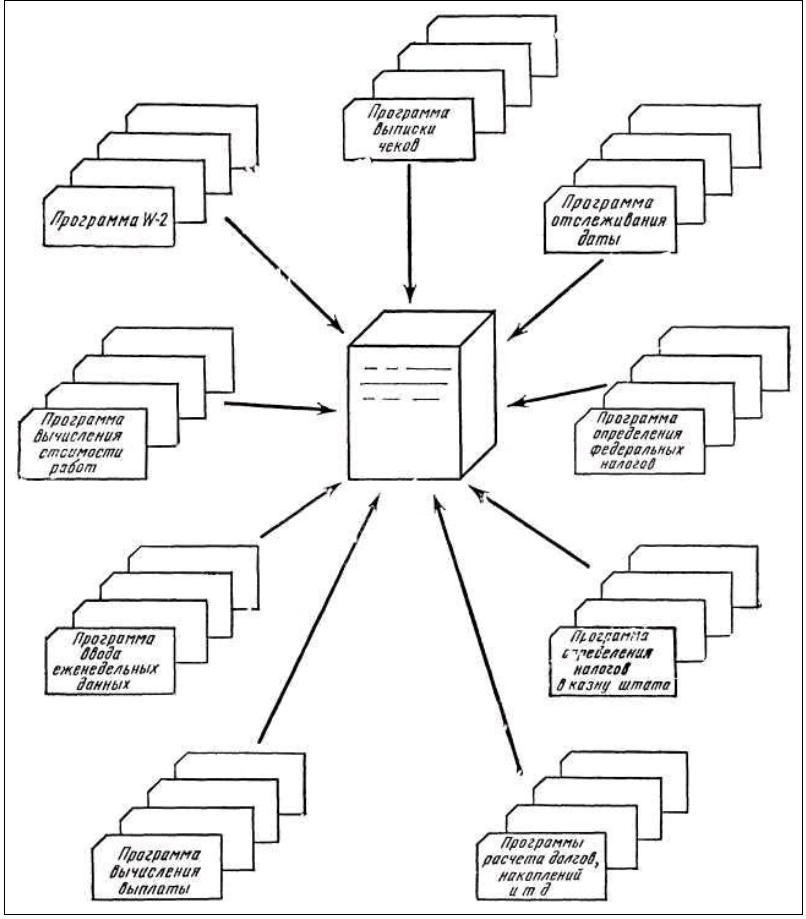


Рис. 5.48. Система программного обеспечения для составления платежных ведомостей.

Для проверки программы в 100 строк, которая вычисляет мой годовой доход, мне нужно составить всего несколько — вряд ли больше полудюжины — вариантов теста.

Для проверки программы размером в 1 млн. строк, предназначенной для управления ракетой, мне потребуются сотня или около этого помощников, целый год времени, ракетный полигон и огромное количество всяческой аппаратуры.

Давайте рассмотрим относительно простой пример — систему формирования платежных ведомостей, и на нем попытаемся понять идею тестирования с возвратом. Если разбить эту систему на составные части, получится нечто вроде рис. 5.48.

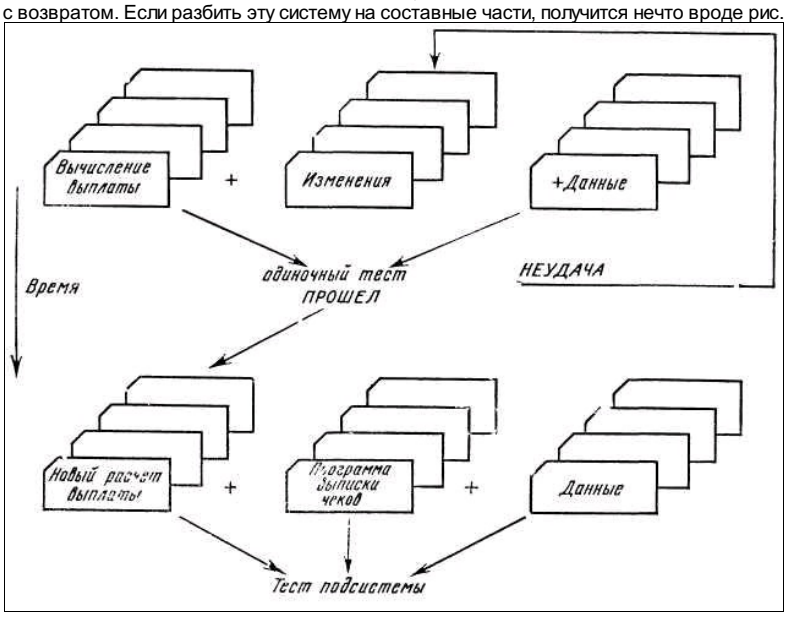


Рис. 5.49 Тестирование сверху вниз.

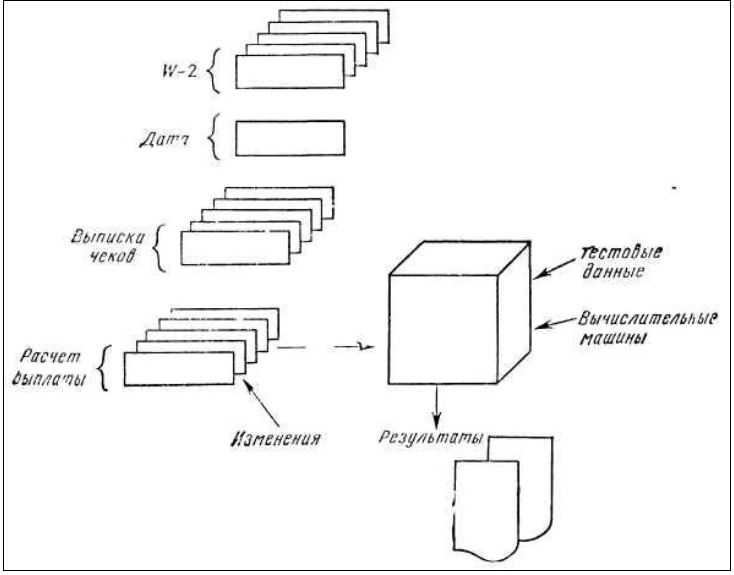


Рис. 5.50. Тестирование системы программного обеспечения.

Все эти программы достаточно независимы, но все поставляют данные друг для друга. Изменения в программе вычисления выплаты, возникающие, например, из-за нового профсоюзного договора, должны быть проверены в совокупности не только с программой контроля выписанных чеков, но также с программой вычисления даты, с программой налогообложения и т.д. (См. рис. 5.49.) То, что программа правильно работает совместно с программой контроля чеков, не означает, что она будет верно работать вместе с другими программами. Необходимо выполнить несколько тестирований.

И наконец, нужно выполнить общесистемный тест (рис. 5.50) со всеми подпрограммами.

Даже если изменения вносятся только в одну программу, повторному тестированию подлежит вся система. Этот процесс называется *тестированием с возвратом.* Проверять работу только измененной подпрограммы недостаточно. Недостаточно полное тестирование такого рода повышает вероятность неудач.

Физическая сохранность

Не раз после всеобъемлющего тестирования и успешного прохождения всех тестов система на следующее утро уже не работала, потому что руководство проектом не обеспечило ее защиты. В одном крупном проекте, разработки которого проводило одно из отделений фирмы IBM в Австралии, руководители программистских коллективов решили, что они обязаны охранять работающие строки программ. Охрана программ должна проводиться и для больших военных проектов, и для внутрифирменных разработок операционных систем. От кого же? От жуликов? Нет, от чересчур усердных программистов!

Даже после проведения всех дорогостоящих тестов могут найтись рьяные программисты, которые захотят вставить в систему еще одну, *последнюю* функцию или исправить одну маленькую ошибку в своей подпрограмме и в три часа ночи, без всякого разрешения, прокравшись в машинный зал и схватив основной диск с оттестированной системой, внести туда изменения. И пустят маленький тест для проверки своих исправлений. Этот тест проходит — для их подпрограммы.

А в девять утра “оттестированная” система на ленте или диске будет передана на использование или демонстрацию — и вот в ней начинают обнаруживаться десятки разных отказов во многих других частях. Она не работает! Но еще прошлой ночью все было в порядке! Руководство очень быстро сообразило, что весь возглавляемый им весьма дорогостоящий процесс был нарушен — хотя и без злого умысла, но с ужасными результатами. Главные файлы должны храниться под замком с соблюдением всех мер безопасности.

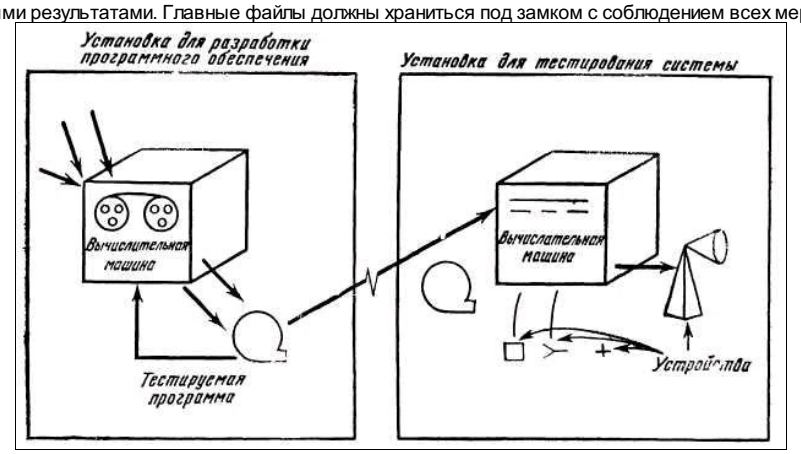


Рис 5.51. Два вычислительных центра — один для разработки, другой для тестирования.

Для тестирования обычно требуются очень крупные денежные средства. Проведение тестирования требует крупных и дорогих машин и программ. Это *могут* быть те же самые машины, что использовались и для разработки, но могут быть и другие. Все зависит от того, насколько велика система, является ли она аппаратно-интенсивной или программно-интенсивной. Если система достаточно велика, нам придется использовать для разработки и тестирования разные вычислительные комплексы (см. рис. 5.51).

Тестирующий комплекс проверяет не только новые или изменившиеся программные модули, но также все изменения в аппаратуре радиолокаторов, датчиков, линий связи, дисплеев и т. д. со всем существующим и оттестированным программным обеспечением.

Тестирование — средство обучения

В отделениях тестирования работают очень квалифицированные, всесторонне развитые люди. Тестовики должны уметь видеть всю систему, ее цели, структуру, иначе проверить ее нельзя. Лишь очень немногие среди людей, работающих в большом проекте, еще обладают таким кругозором. Остальные строят отдельные части, и лишь немногие проектировщики, руководство и занятые в тестировании видят целое.

При тестировании происходит целеустремленный поиск слабых мест, а для этого нужно проштудировать всю систему. Это хорошая тренировка. В течение многих лет мы замечаем, что люди, начинавшие с тестирования, становятся проектировщиками и руководителями новейших систем. Мы всегда преднамеренно поручаем тестирование очень способным молодым программистам, поскольку знаем, что уклонение от тестирования при реальном использовании приведет систему к краху. Лучше потерпеть крах до реального использования!

Независимость тестирующих групп

Тестирующая организация должна быть независимой, насколько это только возможно, ее отчеты должны направляться по цепочке руководства как можно выше. Если руководитель тестирования подчинен руководителю разработки, сразу начнут сказываться человеческие слабости — руководитель группы тестирования может сильно волноваться, критикуя систему, ведь критикуемым окажется его начальник.

Продолжительность развития

Случается, что после выпуска первой версии большой программной системы тестовые данные, ленты с тестами, результаты тестирования, сами тестовые программы выбрасывают за ненадобностью. Следует помнить, что группа сопровождения должна будет повторно проводить весь цикл тестирования.

Выводы

Судьба разработки большой программной системы решается в фазе тестирования. Конечно же, мы заранее знаем, что нам никогда не удастся полностью проверить систему, но мы также знаем и то, что чем больше мы проверим, тем лучше система будет работать впоследствии.

В работе по проверке программного обеспечения мы можем следовать следующим ориентирам.

Планирование тестирования должно начинаться одновременно с началом работ по разработке системы. Тестирование никогда не должно прекращаться — и варианты тестов, и аппаратное обеспечение тестов, и группы тестирования должны сохраняться в продолжающейся разработке или сопровождении.

Изменения в одном модуле программы требуют тестирования *как* этого нового модуля, *так* и всех остальных. Это называется тестированием с возвратом. Самая последняя, проверенная версия системы программного обеспечения должна храниться под надежной охраной, взаперти. Для тестирования нужно выделять самых квалифицированных членов группы разработки (кроме того, тестирование это лучший способ обучения специалистов).

Если пытаться извлечь из тестирования большой системы типа V хотя бы минимальную пользу, оно становится весьма дорогостоящим, С помощью тестирования нельзя плохую программу превратить в хорошую, ее можно только улучшить.

При тестировании следует обращать внимание на граничные случаи, это необходимо, чтобы убедиться, что система работоспособна во всем диапазоне, для которого она проектировалась. При тестировании необходимо вводить в систему неправильные данные, исследуя затем полученные результаты.

Медленно развивающиеся процессы следует тестировать с помощью длительных прогонов (24 ч.).

Документирование

Люди не очень-то любят составлять документацию на созданную ими продукцию. В то же время документация — это самое важное из того, что они должны сделать! Некий превосходный программист спроектировал и написал программу определения орбитальных характеристик спутника. Он первым закончил программирование, все работало правильно, память попусту не тратилась. Программа была написана на Фортране и занимала около 4 страниц плотного фортрановского текста. Он знал свою программу вдоль и поперек. Через 3 мес. его попросили добавить к своей программе несколько функций. Он достал документацию и принялся ее изучать. Три или четыре дня он пытался понять, что же происходит в его программе! А ведь он ее сам написал! Сколько бы сил он потратил, если бы это была чужая программа! А теперь представьте, что документации не было.

С такой проблемой часто приходится сталкиваться тем, кто занимается продолжающейся разработкой программ. Вместо того чтобы начинать с изучения требований к программе и ее проекта, им слишком часто достается участь детективов и археологов. “Какая была форма — структура — план — путь?” Почему “они” сделали “это”? После этого начинаются исправления, исключения, добавления. Понятно, что чем более четкий оставлен след, тем проще задача, тем лучше результаты.

Наверное, наиболее распространенная в мире история о программировании — это рассказ о том, что программа работает, но ни один человек не знает, как это происходит, поскольку программист давно ушел, а документации нет. Эту историю часто рассказывают, потому что это весьма обычная ситуация.

Документация нам нужна для того, чтобы

1) напомнить тем, кто создал эту программу, о том, как она устроена;

2) показать тем, кто придет после нас, как устроена программа, каков ее алгоритм.

Самодокументирование

Многие новые технологии программирования позволяют получать самодокументированные программы, значительно снижая усилия, необходимые для документирования.

Под документацией мы подразумеваем не тексты рабочей (машинной) программы. Недостаточно также и текстов исходной программы. Документация, подготавливаемая для фазы продолжающейся разработки, должна содержать

1) хорошо прокомментированный текст программы;

2) схемы, иллюстрирующие проект, и словесное их описание;

3) структурированные словесные описания или схемы процессов, причем первое более предпочтительно;

4) описания данных.

Без всего этого программисту, которому предстоит модифицировать существующую программу, придется решать сложнейшую задачу по определению того, что было сделано до него.

Структурированное словесное описание

Такие описания и читать и понимать легче, чем блок-схемы. Снабженный подробными приложениями, грамотно оформленный, разделенный на модули текст описания усваивается особенно легко, ведь и общая структура текста, и используемые в нем термины вполне понятны любому. Структурированное словесное описание это не программа, но эти понятия близки.

```
PROCEDURE: PWARN $ display warning message
INPUTS:
LOTMP -- time to retry after overtemp condition
LEMI -- time to retry after EMI condition
LACT -- printer queue overflow flag
OUTPUTS:
none
IF LACT set THEN
IF NOR less than TQR (space unavailable) THEN
clear LACT
set LCODE to 134
call procedure POPMG to display operator message
ENDIF
ENDIF
DO-WHILE LACTX is 0 thru 1
IF EMI is set and CSRTC is greater than or equal to LEMI(LACTX) THEN
set LCODE to 321
call procedure POPMG to display warning message
store the new retry time in LEMI
increment retry counter LRCTR
IF LRCTR greater than or equal 3 THEN
clear the FM! flan
ENDIF
ENDIF
IF OVRTMP is set and CSRTC greater than or equal LOTMP(LACTX) THEN
set LCODE to 320
call procedure POPMG to display warning message
store new retry time in LOTMP
increment retry counter LRCTR
IF LRCTR greater than or equal to 3 THEN
clear OVRTMP flag
ENDIF
ENDIF
END-WHILE
END PROCEDURE-OUTPUT INTERRUPT
```

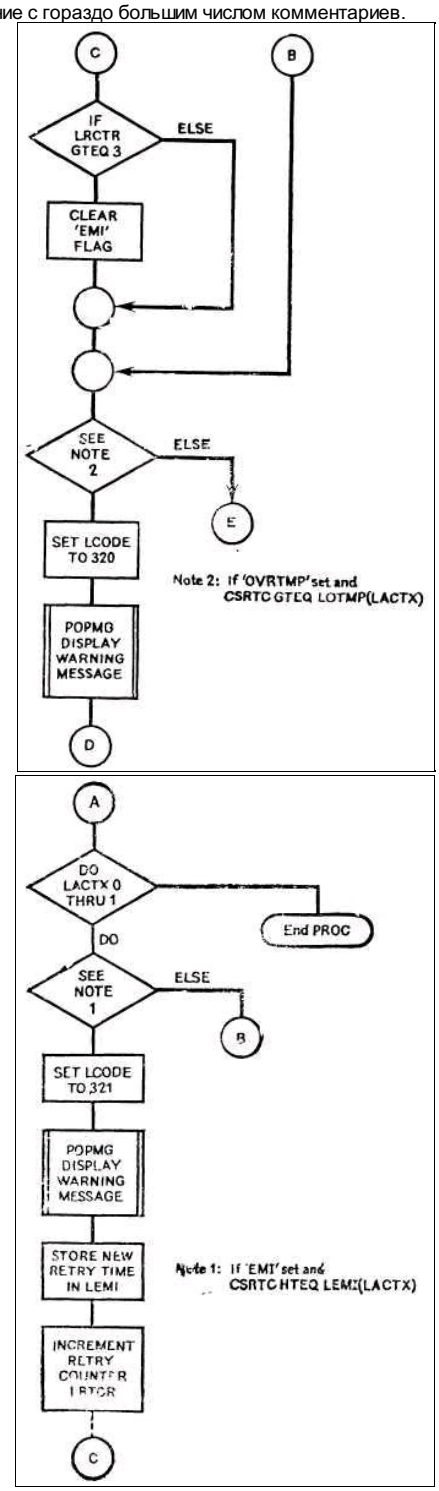
Рис. 5.52. Структурированное описание на языке проектирования программ.

На рис. 5.52 приведено структурированное словесное описание, для которого использован язык проектирования программ PDL. Это описание эквивалентно нескольким блок-схемам, которые изображены на рис. 5.53. Словесное описание лучше блок-схем по следующим причинам:

1) оно располагается на одной странице, и его общая структура легко обозрима;

2) оно содержит больше информации, чем блок-схемы.

На рис. 5.54 приведено другое описание с гораздо большим числом комментариев.



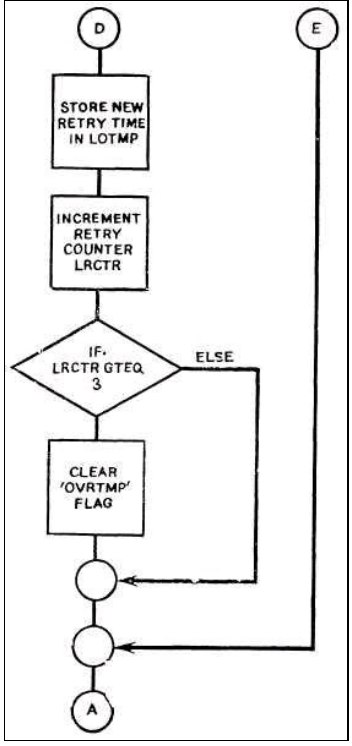


Рис. 5.53. Блок-схемы.

```
PROC SESSION MANAGEMENT * THIS PROCEDURE MANAGES THE
TERMINAL INTERACTION WITH THE USER. LEGAL USER
COMMANDS ARE MOVE AND DELETE *
USE SESSION DATA
DO * PROCESS USER COMMANDS *
GET INPUT (COMMAND) * NEXT USER INPUT *
RUN INPUTCHECK (COMMAND, ERROR)
IF
ERROR = TRUE
THEN
PUT OUTPUT (ERROR)
ELSE * NO ERROR - PROCEED WITH PROCESSING *
IF * DETERMINE TYPE COMMAND *
COMMAND = MOVE
THEN * PROCESS MOVE COMMAND *
INCLUDE MOVE PROCESSING
ELSE * PROCESS DELETE COMMAND *
INCLUDE DELETE PROCESSING
FI
FI GET INPUT (SESSION ~ ON)
WHILE * KEEP PROCESSING INPUT COMMANDS AS LONG
AS SESSION ON INDICATOR IS ON (TRUE) *
SESSION ON = TRUE
OD
CORP
DATA SESSION DATA
*ABSTRACT DATA TYPES & COMMENTS *
ATAD
```

Рис. 5.54. Структурированное описание.

Служебные слова CORP,OD, F1 и ATAD представляют собой “закрывающие скобки” для слов PROC, DO, IF и DATA. Если правила ясны, то документы такого типа читаются очень легко.

Строки типа “IF * ОПРЕДЕЛЕНИЕ ТИПА ПРИКАЗА *” необходимо еще будет переводить на язык, с которого можно осуществлять трансляцию, но уже на этом уровне совершенно ясно, что нужно делать. В совокупности с хорошим общим описанием проекта такого рода документация должна быть вполне достаточна для продолжающейся разработки. Документы этого уровня можно обрабатывать с помощью машин, но транслировать их в рабочую программу еще невозможно.

Документация для других целей

Знать, что происходит в программе, должны не только люди, занимающиеся сопровождением, но и многие другие.

— Оператору, человеку, нажимающему кнопки в машинном зале, нужно подробно рассказать, что, когда, при каких обстоятельствах делать.

— Пользователь, сидящий за терминалом, является одним из тех, для кого строилась система. Пользователь должен иметь достаточно информации о том, что, как, по чему, когда происходит в системе. Информация должна быть изложена на достаточно понятном уровне.

— Руководителям пользователей необходима документация нескольких разных уровней. Что система делает? Чего она не умеет делать? Что возможно? Легко? Трудно?

— Для управления ходом разработки руководители разработки должны регулярно получать отчеты о состоянии дел и результаты тестирования. Большая часть этих документов в конце концов выбрасывается, поскольку почти вся она подчинена текущему моменту.

— Руководство пользователей должно иметь возможность знакомиться и изучать планы реализации или ввода в эксплуатацию.

— Руководство пользователей должно периодически изучать планы с требованиями на исходные данные. Что нужно от пользователя, чтобы эти данные не устаревали и были правильными? Группа сопровождения должна иметь намного больше разной документации, чем первичные разработчики, ведь ей приходится и модифицировать, и исправлять систему.

Отслеживание связей

Если взглянуть на заднюю панель любой очень большой вычислительной машины, станет очевидно, что нам совершенно необходима схема или хотя бы список, содержащий сведения о каждом проводке из точки XYZ в точку QLR. Это же относится и к программному обеспечению. Рассматривая каждый модуль как отдельную схему, можно сообразить, что нам потребуется отслеживать, кто, что и для кого делает. Но визуально представить себе программу труднее чем электронное устройство. Что такое модуль? В лучшем случае это наименьшая отдельно транслируемая часть программы, но такое толкование крайне изменчиво и в большой степени зависит от авторов. Один модуль может выполнять несколько функций. Для управления нашей большой программной системой нам нужно иметь

таблицу привязки функции к модулям, и наоборот. Часто оказывается полезной схема вроде представленной в табл. 5.5 (см. также рис. 5.55)

Изучим теперь каждый столбец в табл. 5.5 и посмотрим, о чем же они нам рассказывают:

Столбец 1. Описание функции. Насколько это возможно, оно должно говорить само за себя.

Столбец 2. Номер модуля. Пользуясь этим номером, можно получать доступ к описанию модуля, тексту исходной программы и даже к рабочей программе, получающейся из модуля.

Столбец 3. Автор программы.

Таблица 5.5.

Описание функции	Номер модуля	Автор	Данные поступают от	Данные готовятся для	Вызывает	Вызывается
Программа	848А	Дэниэлс	848А	437 849 Печати	849 ОС Планировщик	
Программа печати чеков	852	Шварц	849	Нет	Возврат	831
Расчет профвзносов	857	Трэверс	839	858	858	852
Калькуляция	1612	Уард	442	894	1614	1610
			857	1631		
Печать калькуляции	1614	Уард	1612	Нет	Возврат	1612

Столбец 4. Какие модули или устройства формируют данные, используемые в данном модуле; имеются в виду непосредственные *источники,* а не вся их совокупность.

Столбец 5. Какие модули или устройства *получают* данные от данного модуля.

Столбец 6. Каким модулям может *передаваться управление* из данного модуля.

Столбец 7. Каким модулем *вызывается данный.*

Подобная автоматизированная система позволяет упорядочить и отрегулировать процесс обнаружения и устранения ошибок. Если размеры системы доходят до нескольких сотен тысяч строк текста программ и больше, число модулей и функций может достигать не одной тысячи. Автоматизированная система становится совершенно необходимой.

К счастью, в настоящее время уже существует множество инструментальных программ, помогающих нам проводить отслеживание требуемой информации, обеспечивающих легкий доступ к ней и возможности модификаций. И опять при разработке программ приходят на помощь вычислительные машины.

Избыток документации

Документация должна в точности соответствовать тем целям, для которых создавалось программное обеспечение. Если программы будут работать на 500 машинах, на борту каждого корабля ВМФ США, их надо документировать более тщательно, чем программы, используемые лишь однажды и затем выбрасываемые. Документацию для этих последних программ можно составлять на оборотной стороне конверта.

Если пользователей у программы нет, составлять пользовательскую документацию не следует. Звучит это довольна глупо, но почему-то часто все-таки случается. В “Стандартах” отмечена необходимость пользовательской документации, т.е. документов, в которых говорится о том-то и том-то, поэтому руководство требует, чтобы эти документы составлялись. И вот создается двухметровая кипа документов, ни один из которых ни разу не будет использован.

Исключение блок-схем

Блок-схемы больше не считаются необходимыми при составлении документации, за исключением самых высоких уровней. Готовить их трудно: в клеточки блок-схем можно вписывать весьма ограниченные сведения, занимают блок-схемы много страниц, они редко поддерживаются на уровне самых новых версий. Блок-схемы могут оказать полезную помощь при проектировании, но не при документировании.

История проекта

Занесение в документацию сведений о причинах принятия тех или иных решений при проектировании оказывается исключительно полезным, когда приходит время вносить в программы исправления. Почему они выбрали именно такое построение? Намеренно ли был выбран определенный путь решения определенных вещей? Почему? Если такие решения хорошо задокументировать, они помогут вносящим изменения избежать дорогостоящих ошибок. На этой стадии история проекта и заметки по решениям, принятым в проекте, могут быть очень полезными.

В общем должен еще раз подчеркнуть, что большие программные системы типа V требуют кучу документации, и так же как для эффективного проведения разработки программного обеспечения абсолютно необходима некоторая инфраструктура, так и для доводки и развития этого обеспечения необходима отличная документация.

“Как” — это “что”. Требование — это проект — уровни детализации

Люди постоянно стремятся провести разграничительную линию между понятиями *как* и *что*. И это им никогда не удается, поскольку эти понятия едины и неразрывны. Все зависит от того, в каком отношении к этим понятиям вы находитесь.

Документы выполняют сразу две различные роли — во-первых, они нужны для управления разработкой, и, во-вторых, они нужны для управления продолжающейся разработкой, исправлением и использованием. В первом качестве они до некоторой степени опережают события, показывая, что предстоит сделать. С другой стороны, документация до некоторой степени “отстает”, в ней фиксируется то, что уже сделано. По мере создания системы такая двойственная роль начинает доставлять все больше неприятностей. Посмотрим, почему это происходит.

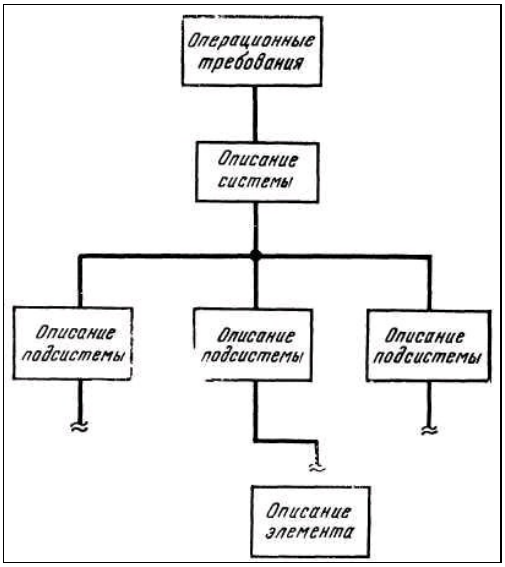


Рис. 5.56. Уровни документации, применяемые в системе Министерства обороны США.

После завершения нашей разработки мы получим целую иерархию документов, несколько их уровней, каждый из которых описывает систему все подробнее.

Такие иерархии документов существуют во всех технических отраслях. Министерство обороны установило весьма формализованные правила на составления спецификаций. В общих чертах схема таких спецификаций представлена на рис. 5.56.

По мере продвижения дел в нашем проекте наступает момент, когда мы *получаем возможность* написать спецификацию системы. Уровень детализации, с которым надо создавать эту спецификацию, весьма проблематичен. Какую же степень подробности

нужно считать достаточной?

Чтобы проектировать новую сложную систему, нам нужно подробнейшим образом изучить *некоторые важные аспекты* этой системы, в особенности ее отличия от предыдущих, это необходимо, чтобы система могла функционировать. Но все эти детали не являются принадлежностью спецификации системы. И все же, поскольку они у нас есть, мы испытываем искушение включить их туда. Однако они чересчур подробны для описания столь высокого уровня. Они раздражают читателя, который не понимает да и не нуждается в таких деталях.

С другой стороны, что нам необходимо для утверждения спецификаций верхнего уровня? Нам может потребоваться большое число разных сведений по различным пунктам, *не* входящим в данный документ, но собранным в различных местах проекта. Нам могут понадобиться *черновики* будущих документов или детальные эскизы и описания, содержащиеся в рабочих документах, которые, конечно же, сосредоточены на более низких уровнях.

Еще раз повторяясь, можно сказать, что после достижения *некоторого* уровня детализации делаются попытки перенести его на документы более высокого уровня. В этом и есть ошибка. Документация высокого уровня должна быть свободна от этих подробностей. Для сопровождения спецификаций высокого уровня можно привлекать внутренние рабочие документы, но объединять документы не следует.

Требования должны содержать сведения о том, что является необходимым, а спецификация проекта должна описывать, как нужно этого достигнуть. Всегда нужно помнить, однако, что каждая спецификация есть одновременно и “как”, и “что” — все зависит от того, с какой стороны вы на них смотрите.

Реальная ситуация

Необходимость документации совершено понятна. Столь же понятно и то, что часто не существует никакой документации — абсолютно никакой. Плохая документация — это одно, отсутствие документации — это совершенно иное. Существует столько программ без документации, что проблема кажется глобальной. Документация на программное обеспечение является более необходимой, чем документация па аппаратуру, ведь программы неосязаемы и абстрактны. Программа без документации похожа на невзорвавшуюся бомбу. В один прекрасный день она может взорваться.

ГЛАВА 6. РУКОВОДСТВО РАЗРАБОТКОЙ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Эта глава посвящена описанию тесно связанных между собой вопросов, возникающих на протяжении различных стадий процесса разработки. Описание построено так, что каждая проблема рассматривается в момент своего наиболее вероятного возникновения. Все эти проблемы непременно возникают, если пренебрегать некоторыми предупреждениями. Некоторые из них появляются на ранних стадиях разработки проекта, некоторые позднее. Понимание этих проблем досталось мне на многолетнем опыте, иногда весьма тяжелом. После разъяснения многие из них могут показаться очень простыми, но, однако, часто их упускают из вида.

Давайте поднимемся на несколько ступеней по служебной лестнице и предположим, что на нас возложена обязанность по управлению разработкой большого проекта. Мы определим очертания всех наших подсистем и для каждой подсистемы назначим своего компетентного руководителя. Конечно же, мы захотим следить и за ходом разработки нашего программного обеспечения. Мы слишком обеспокоены работами в других подсистемах, чтобы подробно заниматься управлением разработкой программного обеспечения; это входит в обязанности человека, которого мы самым тщательным образом выбрали в качестве руководителя разработкой программного обеспечения. В этой и следующих главах мы концентрируем свое внимание на вопросах, служащих для высшего руководства индикаторами “состояния здоровья” проекта. Мы посмотрим, как надо организовать группу программистов; как выбрать руководителя разработкой программного обеспечения; каким образом определить, разрабатывать ли обеспечение самим или заказывать его на стороне; каким образом подобрать подрядчика. Мы внимательно изучим некоторые места, в которых проекты спотыкались в прошлом, а также места, которые представляют грозную опасность и сейчас.

На что надо обращать особое внимание? На что направлены наши поиски? Что вызывает наши споры? Какие моменты наиболее показательны?

Системы, подсистемы и программное обеспечение

“Закон Мьюира: Как только мы захотим отделить какой-нибудь объект от других, мы обнаружим, что он связан со всем на свете”. *{Block A. Murphy's LawBook Two (Los Angeles, Calif.: Price/Stern/ Sloan. Publishers, Inc., 1980).}*

Общесистемная незамкнутость

Существует только одна система — Вселенная. Все другие системы можно называть так только с натяжкой: они не замкнуты.

Какую систему мы имеем в виду, говоря о системе здравоохранения Соединенных Штатов? О системе, очерченной на рис. 6.1 сплошной линией? Или пунктирной линией?

Рис. 6.1. Определение системы.

А может быть, эта система имеет очертания, ограниченные линией с точками? Задумываемся ли мы о том, что наш собеседник может представлять себе систему, о которой мы с ним разговариваем, совершенно иначе, чем мы?

Взгляд сверху вниз

Если бы я руководил более или менее крупной компанией, я стремился бы к тому, чтобы платежные ведомости печатались в срок, а при резервировании билетов на самолеты не возникало никаких ошибок. Предположим, что ведомость подготавливается недостаточно эффективно или ее подготовка затягивается. Придется мне самому исследовать возникшую проблему. Я буду изучать дела уровень за уровнем. Начну я с человека, отвечающего за ведомости, и попрошу его описать мне систему сверху вниз. Он может разбить систему на пять компонент: 1) процедуры; 2) люди; 3) вычислительные машины/системы; 4) входные данные и 5) распределение. На этой стадии у меня возникает возможность более глубоко ознакомиться с любой из этих пяти компонент. Поскольку нас интересует вычислительная сторона дела, я попрошу описать мне следующий уровень разбиения именно области вычислений. Это разбиение приведено на рис 6.2.

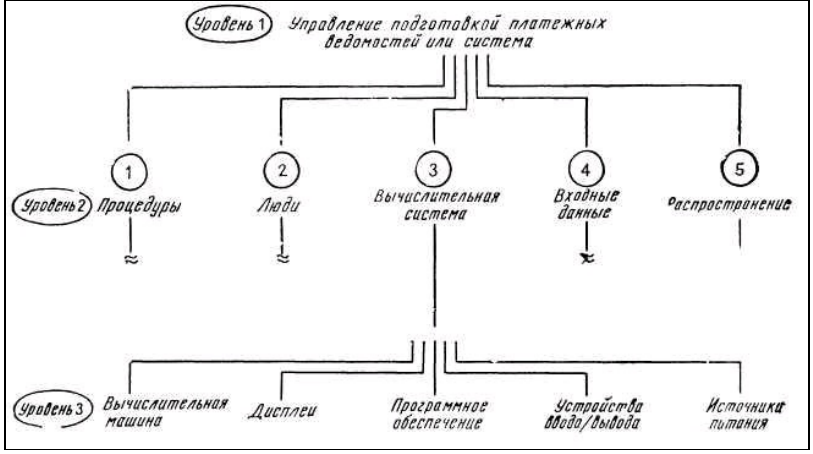


Рис 6.2 Деление системы на подсистемы.

Можно заметить, что отдельные элементы второго уровня сложнейшим образом переплетаются между собой. Выбор покупаемых нами дисплеев (аппаратура) и то, что мы собираемся на них высвечивать, очень сильно влияет на 1) процедуры и 2) людей, а возможно, и на 4) входные данные и 5) распределение. Помните, существует только одна система— Вселенная, все остальные системы имеют прорехи! Теперь перейдем к третьему уровню (рис. 6.3). Пока мы отбросим подсистемы 1, 2, 4 и 5, в реальной же ситуации этого сделать нельзя (описывать эти подсистемы нам не позволяет размер книги). Все нами сказанное остается верным и при переходе на уровень 4 (см. также рис. 6.3).

На четвертом уровне, как и на всех других, мы можем произвольно выбирать дальнейшее разбиение системы. На этом уровне мы можем составить дополнительные системы классификации, для примера мы остановимся на схеме уровня 5 (рис. 6.4).

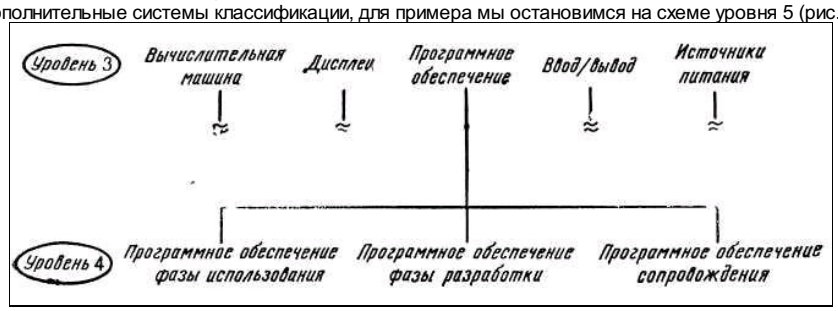


Рис. 6.3. Следующие уровни подсистем

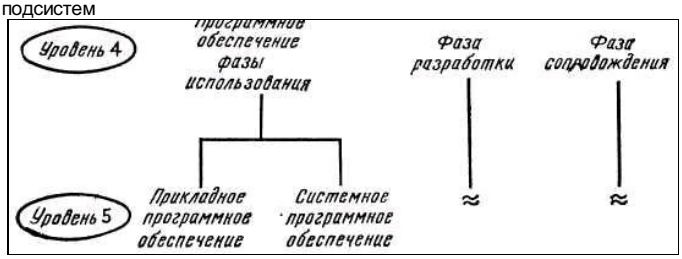


Рис. 6.4. Подсистема программного обеспечения.

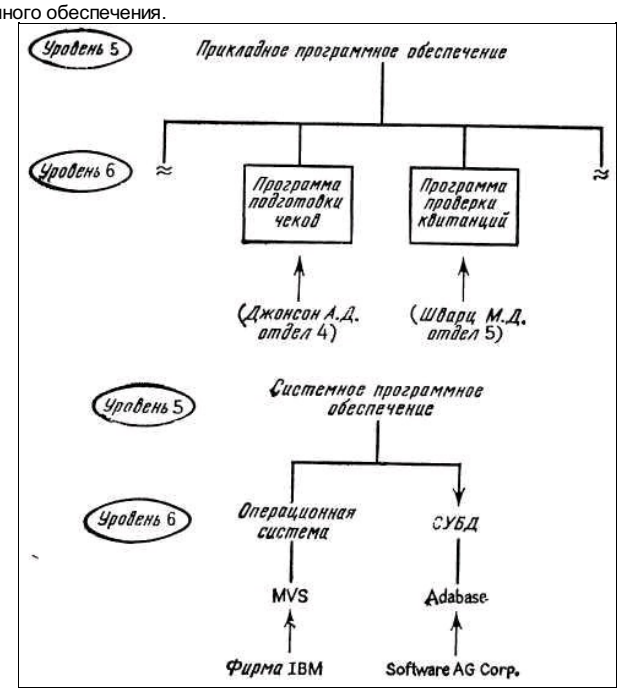


Рис. 6.5. Авторы программ

Нам нужно выяснить, почему платежные чеки оказываются неправильными, для этого нужно изучить программы, работающие в фазе использования. Два оставшихся раздела программного обеспечения будут нами исследоваться только I том случае, если проблемы фазы использования приведут нас назад к одному, а то и к обоим этим разделам. Перейдем к уровню 6 (рис. 6.5). Здесь мы показываем обе ветви, указывая в скобках авторов программ.

Ограниченный объем книги опять не позволяет нам привести полный список всех выполняемых отдельных программ. Мы могли бы и должны были бы продолжить составление диаграммы уровня 7, на которой надо показать модули всех программ, но опять-таки на это не хватает места. Матричная диаграмма в табл. 6.1 показывает, какого рода подробности нам необходимы. Теперь, если выяснится, что все проблемы относятся к области “чистой выплаты”, мы начнем изучать и тестировать модули, влияющие на чистую выплату.

Таблица 6.1. Программы, выполняемые для печати платежных ведомостей

Программа	Размер	Автор
Программа выписки чеков	8402 Стп*	Джонсон А.Д., отдел 4
Печать квитанций	462 Стп	Шварц М.Д., отдел 5
Программа	892 Стп	Дэниэлс Р.М., отдел 11
Удержания из заработной платы	440 Стп	Абадан Д.Р., отдел 442
Расчет даты	414 Стп	Уитерс М.Р., уволился
Отчет по налогам штата	317 Стп	Джонсон А.Д., отдел 4
Расчет профсоюзных взносов	219 Стп	Трэверс Д., отдел 41
Печать профсоюзных взносов	44 Стп	Трэверс Д., отдел 41

*СТП — строки текста программы на исходном языке

Мы достигли достаточно точного и подробного видения проблемы. Ограничимся рассмотрением уровня 6, поскольку он более удобен для записи и чтения, хотя на практике лучше работать с более подробным уровнем 7. Мы хотим отметить все, что выполняется во время использования. Нам нужно для этого перечислить все программы, их размеры и их авторов. Результаты сведены в табл. 6.1.

Теперь для приведения такой системы в порядок, для ее доводки нам нужны люди (системные аналитики и программисты), которые понимают, что должна делать каждая программа, как эти программы взаимодействуют между собой, каким образом все выполняемые машиной программы должны составить единую систему.

Мы не ждем, что А.Д.Джонсон, автор программы выписки чеков, будет понимать, каким образом СУБД выполняет возложенные на нее обязанности. Мы можем лишь требовать от него, чтобы он знал, что может делать СУБД, как обратиться к ней с требованием выполнить работу, как проверить, что работа выполнена.

То же самое относится и ко всем другим программам. Небольшая группа старших аналитиков должна иметь возможность просмотреть все данные и с помощью авторов и поставщиков программы найти ошибку или ошибки. Ошибка может находиться в этой подсистеме на шестом уровне, но может быть и в любой другой подсистеме процедур, людей и т.д. Конечно, ошибка может быть и в способе взаимодействия какой-нибудь подсистемы с одной или несколькими другими.

Все это сложно, очень сложно, но совершенно необходимо. За несколько часов мы можем “сделать” такую ведомость, что потом ее не удастся привести в нормальное состояние и за целую неделю, даже после стократного увеличения числа людей, занятых этой работой. Мы можем это же проделать и с системой управления авиалиниями, и с другими работами.

Зачем наряду с программными системами собственного производства мы используем еще и покупные? Такой подход повышает эффективность, он более экономичен, к тому же продаются уже готовые к использованию программы.

Различные подсистемы обработки данных в одной системе

Система состоит из множества подсистем — одна из них обычно есть вычислительная система, например система наземного контроля. Но и во всех других подсистемах тоже могут быть и вычислительные машины, и программное обеспечение! (См. рис. 6.6.) Вычислительные машины и программное обеспечение могут входить в состав любой подсистемы как ее часть. Спутниковая подсистема может иметь в своем составе сразу несколько вычислительных машин, на каждой из которых имеется свой комплект программного обеспечения (см. рис. 6.7).

Один простой вопрос может сбить с толку всю организацию: “Кто должен отвечать за программное обеспечение?”. Разумеется, для программного обеспечения наземного контроля необходим компетентный, очень квалифицированный руководитель разработки программного обеспечения проекта. Должен ли он отвечать и за то обеспечение, которое работает на самом спутнике? Возможно. А возможно, и нет.

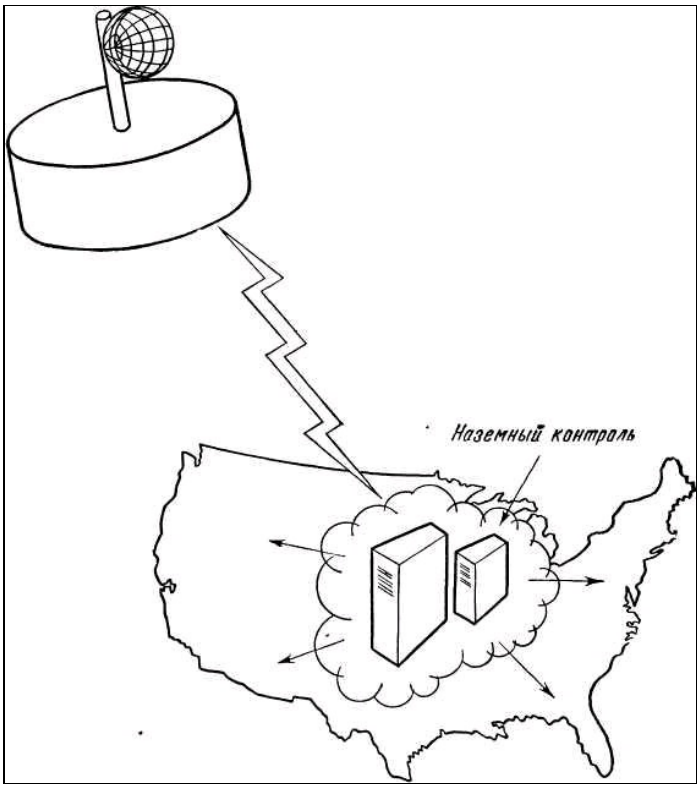


Рис. 6.6. Спутниковая система связи.

Должен ли руководитель отдельной бортовой спутниковой подсистемы заниматься и вопросами программного обеспечения? Вероятно, нет. На эти весьма важные вопросы нет простого ответа.

Отделение программного обеспечения от аппаратуры

На некоторое время постараемся забыть об аппаратуре и обратимся только к программному обеспечению. Но вот удастся ли нам отделить программное обеспечение от аппаратного? Хотя это подчас и затруднительно, но такое выделение возможно. Особое значение оно приобретает при определении границ работ различных организаций и ревизионных групп.

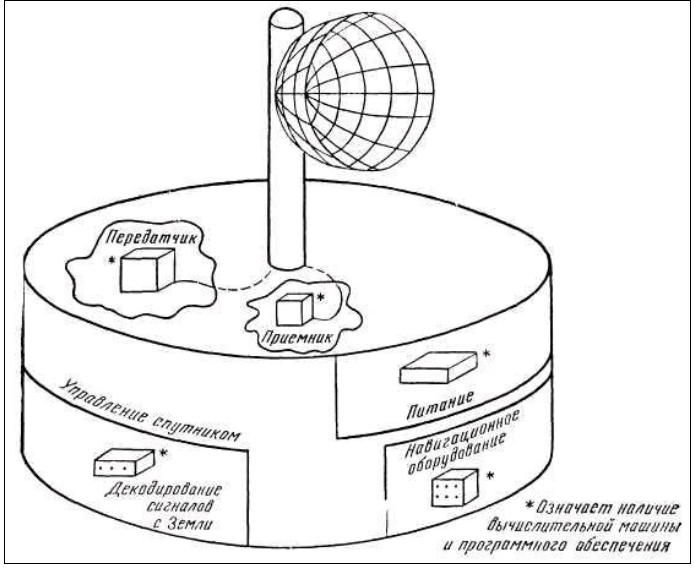


Рис. 6.7. Спутниковая подсистема.

Рассмотрим для примера банковскую систему с диалоговыми терминалами, служащими для проведения банковских операций. Начать мы можем с рассмотрения ее подсистем. Чтобы не разбрасываться, мы будем принимать во внимание только три подсистемы: подсистему связи, аппаратную подсистему обработки данных и подсистему программного обеспечения, которая в свою очередь входит в подсистему обработки данных. Нам нужно определить, откуда возникает ограничение на количество подключаемых терминалов, которых может быть, скажем, не более 12.

Изучая подсистему связи, мы можем обнаружить, что линии, по которым происходит передача данных на обработку или после этой обработки, имеют ограниченную пропускную способность в 12 сообщений, что и является ограничивающим фактором. Если это действительно так, мы можем считать, что узкое место найдено. Но предположим, что линии связи способны передавать любой мыслимый объем информации.

Обратимся теперь к вычислительной машине. Может оказаться и так, что невозможность управления тринадцатым терминалом связана с ограниченной вычислительной мощностью машины, которая не может гарантировать обслуживание линий без потерь информации в периоды пиковой нагрузки. Поскольку этот вопрос упирается в деньги, он очень важен. Узким местом при этом становится мощность процессора.

Допустим теперь, что и линии связи, и процессор вполне способны управлять более чем двенадцатью терминалами. Программисты могли написать программы так, что они работают с 12 и только 12 терминалами, тем самым просто не существует программы, контактирующей с тринадцатым терминалом, даже несмотря на то что вся аппаратура имеет полную возможность для этого. Если узкое место найдено именно здесь, программистов можно попросить вникнуть в суть дела и модифицировать программы так, чтобы можно было работать с дополнительными терминалами. Это достаточно просто и не должно отнять много времени, конечно, в том случае, если программист, выполнявший первичную разработку, оставил после себя документацию на программу, все еще работает в этом банке и не занят в других критических разработках.

В некоторых системах может быть сразу несколько узких мест в разных подсистемах. Эти узкие места могут различаться тем, что одни из них связаны с аппаратными ограничениями, а другие возникли из-за чисто программных причин. “Балансировка системы” состоит в том, чтобы избегать ситуаций, когда одна подсистема оказывается намного слабее остальных. Мы уже говорили, что хорошее программное обеспечение может заставить хорошо работать плохую аппаратуру; верно и обратное утверждение. На с.168 приведен пример обработки запросов, которая может проводиться аппаратурой несколькими разными способами. При этом внешние характеристики системы определяются программным обеспечением.

Перегрузка аппаратной подсистемы, которую можно принять за неполадку с программным обеспечением

Когда к военно-морской тактической системе обработки данных было подключено программное обеспечение линий связи LINK 11, система стала останавливаться либо выдавать неправильные результаты. В новой программе было что-то не так!

Нет, все было правильно. Дело заключалось в том, что подключение новых функций перегрузило машину. Нужно было сделать слишком много, машина стала отставать, количество входных данных “стало больше”, чем машина могла обработать за отведенное для этого время.

Длительное время велись поиски, исследования новой программы, пока наконец не было осознано, что машина просто перегружена.

Стоимость и график разработки предсказать невозможно

Термин “разработка” используется настолько широко, мы так часто ею заняты, что иногда вводим себя в заблуждение и начинаем думать, что полностью владеем этим процессом. Всякий сразу согласится с тем, что затраты и результаты исследований предвидеть нельзя. И каждый сразу же согласится с тем, что производство даже очень сложной продукции можно и контролировать и предсказывать. Ну а что же такое разработка? В словаре говорится, что разработка это 1) “постепенное развертывание” и 2) “воплощение в жизнь”. Разработка не относится к изученным и просчитанным видам человеческой деятельности — будь то в программировании или в других областях. Ни в одной отрасли техники невозможно предсказать время и суммы, необходимые для разработки новых вещей. Мы можем предсказывать производство, но производство имеет циклический характер по самому своему определению, в производстве есть процесс обучения, и, если следовать ему, как мы знаем, можно добиться того, что последний экземпляр окажется дешевле первого. Создавая объемистую книгу, посвященную изучению живых организмов и организаций, Дж. Г. Милер в *“Живых системах” {James Grier Miller, 'Living Systems' (New York: Me Graw-Hill Book Co., 1978).}* цитировал исследования Джоан Вудворд, проводившиеся в Англии, и сразу комментировал их.

Еще одним исследователем, распознавшим важность производственник процессов, оказалась Вудворд. В одном из относительно редких исследований, в которых собирались данные по большому числу организаций, она проанализировала 100 промышленных предприятий Южного Эссекса в Англии... Внутри часто выделяемых категорий “штучного”, “серийного” и “массового” производств Вудворд провела дальнейшее деление на 11 типов производственных подсистем. “Цельной продукцией она называет такую продукцию, которая производится как отдельная единица — поодиночке, сериями или массовым производством”. “Объемная продукция измеряется с помощью веса или объема, к ней относятся химикаты, жидкости и газы”.

В своей книге Вудворт комментирует различные виды процессов, сводка которых дана на рис. 6.8.

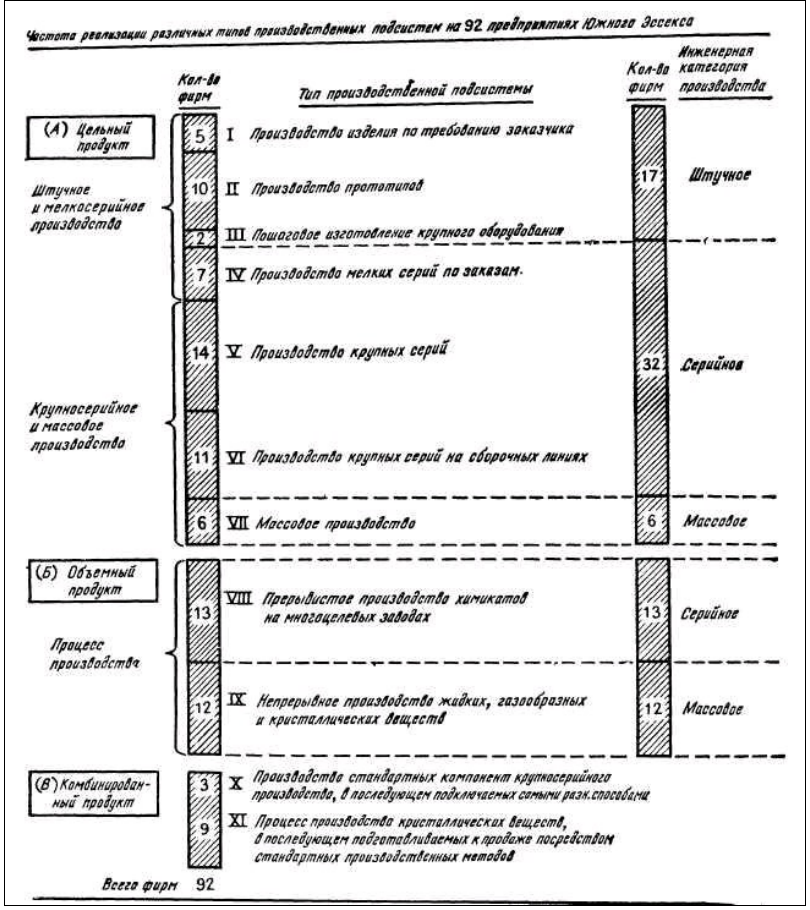


Рис. 6.8. Производственные системы. (Joan Woodward, Industrial Organization Theory and Practice (London. Oxford University Press, 1965.) Печатается с разрешения.)

С продвижением по шкале от систем типа I к системам типа XI резко возрастает возможность развивать контроль над производственными операциями, физические ограничения па продукцию становятся более изученными и понятными... Однако наиболее хорошо разработанные процедуры контроля за продукцией можно найти в фирмах, выпускающих серийную продукцию, где имеется некоторая степень сомнения в предсказанных результатах. Производству предшествуют энергичные и продолжительные попытки обойти физические ограничения путем постановки более сложных целей. Трудности изучения эффективности контроля становятся наибольшими в штучном производстве, особенно на стадии производства прототипа. Предсказать результаты работ по разработке ни в терминах временных затрат, ни в терминах денежных вложений не удается никогда *{Woodward Joan, Industrial Organization Theory and Practice (London. Oxford University Press, 1965).}*.

Я хочу пояснить, что же это все означает для меня. Вудворд говорит, что наиболее сложным из всех производственных процессов с точки зрения управления является процесс штучного производства. Программа — это “штучная продукция”, она характеризует процесс производства как трудноконтролируемый, а затем добавляет, что “предсказать результаты работ по разработке ни в терминах временных затрат, ни в терминах денежных вложений не удается никогда”. Насколько же возрастает верность всего сказанного по отношению к разработке программ, которые производятся поштучно и которые по сути являются чем-то *неосязаемым.* Разработка, по определению, не может быть связана предварительными расчетами денежных затрат, графиков и результатов. Если вам кажется, что вам удалось все предусмотреть, значит вы занимались не разработкой, а каким-то другим делом, которое следует называть как-нибудь иначе в зависимости от того, что в действительности делалось.

“***Эффект заброшенных функций” при разработке больших программ***

Для выбора эволюционного подхода к разработке программного обеспечения (а следовательно, и системы) существует и другая, не принимаемая во внимание причина, кроме той, что в течение некоторого времени нам может недоставать знаний об исходных требованиях. Во всем мире можно найти лишь несколько методов или групп (если это вообще возможно), которые способны за один проход создавать те сложные системы, которые вводятся в действие в настоящее время. Эти системы слишком сложны и велики, чтобы их можно было разработать “за один проход”.

Здесь мы сталкиваемся с явлением под названием “заброшенные функции”. Необходимо выработать некоторый приблизительно определенный набор функций. По мере приближения срока сдачи работ руководитель разработки начинает понимать, что реализовать *все* обещанные функции в срок нельзя. Как воздушный шар, теряющий высоту, группа разработчиков избавляется от балласта “необязательных” функций. График “выдерживается”, работа завершается “успешно”, несмотря на то что в потайной комнате несколько расторопных людей поспешно подключаются к работе, которую должна была бы выполнять вычислительная машина. Теперь все заботы по подключению к системе этих заброшенных функций ложатся на плечи членов группы продолжающейся разработки. Поскольку фаза разработки, построенная по методу “большого взрыва” заканчивается, все оставшиеся недоработки маскируются под названием “сопровождение”. Число занятых в этой работе людей обычно значительно уменьшается; группа “сопровождения” по сути является группой разработки (см. рис. 6.9)

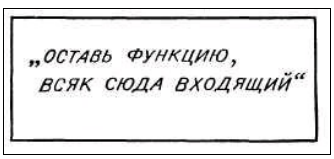


Рис. 6.9. Заброшенные функции. [Такую табличку надо было бы повесить на все обещанные функции.]

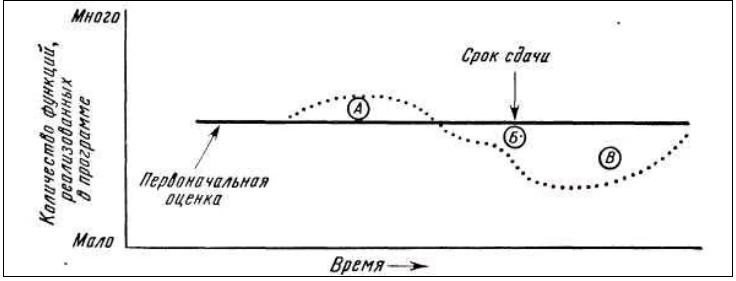


Рис. 6.10. Число функций по отношению к моменту сдачи системы.

Рис. 6.11. Миф об уменьшении занятости.

По ходу разработки предполагаемое число вводимых в строй функций изменяется. Поначалу чувство эйфории приводит к тому, что разработчики программного обеспечения обещают сделать даже больше функций (см. рис. 6.10). Реальность принимается во внимание только при приближении даты сдачи системы. График вступает в свои права, и функции начинают отбрасываться.

Проект объявляется успешно завершенным, хотя многие функции, которые должна была бы выполнять вычислительная машина, выполняются весьма способными людьми, вооруженными острыми карандашами и сидящими в потайных комнатах.

Функции, "заброшенные" в целях выполнения графика, продолжают реализовывать. (Рис. 6.10.) Этим занимается группа операций и сопровождения на собственные средства, поскольку никто не горит желанием признать реальное положение дел. Это нежелательно, но так случается в жизни.

"Привычка, — говорит Марк Твен, — состоит в том, чтобы выделить для каждой вещи свое место, а затем рассовать все по-другому".

"Мы восходим на небеса по руинам наших самых сокровенных надежд, обнаруживая в конце концов, что наши неудачи были на самом деле нашими победами". Эмос Элкотт.

Рис. 6.11 представляет собой иллюстрацию к мифу о количестве занятых в разработке крупной сложной программной системы людей. Эта диаграмма оказывается правильной только для небольших простейших программ. И все же ее продолжают выдавать за график распределения усилий при разработке программного обеспечения!

Планирование развития

Как же проектировать систему, учитывая ее возможное развитие, заранее зная, что требования к ней могут измениться? Надо руководствоваться по крайней мере девятью следующими принципами:

1, Мы должны проектировать наши программы так, что бы они обладали максимально возможной *модульностью.* Даже в том случае, если это приведет к росту выполняемой рабочей программы и затяжке сроков разработки.

2. Модули следует группировать так, чтобы взаимодействие между ними было минимальным.

3. Нам следует применять методику упрятывания информации даже в тех случаях, когда она приводит к увеличению размеров программы. Это даст нам возможность при внесении исправлений уменьшить количество изменяемых модулей. Упрятывание информации есть результат высокой связности.

4. Нам следует использовать табличные методы управления логикой работы программы. Изменяя строку таблицы, вы меняете алгоритм. Мы использовали этот метод в системе диспетчерского контроля авиалиний. Структура авиалиний определялась таблицами, что избавляло нас от необходимости изменять алгоритмы программ, работавших в разных центрах (всего был 21 центр). Таким образом, у нас была одна программа и 21 таблица для 21 центра.

5. Необходимо устанавливать стандарты программирования и настаивать на их выполнении. Стандарты не пользуются популярностью, но необходимы.

6. Следует организовать строгий и детальный контроль. Мы затронем этот вопрос в разделе, посвященном руководству проектом.

7. Нам нужно заранее планировать прикрепление основных членов группы к работам по разработке последующих вариантов и версий.

8. Для обеспечения сопровождения программ нужно за ранее планировать сохранение средств тестирования и разработки.

9. Все это необходимо предусмотреть при составлении бюджета.

Плачевные последствия недостатка капиталовложений, призванных способствовать развитию системы, не всегда сразу бросаются в глаза; иногда они совершенно замаскированы. Конечно, причиненные неприятности видны всем, но причины их не ясны.

Например, нам нужно добавить к некоторой продукции, предназначенной для продажи, новые функции, которые могут повысить ее конкурентоспособность. Все эти функции можно реализовать только программными средствами, и именно программными средствами их и надо реализовывать. По предварительной оценке, исправление программ можно провести за год. "Год!!" — следует реакция.— "Это безумие". Но меньше чем за год управиться не удается.

Причина заключается в том, что исправляемые программы по своей структуре напоминают бетонный блок! На начальном этапе создания программного обеспечения не было сделано ничего, что способствовало бы в дальнейшем облегчению модификации программ.

Но, что еще хуже, на начальном этапе проектирования игнорировалась необходимость делать программы "понимаемыми"! Это приводит к тому, что лишь небольшое число людей, часто работающих на самом нижнем уровне, понимают, что делается в конкретных программах,— и вот *им то* и приходится принимать *ответственные* решения.

Занятость

Число программистов не просто *сохраняется* на низком уровне, оно уменьшается, это вызвано тем, что при малейшей вашей невнимательности ваших программистов переманивают другие. Нехватка программистов одновременно и тяжела и страшна. Нехватка разработчиков программного обеспечения еще страшнее, причем положение с ними постоянно ухудшается. Типичный ход работ над проектом напоминает приведенную на рис. 6.12 диаграмму. Сплошная линия обозначает планируемое число занятых в проекте. Точечная линия обозначает фактическую занятость. Дефицит на стадии *А* восполняется дополнительными затратами на стадии *В*.

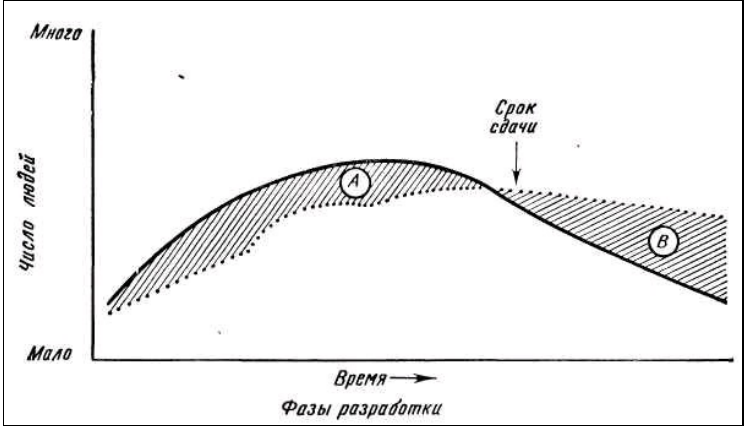


Рис. 6.12. Подключение людей.

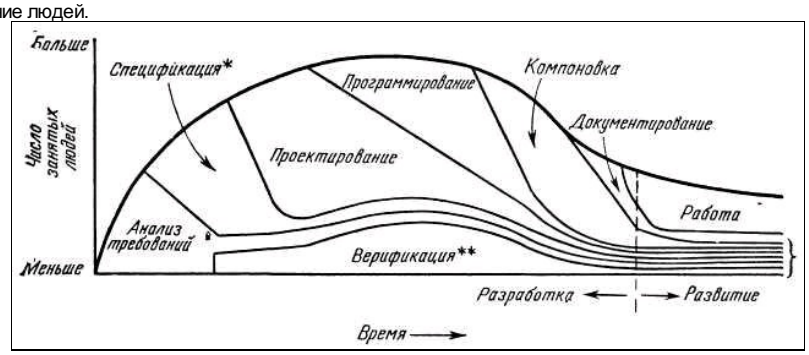


Рис. 6.13. Распределение людских ресурсов при разработке программного обеспечения.

Разработчики *постоянно* утверждают, что дефицит, возникающий на стадии А, будет преодолен “уже в следующем месяце”, но сделать этого не удается почти никогда.

Как мы уже видели, более точная модель того, что происходит в крупных проектах, соответствует диаграмме на рис. 6.13.

Эволюционный подход к разработке больших систем

Из всего вышеизложенного можно сделать вывод, что *необходимо* планировать *развитие* разработки и построения такой системы, которая сможет с течением времени претерпевать изменения, однако об этом часто забывают.

А ведь если этого не делать, нам придется латать и связывать систему из кусков, наша система получится хрупкой, и пользователи могут начать ее игнорировать.

Первое, что нужно сделать при разработке системы, которая будет подвержена длительному развитию, *это позаботиться об удержании на месте группы разработчиков*.

Вероятно, именно в этом заключается наиболее значительное отличие систем запуска человека в космос (NASA) и диспетчерского контроля авиалиний (FAA) от огромного множества других больших систем. И NASA, и FAA планировали и финансировали сохранность группы разработки на период *до 10 лет!* Обе организации понимали, что их системы будут развиваться в течение столь длительного времени.

Диаграмма занятости, приведенная на рис. 5.6, не подходит для систем такого рода; в больших системах после их сдачи *не наблюдается* падения численности занятых людей, разработчики должны оставаться на месте для выпуска следующих вариантов и версий.

Эта идея о необходимости “версий” была подана мне в 1970 г., когда в комитете Конгресса готовились к публичному слушанию дела о разработке системы диспетчерского контроля за авиаперевозками. Один из экспертов конгресса, изучавший вместе со мной материалы, пришел к выводу, что FAA и IBM будут подвержены критике, поскольку “тот факт, что выпущено семь версий программы, *доказывает,* что группа не знала, что нужно делать”. Я сказал ему, что подготовлюсь к ответу на это обвинение за несколько дней.

Случилось так, что буквально на следующий день после этого я встретился со своей хьюстонской группой и узнал, что у них была “версия 14.7” — т. е. всего, если отвлечься от непонятных обозначений с десятичными дробями, было по крайней мере четырнадцать версий,— и все-таки человек высадился на Луну!

“Зачем так много версий?” — спросил я. Мне привели две причины. Во-первых, сотни операторов управляющих пультов, взаимодействующих с вычислительной машиной, *не в состоянии воспринимать* изменения в операторских процедурах большими дозами — им надо подавать их по частям. *Кроме того,* даже сейчас никто не может предвидеть, что захотят в дальнейшем пользователи и что будет необходимо добавить в систему управления.

Все это я передал эксперту Конгресса. Мне показалось, он воспринял смысл всего этого. Но либо он не разговаривал с членом Конгресса, либо конгрессмен не захотел его выслушать, при публичных слушаниях нас подвергли суровой критике за множество исправлений системы, число версий которой показывало, что мы не знали, что делаем!

Что мы имеем — или должны иметь — (с точки зрения программного обеспечения) в первой версии программной системы, состоящей из двух весьма различающихся множеств программ, которые будут выполняться одновременно:

1. Множество системных программ, которые будут составлять график выполнения прикладных программ на машине и управлять внешним окружением.

2. “Начальное множество” прикладных программ, с которыми пользователь может начать работу со своей системой и а) извлекать из нее пользу и б) находить новые и более хорошие способы работы, которые можно будет добавлять в последующие версии или варианты программ. Такой процесс подключения новых функций продолжается в течение всего периода жизни системы.

Почему от такого подхода частенько отказываются? Имеются по крайней мере три причины, которые мешают принять этот эволюционный подход.

Во-первых, он, по-видимому, дороже стоит. Введение в системные программы такой инфраструктуры, которая позволяет им легко воспринимать новые функции прикладных программ, стоит денег, а все преимущества этой инфраструктуры становятся видны только на фазе сопровождения программ, да и тогда они видны только посвященным. *Показать* эти преимущества нельзя никоим образом, и руководство может только смутно ощущать то, что ему говорят непосредственные технические исполнители. Лишь подлинно мудрый руководитель не побоится затрат на все это. Построение гибкой системы приводит к повышению затрат при разработке; однако общая стоимость жизненного цикла снижается.

Во-вторых, такая инфраструктура программ требует дополнительных затрат машинных ресурсов в фазе использования; необходимы и дополнительная память, и время процессора. Оба этих ресурса часто оказываются дефицитными.

В-третьих, задача проектирования такой инфраструктуры не относится к легким, требующим лишь технических усилий. Для проектирования такой гибкости структуры нужны крайне талантливые люди.

Задачи руководства программным обеспечением проекта

Руководство разработкой программного обеспечения весьма непростое дело. Нужно решать и управлять решением огромного количества мелких, но и важных задач. Ниже следует список, представляющий собой оглавление “Военного стандарта ВМФ США 1679” — разработку программного обеспечения систем вооружения. Все основные пункты мы уже рассмотрели, но и более мелкие могут играть важную роль и сейчас, и в дальнейшем. Этот список прекрасно иллюстрирует трудности задачи разработки:

Общие требования

- Руководство разработкой программного обеспечения
- Требования к проектированию
- Формирование программ
- Гарантия качества
- Руководство конфигурацией
- Управление подрядными работами
- Отклонения и отказы от требований

Подробные требования

- Требования к производительности программ
- Вспомогательная информация для требований о производительности программ
- Анализ производительности программ для вычислительных машин
- Области применения
- Функции
- Документация, необходимая для требований по производительности программ
- Описание системы вооружения
- Функциональное описание
- Подробные функциональные требования
- Регулируемые параметры
- Системные ресурсы
- Требования к проектированию программ
- Вспомогательная информация для требований к проектированию программ
- Анализ проекта программ для вычислительных машин
- Документация, необходимая для требований к проектированию программ
- Распределение функций
- Функциональная схема программы
- Распределение ресурсов и резервы
- Проектные ограничения
- Проектирование базы данных
- Межсистемные взаимодействия
- Стандарты программирования
- Управляющие структуры
- Вставляемые/копируемые сегменты
- Структура входов-выходов
- Отслеживание связей в программах
- Самомодифицируемость
- Рекурсивные программы
- Размер
- Ветвления
- Перемещаемость
- Формат текста программ
- Соглашения, принятые при программировании
- Символическая параметризация
- Система именования
- Численные соглашения
- Символические константы и переменные
- Выражения из разнотипных операндов
- Группирование
- Значащие цифры
- Структурированные словесные описания
- Резюме
- Комментарии и применения в программах
- Формат входных записей
- Эффективность выполнения
- Включения/копирования сегментов на исходном языке
- Операторы входного языка
- Блок-схемы
- Производство программ
- Организация производства программ
- Руководство ресурсами
- Язык
- Использование библиотек и управление ими
- Последовательная нумерация
- Распечатки
- Распечатки программ
- Распечатки перекрестных ссылок
- Карты загрузки
- Регенерация программ
- Выполнение программ
- Анализ выполнения программ
- Нефункциональное выполнение
- Функциональное выполнение
- Тесты программ
- Тесты модулей
- Тесты подпрограмм
- Тесты производительности программ
- Комплексный тест систем (ы)
- Отчетность об ошибках в программном обеспечении
- Категория отчетов об ошибках в программном обеспечении
- Приоритет отчетов об ошибках в программном обеспечении
- Рассылка отчетов об ошибках в программном обеспечении
- Гарантия качества
- Обеспечение гарантируемого качества
- Уровни отчетности
- Участие в обсуждениях
- Пересмотр планов
- Проектирование программ
- Кодирование программ
- Тесты
- Представляемые элементы

Отчетность
Авторство
Приемлемость программ
Дополнительные требования к приемлемости программ
Требования к тестам качества программного обеспечения для проверки приемлемости программ
Окружение тестирования
Тестируемое программное обеспечение
Документация тестов качества программного обеспечения
Выполнение тестов качества программного обеспечения
Продолжительность тестирования качества программного обеспечения
Входные данные для тестов качества программного обеспечения
Тестирование качества тестирования качества программного обеспечения
Возможность сокращенного тестирования качества программного обеспечения
Тесты качества программного обеспечения и вспомогательные программы сопровождения
Ошибки во время тестирования
Ограничения на тесты качества программного обеспечения
Ограничения из-за ошибок
Временные ограничения
Руководство конфигурацией
Идентификация конфигурации
Основные варианты
Определение документации
Управление конфигурацией
Изменения в программном обеспечении
Изменения в документации
Панели управления конфигурацией программного обеспечения
Вычисление статуса конфигурации
Руководящий контроль
Организация руководства
Требования к ресурсам
Обзоры положения дел
Предметы обзоров положения дел
Пункты, по которым проводятся обзоры положения дел
Обзоры документации
Специальные обзоры
Инспекции и прослушивания

Этот список ни в коей мере нельзя считать полным. Любой список, составленный более года назад, можно считать устаревшим. Однако список этот оказывается полезным и для демонстрации широты приложения усилий на узком участке, и для проверки, не проглядели ли мы чего-нибудь.

Результаты процесса разработки

Что же мы получаем в результате процесса разработки программного обеспечения? Основные результаты таковы:

1. Конечно же, *программы,* которые будут выполняться, ну а что же еще?

2. *Руководства для пользователей.* Инструкции и описания, которые сделают систему понятной для пользователя и помогут работать с ней.

3. *Материалы для сопровождения программ.* Материалы, необходимые для продолжающейся разработки, не отличаются от тех, что требуются для обеспечения первичной разработки программ. Планы тестирования, результаты тестов, спецификации и все остальное, что нами использовалось, а также хорошая документация.

Все, что изображено на рис. 6.14, является результатом процесса разработки программного обеспечения. Всех подробностей, однако, эта схема не содержит. Например, в качестве подмножества к проекту системы мы могли бы указать список интерпретирующих и моделирующих программ, которые пишутся для того, чтобы создаваемый нами проект программы оказался успешно завершенным. Эти программы, весьма ценные для групп сопровождения, также являются результатом усилий по разработке программного обеспечения.

План разработки или проекта

Любой важный проект должен разрабатываться по плану. Сегодня такие планы стараются делать так, чтобы их можно было обрабатывать на вычислительных машинах и хранить в памяти инструментальной машины.

План разработки содержит огромное множество разных пунктов, расписанных в мельчайших подробностях. Нужно быть постоянно в курсе их выполнения, чтобы этим планом можно было руководить. План проекта или разработки содержит:

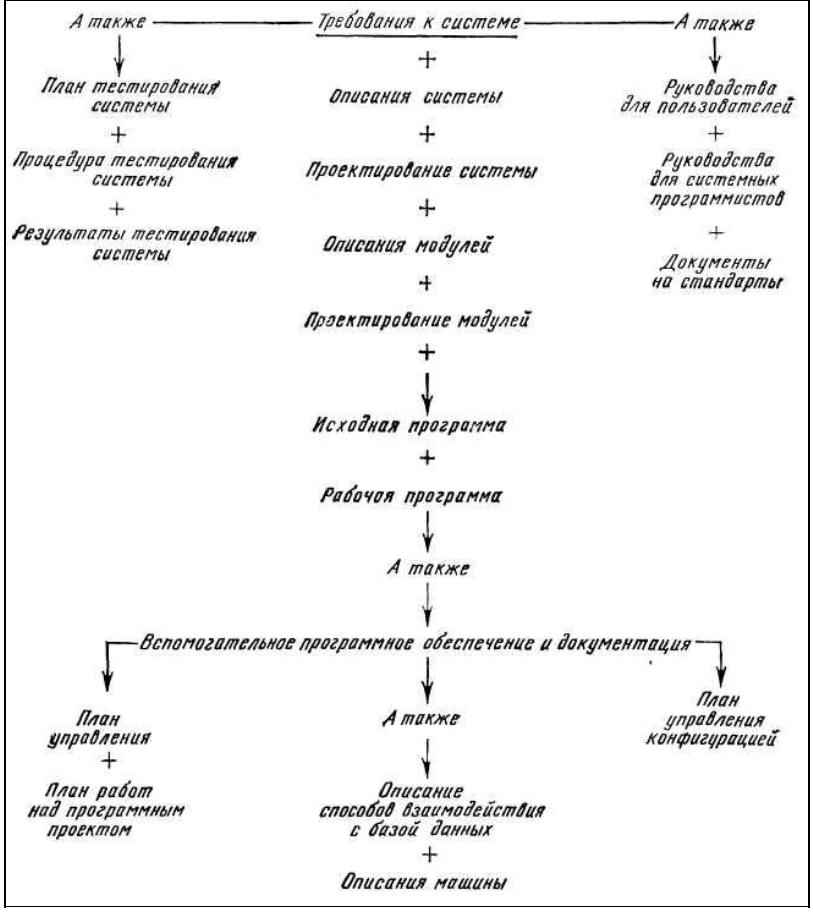


Рис. 6.14. Результаты процесса разработки программного обеспечения.

1) спецификации требований;
2) рабочую схему организационной структуры с указанием сроков реализации;
3) схему расстановки кадров;
4) бюджет;

5) документирование рабочих стандартов. Руководитель разработки поочередно обращается то к плану, то к результатам его воплощения, модифицирует план и опять начинает этот цикл сначала. Рис. 6.15 выглядит просто, но следовать предложенной на нем схеме очень трудно. Самое важное место на схеме отводится пересмотру графика, бюджета или функций (см. маленький прямоугольник слева). Когда дела идут скверно, чем-то приходится поступиться — и этим должно быть что-то из этой тройки. Руководство обычно медленно соглашается с изменениями плана

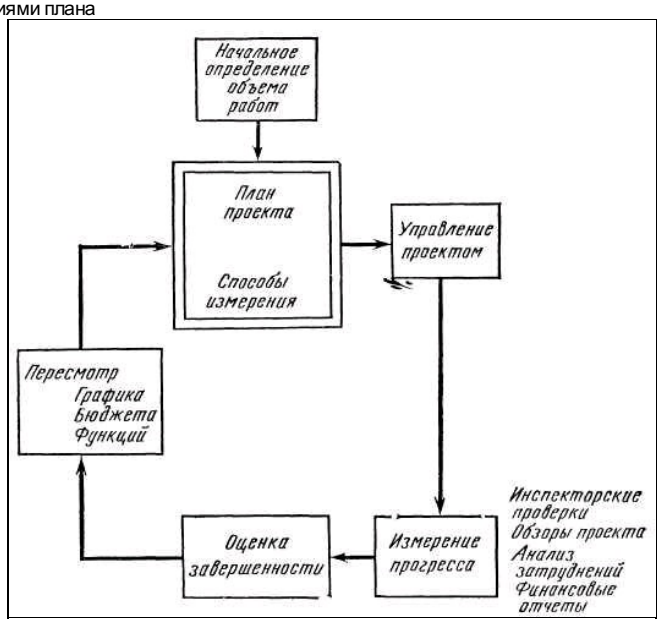


Рис 6.15. Цикл планирования и управления.

Заметьте, что входная стрелка к прямоугольнику, обозначающему план проекта, ведет от начального определения объема работ или предварительных оценок. Обратимся же теперь к изучению способов оценок и тому, что им предшествует — производительности труда.

Производительность труда и оценки

Производительность труда — это скорость производства окончательных программ, вполне готовых к работе. Определяется она как отношение объема работ, выполненных при разработке, к продолжительности этих работ. Производительность труда обычно измеряется в строках программы, отнесенных к человеко-месяцу (или человеко-году). Ниже мы рассмотрим некоторые проблемы, связанные с введением такого способа измерения.

Оценка состоит из двух частей. Во-первых, нам нужна оценить размеры того, что нам предстоит построить, а во-вторых, мы должны оценить производительность труда, которой мы должны достигнуть во временном и денежном выражениях и в терминах занятости персонала. Для общей оценки второй части нам необходимо обладать некоторым понятием о максимальной и средней скоростях реализации того типа и в той области деятельности, которой нам предстоит заняться, т.е. некоторыми оценками производительности труда. Поэтому сначала мы обратимся к изучению производительности труда, а затем перейдем к изучению оценок.

Производительность труда при разработке программного обеспечения

Нам не известно, как надо измерять производительность труда при программировании или разработке программного обеспечения. Мы только начинаем разрабатывать терминологию и средства измерения. Многого мы еще *не* понимаем. Для описания трудности работы мы используем такие термины, как “тяжелая” или “очень тяжелая”, “большая” или “огромная”. От таких терминов особой помощи ждать не приходится.

Более полезным оказывается способ измерения, связанный с подсчетом числа команд или строк текста программы (СТП). Используя это понятие, мы, например, говорим, что данная работа или функция — скажем, составление платежной ведомости — требует 50 тыс. строк программы.

Это единственный способ измерения, который реально используется в настоящее время. Но и в нем есть изъяны. В этом случае поощряются плохие разработчики, которые для реализации функции пишут очень много команд.

	Программист А	Программист В
Время	2 мес	1 мес
Строки текста	2000	900
Производительность труда	1000 СТП/чел.-мес	900 СТП/чел.-Мес

Наше измерение в строках текста за единицу времени приводит к поощрению неэффективного проектирования или небрежного программирования. Программист А работает по плохому проекту и имеет “рыхлую” программу и поэтому за единицу времени смог написать больше строк текста программы.

База данных по производительности труда

В этой области можно собрать весьма интересные статистические данные. Те, кто утверждает, что знаком со средними цифрами производительности, ошибаются; для определения средних величин производительности труда еще не собрано достаточно данных.

Строки текста можно уподобить числу мазков, которые наносит маляр при покраске дома. То, что один маляр нанес меньше мазков, чем второй, не означает ровным счетом ничего до тех пор, пока мы не сравним результаты работы обоих.

Когда я захотел представить группу, добившуюся колоссальной производительности труда при работе над проектом для газеты “Нью-Йорк тайме”, к премии за выдающиеся достижения, я обнаружил, что во всей фирме IBM не имелось данных о производительности труда.

Мы решили, что начнем измерять строки программ. У нас уже были, в наших отчетах по контракту, записи о часах работы каждого члена группы, времени, использованном на каждую работу, о категориях всех наших сотрудников (руководитель, аналитик, программист и т. д.) и об их заработной плате. У нас были сведения обо всем — так мы думали. И почти сразу же мы обнаружили, что никто из нас не имел точного определения “строки программы”! А также *и* “человеко-месяца”!

Определения терминов “строка программы” и “человеко-месяц”

По поводу того, что следует называть строкой программы, имеется большая путаница.

Два значения термина “строка программы”

Полезно знать число строк в рабочей программе, которое помогает оценить, сколько памяти нужно для запоминания команд *рабочей программы*.

Полезно знать или иметь оценки числа строк исходной программы, которые помогают измерять и оценивать производительность труда, или число программистов, необходимых для выполнения данной работы. Исходные программы обычно пишутся на языках высокого уровня.

И, хотя строка программы, написанной на языке высокого уровня, обычно называется оператором, мы редко используем число операторов, написанных за день, как меру производительности труда, в качестве стандартной меры мы предпочитаем строку программы. Обе меры полезны — и число исходных строк, и число строк рабочей программы, но следует быть осторожными и по небрежности не путать их между собой. Каждая из них полезна для своих целей.

Итак, термин “строка программы” имеет два значения, причем полезными оказываются оба значения:

— Как мера производительности труда программистов при разработке — *строка текста на языке высокого уровня, строка исходной программы*.

— Как мера *рабочей программы,* которая используется в реальной работе. Сколько требуется памяти, сколько усилий нужно затратить на сопровождение этих строк программы на машинном языке.

Даже получив эти определения, нельзя выйти из тупика.

Определение строки текста исходной программы. Строка текста программы — это любая входная запись длиной до 80 символов (выполняемая и невыполняемая), написанная программистом или используемая в нашей системе программного обеспечения во время его работы. Имеется по крайней мере два типа таких операторов. К одному типу относятся операторы, не предназначенные для переноса в рабочую программу, к которым относятся операторы, написанные, отлаженные, использованные и исключенные (возможные примеры исключаемых операторов: некоторые программы имитации, некоторые моделирующие программы, некоторые тестовые программы), не выполняемые во время использования программы. К этому же типу относятся примечания и описания данных. К другому типу относятся операторы, из которых формируется рабочая программа и которые используются в момент работы программы.

Если мы не будем включать в наши подсчеты строки с примечаниями, программисты, пытаясь повысить свою продуктивность, перестанут писать примечания к своим программам. Это будет неправильно; примечания нужны нам для того, чтобы облегчить проведение работ в фазе сопровождения проекта. С другой стороны, мы не хотим, чтобы программисты писали по две страницы примечаний к программам из 30 операторов. Для написания примечаний должны быть установлены некоторые нормы. Но включать примечания в подсчеты числа строк нужно.

Мы должны учитывать *все* строки программы, операторы, написанные или использованные нашими разработчиками, независимо от того, работают они на фазе использования или нет. Все они являются неотъемлемыми частями проекта. Мы также должны учитывать число строк рабочей программы, что может помочь нам понять работу нашей системы во время ее использования.

Определение строки рабочей программы. Строка рабочей программы представляет собой одну команду на машинном языке, которая либо непосредственно написана на этом языке, либо получена с помощью процесса трансляции с какого-либо другого языка, либо “введена” в нашу систему из другого проекта. Нужно отслеживать полное число всех этих команд.

Категории программного обеспечения

Необходимо различать некоторые типы программного обеспечения, разработка которого проводится нами:

1) операционное обеспечение — как системное, так и прикладное, лишь бы оно использовалось во время работы системы;

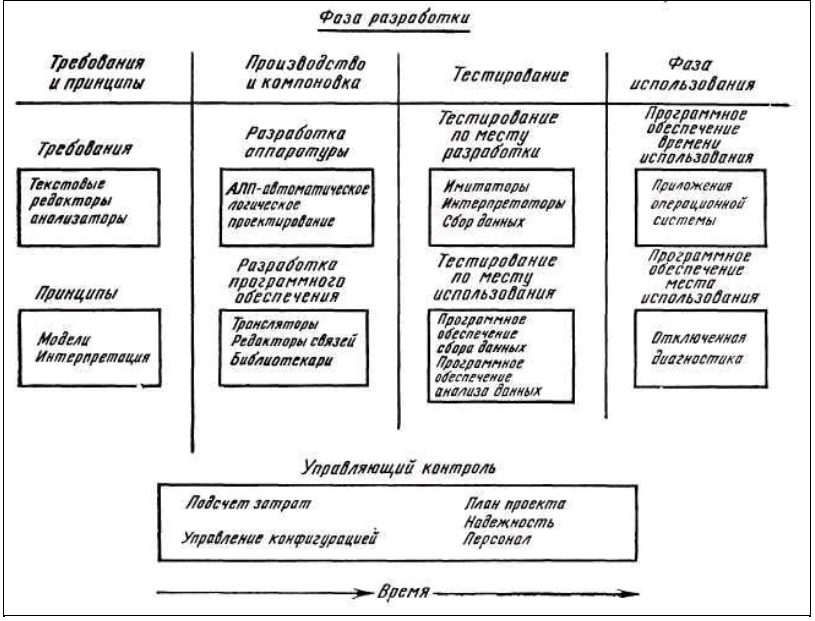


Рис. 6.16. Программное обеспечение используемое/создаваемое на различных стадиях фазы разработки/использования.

2) инструментальное обеспечение — программное обеспечение разработки; программное обеспечение тестирования.

Если в системе используется несколько вычислительных машин, программное обеспечение для каждой из них, если оно у них разное, необходимо отслеживать отдельно. Мы должны отдельно учитывать СТП, написанные для:

1) программного обеспечения фазы использования или операционного программного обеспечения;

2) сопровождения разработки операционного программного обеспечения, включая и “выброшенные программы”, например управляющие программы;

3) сопровождения разработки аппаратуры, включаемой в систему;

4) проведения тестирования системы;

5) сопровождения проектирования, например моделирующие программы;

6) руководящего контроля, например PERT, подсчета стоимости.

В последних категориях следует ожидать намного большего числа строк программ, чем в первой. Диаграмма, иллюстрирующая эти различия, приведена на рис. 6.16.

Определение человеко-месяца

В подсчеты с помощью человеко-месяцев (ЧМ) включаются все участники, поддерживающие разработку программного обеспечения от начала работы над проектом и до выпуска в свет готовой системы. Работники, проводившие анализ требований, проектировщики, секретари, руководители, библиотекари, охрана — все они должны учитываться (пропорционально вкладу в создание программного обеспечения) при подсчетах в человеко-месяцах. Некоторый процент от человеко-месяцев, отнесенных к инженерной части, должен быть подсчитан как вклад в программное обеспечение, точная доля определяется совместно руководством технической частью проекта и разработкой программного обеспечения. Необходимо учесть все часы, занятые работой над проектом, от самого начала до самого конца.

Изменчивость

Не следует думать, что отношение СТП/ЧМ остается постоянным для разных типов программного обеспечения. Создавать тексты для использования в фазе разработки "легче, чем для использования на стадии реальной работы. Управляющее, или системное, программное обеспечение (операционное) должно, видимо, быть более сложным, чем прикладное (операционное) обеспечение. Первые варианты могут оказываться более сложными, чем более поздние, так как постепенно приходит опыт, и группы программного обеспечения начинают более эффективно работать именно как труппы.

Вопросник по производительности труда

Обратимся теперь к данным, собранным нами в базе данных по производительности труда. Мы хотели выяснить, что же влияет на производительность труда, измеряемую в строках программы, а для этого нам нужно было собирать множество различных данных о проводимой работе. Нам было необходимо знать окружение, в котором работает заказчик. Есть ли у заказчика достаточный опыт для работы в данной прикладной области? Имеется ли у заказчика достаточный опыт в области обработки данных или хотя бы вообще по автоматизации? Участвовал ли заказчик в составлении проекта? В определении требований? В разработке программ? Было ли взаимодействие с заказчиком очень сложным или его можно отнести к средней сложности?

Мы хотели знать “цену проекта”, его оцениваемую стоимость. Не требовали ли от нас больше разных переездов, чем это вызывалось необходимостью? Находилась ли группа в непосредственной близости от заказчика? Было ли место расположения разработчиков “хорошим”? Или плохим? (В одной крупной разработке нам приходилось возить программистов по утрам за 50 миль, а по вечерам привозить,их обратно, для того чтобы они смогли получить доступ к машине, на которой проводилась разработка, и вступать в контакты с заказчиком.) Не было ли так, что машинное время выделялось только с полуночи до 4 ч утра?

Мы хотели знать долю программистов-разработчиков, участвовавших в составлении проекта. Нам хотелось знать максимальную скорость работы, точнее, насколько хороша была программистская группа — действительно хорошая, средняя или плохая. Была ли документация по требованиям скудной, средней или пространной? Был ли “стабильным” проект? Кто предлагал изменения — пользователи или разработчики? Какой использовался язык? Какими ресурсами обладала машина, на которой работала система? Какие вычислительные машины использовались во время разработки? Не велись ли параллельные работы по разработке аппаратуры вычислительной машины, на которой должна была действовать система? Затем нам нужно было узнать о методах, которые применялись руководством работ.

Был ли составлен план, использовался ли он для:

— оценки изменений в проекте системы и внесения этих изменений
— оценки изменений в проекте программного обеспечения: и их внесения
— тестирования (на всех уровнях)
— извещений об ошибках в программах, выявленных и исправленных
— использования вычислительных машин
— разработки вспомогательного вычислительного оборудования
— защиты важнейшей информации
— контроля денежных затрат
— руководящего контроля
— документирования
— стандартизации методов кодирования

Не использовались ли необычные устройства ввода/вывода или средства отображения? Не было ли это оборудование использовано каким-либо необычным образом? Работала ли программистская группа на данной машине прежде? На этом же языке? С приложениями таких же размеров и сложности? Работали ли ведущие программисты прежде того в совместных проектах? Каково было окружение разработки? Работа велась с выделением пультового времени? Или в режиме разделения времени? Использовались ли удаленные терминалы? Привлекались ли программисты к непосредственной работе на машине? Каким было время ожидания решения? Применялся ли диалоговый режим? Как долго? Менее 2 ч, от 2 до 7 ч, от 8 до 24 ч, более 24 ч? Какая доля программ разрабатывалась с

использованием:

— библиотеки инструментальных программ,
— программного библиотекаря,
— структурного программирования,
— сквозного контроля,
— программирования сверху вниз,
— метода главного программиста?

После этого мы приступили к оценкам сложности
— приложений,
— алгоритма программ,
— межпрограммных связей,
— внешних связей,
— структуры базы данных.

Какая доля программ пришлась на
— нематематические приложения и форматирование ввода/вывода,
— математические и вычислительные программы,
— программы управления центральным процессором и
— вводом/выводом,
— программы возвратов и восстановления после сбоев?

Какие ограничения повлияли на проект:
— в оперативной памяти,
— временные,
— возможностей ввода/вывода?

Проводилась ли классификация программ? Из скольких модулей они состояли? Сколько классов было в базе данных? Сколько страниц в документации? Сколько в рабочей? А сколько в сопроводительной? Сколько памяти занимает операционная система при выполнении программ? Нам нужно было составить хронологическую таблицу обнаружения ошибок с данными по разным категориям. Нам, конечно же, хотелось иметь сведения о численности персонала, занятого в

— управлении,
— администрации,
— программировании,
— анализе,
— технических работах.

Нам был необходим список вычислительных ресурсов. Нам нужно было знать, какие усилия были потрачены на интерпретацию и моделирование.

Очевидно, что сбор всех этих данных еще больше усложнил и без того сложный процесс управления разработкой программного обеспечения. Часто мои подчиненные говорили мне, что у меня есть два варианта — я могу составить график или провести исследование и получить данные, нужные для работы. Еще чаще никаких данных получить я не мог. Но, хотя и медленно, с годами эти сведения приходили. Приводя список большинства этих вопросов, я хотел показать, что существует так много переменных и разных мнений, что указание любого одного элемента производительности труда в качестве доминирующего будет сверхупрощенным. Прежде чем мы сможем использовать нашу базу данных как средство управления при предсказании производительности труда, нам необходимо собрать огромное множество сведений о разработке программного обеспечения для сотен и сотен разных проектов.

Ошибки при подсчете СТП

Наиболее широко распространенная ошибка, возникающая в связи с подсчетами числа строк программ, заключается в игнорировании творческой стороны дела и придании особой важности подсчетам и вознаграждению людей за показатели в строках программы за человеко-месяц! Многие руководители загипнотизированы этими подробностями и забывают об изобретательской и творческой активности.

Многие люди, измеряя человеко-месяцы, ошибочно подсчитывают только время, затраченное программистами. Мы уже видели, что в очень больших проектах более 50% всех усилий предпринимается группами сопровождения и управления. Это тоже прямые затраты, их также необходимо учитывать.

Большой ошибкой является подсчет числа СТП за человеко-месяц уже в середине разработки. Если вы оценили полное число строк программы и среднее число строк за человеко-месяц и “заключили соглашение” на выполнение работ, вы должны постараться не впасть в ошибку “измерений в середине”. Не следует считать уровень производительности труда, достигнутый на каком-то этапе, постоянным и использовать его для расчета производительности труда на последующих этапах.

Как-то у нас был контракт на западном берегу, в котором мы недооценили объем программ примерно вдвое. Мы сменили руководителя — и новый не хотел отвечать за чужие грехи. Спорить с этим было трудно.

Этот новый руководитель утверждал, что наши цифры по производительности труда также были неверными. Он взял производительность в момент времени *x* (см. рис. 6.17) и заявил, что он попытается достигнуть именно *такой* же производительности для всего проекта в целом.

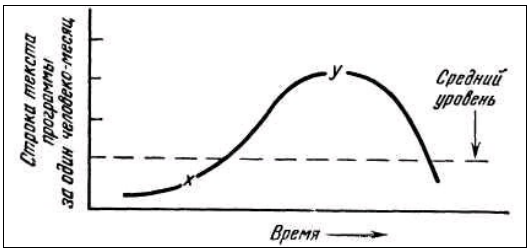


Рис. 6.17. Изменения отношения строк текста программы к человеко-месяцам во время работы над программным обеспечением для проекта

Это так же неверно, как и использование скорости работы в точке *(y)* в качестве уровня, который будет достигнут начиная с самого первого дня. Мы должны пользоваться *средним* значением.

Выбирать уровень в какой-то определенный момент времени — значит, впадать в серьезную ошибку! Единственно, когда можно измерить число строк программ, написанных для проекта, это в конце, после завершения всех работ. Не существует никаких цифр, на которые можно было бы вполне положиться вплоть до самого конца. Имеют смысл только полные подсчеты, а не промежуточные.

В самом деле, в проекте для Западного берега мы предсказывали, что первые написанные программы (системное программное обеспечение) отнимут гораздо больше времени и сил в расчете на одну СТП, чем последние. Нам возражали, заявляя, что уровень, достигнутый на ранней точке *x,* будет оставаться на постоянном уровне з течение всей работы.

Уровень производительности труда при создании программного обеспечения, достигнутый в некоторый момент работы, не следует использовать как постоянную для расчета будущей производительности. Производство строк программы не остается постоянным. Оно начинается достаточно медленно (с нуля), резко поднимается, а затем снова спадает до нуля.

Закон больших чисел. Из приведенного списка вопросов становится совершенно ясно, что число переменных, играющих роль в их решении, очень велико. Из этого следует, что реальное изучение информации, собранной в базе данных, не может быть проведено очень быстро. Когда я ушел из фирмы IBM в 1977 году, эти данные были еще слишком отрывочными. Во избежание вводящих в заблуждение статей и выводов, я придал данным статус “собственности компании”. Из этого было лишь одно исключение.

Неверная интерпретация содержимого базы данных. В начале 1977 года я допустил публикацию одной статьи в журнале IBM System Journal, написанную Уолстоном и Феликсом с условием, что будет опубликована только часть данных и что будет ясно указано, что приводимый анализ является предварительным. И тем не менее статью часто цитируют, как будто это окончательный отчет, и это цитирование не всегда верно.

В статье утверждалось следующее: *{Walston C.E. and Felix C.P. A Method of Programming Measurement and Estimation, IBM*

Sym. J, 16, № 1 (1977), 54—73.}

- *Данные были собраны по шестидесяти завершенным разработкам программного обеспечения. Использовалось двадцать восемь разных языков программирования. 28 языков для 66 различных вычислительных машин. Размеры программ изменялись в диапазоне от 4000 строк исходного текста до 467 000.*
- *Рабочие программы в статье не рассматриваются; сведения касаются только исходных программ. В статье приводится список 29 переменных из 69 представленных в отчетах, отбирались переменные, в наибольшей степени влияющие на производительность труда. В статье полностью игнорировались взаимодействия между этими 29 переменными. Там говорилось, что "анализ переменных проводился независимо друг от друга, возможность корреляции переменных и эффекты этого взаимодействия во внимание не принимались". Таблица 3, приведенная в статье, имела такую первую строчку:*

	Медиана 50%	Квартили 25—75%
Строки исходной программы за один человеко-месяц	274	150 — 440

Теперь мне хочется дать свои комментарии по поводу этой статьи. Рассмотрено менее одного завершенного проекта в расчете на одну вычислительную машину. Каждый из рассмотренных языков применялся в 2.1 проекте. Различия в масштабах были даже большими, чем от 4000 до 467 000 строк. Космические проекты Хьюстона были разделены на несколько "программ", каждая из которых имела отдельный отчет. Но работать по отдельности они не могли.

В одной таблице были сведены *средние* (медианные) значения отношения числа строк к человеко-месяцам; таблица, показывавшая 29 наиболее важных переменных, содержала именно средние значения. В табл. 3, приведенной выше, отбрасывались верхние и нижние 25 % значений. Все это делало данные гораздо более регулярно выглядящими, чем это есть на самом деле. Цифры, собранные на стр.291 в графе оценок производительности, показывают невероятно широкий диапазон производительности, это осталось для нас непонятным.

В статье не рассматриваются рабочие программы. Но именно они являются результатом работы, а не исходные программы. В базе данных есть сведения по рабочим программам. Игнорировать рабочие программы, значит также игнорировать коэффициент повышения производительности, вносимый языками высокого уровня. Язык программирования не вошел в 29 наиболее важных факторов. Но так дело обстоит только в статье, а не в реальной жизни. Кстати говоря, среди того, что особенно поразило нас после того, как мы впервые просмотрели базу данных, была доля программных проектов, в которых все еще использовались ассемблеры. Таких было более 50%.

Многие из рассматривавшихся 60 проектов были повторными реализациями, проводившимися теми же группами.

В статье говорится, что никакие взаимовлияния переменных не рассматриваются. Но ведь очевидно, что такое взаимодействие существует!

В статье предполагается, если не прямо утверждается, что фактором, влияющим на производительность в наибольшей степени, является "сложность взаимодействия с заказчиком". Таблица содержит строку с таким заголовком и некоторые числа; никаких комментариев в тексте нет. Известно, что учитывались отчеты, поступившие с разных концов света. База данных составлялась в Гейтсбурге, шт. Мэриленд,— крупнейшие разработки проводились в Хьюстоне, шт. Техас; Атлантик-Сити, шт. Нью-Джерси; Лос-Анджелесе, шт. Калифорния; а также в Моррис-Плейне, шт. Нью-Джерси. Мы решили, что оценить фактор взаимодействия с заказчиком иначе, как заявив, что он выше "нормальной сложности", невозможно. Нам не удавалось сравнить мнения руководителей из Техаса с мнением руководства в Лос-Анджелесе или в Сайгоне, или в Токио. Их мнения были чересчур личными.

Я не верю в то, что сложность взаимодействия с заказчиком была или остается наиболее важным фактором, воздействующим на производительность. Слишком недостаточны данные, которыми мы располагаем, чтобы можно было делать такой вывод.

Я видел три документа — книгу, авторский экземпляр и статью, в которых данная статья цитировалась неверно! В этих документах утверждалось, что "фирма IBM говорит то-то и то-то". IBM никогда этого не говорила.

И все же собранная в IBM база данных стала наиболее ценным источником. Фирме следует продолжить публикацию дальнейших открытий, не взирая на риск быть неправильно понятой.

Я разрешил эту публикацию в 1977 году; я отвечаю за большую часть всех недоразумений, о которых я уже рассказал.

Сейчас можно сказать, что публикация статьи была ошибочной. Подразумевалось, что это будет промежуточный отчет о *незавершенных,* но весьма значимых исследованиях. Она неправильно интерпретируется, но может быть более хладнокровные люди сумеют извлечь ценные сведения из этих данных, *полностью отдавая себе отчет* в том, что сведения имеют предварительный характер.

Хорошей базы данных, из которой можно было бы извлекать достоверные сведения, в настоящее время не существует. Данные, имеющиеся в нашем распоряжении, поразительнейшим образом отличаются по уровню достигнутой производительности труда, а почему это так, мы не понимаем. Уровни производительности труда в рамках одной и той же работы могут доходить от 1000 СТП за человеко-месяц до 60 СТП за человеко-месяц. Мы до сих пор не понимаем причины таких расхождений.

Проведение подобных замеров обходится дорого, но если мы надеемся полнее разобраться в динамике процесса производства программного обеспечения, мы должны продолжать сборы такого рода информации. До тех пор, пока не будет собрано достаточно большое количество данных со всеми необходимыми подробностями, мы не можем считать, что полностью овладели процессом разработки программного обеспечения.

До тех пор, пока этого не сделано, идея использования "баз данных" истории проектов для предсказания денежных и других затрат остается совершенно наивной. Она совершенно бессмысленна; воздействующих факторов слишком много.

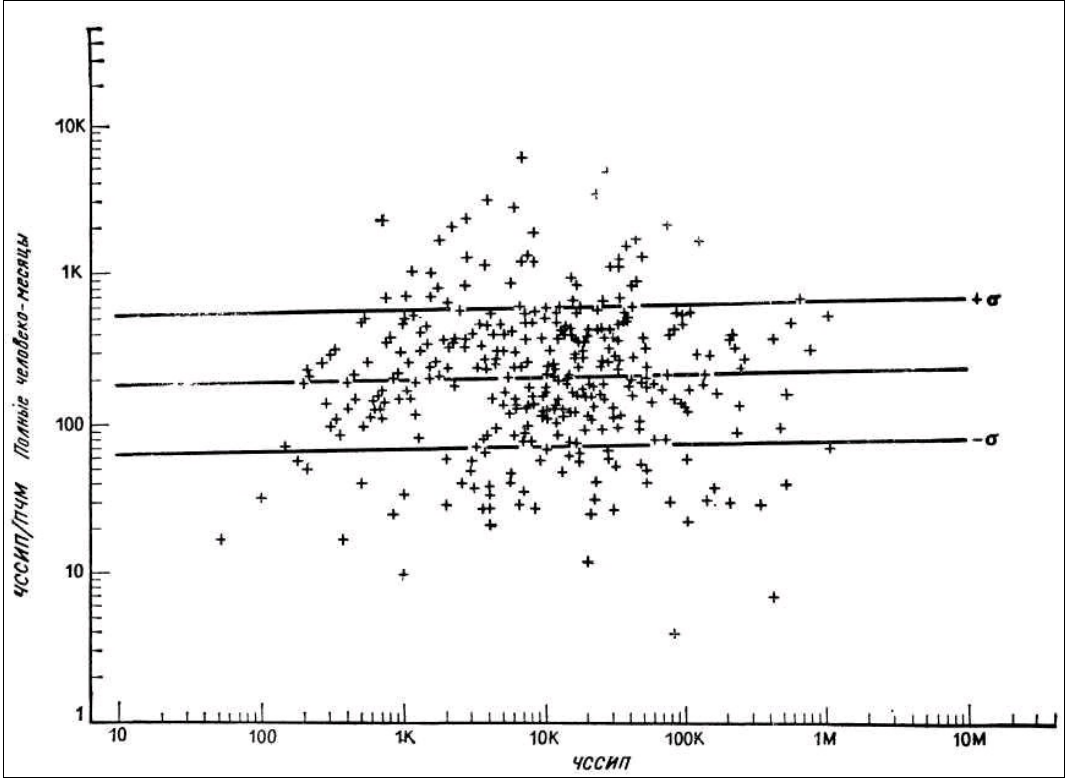


Рис. 6.18. Зависимость производительности труда от числа строк создаваемой исходной программы.

Множество чисел, интерпретируемых как чисто случайные.

Рис.6.18 заимствован из исследования Правительства США, посвященного изучению соотношения числа строк сдаваемой исходной программы (ЧССИП) и производительности (СТП/ЧМ). Заметьте, что по обеим осям разметка нанесена в логарифмическом масштабе. Единственное, что показывают эти данные, это, что никаких полезных оценок на их основании сделать нельзя. Разнообразие их просто поразительно.

Рисунок полезен только тем, что показывает отсутствие у нас достаточного количества данных. На этом графике для программ размером в 100 000 СТП внутрь области, ограниченной линиями, проведенными на уровне отклонений от —*а* до +а, попали проекты с коэффициентами и 80 и 800 ЧССИП/ЧМ. Такой разброс не может быть полезным для прогнозирования!

Предупреждение. Использовать строки программ как меру производительности труда и метод оценки в настоящее время просто абсурдно. Неизвестно, каким образом можно сопоставлять реализуемую функцию и строки программы. Если руководство настаивает на повышении скорости создания строк программы, программист всегда может написать “рыхлую”, “пухлую” программу, которая сделает хорошими все показатели по производительности труда, но которая абсолютно не подходит для настоящего дела, так как для выполнения такой программы может понадобиться слишком много памяти и процессорного времени. Мне встречалось много таких “хороших” руководителей, требующих от каждого программиста как можно больше строк программ в месяц.

Форма отчетности по строкам программ

Желательно, чтобы отчеты по проектам выглядели как-нибудь так:

Название: Программа наведения ракет: с математическим уклоном; загружается с перекрытием; критическая по времени

Функции: Расчет курса, скорости и всех управляющих сигналов, необходимых для ведения ракеты к точке перехвата другой ракеты.

Не включено в программу:	а) Отслеживание всех ракет; делается другой программой.		
	б) Прием и передача всех сигналов; делается где-то в другом месте.		
Всего выполняемых стп — в рабочей программе	36441		
Написано СТП (выполняемых) ЯВУ	7014		
машинный язык	1314		
Не написано, но выполняется	8424		
Человеко-месяцев (прямой подсчет)	340 человеко-месяцев		
Программы*, написанные, но не сохраненные (выброшенные)	26584		
	8144 Тесты		
	11240 имитаторы		
	3100 моделирование		
	4100 проч.		
Программы *, написанные и оставленные, но не выполняемые	50445 всего		
	22814 тестовые программы последней версии		
	18416 примечания		
	1114 моделирование		
	8101 варианты данных		
Время от начала работы до сдачи	18 месяцев		
Время первой итерации	Составления спецификации требований	4 месяца	Составления проектной документации	8 месяцев

*) Имеются в виду рабочие программы.

Этот пример приведен здесь для иллюстрации некоторых принципов, изложенных на предыдущих страницах. Некоторых, но отнюдь не всех В полную отчетную форму необходимо включать подробные ответы на многочисленные вопросы, в которых, как мы уже установили, мы нуждаемся — какой тип вычислительной машины использовался для трансляции и т.п.

Некоторые результаты сбора статистики

В результате всех этих усилий мы пришли к тому, что никто не способен предсказывать производительность труда для выбранного случайным образом большого программного проекта. Слишком уж много параметров, слишком много качественных

факторов, оказывающих значительное воздействие на исход дела. Мы можем оказаться недалеко от истины только для некоторых работ, которые ведутся проверенной группой в хорошо известной прикладной области.

Ниже приводятся некоторые промежуточные результаты замеров, проводившихся в IBM над программными проектами. Работы сгруппированы на основе предсказаний их руководителей по степени трудности. Посмотрите, какие наблюдаются грандиозные различия в производительности труда, измеренной в строках текста программ (СТП) в месяц.

	Достигнутая производительность труда
Маленький/легкий * (300—900)	
1	517
2	615
Средний/легкий * (200—600)	
3	199
4	39
5	100
6	286
7	524
8	28
9	109
10	167
11	274
12	1071
Средний/трудный* (100—300)	
13	562
14	403
15	128
16	81
17	68
18	400
19	253
20	227
21	250
22	163
23	198
Большой/трудный * (50—150)	
24	186
25	68
26	284
27	182
28	229
29	46
30	120

*) Производительность труда, предсказанная руководством проекта.

Оценка

Оценка размеров программы

Если бы нам заранее были известны размеры программ, которые нам предстоит написать, наше положение было бы намного лучше уже с самого начала разработки большого проекта.

Предложив опытному разработчику изучить уже работающие системы, мы могли бы получить хорошие оценки числа строк программы для тех работ, которым можно найти аналоги среди ранее запрограммированных.

Здесь мы со всей очевидностью сталкиваемся с неким порочным кругом: “Вам нельзя доверить работу, у вас нет опыта”. “Как же я могу получить опыт, если мне не дают работать?” Наше положение именно таково. Мы хотим, чтобы оценки делались кем-то, кто уже делал это в прошлом. Или делал что-то, похожее на то, что мы пытаемся сделать.

Теперь нам очень пригодятся наши знания о пяти типах использования вычислительных машин и таксономии программного обеспечения. То, что человек имеет опыт разработки крупных *прикладных* систем, не означает, что он сможет оценить усилия, необходимые для создания *системного* программного обеспечения. Эта ситуация должна служить вам в качестве предупреждения. Не надо слушать прикладников, пытающихся оценивать трудности системного обеспечения! Не позволяйте людям, разрабатывавшим пакетные программы, оценивать системы реального времени. Это же относится и ко всем другим категориям.

Факторы, определяющие трудность разработки

На усложнение или облегчение разработки влияет сразу делая комбинация факторов. Прежде, чем закончить главу я хочу перечислить 27 из них. Все они разбиваются на три основные категории:

(A) Функция, которую надо выполнить
(B) Окружение в фазе использования
(C) Факторы, действующие в фазе разработки

Трудность разработки программ = (A) X (B) X (C)

Таблица 6.2. Трудности разработки

	Тип использования				
	I	II	III	IV	V
Функциональные					
Функции, которые надо реализовать, количество					+
Функции, которые надо реализовать, сложность		+			+
Функции, которые надо реализовать, ясность					+
Взаимодействие людей с системой			+	+	++
Число различных пользователей системы			+	+	+
Число запусков программы			+	+	+
Число машин, на которых будет выполняться система					+
Функции, которые надо реализовать, взаимодействия				+	
Элементы данных					
Ожидаемая частота внесения изменений в систему			+		
Взаимодействия с другими системами					+
Окружение фазы использования					
Центральный процессор	-		+	+	+ +

Ввод/вывод	+	-	+	
Основная память	-		+	
Вспомогательная память		+		
Надежность/доступность		+	+	+ +
Реальное время			+	+
Факторы фазы разработки				
Адекватность операционной системы		+	+	+ +
Время, выделенное на создание программного обеспечения			+	
Доступность средств разработки; языки, отладчики и т.д.			+	
Доступность машин при разработке программного обеспечения			+	
Знакомство группы, проводящей разработку с аппаратурой				
Знакомство группы, проводящей программирование с инструментальным программным обеспечением				
Число модулей				
Стабильность средств создания программного обеспечения				
Стабильность вычислительной машины			+	
Квалификация пользователя			+	
Нормальная трудность				
Легче	-			
Труднее	+			
Намного труднее	+ +			

Схема трудности разработок. Таблицу 6.2 можно использовать как справочник, позволяющий определить, на что нужно обращать внимание, приступая к разработке системы. Расставленные в ней оценки "+", "-" — укажут вам,, облегчает ли (-) данный фактор разработку или усложняет (+) ее при переходе от систем типов I и **II** к системам типов **III,** IV или V. Если на схеме отмечен знак (-), производительность труда будет повышаться, если стоит знак (+), производительность будет ниже. Сначала рассмотрите схему, а затем приступайте к изучению объяснений по каждому из 27 пунктов.

Функциональные факторы

1. Функции, количество	Чем больше функций, тем больше нужно написать программ для их реализации. С ростом размеров программ трудность разработки программного обеспечения возрастает нелинейно.
2. Функции, сложность	Программа вычисления коррекции орбиты для полета к Луне в 50 или более раз труднее, чем программа добавления очередного взноса за покупку в месячную кредитную карточку. Управляющие программы логически сложны, а логическая сложность представляет большую проблему, чем сложность научная.
3. Функции, ясность	Некоторые функции (вспомните платежные ведомости) ясны и понятны. Другие находятся лишь в. головах старых мастеров, и при попытке записать их па бумаге может возникнуть страшная путаница, а ведь без записи нельзя программировать. Особенно трудно работать с системами типа V.
4. Незамкнутый цикл, человек взаимодействует с работающей системой	Системы, одним из элементов которых является человек, более сложны, чем какие-либо другие. Информацию для человека надо готовить особым образом; ответы, исходящие от него, надо приспосабливать к обстоятельствам; среди этих ответов может наблюдаться значительное разнообразие; человеку надо давать и дополнительную информацию, причем последовательность запросов нельзя сделать достаточно строгой, чтобы не получилось так, что он откажется от пользования системой.
5. Число различных пользователей программы	Разные пользователи по-разному воздействуют и на вычислительную машину и на ее программное обеспечение. Стало аксиомой утверждение, что чем больше пользователей у программы, тем больше ошибок будет в ней обнаружено. Под пользователем понимается не просто отдельная вычислительная установка. Машины и программы для управления ракетами могут располагаться в сотнях различных мест, но способ использования у всех будет один и тот же.
6. Число запусков программы	Если программа должна работать постоянно, нам надо позаботиться об эффективности (сколько аппаратуры необходимо для ее выполнения) в фазе использования. Если программа будет выполнена лишь единожды, ее эффективность не должна нас особенно волновать.
7. Число машин, на которых будет выполняться данная программа	Если разрабатываемая программа будет выполняться только на одной машине, нам можно несколько меньше обращать внимания на используемые ею машинные ресурсы. Программа для радиолокатора, выполняемая на сотнях вычислительных машин, на сотнях кораблей, должна быть до предела отточена и сжата. Лишние затраты памяти будут умножаться в сотни раз.
8. Функции и их взаимодействия	Некоторые сложные задачи весьма слабо взаимодействуют со всеми другими задачами, которые решаются на той же машине; другие же имеют настолько тесные связи, что оказываются буквально переплетенными.
9. Элементы данных	Число и размеры элементов данных, их разнообразие, их взаимодействия, их изменчивость, все это может иметь огромное влияние на размер программы и на трудности, возникающие в фазе проектирования. Брукс замечает: “Покажите мне свои данные, и я смогу больше рассказать о вашей программе, чем в том случае, если мне покажут блок-схемы”.
10. Ожидаемая частота внесения изменений в программу	Если моя программа стабильна, т.е. изменяется не слишком часто, мне можно строить ее совершенна иначе, чем в том случае, когда я ожидаю, что она будет часто подвергаться изменениям.
11. Взаимодействие с другими системами	Если наша вычислительная система должна полностью находиться в распоряжении каких-либо других систем, мы должны обязательна программировать так, чтобы учесть все типы возможных прерываний.
ФАКТОРЫ ФАЗЫ ИСПОЛЬЗОВАНИЯ	
12. Доступная мощность ЦП	Для выполнения задания каждая программа использует ЦП. Некоторые программы используют его более эффективно, чем другие. Когда ресурсы ЦП оказываются недостаточными, программистов просят так провести проектирование программ, чтобы они не просто выполнили задание, но выполнил его с учетом заранее описанных ограничений недостаточных ресурсов ЦП. И программисты пробуют и проверяют свою работу, и пробуют снова и снова. Программы будут работать, но с точки зрения времени программирования, это стоит очень дорого.
13. Доступные пути ввода/вывода	Логически эта проблема не отличается от работы в условиях недостатка ресурсов ЦП, но в данном случае наибольшая потребность возникает в числе путей, по которым данные передаются в машину и из нее. Много стараний должны приложить программисты, чтобы передать весь поток данных через небольшое число “портов”, имеющихся в машине.
14. Доступная основная память	Опять та же проблема, но уже со стороны памяти. Для хранения программ требуется память, а если памяти не хватает, программист должен “подкачивать” программы в основную память для выполнения и затем откачивать их во вспомогательную память. Для этого приходится затрачивать как время, так и опять-таки пространство. Может быть и другой вариант, в котором приходится втискиваться в отведенные рамки, создавая проект, а затем снова пытаются втиснуться в них. В 1946 году фон Нейман написал в одной из своих работ, что память является ключом к производительности. Это верно и в наши дни; скорость работы с памятью и ее объем есть критические величины.
15. Доступная	

15. Доступная вспомогательная память	Еще раз та же самая проблема; на этот раз со стороны вспомогательной памяти.
16. Надежность/Последствия отказов	Двухдневный отказ в системе типа I может быть вполне допустимым. Конечно, он причинит неудобства, но катастрофы за собой не повлечет. Однако, ни часовой, ни 10-минутный сбои нельзя допускать ни в системе управления авиалиниями, ни в системе управления полетом космического корабля, ни в системе противоракетной обороны. Если последствия отказов значительны, нельзя пользоваться стандартной операционной системой (распространяемой поставщиками аппаратуры). Та же проблема возникает и в системах реального времени. В этих случаях либо пишут новую операционную систему, либо модифицируют старую.
17. Ограничения реального времени	Если печать платежной ведомости длится 2.5 ч. вместо запланированных 2 ч, кого это по-настоящему волнует? Возникает лишь легкая досада. Но, если радиолокатор поворачивается за 6.4 с, вычислительная машина обязана принять от него данные, закончив к этому времени обработку предыдущей порции. Если этого не будет, система переполнится. В системе реального времени стандартная операционная система применяться не может. Сколько разных объектов может отслеживаться одновременно? Сколько заданий необходимо завершить перед тем, как приступать к решению новой задачи? Чем больше возможностей и необходимости[1] такого рода, тем больше приходится создавать системных программ для управления потоком данных и содержания его в порядке. Примеры: системы разделения времени; управляющие и командные системы; системы диспетчеризации воздушного транспорта.
ФАКТОРЫ ФАЗЫ РАЗРАБОТКИ	
18. Адекватность операционной системы	Производители вычислительных машин обычно выпускают или сдают в аренду, или продают большие операционные системы вместе с аппаратурой, которая помогает распределять и управлять работой различных аппаратных компонентов вычислительной машины Однако эти операционные системы могут не вполне подходить для разработки программного обеспечения.
19. Время, выделенное на создание программного обеспечения	Временной график становится доминирующим фактором в большинстве современных систем типа V. Причина кроется в том, что вычислительная машина и ее программное обеспечение обычно представляют собой лишь часть некоторой сложной системы. Спутники, корабли, оружие, ракеты, здания, фирмы — все это может задавать общий темп — а программное обеспечение должно быть готово одновременно с другими частями системы. Обычно график выдерживается — естественно за счет тех функций, которые надо выполнять. Сдается меньший набор функций, чем планировалось заранее, а недостающие вводятся в систему в более поздние сроки.
20. Доступность средств разработки	Средства создания программного обеспечения, как и любой другой инструментарий, существенно воздействуют на производительность труда. Перед тем, как новую машину начнут использовать в качестве инструмента, ее тщательно изучают. Богатый набор мощных средств значительно повышает производительность труда.
21. Доступность машин для разработки программ	Этот пункт имеет два аспекта. Во-первых, люди, строящие системы, понимают, что без вычислительной машины при разработке программного обеспечения обойтись очень трудно. Когда машины нет, либо она недостаточно мощна, все страшно замедляется. В то же время люди, незнакомые с ситуацией, не понимают того, что вычислительная машина представляет собой *основное* средство разработки во всем процессе создания программного обеспечения. Машина на стадии разработки используется столь же интенсивно, как и на стадии использования. Ну и конечно же машина необходима для тестирования программного обеспечения.
22. Знакомство группы, проводящей программирование, с аппаратурой	Во всех областях человеческой деятельности нужно отводить время на приобретение опыта, это же относится и к программированию. Если наша группа программистов уже работала с данной аппаратурой, значит в прошлом она уже приобрела опыт, и от этой группы можно ожидать работы с большей производительностью труда.
23. Знакомство группы, проводящей программирование, с разработкой программного обеспечения	Время на приобретение опыта, которое, как мы видели, нужно для работы с аппаратурой, нужно и для изучения инструментального программного обеспечения.
24. Число модулей	Число модулей и связей между ними относится к ряду факторов, значительно влияющих на сложность выполняемой работы. Эта область сходна с пунктом “функции, их взаимодействия”, но имеет и отличие от него в том смысле, что числа модулей не обязательно связано с числом функций, поскольку количество модулей связано с методикой проектирования программ, а количество функций неотъемлемо от выполняемого задания.
25. Стабильность средств программного обеспечения	Если средства создания программного обеспечения подвержены изменениям, этот и без того сложный процесс еще усложняется дополнительными, часто случайными проблемами. Примером может служить транслятор, в котором есть ошибки и который поэтому создает неправильные рабочие программы.
26. Стабильность аппаратуры	Если используемая в фазе выполнения вычислительная машина находится в состоянии доводки, программисты часто не могут разобраться, где находится ошибка, в программе или в аппаратуре. Это вносит постоянную путаницу. Это, пожалуй, наихудшая из всех ситуаций, с которыми может столкнуться группа, разрабатывающая программное обеспечение.
27. Квалификация пользователя	Пользователь есть настоящий заказчик и важнейшее звено группы разработки. Пользователь, ранее работавший с системой типа V, уже прошедший через весь этот процесс, является наилучшим из всех возможных партнеров разработчиков. Неопытный пользователь может полностью парализовать деятельность прекрасной группы разработчиков, поскольку новоиспеченные пользователи прокладывают себе дорогу к цели урывками.

В книге “Мифический человеко-месяц” Брукс написал, что системное программирование по крайней мере в три раза труднее, чем программирование трансляторов с языков высокого уровня, которое в свою очередь в три раза сложнее прикладного программирования. Системные программы, которые должны работать в режиме реального времени, в три раза сложнее обычных системных программ. Программы, не имеющие права на малейший сбой, в два или три раза сложнее системных программ. Программы, которые должны работать в диалоговом режиме, по крайней мере в два раза сложнее тех, что работают без диалога.

Основной вклад в стоимость разработки программного обеспечения вносят:

1. Масштаб. Количество реализуемых функций.
2. Ясность. Степень, до которой понятны все реализуемые функции.
3. Логическая сложность. Число условных переходов в расчете на сто команд.
4. Последствия сбоев. Сколько усилий при проектировании и программировании нужно затратить, чтобы удовлетворить всем требованиям по обеспечению надежности и восстановления.
5. Взаимодействия с человеком. Насколько часты и интенсивны взаимодействия с системой.
6. Требования реального времени. Сколь быстро должна быть выполнена нужная функция.
7. Стабильность инструментального программного обеспечения. Достигло ли оно нужного уровня стабильности и зрелости?
8. Стабильность вычислительной машины, на которой будет выполняться система. Достигла ли нужного уровня стабильности вычислительная машина, на которой будет выполняться система.

Это самые главные из 27 факторов, которые мы перечислили выше.

Теперь на примере этих 8 факторов продемонстрируем, как стоимость программ может возрасти с одного доллара за строку до тридцати двух с половиной. Если рассмотреть все 27 факторов и использовать наихудший вариант по всем из них, мы можем получить увеличение стоимости до двухсот долларов за строку.

Числа, которыми я буду тут пользоваться, не являются данными из какого-либо конкретного обзора или исследования. Это приблизительные цифры, основанные на моих собственных суждениях, я их выбрал не из-за их значений, а только для иллюстрации принципа возрастания удельной стоимости программы, и из-за того, что они находятся в полном относительном соответствии друг другу. Я считаю, что сильнее всего удельная стоимость программ увеличивается из-за нестабильности машины, используемой в фазе выполнения. Я приписываю этому фактору коэффициент 20.

В качестве исходной величины я выбираю удельную стоимость программы в 1 доллар за строку текста. В данном примере я буду игнорировать затраты, меньшие этой величины.

Программа, которой нельзя сбиваться, обойдется мне дороже, чем программа, которая выполняет ту же функцию, но при этом допустимы сбои. На сколько же это обходится дороже? Я оцениваю этот фактор — надежность, — коэффициентом 15. Следовательно, если создание "ненадежной" программы стоит один доллар за строку, то надежная программа будет обходиться по 15 долларов за строку.

Каждому из восьми основных факторов я могу приписать такие коэффициенты:

Масштабность (размер)	от 1 до 8
Ясность	от 1 до 10
Логическая сложность	от 1 до 10
Последствия сбоев	от 1 до 15
Взаимодействие с человеком	от 1 до 5
Требования реального времени	от 1 до 5
Стабильность вспомогательного программного обеспечения	от 1 до 10
Стабильность вычислительной машины в фазе использования	от 1 до 20

В самом плохом случае для всех и каждого из этих восьми факторов стоимость одной строки текста программы будет являться суммой максимальных значений коэффициентов, или 83 доллара за строку.

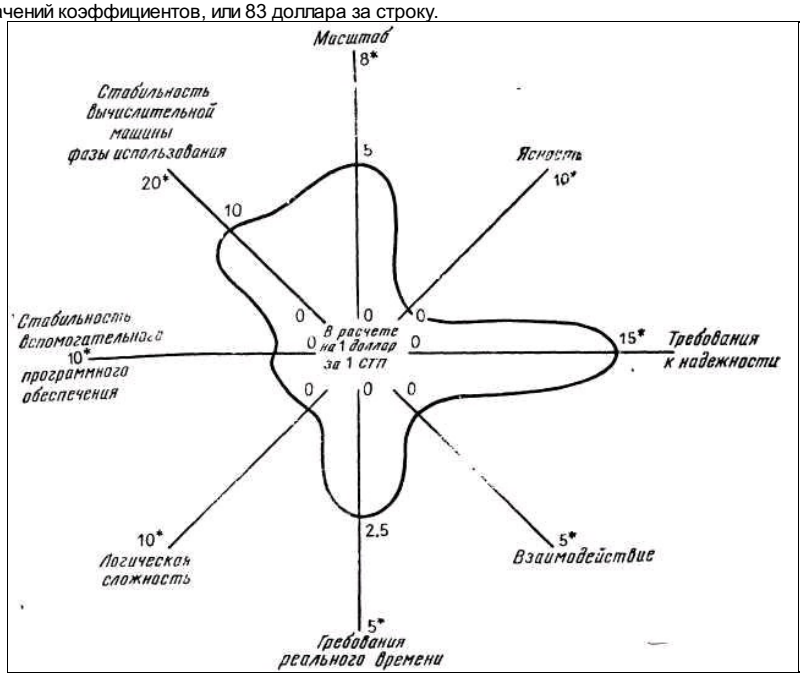


Рис. 6.19. Из чего складывалась общая оценка.

Однако в реальной ситуации редко может случиться так, что для каждого фактора будут наиболее неблагоприятные условия. Нужно попробовать оценить относительную трудность каждого из них для конкретной задачи по разработке. На рис. 6.19 приведены мои оценки для пакета программ размером в 10 000 строк для управления ракетой. Эти программы не должны сбиваться, должны обрабатывать данные, поступающие от радиолокатора каждые 4 с, никак не взаимодействовать с пользователем и работать совместно со стабильным вспомогательным программным обеспечением на "вполне стабильной" вычислительной машине. Логическая сложность минимальна, а ясность просто на отличном уровне.

Моя оценка стоимости — 32,5 доллара за строку — была получена суммированием стоимостей, показанных на диаграмме. Я начал с 1 долл. за строку, мне нужно было лишь определить множители.

По моему мнению, коэффициенты надо расставить так:

— множитель для логической сложности отсутствует
— множитель для ясности отсутствует
— множитель для взаимодействия с пользователем отсутствует
— множитель для стабильности вспомогательного программного обеспечения отсутствует
— множитель 5 для размера
— множитель 2,5 для реального времени
— множитель 15 для надежности
— множитель 10 для нестабильности вычислительной машины, применяемой в фазе использования

Все это вместе дает общую оценку в 32,5 доллара за строку. Еще раз повторю, что отдельные факторы должны умножаться на соответствующие коэффициенты, а общая стоимость определяется не перемножением, а суммированием по всем факторам.

Данный пример я рассматривал не ради численных коэффициентов, а лишь как иллюстрацию к методике. Для определения примерных оценок стоимости следует проводить два цикла подобных рассуждений; сначала для максимальных значений множителей по всем факторам, а затем для примерных значений по данной конкретной разработке. Числа, которые использовались в данном примере, выражают также мое мнение об относительной трудности, вносимой каждым отдельным фактором.

В представленном здесь методе игнорируется тот факт, что на самом деле трудность пропорциональна произведению отдельных факторов.

Если бы я применял эту методику в реальной ситуации, мне нужно было бы подсчитывать вклад всех 27 факторов, принимая во внимание те из них, которые, по моему мнению, оказывают наибольшее влияние на конкретное положение дел.

Опыт подсказывает мне, что те, кому удавалось сделать оценки наилучшим образом, проделывали их в обратном порядке. Они оценивали число людей, необходимых для выполнения той или иной функции, и время, необходимое этим людям для выполнения задания, а уже затем определяли, сколько строк в день будут писать эти люди, выводя отсюда полное число строк, необходимое для реализации функции. Я назвал такой порядок обратным. Но, возможно, это и есть прямой путь, а тот метод, который был описан ранее, напротив, является обратным.

Как проводить оценки

Их надо проводить осторожно! Все, что может испортиться, обычно портится. Оценки должны проводиться наиболее квалифицированным специалистом в автоматизируемой области. При необходимости следует привлекать консультантов со стороны. Приглашенные консультанты должны проверять все самые важные оценки. Не следует доверять оценкам тех, кто никогда ранее не участвовал в разработках программ данного типа или объема.

Проведение правильных оценок и правильное проведение оценок. Наши оценки остаются с нами до конца. Но, после того как мы провели их, мы составляем бюджет. А после составления бюджета мы приступаем к разработке. И здесь правильное проведение оценок становится более важным, чем сдача полного набора функций в первый срок. Возможны такие варианты: пересмотр в сторону увеличения стоимостных оценок, задержка выполнения графика работ или уменьшение числа функций с целью удовлетворения первоначальным оценкам. В конце концов, это были всего лишь оценки.

Предположения при проведении оценок

Хороший руководитель разработкой программного обеспечения может явно сформулировать предположения, на которых основываются его оценки. Одними из важнейших являются предположения, касающиеся окружения разработки. Наиболее существенным является состав исполнителей, и не только с точки зрения их количества, но также и с точки зрения качества.

Очень важно состояние дел по определению требований. Иногда случается так, что руководитель работ всей системы в целом заявляет, что требования уже зафиксированы. Хороший руководитель программного обеспечения знает, что в больших проектах это редко бывает так на самом деле, но в начальной стадии он еще не может доказать этого. Поэтому в своих оценках он должен еще учитывать возможные дополнительные работы. О такого рода предположениях мы еще поговорим в разделе, посвященном руководству работами.

Идеальные графики. Идеальными можно считать те графики, которые основываются на том, что все поступает вовремя — новый радиолокатор, спутник, вычислительная машина и т. д. Так не бывает никогда, но, если вы хотите выдержать конкурентную борьбу или получить одобрение своего проекта, вы должны с самого начала объявить о своем намерении придерживаться графика. Только не надо верить в это самому! В этом кроется еще один “социальный” аспект данной области! Мы создаем идеальные графики, чтобы получить премию, чтобы проект успешно прошел все защиты.

Организация усилий по разработке программного обеспечения

Правильное решение организационных вопросов лежит в основе большинства наших изумительных достижений, но в то же время мы очень часто путаемся при выборе наиболее правильной организационной структуры. В классической ра боте Питера Дракера “Организационные принципы” *{Drucker P. Concept of the Organization (New York: The John Day Co., Inc., 1972).}*, вышедшей в 1945 г., указывается, что высочайший уровень промышленного производства, достигнутый Соединенными Штатами во время второй мировой войны, основывался не столько на технологических достижениях, сколько на достижениях в области организационной.

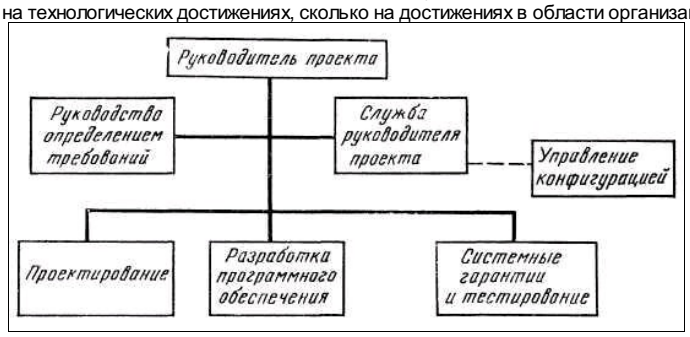


Рис. 6.20. Организация разработки программного обеспечения.

Лишь очень небольшое число организационных принципов имеют для программного обеспечения существенные отличия от других областей разработки. Значительные различия в организации могут наблюдаться при разработке программного обеспечения разного объема и типов, кроме того, организационная структура может изменяться в процессе разработки по мере завершения ее отдельных стадий.

В дальнейших рассуждениях мы будем пользоваться схемой, приведенной на рис. 6.20.

Ключевые моменты больших проектов

1. Руководитель выработкой требований. Эта должность редко встречается в организациях, занимающихся программированием, однако функции, относящиеся к ней, имеют важное значение. Когда однажды я спросил руководителя одного проекта, оцениваемого в 100 млн. долларов, “кто отвечает за выработку требований”, он пришел в крайнее замешательство. По его словам, отвечал за требования он сам. Это означало, что никакого конкретного ответственного не было! Когда время начнет поджимать, кто сможет отстоять интересы пользователя?

2. Отдел руководителя программного проекта следит за бюджетом, графиками, оборудованием, хозяйственными вопросами и т.д. Сильная, мощная группа может снять множество проблем.

3. Обязательно должен быть назначен руководитель проектированием, проектировщик или главный архитектор. Этот человек не должен заменять руководителя разработки (руководителя производством или руководителя реализацией). Проектирование относится к наиболее творческим видам деятельности и существенно отражается на успешном исходе работ по разработке крупного программного проекта. Проектирование и руководство — это совершенно разные виды деятельности.

4. Руководитель разработки программным обеспечением должен управлять работой всех групп, которые создают рабочие программы.

5. Руководство конфигурацией (РК). Постоянный комитет, в который входят по крайней мере руководитель разработки, руководитель выработкой требований и главный руководитель проектом, должен раз в неделю обсуждать все предполагаемые изменения, результаты тестирования и со стояние системы. Это единственный способ быть в курсе всех событий, происходящих при разработке крупной программной системы.

6. Группа системных гарантий и тестирования должна создаваться уже на самом раннем этапе, причем оставаться совершенно независимой от группы разработки.

7. Организация должна максимально возможным образом отражать состояние проектируемых работ. Если при проектировании выяснится, что необходима отдельная подпрограмма для обработки входных данных, то при условии, что объем ее достаточно велик, нужно обязательно создать отдельную группу по входной информации.

Среди наиболее часто встречающихся ошибок, которые попадались на моем пути, можно отметить такие:

1. Нет проектировщика. “Проект будет развиваться!”
2. Нет независимой группы тестирования.
3. Нет независимой группы по определению требований.

Общая организация труда при создании программного обеспечения будет обсуждаться нами в гл.8, где мы покажем, что должны иметь большие организации, если в них действительно серьезно относятся к руководству программным обеспечением.

Надзор над разработкой

Ни одну разработку крупной программной системы нельзя контролировать без аккуратного, внимательного надзора. Для понимания того, идет ли проектирование в соответствии с графиком, опережает его или отстает, совершенно необходимо заранее определять даты завершения работ по каждой подпрограмме или модулю и следить за соответствием прогресса и затрат.

Имеется много способов отслеживания множества дат завершения отдельных работ, которые можно выделить в большом проекте. Хорошо работают схемы, составленные па методике PERT или GANTT, если, конечно, их применяют, но на них приходится тратить и людские и денежные ресурсы. Хорошую помощь при прослеживании всех необходимых подробностей могут оказать вычислительные машины и работающее на них программное обеспечение.

Имеются десятки и десятки систем “контроля за проектом”, которые позволяют совершенно точно прослеживать ход разработки, *если* их применяют на практике. Но руководство при этом должно проверять и перепроверять точность всех поступающих отчетов и то, что система не подвергается губительным воздействиям. Если что-нибудь идет не так, как это было бы нужно, такие системы способны выдать сигналы об этом уже в самые первые моменты нарушений. В этот момент руководители самых нижних уровней обычно пытаются “оправдаться”, считая отклонения простыми случайностями и надеясь решить все возникшие проблемы уже в ближайший же месяц.

Хороший руководитель программистами знает и имеющиеся в его распоряжении средства, и то, что они сообщают ему. Возможно, что ему начнут передавать неприятные сообщения, и если он достаточно компетентен, то должен обратить на это внимание.

В каждом большом проекте должна существовать механизированная система выпуска отчетов по проекту. Каждую неделю с помощью вычислительных машин необходимо выпускать документы, в которых отражается все, что произошло за это время.

Руководители просто обязаны изучать эти документы. Помните: затраты не могут служить мерой прогресса.

Управление

Разрешать людям делать то, что они хотят, можно обычно только в тех случаях, когда нам не надо пытаться уложить все работы в рамки какого-нибудь графика или бюджета. Во многих рабочих ситуациях было бы безумием разрешить людям делать все, что им угодно. В то же время для большинства рабочих ситуаций характерно, что они допускают возможность прослеживать, что же происходит.

Разработка программного обеспечения оставалась невидимой в течение 30 лет. С огромной скоростью она становится все более наглядной. Люди могли делать все, что пожелают. Для небольших, относительно самостоятельных программ это хорошо. В большие системы, состоящие из огромного числа взаимодействующих программ, это вносит мною бед.

Руководство крупными программными разработками нуждается в возможности прогнозирования и управления. Все части проекта должны соответствовать друг другу, поэтому после утверждения проектной документации свобода выбора из различных вариантов должна быть сведена к абсолютному минимуму. Исполнителя нужно держать под *жестким контролем.* (В качестве примера того, как самостоятельность, независимо от того насколько хорошо она обоснована и прочувствована, может уничтожить плоды труда людей, смотрите пример ошибки на с.232.)

Экономия усилии

Экономию усилий можно только приветствовать в небольших и не очень важных проектах разработки программного обеспечения. На крупные разработки она действует *смертельно,* такое же действие она оказывает и на группы сопровождения.

Экономия усилий с помощью обходных путей обычно связана с нарушением стандартов, а ведь только стандарты и правила могут позволить нам построить большую, сложную систему. Конечно, обходной путь может привести к ускорению работы, поскольку по крайней мере часть проблем просто игнорируется. Экономия усилия обходится очень дорого и приносит неудобства в течение всего периода жизни программной продукции или программного обеспечения проекта.

Управление конфигурацией

Необходимо, чтобы вся система, все, что в ней происходит, все вносимые изменения и их последствия были полностью управляемыми. Это необходимо при разработке сотен тысяч строк текста программ с помощью сотен программистов. Помочь в осуществлении такого управления могут многие автоматизированные (снабженные вычислительными машинами) системы, но важнейшее значение имеют качество работы руководителей и время разработки.

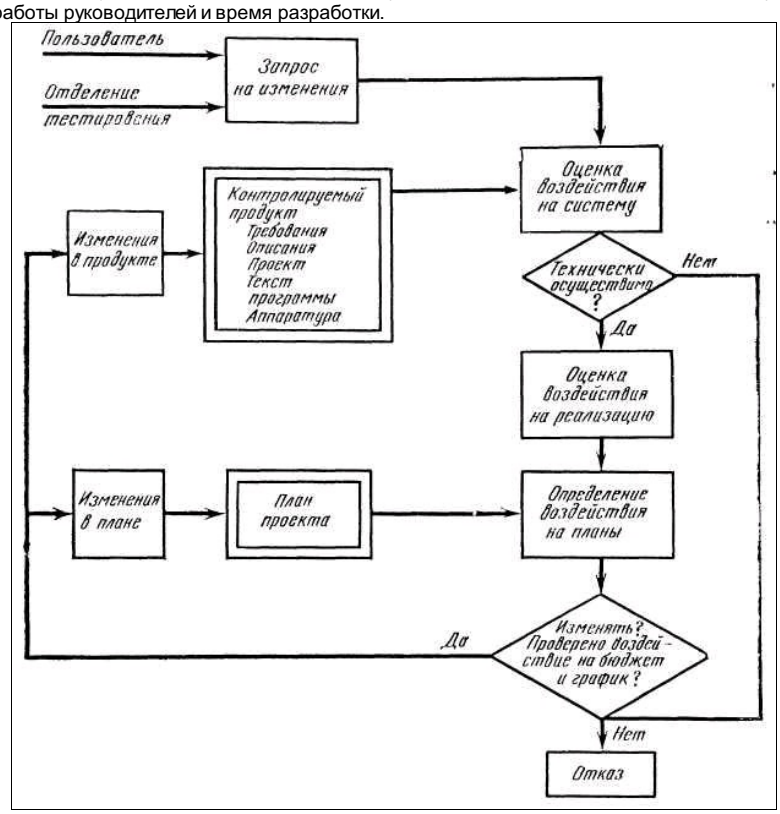


Рис. 6.21. Цикл управления конфигурацией.

Существенное влияние оказывают еженедельные совещания основных руководителей. Участие заместителя вместо начальника не допускается. На этих совещаниях принимаются важнейшие решения, связанные с графиками работ, бюджетом, функциональными возможностями системы и распределением людских ресурсов. На них определяются приоритеты работ. Последовательность событий можно проследить на рис. 6.21. Ключевым моментом является руководство.

Автоматизированная матрица модулей/функций

На примере системы по составлению платежных ведомостей (в конце гл.5) мы видели, что частями системы могут становиться многие программы довольно значительных размеров. Конечно, каждая из перечисленных нами программ состоит из множества более мелких модулей команд.

В небольших системах число функций и модулей небольшое, но стоит системе вырасти хотя бы до средней величины, как число модулей и функций значительно увеличится. Очень большую помощь группе управления конфигурацией в деле отслеживания команд, выполняющих ту или иную функцию, может оказать простая матрица, в которой столбцами являются сведения о модулях, а строками — сведения о функциях. Многие поставщики могут предоставить программы автоматической обработки таких матриц, рассчитанные даже на те случаи, когда число функций и модулей достигает нескольких тысяч.

Ключ к успеху — руководитель разработки

За многие годы я понял, что, несмотря на всю важность инструментальных средств и проверок и их сбалансированного использования, наиболее важным для успешного завершения работы является выбор ответственного руководителя. Я видел многие неудачи “великих” компаний и великолепные работы “середнячков”, которые целиком были связаны с личностью руководителя разработкой программного обеспечения. Какими же качествами должен обладать этот руководитель и каким образом следует выбирать наиболее подходящую кандидатуру?

Выбор руководителя разработкой программного обеспечения

Первое качество, на которое надо обращать внимание при выборе руководителя разработки программного обеспечения, это не его технические возможности и даже не его личный опыт, а его эмоциональная зрелость и “неуступчивый” характер. Хорошие руководители разработкой умеют соразмерять цели с реальностью и могут твердо стоять на своем, говоря “нет” непрерывным требованиям ввести “еще несколько” функций, совсем немного здесь и еще чуть-чуть там. Такое наращивание функций называется “потерей элегантности” и может оказаться фатальным. Они знают, что время, необходимое на выполнение n + 1 тривиальных вещей, вдвое больше времени, необходимого на выполнение n вещей, если n стало уже достаточно большим (возражение Логга на закон Грея)[1]*{Dickson P. The Official Rules, New York: Delacorte Press, 1978.}*. Они знают, что Брукс был прав, когда в книге “Мифический человеко-месяц” написал: “Как получается, что программы опаздывают на год? Это происходит постепенно”. Они знают, что *требования* являются первой линией обороны. Не допускай “врага” к этим позициям, и ты будешь контролировать все события.

Вторым необходимым качеством руководителя является внимательность к деталям. Большинство хороших руководителей

разработок программного обеспечения тратят очень много времени на изучение мельчайших подробностей работ. И не напрасно.

Третьим желательным качеством руководителя программным проектом является способность к руководству вообще.

Затем следует учитывать знание области, в которой будет проходить работа. Затем опыт. Приходилось ли ему ранее участвовать в какой-нибудь столь же крупной разработке? Не забывайте о “принципе Питера”. Многие люди, бывшие способными заместителями, никогда не могут стать хорошими руководителями! Разработка программ объемом в миллион строк намного более чем в 10 раз сложнее разработки в сто тысяч строк.

Какие качества нежелательны?

Избегайте “коммерческих” руководителей. Многие из тех, кто хорошо чувствует себя в коммерческой деятельности, имеют большую склонность к компромиссам. Мне редко попадались руководители, которые смогли распространить качества, нужные в коммерции, на руководство проектами. Часто случается, что коммерческий руководитель пытается найти компромиссы, чтобы удовлетворить сразу всех.

Технический опыт

Для того чтобы стать программистом, не обязательно иметь техническое образование. Но с руководством большими сложными структурами в программном обеспечении, нахождением оптимальных “решений” при соединении сотен отдельных частей, распределением работы между сотнями технических разработчиков — со всем этим гораздо лучше справится человек, имеющий техническое образование. Изучение некоторых новейших средств структуризации этих сложных видов деятельности показывает, что они относятся к самым передовым областям математики и техники.

Карьера разработчика программного обеспечения

Некоторые аспекты деятельности руководителя разработкой программного обеспечения могут внушать беспокойство. Оценки не бывают точными и объективными. Единственной “хорошей” методологией остается личный опыт, а проведение оценок можно отнести к области черной магии. Не известно еще, как измерить прогресс в этой области. Определить, на сколько проект близок к завершению, можно, только пользуясь хорошими методами автоматизации разработки и системой слежения за процессом. Для предсказания, измерения и численного определения производительности труда нет ни одного хорошего метода:

1. Трудно сказать, насколько велика данная работа.
2. Вы не можете сказать, какова будет (или даже была) производительность труда ваших людей.
3. Вы не можете указать процент выполненной работы.
4. Трудно сказать, в каком направлении идет работа!
5. Трудно сказать, насколько быстро вы движетесь!
6. Трудно сказать, насколько далеко вы продвинулись! Найдется ли человек, который захочет взять на себя руководство работой при таком спектре нерешенных проблем?

Пять стадий развития всех новых проектов

1. Эйфория.
2. Разочарование.
3. Поиск виновных.
4. Наказание невиновных.
5. Награждение непричастных к делу.

Каждый руководитель проектом должен осознавать, на какой стадии развития находится проект, и знать, что первую группу руководителей обычно снимают при возникновении первых же трудностей. Принимаясь за работу, необходимо знать, какому риску вы подвергаетесь.

Печальная участь первопроходцев

Сказано точно. Изучение новых разработок показывает, что первый руководитель и второй, пришедший ему на смену, обычно кончают одинаково — их снимают, Только после одного или двух “кровопусканий” главное руководство может занять реалистические позиции. Почему люди вообще хотят стать руководителями больших разработок программного обеспечения? Потому что это наиболее быстрый способ продвижения по служебной лестнице, “путь наверх” — если, конечно, вы сможете удержаться.

Вы будете прекрасно вознаграждены, если преуспеете.

На этом пути можно достичь немыслимых высот, что принесет вам большое моральное удовлетворение. Лучшие руководители разработками программного обеспечения мне рассказывали с гордостью о том торжественном моменте” когда зажигались все огни и вся общенациональная система вступала в действие!

При управлении большими работами должно быть очевидно, что нужно пользоваться минимально возможным числом исследований и новейших методов.

Почему же это должно быть очевидно? Это не так очевидно, как можно было бы подумать! В 1961 г. комитет” созданный по предписанию президента Джона Ф. Кеннеди для расследования причин воздушной катастрофы над Нью-Йорком, вынес рекомендации по автоматизации управления авиалиниями. Комитет настоял на том, чтобы Федеральное авиационное агентство (FAA) для всех новых систем покупало только “отработанные” вычислительные машины. Это ограничение было наложено потому, что в конце 1950-х г. FAA стала вкладывать деньги в разработку вычислительных машин — и деньги стали уходить в эту область, *а* не на разработку собственно систем управления авиалиниями.

Похожая история произошла в середине 1970-х г. в системе здравоохранения. Разработчики перестали заниматься созданием системы, которая могла бы удовлетворить запросы врачей, медицинских сестер, администраторов, специалистов и обслуживающего персонала, и занялись разработкой “сети” мини-ЭВМ и распределенной базы данных.

Ни FAA, ни медицине не были нужны никакие новаторства в области аппаратуры по обработке данных. И без этого было достаточно хлопот с системными проблемами. При разработке больших систем не разрешайте вашим сотрудникам заниматься изобретательством. Пусть этим занимаются в научно-исследовательских центрах.

Небольшая консультационная компания потеряла 300 000 долларов — доход за несколько лет — при попытке закончить работы по контракту, в которых использовалась машина фирмы IBM “Series 1”. Аппаратура была великолепной; а системное и инструментальное программное обеспечение было отработано не до конца. Каждый раз оказывалось, что люди натыкались на какую-нибудь неисследованную проблему. Программное обеспечение было новым! По этой причине фирма IBM не давала по нему никаких гарантий. И контракт был передан IBM.

Будет ли удовлетворен настоящий пользователь?

Совершенно необходимо руководить определением требований на всем пути от пользователей до программистов. Звучит это просто, но следы могут потеряться сразу в нескольких разных точках. Во всем цикле создания большой системы участвует очень много людей и организаций.

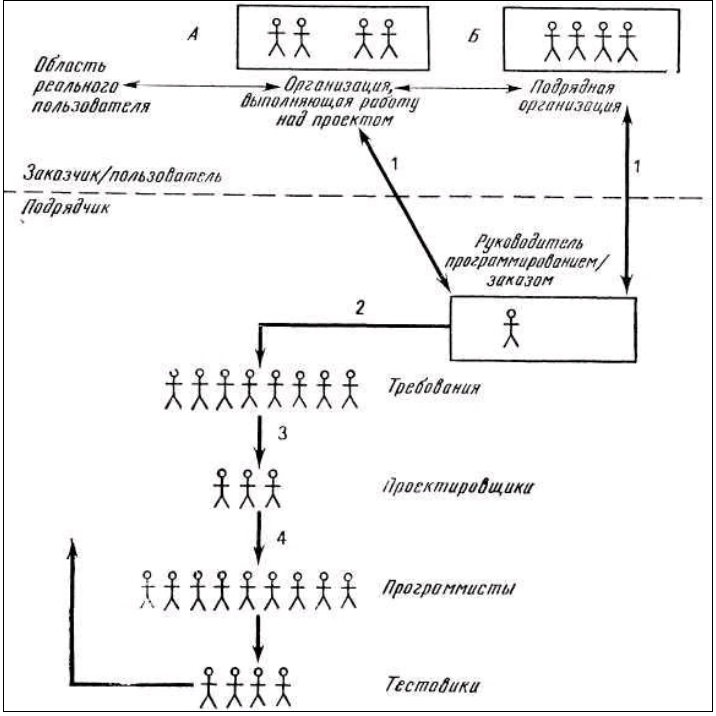


Рис. 6.22. Организации, занятые в одном проекте.

На рис. 6.22 показана весьма типичная организация работ над большим проектом. Истинные требования, поступающие от пользователя, могут подвергаться искажениям и на пути Л, и на переходе *В.* Меткой 1 мы преднамеренно отметили сразу два пути. Какой из них "правильный"? Они находятся в постоянном конфликте.

Пути 2, 3 и 4 тоже не являются "чистыми". Сколько проектов терпели неудачи из-за того, что проектировщики или программисты решали, что *они знают,* что надо делать, а люди, стоящие выше на служебной лестнице, несут околесицу? Таких проектов было слишком много! Я видел системы, где программисты знали больше, чем проектировщики, и поэтому строили систему по-своему. Видел системы, где проектировщики "знали" все, что требовалось. Видел и системы, где ни один человек ни разу не поговорил с подлинным, настоящим пользователем. Короче говоря, искажения могут возникать и в точках *А* или В, и в точках 1, или 2, или 3, или 4, и все они плохо влияют на систему в целом.

Руководитель проектом, в который входит разработка большой программной системы, должен постоянно проверять, проверять и проверять, чтобы быть уверенным в том, что строящаяся система будет принята к использованию.

Важным шагом на пути создания действительно полезной системы является введение в группы проектировщиков й управления конфигурацией людей, которые уже раньше выполняли необходимые функции.

Прослушивания

Я встречался с разными результатами прослушиваний, с такими, например, когда утверждалось, что все идет прекрасно — и так и было на самом деле,— и с такими, когда говорилось, что все идет хорошо — а затем наступала катастрофа! Иногда предупреждали о предстоящей неудаче, и она случалась. А бывало и так, что предсказанная катастрофа так и не наступала.

Другими словами, при прослушиваниях часто возникают ошибочные выводы! Те, кого вызывали на прослушивание, заявляли: "Мы знаем больше, чем эти эксперты; *они* ошибаются". И верно, часто так оно и было. Каким образом руководитель может определить, кто же прав? Не означают ли эти ошибки, возникающие в результате прослушиваний, что сама идея таких прослушиваний неверна?

Прослушивания очень важны

Несмотря на то что эксперты часто ошибаются при прослушиваниях в своих оценках, прослушивания являются очень важным инструментом руководителя. Независимо от того, правильны ли выводы прослушиваний, сам их процесс вносит в организацию дополнительные средства поддержания дисциплины. Происходит взаимообмен идеями по организации, проектированию и ведению процесса разработки. Часто благодаря одному только усилению внимания в проекте возникают улучшения.

Что необходимо выносить на прослушивания — и когда?

Все работы определенного объема должны прослушиваться каждые полгода. Любая работа, в которой наметились признаки неудачи, должна подвергаться немедленному прослушиванию. Такими признаками могут служить нарушения графика работ, чрезмерные сверхурочные работы, низкий уровень использования инструментальной вычислительной машины, недоукомплектованность персоналом и т.д.

Что такое "прослушивание"?

Прослушивания отличаются и от сквозного контроля, и от инвентаризации. При прослушивании проводится выборочная проверка ключевых моментов программной продукции или программ, созданных для обеспечения проекта. Это формальная проверка, обычно выполняется людьми, незанятыми в работе над этим проектом. Сквозной контроль есть детальный обзор, он похож на прослушивание в миниатюре. Обзоры проходят более поверхностно, чем прослушивания, при них не вдаются в особые подробности. Инвентаризация заключается в составлении списка того, что уже проделано, и тем напоминает прослушивание, но проводится обычно совершенно в других целях.

Кто должен участвовать в прослушиваниях?

Производить прослушивания должны самые лучшие, опытные специалисты. Они обязательно должны иметь опыт работы в данной области. Посылать на прослушивание диалоговых систем людей, занимавшихся разработкой пакетных программ, не имеет смысла. Наилучшими экспертами могут быть временно свободные разработчики. Группы экспертов, существующие достаточно продолжительное время, становятся неэффективными.

В чем вред прослушивании?

Прослушивания оказывают разрушительный эффект. Руководитель проектом прямо скажет вам, что за каждый день, который отводится на прослушивание, он должен резервировать 2 дополнительных рабочих дня. Он считает, что такое воздействие слишком велико, чтобы подвергать ему проект. Участвуйте в прослушивании и не соглашайтесь на изменения в графике. Типичный ответ руководителя: "Посмотрим".

Отчеты на прослушиваниях — делайте их устно

Ценность имеют и устные и письменные отчеты. Неоднократно я замечал, что мои эксперты старались устно заявить о глубине возникших проблем, обнаруженной путанице или некомпетентности, отказываясь излагать это письменно. Запись обычно приводит к смягчению оценок до такой степени, что существо дела теряется.

Первый выход на прослушивание или в группу инвентаризации

Когда эксперты первый раз посещают разработчиков или группу сопровождения с целью прослушивания, они должны следовать таким, на мой взгляд весьма ценным, советам:

1. Не задавайте наводящих вопросов. Выбор материала для прослушивания, который сделан руководителем разработки, тоже многое вам скажет.
2. По крайней мере первый день целиком посвящайте прослушиванию; отложите выяснения на второй или третий день.
3. Обратите внимание на то, кто представляет прослушиваемых: коммерческий директор или сам руководитель программным

обеспечением.

4. Обратите внимание на используемые термины. Используется ли, например, слово “модульность”? “Упрятывание информации”?

5. Требуйте четкого определения каждого термина.

6. Попросите посмотреть текущую схему организации работ и краткие резюме ведущих разработчиков.

7. Попросите *посмотреть* документацию по стандартам; обратите внимание на даты их изменения.

Кадры и инструментарий

Руководитель разработкой программного обеспечения должен заботиться об очень многих вещах, но два момента являются определяющими — люди, которые будут выполнять работу, и средства, которыми они будут это делать. Выделение людских ресурсов может быть двояким! С упором на количество или на качество. В крупных разработках нам нужно и то и другое! Ключом к успеху является качество. Правильно подобранные люди способствуют успеху всего проекта, и наоборот. Хорошие работники стоят тех денег, которые им платят; но встречаются они редко!

После завершения планирования и проектирования необходимо привлечь к работе большое количество людей. Здесь опять нас ждут неприятности. Мир страдает от хронического недостатка разработчиков программного обеспечения. Не надо недооценивать суровости такого положения. Если вы по плану должны выполнить работу в 120 человеко-лет за три года, но не сможете найти больше 20 квалифицированных специалистов, ваши 120 человеко-лет могут потребовать целых 6 лет. Компании по производству программного обеспечения сталкиваются с той же проблемой, поэтому передача кому-нибудь контракта на разработку не гарантирует вас от встречи с этой ситуацией.

Поскольку средства, используемые для программирования, очень сильно влияют на производительность труда, от них также сильно зависит успех или неудача проекта. В предыдущих главах мы уже обсуждали этот вопрос.

Среди самых основных инструментов можно упомянуть вычислительные машины, формирующие рабочие программы, программное обеспечение, выполняемое на этих машинах, людей, разбирающихся в вопросах поддержания рабочего состояния инструментальных средств, системы тестирования, а также механизмы управления конфигурацией.

Часто приходится сталкиваться с проблемой недостаточной мощности центрального процессора, на котором должно работать инструментальное программное обеспечение. Этот недостаток приводит к увеличению времени ожидания решения и снижению производительности труда. Преодолеть беды, вызванные плохим инструментарием, удается редко; графики из-за него срываются, а затраты увеличиваются.

Тем, кто отваживается первым использовать новый язык программирования или операционную систему, это обычно очень дорого достается.

Купить или сделать

Наиболее ответственным можно считать решение купить готовое программное обеспечение или построить его самому. Обсудим сначала само это решение и последствия, к которым оно приводит, а затем посмотрим, из каких компонент оно состоит. (См. рис. 6.23.)

Выбор — покупать или строить самим — в первую очередь зависит от наличия людей, способных вести разработку программного обеспечения. Если в моей организации таких людей нет, то мне ничего не остается, как решиться на покупку. Но, даже если такие люди найдутся, перед нами может встать необходимость решить, какие именно разработки им поручить.

Вообще говоря, если можно приобрести стандартный пакет, который сможет работать на моей системе, следует его купить. Это может сэкономить скудные, быстро становящиеся еще более скудными ресурсы разработчиков программного обеспечения. И, несмотря на большие затраты на аппаратуру, такое решение обычно оказывается приемлемым.

Рис. 6.23. Покупать или создавать программное обеспечение самому.

Стандартный пакет имеет и свои достоинства, и свои недостатки. (См. табл. 6.3.) Но, невзирая на некоторые минусы, при малейшем шансе на успех мы должны использовать стандартное обеспечение.

Таблица 6.3 “За” и “против” стандартных пакетов

Преимущества	Недостатки
Доступен сразу же	Универсален и, значит, не очень точно подходит
Модифицируется и исправляется поставщиком	Тратит некоторые ресурсы ЦП
Позволяет использовать разработчиков программного обеспечения на других работах	Не точно соответствует данной прикладной области
	Зависит от некоторой посторонней организации

Разрабатывать самим или заказывать на стороне

Если программное обеспечение нужно разрабатывать, должны ли мы делать это сами или можно заказать его на стороне? Посмотрите на табл. 6.4 — в ней перечислены все “за” и “против”.

Таблица 6.4. Разрабатывать программное обеспечение самому или заказывать на стороне?

Разработка внутри	Заказ на стороне
"За"	Сохраняются людские ресурсы
Улучшается контроль	Разработку ведут более квалифицированные и опытные специалисты
Уменьшается стоимость	Затрудняется процесс сопровождения
Создается готовая группа сопровождения	
"Против"	
Требует значительных людских ресурсов	

Как заказывать разработку программного обеспечения

Прежде всего необходимо *хорошее и ясное* описание того, что важно для вас и вашего контрагента. Если для составления такого описания вам нужна помощь, не скупитесь на расходы Она сохранит вам в дальнейшем много денег. Затем вам нужно записать ваши предложения с указанием цен, методов, проекта, расстановки людей и сроков действия контракта. Мы обращаем ваше внимание на необходимость знания точной расстановки людей. Требуйте, чтобы вас познакомили с теми, кто будет работать по вашему контракту. Наконец,, вам понадобятся устные предложения от некоторых выбранных вами людей.

Здесь вам может потребоваться знакомство с руководителем программирования — причем самого высокого ранга,— который будет полностью отвечать за программирование. От него полностью зависит ваш успех или неудача.

Насколько полно сумеет разобраться в поданных ему устных предложениях руководитель группы программистов? Сколько из них будут им приняты? Достаточно ли он решителен в своих поступках? Четко ли формулирует решения? Обладает ли он всеми необходимыми знаниями? Уклоняется ли он от вопросов или старается найти на них ответы? Необходимо тщательно изучить все черты руководителя группы программистов, уже перечисленные нами ранее. Если *у* вас возникают затруднения, попробуйте задать несколько неприятных вопросов и обратите внимание и на реакцию на них, и на ответы.

Конечно же, вы захотите оценить всю выбранную вами организацию, ее высшее руководство и его отношение к вашему заказу. Ввиду нехватки программистов очень большое значение для успешного ведения дел по вашему проекту приобретает поддержка высшего руководства, тем более что эта организация занимается не только вашими работами.

Присутствует руководитель на устных обсуждениях? Какие *именно* решения там принимаются? Записывайте все устные предложения. Это обычно заставляет людей быть более аккуратными в своих высказываниях.

Достаточно ли велика выбранная вами организация, чтобы ваш заказ был выполнен? Есть ли у нее резервы как людские, так и

материальные, которые можно пустить в ход в кризисной ситуации? Выполнялись ли в ней ранее подобные работы?

Не путаются ли эти люди в ваших каверзных вопросах? Если вам удастся их запутать, остерегайтесь. Вы выбираете не тех, кто будет потом торговать готовой продукцией, а тех, кто должен руководить ее разработкой. Только после этого выбора можно переходить к обсуждению проекта, условий соглашения и других аспектов подаваемых предложений.

Кто еще входит в руководящую группу вашего заказа? Достаточно ли они свободны для этого? Заняты ли они еще в каких-нибудь проектах? Не приглашают ли их еще в какие-либо проекты? Или им поручается только ваш проект?

Действуют ли в этой организации стандарты программирования? Они опубликованы? Когда это было? Подвергались ли они изменениям? Пользуются ли ими? Достаточно ли они надежны? Или они представляют собой не более чем словесные упражнения и их знают только понаслышке?

Какие методы управления используются в данной организации? Знаком ли с ними руководитель работ по программированию? Применяются ли они на практике? Имеются ли реальные примеры применения этих методов?

Осведомлен ли руководитель работ по программированию о возможных ловушках, имеющихся в вашем проекте? Знает ли он, что наибольшую опасность представляет ошибка в определении требований? Остерегайтесь навязывать ему свое собственное мнение по различным вопросам.

Не происходит ли так, что, соглашаясь на ваш заказ, эта организация одновременно отказывается подписать более тесное письменное соглашение? Хорошо бы, это было так. Это показывает, что там имеются опытные люди, разбирающиеся в вопросах крупномасштабных разработок. Излишняя услужливость часто возникает от простого невежества или, что еще хуже, от стремления к собственной выгоде.

После подписания контракта спеси у вас должно поубавиться. Ваш подрядчик становится вашим партнером, а избавиться от партнера можно только с большим трудом. Вместе с организацией-разработчиком вы отныне составляете единую команду. Вы почти полностью теряете возможность воздействовать на подрядчика. Смена подрядчика, судебное преследование, простое обращение в суд — все это стоит больших денег.

Один из способов сохранить возможность воздействовать на события — ежемесячно звонить руководителю подрядной организации и предлагать ему вместе с вами еще раз обсудить проект, а может быть, просто позавтракать.

Это послужит стимулом для обсуждения проекта внутри подрядной организации на высоком уровне, что должно благотворно сказаться на всем проекте в целом. Руководитель проекта может злиться по поводу того, что вы обращаетесь к его начальству помимо него самого, но дело стоит того.

Все эти методы я постарался суммировать в табл. 6.5.

Таблица 6.5. Как выбирать подрядчика

1. Необходимо собрать устные и письменные предложения
2. Крайне важна сама форма заключаемого контракта
3. Значение имеет не подрядная организация, а личность руководителя программистской группы
4. Стандарты — есть ли они у подрядчика? Опубликованы? Придерживаются ли их?
5. Достаточно ли людских ресурсов?
6. Назначения на руководящие должности
7. Опыт предыдущих разработок
8. Способность к напряженной работе и резервы

Вид заключаемого контракта

Единственным способом, который может применяться при заключении контракта на разработку крупных систем типов III, IV или V, нужно считать способ, при котором сумма контракта первоначально не фиксируется. Из этого правила может быть одно исключение, о котором будет сказано чуть позднее. Контракт без начальной фиксации суммы практикуется в Соединенных Штатах уже несколько десятилетий как основной вид контрактов на разработку. Он выдержал проверку временем, поскольку оказался выгодным как подрядчику, так и заказчику. Он используется в тех случаях, когда заранее не удается установить все необходимые требования и методику разработки в рамках жестко заданной суммы. Тем единственным исключением, которое может встретиться при заказе разработки систем типов III, IV или V, оказывается случай, когда строится в *точности* такая же система, какая в настоящее время уже используется на практике. Такое случается, хотя и редко.

Я не советую моим читателям настаивать на фиксации суммы заказа в тех случаях, когда требования не определены точным и исчерпывающим образом. Это может отпугнуть наиболее ответственных разработчиков и открыть путь для привлечения неопытного персонала или похуже того.

Что делать, когда все идет прахом

Когда нам становится ясно, что с разработкой программного обеспечения случилась беда, первое, что мы хотим узнать, — почему. Чаще всего причиной оказываются неверные предварительные оценки, довольно часто — нечетко установленные требования, а иногда в основе неприятностей лежит неправильное руководство. Мы обсуждали вопросы, как обращаться с требованиями и как проводить оценки.

Сменить руководителя легко, но это не всегда приводит к хорошим результатам. Это всегда большая неприятность. Новый руководитель часто бывает не лучше прежнего. Бывают, однако, случаи, когда другого выхода нет. Мне приходилось сталкиваться с руководителями, имевшими великолепные послужные списки, которые в результате каких-нибудь жизненных неприятностей тем не менее теряли свои прекрасные качества и не могли работать так же, как раньше. Иногда они просто достигали согласно принципу Питера, своего уровня некомпетентности, так что столь большая работа оказывалась им не по плечу. Удаление руководителя всегда оказывает сильное воздействие на всех остальных сотрудников. “Если это случилось с ним, значит, и со мной может произойти то же самое”. Решаться на это нужно в самую последнюю очередь.

Прежде чем идти на крайние меры, попробуйте укрепить силы, обратившись за помощью к руководителю проекта. Подключите руководителя группы определения требований. Руководителя группы проектировщиков. Подключите даже руководителя производством! Иногда — и весьма часто — это помогает наладить дело.

Когда же этих мер недостаточно, то мы меняем руководителя, при этом хотим сделать правильный выбор. Нам совершенно не хочется возвращаться к такому же положению еще через год и снова делать перестановки на руководящих должностях.

Поиски замены для руководителя

Вероятность того, что проект, в котором заменяется руководитель, провалится, очень велика. Один руководитель в нем уже “поработал”; после этого остались очень серьезные проблемы. Кому захочется влезать в эти неприятные вопросы? Всякий, кто обладает высокой репутацией и блестящим послужным списком, знает, что это рискованно, вряд ли может привести к успеху, вообще полно подводных камней, что это положение чревато 80—100-часовой рабочей неделей, повышенной нервозностью, трениями и явной борьбой в коллективе. *Новички* могут воспринимать его как возможность составить себе имя, выдвинуться наверх, но ведь они — *новички,* т.е. люди непроверенные и неиспробованные. В настоящее время существует очень мало хороших, полностью оправдавших доверие руководителей разработкой программного обеспечения. Проблема эта не проста.

Неуправляемый гигант

Огромный, неуправляемый проект подобен глухому омуту, в который бросаются очертя голову! Полный хаос! Сотрудники спят на работе, они почти помешались, стали вспыльчивыми, ссорятся, увольняются с работы — им предстоит прыжок в омут,— но они со всей очевидностью неспособны ничем управлять, кроме самых непосредственных, неотложных дел. Это случается *нередко*. График выдерживается только благодаря исключению некоторых функций. Это приводит к тому, что *операторы* должны выполнять большинство этих функций, только основываясь на собственных ощущениях, собственных рассуждениях и предположениях. Но работа движется! Проект объявляется успешно завершенным. А иногда случается так, что проект не работает, его приходится откладывать на год, а то и вовсе закрывать. Многие проекты были закрыты после того, как суммы затрат в них превысили *50 млн. долларов.*

Иногда же, несмотря на успешное завершение, программное обеспечение оказывается в таком беспорядке, что его приходится переделывать. Группа разработчиков начинает работу заново непосредственно после сдачи программной системы.

Стандарты программного обеспечения

Стандарты программного обеспечения необходимы группе разработчиков для всех крупных программных систем. Множество иерархически структурированных стандартов программирования нужно любому единому коллективу; это позволяет не вводить неоправданных ограничений. В каждой крупной организации должен быть директор по программному обеспечению. Все стандарты должны быть документированы, разосланы и введены в действие с проверкой исполнения. Существенную роль играет постоянное их изучение.

Если какая-нибудь достаточно крупная группа программистов не следует этим правилам, она не может гарантировать постоянно высокий уровень качества своей продукции. Случайные успехи, конечно, возможны, поскольку некоторые руководители все же знают, как надо работать, но надежной работы ожидать не приходится. Качество разработки программного обеспечения можно повысить, если обратиться к опыту технических дисциплин.

Методы, ставшие общепринятыми во всех других отраслях, в программном обеспечении кажутся абсолютными новинками. Фирма IBM выдала одному из своих служащих премию в 75 тыс. долларов за идею проведения обзоров состояния дел по проводящейся разработке с привлечением коллег и руководящих лиц — за так называемый "сквозной контроль", а ведь в промышленных отраслях этот процесс — обычное дело.

В практике разработки программного обеспечения наблюдается столь быстрый прогресс, что многие организации, занимающиеся этой разработкой, безнадежно отстают от него. Для внедрения среди авторитетной группы программистов правильных, коммерчески обусловленных методов требуются и денежные вложения, и строгий контроль, и сильное руководство.

Стандарты должны иметь иерархическую структуру. Стандарты для аппаратно-интенсивного программного обеспечения могут быть менее строгими, чем стандарты для обеспечения программно-интенсивного.

Значительную роль в деле повышения возможностей использования скудных ресурсов разработчиков программного обеспечения играют последовательная терминология и последовательно составленный набор руководящих документов.

Приведенное ниже множество "стандартов" (табл. 6.6) установлено для крупных программных разработок. У каждого правила могут быть свои исключения, но в целом его следует принимать за основу в начале работ. Введены ли они в действие? Опубликованы и им уже начали следовать? Понятны ли они? Изменялись ли? Кем?

Стандартные разделы программного обеспечения (например, операционная система) вошли в нашу повседневную практику обработки данных, и, хотя мы уже не рассматриваем их в качестве стандартов, они оказывают именно такое действие — определяют стандартные способы выполнения заданий и стандартное деление работ, выполнение которых проводится с их помощью.

Таблица 6.6. Методы производства программного обеспечения

1. Программы операционной системы	Стандартное разделение программного обеспечение
2. Программы системы управления базой данных	
3. Программное обеспечение системы связи	
4. Программы ввода/вывода	
5. Программы работы с дисплеями	
6. Программы информационной системы	
7. Использование управления конфигурацией	
8. Обеспечение необходимого качества	
9. Документация, описывающая функцию программы по подпрограммам	
10. Диаграммы, отображающие взаимоотношения оборудования и программ, с обозначением внутренних и внешних потоков данных	
11. Языки высокого уровня	
12. Использование анализа и оценок необходимых ресурсов (ЦП, память)	
13. Использование структурного программирования	
14. Размеры модулей небольшие; функциональное разграничение ярко выраженное, обязательное упрятывание информации	
15. Список ограничений, возникших при проектировании, и принципов построения проекта	
16. Частый сквозной контроль	
17. Использование библиотекарей	
18. Внесение исправлений в рабочие и исходные программы	
19. Использование средств автоматизации сопровождения	
20. Проектирование высокой производительности системы	
а) функциональной: доля удовлетворенных требований	
б) технической: точность, методология, проверка алгоритмов	
в) операционной: восстанавливаемость после сбоев	
г) удовлетворение временных ограничений на производительность	

Важнейшей причиной применения системы управления базой данных должно быть стремление автоматизировать процесс внесения исправлений и сэкономить силы программистов.

Ничто не дается бесплатно — средства на стандартизацию тратятся с первых же шагов

Из-за того что мы настойчиво придерживаемся стандартов, фаза разработки программного обеспечения может потребовать больше времени и будет стоить немного (а может быть, и значительно) дороже. На фазе использования могут произойти подобные вещи. Мы будем вынуждены тратить время на выполнение дополнительных команд, вставленных нами в программу для обеспечения модульности и лучшей читаемости. Нам потребуется несколько больше памяти. Зачем нам это все надо? Для фазы продолжающейся разработки. В этой фазе мы сэкономим столько, что оправдаем все расходы на предыдущих двух.

Склонность к фантазированию

Наша отрасль еще столь молода, что все новейшие фантазии воспринимаются в качестве новых великолепнейших методов, способных решить множество проблем. Возможно, этому способствуют средства массовой информации, стремящиеся повысить тиражи журналов и поднимающие шумиху вокруг *нового метода.* Возможно, на это влияет и желание практиков — или исследователей — увидеть свою фамилию напечатанной под статьей, говорящей, что то-то и то-то можно считать *пробным критерием* успеха.

Вероятно, действуют все эти причины. Но мудрый руководитель помнит, какая участь ожидает первопроходцев, и ждет, пока очередное новшество не будет опробовано кем-то другим и зарекомендует себя, и только потом начинает применять новинку на практике.

Репутацию фирмы-производителя нельзя считать мерой надежности. Сколько пользователей пострадало, когда фирма IBM испортила первую версию системы OC/360? Системы реального времени для модели 67? Программное обеспечение для Series 1?

Сети вычислительных машин, универсальные компиляторы и распределенная обработка данных — вот лишь немногое из списка фантастических замыслов, прорвавшихся в промышленное производство.

Часто самое высшее руководство бывает склонно к самым диким фантазиям. А представители другой крайности — технический персонал, программисты, отказываются применять новые методы на практике. Но программистов надо ограничивать именно в их собственной области деятельности! В противном случае, они, подобно детям, играющим в кубики, начнут с увлечением строить очаровательные, запутанные, закрученные и неразборчивые конструкции. Они будут писать вложенные циклы на трех страницах!

Цикл, размером в три страницы, это логический кошмар. Это еще более сложное образование, чем

Литота — двойное или множественное отрицание, как в предложении Гарольда Ласки: "Я до конца не уверен в том, будет ли правдой утверждение, что Мильтон, который прежде не казался не похожим на Шелли семнадцатого века, не стал, в противовес опыту даже более горьких лет, более близким основателю иезуитской секты, которая вряд ли способствовала развитию в нем терпимости" *{David Hackett Fischer, Historians Fallacves — Towards Logic of Historical Thought (New York: Harper & Row, Publischers INC., 1970).}*.

Разумеется, циклы размером в три страницы могут прекрасно работать. И, если речь идет о программах типов "зубочистки" или "молотка", проблемы не возникнет. Программиста можно за это похвалить. Но если нам придется несколько раз возвращаться к этой программе и модифицировать ее, лучше сразу покончить с программистом. Прежде, чем мы сможем модифицировать и расширить эту программу, нам придется распутать невероятный логический клубок.

Сопротивление нововведениям

Программисты и их руководители очень часто сопротивляются внедрению новых идей. От них одно беспокойство. Выполняющим работу людям более по душе известные, небрежные и поспешные методы, чем методы, вносящие в процесс работы некоторую

строгость. Люди хотят быть не механическими исполнителями, а волшебниками.

Обычно сама идея введения стандартов на программное обеспечение отвергается руководителем разработки программного обеспечения. Отказ от стандартов обычно сопровождается такими высказываниями: “Я уже делал нечто подобное”, или “Мы не нуждаемся в подобных накладных расходах”. Такой начальник просто отстал от жизни. Дело здесь осложняется тем, что он все же остается руководителем! Теми или иными средствами он выполняет свою работу. Он стал боссом, лидером, добился успеха, а эти новые методы очень странны, они таят в себе опасность. Они могут лишить его действия некоторой таинственности, которой они раньше обладали. Очень вероятно, что благодаря этим новым методам ему с новой силой придется вести конкурентную борьбу за руководящую должность.

“Высокое начальство” мало что знает о всех этих новшествах. Даже если есть выбор, он предпочитает соглашаться с руководителем, ретроградом или сторонником новых идей! И может статься, что применение новейшей технологии будет отложено до лучших времен.

Сопротивление новшествам — это не всегда плохо. Мы уже видели, что в этой *новой* отрасли имеется тенденция принятия осторожных решений. Предусмотрительное руководство не стремится к поспешным решениям, связанным с принятием новейших методов, разработанных сравнительно недавно Но предусмотрительное руководство не отвергает новые, *проверенные* методы, которые уже устоялись и использовались в течении двух или трех лет

Рассуждая о значении слов, Шалтай Болтай сказал: “Вопрос в том, кто за это отвечает?” Его слова вполне могли относиться к разработке программного обеспечения — кто за это отвечает? Руководство или подчиненные? Из-за новизны нашей отрасли, малочисленности практических работников ключевые решения часто принимаются подчиненными.

Можно ли разрешить сумасшедшим управлять психиатрической лечебницей? Для внедрения новых проверенных методов необходимо жесткое руководство. Старые методы сопротивляются новым. Эти новые методы пугают всех. Кому нужны эти нововведения? Они еще не проверены. “Начальство в этих вещах не разбирается!” (См. табл. 6.7.)

Таблица 6.7. Естественная борьба между программистами и их руководством

Программисты хотят	Руководство хочет
Красоты в программах	Высокой производительности программ
“Чистых” решений	Понятных решений
Сложности	Простоты
Напряженности	Легкости использования программного обеспечения
Артистичных решений	Легкости модификации программного обеспечения

Изменения дорого обходятся с самого начала

В начале 1970-х годов мы издали приказ, по которому все новые программы, создаваемые с участием 4400 сотрудников Центра федеральных систем фирмы IBM, должны разрабатываться с применением методов структурного программирования. Мы обнаружили, что должны переучить примерно 2600 программистов и их руководителей. Это означало 5200 человеко-недель или 100 человеко-лет напряженных усилий, не считая планирования, затрат на подготовку преподавателей, материальное обеспечение.

Мы создали управляющий центр, где один раз в неделю проводились отчеты с участием директора школы структурного программирования, на которых обсуждались вопросы распространения опыта, планы, графики, ресурсы и потребности. Мы организовали консультации для помощи в работе над проектами, в которых применяли новые методы. Для введения столь широкомасштабных изменений одних лекционных занятий недостаточно.

После этого мы провели прослушивание всех проектов, что позволило нам убедиться в том, что новыми методами действительно пользуются. Это был огромный труд, без которого мы могли бы только на словах агитировать за переход на технологию структурного программирования. Оправдались ли эти затраты? Безусловно. Внедрить структурное программирование, как и любую другую технологию, непросто. На это нужны и деньги, и люди, и ресурсы, и самоотверженность, к тому же приходится преодолевать неистовое сопротивление со стороны практических работников.

Неискренние заверения. Сегодня никто не станет возражать против структурного программирования. В начале 1970-х годов, как мы уже видели, такие люди были, мы уже сталкивались с ними. Но и теперь многие организации, заявляющие что применяют методы структурного программирования на самом деле этого не делают.

Разный смысл слова “структурный”. Структурное программирование так сильно обогатило профессию программистов, что я не решаюсь привести хотя бы один довод против него. Однако я должен сказать, что структурное программирование — это средство для программирования в небольших размерах. Его помощь при проектировании системы в больших масштабах невелика. Оно делает программирование более наглядным и более управляемым по многим уже перечисленным причинам, но наибольшая помощь оказывается им в проектировании и при собственно программировании.

Нам следует быть осторожными и не путать методы, применяемые в одном из разделов разработки программного обеспечения, с методами, используемыми в других частях этого процесса. Люди, занимающиеся торговлей программной продукции, приклеивают слово “структурный” к любому применяемому методу. Раз метод “структурный”, товар будет продан.

Структурное проектирование не имеет ничего общего со структурным программированием. Оно подчиняется законам структурного программирования не в большей степени, чем законам программирования по всем другим методикам.

В моду входит термин “структурные требования”. Он ничего общего не имеет с терминами структурное программирование и структурное проектирование. Сейчас уже появляется структурная документация, структурный английский язык и т.д. и т.п.

Если мы можем точно *определить* данный термин, как это сделали для структурного программирования Милс, Лингер и Уитт, то *может быть* этот термин и окажется полезным. В общем случае, все это отдает торгашеским духом, о чем можно только пожалеть — ведь у метода наверняка есть свои достоинства, просто его не надо называть “структурным”.

Все это говорится вовсе не для того, чтобы бросить на новые методы тень. Многие методы очень хороши. Но их похожие названия требуют некоторой совместимости, которой на самом деле нет.

Разработка или продолжающаяся разработка как наиболее дорогостоящая фаза

Полная стоимость системы программного обеспечения часто складывается всего из двух частей — стоимости первоначальной разработки и стоимости продолжающейся разработки. (См. рис. 6.24)

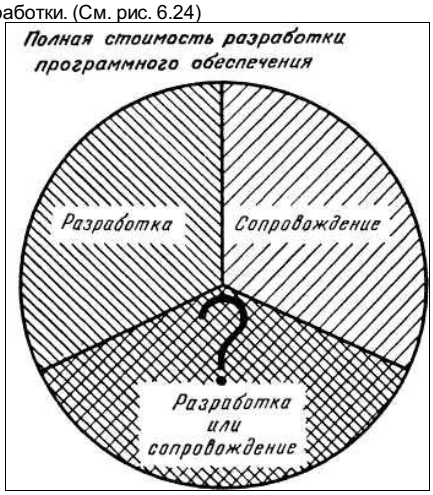


Рис. 6.24. Общая стоимость разработки программного обеспечения

Продолжающаяся разработка (сопровождение) может составлять большую часть полной стоимости — до 70 или 80 %. Но может быть и так, что ее доля будет очень небольшой. Такие колебания зависят от нескольких факторов. Наиболее важными среди них

оказываются продолжительность срока использования программного обеспечения — 1 год или 20 лет; стабильность внешнего окружения, влияющего на программное обеспечение; а также качество программного обеспечения, достигнутое во время разработки.

Мне не раз демонстрировали схемы, на которых указано, что “70% затрат на программное обеспечение было сделано при его сопровождении”. Это чрезмерное упрощение! Для очень большого числа программных систем это просто неправда.

Одна причина оптимистических оценок

Почему наши оценки столь оптимистичны? Часто мы просто боимся, что узнав наши настоящие прогнозы, начальство остановит работы над проектом. И мы выдаем оптимистические планы, основанные на том, что все пойдет без сучка, без задоринки. Получив их, руководство скорее всего просмотрит их поверхностно, поскольку оно мало что понимает в программном обеспечении. А мы будем продолжать нарушать графики и перерасходовать бюджет.

Все это, конечно же, относится и к аппаратным проектам! Но может быть в программном обеспечении число серьезных проблем превышает нормальную дозу, которая имеется во всех сложных областях человеческой деятельности? Да, это связано с новизной отрасли, а следовательно, с отсутствием в ней должного уровня дисциплины, применением новейших методов. Добавим сюда высокий уровень абстрактности.

Когда опасность угрожает всему проекту в целом, когда его пытаются отменить из-за слишком больших затрат, разработчики с новой энергией выдают новую порцию оптимистических прогнозов. Программное обеспечение подвергается наибольшим переделкам, наверное потому, что его мало кто понимает. Уменьшить стоимость спутника на часть, которая относится к программному обеспечению, представляется часто наиболее очевидным решением.

Научные исследования в программном обеспечении

Определение требований и проектирование с начала 1950-х годов и до конца 1960-х имело целью помочь программисту. С начала 1970-х годов была поставлена цель облегчить руководство программным обеспечением. Определение требований и проектирование программного обеспечения должно оказывать помощь руководителям системы, особенно в части определения требований и проектирования ее развития

Помощь стала оказываться на самых начальных и конечных этапах процесса разработки. Для проектирования были созданы средства записи алгоритмов с помощью автоматов с конечным числом состояний, языки проектирования, формальные грамматики, абстрактные машины, исследовательская работа переместилась теперь в область определения требований. Начало этому уже положено, но многое еще предстоит сделать.

Основные денежные средства, которые будут направлены на научные исследования в программном обеспечении в последующие несколько лет, необходимо использовать именно в этих передовых областях процесса разработки.

Слишком много средств направляется в область развития языков. Происходит это потому, что в настоящее время эта область получает наибольшее число голосов при распределении средств — у нас развелось очень много экспертов по языкам программирования.

Отсутствие методов представления программ

Программа — это список команд, которые заставляют вычислительную машину выполнять некоторую работу. Программа может существовать в статической форме — будучи написанной или напечатанной — или в динамической форме, как состояние электронов в устройстве памяти Различие это существенно.

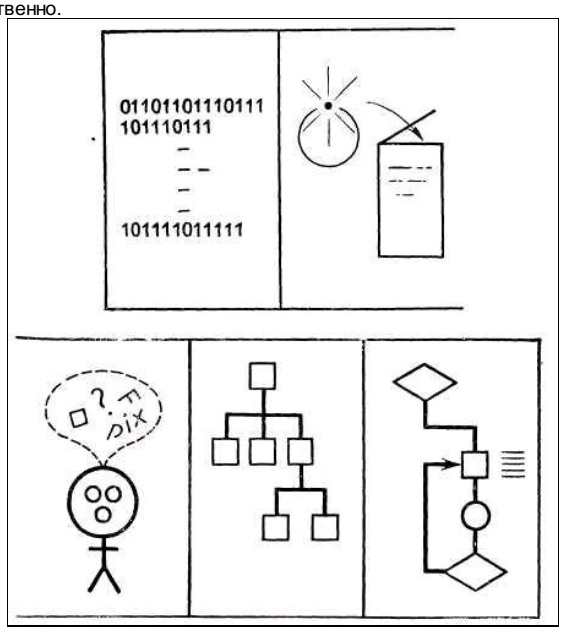


Рис. 6.25. Различные воплощения планов и программ. Вверху: Версия программы на машинном языке и в памяти машины Внизу. Та же версия в мозгу ее создателя и в виде проекта на бумаге.

Многие предпочитают представлять себе программы как нечто статическое, при этом программы выглядят так же, как и многое другое записанное на бумагу. Они при этом приобретают некоторую осязаемость и материальность.

А другие представляют программы в динамической форме, в которой они находятся в вычислительной машине, где они постоянно изменяются и выполняются со скоростью нескольких миллионов команд в секунду. И при этом в программе меняется все: постоянно меняются данные, происходят разные события, программа переходит в другие состояния, и выполняются другие ее фрагменты.

Можно провести некоторую аналогию с серией фотографий и художественным фильмом. Напечатанная программа столь же плохо отражает действительность, сколь фотография двигателя истребителя или список его составных частей плохо иллюстрирует сам самолет, а набор статичных фотографий плохо представляет художественный фильм.

Тот факт, что программа может одновременно находиться в нескольких формах, оказывается источником путаницы. И не только источником путаницы для новичков, но также источником ошибок в реальных системах, в которых одна форма может подвергаться модификациям, в то время, как другая не изменяется. Через некоторое время при этом путаница может достигнуть невероятных размеров, никто не будет в состоянии понять, какая же версия программы *настоящая,* (см. рис. 6.25).

Разрабатывать программы так же, как и аппаратуру?

Как же так получается, что мы научились создавать большие, невероятно мощные вычислительные машины, работающие очень надежно, но при этом испытываем огромные трудности всякий раз, когда пытаемся создать большую программную систему? Между разработкой программного обеспечения и разработкой аппаратуры имеются значительные различия. Основными среди них можно назвать такие:

1. Доходы от аппаратуры начинают возникать только на фазе производства. Этот длительный систематический процесс начинается сразу после завершения ее разработки. При *разработке* аппаратуры встречаются те же трудности, что и при разработке программного обеспечения. Разбогатеть на разработке аппаратуры нельзя.

2. Взаимосвязи программного обеспечения неограниченны. Ограничениями аппаратуры являются законы физики. Столько-то объектов могут иметь столько-то взаимосвязей. С программным обеспечением дело обстоит не так.

Фаза разработки продукции состоит именно в том, на что указывает ее название — чтобы достичь желаемого результата вносятся изменения. Это творческая деятельность, не подчиняющаяся дисциплине и предварительным прогнозам. Можно ожидать любых сюрпризов. Этот полный сюрпризов этап наблюдается и при разработке аппаратуры, и при разработке программного обеспечения.

Разработчики аппаратуры не могут предвидеть необходимое им время и требуемые финансовые затраты. Обычно на контрактах по разработке деньги теряются, возмещаясь затем после начала производства. У разработчиков программного обеспечения нет фазы производства.

Инженеры, рассматривая сорванные планы программного обеспечения, удивляются вслух, почему программисты не могут работать так, как это делают аппаратчики. Это отчасти обосновано, мы это скоро увидим, но прежде мне хочется отметить, что для крупных программных разработок такой подход скорее неверен.

Аппаратуру можно пощупать, почувствовать, посмотреть. Программное обеспечение абстрактно. Никто и никогда не сможет физически увидеть или потрогать большую программу. Можно подержать в руках распечатку или магнитную ленту, но это всего лишь одно из представлений программы.

Программное обеспечение неосязаемо. Оно еще более неосязаемо, чем художественные фильмы. Представьте себе такой фильм, состоящий из миллионов отдельных кадров, движущихся на высокой скорости через световой луч и проектируемых на экран. Просмотр пленки, полоски шириной 35 или 70 мм или серии фотографий, не означает просмотра фильма. Оценить художественный фильм можно по конечному результату, а результат — это демонстрация на экране. Еще в большей степени подобные рассуждения применимы к программе или программному обеспечению. Статический просмотр написанных или напечатанных команд, хотя и может принести пользу, но не является конечным результатом программы. Конечным результатом можно считать только работающую программу, а не статические представления.

Мы не увидим программы, если она достаточно велика, даже если тысячу раз на нее взглянем. Миллион строк текста программы умещается на двух с половиной километрах бумажной ленты! Будь мы хоть семи пядей во лбу, мы увидим только статическое представление. До своего вступления в действие, до того как она начнет выполняться, ее потоки, взаимодействия, соединения, границы и т.д. невидимы.

Выполняемая программа, так же как и демонстрируемый фильм, непрочна и неуловима. Это серия событий, происходящих во времени. Для воспроизведения программ, изучения ее отдельных частей, их модификации нам необходимы ее статическое представление.

Если бы мы знали, что программа правильна, если бы мы знали, что нам не потребуется ее модифицировать, и, если бы у нас был способ по желанию воспроизводить программу по содержимому памяти, который давал бы нам гарантию не потерять программу, нам не были бы нужны никакие статические представления программ. Для того, чтобы сделать более наглядными большие работающие программные системы, нам нужны новые методы.

Для множества команд, находящихся в памяти машины, нет никаких ограничений на количество вариаций последовательностей или взаимосвязей. Я могу переключить все, что хочу, куда только захочу. Я имею *абсолютно полные возможности по изменению взаимосвязей.* А так как я еще могу считывать в память дополнительные команды с диска или ленты, у меня практически нет никаких ограничений на количество команд. Последовательности команд в миллион строк становятся в современном мире обычными.

Тирания физических ограничений на взаимосвязи исчезла. Вплоть до появления запоминаемых команд, машинам объясняли, что они должны будут делать, с помощью их собственной физической конструкции. Если у этих машин были возможности по организации условных переходов, их можно было называть специализированными вычислительными машинами. Джон фон Нейман в меморандуме 1946 года указал, что до появления универсальных вычислительных машин, их предшественники "*получали инструкции для работы с помощью конструкции."*(Курсив мой.)

Физические ограничения запрещали многие возможные ветви или последовательности. Рояль можно считать аппаратурой, у него имеется 88 клавиш. Ноты — это команды пианисту, следуя которым он производит музыкальные звуки, а число различных сочетаний нот неограниченно. А понятие "правильности" весьма субъективно. То, что приятно одному, режет слух другому. Но все это — музыка.

Можно привести и другую аналогию — со словарем (аппаратурой) и романом — представляющим собой взаимосвязанные слова. Словарь точен и ограничен, даже если он очень большой. Но с помощью ограниченного числа слов мы можем написать команды для выполнения любых действий или создать неограниченное число произведений художественной литературы (см. табл. 6.8)

Инженеры учатся строить сложные машины; программисты же не всегда обладают техническими знаниями и не знают множества передовых идей по управлению сложностью. Разработчики программного обеспечения часто передают разработку сложных систем в руки людей, не имеющих никакого опыта в данной области. Такое положение нужно исправлять. Для построения систем управления процессами, нам совершенно необходимы руководители, владеющие техническими дисциплинами.

Программное обеспечение — это молодая область человеческой деятельности. Практический опыт накапливался не так уж долго, а люди, ведущие практические работы все еще продолжают учиться, как надо управлять этой областью. Терминология наша еще слишком скудна и запутана.

Таблица 6.8. Различия между аппаратурой и ее обеспечением

Ограниченные связи	Неограниченные связи
Аппаратура	Обеспечение
Рояль	Мелодии, музыка
Словарь	Романы, поэмы и т.д.
Пленка и проектор	Художественные фильмы
Вычислительная машина	Программы

Программное обеспечение является гибким и может быть подвергнуто модификациям. Люди модифицируют программы, но им никогда не приходит в голову начать модифицировать установленную у них аппаратуру. Аппаратура редко производится таким образом, чтобы ее было легко менять, а именно этого мы хотим добиться и добиваемся от программного обеспечения. В этом вопросе программное обеспечение существенно отличается от аппаратуры, может быть, за исключением только аппаратно-интенсивного программного обеспечения.

Вернемся к с.58, на которой мы указали наиболее широко распространенные отличия в жизненных циклах аппаратуры и программного обеспечения.

В жизненном цикле программного обеспечения отсутствует фаза производства. Иногда в его жизненный цикл включается фаза сопровождения. Эти два факта оказывают огромное воздействие на всю экономику создания программного обеспечения в современном мире.

Сходство между аппаратурой и программным обеспечением

Надежность. Для того, чтобы удовлетворить требованиям к надежности, выдвигаемым для вычислительных машин, работающих в системе Министерства обороны США, их изготовители должны несколько иначе подходить к построению, более интенсивно их проверять и использовать другие компоненты. Обычно такое повышение надежности и обеспечение работы в таких условиях удваивает стоимость машины по отношению к ее коммерческой версии. Если требуется обеспечить безопасность при сбоях, представление тех же функций стоит уже намного дороже

Космос. Если вычислительную машину собираются отправлять в космос или ставить на борт самолета, кроме понятных требований к укреплению ее конструкции, выдвигаются требования к уменьшению размеров и максимально возможному облегчению веса. И опять приходится останавливаться на выборе других компонент. И опять это приводит почти к удвоению стоимости. Стоимость программного обеспечения значительно возрастает и в том случае, когда размеры команд (и чисел) приходится уменьшать из-за того, что для их хранения выделяется недостаточно памяти.

ГЛАВА 7. НЕКОТОРЫЕ НОВЫЕ ВАЖНЕЙШИЕ ПРИНЦИПЫ ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ

Работая в области программного обеспечения, как и в любой другой новой области, легко стать жертвой непродуманных идей, на первый взгляд очень красивых и соблазнительных. Для чего людям нужна распределенная обработка данных? Для чего им многопроцессорная обработка? Не очень многие способны вообще дать определения этих понятий, еще меньше тех, кто может перечислить преимущества этих способов перед другими! Первой реакцией на все подобные соблазны должен быть скептицизм.

Когда я возглавлял общее руководство группой программистов фирмы IBM, я узнал от руководителя группы внутренней обработки данных (мы работали над составлением платежной ведомости для 15 тысяч человек, разбросанных по всей стране), что нам нужно "купить" удаленную вычислительную машину, которая будет использоваться для разгрузки центральной машины, выполняя "удаленные вычисления". Я ответил: "Прекрасно, а теперь покажите мне, сколько мы на этом сэкономим". Мы провели три совещания общей продолжительностью в четыре с половиной часа и пришли к выводу, что денег нам на этом сэкономить не удастся, более того, возникнут дополнительные затраты, но зато "может" появиться некоторая "эффективность". Эффективность эта была несущественна (мое мнение). Предложение поэтому было отвергнуто. Этот случай показал мне, что мой руководитель внутренней обработки не может

четко мыслить.

Многопроцессорная обработка и мультипрограммирование

Мультипрограммирование — это метод, применяемый в системном программировании для такого управления вычислительной машиной, при котором переключение с одной программы (например, составляющей платежную ведомость) на другую (занимающейся, к примеру, инвентаризацией) происходит без загрузки и откачки какой-либо из этих программ. Такой прием приводит к повышению уровня использования ЦП. Одновременно в памяти машины может располагаться несколько десятков программ, которые выполняются, периодически переключаясь с одной на другую. Тем самым резко уменьшается время простоев ЦП, вызванное ожиданием ответов от диска или другими действиями по вводу/выводу.

При многопроцессорной обработке к общей памяти или нескольким общим блокам памяти подключается сразу несколько центральных процессоров, которыми управляет одна операционная система. Многопроцессорность практически всегда подразумевает мультипрограммирование.

Два ЦП могут обладать и не одинаковыми возможностями. Для увеличения скорости решения задач один ЦП может делаться ведущим, а второй — вспомогательным, но при этом используются две разные операционные системы и, по нашему определению, нет никакой многопроцессорной обработки Высокопроизводительный центральный процессор может соединяться с медленным ЦП, на котором, однако, имеется много разнообразных устройств ввода/вывода. Медленный ЦП будет выполнять все функции по обеспечению ввода/вывода, оставляя быстрому ЦП заботы по выполнению выстроенных в очередь заданий. Такой метод часто называется методом присоединенного вспомогательного процессора. С появлением мультипрограммных операционных систем этот метод применяется все реже.

Производительность при многопроцессорной обработке

Если в состав многопроцессорного комплекса входят машины в 1 МКС — каждая из них выполняет миллион команд в секунду — и у нас их четыре, мы, конечно же, получим систему в 4 МКС, не так ли? Нет, не так. Когда сразу несколько ЦП пытаются обратиться к одному блоку памяти, возникает эффект блокирования. В некоторых приложениях специально формулируются требования по синхронизации, позволяющие достичь целостности данных, когда несколько ЦП пытаются обратиться к одному файлу данных или к одной физической ячейке памяти. При этом может возникать некоторое снижение производительности, измерить которое и понять очень сложно. (См. рис. 7.1.)

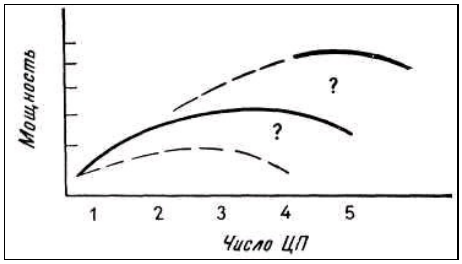


Рис. 7.1. Производительность многопроцессорной системы.

Если кто-то говорит, что такая-то и такая-то системы имеют по 16 ЦП, можно побиться об заклад, что либо 1) они работают с очень специальной задачей в крайне жестких граничных условиях, либо 2) такие системы не более чем любопытная лабораторная диковина, которая вряд ли может быть применена в настоящих приложениях. На путях внедрения 16-процессорных систем в практику универсальной обработки данных еще остается много препятствий.

Существуют, однако, и замечательные примеры применения многопроцессорных систем в системах реального времени. Система диспетчеризации авиалиний FAA имеет четыре процессора, четыре программируемых канала ввода/вывода и несколько десятков отдельных блоков памяти. В качестве многопроцессорной системы она начала работать более шести лет назад в 21 центре. Это очень дорогостоящая разработка, причем средства защиты от отказов (продолжения работы в более ограниченных условиях, когда часть аппаратуры отключается) еще до конца не запрограммированы. Но многопроцессорная обработка уже ведется.

Готовность

До появления многопроцессорной обработки надежность системы гарантировалась с помощью дублирования — две одинаковые системы ставились бок о бок, и каждая из них выполняла все задание целиком. Этот метод применялся в десятках систем и работал весьма удовлетворительно, И в военно-воздушных силах — в системе раннего оповещения о появлении баллистических ракет, — и в большинстве систем NASA по освоению космического пространства в начале 1960-х г. с успехом применялся этот метод. Его используют и сейчас, но доля его в общем числе разработок намного снизилась.

Почему? Какой же недостаток можно усмотреть в таком способе обеспечения работоспособности? В основном стоимость. Ведь дублирующая система фактически не работает до тех пор, пока не выведена из строя основная.

Для того чтобы определить, что надежнее — дублирование или многопроцессорность, разберем простой пример. Для простоты рассмотрим случай многопроцессорной системы всего с двумя блоками памяти и двумя ЦП, не обращая внимания на каналы ввода/вывода и другие детали. (См. рис. 7.2.) Единственная разница между системами состоит в том, что устройства памяти разделяются обоими процессорами (доступны им обоим). Следовательно, потеря одного ЦП в системе 1 и одного блока памяти в системе 2 не должна приводить к прекращению функционирования. Теперь нам надо представить себе, что мы хотим определить *вероятность* готовности системы к продолжению работы.

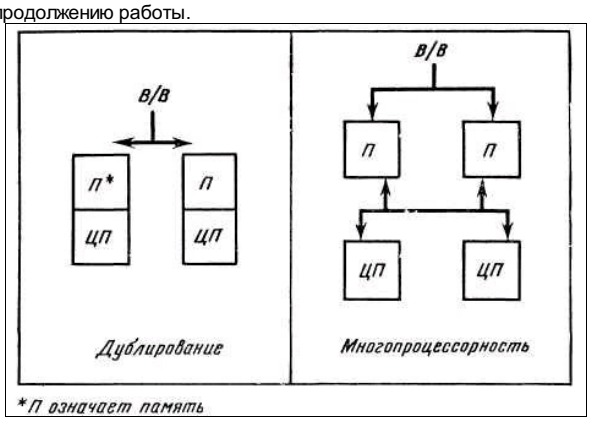


Рис. 7.2. Дублирование и многопроцессорная обработка.

Нам надо принять во внимание среднее время между отказами для каждого *устройства* и вероятность *одновременных* отказов, учесть надежность дополнительных целей, управляющих переключением с одного устройства на другое, а также среднее время ремонта каждого элемента системы.

Интуитивно может показаться очевидным, что многопроцессорная система имеет более высокий коэффициент готовности, но это не так, по крайней мере не всегда так. Нужно обязательно провести исследование и расчеты, очень длительные расчеты. В 1963 г. глубокое изучение этого вопроса, проведенное исследовательской группой IBM, созванной В. "Пирсоном (вице-президент IBM, а затем председатель отделения) для того, чтобы ответить на запрос, поступивший из FAA, привело к созданию системы диспетчеризации авиалиний Соединенных Штатов.

Технический отдел IBM горячо доказывал, что в данном случае дублирование даст более хороший результат, система будет в более высокой степени готовности. В качестве предполагаемого руководителя выполнением заказа для FAA я спорил с ними, доказывая, что многопроцессорный вариант более надежен (исходные требования от FAA прямо приводили к многопроцессорной системе). Многие часы провели мы в исследовательском комитете. К моему удивлению, было выработано решение, в котором система с

дублированием объявлялась более надежной. В тот зимний день 1963 г., проведенный в Поукипси, шт. Нью-Йорк, я просто отказывался верить своим ушам. Лирсон склонялся к тому, чтобы в IBM делали систему с дублированием. “Но ведь требования прямо ведут к многопроцессорности!” По некоторым причинам я никак не мог объяснить это Лирсону. Он спросил, все ли собравшиеся согласны с тем, что в требованиях есть предпосылки многопроцессорной системы. Все были согласны. Лирсон решил остановиться на многопроцессорном варианте. Наш проект был принят.

Причины многопроцессорной обработки

Многопроцессорная обработка обусловлена тремя главными причинами — две из них довольно простые, третья же очень сложна.

Прежде всего многопроцессорность *увеличивает мощность* (пропускную способность) на самом высоком уровне производительности, достигнутом вычислительной техникой. Если мы не способны построить более быструю вычислительную машину, мы можем объединить в одной конфигурации два ЦП.

Вторая причина состоит в возможности обеспечить *безболезненный рост,* не требуя дополнительного пространства, не срывая работ, мы получаем возможность увеличивать вычислительную мощность, убирая один из ЦП и заменяя его другим, более мощным и совместимым по программному обеспечению.

Третья причина заключается в необходимости обеспечить *бесперебойную работу.* Не просто защиту от отказов, а именно бесперебойную работу. Простая защита от отказов приводит к тому, что для каждого критичного устройства в систему включается еще одно резервное, и если происходит отказ, то к линии подключается резерв, а работа продолжается. Бесперебойная работа подразумевает наличие нескольких способов обработки кризисных ситуаций, используя в качестве резерва несколько ЦП и блоков памяти и исключая из системы после каждого очередного отказа некоторую часть выполняемых ею функций. Аппаратура для бесперебойной работы проще, а вот программное обеспечение построить для нее практически невозможно. Количество комбинаций возможных в системе отказов очень быстро становится огромным. Обработать программными средствами один отказ в памяти не очень сложно, но, если сразу же может возникнуть второй отказ, третий, мы тут же сталкиваемся с сотнями различных комбинаций, и в программах нужно отражать их все. Очень скоро программа восстановления становится бесконечно большой. Если мы остановимся только на одном уровне резервирования, как при простой защите, это еще вполне выполнимо. Если же нам требуются два уровня, на которых придется осуществлять защиту, а в нашем распоряжении десятки разных устройств, каждое из которых подвержено отказам, перед нами встает уже гораздо более сложная задача.

Целостность данных

Если я пользуюсь своим банковским счетом один и у меня есть несколько чеков, суммы которых я должен вычесть из суммы своего вклада, вычисленной накануне, я могу сделать это последовательно на одной машине. Но, если у меня два чека, один находится у вас, а другой я должен обработать сам, нам никак нельзя делать это одновременно, находясь за разными терминалами и по отдельности вычисляя новую сумму, иначе мы получим неверные результаты.

Предположим, что на моем счете находится 575 долларов и у меня есть чек на 35 долларов, а у вас — на 100 долларов. Если баланс начну подводить я, то, увидев сумму 575 долларов, я должен буду производить вычитание из нее. Если в это же время вы тоже начнете производить вычитание из этой же суммы, мы ошибемся. Я получу результат в 500 долларов, а вы в 475 долларов. Кто бы ни *написал* в основной файл новое значение *последним,* он напишет неверное значение. В данном случае не была обеспечена целостность процесса. *Мы не имеем права разрешать внесение изменений в основной файл более чем одному пользователю за раз.* Значение баланса должно быть заблокировано все время, пока кто-то его изменяет. Это справедливо и для многих других типов данных — мест расположения самолетов, медицинских сведений и т.д. Это же относится и к сложным вычислительным комплексам. Мы должны следить за целостностью данных и в многопроцессорных системах, и в системах распределенной обработки данных, и в сетях.

Как мы это делаем? С помощью системных программ.

Реализовать распределенную обработку легко; но создать распределенную базу данных, используемую в реальном времени, *очень* сложно.

Аппаратные решения этих сложных конфигураций известны уже не один десяток лет. С аппаратурой у нас проблем нет.

Проблема обеспечения использования вычислительных машин в таких конфигурациях заключается в разработке программного обеспечения для систем реального времени. В случае нормального использования вполне подходят системные программы, распространяемые на рынке. Чтобы разобраться, в чем же здесь трудность, нужно быть очень внимательным. Создать программное обеспечение, заставляющее работать такие конфигурации, нетрудно, однако очень трудно создать такое обеспечение, которое позволяло бы работать при “деградации” системы, только за счет уменьшения производительности бесперебойной работы.

Самым важным вопросом, связанным с этими сложнейшими конфигурациями, можно считать такой — *почему* нам пришлось к ним обратиться? Очень часто люди желают и даже настаивают на их применении, не имея никакого понятия о том, зачем они принимают на себя дополнительную ношу создания необычной системы. Еще больше таких пользователей, которые хотят работать на новейших, сложнейших конфигурациях, не будучи даже в состоянии сформулировать все предполагаемые выгоды, которые возникнут от этого.

Сети

Сетью называется большое число вычислительных машин — обычно географически разбросанных,— соединенных между собой линиями связи или каналов. Есть две основные причины возникновения сетей.

1. Желание организовать связь. При этом вычислительные машины играют вспомогательную роль, а сеть существует для передачи информации.
2. Желание создать сеть машин. Такая сеть нужна для вычисления, и линии связи обеспечивают возможность этих вычислений.

Очевидно, что между этими двумя типами существует некоторое перекрытие. И с ним уже десяток лет борется FCC (Федеральная комиссия по связи — Federal Communications Commission).

Стоимость сетей первого типа — сетей связи — должна определяться их задачами. Либо они себя оправдывают, либо нет. И если существует более дешевый способ управления системой, то вычислительные машины использовать нет необходимости. Экономическую целесообразность использования вычислительных сетей показать не так просто. Сеть Агентства министерства обороны США по передовым исследовательским проектам (Advanced Research Projects Agency — ARPA) можно считать великолепным техническим достижением, доказавшим, что можно объединять в одной сети множество разнотипных вычислительных машин, причем все протоколы, написанные для этого, будут верно работать. Вычислительных же сетей, действующих в частном секторе и дающих реальный доход, очень мало, если они вообще существуют. Некоторые крупные компании, занимающиеся работой в режиме разделения времени, считают экономически более выгодным использовать центральный вычислительный комплекс, состоящий из расположенных в одном месте нескольких вычислительных машин, и коммуникационную сеть, собирающую информацию для централизованной обработки. При этом можно обойтись единым руководством, одной бригадой операторов, единой охраной и т.д. Кроме экономических могут, однако, существовать и другие причины, приводящие к распределению вычислительных машин. Надежность, защита от стихийных бедствий и забастовок, исключение необходимости создания огромных сложнейших систем программного обеспечения могут оказаться вполне достаточными предпосылками.

Заказ на создание сети для фирмы General Motors

В середине 1974 г. я получил приглашение от отделения обработки данных фирмы IBM (ООД): не смогу ли я совместно со своими лучшими специалистами затратить один день на рассмотрение состояния дел в работе над предложениями, которые ООД собиралось передать в General Motors? Я согласился.

Просить помощи не в правилах ООД. Но о предложениях, готовящихся для General Motors, уже складывались легенды — их масштаб и сложность, как говорили, не имели прецедентов.

Итак, я и четыре моих лучших и самых опытных проектировщика провели целый день в ООД, пытаясь разобраться в запросе фирмы General Motors и в предложениях, готовящихся в качестве ответа на него.

Запрос *был* весьма объемистым — пачка бумаги около 15 см толщиной, — и ответ на него был весьма *объемистым* и сложным. Несколько часов мы провели, пытаясь пробиться сквозь дебри звезд, колец, пакетов,— и несколько часов, пытаясь заставить работать это огромное сборище машин стоимостью в несколько десятков миллионов долларов.

Люди из ООД подробно проинформировали нас, и нам стали очевидны два факта 1) техническая компетентность и 2) путаница в целях.

Зачем огромной вычислительной машине, выполняющей инвентаризацию, составление платежных ведомостей и планирующей производство в городе 1, взаимодействовать с большой системой разделения времени, расположенной в городе 2? И зачем этим двум машинам еще взаимодействовать с мощной машиной для научных расчетов, которая стоит в городе 3?

Потому что так требуется в запросе на систему! Может быть, для выравнивания загрузки, может быть, для разделения данных,

может быть, для...?

Ясного ответа на эти вопросы не было! Способ взаимодействия и используемые при этом соглашения зависят от того, зачем вы объединяете все эти вычислительные машины. “Как” идет следом за “почему”.

Известно, что фирма General Motors так и не подписала контракт! Несмотря даже на то, что IBM и некоторые фирмы потратили по миллиону долларов, чтобы ответить на все запросы.

Фирма General Motors не новичок в деле использования вычислительной техники. И даже этот весьма квалифицированный пользователь не избежал соблазна, исходящего от “новых” методов, “новых” использований.

Распределенная обработка?

Распределенная обработка нужна и всегда была нужна, но пользоваться ею нужно в оправданных случаях. Это не панацея и не волшебное лекарство. Всему свое место. Слишком часто все эти “новые” подходы используются вслепую.

Выводы

Все уроки и правила, извлеченные нами из всего предыдущего, продолжают действовать. Обязательность ограничений на взаимодействия распространяются и на программные модули, людей, аппаратные модули, вычислительные машины, распределенные процессоры, и на машины в сети.

Если мы не будем достаточно осторожны, оценивая размеры программ, организующих взаимодействие, то мы вскоре обнаружим, что получившаяся система 80 или 90 % времени тратит не на работу, а на это взаимодействие. Каждое подключение к сети независимого вычислителя (вычислительной машины или одиночного центрального процессора) усложняет связь между всеми остальными частями сети. Надо следить, чтобы эта сложность не затрудняла выполнение основной задачи системы — обработки данных.

Прежде чем использовать эти новые подходы, надо обязательно тщательно проанализировать все их достоинства и недостатки. Недостатки вещь реальная. Если бы их не было, подавляющее большинство используемых ныне систем были бы многопроцессорными и распределенными. Однако это не так, значит, что-то этому мешает.

Разработчик, который хочет применять самые новые и самые современные методы, просто искатель приключений.

ГЛАВА 8. ПЕРСПЕКТИВЫ

В заключение я попытаюсь дать несколько обобщающих комментариев, стараясь при этом не особенно повторяться. Но перед этим я рискну высказать несколько принципов и предсказаний.

Использование промышленных методов при разработке программного обеспечения

Разработка программного обеспечения становится все более управляемым процессом. Некоторые из наиболее мощных приемов, нашедших свое применение в последние несколько лет, многие десятилетия использовались в промышленности и вполне себя там оправдали. Вот примеры этих методов:

Работа на основе ветвящейся структуры
Руководство и управление конфигурацией
Возможность слежения
Инспекторские проверки (сквозной контроль)
Управление качеством
Верификация
Тестирование
Выборочные прослушивания
Руководство проектом

Программное обеспечение медленно теряет свое прежнее состояние невидимости, чему способствует введение (болезненное) стандартов, правил, практических методов, стандартных частей (системное программное обеспечение) и других хороших приемов руководства. Введение этих приемов приводит к первоначальному увеличению затрат, но столь значительно улучшает результаты на протяжении всего жизненного цикла, что каждый, считающий себя серьезным человеком разработчик программного обеспечения должен пользоваться ими везде, где это возможно. Я уверен, что по мере того как программное обеспечение будет становиться все более управляемым, мы будем применять еще более индустриальные методы.

Начальник исследовательского отдела корпорации Fortune — 100, занимающейся обработкой информации, недавно провел очень интересную аналогию между состоянием дел в программном обеспечении и промышленной революцией. Рассмотрим подробнее эту аналогию, из нее можно извлечь некоторые полезные уроки. Эти процессы произвели подлинную революцию в производстве.

Период разобщенности	Работа на дому. Все гончары выполняли весь цикл работ в своей мастерской.
Объединения мастеров	Достигается экономия при выполнении вспомогательных работ. Основной производственный процесс не меняется. Но теперь централизованная группа сборщиков глины поставляет материал всем гончарам. Централизованная группа торговцев заведует всей торговлей. Кувшины, однако, изготовляются по старой технологии. Этот шаг не зависит от последующих шагов.
Разделение труда	Отдельные функции передаются тем людям, которые особенно хорошо выполняют именно их. Отделка всех кувшинов выполняется специальной группой художников. В книге “Здоровье нации” Адам Смит написал, что 10 неквалифицированных работников, работая независимо друг от друга, с трудом могут изготовить за день всего по одному гвоздю, но при разделении труда те же 10 человек в состоянии изготовить более 48 000 штук. Это начало понятия о сборочной линии.
Автоматизация	Значительная экономия средств позволяет выделить фонды на внесение исправлений в основной процесс, проходящий в мастерской. Горны и печи для обжига начинают использоваться с невероятной интенсивностью. Штамповка автоматизируется с помощью использования ветряных мельниц и паровых двигателей.
Сборочная линия	Разделение всего процесса на ряд отдельных операций, выполняемых над каждым изделием в установленное время.
Взаимозаменяемые части	В конце 1790-х гг. Эли Уитни предпринял попытку изготовить за 2 года 10 000 мушкетов. “Абсурд”,— сказали ему. К 1807 г. он делал по 2000 штук в год, в шесть раз больше, чем государственный завод в Спрингфилде, шт. Массачусетс. Секрет заключался во взаимозаменяемости частей, достигнутой благодаря прогрессу в металлургии и металлообработке. Взаимозаменяемость частей стала одной из фундаментальных основ сборочных линий.

Разработка программного обеспечения все еще находится на ранних стадиях своей промышленной революции. Уже пройден этап объединения, возникли некоторые формы разделения труда, началась автоматизация, но вот взаимозаменяемости частей пока еще не достигли. (См. рис. 8.1.)

Да	До некоторой степени	Началась	Начались	Еще нет
Коллективный труд	Специализация работ	Автоматизация	Сборочные линии	Взаимозаменяемые части

Рис. 8.1. Аналогия между развитием программного обеспечения и ходом промышленной революции.

Мы еще только учимся, как надо организовать “производство” программного обеспечения. Мы одновременно заняты разработкой и инструментальных средств, и технологии. Мы движемся вперед с огромной скоростью, возможно, быстрее, чем многие могут себе представить.

Терминология

Недавно я был представлен новому помощнику секретаря комиссии по вопросам военно-морского флота, в которой я возглавляю комитет по вычислительным машинам,

“Единственное, что ненавижу больше, чем вычислительные машины, это программное обеспечение”, — сказал он мне. Его антипатия неудивительна, такая реакция обычно возникает **в** ответ на крайнюю степень непонятности программного обеспечения для большинства интеллигентных людей. Область программного обеспечения напоминает семантическое болото. Общепринятой терминологии в ней просто не существует.

Броских названий немало, и они хороши; они застревают в мозгу и легко запоминаются. Но там, где они вводят в заблуждение, они наносят ощутимый вред.

“Инвертированные файлы” — звучит очень ясно, но ясности никакой нет. Это вовсе не файлы, а просто таблицы, в которых есть сведения о файлах. В них содержатся описания логических отношений между некоторыми областями файлов. Термины “операционная

система" и "база данных" не очень информативны. "Метод доступа" — это не метод.

Как инженеры мы заслуживаем презрения со стороны людей, работающих в более традиционных областях, лишь за то, что мы даже не *пытаемся* быть точными в своих формулировках. То, что мы это знаем, но даже не желаем как-то исправить, непростительно с нашей стороны. Точность и ясность можно считать "торговой маркой" понимания.

Организационные требования

Перечислим мероприятия, которые следует проводить в каждом серьезном проекте связанном с созданием информационных систем. Программное обеспечение начинает становиться фактором, *отличающим* одну информационную систему от другой. До сих пор мы говорили об организации работ над товарными программами и программным обеспечением проектов. Какие организационные мероприятия должна проводить крупная корпорация в отношении программного обеспечения?

Что делать?

Необходимо предпринять несколько важнейших действий:

1) назначить единого директора по программному обеспечению;

2) производить в масштабе всей фирмы обучение ведущих специалистов;

3) установить стандарты на программное обеспечение;

4) запретить принятие решений на слишком низких уровнях.

Единый директор по программному обеспечению

В каждой крупной организации должен быть директор или заместитель директора по программному обеспечению. Программное обеспечение имеет огромное значение, и поддержка всех практических работников находится пока на столь недостаточном уровне, что во *всякой* крупной организации следует иметь официального лидера по программному обеспечению Организация, ведущая работы по управлению информацией, вообще не может продвигаться вперед без официального руководства работами по программному обеспечению.

Этот человек обязан:

1) установить стандарты на программы на уровне данной организации;

2) пересматривать и оценивать стандарты различных под разделений, их планы, расстановку сил, пересматривать работы по разработке программного обеспечения;

3) проводить обсуждения;

4) контролировать расходы по определению требований и проектированию программного обеспечения;

5) давать советы высшему руководству по вопросам, касающимся возможностей и направлений развития областей, на которые оказывает влияние программное обеспечение.

Это не специфически профессиональная работа, это работа не для того, кто проявил себя в разработке программного обеспечения. Такой талант должен использоваться по прямому назначению. Эта работа связана с проведением общего руководства, которое требует от человека деловой сметки, знаний в области программного обеспечения и умения принимать ответственные решения. Одной лишь квалификации в программировании здесь совершенно недостаточно.

Когда я посещаю крупную организацию, то всегда прошу познакомить меня с принятыми в ней стандартами на программирование. Я прошу познакомить меня с тем, кто следит за программным обеспечением подобно тому, как главный инженер выполняет свои функции. Слишком часто оказывается так, что ответом бывает только молчание; никто подобных функций не выполнял Я сразу делаю заключение, что в этой организации программному обеспечению не уделяют должного внимания. Очень часто это происходит в результате того, что вопросы, связанные с программным обеспечением, просто игнорируются.

Стандарты

Отсутствие общих стандартов на предприятии приводит к тому, что подходы к проблемам отдельных групп начинают все более различаться между собой. При попытке общения друг с другом эти группы сталкиваются с проблемой, возникшей еще при строительстве вавилонской башни.

Все ключевые решения, касающиеся программного обеспечения, должны приниматься на соответствующих уровнях, они должны быть обозримыми и технически выполнимыми. Отсюда следует, что на уровне предприятия нужно обязательно стандартизовать *определенные практические методы, программирования и термины.* Без стандартизации внутренняя структура программного обеспечения, используемого в рамках предприятия, не будет определена в течение очень длительного периода. Любая организация, не *настаивающая* на следовании стандартам программирования, не может серьезно заниматься обработкой информации независимо от того, знают там это или нет. В то же время нельзя допускать, чтобы стандарты принимались без должного изучения и обсуждения их на высшем уровне. Необходимо иметь стандарты предприятия, подразделения, стандарты на программное обеспечение, продаваемое на рынке, и на программное обеспечение проектов. Все они должны быть совместимыми, на могут не совпадать.

Решения на низших уровнях

Добавлять новую функцию в программу или нет, должен ли пользователь иметь возможность выбора параметров и функций, какие нужны функции, сколько, как скоро их следует запрограммировать? Все эти вопросы входят в компетенцию руководителей разработкой. И, если их спросить, они, конечно же, ответят, что принимают эти решения сами. Однако знакомство с некоторыми работами по разработке программ, выпускаемых на рынок, показывает совсем другое. Руководитель работ по изготовлению какой-либо продукции запрашивает мнение руководителя работ по ее программному обеспечению, "когда можно будет сделать то-то и то-то". В большинстве случаев руководитель разработкой программного обеспечения спускается еще на пару уровней ниже п спрашивает мнение старших программистов. Именно они *и* "решают", когда можно будет сделать это "то-то и то-то", основываясь на собственных критериях и предстоящих работах.

Почему? Как мы оказались на столь низком уровне? Потому что не было иной документации, кроме той, которую способен понять только старший программист?

Ясно, что этот случай связан с неудачей в выборе руководства, и проблема эта имеет глобальный характер. В области программного обеспечения имеет значение даже сама форма, в которой выпускается необходимая документация.

И все же руководство будет уверять при обсуждении, что все идет хорошо, некоторые проблемы хотя и существуют, но серьезной опасности не представляют. Они не пытаются скрыть истинные факты; в большинстве случаев они просто не понимают, насколько плохи дела. Положение, в котором ключевые решения, касающиеся создания новой продукции, принимаются старшими программистами, необходимо совершенно исключить из практики. Руководство просто не понимает, что в таких случаях с ним делают.

Болезненные изменения

Переориентация на другое программное обеспечение может привести к чувствительным затратам по двум причинам. Первой оказывается новизна данной области. Многие исполнители просто незнакомы с ее размерами и характеристиками. Вторая причина связана с первой, но может удивить кого угодно. Большинство людей, занимающихся практическим программированием, отстали от века! *Их* просто необходимо или направить на учебу, или уволить. Наша отрасль развивается столь быстро, что большинство руководителей разработок обеспечения используют методы 20-летней давности. Для 90% разработчиков программного обеспечения это "средневековье".

Большинство будет упорно это отрицать и сопротивляться всем изменениям. Это в некоторой степени защитная реакция ("Кто это там будет меня учить?"), и в некоторой степени она основывается на высокой стоимости повышения квалификации ("Я не могу позволить себе тратить время, деньги".). В таком последнем утверждении есть доля истины. Отсюда еще раз напрашивается вывод о необходимости иметь директора предприятия по программному обеспечению.

Затраты выражаются не только в деньгах, но также во всевозможных мучениях, которые приходится причинять ценным специалистам, обучая их странным для них методам. Многие из этих методов не более чем технические средства, позволяющие несколько снижать сложность прилагаемых усилий. Эти методы хорошо преподавать в высших учебных заведениях, в курсах по программированию, но "сложившихся" практиков они только пугают.

Обучение ведущих исполнителей

Мы вовсе не хотим сделать всех практических работников экспертами по программному обеспечению. Это и невозможно, да и не может принести хороших результатов. Но мы хотим, чтобы все исполнители имели верное представление о программном обеспечении, были способны воспринимать и высказывать суждения по вопросам, которые исходят от связанных с ними экспертов.

Эта высота вполне достижима, к тому же она весьма важна для любой компании, связанной с обработкой информации. Все исполнители, не имеющие четких познаний о программном обеспечении, подвержены опасности уйти в небытие.

Следует постоянно проводить серии семинаров и производственных совещаний различной продолжительности, предназначенных для исполнителей разных уровней подготовки. При этом обязательно нужно пользоваться стандартной терминологией, применяемой на

данном предприятии.

Прогнозы

1. Программное обеспечение станет решающим фактором, влияющим на качество продаваемых крупных информационных систем.

2. Недостаточное число программистов задержит использование “интегральных схем”. Тот недостаток, который существует сейчас, со временем еще больше усугубится.

3. Вычислительная техника станет настолько дешевой, что появятся “системные конструкторы”. Они будут связывать квазиуниверсальные программы, создавая “обрабатывающие системы”. До этого должны, конечно, появиться более четкие и аккуратные механизмы взаимодействия про грамм, но этот процесс уже начался. Получающаяся в результате система будет далека от оптимальной, но работать она будет вполне удовлетворительно. Это можно на звать *расточительством* по отношению к вычислительной машине, но никого этот факт не будет волновать.

4. Разрыв между стремительным развитием микроэлектроники и инерцией разработки программного обеспечения ста нет наиболее известным примером неодолимой силы и не движимого объекта. В результате разработка программного обеспечения *достигнет* небывалых высот. Это обязательно случится. В разработке программного обеспечения произойдет подлинная революция.

Появятся новые языки общения пользователей с вычислительными машинами; это будут языки не такого типа, как Ада и Паскаль, но языки типа “Запрос с помощью примеров”. Язык Ада чересчур сложен, и изучать его очень трудно. Пользователь с помощью некоторых приказов “заставляет” вычислительную систему добиться нужного результата — зарезервировать место, сбить ракету, проложить курс, а программист с помощью операторов языка “заставляете систему получить рабочую программу.

Но пропасть между языками высокого уровня, используемыми при программировании, и языками пользователей быстро становится меньше. Небольшое изменение в концепциях приведет к тому, что можно будет “перехватывать” эти приказы, исходящие от пользователей, и формировать по ним рабочие программы, а не просто их “обрабатывать”.

Языки будут выходить на все более высокие уровни. Пользователи станут программистами, но это слово может в дальнейшем получить несколько иной смысл, чем тот, который мы вкладываем в него сейчас.

Прогресс

Не следует удивляться тому, что разработка программного обеспечения столь трудна. Мы уже видели, что всякая разработка трудна по определению. Если бы она была прогнозируемой, она не была бы разработкой.

Снова обратимся к мостам — простым, старым, скучным мостам и вспомним, что в 1870-х г. в США падало по 40 мостов в год. Новая технология подчинялась контролю человека чрезвычайно медленно.

Программное обеспечение это не просто самое позднее из сложнейших начинаний, оно оказалось таким трудным потому, что оно “невидимо” и не может быть воспринято никакими органами чувств человека. Кроме того, оно зависит от сотен принятых решений или конструкций

Как и все новые области, программное обеспечение страдает от взаимосвязанных проблем “семантической распущенности” и засилья любителей и шарлатанов.

За последнее десятилетие в деле разработки программного обеспечения произошел разительный прогресс. Гораздо более разительный, чем кто-либо мог предвидеть. Вступили в действие огромные системы — причем работают они хорошо и надежно. Все подверглось развитию — системное программное обеспечение, структурное программирование, языки, автоматические библиотекари, кросс-трансляторы и инструментальные комплексы. Мы сделали программирование более наглядным. Это звучит тривиально, но ведь просто жизненно необходимо, чтобы разработка программного обеспечения была управляемой и контролируемой.

Структурированные программы стали доступными для чтения. Великолепнейший сдвиг.

Библиотекари ведут учет написанных программ. Программы стали собственностью не программистов, а тех организаций, в которых они работают. Появилась возможность видеть, как работают программисты. Мы можем проводить инспектирование написанных ими программ.

Сделав программное обеспечение более наглядным, мы смогли сделать его более управляемым. Мы повысили производительность труда.

Мы стали пользоваться общими программами, входящими в операционные системы и СУБД. Передача этих программ в пользование тысячам людей сделала ненужными огромные затраты на перепрограммирование.

В нашем распоряжении появились тысячи специалистов с высшим образованием, специализирующихся в области информатики, знающие теоретические основы множества новейших методов и инструментария.

Мы видим, насколько далеко мы ушли всего за 10 последних лет.

Руководство обязано не допускать “технологического тумана”. “Пойди туда, не знаю куда, принеси то, не знаю что”. Частенько еще случается, что руководители дают запугать себя со стороны технических исполнителей, пользующихся длинными, но непонятными словами. Каждый должен быть заинтересован в том, чтобы выяснить значения всех слов, используемых в программном обеспечении.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ