Введение в технологию программирования

2-е издание, исправленное

Терехов А.Н.

Национальный Открытый Университет "ИНТУИТ"
2016

УДК 004.42 (075.8)
ББК 18
Т35
Технология программирования / Терехов А.Н. - М.: Национальный Открытый Университет "ИНТУИТ", 2016
ISBN 978-5-94774-669-3

В достаточно популярной форме излагаются основные аспекты жизненного цикла создания и сопровождения программных продуктов, организации коллективов программистов, сведения о стандартах качества.

Как пример наиболее трудной, по мнению автора, задачи в этой области рассматриваются вопросы создания встроенных систем реального времени.

Понятие технологии программирования, жизненный цикл программы и постановка задачи

Понятие технологии программирования

Технология – это набор правил, методик и инструментов, позволяющих наладить производственный процесс выпуска какого-либо продукта [1]. Разумеется, это определение нельзя назвать полным. Надо упомянуть процессы планирования, измерения, оценки качества, ответственность исполнителя и многое другое. Но в мою задачу не входит написание статьи для энциклопедии на слово "технология" – я дал какое-то представление о его значении и для начала разговора этого вполне хватит. Подчеркну только еще раз производственный характер понятия "технология". Почему-то именно слово "производство" сильно раздражает моих коллег-математиков, занимающихся программированием. "Ты бы еще технологию создания романа "Война и мир" создал". Недаром же известный специалист Дональд Кнут назвал свой знаменитый многотомный труд "Искусство программирования". Наконец, мне удалось выяснить отношение к этому понятию самого Д. Кнута. Несколько лет назад наш университет присвоил этому несомненно блистательному ученому звание почетного доктора. По такому случаю наш декан собрал для встречи с Д. Кнутом десять-двенадцать руководителей лабораторий и заведующих кафедрами, имеющих отношение к программированию. Дело было летом, Д. Кнут был с женой, переводчица поминутно смотрела на часы, напоминая, что гостям пора "на фонтаны", сам почетный доктор откровенно дремал, словом, это было скучнейшее мероприятие. Я решил пошутить, чтобы как-то развеять унылую атмосферу. И когда подошла моя очередь выступать, я вместо тридцатисекундного перечисления заслуг нашего коллектива сказал, что лично мне профессор Кнут доставил много неприятностей. Тот аж подскочил: "Я же в первый раз вас вижу!" Тут я объяснил, что много лет занимаюсь промышленной технологией программирования, а многие коллеги, включая и тех, кто сидит сейчас за этим столом, ссылаясь на его книгу, не признают это наукой, имеющей отношение к математике. Профессор очень разволновался, принял мою шутку вполне всерьез и пожаловался, что все это происходило и с ним: "Вы знаете, что я много лет занимался созданием и продвижением на рынок издательской системы TEX? Это что – не

производство? Могу я на старости лет написать книжку с теоремами?"

В общем, я получил огромный заряд положительных эмоций.

Разумеется, я понимаю, что Д. Кнут известен скорее своими теоретическими работами, а TEX – это скорее от безысходности, он задумал его после выпуска первого тома "Искусства программирования", поражаясь скудости рынка инструментальных средств для набора математических текстов, прервался на несколько лет для реализации TEX и только потом продолжил свой титанический труд. Для него промышленное программирование – скорее бравада, что вот, мол, мы все можем, но все равно приятно получить поддержку от такого знаменитого ученого.

Мое первое столкновение с промышленностью состоялось в 1980-м году. К тому времени я уже несколько лет руководил лабораторией системного программирования, занимались мы самыми трудными проблемами, часто имеющими весьма отдаленное отношение к практике, но нас это нисколько не беспокоило. "Мат-мех – лучше всех" – в глубине души я и сейчас в этом уверен. Так вот, в декабре 1980-го года меня пригласили в кабинет ректора ЛГУ, там было человек двадцать в военной форме и несколько партийных деятелей во главе с начальником оборонного отдела Обкома КПСС. Мне было сказано, что оборонная промышленность столкнулась с массой проблем при создании программного обеспечения (ПО) систем оборонного назначения, поэтому решено, что ЛГУ в лице нашей лаборатории должен помочь. Моего согласия никто не спрашивал, да в те времена этого и не требовалось. Так я начал работать с ЛНПО "Красная Заря", "Импульс", "Морфизприбор", "Ленинец", "Аврора", "Гранит" – ведущими предприятиями Ленинграда, работающими в интересах различных родов войск и ведомств. С большинством этих предприятий мы работаем до сих пор.

Нас использовали как "пожарную команду". Много лет сотрудники названных и других предприятий разрабатывали ПО для важных заказов, когда же все сроки (и деньги!) истекали, приглашали нас, чтобы мы навели порядок. Нельзя сказать, что до этого мы жили в башне из слоновой кости. У нас были договора с НИЦЭВТ по созданию транслятора с языка Алгол 68 [2] для только что созданного (мы

говорили "создранного") семейства ЕС ЭВМ, так что мы одними из первых освоили новую операционную систему OS/360. Пользуясь открытостью университета, к нам все время приходили какие-то люди (в том числе и военные) с просьбой найти ошибку в программе. Не было случая, чтобы мы искали ошибку больше двух дней.

Но все это не идет ни в какое сравнение с трудностями нашего понимания проблем реальной промышленности. Одно дело, когда у тебя в подчинении пятнадцать человек в университетской лаборатории, которых никогда не надо подгонять, сомневаться в их способностях, когда царит дух соревнования (не только между собой), когда все понимают друг друга с полуслова. Совсем другое дело — триста человек с совершенно разными способностями и образованием, явно отсиживающие свои восемь часов, работающие в кодах (буквально с инженерного пульта ЭВМ), причем все это погружено в атмосферу секретности. Мне никогда не забыть, как одна женщина целый час объясняла мне схему распределения памяти, которую она вычитала у одного японского автора. С большим трудом удалось выяснить, что она имеет в виду стек. Другой программист запомнился мне тем, что для возврата из процедуры использовал регистр Р7, хотя остальные триста человек для этой цели применяли Р10. Он еще доказывал, что именно он прав, так как Р7 – индексный регистр, поэтому передача управления по его содержимому выполняется на целую микросекунду быстрее. Мне так и не удалось объяснить ему важность соблюдения соглашения о связях.

Первыми из военных организаций к нам обратились сотрудники ЛНПО "Красная Заря" с просьбой помочь в программировании широкого класса задач управления и связи, в частности, в создании функционального программного обеспечения (ФПО) телефонных станций, управляемых специализированными ЭВМ (СЭВМ). Несколько лет ушло на изучение предметной области, пробные реализации и решение организационных вопросов. У сотрудников ЛГУ был исходный принцип важности использования алгоритмических языков высокого уровня (АЯВУ), основанный на опыте предыдущей работы. Однако начинать пришлось совсем не с этого, а с повышения общей культуры программирования разработчиков ФПО. Дело в том, что в области встроенного ФПО реального времени традиционно используются СЭВМ с нестандартной архитектурой, ориентированной на заданную

предметную область (правда, неясно, в чем должна выражаться такая ориентация: например, если ЭВМ хорошо выполняет какие-то специальные операции, но плохо — условные переходы и вызовы процедур, которые встречаются гораздо чаще, можно ли считать, что данная ЭВМ соответствует предметной области?). Нестандартность архитектуры и малая тиражность таких СЭВМ приводят к отсутствию достаточно развитых операционных систем, трансляторов, средств отладки и других, ставших уже привычными, инструментов программирования. Поэтому мы столкнулись с работой на перфокартах и непосредственно за пультом СЭВМ "на тумблерах".

Мы пытались воспользоваться известными в то время технологиями, однако оказалось, что, например, Р-технология [3] не имеет никаких средств настройки на СЭВМ, а предлагаемые в ней графический стиль программирования и программа-организатор с ручным вводом мало помогают в решении задач реального времени; технология РУЗА позволяет автоматизированным образом (но с большими доработками) построить кросс-ассемблер нужной ЭВМ и интерпретатор ее системы команд, а также осуществить некоторую регламентацию работы (например, стандартизовать имена объектов). Р-технология была отвергнута практически сразу (за неимением конкретных программных средств на заданных нам СЭВМ), РУЗА в течение полугода была настроена на одну СЭВМ, однако полностью учесть все особенности СЭВМ так и не удалось; кроме того, параметрически настраиваемые кросс-ассемблер и интерпретатор замедляют работу в 5-7 раз.

Мы решили, что из-за таких простых программ, как ассемблер, не стоит "городить огород", и за короткое время реализовали новые кросс-ассемблер и интерпретатор, которые вместе с текстовым документатором и некоторыми сервисными программами составили основу первой внедренной нами в производство (1984 год) технологической системы, интенсивно использовавшейся сотнями разработчиков ФПО. Разумеется, мы отчетливо понимали ограниченность возможностей такой технологии, но ее популярность имела свои причины:

- применение вместо СЭВМ широко доступной ЕС ЭВМ с многопользовательской ОС и широкими сервисными возможностями;

- работа одновременно многих пользователей за терминалами, а не с перфокартами;
- богатые отладочные возможности интерпретатора СЭВМ, недостижимые непосредственно на СЭВМ;
- впервые осознанная разработчиками ФПО ценность документации на машинном носителе, легкость ее оформления, исправления и тиражирования;
- практически неограниченные возможности развития технологии, удивительно быстро схваченные разработчиками ФПО, в результате чего практически ежемесячно появлялись новые идеи и предложения, большинство из которых было быстро реализовано.

На протяжении всей совместной работы сотрудников ЛГУ и "Красной Зари" имело место противостояние двух позиций относительно понятия технологии программирования. Сотрудники ЛГУ, в основном, подразумевали под ним широкое использование инструментальных средств, а сотрудники "Красной Зари" настаивали на том, что технология — это, прежде всего, набор формальных методик и регламентирующих средств, позволяющих, в частности, на каждом этапе провести экспертизу, архивацию и измерение объема и качества проделанной работы. Такой подход вызывал постоянное раздражение профессиональных программистов.

Разумеется, мы понимали, что работа без четких сроков и ответственности перед соисполнителями возможна только исследовательская и экспериментальная, да и то в ограниченных объемах, но работа по конкретным заказам в промышленности дала массу новых впечатлений и импульсов для изучения. Например, уход исполнителя в самом разгаре работ. Оказалось, что в промышленности это довольно частое явление (отсюда необходимость архивации и других средств отчуждения результатов работы от исполнителя); в коллективе из нескольких сот человек всегда есть люди, попросту не желающие работать, и хотя мы понимаем, что оценка работы программиста в байтах весьма сомнительна, это не отменяет необходимости учета индивидуально выполненной работы (измерение). Даже такое "приземленное" соображение, как разделение ответственности за принятые решения и ошибки, также нельзя сбрасывать со счетов (отсюда строгое документирование).

Особенно много разночтений вызывала необходимость оформления постановки задачи по стандартным правилам. Дело в том, что аккуратное оформление требует усилий, причем не только технического порядка, а программисту кажется, что, имея в руках такой мощный инструмент, как АЯВУ, легче сразу выразить свое понимание задачи в программе, а не в каких-то таблицах и диаграммах. Но непосредственное программирование лишает программиста обратной связи с функционалистом (постановщиком задачи) и уж тем более — с заказчиком. Большинство сложных систем невозможно сдать в эксплуатацию из-за огромного количества сравнительно мелких замечаний, вызванных разночтениями и неясностями в постановке задачи.

Мы настаивали на том, что, занимаясь вопросами документирования, ценообразования, способами регламентирования и контроля за ходом работ, нельзя забывать, что основным результатом применения технологии является программа, действующая в заданной вычислительной среде, хорошо отлаженная и документированная, доступная для понимания и развития в процессе сопровождения ("нам нужны не приборы в принципе, а приборы в корпусе").

Только после двух-трех лет работы в промышленности мы осознали, что нельзя все сводить к программному инструментарию. Поначалу мы с гневом отказывались от требований начальства детально документировать, кто, что и за какой период написал, но оказалось, что в большом коллективе всегда находятся милые в общении, всеми любимые организаторы всевозможных мероприятий, которые вообще ничего не делают по работе. Первую сдачу проекта для Управления правительственной связи КГБ я завалил, так как буквально перед самой сдачей кто-то стащил одну (!) перфокарту, а эти товарищи всегда начинали с чистой машины и полной перетрансляции. Вредителя так и не нашли, зато я получил хороший урок. Мы быстро реализовали контрольно-учетные программы, архивы с контролируемым доступом, многоуровневые системы сбора версий ПО и тому подобные "шпионские штучки". Так я впервые осознал разницу между "программированием для себя" и "программированием для хозяина", о которой так красочно рассказывал академик А. П.Ершов.

В более поздних публикациях эту разницу стали выражать более

канцелярским стилем – просто программа и программный продукт.

Жизненный цикл программы

Итак, мы знаем, что программный продукт является результатом некоего производственного процесса. Этот процесс нужно спланировать, оценить ресурсы, для чего, в свою очередь, требуются более или менее точные спецификации, что же необходимо заказчику. Затем продукт надо спроектировать в виде системы, состоящей из многих компонентов, описать функции этих компонентов и их связи между собой, после чего компоненты нужно запрограммировать, автономно отладить, собрать вместе, провести комплексную отладку, подготовить документацию на систему, обучить пользователей, провести опытную эксплуатацию и организовать сопровождение системы на весь период ее эксплуатации.

Разумеется, это лишь приблизительная схема, которая может варьироваться в широких пределах, но, тем не менее, она дает представление о том, что такое "жизненный цикл программы" (ЖЦП) [4]. Почему "цикл"? Потому что редко разработка развивается столь прямолинейно, хотя одну из первых моделей ЖЦП действительно назвали "водопадная", подчеркивая тот факт, что к предыдущей фазе проектирования вернуться невозможно. Действуя по этой модели, коллектив последовательно разрабатывает проект – от исходной концепции до комплексного тестирования. Модель требует определить опорные точки, в которых будет оцениваться сделанное и решаться вопрос о том, можно ли двигаться дальше. Такой подход хорош для проектов, в которых требования легко формулируются с самого начала, но не годится для сложных, когда требования могут неоднократно меняться. Кроме того, водопадная модель вынуждает готовить огромную массу документации и требует единообразной процедуры оценки результатов на каждом этапе. Эти две особенности часто приводят к синдрому "аналитического паралича", напряженным отношениям между разработчиками, заказчиками и пользователями.

В реальной жизни, конечно же, чисто водопадная модель не применяется. Появляются новые требования заказчиков, изменяется аппаратура, находятся такие ошибки, которые невозможно исправить, не затронув результаты предыдущих фаз и т.д. Развитие системы – это

именно циклическое повторение практически всех фаз, постоянный возврат к предыдущим фазам. Если не принять специальных мер (что, собственно, и является предметом технологии программирования), процесс может стать бесконечным.

Одной из первых практически полезных моделей ЖЦП стала модель создания прототипов. С самого начала разработчики пытаются выделить основные, существенные требования заказчика и реализовать только их в виде работающего прототипа системы. Этот прототип демонстрируется заказчику. Часто бывает, что заказчик в ужасе кричит, что его неправильно поняли, он хотел совсем другого, зато теперь он хоть может внятно сформулировать свои требования, глядя на работу прототипа. Цикл разработки и показа прототипа повторяется несколько раз, пока заказчик не скажет: "Да, это, кажется, то, что мне нужно". Только после этого дорабатываются куски, выброшенные в начале разработки, подготавливается документация, короче, делаются многие вещи, на которые время было бы потрачено зря, если бы их делали для самого первого, неудачного прототипа.

Для небольших систем, особенно для тех, в которых велик процент интерактивных (взаимодействующих с пользователем) компонентов, такая модель работает, хотя каждый раз, когда я про нее рассказываю, мне вспоминается анекдот. Празднуется золотая свадьба. Набежали журналисты, спрашивают: "Как же так, все семьи разваливаются, а вы прожили пятьдесят лет?" Глава семьи отвечает: "Как только мы поженились, сразу договорились, что все важные жизненные проблемы решаю я, а жена не спорит, а остальные – решает жена, а я в них не лез. Так и прожили". Журналисты не унимаются: "А что такое – важные проблемы? Ну, например, назовите проблему, которую Вы решали в последнее время". Глава семьи отвечает: "Ну как же, как же, недельки две назад я долго размышлял, вернется Далай-Лама в Тибет или нет".

В этом анекдоте, как всегда, отображена вся правда жизни. На моей памяти много случаев, когда важные аспекты просто упускались при разработке прототипа, из-за чего позже все приходилось переделывать заново.

Некоторым обобщением модели создания прототипов является спиральная модель, в которой разработка приложения выглядит как

серия последовательных итераций. На первых этапах уточняются спецификации продукта, на последующих — добавляются новые возможности и функции. Цель этой модели – по окончании каждой итерации осуществить заново оценку рисков продолжения работ. Программисты часто увлекаются технической сутью выполняемого проекта и не видят общей картины, особенно в части производственных затрат. Нам все кажется, что вот еще немного, еще чуть-чуть, и все проблемы будут решены, но "асфальтовая топь" (по выражению Ф. Брукса[5]) засасывает нас, не давая шансов достичь твердых осязаемых результатов. Один мой знакомый бизнесмен представил эту проблему так: "Вот ты затратил 100 000 долларов, но задачу пока не решил. Нужно сто раз подумать, что лучше — истратить еще столько же, чтобы успешно завершить проект, или через год снова оказаться перед тем же выбором, но тогда уже с риском потерять не 100 000, а 200 000 долларов?"

В силу своей итеративной природы спиральная модель допускает корректировки по ходу работы, что способствует улучшению продукта. При большом числе итераций разработка по этой модели нуждается в глубокой автоматизации всех процессов, иначе она становится неэффективной. На практике у заказчиков и пользователей иногда возникает ощущение нестабильности продукта, так как они не успевают уследить за слишком быстрыми изменениями в нем.

Один очень важный вывод мы можем сделать даже при таком начальном знакомстве с понятием ЖЦП. Собственно программирование не является единственным занятием коллектива, занятого промышленными разработками. Более того, оно не является даже главным, наиболее трудоемким делом. Многие исследования отдают на фазу программирования не более 15-20% времени, затраченного на разработку (сопровождение вообще бесконечно). Может быть, эти цифры заставят вас задуматься о важности и других аспектов образования – от умения найти и обосновать эффективный алгоритм до искусства владения родным языком, как устным, так и письменным.

Постановка задачи. Оценка осуществимости

Обычно заказчик выдает две-три страницы текста задания и сразу же

просит оценить время исполнения заказа и его стоимость. Надо быть сумасшедшим, чтобы на это согласиться. Нередки случаи, когда целые коллективы ошибаются в пять-десять раз и попадают в кабалу или теряют профессиональную репутацию. Чтобы избежать такой ситуации, нужно предложить заказчику оформить начальный договор на две-четыре недели с тем чтобы два-три системных аналитика разобрались в задаче, с помощью каких-то инструментальных средств выполнили декомпозицию системы на компоненты, прикинули возможные объемы этих компонентов и, соответственно, время их реализации. Такая начальная стадия ЖЦП называется "оценкой осуществимости".

Можно, конечно, выполнить эту работу за свой счет (и многие крупные предприятия так и делают), но, во-первых, отношение к внутренним разработкам – более спокойное, а, во-вторых, оплаченный договор гарантирует серьезность намерений обеих сторон. К сожалению, часто бывает, что недобросовестные заказчики пользуются таким приемом, чтобы бесплатно получить идеи разработки или просто выполнить экспертизу оценок своих специалистов без всякого намерения передать разработку на сторону.

Постановка задачи – наиболее творческая часть ЖЦП, которая поднимает почти философские проблемы.

Требуется описать поведение разрабатываемой системы. Эта система получает какие-то сигналы из ее окружения, поэтому надо описать поведение окружения, но окружение само зависит и изменяется под влиянием системы, ее сигналов, особенно аварийных.

Разрешают это противоречие, постепенно уточняя поведение как системы, так и ее окружения. Для действительно важных систем заказчик требует разработки имитационных моделей системы и окружения, не уступающих по сложности и детальности самой системе. Была у меня однажды трагикомическая ситуация, когда военные заказчики потребовали модель системы, точно соответствующую реальной жизни. Никакие мои объяснения про сущность моделирования не помогали. Кончилось тем, что я сказал: "Хорошо, я сделаю такую модель, но саму систему делать не буду. Зачем? Ведь модель и так все делает". Это произвело на них впечатление. Хотя, конечно, стремиться к более точным моделям нужно всегда.

Несколько слов о декомпозиции системы. Я не устаю удивляться, насколько точнее оценка сложности системы, которая является суммой оценок ее компонентов, полученных в результате декомпозиции, чем первоначальная оценка системы в целом. Ведь делают эти оценки одни и те же системные аналитики, наверняка у них уже было примерное представление о системе и ее компонентах, когда они давали первоначальную оценку. И тем не менее, эти же мысли, положенные на бумагу, особенно с помощью каких-то средств формализации, приводят к совершенно другому результату.

Однако, формализация формализации рознь. Тридцать лет назад бытовало мнение, что можно и нужно каждую задачу формально специфицировать с помощью логики предикатов чтобы можно было доказать корректность ее решения[6]. Для каждой программы `P` нужно задать предусловие `A`, описывающее состояние среды (в частности, переменных программы) перед исполнением программы, и постусловие `S`, описывающее состояние после завершения ее исполнения:

```
A {P} S
```

Очевидно, что `A` и `S` – несопоставимы, поскольку описывают состояние в разные моменты времени. Можно "протянуть" предусловие `A` через текст программы, изменяя его соответственно каждому проходимому оператору, при этом логическая формула предусловия фантастически быстро растет, например, после условного предложения будет дизъюнкция двух вариантов, описывающих текущие предусловия после веток `then` и `else` – мы же не знаем, какая именно ветка будет исполняться. Обозначим результат протягивания через `A'`. Тогда остается доказать истинность импликации `A' => S`, и программа корректна! Мы тоже потратили несколько лет, работая в этом направлении, например, Н. Ф. Фоминых в 1976-м году защитил под моим руководством дипломную работу, реализовав "протягиватель" для Алгола 68. Все эти предусловия и постусловия были очень громоздкими – много больше, чем текст программы и затраты на их создание – тоже.

Собственно доказательством теорем мы не занимались, опираясь на тот факт, что у нас в Ленинграде активно работала одна из лучших в мире группа математиков ТРЭПЛО (теоретические разработки

эвристического поиска логического обоснования). Мы пригласили к нам в лабораторию системного программирования одного из самых активных ее членов Юрия Маслова – автора наиболее популярного в то время обратного метода выводимости [7] – выступить у нас на семинаре. Слушали его целый день, и к вечеру я осмелился спросить, когда же ЭВМ будет доказывать не учебные, а настоящие теоремы? Ответ был ошеломляющим. Сложные, но короткие – уже сейчас, длинные – никогда. Наш практический интерес к этой теме иссяк. Позже это направление трансформировалось в доказательство корректности преобразования одной программы в другую, проверку доказательства, сочиненного человеком, но это уже не имеет отношения к нашей теме.

Сегодня, когда мы говорим "формализация постановки задачи", мы подразумеваем разработку последовательности моделей, каждая из которых описывает систему и ее окружение с различных точек зрения с постепенной детализацией. Существенно, что все представления о системе, полученные в разных моделях, должны собираться в едином репозитарии (некоторой специальным образом устроенной базе данных) с тем чтобы иметь возможность сквозного проектирования, при котором каждая последующая модель использует результаты предыдущей и уж никак им не противоречит. Соответственно, и все возможные проверки должны быть сквозными.

Планирование, управление и тестирование

Планирование

Результатом фазы оценки осуществимости являются детальная спецификация, план работы и оценка стоимости. Наиболее традиционной формой плана можно считать сетевой график, который представляется в виде ориентированного графа с двумя выделенными вершинами – начало и конец работы. Вершинами графа являются события, соответствующие пунктам плана, а ребрами – работы. События должны выражаться глаголами совершенного вида: "тексты программ переданы в базу данных исходников", "все тесты пропущены", "группа оценки качества дала положительное заключение" и т.д., но уж никак не "выполняется прогон тестов". Ребра нагружаются оценками длительности работ, например, в днях или неделях.

Сетевой график – очень удобный инструмент: во-первых, на нем четко видна зависимость работ друг от друга, во-вторых, на основе сетевого графика можно вычислить длительность всей работы, в-третьих, пользуясь результатами этих расчетов, можно попытаться оптимизировать длительность и затраты на работу.

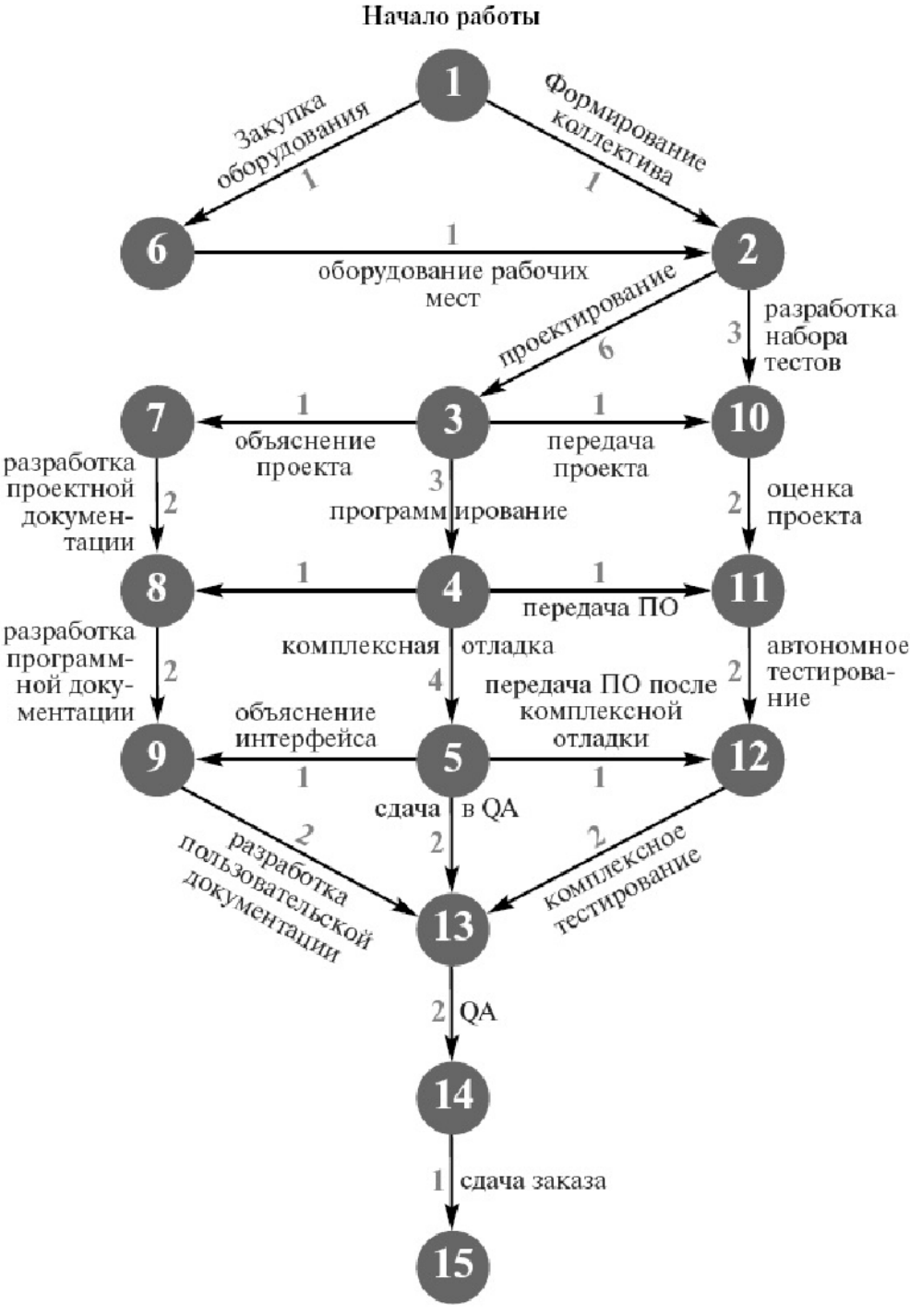
Длительность вычисляется следующим образом. Суммируем длительности работ по всем возможным путям в графе. Тот путь от начала к концу, который является самым длинным, объявляется критическим, потому что задержка любой работы, лежащей на этом пути, приводит к задержке всей работы в целом. Понятно, что критических путей (с одинаковой длительностью) может быть несколько.

Рассмотрим пример.

Здесь приводится классическая задача планирования: нормально работающая, полностью загруженная компания получила заказ, от которого по разным причинам невозможно отказаться. Перечислим названия событий, т.е. узлов в графе:

1. начало работы;
2. коллектив сформирован, рабочие места подготовлены;

3. проектирование завершено;
4. программирование завершено;
5. комплексная отладка завершена;
6. оборудование закуплено;
7. группа технических писателей получила описание проекта и необходимые пояснения от проектировщиков;
8. то же для ПО, разработка проектной документации завершена;
9. группа технических писателей получила всю необходимую информацию об интерфейсах с пользователем, разработка программной документации завершена;
10. группа оценки качества (Quality Assurance – QA) разработала тесты;
11. группа QA оценила проект положительно;
12. группа QA завершила автономное тестирование;
13. группа QA завершила комплексное тестирование, получила всю документацию и действующий вариант системы;
14. проверка качества (проблемам качества будет посвящена отдельная лекция) завершена;
15. конец работы (конечно, это не конец, будет еще сопровождение, но пример-то надо закончить).


Начало работы
1
Закупка оборудования
1
Формирование коллектива
1
6
1
оборудование рабочих мест
2
проектирование
6
3
разработка набора тестов
7
1
объяснение проекта
3
1
передача проекта
10
разработка проектной документации
2
3
программирование
2
оценка проекта
8
1
4
1
передача ПО
11
разработка программной документации
2
комплексная отладка
4
передача ПО после комплексной отладки
2
автономное тестирование
9
объяснение интерфейса
1
5
1
12
сдача в QA
2
2
2
разработка пользовательской документации
13
комплексное тестирование
2 QA
14
1 сдача заказа
15
Конец работы

Под каждым ребром графа записана планируемая длительность соответствующей работы (в неделях). Еще раз повторю, что это только пример, прошу не придираться к техническим деталям, в частности, разумеется, каждая передача (3-7, 3-10 и т.д.) не может длиться более одного-двух дней, но не хотелось возиться с дробями.

Критическими путями являются пути 1-6-2-3-4-5-9-13-14-15 и 1-6-2-3-4-5-12-13-14-15, т.е. вся работа не может быть выполнена быстрее, чем за двадцать одну неделю. Понятно, что с точки зрения оптимальной загрузки коллектива было бы лучше, чтобы все пути в графе от начала к концу имели примерно одинаковую длительность с тем чтобы как-то уменьшить длину критического пути. Например, есть соблазн заставить группу QA проводить даже начальное тестирование, уменьшив нагрузку на программистов, работа которых находится на критическом пути. Но тогда очень трудно определить границы ответственности, программисты начинают выдавать откровенную халтуру и в результате сроки даже удлиняются. В реальных проектах, где работ очень много, все-таки удается путем перераспределения работ улучшать сетевой график, по крайней мере, к этому стремятся все руководители.

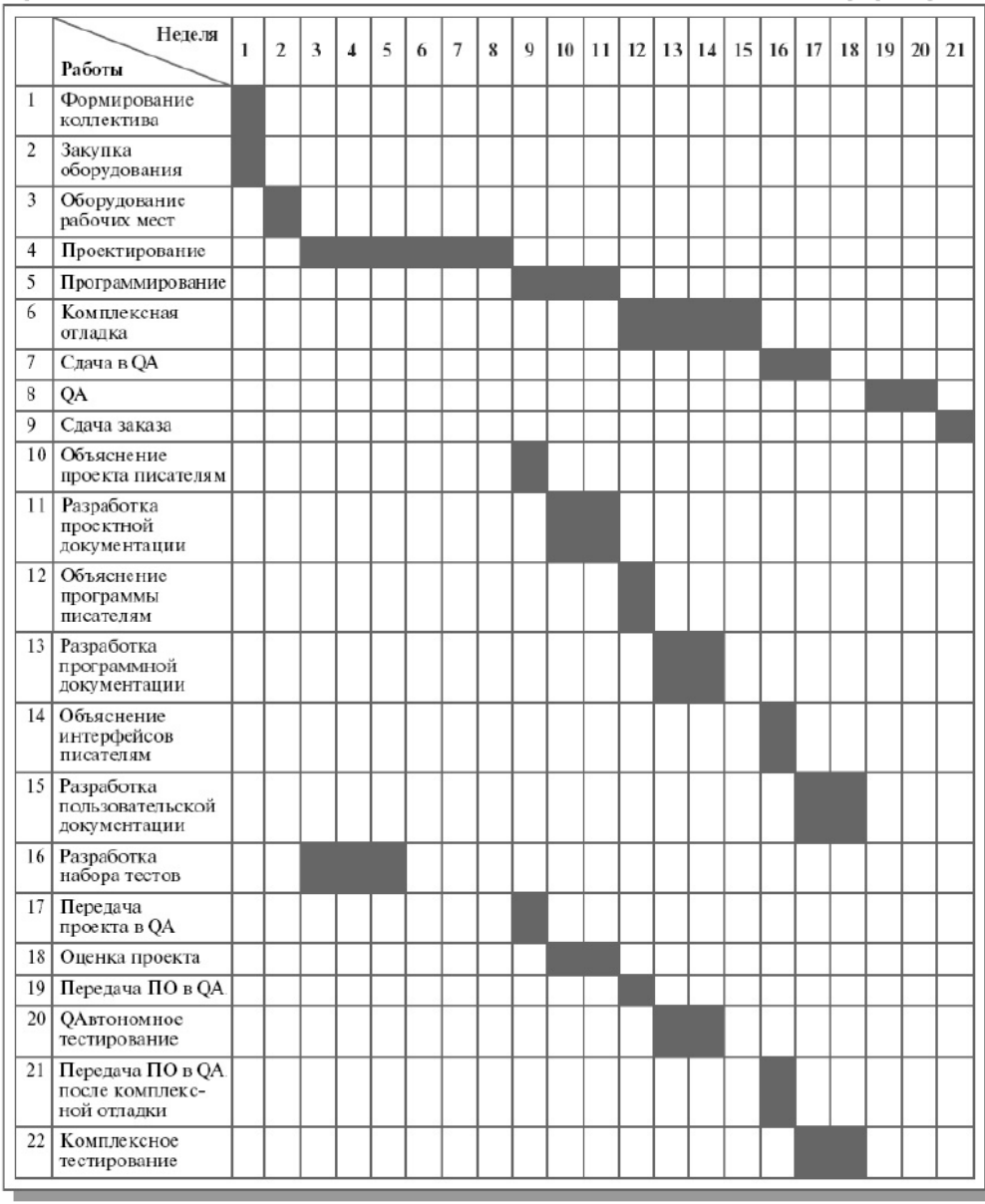
Еще несколько замечаний по данному примеру. Явно неудачно спланированы работы между событиями 1, 2, 6. Коллектив сформирован за одну неделю, а рабочие места еще не готовы. Группа технических писателей начинает работать на шесть недель позже проектировщиков, а группа QA имеет трехнедельный перерыв перед завершением проектирования и т.д.

Эти проблемы действительно трудны, каждая компания решает их по-своему, например, очевидным решением является использование одной и той же группы QA или технических писателей для нескольких групп разработчиков.

Еще одной популярной формой графического представления плана работ является диаграмма Ганта (Gantt). Диаграмма Ганта представляет собой прямоугольник: слева направо равномерно отсчитываются периоды времени (недели, месяцы), сверху вниз перечисляются работы, причем каждая работа представляется в виде отрезка, начало и конец которого размещаются в соответствующем периоде.

Для сравнения с сетевым графиком нарисуем диаграмму Ганта для того

же примера:



Если в сетевом графике особенно наглядно видны зависимости работ друг от друга, (например, работа 12-13 может начаться только после завершения работ 5-12 и 11-12), то в диаграмме Ганта основной упор делается на то, что происходит в каждую конкретную неделю, например, видно, что группа QA имеет перерыв в работе в 2 недели между работами 11-12 (автономное тестирование) и 12-13 (комплексное тестирование). Именно поэтому "большие начальники" предпочитают

диаграммы Ганта: в конце каждой недели проводишь вертикальную линию и сразу видишь, какие работы велись, что должно кончиться, а что начаться. Должен признаться, что я и сам нашел пару ошибок в расчетах по сетевому графику, пока строил диаграмму Ганта для этого примера. Тем самым я еще раз убедился, что диаграмма Ганта нагляднее сетевого графика, или в том, что я уже большой начальник, а не технический специалист. Технические менеджеры предпочитают сетевые графики, так как им важнее информация о том, что от чего зависит, да и пересчитывать критические пути им приходится практически каждую неделю.

На диаграмме Ганта принято над каждым отрезком работы указывать, сколько сотрудников принимает участие в этой работе (в сетевом графике это возможно, но не очень удобно, поскольку надо указывать еще и длительность работы). Опять-таки, начальники любят смотреть, сколько сотрудников занято в каждой неделе, что легко увидеть как раз на диаграмме Ганта.

Управление

На самом деле управление[8] почти неотделимо от планирования. Вы спланировали работу, приступили к ее исполнению. Каждую неделю Вы собираете еженедельные отчеты от руководителей групп и оцениваете состояние каждой работы по сравнению с сетевым графиком. Если все в порядке – можно плевать в потолок и ждать конца следующей недели. К сожалению, так бывает очень редко. Допустим, какая-то работа затянулась. Если она не на критическом пути – ничего страшного. Нужно провести воспитательную работу, дать дополнительный срок (причем надо следить, чтобы от этого увеличения не возник новый критический путь!), продумать какие-то меры, чтобы не допустить повторения подобных ситуаций. Если же проваленная работа лежит на критическом пути, необходимо перепланирование, причем основной проблемой является невозможность (чаще всего) сдвига сроков завершения. Мы только что (при планировании) занимались оптимизацией сетевого графика, прошло всего несколько недель — и вот мы снова занимаемся тем же самым.

Даже сама по себе задача определения задержки выполнения какой-то

работы является весьма нетривиальной. Люди имеют обыкновение выдавать желаемое за действительное, а уж менеджеры – и подавно. Каждый думает, что возникшую небольшую задержку он обязательно ликвидирует в ближайшие дни, поэтому можно и не сообщать о ней начальнику. Кроме того, хотя при составлении плана делалось все возможное, чтобы сформулировать все события максимально точным образом, всегда есть лазейка, чтобы представить недоделанную работу завершенной. Тут многое зависит от психологического климата в коллективе, от уровня доверия, установившегося между руководителями и подчиненными.

Не меньшее значение имеет и уровень технической оснащенности: одно дело, когда решение принимается только на основании устного сообщения, и совсем другое – когда все результаты работы (не только программы, но и схемы, алгоритмы, описания и т.д.) собираются в едином репозитории, когда имеется возможность выполнения различных формальных проверок, когда руководитель может просмотреть результаты в их развитии, оценить работу не только группы в целом, но и каждого ее участника.

Графически проблему управления можно представить в виде треугольника:

В этом треугольнике каждая вершина зависит от двух других. Например, если работы не укладываются в сроки, можно попытаться договориться с заказчиком об их удлинении (очевидно, что стоимость работ также

возрастет). Если сроки сдвинуть нельзя, необходимо договариваться с заказчиками об уменьшении объема работ (т.е. выкинуть какие-то функции из спецификации), либо привлекать дополнительные ресурсы. Кто-то из классиков сказал: "Добавлять людей в горящий коллектив – это все равно что заливать пожар керосином" – ведь новых людей нужно ввести в курс дела, отвлекая на это основных разработчиков, но даже после этого вряд ли удастся ускорить работу, так как увеличивается количество интерфейсов и согласований между участниками. В тех случаях, когда сроки сорваны, ресурсы исчерпаны, а важная работа не выполнена, легче эту работу заново спланировать и поручить другому коллективу.

Опытные руководители проектов отчетливо понимают эти проблемы и решают их, в первую очередь, за счет квалифицированного проектирования, учета возможных рисков, обеспечения возможности регулярного общения между разработчиками, выделения максимально независимых компонентов (которые могут быть переданы дополнительным разработчикам), за счет скрытых ресурсов, о которых разработчики даже не подозревают. Например, широко распространена практика, когда руководитель договаривается с заказчиком об одних сроках, а разработчикам сообщает другие, более сжатые сроки.

Неплохо иметь "пожарную команду" из двух-трех высококлассных специалистов, которые в промежутках между "пожарами" занимаются исследованиями или развитием инструментальных средств.

Некоторые авторы предлагают в случае выхода из графика применять "метод пряника", т.е. предлагать разработчикам дополнительные деньги за сверхурочную работу, но, может быть, это не очень верно. Т.е. такая мера может сработать раз или два, но потом разработчики интуитивно (а, может быть, и не совсем интуитивно) начинают затягивать работы, ожидая дополнительных подачек. Сидит такой имитатор деятельности, месяц бьет баклуши, но за два-три дня до сдачи начинает работать ночами и утром в день сдачи с воспаленными глазами сдает руководителю действующую программу, но не прошедшую через регулярную процедуру QA, не проверенную в комплексе с другими программами и т.д.

Жертвенность на работе – это, в первую очередь, признак

непрофессионализма.

Так как же все-таки управлять программистским коллективом? Добавлять людей нельзя, сулить премии нельзя, давить своим авторитетом на подчиненных нельзя, а что же можно? Хотелось бы получить ответ от каких-то известных в этой области специалистов, причем не на уровне общих рассуждений, а в виде конкретных рекомендаций и методик, иначе и вправду системное программирование сильно смахивает на искусство, а не на промышленную дисциплину.

Многие годы для нас таким авторитетом является Ф. Брукс, который руководил отделением программирования IBM (несколько тысяч человек) в шестидесятые годы, как раз тогда, когда создавалась знаменитая серия IBM/360. Покинув IBM, Ф. Брукс стал профессором университета Чапел-Хилл в Северной Каролине и написал ставшую бестселлером книгу "Как проектируются и создаются программные комплексы. Мифический человеко-месяц. Очерки по системному программированию". Эта книга в 1979 году была переведена на русский язык и издана в издательстве "Мир". За несколько лет до этого по СССР ходил "левый" перевод этой же книги, выполненный (и, кстати, гораздо лучше, чем официальный) академиком А.П. Ершовым. Через 20 лет вышло юбилейное переиздание книги[5], дополненное поздними статьями Ф. Брукса и подробным анализом, какие идеи, высказанные 20 лет назад, сохранили свою силу, а какие оказались ошибочными. Кроме несомненного дара системного программиста и огромного опыта руководства большими коллективами Ф. Брукс обладает ярким литературным талантом – многие фразы из книги стали общеизвестными лозунгами программистов. В декабре 2000 года мне удалось побеседовать с ним прямо на его рабочем месте, и личная встреча только усилила мои впечатления об этом замечательном ученом.

Так вот, читая эту книгу, можно обнаружить массу полезных и красиво изложенных советов, как организовать коллектив, какие бывают варианты жизненного цикла ПО, какую документацию и насколько подробно надо готовить, и еще множество интересных вещей. Но конкретных советов, что делать, когда гром уже грянул, когда работа, лежащая на критическом пути, не выполнена в срок, вы не найдете.

Основная идея состоит в том, что при правильной организации, с хорошо подготовленным коллективом можно уменьшить вероятность срывов, предугадать их на возможно более ранних сроках, но если это все же случилось – только личный опыт и интуиция руководителя помогут выбраться из этой ситуации с наименьшими потерями. Одна из более поздних статей Ф. Брукса "No silver bullet" как раз и посвящена обоснованию того факта, что нет волшебного простого средства, с помощью которого можно легко решить проблемы разработки ПО и, в частности, проблемы управления.

Но ведь люди работают, сроки горят, значит, какие-то выходы из этой ситуации есть. Я спросил Лена Эрлиха, нашего американского заказчика, как поступают американские менеджеры в ситуации, когда заваливается критически важная работа. Ответ был прост, как выстрел. Подключаются 1-2 мощных специалиста, проводятся совещания по ходу работ – каждый день, а если мало, то дважды в день, обязательно в присутствии высшего руководства. Те функции, ошибки в которых исправить трудно, безжалостно выкидываются. Главное, чтобы в срок заработала основная масса функций – тогда есть шансы договориться с заказчиком о дополнительном времени для завершения работы, если же к нужному времени показать нечего, скандал неминуем.

В общем-то, ситуация классическая: ученые говорят, что "добавлять людей в горящий коллектив — это все равно что заливать пожар керосином", а практики добавляют, ужесточают контроль и часто побеждают. "Ученый думает, что знает, но у него не совпадает, у практика все совпадает, но почему – никто не знает".

Чтобы не завершать параграф на такой грустной ноте, приведу перечень возможных корректирующих действий менеджера из внутренних правил одной крупной западной фирмы:

- перераспределите работы, лежащие на критическом пути, так, чтобы они исполнялись более опытными членами коллектива;
- увеличьте команду исполнителей временными сотрудниками (а не кадровыми);
- перераспределите исполнителей в нескольких командах (не только в "горящей");
- упростите требования к работе;

- не отвлекайте команду, постарайтесь не прерывать их работу;
- организуйте дополнительное техническое обучение;
- если возможно, используйте средства автоматизации разработки;
- организуйте сверхурочную работу, возможно, многосменную;
- перепланируйте всю работу, уменьшив число работ на критическом пути (особенно проверьте зависимости одних работ от других).

Как видите, все на уровне здравого смысла, никакой "серебряной пули", но здесь следует привести мою любимую фразу: "На свете есть множество общеизвестных, но не общепринятых истин". Если не получается руководить программным проектом, то почему бы не попробовать работать по правилам?

Тестирование, обеспечение качества

Для начала вспомним три аксиомы одного из самых первых советских программистов с несколько необычной фамилией Шура-Бура.

Аксиома 1. В каждой программе есть ошибка.

Аксиома 2. Если в программе нет ошибок, значит, в исходном алгоритме есть ошибка.

Аксиома 3. Если ни в программе, ни в алгоритме ошибок нет, то такая программа никому не нужна.

При всей шутливости этих аксиом в них отражена суровая правда жизни.

Исходные идеи тестирования абсолютно понятны [9].

Вы написали программу, хотите убедиться в правильности ее работы, поэтому пропускаете один или несколько тестов и смотрите, что получилось. Понятно, что тесты должны быть простыми настолько, чтобы можно было предсказать правильный ответ, иначе как вы определите, правильно ли программа сработала?

А что делать со сложными тестами и, вообще, сколько тестов нужно

пропустить? Ответ очевиден – бесконечно много, поэтому с помощью теста можно доказать наличие ошибок (если тест был достаточно "зубастым"), но нельзя доказать их отсутствие.

Бесконечность тестов определяется наличием циклов, рекурсии, разнообразием значений данных. Ситуация обычна для нашей специальности. Теоретическая невозможность не снимает с нас ответственности за поиск практических способов проверки – пусть не полных, но дающих какую-то степень уверенности. Итак, что же можно сделать?

Еще 30 лет назад был популярен критерий полноты тестирования, при котором набор тестов гарантировал, что по каждому ребру графа управления программы исполнение программы хотя бы один раз пройдет. Такой набор тестов можно создать, часто даже автоматически.

Чтобы проверить цикл, нужно создать тест, гарантирующий прохождение цикла 0, 1 и 2 раза. 0 – вдруг условие входа в цикл сразу было ложным, 1 – нормальное прохождение, 2 – не столь очевидно. Возможно, в конце цикла есть присваивание переменной, которая в начале цикла только читается. Понятно, что однократный проход по циклу не даст возможности проверки важного и весьма вероятного варианта.

Изложенные критерии носят только эвристический характер, но весьма полезны. Таким образом, мы видим, что программирование и тестирование – совершенно разные типы деятельности, требующие разных исполнителей.

Программирование – это конструктивный созидательный процесс, который требует высокой квалификации и определенного оптимизма (см. аксиомы Шуры-Буры).

Тестирование – деструктивный процесс, который требует высокой дотошности, подозрительности, пессимистичности, но, вообще говоря, не требует высокой квалификации. Пусть не обижаются на меня настоящие системные программисты, которые создают инструментальные средства автоматизации тестирования, я говорю об обычных тестерах. В каждом коллективе есть люди с неуживчивым характером, любящие покритиковать коллег по работе. Цены им не будет

в группе тестирования.

Еще раз повторим, что тестирование нужно вовсе не для того, чтобы показать, что программа работает правильно – это невозможно. Тестирование – это процесс исполнения программы с целью нахождения ошибок. Хороший тест – это тест, на котором с большой вероятностью ошибка найдется.

Теперь приведем несколько организационных и технологических соображений относительно тестирования.

Тестирование (не считая начальных отладочных тестов) не должно проводиться самими авторами программы. Наблюдается определенная "замыленность" взгляда автора на программу, поэтому он и тесты будет готовить на те варианты, которые предусмотрел в программе.

Должна быть конкуренция между программистами и тестировщиками, нужен дух соревнования: "Я все равно тебя поймаю" против "Врешь, не поймаешь".

Необходимо уметь оценивать вероятность и примерное количество оставшихся ошибок. Например, известен метод оценки количества рыб в пруду. Ловят, скажем, 100 рыбок, помечают их и выпускают обратно в пруд. Затем снова ловят 100 рыбок. Если почти все они мечены, значит, в пруду примерно 100 рыбок и есть, если же меченых попалось мало, то рыб, скорее всего, больше, причем в той же пропорции, какова доля непомеченных из 100 вновь пойманных. Во многих фирмах применяют тот же прием: специальные люди вставляют некоторое количество разнообразных ошибок в программы и возвращают их на доработку авторам с требованием обнаружить ровно столько же ошибок. По тому, какая часть найденных ошибок относится к специально вставленным, а какая – к вновь найденным ошибкам, можно судить о числе оставшихся ошибок. Еще проще – поручить тестирование одной и той же программы двум группам тестировщиков и оценить процент ошибок, найденных обеими группами.

Должно проверяться не только нормальное поведение программы, но и поведение в случае неправильных входных данных и других ошибочных ситуаций на предмет устойчивости программы, осмысленности сообщений об ошибках и т.д.

Всегда должен быть четко оговорен конечный результат тестирования. В идеале еще до начала работ или на одной из ранних стадий (работа 2-10 из нашего примера) должен быть создан и согласован с заказчиком набор тестов, успешный пропуск которых свидетельствует об успешном окончании работы в целом. К сожалению, на практике такое встречается редко, поэтому сдача проекта – это всегда, скажем так, шумный процесс.

Различают тестирование по типам:

- черный ящик (без просмотра исходного текста);
- белый ящик (с изучением исходного текста);

и по объему:

- маленький тест, типа печати "hello, world", чтобы понять, есть ли вообще о чем говорить;
- получасовое тестирование (американцы говорят "Тест на одну сигарету"), обычно проверяют по одному тесту на каждую функцию программы перед серьезным тестированием;
- модульное тестирование;
- комплексное тестирование.

Особого упоминания заслуживают тестирование граничных значений входных данных, тесты на максимальный объем счета, проверка предположений об ошибках ("Я бы сделал ошибку здесь").

Иногда применяется перекрестное чтение особо ответственных участков программы, когда одна группа разработчиков читает программы другой группы (inspection peer review). В США очень популярны Bugs festivals, когда незадолго перед выпуском системы фирма платит определенную сумму за каждую найденную ошибку. Список таких приемов можно расширить, но вряд ли их можно считать общеупотребительными.

В зрелых программистских компаниях для каждого проекта ведется своя база данных ошибок, в которую вносится:

- кто нашел ошибку, дата;

- описание ошибки;
- модуль, в котором ошибка обнаружилась (возможно, это наведенная ошибка, может быть, она вызвана ошибкой в совсем другом модуле);
- версия продукта;
- статус ошибки:
  - open: найдена
  - fixed: исправлена
  - can't reproduce: невозможно воспроизвести
  - by design: ошибка проектировщиков
  - wont fix: это не ошибка (тестеру показалось)
  - postponed: сейчас исправить трудно, исправим в следующей версии
  - regression: исправленная ошибка появилась вновь
- Важность (severity) ошибки:
  - crash: все падает, полная потеря данных
  - major problem: падает частично, частичная потеря данных
  - minor problem: что-то не то, но данные не теряются
  - trivial: сейчас не стоит исправлять
- Приоритет ошибки:
  - highest: невозможно поставить продукт с такой ошибкой, не можем перейти к следующей версии
  - high: поставить не можем, но можем перейти к следующей версии
  - medium: можем и исправим
  - low: косметические улучшения – оставим на следующую версию.

Особенно заметна ценность базы данных ошибок для больших и длительных проектов. Если идентификатор одной ошибки встречается десятки раз в различных письмах, недельных отчетах – это верный признак того, что требуется вмешательство руководства. Все руководители разных рангов знают на память записи из этой базы данных о текущих ошибках с высшей важностью и приоритетом. По завершении каждого проекта база данных ошибок внимательно анализируется. Интересно распределение ошибок по тому, кто их совершил, по времени исправления, кто чаще ошибки находит и т.д. Если в одну неделю разработчик 1 сделал 10 ошибок, а разработчик 2 – только 2, это еще ни о чем не говорит. Но если же за весь период

разработки разработчик 1 сделал 100 ошибок, а разработчик 2 – только 10, то по этим цифрам можно уже судить об их квалификации. Можно также оценивать целые группы или, скажем, качество проектирования, а можно сделать какие-то выводы по мощности и надежности используемых инструментальных средств.

На основе изучения записей в базе данных ошибок руководство проектом принимает решение о возможности выпуска продукта, например, продукт нельзя выпускать, если есть хотя бы одна ошибка с важностью "crash" или с приоритетом "highest/high". Возможна и такая стратегия, когда продукт выпустить обязательно надо, но времени на исправление основных ошибок нет (нужно помнить, что даже небольшие исправления могут повлечь за собой новые ошибки), тогда из продукта просто удаляют часть функций, в которых есть ошибки.

Мы говорили о тестировании, только отдавая дань многолетней традиции. На самом деле, любая зрелая программистская компания имеет большую независимую группу оценки качества ПО (Quality Assurance или просто QA), функции которой намного шире, чем просто тестирование. Для каждого продукта проверяется:

- полнота и корректность документации;
- корректность процедур установки и запуска;
- эргономичность использования;
- полнота тестирования.

QA должна играть роль придирчивого пользователя, но внутри компании. В США разработчики и QA сидят в разных концах коридора, не поощряется даже неформальное общение. В западных компаниях довольно существенную часть зарплаты (примерно 20-30%) составляет бонус, выплачиваемый в конце проекта и только в случае его успешного завершения. Так вот, если QA найдет слишком много ошибок разработчиков, те остаются без бонуса, но если QA принял разработку, а затем пользователи начнут жаловаться, то QA остается без бонуса, хотя жалобы связаны с ошибками разработчиков. Таким искусственным разделением ответственности и непоощрением дружеских отношений между разработчиками и QA владельцы компаний пытаются застраховаться от неудачи на рынке. В США говорят, что каждому продукту дается только одна попытка выхода на рынок. Если

пользователям по разным причинам продукт не понравился, репутация теряется навсегда. Поэтому ошибки так дорого стоят.

Групповая разработка и организация коллектива

Групповая разработка, управление версиями

Один из департаментов нашего коллектива разрабатывает ПО телефонных станций, которое включает в себя следующие компоненты:

- ОС реального времени;
- драйверы телефонного оборудования;
- функциональное ПО;
- программы рабочих мест операторов (РМО);
- БД конкретного экземпляра АТС.

Каждый компонент разрабатывается отдельным коллективом разработчиков, но все компоненты тесно связаны друг с другом. Более того, каждый компонент (кроме, пожалуй, драйверов) состоит из большого числа модулей, разрабатываемых разными специалистами. Таким образом, налицо проблема взаимодействия большого количества разработчиков, каждый из которых может находиться в своей фазе разработки, может внести в свою программу такие изменения, которые не позволят другому разработчику отлаживать его программу и т.д. К сожалению, и в нашем коллективе, в котором слово "технология" царит уже более 20 лет, нередки случаи, когда при выезде на объект БД не соответствует ФПО или версия РМО уже устарела. Наша техническая вооруженность позволяет быстро справляться с этими трудностями, но свести проблемы к нулю можно лишь жесткими организационными мерами, прежде всего, требуется преодолеть многие советские привычки и особенности нашего менталитета.

Еще 20 лет назад у нас в коллективе сложилась трехуровневая система версий, которая в том или ином виде актуальна до сих пор. Разработчик действует в своем файловом пространстве (раньше мы говорили "библиотека разработчика"), делает там, что хочет, и ни с кем ничего не согласовывает. Когда группа разработчиков решает, что какие-то модули отлажены (или когда их руководитель решает, что ждать больше нельзя, сроки поджимают), исходные тексты модулей передаются в библиотеку тестирования. Система или какой-то ее компонент проверяется на всех существующих тестах, причем библиотека тестирования остается

фиксированной, все найденные ошибки заносятся в базу данных ошибок и исправляются в библиотеках разработчиков. Когда все тесты пропущены, библиотека тестирования вместе со списком найденных ошибок копируется в библиотеку предъявления, а библиотека тестирования готова к приему новой версии из библиотек разработчиков.

Подчеркнем еще раз, что в библиотеку предъявления перенос осуществляется сразу же, когда пропущены все тесты, а вот перенос в библиотеку тестирования – только по мере готовности группы разработчиков выдать новую версию системы.

В чем же состоит роль библиотеки предъявления? Дело в том, что большие системы создаются большими коллективами с иерархической системой подчинения (позже мы увидим, что это не всегда так). У руководителя любой группы есть начальник, а у самого большого начальника есть заказчик, словом, твой начальник всегда может потребовать у тебя версию для комплексирования системы на его уровне. Так вот, лучше отдать ему пусть немного устаревшую версию, но с известным списком ошибок (пусть на следующем уровне часть тестов, где используются ошибочные функции, пока не гоняют), чем более новую, "с пылу, с жару", но с непредсказуемым поведением. Итак, библиотека предъявления ждет своего часа, когда начальник ее затребует, а может и так случиться, что библиотека предъявления успеет несколько раз обновиться, пока ею заинтересуются.

Описанная выше трехуровневая система версий является слишком крупноблочной. В современных технологиях используются специальные инструментальные средства, позволяющие фиксировать каждое изменение и откатываться к нужной точке, например, Visual Source Safe или PVCS. Обычно такие системы версионирования устанавливаются на сервере, разработчики играют роль клиентов (тем самым автоматически решаются технические проблемы, например, разрешение конфликтов при одновременном обращении, вопросы back-up и др.), а система предоставляет следующие методы:

- Check-out: взять текст на исправление;
- Check-in: вернуть обратно;
- Undo check-in: отмена всех изменений, сделанных во время

последней сессии;
- Get latest version: берется один файл или файлы целого проекта в том состоянии, в котором их застал последний check-in.

Последняя версия обычно хранится отдельно.

Само собой разумеется, что в системе версионирования можно хранить исходные тексты программ, двоичные файлы, документацию и т.п.

Насколько хорошо у нас поставлена система версионного контроля, как-то раз я убедился лично. Однажды поздно вечером я зашел за своей женой, которая также работает на нашем предприятии, и говорю: "Пошли домой, устал до смерти, больше не могу работать". Она мне: "Пока check-in не сделаю, уйти не могу, иначе меня завтра ждут большие неприятности". Это меня позабавило – я же генеральный директор, какие могут быть неприятности, если я сам о чем-то прошу? Она мне спокойно объясняет, что в данном случае никакой роли не играет, кто я и кто она. Каждую ночь у нас идет автоматическая сборка разрабатываемого продукта, затем автоматическое тестирование. Сборка начинается с перетрансляции абсолютно всех модулей (чтобы исходные тексты не разошлись с объектными модулями). Если в текстах есть формальные ошибки, трансляция завершится с аварийным кодом возврата, сборка выполняться не будет, а утром как виноватые исполнители, так и их руководители получат автоматически сгенерированные сообщения с исчерпывающим объяснением ситуации. Это какой-то страшный бездушный молох, машина, в которую даже генеральный директор влезть не может.

В результате мне пришлось прождать больше часа, пока все формальности не были выполнены.

Психология программирования

В первый раз я столкнулся со специалистами по инженерной психологии в 1968 году. На мат-мехе была установлена польская ЭВМ ODRA 1204 с хорошей операционной системой и достаточно полной реализацией Алгола 60. На этой ЭВМ мы впервые получили возможность посимвольного ввода/вывода информации, используя телетайп или перфоленту. До этого мы могли работать только с целой

колодой перфокарт или с целой перфолентой. Мы реализовали один из первых в СССР (и уж точно – первый в нашем университете) диалоговый корректор текстов (программа dico [10]). Г.С. Цейтин придумал идею и показал примеры основных операций, я написал на Алголе 60 почти все программы, а С.Н. Баранов подготовил хорошую документацию. Программа быстро стала популярной, после чего Александр Марьяненко (сотрудник института комплексных социальных исследований) попросил разрешения провести некоторые психологические исследования проблем диалогового редактора. Мы, разумеется, согласились, хотя совершенно не верили, что это принесет пользу. Как оказалось – зря не верили. Симпатичная дипломница А. Марьяненко провела 2-3 месяца, наблюдая за нашей работой. Она была тихой и незаметной, мы ее присутствия практически не ощущали. Зато после этого она выдала целый ряд наблюдений и рекомендаций, оказавшихся очень ценными. Например, она заметила, что некоторые часто встречающиеся вместе команды кодируются символами в разных регистрах, из-за чего оператор вынужден чаще нажимать на клавиши, а некоторые команды были названы неудачно, что провоцировало пользователей на ошибки. Но больше всего мы удивились, когда она заметила, что отладка модуля размером в одну страницу может быть не на проценты, а в разы проще отладки модуля размером в одну страницу и еще 4-5 строк на другой странице.

Оказалось, что это связано с принципами организации человеческой памяти. Есть сверхоперативная память, связанная, в основном, со зрением. Эта память имеет очень быстрый доступ, но очень мала – 7-9 позиций. (Говорят, что у профессиональных программистов эта память имеет 20-25 позиций.)

Существенно больше оперативная память, в которой и происходит вся основная мыслительная деятельность, но данные в ней не могут храниться долго. Наконец, самая большая — долговременная память. Человеку непросто заложить туда данные, но хранятся они долго.

С устройством памяти связан принцип "центрального" (в отличие от "периферического") зрения. Человек хорошо воспринимает какую-то точку и то, что ее окружает.

Если при отладке программы автор должен обозревать больше, чем

одну небольшую страницу текста, он не может полноценно воспринять программу. "Листать вредно".

Еще один важный принцип инженерной психологии также связан с устройством человеческой памяти – принцип возможно раннего обнаружения ошибок. Если программист написал программу и тут же заметил ошибку, то ее исправить сравнительно просто. Если же ему сообщают о найденной ошибке через полгода, ее исправление превращается в проблему. Именно по этой причине мы являемся сторонниками статических АЯВУ (в которых транслятор в каждой точке программы "знает" виды обрабатываемых значений), а не динамических, в которых типы данных определяются во время счета. В динамических языках простое несовпадение типов может обнаружиться и через полгода, когда сложится "нужное расположение звезд на небе". Словом, в инженерную психологию как в науку мы поверили. Поэтому, когда тот же А. Марьяненко несколькими годами позже предложил провести психологическое обследование нашей лаборатории системного программирования, мы согласились. Поводом послужил тот факт, что в этот момент я оказался самым молодым руководителем лаборатории ЛГУ, и психологам было интересно исследовать взаимоотношения молодого руководителя с его старшими коллегами.

Нам раздали анкеты со множеством странных вопросов типа: "Как ты думаешь, что скажет о тебе такой-то сотрудник?". Содержание каждой анкеты не раскрывалось (таковы правила социологических опросов), но суммарные результаты, полученные, кстати, на ЭВМ (!) нам были представлены. Результаты были интересные, содержательные, многие рекомендации мы использовали. Но один факт меня просто поразил. А. Марьяненко предсказал мне, что два сотрудника не более чем через полгода лабораторию покинут. Как же так? Я же их учил, играл с ними в баскетбол, мы вместе решали трудные задачи, никогда ни одного плохого слова я от них не слышал. Я тогда промолчал о предсказании: если это ошибка, то зачем травмировать хороших людей, а если правда, то что тут поделаешь. Через полгода они ушли с громким скандалом, написав бумагу, в которой обвинили меня во всех смертных грехах. Я был потрясен, но не их уходом, а сбывшимся предсказанием.

Я стал читать книги по психологии, это оказалось не так интересно для математика, но какие-то принципы я запомнил.

Многое в поведении людей объясняет пирамида Маслова [11]:

Диаграмма Маслова носит совершенно тривиальный характер – если человеку нечего есть и негде жить, то что ему до высоких материй? Но совокупное понимание человеческих потребностей и их выстроенная последовательность очень важны. В каком-то смысле каждый программист проходит весь путь от простого зарабатывания денег до понимания того, как важно быть членом команды, признания товарищей, достижения высокой самооценки.

Любой человек принадлежит к одному из трех типов:

Лидер. Человек, который стремится управлять другими людьми, проектами, для которого нестерпимо быть просто "винтиком" в сложном механизме. Из лидеров выходят политические деятели, менеджеры разного уровня, для них, прежде всего, важен личный успех.

Технарь. Человек, который получает настоящее удовольствие от самого

процесса поиска решения, например, программирования, которому по большому счету наплевать, что есть начальники и подчиненные, что о нем не пишут газеты и т.д. Главное – решить задачу.

Общительный. Эти люди разносят информацию, популяризуют результаты других людей. Психологи говорят, что 60% женщин относятся именно к этой категории.

Знание этих категорий очень важно для менеджеров программистских коллективов. Если собрать коллектив из одних лидеров, то будет постоянная борьба за власть, даже самые лучшие идеи не будут доведены до реализации, да и обмена идеями, скорее всего, не будет.

Команда из одних технарей не будет соблюдать бюджет и сроки, каждый будет сидеть в своем углу и решать ту задачу, которая ему больше нравится, а не ту, решение которой необходимо в данный момент. Для мат-меха это очень типичная ситуация.

Наконец, в коллективе, состоящем преимущественно из общительных людей, будут самые веселые праздники, туда всегда будет приятно зайти, но работа спориться не будет. И такие коллективы я встречал неоднократно.

Понятно, что хороший коллектив должен включать удачное сочетание лидеров, технарей и людей, ориентированных на общение. Как говорится, "мамы разные нужны, мамы всякие важны".

Организация коллектива разработчиков

Помните историю из Библии о Вавилонской башне? Адам и Ева вкусили запретный плод, в результате чего люди стали быстро размножаться, и говорили они, очевидно, на одном языке. Дела у людей шли хорошо, но возгордились они сверх всякой меры и чтобы доказать Богу, что они его не боятся, решили построить башню до самого неба. Бог с ними разобрался очень быстро – взял и смешал разные языки, люди потеряли возможность общаться, а, значит, управляемость, в результате амбициозный проект так и не был завершен.

Проблема интерфейсов – главная проблема в организации коллектива.

Если есть n сотрудников, то имеется $Cn^2 = n*(n-1)/2$ парных интерфейсов, а ведь иногда надо обсудить проблему и втроем, и вчетвером. Таким образом, с ростом коллектива быстро растет количество интерфейсов внутри него, при каждом взаимодействии могут возникнуть споры, непонимание, разночтение и другие конфликты. Как же с этим бороться?

Понятно, что ставить перед собой задачу абсолютного решения этой проблемы не стоит, но какие–то пути решения я покажу. А уж дальше можно надеяться только на личный опыт руководства коллективами.

Основная идея состоит в построении иерархии подчиненности. Есть руководитель темы или отдела, у него в подчинении несколько руководителей групп, у каждого руководителя группы в подчинении несколько специалистов. Разумеется, уровней иерархии может быть и 2, и 3, и 4, но все-таки не 15 и не 20. Каждый руководитель должен сформулировать задачу своим подчиненным так, чтобы минимизировать интерфейсы между ними, т.е. максимально локализовать решаемые ими задачи. Это не всегда возможно, не у всех руководителей получается, но так и различаются руководители и их команды.

В незапамятные времена многие верили в закон Конвея: "Каждая система структурно подобна коллективу, ее разработавшему". В 1976 году мне пришлось довольно долго беседовать с руководителем разработки трансляторов с языка ПЛ1 для IBM/360 по фамилии Маркс. Родился он в тот же год, что и я, закончил университет тогда же, когда и я, занимался очень похожей работой (в то время я был одним из руководителей разработки транслятора с языка Алгол 68), в общем, нам было интересно побеседовать. Я его спросил: "Почему PL1/F имеет 51 просмотр, когда с Алголом 68 мы справляемся за 6?". Он отвечает: "А что тут понимать, у меня в подчинении был 51 программист. К тому времени, когда мы начали реализовывать оптимизирующий транслятор с ПЛ1, у меня было 100 человек, поэтому PL1/ opt имеет 100 просмотров". Я и тогда понимал, что это глупо. Между каждой парой просмотров нужно организовывать файл на промежуточном языке; один просмотр – пишет, другой – читает, добавьте разные упаковки и распаковки, другие служебные действия – и вы поймете, почему трансляторы с ПЛ1 работали так медленно.

Но не все так просто в этом мире. Мы делали транслятор чуть ли не 10 лет, а они "слепили" свой транслятор за два года. Тот же Маркс позавидовал мне, что у нас есть такая замечательная возможность заниматься любимой наукой, не торопясь, исследовать разные варианты, придумывать новые методы, публиковать монографии и т.д. С понятием рынка ПО в те годы мы были совершенно не знакомы, да и сейчас, через 30 лет, далеко не все наши программисты понимают законы рынка: как важно первым выдать на рынок пусть не самый лучший, но работающий продукт, соответствующий определенным потребностям рынка. Таким образом, если вы проектируете систему в соответствии с законом Конвея и при этом вам легче уложиться в сроки и средства, значит, так и нужно, а всякие изыски оставьте университетским "яйцеголовым" (так в Америке обзывают нас и наших коллег из университетов всего мира).

Со временем закон Конвея, который был сформулирован его автором только в качестве шутки, был обобщен до довольно конструктивного метода – матричного. К чему подталкивает закон Конвея? Каждый специалист выполняет только свой небольшой кусок работы, но делает это всегда, для всех заказов такого же типа, набивает руку, почти не думает, совершает мало ошибок, т.е. процесс превращается в производственный.

Итак, пусть у нас есть n специалистов и m однотипных заказов. Строим матрицу из n строк и m столбцов. Элемент в i-ой строке из j-ого столбца обозначает работу i-ого специалиста для j-ого заказа. Например, один специалист хорошо делает синтаксические анализаторы, другой – оптимизаторы, третий – генераторы и т.д., а однотипные заказы – это трансляторы с разных языков для разных платформ. Тем самым каждый специалист подчинен двум руководителям – административному (он же принадлежит какой-то группе, отделу) и руководителю проекта, в котором он принимает участие в данный момент.

У матричного метода есть свои плюсы и минусы. С одной стороны, узкая специализация по конкретной теме позволяет надежнее планировать сроки, добиться той самой повторяемости результатов, которой требует CMM уровня 2. С другой стороны, нарушен принцип единоначалия (как когда-то в Красной Армии были командиры и комиссары, причем с одинаковыми правами и ответственностью; жизнь

быстро доказала неправильность такого решения). Часто возникает ситуация, когда руководителю проекта нужен какой-то конкретный специалист, а начальник отдела загрузил его работой для другого заказа.

Есть и еще один недостаток матричного метода, не столь очевидный. Конечно, занимаясь годами одним и тем же, специалист оттачивает свое мастерство, но в целом он ограничен, не расширяет свой кругозор и т.п. "Специалист подобен флюсу – полнота его односторонняя". Предположим, некий сотрудник института много лет успешно занимался темой X, а по прошествии какого-то времени тема X перестала быть актуальной (такое в нашей науке бывает очень часто). И что он будет делать?

У Брукса описана совершенно другая модель организации коллектива – бригада главного хирурга. Сложные операции всегда делает один человек – главный хирург, но ему помогает целая бригада – кто-то делает надрезы, а потом зашивает, кто-то подает инструменты, следит за показаниями приборов и т.д. Примерно такой же схемой воспользовались Миллз и его коллеги для разработки информационной системы газеты "Нью-Йорк Таймс".

Все программы, документацию, основные тесты пишет один человек – главный программист. У него есть заместитель, который участвует в проектировании, обсуждениях, критикует решения главного программиста, но ни за что не отвечает. В случае болезни или по какой-то другой важной причине заместитель может заменить главного программиста.

В бригаде есть тестер, секретарь, библиотекарь, продюсер с вполне понятными функциями. Возможно подключение еще каких-то узких специалистов. Оказалось, что такая бригада может работать в 3-5 раз быстрее традиционных команд программистов, поскольку не нужно тратить время на согласование деталей интерфейсов с другими программистами, изделие получается цельным и "элегантным", выполненным в одном стиле. Главный программист выбирается из числа наиболее опытных и высококвалифицированных, он может себе позволить сосредоточиться на основной задаче и не тратить время на всякие "бытовые" функции, которыми полна наша повседневная жизнь.

Эта модель организации коллектива не нашла широкого применения.

Во-первых, ее трудно масштабировать. Сколько строк исходного кода может написать один программист? Ну, 50 тысяч, ну, скажем, 100. А если нужно миллион строк? Тогда все проблемы интерфейсов возвращаются, да еще на более сложном уровне, поскольку двум главным программистам договориться гораздо труднее, чем двум рядовым. Во-вторых (или все-таки во-первых?), где вы найдете множество программистов, готовых безропотно подчиняться главному программисту? Наша специальность подразумевает наличие твердого характера, творческого начала, гордости за свое детище.

Организация коллектива разработчиков в компании Microsoft

Относиться к компании Microsoft можно по-разному – слишком агрессивны, чрезмерное стремление к монополизму, программы огромны и неэффективны, документация часто непонятна и, опять-таки, очень объемна. Тем не менее, абсолютное большинство пользователей работает именно на продуктах этой компании, Microsoft поддерживает университеты по всему миру, да и со своим бесконечным ПО успешно справляется. Поэтому в лекциях по технологии программирования мы не можем пропустить опыт Microsoft в этой области[12].

Служба Microsoft Consulting Services провела анализ результатов выполнения большого количества программных проектов. Оказалось, что только 24% проектов можно признать в той или иной степени успешными, 26% не были завершены, а остальные столкнулись с большими проблемами, например, бюджет был превышен вдвое или затрачено в 1,5 раза больше времени.

Основными причинами неудач были признаны следующие:

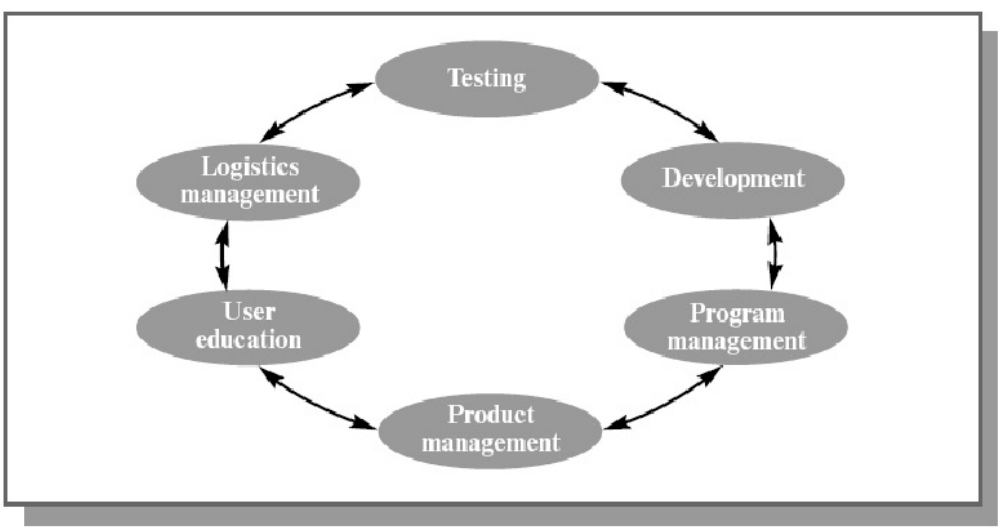
- постоянное изменение требований;
- нечеткие или неполные спецификации;
- низкое качество кода;
- слишком широкая постановка задачи;
- ошибка в подборе кадров;
- плохая организация работы;
- нечетко сформулированные цели.

Для преодоления этих трудностей был предложен набор моделей Microsoft Solution Framework (MSF), в котором учтен опыт, накопленный группами разработки программных продуктов.

Самыми революционными оказались модель команды разработчиков и модель процесса разработки. Первая модель (team model) описывает, как должны быть организованы коллективы и какими принципами им надо руководствоваться для достижения успеха в разработке программ. Разные коллективы могут по-своему применять на практике различные элементы этой модели – все зависит от масштаба проекта, размера коллектива и квалификации его участников.

Формирование коллектива – сложная задача, которая должна решаться с помощью психологов. Вот некоторые основные положения:

- не должно быть команды из одних лидеров;
- не должно быть команды из одних исполнителей;
- в случае неудачи команда расформировывается;
- система штрафов (если проект проваливается – наказывают всех).



Этот "бублик" описывает только роли, за ними могут скрываться несколько человек, исполняющих каждую роль. Самое удивительное, что в этой модели не предусмотрено единоначалия – все роли важны, все роли равноправны, поэтому MSF называют моделью равных (team of peers).

Program management – управление программой. Исполнитель этой роли отвечает за организацию (но не руководит!): осуществляет ведение графика работ, утренние 15-минутные совещания, обеспечивает соответствие стандартам и спецификациям, фиксацию нарушений, написание технической документации.

Product management – управление продуктом. Исполнители этой роли отвечают за общение с заказчиком, написание спецификации, разъяснение задач разработчикам.

Development – наиболее традиционная роль – разработка и начальное тестирование продукта.

User education – обучение пользователей. Написание пользовательской документации, обучающих курсов, повышение эффективности работы пользователей.

Logistic management – установка, сопровождение и техническая поддержка продукта, а также материально-техническое обеспечение работы коллектива.

Testing – тестирование. Выявление и устранение недоработок, исправление ошибок, другие функции QA.

Все решения принимаются коллективно, разделяется и ответственность в случае провала проекта.

В MSF утверждается, что такую модель можно масштабировать, разбивая систему по функциям. Лично у меня это утверждение (как и коллективная ответственность) вызывает большие сомнения.

Модель процесса определяет, когда и какие работы должны быть выполнены.

Перечислим основные принципы и практические приемы, лежащие в основе модели:

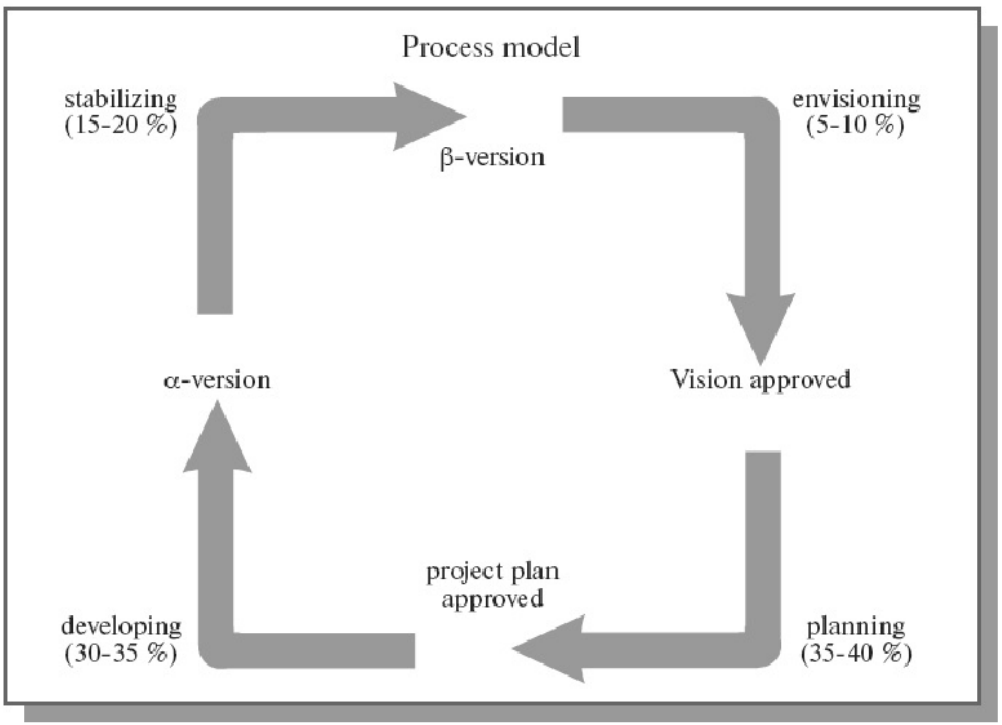
- итеративный подход (последовательный выпуск версий);
- подготовка четкой документации;
- учет неопределенности будущего;

- учет компромиссов;
- управление рисками;
- поддержание ответственного отношения коллектива к срокам выпуска продукта;
- разбиение крупных проектов на более мелкие управляемые части;
- ежедневная сборка проекта;
- постоянный анализ хода работ.

Process model имеет три основные особенности:

- разбиение всего процесса на фазы;
- введение опорных точек;
- итеративность.

Весь процесс разбивается на четыре взаимосвязанных фазы. Прежде чем переходить к следующей фазе, на предыдущей должны быть получены определенные результаты.



В принципе, ничего нового тут нет, это известная спиральная модель,

но необычно разбиение трудоемкости по фазам, связанное с тем, что и назначение каждой фазы весьма своеобразно.

Envisioning – выработка единого понимания проекта всеми членами коллектива. Эта фаза заканчивается разработкой формализованного документа:

- problem statement — описание задачи объемом не более одной страницы;
- vision statement — от чего хотим уйти, чего хотим добиться;
- solution concept — что хотим внедрить и как;
- user profiles — кто будет этим пользоваться;
- business goals — возврат инвестиций;
- design goals — конкретные цели и ограничения продукта, его конкретные свойства.

Planning — планирование очередного цикла разработки:

- функциональные спецификации;
- план-график работ;
- оценка рисков.

Developing — разработка, причем рекомендуются различные технологические приемы, например, переиспользование кусков кода, программирование по контракту, написание защищенного от ошибок ПО и т.д.

Stabilizing — создание стабильной версии, готовой к использованию.

Важную роль играют опорные точки (milestones), в которых анализируется состояние работ и производится их синхронизация. В этих точках приложение или его спецификации не замораживаются. Опорные точки позволяют проанализировать состояние проекта и внести необходимые коррективы, например, перестроиться под изменившиеся требования заказчика или отреагировать на риски, возможные в ходе дальнейшей работы. Для каждой опорной точки определяется, какие результаты должны быть получены к этому моменту.

Каждая фаза процесса разработки завершается главной опорной точкой (major milestone). Характеризующие ее результаты видны не только коллективу разработчиков, но и заказчику. Главная опорная точка – это момент, когда все члены коллектива синхронизируют полученные результаты. Назначение таких точек в том, что они позволяют оценить жизнеспособность проекта. После анализа результатов коллектив разработчиков и заказчик совместно решают, можно ли переходить на следующую фазу. Таким образом, главные опорные точки – это критерии перехода с одной фазы проекта на другую.

Внутри каждой фазы определяются промежуточные опорные точки (interium milestones). Они, как и главные, служат для анализа и синхронизации достигнутого, а не для замораживания проекта. Но, в отличие от главных опорных точек, промежуточные видны только членам коллектива разработчиков. Промежуточные опорные точки отмечают более скромные достижения и разбивают большую задачу на мелкие части, выполнение которых легче контролировать.

Итеративность процесса заключается в его многократном повторении на протяжении всего цикла создания и существования продукта. На каждой успешной итерации в продукт включаются только те новые средства и функции, которые удовлетворяют изменяющимся требованиям бизнеса.

Важную роль в MSF играет postmortem – так раньше называли распечатки памяти после аварийных завершений программы. В этом документе описывается, что было хорошо и какие возникали проблемы, т.е. такие знания, которые можно накапливать для использования в следующих проектах.

Документирование, сопровождение, реинжиниринг и управление качеством

Документирование

Одним из главных отличий просто программы от программного продукта является наличие разнообразной, хорошо подготовленной документации. Чтобы как-то структурировать этот параграф, воспользуемся ГОСТом ЕСПД (Единая Система Программной Документации) еще советских времен, кстати, до сих пор не обновлявшимся [13].

Самым главным документом является ТЗ (Техническое Задание), в котором описываются цели и задачи работы, заказчик и исполнители, технические требования, сроки и этапы, требования секретности, форс-мажорные обстоятельства и правила предъявления результатов. Технические требования, в свою очередь, делятся на функциональные, экономические, требования по надежности, эффективности, защищенности от несанкционированного доступа и др. ТЗ должно быть составлено таким образом, чтобы исключить возможные разночтения, все требования должны быть сформулированы так, чтобы их можно было проверить однозначным образом. Самая главная ошибка начинающих руководителей – это неаккуратно составленное ТЗ или неоднозначные спецификации. В моей практике было много случаев, когда заказчики, пользуясь неопытностью наших руководителей, отказывались от финальных платежей, требовали дополнительных работ или еще каких-то преференций. С другой стороны, при утверждении ТЗ, я как генеральный директор, не могу уследить за всеми деталями, поэтому еще много лет назад взял себе за правило проверять только финансовую сторону договора, а уж если руководитель договора прозевал что-то по технике, то пусть ему это будет уроком. Разумеется, в таком случае пострадает и предприятие, но надо же как-то учить руководителей.

Следующим по важности документом является ПМИ (Программа и Методика Испытаний). Структурно ПМИ подобна ТЗ – практически для каждого пункта ТЗ в ПМИ говорится, как этот пункт будет проверяться. Способы проверки могут быть самыми разными – от пропуска

специального теста до изучения исходных текстов программы, но они должны быть предусмотрены заранее, а не придумываться в момент испытаний. Новички приступают к составлению ПМИ непосредственно перед завершением работ, а опытные руководители составляют ПМИ практически одновременно с ТЗ (хотя бы в общих чертах) и согласовывают ее с заказчиками вместе с ТЗ. Именно хорошо составленная ПМИ является гарантией успешной сдачи работ.

Руководство системного программиста, на современном чисто русском языке называется руководством по инсталляции. В нем описывается порядок установки системы на ЭВМ, как проверить корректность поставленной системы, как вносить изменения и т.п. Обычно это простой короткий документ; в противном случае система, наверное, плохая, сделанная непрофессионалами.

Руководство оператора (пользователя) – это, собственно, основной документ, описывающий, как пользоваться системой. В хорошем руководстве сначала описывается идея системы, основные функции и как ими пользоваться, а уже потом идет описание всех клавиш и меню. Многие современные книги по программам Microsoft представляют собой яркие примеры никуда не годных руководств – можно прочесть 100 страниц и не понять основных функций.

Руководство программиста. Часто это самый объемный документ, описывающий внутреннюю организацию программы. Обычно этот документ идет в паре с документом "текст программы" – одностраничный документ с оглавлением дискеты или CD. Руководство программиста дает заказчику возможность дописать какие-то новые фрагменты программы или переделать старые. В современной литературе этот документ называется SDK (Software Development Kit). Продукт, снабженный SDK, может стоить на порядок дороже, чем такой же продукт без оного, так что можно понять, насколько трудно создать действительно полезный SDK. Сейчас не очень принято продавать исходные тексты программ – проблемы с интеллектуальной собственностью, даже при наличии SDK трудно "влезть" в чужую программу – как говорится, себе дороже. Поэтому большое распространение получили API (Application Program Interface). Программа передается только в виде DLL (библиотека двоичных кодов), но известно, как обратиться к каждой функции из других программ, т.е.

известно имя точки входа, количество, типы и значения параметров. Наличие множества API, конечно, хуже, чем наличие исходных текстов (например, нельзя переделать что-то в середине функции), зато много проще в использовании. С другой стороны, все большую популярность приобретает FSF (Free Software Foundation) [14]. Основателем этого движения был Ричард Столман, который забил тревогу по поводу попыток крупных фирм запатентовать многие основные алгоритмы и программы: "Дойдет до того, что они запатентуют понятия "цикл" и "подпрограмма", что мы будем тогда делать?" FSF представляет собой собрание программ в исходных текстах; любой программист может свободно использовать их в своих целях, но все добавления и улучшения, которые он сделал, тоже следует положить в FSF. Таким образом, FSF представляет собой одно из самых больших доступных хранилищ программ. Многие ведомства (например, военные) просто не могут использовать программы, текстов которых они не имеют. Мы уже много раз пользовались FSF, естественно, соблюдая все правила игры.

Остальные документы, перечисленные в ЕСПД, носят формальный или необязательный характер, поэтому мы их обсуждать не будем.

Сопровождение

Случилось чудо — вам удалось сдать программу, а заказчик начал ее промышленное использование. Вы облегченно вздохнули, обрадовались и, как всегда, рано. Начинается этап сопровождения ", который длится, пока программа живет. В процессе эксплуатации возникают новые требования, всплывают ошибки, объемы обрабатываемых данных растут, соответственно, растут и требования к эффективности. Нравится вам это или нет, "живущую" программу придется все время изменять.

Основными задачами сопровождения являются:

- исправление ошибок;
- регулярное проведение замеров производительности (профилирование);
- улучшение "времяпожирающих" мест в программе (еще говорят "бутылочное горлышко");
- улучшение документации;

- расширение функциональности;
- принятие решения о прекращении эксплуатации программы или ее реинжиниринге (переводе программы на другую платформу).

Очень трудно сопровождать программу, в процессе разработки которой о сопровождении никто не думал. Такие простые вещи, как разумные комментарии, выбор идентификаторов и ступенчатое расположение строк исходного текста, заметно облегчают сопровождение, но часто разработчики об этом не заботятся. Только на этапе разработки можно предусмотреть средства для профилирования, снятия трасс, отладочных печатей и т.п. "Программист, который в процессе разработки продукта пользуется отладочными средствами, а перед сдачей программы в промышленную эксплуатацию выкидывает их, похож на человека, который ходит по берегу в спасательном жилете, а, уходя в море, оставляет жилет на берегу".

Важным элементом сопровождения является "горячая линия" – 24 часа в сутки, 7 дней в неделю должен быть доступен специалист (по телефону или по электронной почте), готовый ответить на вопрос, принять замечание, просто поговорить с пользователем, и этому специалисту нужны инструментальные средства для быстрой подготовки ответов. Словом, вопросы сопровождения должны быть предусмотрены при разработке.

Сопровождение – это особое искусство. Старые программы (legacy – унаследованные) обычно состоят из сотен или даже тысяч модулей общим объемом в миллионы строк исходного кода. Вряд ли кто-то возьмется все прочитать, понять и запомнить. Обычно сопровождающий программист, исправляя ошибку, лишь примерно знает, где ее искать. Более того, после нахождения неправильного фрагмента не всегда ясно, как повлияет исправление на другие участки кода.

Часто бывает, что, исправив одну ошибку, получаешь две других.

Профессиональные организации, получив контракт на сопровождение большой программы, тратят время и деньги на ее изучение, анализ модульной структуры, потоков управления и т.д. Такое действие называется reverse engineering (возвратное проектирование). Иногда

используется термин knowledge mining (дословно – "откапывание знаний") – восстановление утраченных знаний о программе только на основе ее текста. Если есть подходящие инструментальные средства, то очень полезно удалить недостижимые участки кода (в старых программах их объем может достигать 30%), провести удаление goto и реструктуризацию программы (гораздо легче понимать стройную иерархическую структуру, чем "клубок змей" или "спагетти"). Есть и другие специальные средства, которые мы рассмотрим в следующем параграфе.

Реинжиниринг

Особый интерес вызывает вопрос, когда же следует прекращать сопровождение. На этот вопрос отвечает известная диаграмма:

Здесь по горизонтали растет важность программы для бизнеса компании, а по вертикали – сложность внесения исправлений. Так вот, если исправлять несложно, но и важность – небольшая, то надо просто сопровождать, исправляя ошибки (corrective fixing), если же программа важна для компании, то можно и улучшать ее (adaptive fixing). Если исправлять сложно, а важность – небольшая, то легче программу выкинуть, если же программа важна, содержит определенные знания бизнес-процессов, но сопровождать ее трудно, например, потому, что авторы программы покинули компанию или компания вынуждена сменить платформу, то программу следует подвергнуть реинжинирингу [15].

Попросту говоря, реинжиниринг – это перевод программы со старых языков (Кобол, ПЛ1, Адабас-Натурал и т.п.) на новые, такие как Java, Visual Basic, C++. На самом деле, все гораздо сложнее. Смена операционной системы, используемой базы данных, стиля организации диалога с пользователем – каждая из этих задач весьма нетривиальна. Если же добавить естественное желание реструктурировать программу, выкинуть отмершие части, перейти к объектно-ориентированной организации программы и т.д., да еще вспомнить, что все эти задачи требуют перебора и анализа путей в графе управления программы, число которых растет экспоненциально относительно числа вершин, то задача вообще выглядит нерешаемой. Мы потратили на нее много лет по заказу компании Relativity technologies (США, Северная Каролина) и создали продукт Rescueware, который известная консалтинговая компания Gartner Group уже дважды признала лучшим в мире в области legacy understanding и legacy transformation. Забавно, что сначала американцы обратились в несколько известных университетов США, но отовсюду получили ответ, объясняющий невозможность решения. На наше счастье, среди заказчиков был выходец из СССР (Лен Эрлих), который посоветовал обратиться к русским математикам, справедливо утверждая, что русские могут сделать то, чего американцы не могут. Совершенно случайно ГП "Терком" оказался в числе трех российских компаний, к которым обратились американцы. Опыта общения с иностранными заказчиками у нас не было, как определять цену работы, мы не знали. Кстати, с этим вопросом связана забавная, но поучительная история. Принимал я американцев в своем кабинете, наскоро переделанном из обычной аудитории. На полу в линолеуме была большая дырка, на которую мы не обращали внимания. Через много лет я узнал, что эта дырка уменьшила стоимость самого первого американского заказа ровно в два раза.

Мы долго ломали голову, как подступиться к этой задаче. Мы все были математиками, поэтому хорошо понимали, почему американские ученые отказались от нее. Но потом сообразили, что в большинстве случаев нам и не нужен полный перебор путей в графе, а в тех случаях, когда без него не обойтись, можно за один перебор решить сразу несколько задач или одну задачу, но сразу для многих переменных. На тему реинжиниринга у нас опубликовано множество работ, все они выложены на сайте кафедры системного программирования.

Чтобы дать хоть какое-то представление о системе Rescueware, перечислим некоторые ее возможности:

1. Инвентаризация. Осуществляется так называемый ослабленный синтаксический анализ (точный анализ затруднен из-за огромного числа диалектов старых языков), какие файлы включения используются, кто кого вызывает, что из этого есть в наличии. Случая не было, чтобы заказчик выдал все сразу, не забыв ни одного файла. На основании различных метрик оценивается сложность работы, трудоемкость и приблизительная стоимость.
2. Строится дерево синтаксического разбора (видонезависимый и видозависимый анализы), пользователю в диалоговом режиме предоставляется возможность дополнительного изучения исходного приложения, представленного в виде гипертекста – на экране одновременно в разных окнах видны дерево разбора и исходный тест. Если пользователь смещается по дереву, автоматически перемещается и курсор в исходном тексте, и наоборот. Гипертекст, как и другие программы визуализации, был разработан группой М. Бульонкова в Академгородке (г. Новосибирск), много лет работающей в тесном контакте с нами [16].
3. По оригинальным алгоритмам исходное приложение преобразуется таким образом, чтобы уменьшить или вообще свести к нулю число операторов goto, разбить приложение на процедуры или на объекты (напоминаю, что в языке Кобол вообще не было понятия "процедура"), удалить ставшие со временем недостижимыми участки кода; реализуются различные методы глобальной оптимизации. Здесь следует отметить, что в отличие от обычной компиляции, в которой главным требованием является семантическое соответствие объектного кода исходному, (при этом сам объектный код никто не читает), в процессе реинжиниринга создается новый исходный код на другом языке, который будет сопровождаться этими же или другими программистами, поэтому на первый план выходят такие странные для компиляции вопросы как "естественность" представления программы, структурное подобие исходному приложению, наглядное форматирование текста и т.п.
4. Генерация нового исходного текста на целевом языке с учетом только что перечисленных требований.

5. Особого упоминания заслуживает BRE – Business Rules Extraction – выделение бизнес-правил. Представьте себе, что у Вас есть приложение (как обычно, написанное на разных языках), которое выполняет 10 функций. 7 из них уже безнадежно устарели, оставшиеся 3 очень важны для вашего бизнеса. Приложение старое, описания исходных алгоритмов уже не сохранились или сильно отстали от реальной программы, авторы давно разбежались кто куда. Таким образом, единственным носителем актуального знания осталась работающая программа, но ее сопровождение обходится очень дорого, поэтому хотелось бы выделить из текста только те описания и операторы, которые задействованы в нужных трех функциях, а весь остальной текст выкинуть. Скорее всего, получившаяся программа будет больше чем 30% от исходного текста, но все-таки намного меньше, чем вся исходная программа, поэтому сопровождать ее будет проще и дешевле.

Идея BRE очень проста – находите в конце графа программы узлы, где возникают конечные значения нужных функций, и двигаетесь назад, выбирая только те узлы и ребра, которые влияют на промежуточные вычисления уже помеченные, как нужные.

Можно и наоборот – если известно, что из 100 входных данных интересными остаются только 40, а остальными 60 можно пренебречь, можно пройтись по графу программы слева направо аналогичным образом.

Разумеется, на практике все гораздо сложнее — чего стоят одни массивы, в которых разные элементы влияют на разные функции. Тем не менее, нам удалось добиться практически значимых результатов.

Управление качеством

Согласно современным международным стандартам, качество программного обеспечения, как и любого другого продукта, – это его соответствие потребностям заказчика [17]. Самый верный путь повысить качество ПО – улучшить процесс создания и сопровождения продукта.

Качество программного обеспечения (программных средств, инструментов, приложений), используемого в коммерческих и государственных структурах, служащего основой всемирной сети и разнообразных информационных систем, является критически важным фактором. Сегодня деятельность многих организаций, предприятий и, особенно, высокотехнологичных компаний напрямую зависит от качественной обработки информации соответствующими компьютерными системами.

Применение некачественного программного обеспечения может привести к серьезным потерям и затратам на восстановление утерянных данных и программной системы в целом. Предприятия наиболее уязвимы в период внедрения и использования новых программных продуктов и информационных систем. Именно в это время ущерб от некачественного программного обеспечения наиболее вероятен.

По данным исследования Тома де Марко (США, 1982 г.):

- 15% всех программных проектов так и не достигли своего завершения;
- превышение стоимости проектов на 100-200% не является чем-то необычным;
- превышение стоимости на 30% в программной индустрии считается успехом.

Проблемы, выявленные в ходе более чем 150 экспериментов по оценке различных программных проектов:

- помехи со стороны правительства – 10%;
- технологические факторы – 12%;
- внешние поступления – 12%;
- оборудование – 15%;
- недостаточность контроля – 35%;
- недостаточность/недостатки планирования производственных процессов – 55%.

Выдержка из статьи "Software’s Chronic Crisis", появившейся через 12 лет после публикации этого исследования (автор W. Wayt Gibbs, журнал

"Scientific American", сентябрь 1994 года): "Исследования показали, что на каждые 6 крупных систем программного обеспечения, запущенных в действие, приходится 2 (т.е. 30-35%) системы, разработка которых была прекращена из-за невозможности добиться удовлетворительного функционирования. Средний проект разработки программного обеспечения затягивается на половину первоначально запланированного срока, крупные проекты – и того хуже. Три четверти всех больших систем либо выполняют не все функции, которые на них возлагались, либо не используются вовсе".

По данным Департамента по Торговле и Промышленности Великобритании (DTI), при внедрении проектов информационных технологий на предприятиях потери из-за некачественного программного обеспечения составляют в среднем около 20% от общего объема потерь. По разным оценкам, аналогичный показатель для России достигает величины от 30 до 50%.

Может показаться, что управлять количеством ошибок в коде невозможно и их число зависит только от способностей конкретного разработчика. Это не совсем так. В конце 80-х – начале 90-х годов в программной индустрии был проведен ряд серьезных исследований эффективности труда программистов.

Вот данные американского института программной инженерии SEI. Профессиональные программисты со стажем 10 и более лет на 1000 строк кода (СК – строка исходного текста, которая содержит программную инструкцию) допускают в среднем 131,3 ошибки. Следовательно, большая система размером миллион СК может содержать 100 тысяч ошибок! Из них до 50% выявляется на этапе компиляции (если транслятор имеет развитую систему предупреждений). На этапе тестирования отдельных модулей (типичный программный модуль по определению SEI содержит от 5 до 5 тысяч СК) обнаруживается половина оставшихся ошибок. Получается, что перед этапами внедрения и комплексного тестирования в продукте еще скрывается 25 тыс. ошибок. На их устранение на заключительных этапах тратится от 10 до 40 человеко-часов на ошибку, т.е. на доведение продукта до идеального состояния потребуется 125 человеко-лет работы!

Другая статистика. Типичный небольшой проект имеет объем 50 тысяч СК. Его создают 5 программистов, делая при этом 100 ошибок на тысячу СК. 50% ошибок выявляется на этапе компиляции с незначительными расходами времени, устранение ошибок на этапе тестирования занимает 90% времени. Стоимость устранения одной ошибки в готовом продукте оценивается в 4 тыс. долл. (по данным IBM, устранение ошибок в продуктах компании, введенных в эксплуатацию, обходится в 20 тысяч долларов на ошибку).

Корень проблемы — в неправильных акцентах при управлении качеством ПО. Корреляция между числом ошибок, обнаруженных при тестировании отдельных модулей, и числом ошибок, найденных пользователями в готовом продукте, равна 0,91. Отсюда вывод – если на тестирование поступит некачественный продукт, он таким и будет выпущен в продажу!

Таким образом, организация производства качественного программного обеспечения является ключевой проблемой управления процессом разработки ПО, которой в индустриально развитых странах стали активно заниматься примерно с середины 1960-х годов. К числу бесспорных достижений теории менеджмента качества относятся современное понятие качества и его смысловое наполнение, а также известные модели систем качества серии ISO 9000.

Анализ известных нормативных документов дает основание предполагать, что при коммерческой эксплуатации программного продукта к числу наиболее значимых показателей качества, базирующихся на обоснованных претензиях пользователей, можно отнести:

- неадекватность функционирования программного продукта;
- недостаточное взаимодействие продукта с другими программными, аппаратными и телекоммуникационными средствами;
- отказы программного продукта в процессе применения по назначению;
- замедленное время работы программного продукта и задержки предоставления им промежуточной и выходной информации;
- неполнота отражения информации;

- несоответствие хранимых данных и информации, вводимой оператором;
- потеря актуальности информации, циркулирующей в информационной системе;
- нарушения конфиденциальности информации;
- содержание сопроводительной документации и справочной системы программного продукта.

Кроме таких "первичных" данных о качестве, идущих непосредственно от потребителя, в роли показателей могут выступать: число строк кода в стандартном модуле, количество выявленных ошибок на 1000 строк кода, вероятность появления специфических ошибок, параметры сложности программы, стоимость единицы кода, цена "человеко-месяца", статистические характеристики процессов (математические ожидания, дисперсии, корреляционные функции и т. д.) и другие оценочные параметры.

Отсюда следует "генеральная" задача разработчика ПО – на основе анализа взаимной корреляции ранжированных потребительских, технологических и технических характеристик создать систему целевых показателей (метрик), которая задает ориентиры разработки и критерии оценки ее качества.

Такая задача трудна даже для искушенного разработчика, однако использование ранее наработанного опыта и знаний, накопленных в межпроектных базах данных и отраженных в корпоративных и международных стандартах, позволяют существенно уменьшить "размерность" задачи.

Ранее было отмечено, что из-за отсутствия адекватных систем управления качеством во многих проектах (особенно крупных) временные и экономические показатели значительно выше запланированных. Но даже превышение показателей не гарантирует выполнения технических требований. Ряд крупных IT-проектов, как в Европе, так и в Америке, не достигли заявленных результатов, будучи не в состоянии реализовать требуемые технические и технологические параметры, хотя отпущенные время и средства были значительно превышены.

Система управления качеством является частью системы управления организацией. Цели в сфере качества дополняют основные (стратегические) задачи организации. Различные части системы управления организации-разработчика могут быть объединены вместе с системой менеджмента качества в единую, унифицированную систему управления с общими элементами. Это способствует эффективному планированию, распределению ресурсов, установлению взаимодополняющих целей и реальной оценке эффективности.

Инициативы внедрения систем качества в широких масштабах в Японии в начале 50-х годов, поддержанные правительственными программами, обеспечили быстрый рост конкурентоспособности и выход страны на лидирующие позиции в ряде областей промышленности. Активное внедрение новых решений, направленных на обеспечение качества, в США и Европе началось в начале 60-х годов.

Если говорить о программировании, то идеи реализации качества на базе создания стандартного процесса разработки и сопровождения ПО пришли в эту область из промышленности в ответ на программный кризис конца 60-х годов [Wheeler Sh., Duggins Sh. Improving software quality – ACM Proceedings of the 36th annual conference on South-East regional conference, April 1998].

Среди стандартов в области разработки систем качества, оценки качества процессов и уровня зрелости компаний, разрабатывающих программное обеспечение, сегодня наиболее популярными являются: ISO 9000 (версии 1994 и 2000 гг.), ISO 12207, TickIT, SEI SW-CMM, Trilium, ISO 15504 (SPICE), CMMI.

Компьютерная программа представляет собой материальный объект, но она строится на абстрактных идеях и состоит из сложнейших виртуальных конструкций. Это в корне отличает ее от физических объектов, с которыми имеет дело обычное производство, и поэтому невозможно создать качественное ПО, не решив ряд типичных проблем:

- команда разработчиков, как правило, состоит из творческих личностей, которых часто бывает трудно привести к "общему знаменателю";
- каждый программный продукт неизбежно содержит ошибки,

отражающие квалификацию и индивидуальный стиль разработчика – разнообразие и нестандартность ошибок сильно усложняет процесс достижения стандартных целей в области качества;

- каждый успешный проект по-своему уникален. Он подобен мозаике со сложным рисунком, поэтому очень трудно выделить из него некий базовый процесс-клише, который можно было бы применить в дальнейших разработках;
- по сходной причине трудно поставить производство сложного и уникального ПО на поток, так как часто для его разработки требуется создание сопутствующего специального программного "инструментария" для проектирования, оптимизации и тестирования;
- одна из специфических проблем программирования состоит в том, что продуктивность в этой области растет очень медленно, если растет вообще – по некоторым оценкам, средний программист способен создать 10-50 строк операторов в день. Кроме того, эти оценки должны быть уменьшены для больших систем, так как увеличенная нагрузка требует значительных затрат (в относительных единицах). Это обстоятельство резко отделяет программирование от других видов деятельности, где поточное производство радикально уменьшает цену единицы продукта. Вся экономика строится на этом далеко не новом принципе – но только не производство программного обеспечения!

В последние годы постепенно сложилась определенная культура, появились традиции и стиль работы в сфере создания программных продуктов.

	Продукт (методики разработки)		Проект (методики менеджмента проектов)		Персонал (методики управления персоналом)
1	Определение продукта (состав, клиентская среда и требования)	1	Оценка стоимости проекта в целом (как правило, в человеко-месяцах)	1	Набор персонала (как правило, на конкурсной основе)
	Процессы				Отбор команды

2	оценивания (определения критериев)	2	общего и оперативных планов	2	проекта и обучение персонала
3	Оценка альтернативных подходов к реализации продукта	3	Оценка трудозатрат в структуре проекта	3	Создание команды проекта и распределение ответственности
4	Управление требованиями	4	Оценка и менеджмент рисков	4	Оценка потенциальных возможностей участников проекта
5	Управление субподрядом	5	Создание структуры пооперационного перечня работ	5	Создание структуры пооперационного перечня работ
6	Выполнение начальной оценки	6	Оценка ключевых стадий и составление графика работ	6	Взаимодействие и общение на уровне групп и команды в целом
7	Отбор инструментов, методов, стандартов, разработка внутренних стандартов	7	Отбор инструментов управления проектом	7	Обучение проектных команд для получения оптимальных результатов
8	Настройка процессов на выполнение требований ТЗ	8	Знание стандартов процесса и настройка стандартов на процесс	8	Эффективное представление и использование навыков
9	Понимание действий по разработке продукта (полное	9	Формирование метрических показателей для измерения	9	Успешное ведение переговоров с внешними и внутренними

9	представление о необходимых действиях в течение всего ЖЦ)	9	параметров процессов и управления процессами	9	участниками проекта, совместные обсуждения и оценки
10	Управление изменениями	10	Отслеживание процессов (процессный мониторинг и совершенствование качества)	10	Организация эффективных встреч для планирования совместных работ
11	Отслеживание качества продукта в ходе разработки (мониторинг качества рабочих продуктов)	11	Отслеживание процесса разработки (мониторинг соответствия стандартного процесса разработки ПО принятым требованиям)	11	Вопросы индивидуальной и коллективной интеллектуальной собственности
		12	Отслеживание хода разработки проекта		

Наилучшие практики, взятые на вооружение ведущими компаниями, образовали так называемые "Тридцать четыре необходимые компетенции для управления программными проектами, стандартизации процесса разработки ПО и реализации его качества".

Эти "компетенции" нашли отражение в большом количестве национальных и корпоративных стандартов разных стран, которые послужили основой для создания нескольких международных стандартов, широко применяемых в настоящее время.

Стандарты ISO, SW-CMM. CASE-технологии

Стандарты ISO

В 1947 году в Лондоне представители 25 стран решили создать международную организацию, основной задачей которой стала бы координация разработок и унификация международных стандартов. Новая организация получила название International Organization for Standardization (ISO). В настоящее время ее членами являются около 100 стран.

Главной причиной появления единых стандартов стало желание инициаторов создания этой организации устранить технические барьеры в торговле, которые возникли вследствие того, что в разных странах для одних и тех же технологий и товаров действовали разнородные стандарты. Сегодня стандартами ISO "перекрыты" многие технологические отрасли – от программирования и телекоммуникаций до банковской и финансовой сферы.

По уставу членом ISO может стать "самая авторитетная в стране организация, занимающаяся выработкой стандартов". Таким образом, интересы какой-либо страны в ISO может представлять только одна организация. Помимо основных членов, в ISO входят так называемые члены-корреспонденты (как правило, ими становятся относительно крупные "стандартообразующие" организации отдельных стран, однако еще недостаточно мощные, чтобы распространить свое влияние на стандартизацию технологий по всей стране). Члены-корреспонденты не принимают активного участия в разработке международных стандартов, но имеют полный доступ к интересующей их информации. Наконец, есть еще и члены-подписчики – они представлены организациями развивающихся стран. Россия на сегодня имеет статус члена-корреспондента ISO.

Выполнение технической работы в ISO возложено на 2700 технических комитетов, подкомитетов и рабочих групп, в состав которых входят представители правительственных, промышленных, научно-исследовательских и юридических кругов (всего около 500 организаций). Каждая организация – член ISO – имеет право включить

своего представителя в любой комитет, в деятельности которого она особо заинтересована.

Новые стандарты рождаются в соответствии с тремя принципами.

Во-первых, они являются результатом консенсуса всех заинтересованных сторон – производителей, поставщиков, потребителей, профессиональных разработчиков, правительственных и исследовательских организаций.

Во-вторых, стандарты имеют действительно мировое распространение и удовлетворяют как производителей, так и потребителей.

В-третьих, появление новых стандартов диктуется исключительно требованиями свободного рынка, а не чьей-то злой или доброй волей. Если рынок созрел для нового стандарта, то такой стандарт появляется.

Процесс создания нового стандарта включает три этапа. Обычно инициатива его разработки исходит от производителей, которые доводят базовые предложения стандарта до своего представителя в ISO. Если эта организация признает целесообразность создания нового стандарта, то соответствующая рабочая группа определяет техническую область, на которую предполагаемый стандарт будет распространяться. На втором этапе происходит выработка технических спецификаций, в ходе которой представители различных стран стремятся достичь консенсуса. На заключительном этапе первая версия стандарта утверждается (за стандарт должно проголосовать 75% кворума) и публикуется.

По мере совершенствования технологий, появления новых материалов, методов обработки, повышения требований к качеству и надежности изделий возникает необходимость в пересмотре стандартов. В ISO существует правило: все стандарты должны пересматриваться не реже чем раз в пять лет. Сегодня "перу" ISO принадлежит около 9300 различных стандартов, описание которых занимает 171000 страниц текста на английском языке.

Серия ISO 9000 [18] (управление качеством) включает в себя следующие стандарты:

- ISO 9000-1 (1994 г.). Управление качеством и гарантии качества. Часть 1. Руководство по выбору и использованию.
- ISO 9000-2 (1993 г.). Управление качеством и гарантии качества. Часть 2. Общее руководство по применению стандартов ISO 9001, ISO 9002 и ISO 9003.
- ISO 9000-3 (1991 г.). Управление качеством и гарантии качества. Часть 3. Руководство по применению стандарта ISO 9001 при разработке, установке и сопровождении ПО.
- ISO 9000-4 (1993 г.). Управление качеством и гарантии качества. Часть 4. Руководство по управлению надежностью программ.

Основополагающий стандарт ISO 9001 (1994 г.) задает модель системы качества для процессов проектирования, разработки, производства, установки и обслуживания (продукта, системы, услуги).

Основным преимуществом моделей ISO серии 9000 является их известность, распространенность, признание на мировом уровне, большое количество экспертов и аудиторов и невысокая стоимость услуг сертификации. Универсальность же моделей ISO серии 9000 имеет определенные недостатки: они являются достаточно высокоуровневыми, задают абстрактные модели и не содержат конкретных методологических разработок.

По ISO качество – это совокупность свойств и характеристик продукта, процесса или услуги, которые обеспечивают способность удовлетворять заявленным или подразумеваемым потребностям. Современные способы обеспечения качества базируются на подходах TQM (Total Quality Management). Это управление ресурсами и применение количественных методов анализа для улучшения: разработок, материалов и услуг, поставляемых в организацию; всех процессов внутри организации; степени удовлетворенности настоящих и будущих потребностей клиентов.

В модели ISO 9000 лишь упоминаются требования, которые должны быть реализованы, но не говорится, как это можно сделать. Поэтому для построения полноценной системы качества по ISO помимо основной модели ISO 9001 (1994 или 2000 года), необходимо использовать вспомогательные отраслевые и рекомендательные стандарты. Для организации, занимающейся разработкой программного обеспечения,

такими стандартами являются: ISO 9004-1:94 (ISO 9004:2000), ISO 8402:94 (ISO 9000:2000), ISO 9000-3:91, ISO 10007:95, ISO 10013:95, ISO 12207:95.

Семейство стандартов ISO 9000 версии 2000 года разработано с тем чтобы преодолеть недостатки ISO 9000 версии 1994 года и помочь организациям всех специализаций, типов и размеров внедрить и использовать эффективные системы менеджмента качества.

Подход к системам менеджмента качества является общим и применяется к организациям в любой отрасли экономики, поэтому данный стандарт не устанавливает каких-либо конкретных требований к программным продуктам. Требования к ним могут определяться заказчиками или третьими лицами и содержаться в технических спецификациях, стандартах на продукт, стандартах на процесс, контрактных соглашениях и нормативных документах.

В основу построения системы качества в соответствии с моделью ISO 9000:2000 закладываются следующие принципы:

- концентрация на потребностях заказчика;
- активная лидирующая роль руководства;
- вовлечение исполнителей в процессы совершенствования;
- реализация процессного подхода;
- системный подход к управлению;
- обеспечение непрерывных улучшений;
- принятие решений на основе фактов;
- взаимовыгодные отношения с поставщиками.

При этом методически в полном соответствии с дисциплиной построения сложных систем в стандарте ISO 9000:2000 предусматривается, с одной стороны, построение организационной системы "сверху — вниз": от целей предприятия и его политики – к организационной структуре и формированию бизнес-процессов, и с другой – итеративное развитие организационной системы через механизмы измерения и улучшения.

Внедрение системы менеджмента качества организацией–разработчиком программных продуктов по ISO 9000 версии 2000 года

состоит из нескольких этапов. В их числе выделяются:

- измерение характеристик продуктов для определения эффективности каждого процесса, направленного на достижение соответствующего качества;
- применение результатов измерений для определения текущей эффективности процессов создания и внедрения продуктов;
- определение способов предотвращения дефектов, снижения изменяемости продукции и минимизации доработок;
- поиск возможностей по снижению рисков и улучшению эффективности и производительности технологических и иных процессов;
- выявление и расстановка в порядке важности тех улучшений, которые могут давать оптимальные результаты с приемлемыми рисками;
- планирование стратегии, процессов и ресурсов для получения идентифицированных улучшений продукции;
- контроль результатов улучшений;
- сравнение полученных результатов с ожидаемыми;
- определение подходящих корректирующих действий.

Реализация этих этапов возможна только при наличии в организации системы критериев, показателей и факторов качества, а также методов их измерения и оценки.

Capability Maturity Model for Software (Модель SEI SW-CMM)

В 1982 году Министерство обороны США образовало комиссию, основной задачей которой стало исследование проблем, возникающих при разработке программных продуктов в организациях министерства. В результате деятельности комиссии в декабре 1984 году был создан Институт инженеров-разработчиков программного обеспечения (Software Engineering Institute, SEI) на базе Университета Карнеги-Меллона в Питсбурге.

В 1986 году SEI и корпорация Mitre под руководством Уоттса Хамфри (Watts Humphrey) приступили к разработке критериев оценки зрелости

технологических процессов – Capability Maturity Model (CMM) [19].

Далее события развивались в следующем порядке.

1987 г. SEI публикует: краткое описание структуры CMM; методы оценки процессов разработки ПО; методы оценки зрелости процессов производства ПО; анкету для выявления степени зрелости процессов (для проведения самостоятельного, внутреннего аудита и внешнего аудита).

1991 г. Выпуск версии CMM v1.0.

1992 г. Выпуск версии CMM v1.1.

1997 г. Выпуск очередной (усовершенствованной) версии CMM.

Методология CMM разрабатывалась и развивалась в США как средство, позволяющее выбирать лучших производителей ПО для выполнения госзаказов. Для этого предполагалось создать критерии оценки зрелости ключевых процессов компании-разработчика и определить набор действий, необходимых для их дальнейшего совершенствования. В итоге методология оказалась чрезвычайно полезной для большинства компаний, стремящихся качественно улучшить существующие процессы проектирования, разработки, тестирования программных средств и свести управление ими к понятным и легко реализуемым алгоритмам и технологиям, описанным в едином стандарте.

CMM де-факто стал именно таким стандартом. Его применение позволяет поставить разработку ПО на промышленную основу, повысить управляемость ключевых процессов и производственную культуру в целом, гарантировать качественную работу и исполнение проектов точно в срок. Основой для создания CMM стало базовое положение о том, что фундаментальная проблема "кризиса" процесса разработки качественного ПО заключается не в отсутствии новых методов и средств разработки, а в неспособности компании организовать технологические процессы и управлять ими.

Для оценки степени готовности предприятия разрабатывать качественный программный продукт CMM вводит ключевое понятие зрелость организации (Maturity). Незрелой считается организация, в

которой:

- отсутствует долговременное и проектное планирование;
- процесс разработки программного обеспечения и его ключевые составляющие не идентифицированы, реализация процесса зависит от текущих условий, конкретных менеджеров и исполнителей;
- методы и процедуры не стандартизированы и не документированы;
- результат не предопределен реальными критериями, вытекающими из запланированных показателей, применения стандартных технологий и разработанных метрик;
- процесс выработки решения происходит стихийно, на грани искусства.

В этом случае велика вероятность появления неожиданных проблем, превышения бюджета или невыполнения сроков сдачи проекта. В такой компании, как правило, менеджеры и разработчики не управляют процессами – они вынуждены заниматься разрешением текущих и спонтанно возникающих проблем. Отметим, что на данном этапе развития находится большинство российских компаний.

Основные признаки зрелой организации:

- в компании имеются четко определенные и документированные процедуры управления требованиями, планирования проектной деятельности, управления конфигурацией, создания и тестирования программных продуктов, отработанные механизмы управления проектами;
- эти процедуры постоянно уточняются и совершенствуются;
- оценки времени, сложности и стоимости работ основываются на накопленном опыте, разработанных метриках и количественных показателях, что делает их достаточно точными;
- актуализированы внешние и созданы внутренние стандарты на ключевые процессы и процедуры;
- существуют обязательные для всех правила оформления методологической программной и пользовательской документации;
- технологии незначительно меняются от проекта к проекту на

основании стабильных и проверенных подходов и методик;
- максимально используются наработанные в предыдущих проектах организационный и производственный опыт, программные модули, библиотеки программных средств;
- активно апробируются и внедряются новые технологии, производится оценка их эффективности.

CMM определяет пять уровней технологической зрелости компании, по которым заказчики могут оценивать потенциальных претендентов на заключение контракта, а разработчики – совершенствовать процессы создания ПО.

Каждый из уровней, кроме первого, состоит из нескольких ключевых областей процесса (Key Process Area), содержащих цели (Goal), обязательства по выполнению (Commitment to Perform), осуществимость выполнения (Ability to Perform), выполняемые действия (Activity Performed), их измерение и анализ (Measurement and Analysis) и проверку внедрения (Verifying Implementation). Таким образом, CMM фактически является комплексом требований к ключевым параметрам эффективного стандартного процесса разработки ПО и способам его постоянного улучшения. Выполнение этих требований, в конечном счете, увеличивает вероятность достижения предприятием поставленных целей в области качества.

Начальный уровень (Initial Level – Level 1).

К данному уровню относится компания, которой удалось получить заказ, разработать и передать заказчику программный продукт. Стабильность разработок отсутствует. Лишь некоторые процессы определены, результат всецело зависит от усилий отдельных сотрудников. Успех одного проекта не гарантирует успешности следующего. К этой категории можно отнести любую компанию, которая хоть как-то исполняет взятые на себя обязательства.

Ключевые области процесса этого уровня не зафиксированы.

Повторяемый уровень (Repeatable Level – Level 2).

Этому уровню соответствуют предприятия, обладающие определенными технологиями управления и разработки. Управление

требованиями и планирование в большинстве случаев основываются на разработанной документированной политике и накопленном опыте. Установлены и введены в повседневную практику базовые показатели для оценки параметров проекта. Менеджеры отслеживают выполнение работ и контролируют временные и производственные затраты.

В компании разработаны некоторые внутренние стандарты и организованы специальные группы проверки качества (QA). Изменения версий конечного программного продукта и созданных промежуточных программных средств отслеживаются в системе управления конфигурацией. Имеется необходимая дисциплина соблюдения установленных правил. Эффективные методики и процессы институционализируются (устанавливаются), что обеспечивает возможность повторения успеха предыдущих проектов в той же прикладной области.

Ключевые области процесса разработки ПО этого уровня:

- Управление требованиями (Requirements management).
- Планирование проекта разработки ПО (Software project planning).
- Отслеживание хода проекта и контроль (Software project tracking and oversight).
- Управление субподрядчиками разработки ПО (Software subcontract management).
- Обеспечение уверенности в качестве разработки ПО (Software quality assurance).
- Управление конфигурацией продукта (Software configuration management).

Определенный уровень (Defined Level – Level 3).

Уровень характеризуется детализированным методологическим подходом к управлению (то есть описаны и закреплены в документированной политике типичные действия, необходимые для многократного повторения: роли и ответственность участников, стандартные процедуры и операции, порядок действий, количественные показатели и метрики процессов, форматы документов и пр.).

Для создания и поддержания методологий в актуальном состоянии в организации уже подготовлена и постоянно функционирует специальная группа. Компания регулярно проводит тренинги для повышения профессионального уровня своих сотрудников.

Начиная с этого уровня, организация практически перестает зависеть от личностных качеств конкретных разработчиков и не имеет тенденции опускаться на нижестоящие уровни. Эта независимость обусловлена продуманным механизмом постановки задач, планирования мероприятий, выполнения операций и контроля исполнения.

Управленческие и инженерные процессы документированы, стандартизированы и интегрированы в унифицированную для всей организации технологию создания ПО. Каждый проект использует утвержденную версию этой технологии, адаптированную к особенностям текущего проекта.

Ключевые области процесса разработки ПО этого уровня:

- Цель упорядочивания работы организации (Organization Process Focus).
- Определение (стандартного) процесса организации (Organization Process Definition).
- Программа обучения (Training Program).
- Интегрированное управление разработкой ПО (Integrated Software Management).
- Технология разработки программных продуктов (Software Product Engineering).
- Межгрупповая координация (Intergroup Coordination).
- Экспертные (совместные) оценки коллег (Peer Reviews).

Управляемый уровень (Managed Level – Level 4).

Уровень, при котором разработаны и закреплены в соответствующих нормативных документах количественные показатели качества. Более высокий уровень управления проектами достигается за счет уменьшения отклонений различных показателей проекта от запланированных. При этом систематические изменения в производительности процесса (тенденции, тренды) можно выделить из

случайных вариаций (шума) на основании статистической обработки результатов измерений по процессам, особенно в хорошо освоенных и достаточно формализованных процессных областях.

Ключевые области процесса разработки ПО этого уровня:

- Количественное управление процессом (Quantitative Process Management).
- Управление качеством ПО (Software Quality Management).

Оптимизирующий уровень (Optimizing Level – Level 5).

Для этого уровня мероприятия по совершенствованию рассчитаны не только на существующие процессы, но и на внедрение, использование новых технологий и оценку их эффективности. Основной задачей всей организации на этом уровне является постоянное совершенствование существующих процессов, которое в идеале направлено на предотвращение известных ошибок или дефектов и предупреждение возможных. Применяется механизм повторного использования компонентов от проекта к проекту (шаблоны отчетов, форматы требований, процедуры и стандартные операции, библиотеки модулей программных средств).

Ключевые области процесса разработки ПО этого уровня:

- Предотвращение дефектов (Defect Prevention).
- Управление изменением технологий (Technology Change Management).
- Управление изменением процесса (Process Change Management).

CMM определяет следующий минимальный набор требований: реализовать 18 ключевых областей процесса разработки ПО, содержащих 52 цели, 28 обязательств компании, 70 возможностей выполнения (гарантий компании) и 150 ключевых практик.

В результате аудита и аттестации компании присваивается определенный уровень, который при последующих аудитах в дальнейшем может повышаться или понижаться. Следует отметить, что каждый следующий уровень в обязательном порядке включает в себя

все ключевые характеристики предыдущих. В связи с этим сертификация компании по одному из уровней предполагает безусловное выполнение всех требований более низких уровней.

К преимуществам модели SEI SW-CMM относится то, что она ориентирована на организации, занимающиеся разработкой программного обеспечения. В данной модели удалось более детально проработать требования, специфичные для процессов, связанных с разработкой ПО. По этой причине в SEI SW-CMM приведены не только требования к процессам организации, но и примеры реализации таких требований.

Основной же недостаток SW-CMM заключается в том, что модель не авторизована в качестве стандарта ни международными, ни национальными органами по стандартизации. Впрочем, CMM давно уже стала промышленным стандартом де-факто.

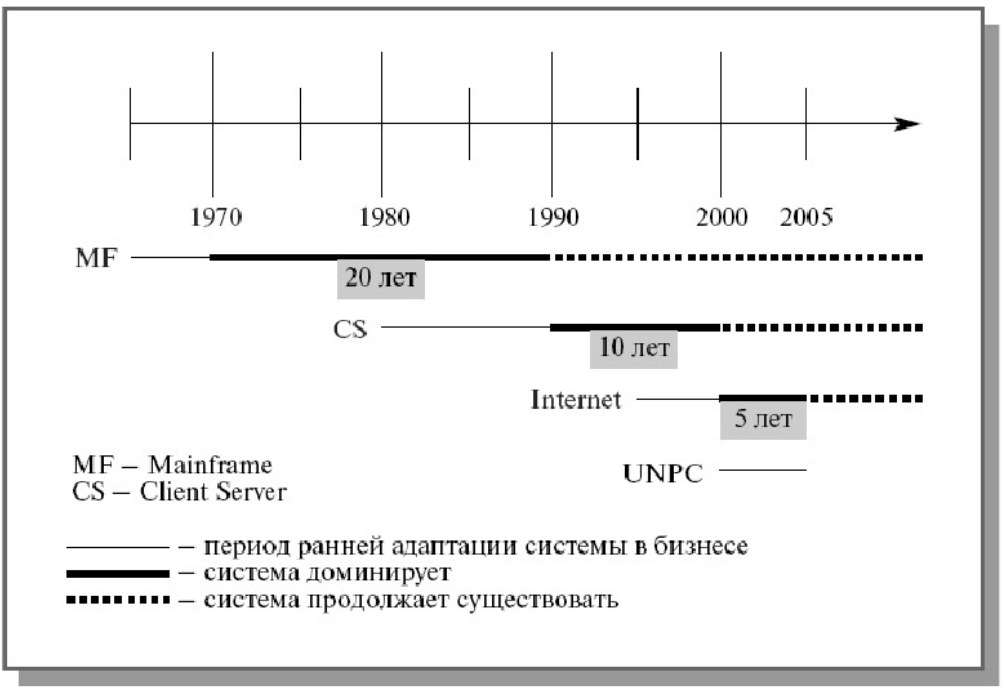
К недостаткам данной модели необходимо отнести также большие внешние накладные расходы на приведение процессов компании в соответствие модели CMM, нежели к моделям ISO 9000. Это связано с меньшей распространенностью модели в мире, меньшим количеством консалтинговых органов и экспертов и, в результате, с гораздо большими внешними затратами на консалтинг и на подтверждение соответствия процессов независимой третьей стороной. Тем не менее, CMM, несомненно, полезнее ISO 9000.

CASE-технологии

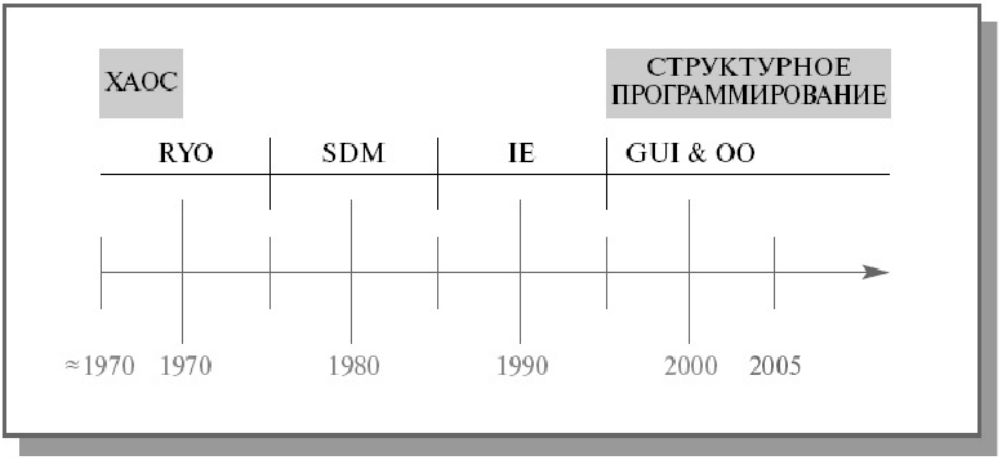
На протяжении всей истории программирования программные проекты все более и более усложнялись, объем работ стремительно увеличивался (особенно это проявилось в бизнес-приложениях), возникла потребность в таком универсальном средстве, которое могло бы помочь как-то структурировать, упорядочить и даже автоматизировать создание ПО. Проблема была глубже — необходимо было как-то объединить заказчиков, разработчиков, программистов, пользователей — причем в условиях постоянно меняющейся ситуации. А для того, чтобы о чем-то договориться, нужен какой-то общий язык. Традиционные языки программирования в силу малой наглядности, избыточности и многословия для этой роли не подходили, и, в конце

концов, стали предприниматься попытки создания четкого графического языка. Реализации графических языков и методологии их использования способствовали появлению программно-технологических средств специального класса — CASE-средств [21]. Аббревиатура CASE расшифровывается как Computer-Aided Software Engineering, т.е. разработка ПО с помощью компьютера.

Изобразим ось времени, в качестве начальной точки возьмем 1965 год. Именно тогда начался период раннего внедрения компьютеров в бизнесе. В то время применялись в основном мэйнфреймы (mainframes) — большие ЭВМ коллективного пользования.



С 1970 по 1990 год мэйнфреймы доминировали на рынке. Мэйнфрейм в то время – это, прежде всего, IBM 360/370. Все данные хранились и обрабатывались на одной очень большой ЭВМ, пользователи работали за экранами терминалов, которые могли только отображать информацию, но никакой обработки не производили. В СССР скопировали мэйнфреймы под именем ЕС ЭВМ. Были клоны IBM 360/370 в Англии, ФРГ и Японии.



В середине 1980-х годов появляется альтернатива мэйнфреймам: системы клиент-сервер (client-server), которые занимали ведущее положение с 1990 по 2000-е годы. В этом случае на клиентском рабочем месте уже могла осуществляться какая-то обработка данных, например, их форматирование, распаковка, простые расчеты. Основные вычисления по-прежнему выполнялись в центре (на сервере).

В середине 1990-х начинается адаптация Internet-систем, которые доминируют с середины 2000-го года. Сейчас начинают появляться новые системы, которым раньше не было аналогов; их принято называть "не ПК" (un-PC). В их число можно включить мобильные телефоны, карманные компьютеры, системы управления автомобилем (например, стоимость программного обеспечения автомобиля Ford уже превышает стоимость железа, из которого он сделан) и многое другое.

Характерно, что с развитием компьютерных технологий они становятся дешевле, а сами компьютеры – более доступными (особенно это проявилось в 1980-90-е гг.); как следствие, создается больше систем на базе каждой конкретной технологии.

Прослеживается такая закономерность: период доминирования каждой последующей технологии сокращается вдвое; одновременно все более многочисленными и масштабными становятся создаваемые системы.

Посмотрим на эволюцию подхода к проектированию систем.

Развитие методологии проектирования

С середины 1960-х до середины 1970-х годов программы преимущественно писались по принципу RYO (Roll Your Own), т. е. каждый писал так, как хотел и как умел — не было определенных подходов к процессу разработки ПО. В середине 70-х годов прошлого столетия возникла идея структурного программирования (SDM, Structure Design Methodology), происходит развитие методологии программирования и технологии моделирования. В Европе главным идеологом структурного программирования считается Дейкстра, а в США – ДеМарко. Разработчики приходят к пониманию, что систему (программу) можно описывать без использования конструкций языка программирования, а на более абстрактном уровне. В этот период времени программисты (и не только) поняли, что моделирование играет немаловажную роль, а вместе с ним и правила для моделирования. К этому моменту относится и введение блок-схем (flow charts). Для этого этапа развития технологии моделирования характерно, что центром программного продукта является процесс, а не данные (для хранения данных использовались индексные файлы и иерархические базы данных).

В середине 1980-х годов происходит очередной прорыв: появляются реляционные базы данных, один из авторов которых — James Martin. Главной частью программного продукта становятся данные и базы данных. Теоретиками была создана реляционная алгебра [21], ставшая основой для построения реляционных баз данных. Отсюда и новый подход к моделированию систем — IE (Information Engineering — методы и средства проектирования прикладных и информационных программ), в которых за основу принимаются не процессы, а структуры обрабатываемых данных.

С появлением персональных компьютеров на уровне систем клиент-сервер развивается графический интерфейс (GUI, Graphic User Interface). В связи с этим рождается объектно-ориентированный подход к проектированию (ОО), объединивший в одной сущности программу и данные.

Стоит отметить два вида объектно-ориентированного программирования, которые по сей день сосуществуют, хотя и велись

споры о правильности каждого из них и нелогичности другого. Object Action заключается в том, что сначала выбирается объект, а потом уже реализовываются его действия, которые впоследствии будут использованы. Action Object же, наоборот, предлагает продумать необходимые действия, а затем выбрать, какой именно объект будет их реализовывать.

Главные составляющие CASE-продукта таковы:

- методология (Method Diagrams), которая задает единый графический язык и правила работы с ним. CASE-технологии обеспечивают всех участников проекта, включая заказчиков, единым, строгим, наглядным и интуитивно понятным графическим языком, позволяющим получать обозримые компоненты с простой и ясной структурой. При этом программы представляются двумерными диаграммами (которые проще в использовании, чем многостраничные описания), позволяющими заказчику участвовать в процессе разработки, а разработчикам — общаться с экспертами предметной области, разделять деятельность системных аналитиков, проектировщиков и программистов, облегчая им защиту проекта перед руководством, а также обеспечивая легкость сопровождения и внесения изменений в систему.
- графические редакторы (Graphic Editors), которые помогают рисовать диаграммы; возникли с распространением PC и GUI. Этими двумя составляющими (так называемые upper case технологии) CASE-технологии поначалу и были ограничены. Диаграммы стало легко рисовать, их появилось множество, но пользы от них было мало – проектирование было развито лишь на уровне рисования. Существовало много проблем: никто не знал все используемые в тот момент технологии (не мог писать и для мэйнфреймов, и для клиента, и для сервера); неясно было, как объединять написанное для разных платформ.
- генератор: по графическому представлению модели можно сгенерировать исходный код для различных платформ (так называемая low case часть CASE-технологии). Генерация программ позволяет автоматически построить до 85-90% объектного кода или текстов на языках высокого уровня, но только для хорошо формализуемых частей программы (прежде всего, для

описания баз данных и для задания форм ввода-вывода информации). Сложная обработка, как обычно, может быть описана с помощью ручного программирования.
- репозиторий, своеобразная база данных для хранения результатов работы программистов (сложилась парадоксальная ситуация: к тому моменту базами данных пользовались все, кроме программистов), происходит переход от "плоских" файлов к системе хранения информации о разработке проекта.

Примеры CASE-средств

Если сначала диаграммы рисовались вручную, то в середине 1980-х годов появляются первые продукты, реализующие CASE-технологию. Компания TI (Texas Instruments) выпускает продукт IEF (Information Engineering Facility), компания KW (Knowledge Ware) создает ADW. Целевыми платформами обоих продуктов были только мэйнфреймы,что являлось их основным недостатком. В 1986 году начинаются разработки продукта HPS (High Productivity System), а в 1990 году образуется компания SEER, которая выпускает HPS на рынок. В 1992-1993гг. по заказу этой компании мы полностью переписали HPS, сохранив их замечательные бизнес-идеи.

Технологии оказались востребованными на рынке: на 1995 год объемы продаж IEF равнялись 200 млн. долл., ADW — 150 млн. долл., HPS — 120 млн. долл. У продукта HPS был более высокий уровень генерации кода (lower case), но более слабый уровень методологии и графических редакторов (upper case).

Годы	Организации	Тип систем
1965 — 1975	немногие	Изолированные
1975 — 1985	многие	Изолированные
1985 — 1995	многие	Enterprise level
1995 — 2000	многие	Department level
с 2000	многие	SOA

В середине 1980-х годов мы также разработали технологию RTST, которая включала в себя все перечисленные выше компоненты, в том

числе генератор в Алгол 68. В чем-то мы даже превзошли американцев, поскольку они умели генерировать только экранные формы, базы данных и стандартные действия CRUD (create, read, update, delete), а мы в дополнение к этому генерировали и бизнес-логику на основе SDL-диаграмм. Эта технология до сих пор активно используется при производстве ПО телефонных станций, но ни о каких продажах в нашей стране пиратского ПО мы и не думали.

Рассмотрим таблицу, характеризующую основные уровни развития использования CASE-технологии в бизнесе:

К середине 1980-х годов системы и проекты разрастаются. Поначалу немногие организации могли создавать мощные системы. Затем, с распространением мэйнфреймов, количество таких организаций увеличилось, но разработка систем по-прежнему никак не координировалась. Для ранних уровней развития проектирования характерная черта – изолированность "островков автоматизации" в "море" организации (этап с 1965 по 1985 гг.). Позже, при помощи IEF, ADW, HPS (на базе IE и CASE-технологий), преобладающим становится централизованное планирование в рамках всей организации (enterprise level). В то время проекты требовали участия 500-1000 человек в течение 1-2 лет.

В середине 1990-х годов такая "гигантомания" признается экономически нецелесообразной, разработка систем переходит на уровень department level, когда проектирование и планирование осуществляются в рамках одного отдела (департамента); для этого требуется работа 2-5 человек в течение одного-двух месяцев. В этот момент были очень популярны средства быстрого прототипирования – Rapid Application Development (RAD), такие как Power Builder, FORTE, Sun Microsystems Powery.

Средства быстрого прототипирования отодвинули на второй план CASE-средства, но, в свою очередь, были задавлены агрессивной политикой Microsoft. На какое-то время самым популярным языком стал Visual Basic. В последнее время наблюдается переход к уровню SOA (Service-Oriented Architecture), основанному на ОО-моделировании.

К 2000 году от графической разработки моделей практически отказались и, как оказалось, зря.

С 1998 года стала набирать силу технология Rational Rose, основанная на объектно-ориентированном подходе и на последовательно уточняющихся графических моделях. Первоначально промышленному использованию такого подхода мешало разнообразие похожих, но различающихся моделей. Заслугой фирмы Rational можно считать то, что она сумела объединить трех классиков объектного проектирования ("three amigos" Рэмбо, Буча и Якобсона), которые создали универсальный язык ОО-моделирования UML — Universal Modeling Language [22].

Технология стала настолько успешной и популярной, что IBM купила фирму Rational Software более чем за 2 млрд. долларов и включила Rational Rose в свою линейку продуктов.

Аналогичный путь проделала технология компании Together Soft, правда, значительно более скромная по своим возможностям. Их купила Borland за 56 млн. долларов.

Что же случилось с первыми CASE-технологиями?

В 1996 году известный собиратель устаревших средств – компания Sterling Software скупила почти все CASE-средства, кроме HPS фирмы SEER Technologies. Идея бизнеса фирмы Sterling Software понятна – есть сотни компаний, использующих CASE-средства, купленные в начале 1990-х годов. В США принято, что ежегодное сопровождение ПО стоит до 20% его продажной цены. Таким образом, "могильщик" Sterling Software собирал огромные деньги только на сопровождении, не ведя никаких новых разработок. Один раз они захотели расширить свой рынок и заказали нам конвертор Cobol $\longrightarrow$ Coolgen (новое название, которое они дали IEF). Мы-то, конечно, все сделали как надо, но, на наш взгляд, компания допустила стратегическую ошибку: лучше было бы заказать нам конвертор Coolgen $\longrightarrow$ Cobol, т.к. большинство пользователей старых средств хотят освободиться от зависимости от них.

В конце концов, компания Computer Associates купила Sterling Software, и они заказали нам конвертор ADW $\longrightarrow$ Cobol, а потом и еще более пяти проектов по реинжинирингу ADW-проектов в Cobol.

Параллельно с этим два конкурента — SEER Technologies и Relativity Technologies — заказали нам конверторы HPS $\longrightarrow$ Cobol. С их стороны

был определенный риск, что жизненно важная для бизнеса информация утечет через нас к конкурентам, но мы их, разумеется, не подвели, и сейчас на рынке активно продаются оба варианта.

Таким образом, мы познакомились и так или иначе приняли участие в создании всех массовых CASE-средств. Этот опыт оказался бесценным для наших собственных работ.

Технология программирования встроенных систем реального времени

Понятие встроенной системы

Встроенная система – это программно-аппаратная система, в которой один или несколько компьютеров управляют аппаратурой в реальном масштабе времени. Масштаб времени зависит от типа системы, но обычно это микросекунды или миллисекунды. Главная особенность встроенных систем состоит в абсолютной обязательности, необходимости уложиться в заданный интервал – если система не уложилась, информация просто безвозвратно пропадет. Типичными примерами встроенных систем являются телефонные станции, системы вооружения, роботы, медицинское оборудование и т.д.

Многие авторы относят создание встроенных систем к числу самых трудных задач; в каком-то смысле, здесь сконцентрированы практически все проблемы системного программирования.

Надежность. Все программы должны быть надежными, но во встроенных системах ошибка может привести к катастрофическим последствиям. Отладка ПО реального времени осложняется тем, что эксперименты трудны, дороги или вообще невозможны, например, ядерные испытания запрещены, но ракеты с ядерными головками производятся. Даже в тех случаях, когда отладка ПО на конкретном оборудовании возможна, часто ошибочную ситуацию трудно воспроизвести, поскольку каждый запуск оборудования может сопровождаться разными временными задержками, а от них зависит последовательность запуска программных процессов. ПО телефонной станции может успешно работать годами, потом, при определенном сочетании внешних и внутренних параметров, сломаться, а после рестарта — снова долго работать без замечаний. Чтобы поймать такую редкую ошибку, нужно проявить чудеса изобретательности.

Реактивность. По определению, встроенная система реального времени должна всегда укладываться в некоторые временные рамки. Отсюда вытекают требования к тщательному анализу времени протекания различных процессов, выделению критически важных задач (здесь, как

всегда, действует правило 20-80 – 20% задач занимают 80% времени исполнения), глубинной оптимизации программ.

Компактность. Оптимизировать нужно не только время исполнения, но и объем используемой памяти. Например, в современных автомобилях карбюратором управляет микроЭВМ, в которой никакой внешней памяти нет. Если программа и данные превысят наличную память микроЭВМ на 1 байт, то нужно будет переделывать всю систему. В обычных программных системах скорость счета обычно "разменивают" на больший объем памяти, здесь же такие приемы не проходят.

Работа с базами данных. Например, все телефонные станции одного типа имеют одну и ту же программу, но каждый экземпляр станции имеет свой набор абонентов, количество соединительных линий к другим телефонным станциям, одним абонентам можно звонить по межгороду, а другим – нет и т.д. Эта информация собирается в базе данных, часто распределенной.

Взаимодействие с человеком. Некоторые встроенные системы не имеют никаких способов взаимодействия с человеком (например,система техобслуживания). Такие системы принято называть "глубоко встроенными системами" (deeply embedded systems). Другие же, наоборот, иногда требуют вмешательства человека-оператора. Это еще более усложняет ситуацию, поскольку для человека важно иметь удобный интерфейс и оперативное отображение текущего состояния всех технических элементов.

Надеюсь, даже этого короткого перечня проблем достаточно, чтобы понять, почему в нашем коллективе разработчики систем реального времени всегда пользовались наибольшим уважением.

Инструментальная и целевая ЭВМ

Для обычных ЭВМ технология разработки ПО в последние годы была более или менее стандартизирована. Такие ее элементы как алгоритмические языки высокого уровня (АЯВУ) и компиляторы для них, текстовые и графические редакторы, различные средства поддержки коллективной разработки и т.п. используются повсеместно. Вряд ли кто-нибудь сегодня решится программировать в кодах ЭВМ без

каких-либо инструментальных средств.

Теперь представим себе, что нам нужно запрограммировать контроллер карбюратора автомобиля, микропроцессор в мобильном телефоне или какое-то другое специализированное устройство. Наверняка там не будет традиционной ОС, многопользовательского режима, трансляторов и многого другого, к чему мы привыкли. Более того, из-за ограничений памяти, отсутствия обычных устройств ввода-вывода и коммерческих соображений (разовая, не массовая разработка!) все это и невозможно реализовать.

Будем действовать следующим образом.

Всю разработку осуществляем на АЯВУ на хорошо оснащенной ЭВМ, которую будем называть "инструментальной". Для получения кода на нужной "целевой" ЭВМ создаем транслятор с нашего АЯВУ, который работает на инструментальной ЭВМ, но в качестве результата выдает код целевой ЭВМ. Такой транслятор называется "кросс-транслятором".

На самом деле это предельно упрощенная схема. Чтобы как можно дольше оставаться на инструментальной ЭВМ, кроме кросс-транслятора, нужно разработать различные имитационные модели, средства снятия и сравнения трасс, средства замера временных интервалов и т.д.

Самую сложную, на моей памяти, схему отладки мы реализовали для военных заказчиков, которым нужно было разработать сложное ПО на Алголе 68 для управляющего вычислительного комплекса (УВК) "Самсон". В качестве инструментальной ЭВМ мы использовали ПЭВМ.

Сначала все программы транслировались в коды ПЭВМ и проверялись на имитационных моделях. Так была проверена правильность основных алгоритмов. Естественно, мы не могли проверить реактивность и другие временные параметры.

Затем все программы были оттранслированы кросс-транслятором в коды "Самсона" и были еще раз проверены на ПЭВМ, но с использованием интерпретатора системы команд УВК "Самсон". Так мы получили размеры реальных объектных кодов, проверили правильность адресации и смогли подсчитать время исполнения всех

фрагментов программы, не зависящих от внешних сигналов.

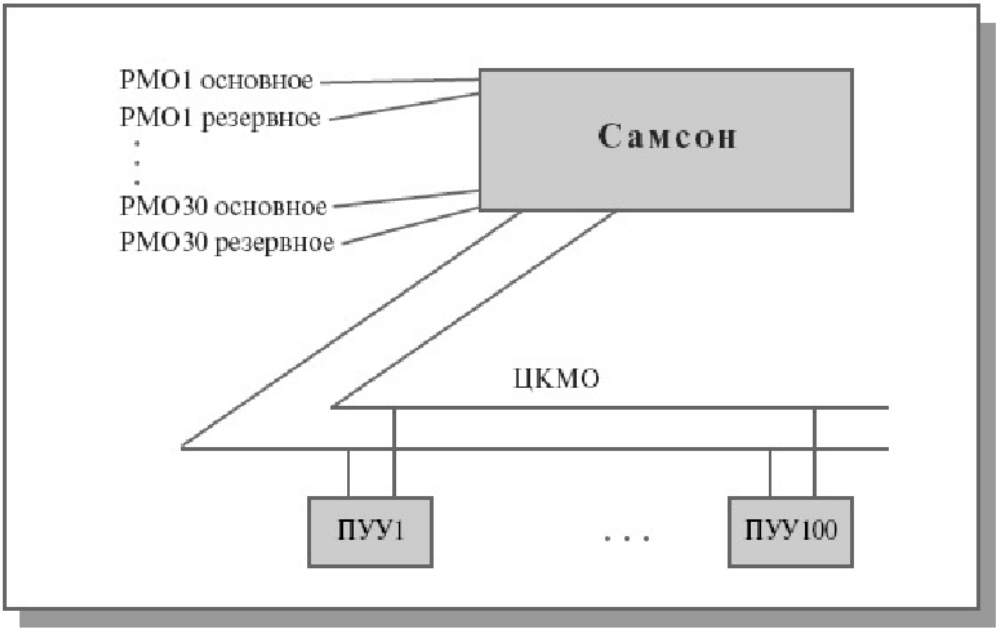
Наконец, на стенде, в котором несколько ПЭВМ и один "Самсон" были связаны в локальную сеть, мы получили возможность запускать программы непосредственно на "Самсоне", но с полным контролем со стороны ПЭВМ (остановка по команде, по адресу чтения и т.п.) "Самсон" в то время был всего один (это обычная ситуация, когда ПО и оборудование разрабатываются одновременно), а разработчиков – много; при этом каждый из них мог запускать программы со своего рабочего места.

Такая "трехступенчатая" схема отладки помогла разработать чрезвычайно важное ПО.

Комплекс вычислительных средств

Крупные встроенные системы реального времени обычно управляются не одним, а несколькими компьютерами, связанными сетью передачи данных. Если все компьютеры и их операционные системы одинаковы, сеть называется гомогенной, в ином случае – гетерогенной. Управлять гетерогенной сетью намного сложнее, хотя в наш век стандартных протоколов связи эти трудности успешно преодолеваются. Понятие "комплекс вычислительных средств" и связанные с этим понятием проблемы мы продемонстрируем на нескольких примерах.

Более 6 лет мы разрабатывали большую междугородную телефонную станцию. Эта АМТС предназначалась для правительственной связи, поэтому главным требованием была надежность. Именно из-за соображений надежности для управления АМТС был разработан довольно сложный комплекс вычислительных средств (КВС).



Во главе КВС стоит троированный "Самсон", его надежность во много раз превосходит надежность любого другого элемента КВС. Именно "Самсон" принимает окончательное решение во всех спорных ситуациях. Через центральный канал межмашинного обмена (ЦКМО) "Самсон" связан с периферийными управляющими устройствами (ПУУ), которых может быть до 100 штук, и ЦКМО, и ПУУ дублированы. Каждая половинка ПУУ связана с одним подканалом ЦКМО, но может быть переключена и на другой подканал. Каждое ПУУ представляет собой небольшую ЭВМ, управляющую телефонным оборудованием (абонентскими комплектами, коммутационным полем, концентраторами и т.д.)

Две половинки ПУУ работают синхронно, выполняя одну и ту же программу. Существует специальная схема, проверяющая совпадение результатов половинок, выдаваемых в ЦКМО. Если "Самсон" получит сигнал о несовпадении результатов (то, в первую очередь, проверяется корректность ЦКМО) – просто посылается контрольный запрос на соседние ПУУ: если ответ поступит, то сбилось ПУУ, если нет, то что-то случилось с ЦКМО. Если вышел из строя один из подканалов ЦКМО, он выводится из конфигурации, второй подканал переводится в другой режим работы, например, каждое сообщение повторяется или кодируется специальным кодом с избыточностью, который позволяет

обнаруживать и даже исправлять ошибки передачи.

Если же с ЦКМО все в порядке то, прежде всего надо выяснить, какая из половинок ПУУ сломалась. "Самсон" запускает короткий диагностический тест в обеих половинках, затем выводит неисправную часть из конфигурации и сообщает оператору о необходимости ремонта. Ввод в конфигурацию половинки ПУУ после ремонта является весьма нетривиальной задачей. Остановить работающую половинку даже на короткое время для выравнивания данных с вводимой половинкой нельзя из-за требований надежности и готовности АМТС в целом. Нам удалось придумать хитрый алгоритм постепенного выравнивания данных, но поскольку несколько поколений студентов заваливало экзамен на этом вопросе, я отказался от мысли изложить содержание алгоритма в лекциях, хотя готов рассказать его интересующимся в индивидуальной беседе.

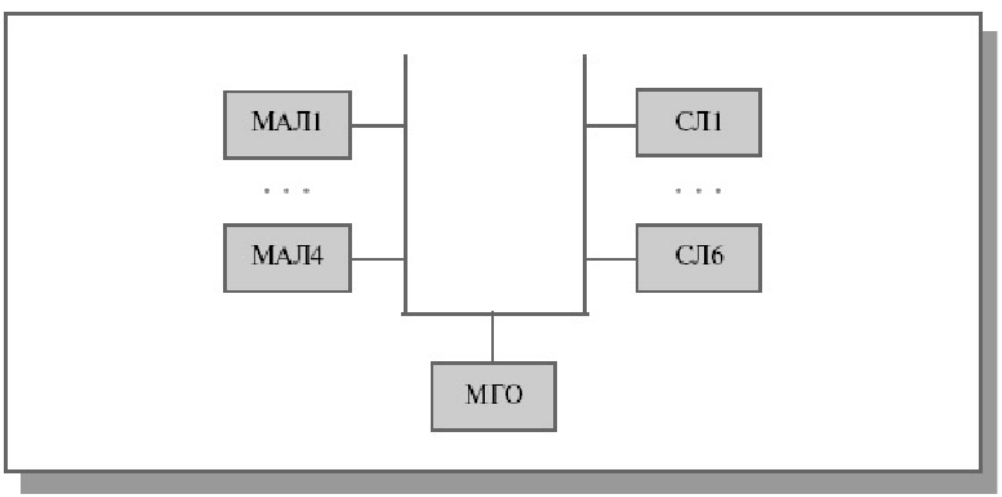
АМТС управляется операторами, сидящими за рабочими местами операторов (РМО), которых может быть до 30 штук. Каждое РМО — это две обычных персональных ЭВМ (основная и резервная). Никаких аппаратных средств резервирования ПЭВМ не имела, поэтому пришлось изобретать программную реализацию. И основная, и резервная ПЭВМ связана с "Самсоном" отдельным, сравнительно медленным последовательным каналом. В "Самсоне" для каждого РМО предусмотрен специальный драйвер, который "знает", какая ПЭВМ в данный момент основная, а какая – резервная. Любая информация для РМО из "Самсона" посылается в обе ПЭВМ, все действия оператора разбиты на короткие транзакции по базе данных конкретного РМО. Если основная ПЭВМ отказала, что видно по тестам, таймерным событиям или просто визуально, оператору достаточно пересесть на резервную ПЭВМ. При этом гарантируется, что все транзакции, завершенные на основной ПЭВМ, закончены и на резервной, поэтому оператору не придется заново вводить более одной команды.

В 1996 году мы успешно прошли государственные испытания этой АМТС, накопили бесценный опыт разработки и комплексной отладки сложных программно-аппаратных средств, отработали технологию программирования RTST на базе SDL, в общем, все было хорошо. Мы были уверены в скором и успешном продолжении работ, но в этот момент государственное финансирование окончательно прекратилось, и

наш отдел разработки средств телекоммуникаций остался без работы.

Поэтому как нельзя кстати оказалось предложение ЛНПО "Красная Заря" переделать или написать заново программное обеспечение для уже существующей сельской АТС "Бета". Станция была разработана в ЛНПО "Красная Заря", сертифицирована и уже серийно выпускалась. Но оказалось, что ПО, сделанное "на коленках", без всякой технологии, очень дорого в сопровождении – трудно исправить ошибку, сложно добавить новую функцию. В результате каждая АТС после полугода эксплуатации обладала индивидуальным ПО, а если после этого исправлялась какая-то незамеченная ранее ошибка или добавлялась новая функция, то приходилось переделывать все версии ПО.

В другой ситуации мы, возможно, и отказались бы от такого предложения, но поскольку другой работы не было, делать было нечего. Только много позже мы поняли, как нам повезло. Мы изучили чужой опыт, познакомились с требованиями рынка, получили подробно изложенные алгоритмы большинства протоколов, используемых на российских каналах связи.



Для начала нас познакомили с общей архитектурой АТС "Бета".

МАЛ — модуль абонентских линий, к которому подключаются до 180 телефонов.

СЛ — соединительная линия к другой АТС.

МГО – модуль группового оборудования.

Каждый из этих модулей управлялся контроллером i8086 и имел

небольшую оперативную память. Все модули связывались между собой общей шиной с пропускной способностью 9,6 кбит/сек.

Нам объяснили, что общая шина является самым узким местом станции, многие протоколы не укладываются в ограничения по времени, поэтому сейчас они думают перейти на скорость 19,2 кбит/сек.

Наш специалист Н.Ф. Фоминых тут же подсчитал, что тогда каждый контроллер будет тратить больше половины своего времени только на обработку прерываний (известно, что при тактовой частоте 4,67 Мгц этот процессор имеет чуть больше 1 млн. операций в секунду).

В результате часа разговоров с разработчиками АТС мы нашли

несколько способов уменьшить нагрузку на общую шину:

1. Для защиты от ошибок при передаче по шине использовался сложный протокол, который обычно применяется при передаче на большие расстояния (сотни километров). А тут внутри одной стойки к каждому пакету добавляется несколько байтов!
2. На каждый полученный пакет каждый модуль должен был послать двухбайтовую квитанцию, подтверждающую правильность приема. Но оказалось, что порядок посылки пакетов жестко определен – всегда МГО сначала посылает какому-то модулю запрос, затем этот модуль должен послать ответ в МГО, причем каждый пакет имеет контрольную сумму своих битов. Н.Ф. Фоминых предложил в каждом пакете иметь всего 1 бит, подтверждающий правильность передачи предыдущего пакета.

Уже первые два предложения снизили нагрузку на шину более чем в 2 раза. Но мы пошли еще дальше. Разработчики АТС "Бета" предполагали, что МАЛ (модуль абонентских линий) будет очень простым, например, он будет принимать номер вызываемого абонента по одной цифре и пересылать ее в МГО.

1. Мы предложили принимать в МАЛ весь номер, а только потом сразу в одном пакете передавать его в МГО. Чтобы понять, сколько цифр надо принять – одну (8 для входа в межгород), две (01,02,03), три (для внутриофисных вызовов), пять (сельская связь) и т.д., в каждый МАЛ пришлось поместить по небольшой таблице, настоящая таблица маршрутизации все равно осталась в МГО. Таким образом, нагрузка на сеть упала еще в 5-7 раз.

Самое забавное и поучительное событие произошло через несколько месяцев после того как мы начали разрабатывать новое ПО для АТС "Бета". Наша операционная система теряла отдельные сообщения, хотя по всем расчетам производительности процессора должно было хватать. После долгих мучительных поисков ошибки в ПО Н.Ф. Фоминых решил проверить правильность работы аппаратуры и обнаружил ошибку в разводке платы процессора. Вместо тактовой частоты 4,67 МГц на вход clock процессора подавалась частота 1 МГц, предназначавшаяся совсем для других целей. Таким образом, все 30 АТС, выпущенные до нас, управлялись в 5 раз более медленными ЭВМ, чем могли бы. Потому и все проблемы были связаны с шиной, т.к. любая работа с ней связана с обработкой прерываний, где производительность процессора критична. Боюсь, если бы разработчики АТС "Бета" обнаружили эту ошибку сами, к нам бы вообще не обратились и тогда бы история нашего коллектива сложилась совсем иначе.

Параллельные процессы

Вообще-то, про параллельные процессы принято рассказывать в других курсах, чаще всего в курсе "Операционные системы" [23]. Но поскольку мы хотим изложить придуманную в нашем коллективе эффективную организацию вычислительного процесса во встроенных системах реального времени, нужно, чтобы читатель понимал "что почем" в мире параллельных процессов.

Можно предложить следующую иерархию разбиения сложной системы на взаимодействующие куски:

1. Самым простым является вызов подпрограммы (`subroutine`).

Часто встречающийся кусок кода оформляется как подпрограмма. В том месте программы, где необходимо использовать эту подпрограмму, ставится команда "переход с возвратом" (`branch and link`), которая запоминает адрес следующей команды в регистре, а затем передает управление в начало подпрограммы. В конце подпрограммы ставится команда перехода по адресу, записанному в регистре. Разумеется, должны соблюдаться определенные соглашения о связях, например, возврат из подпрограммы должен всегда осуществляться по одному и тому же номеру регистра. Каждая подпрограмма должна сохранить значение этого регистра в какой-то памяти, если внутри нее есть вызовы других подпрограмм. В любом случае, накладные расходы на вызов подпрограммы небольшие (2-4 команды).

2. Следующим по сложности является вызов процедуры. К действиям по вызову подпрограммы добавляется захват области памяти для локальных переменных процедуры и различных рабочих данных при входе в процедуру и освобождение памяти при выходе (возврате) из процедуры. Память используется в стековом режиме – последняя занятая память освобождается первой.

   Примерный перечень действий при вызове процедуры:

   - Захват секции памяти.
   - Установка динамической цепочки (адрес начала секции памяти вызывающей процедуры).
   - Сохранение регистров.
   - Сохранение адреса возврата.
   - Передача параметров.
   - Передача управления.

   При возврате нужно восстановить регистры, освободить память и восстановить контекст вызывающей процедуры. Для ЭВМ без аппаратной поддержки вызов процедуры может потребовать от 30 до 150 команд. Когда-то мы очень гордились, что сумели для архитектуры IBM/360 придумать реализацию вызова процедуры за 11 команд.

   Заметим, что описанная реализация вызова допускает рекурсию,

когда процедура вызывает саму себя, а основное направление оптимизации вызовов – сведение вызова процедуры к вызову подпрограммы.

3. Еще более сложным является вызов сопрограммы (`coroutine`). В дополнение к действиям "создать" и "уничтожить" сопрограмму вводится оператор `transfer(p)` – прервать исполнение текущей сопрограммы и передать управление в сопрограмму `p` (точнее, возобновить ее исполнение). При приостановке сопрограммы ее память не освобождается, поэтому когда-то управление может в нее вернуться (опять-таки оператором `transfer`). Такая структура управления полностью детерминирована и очень удобна при моделировании различных процессов. Наиболее известным языком, использующим сопрограммы, является Модула 2 [24]. По времени исполнения вызов сопрограммы мало чем отличается от вызова процедуры, но по объему используемой памяти, разумеется, он обходится дороже.
4. Самым общим, мощным и дорогим средством организации взаимодействия различных фрагментов большой программы являются параллельные процессы. Процесс – это практически независимый объект, он имеет свой исполняемый код, свою область памяти, его состояние определяется содержимым этой памяти и адресом исполняемой в данный момент команды. Если программный код устроен таким образом, что в него никто не пишет, одна копия кода может разделяться многими параллельно исполняющимися процессами (`reenterable`), но рабочая память у каждого процесса должна быть своя. Процессы могут исполняться параллельно, например, на разных процессорах. Во многих операционных системах задача – это процесс, к которому приписаны различные ресурсы (максимально доступные объемы памяти и процессорного времени, объемы дискового пространства и т.д.). Если приложение состоит из нескольких параллельных процессов, они должны иногда взаимодействовать. Это очень непростая задача. Рассмотрим пример.

Пусть два процесса `p1` и `p2` используют общую переменную `S` вида `struct (int a,b)`, т.е. `S` является структурой с двумя целыми

полями. Предположим, процесс p1 решил прочитать значение структуры S, успел прочитать первое поле a, но был по какой-то причине прерван. Процесс p2 записал в структуру S новое значение, потом процесс p1 был возобновлен и продолжил чтение со второго поля b. В результате p1 получит значения полей a и b из разных фаз обработки.

Таким образом, мы приходим к идее монополизации ресурсов или, другими словами, синхронизации действий параллельных процессов. Было придумано несколько чисто программных решений, но все они обладают одним общим недостатком – какое–то время ожидающий процесс "висит" на процессоре в пустом цикле, бесполезно растрачивая самый драгоценный ресурс – время процессора.

Еще в конце 50-х годов XX века известный ученый Э. Дейкстра придумал принципиально новую конструкцию – семафор [5]. Семафор – это структура с одним целым полем, но имя поля неизвестно, поэтому обычными способами ни прочитать, ни изменить это поле нельзя. Семафор допускает только три операции – установить значение семафора (level), увеличить (up) значение на 1 и уменьшить (down) на 1. Если во время исполнения операции down значение семафора стало отрицательным, текущий процесс приостанавливается (suspend), если при выполнении операции up значение семафора из отрицательного становится нулем, один из процессов, приостановленных из-за этого семафора, возобновляется (resume). Считается, что в операционной системе есть очередь готовых к исполнению процессов и очередь процессов, ожидающих неотрицательного значения какого-то семафора, но никакого определенного порядка исполнения не подразумевается. Принципиально новой идеей в семафорах является их неделимость. В операции down между вычитанием единицы и проверкой на неотрицательность никакое прерывание процесса невозможно. В наше время любая память ЭВМ обеспечивает, кроме обычных операций чтения и записи, специальную операцию "семафорное чтение", в которой в одном такте выдается значение байта, а в байт пишется 1. Нетрудно сообразить, как, используя эту операцию, корректно реализовать операции up и down.

Рассмотрим пример:

```
sema s = level 1;
par  begin
    do НАЧАЛО1;
      down s; ДЕЙСТВИЯ1; up s;
      КОНЕЦ1
    od ,
    do НАЧАЛО2;
      down s; ДЕЙСТВИЯ2; up s;
    КОНЕЦ2
    od
  end
```

Здесь два параллельных процесса представлены бесконечными циклами. Каждый процесс включает в себя последовательность операторов, ограниченную операторами `down` и `up`. Начальное значение семафора равно 1.

Действия `НАЧАЛО1` и `НАЧАЛО2` выполняются параллельно, допустим, второй процесс раньше достигает своего оператора `down s`. Тогда `s` получит значение 0, начнется выполнение последовательности `ДЕЙСТВИЯ2`. Пусть теперь и первый процесс достигнет оператора `down s`. Так как текущее значение `s` равно 0, первый процесс приостановится и будет ждать, пока второй процесс не выполнит оператор `up s`. Кстати, никто не гарантирует, что после этого возобновится первый процесс – если второй процесс выполняется быстрее первого, то вполне возможно, что раньше снова сработает оператор `down s` второго процесса.

Я хотел показать на этом примере, что `ДЕЙСТВИЯ1` никогда не будут выполняться параллельно последовательности `ДЕЙСТВИЯ2` – или `ДЕЙСТВИЯ1`, или `ДЕЙСТВИЯ2`, но не вместе. Такие фрагменты параллельных процессов называются критическими интервалами.

На практике столь простая и фундаментальная конструкция, как семафор, оказалась очень опасной. Два семантически связанных действия разнесены по тексту. Кто-то забудет открыть семафор, а кто-то – закрыть.

Позже Т. Хоар предложил более безопасную конструкцию для

синхронизации параллельных процессоров – монитор[26]. Представьте себе дверь в комнату, в замок которой вставлен ключ. Вы открываете дверь, вынимаете ключ, входите в комнату и закрываетесь в ней. Следующий, кто захочет войти в комнату, не найдет ключа и вынужден будет ждать, пока вы не выйдете. В языки программирования ввели оператор `seize m` (схватить ресурс `m`). Если ресурс занят, процесс зависает прямо на этом операторе `seize` – никакого "парного" оператора не надо. Вроде бы пустяк, но оказалось, что этот оператор значительно технологичнее.

Значительно позже появились почтовые ящики, "рандеву" и многие другие средства. Математически они все эквивалентны, т.е. выражаются друг через друга, но с точки зрения технологичности, выразительности, эффективности реализации — существенно различаются.

В нашем коллективе принято пользоваться механизмом сообщений. Параллельные процессы могут использовать операторы `send m` (послать сообщение) и `receive m` (получить сообщение). После оператора send процесс ничего не ждет, а работает дальше, сообщение "заплетается" в очередь входных сообщений того процесса, которому оно предназначено (обычно адресат указывается либо отдельным параметром, либо просто включается в сообщение).

По оператору `receive` процесс просматривает свою очередь входных сообщений. Если она пуста, то процесс "зависает", иначе берется очередное сообщение и обрабатывается.

Нам нравится, что сообщения не только играют роль синхронизации, но и несут семантическую нагрузку – обмен данными.

Иногда требуется более жесткая схема синхронизации, тогда вместо `send` используется оператор `ask m`, после которого процесс приостанавливается и ждет ответа.

Работа с временными интервалами и организация вычислительного процесса. Технологии RTST и REAL

Работа с временными интервалами

В системах реального времени часто возникает ситуация, когда текущий процесс должен быть задержан на определенный период времени. Чаще всего такая ситуация возникает, когда процесс должен дождаться окончания работы какого-то внешнего устройства. Задержку на определенное время осуществляет оператор `delay m`, где `m` – количество каких-то единиц времени, например, миллисекунд.

Мы уже много лет применяем следующую реализацию оператора `delay`. Процесс, в котором исполнялся этот оператор, приостанавливается и ставится в очередь задержанных процессов. Эта очередь упорядочена по времени ожидания. Например, пусть `p1` и `p2` ждут 10 мсек, `p3`, `p4`, `p5` – 30 мсек, `p6`, `p7` – 100 мсек. Тогда в очереди задержанных процессов первым узлом будет узел с временем ожидания 10, к которому "заплетены" `p1` и `p2`, за ним следует узел с ожиданием 30, к которому "заплетены `p3`, `p4` и `p5`, а последним узлом в очереди задержанных процессов будет узел с временем ожидания 100 с "заплетенными" к нему `p6` и `p7`.

В операционной системе есть системный таймер, который "тикает", например, через каждую миллисекунду, т.е. каждую миллисекунду возникает прерывание, обработчик которого пробегает по очереди задержанных процессов и вычитает единицу из времени ожидания каждого узла. Если время ожидания узла превращается в ноль, "заплетенные" к нему процессы переводятся в очередь готовых к исполнению процессов.

Очевидной оптимизацией является хранение в узлах не абсолютных значений времени ожидания, а нарастающих итогов, например, в нашем примере, не 10, 30, 100, а 10, 20, 70. Тогда каждую миллисекунду надо вычитать 1 только из первого узла. В УВК "Самсон" мы ввели так называемое "мягкое" прерывание: каждую миллисекунду мы вычитали 1 из верхушки списка задержанных процессов на микропрограммном уровне и только при обращении верхушки в 0 реально вызывалась

функция ОС.

Все это кажется вполне очевидным, и я не стал бы включать описание этой реализации в курс лекций, если бы не две забавные истории.

В начале восьмидесятых годов XX-го века мы неожиданно обрели конкурента в лице одного пензенского завода. Их бывший директор стал большим начальником в области правительственной связи и, как это принято в нашей стране, стал "сливать" им все заказы с большими деньгами, хотя раньше они никогда телефонных станций не делали. Им за малые АТС давали большие деньги, чем нам за большие АТС, им разрешали использовать импортные микросхемы, нам – нет и т.д. Так вот, они даже с лучшими американскими микропроцессорами того времени хронически не укладывались во временные ограничения. Разобраться, в чем дело, причем втайне от большого начальника, поручили нам. Среди многих неграмотных решений самым несуразным оказалась реализация оператора `delay`. Каждую миллисекунду они приостанавливали текущий процесс, заходили во все процессы (т.е. возобновляли их), проверяли, нет ли там оператора `delay`, если есть – вычитали 1 и снова сворачивали процесс. Процессов было больше 1000, свертка-развертка у них занимала 500-600 команд — понятно, куда уходило время. В результате "разбора полетов" четырех ведущих разработчиков из Пензы прислали к нам на стажировку на месяц, но мат-мех за месяц не закончишь. Хотя свою АТС они все-таки сдали – с пятилетним опозданием.

Другой пример связан с французской АТС МТ/20. В 1979 году СССР официально купил лицензию на право производства этой АТС (что было редкостью в то время). В Уфе построили большой завод, и после многих трудностей, связанных, в основном, с попытками "улучшить" и "советизировать" готовую АТС, наладили массовое производство. Даже к началу XXI века это была самая массовая цифровая АТС в нашей стране (более 3 млн. абонентов). МТ/20 проживет еще 15-20 лет, однако ее центральная ЭВМ 3202 (два больших шкафа) первой не выдержала испытания временем. Уже не выпускаются микросхемы, используемые в этой ЭВМ, давно забыты технологические средства, применявшиеся при ее разработке, наконец, не осталось и инженеров, умеющих ее налаживать. Понятно, что основной ценностью МТ/20 является ее программное обеспечение. Заводчане постепенно модернизировали

основные аппаратные блоки МТ/20, прежде всего, блоки памяти, вызывавшие много нареканий, уменьшили массогабариты и энергопотребление, но не смогли существенно модернизировать ПО.

По заказу завода мы разработали новую ЭВМ, идентичную по архитектуре и системе команд старой 3202. Главной задачей было обеспечение работы оригинального ПО. Это было непросто, точных описаний ЭВМ не сохранилось, многие детали, особенно на стыках с телефонной аппаратурой, пришлось восстанавливать экспериментально. Когда новая ЭВМ, наконец, заработала на французских тестах, причем в 10 раз быстрее старой, оригинальное ПО не заработало.

Оказалось, что в оригинальном ПО оператор `delay` был реализован "жужжанием", т.е. задержка на 10 мсек реализовывалась повторением 100 раз команды со временем исполнения 100 микросекунд. А у нас эта же команда исполнялась всего 10 микросекунд! Пришлось выискивать все такие места и увеличивать число повторений в 10 раз.

Это к вопросу о том, почему я так не люблю дилетантов и троечников.

Организация вычислительного процесса

Обычно считается, что используемая технология программирования должна обеспечить реализацию любой структуры, выдуманной проектировщиком. Однако все известные нам попытки применения общих подходов к такой сложной задаче как ПО встроенных систем реального времени, приводили к неэффективным решениям. По нашему мнению, технология программирования должна отвечать не только на традиционный вопрос "как построить, добиться, оценить", но и "что мы хотим построить". Мы к этому выводу пришли постепенно, по мере накопления опыта выполнения промышленных заказов. Чтобы обеспечить высокое качество, мы разработали технологию, первоначально полностью ориентированную только на программное обеспечение телефонных станций, которая получила название RTST — Real-Time Software Technology [28].

При разработке RTST мы заранее зафиксировали определенные структурные решения и тем самым ограничили разнообразие вариантов

организации вычислительного процесса.

Процессы

Параллельные процессы — классический способ описания поведения сложных систем: независимое управление в каждой подсистеме, локализация данных, развитые средства взаимодействия позволяют описывать системы наиболее понятным для разработчиков способом. Тем не менее, в реальных системах этот способ используется редко из-за низкой эффективности реализации. Дело в том, что в них параллельно существуют тысячи процессов, абсолютное большинство которых находится в неактивном ("подвешенном") состоянии, ожидая каких-либо событий. Традиционным механизмом реализации параллелизма такого типа является диспетчеризация через короткие промежутки времени по сигналам от таймера (системы разделения времени), однако при этом самой частой операцией становится свертка/развертка процессов, сильно увеличивающая накладные расходы системы [27].

Чтобы сделать параллельные процессы эффективными, мы воспользовались особенностью встроенных систем, состоящей в том, что обычно их процессы имеют очень короткие действия — переходы в терминах расширенной конечно-автоматной модели рекомендаций Z.100 МККТТ [29]. Мы считаем, что если процесс получил управление по какому-то входному сигналу, то он должен довести переход до конца (до следующего состояния) и только затем передать управление диспетчеру, который определит, какой из процессов, готовых к исполнению (т.е. получивших сообщения), будет работать следующим. Синхронная организация параллелизма делает возможной простую процедурную реализацию, в отличие от традиционного асинхронного подхода, при реализации которого необходимы процессы. Против синхронной реализации обычно выдвигают два аргумента.

Во-первых, часть сообщений приходит по сети от других ЭВМ или оборудования в непредсказуемое время. Для решения этой проблемы соберем все функциональные процессы в один большой процесс ОС, внутри которого будем использовать синхронную организацию взаимодействия функциональных процессов, а драйверы сети, которые осуществляют буферизацию сообщений (никакой обработки!), будем реализовывать простыми обработчиками прерываний на стеке текущего

процесса. Интересно, что на одно внешнее событие обычно приходится 5-10 внутренних переключений функциональных процессов.

Во-вторых, что будет, если один из процессов все-таки имеет слишком длинный переход? Тогда можно вставить несколько промежуточных фиктивных состояний (на практике это встречается крайне редко).

Такая последовательная и непрерываемая организация переходов позволяет процессам пользоваться глобальными данными (кроме списков входных сообщений, к которым имеют доступ параллельно работающие драйверы сети) без семафоров, критических интервалов и других дорогостоящих средств монополизации ресурсов.

Данные

Традиционный подход к разработке ПО встроенных систем подразумевает создание единой базы данных (централизованной или распределенной), к которой все процессы обращаются через специальные примитивы доступа. В общем случае обращение к некоторой структуре требует ее монополизации на момент обращения, что и неэффективно (несколько обращений к функциям ОС), и ненадежно (одна из самых частых и трудно обнаруживаемых ошибок возникает, когда программист забывает захватить ресурс или, что еще хуже, захватывает, но забывает освободить после использования). Проектирование единой базы данных обычно ведется только в интересах ПО без учета соответствия данных элементам реального оборудования. Это приводит к необходимости довольно сложной генерации данных ПО на основе исходной спецификации оборудования и сильно затрудняет задачу модификации данных в процессе функционирования системы ("на ходу").

Такие особенности встроенных систем, как структурное подобие большей части данных управляемому оборудованию и определенная локализация действий вокруг каждого устройства, прямо подталкивают к использованию объектно-ориентированной парадигмы, однако ее эффективность не очевидна.

Во-первых, не вызовет ли разбиение данных на сравнительно мелкие объекты (с точностью до прибора) значительного расходования памяти?

Действительно, заголовки процессов в большинстве ОС достигают 100-200 байтов, поэтому представление объектов в виде полноправных процессов ОС действительно очень накладно. Но, как мы уже видели, возможны упрощенные синхронные варианты реализации объектов – при этом длина заголовка уменьшается на порядок.

Во-вторых, не подменяем ли мы сложные операции доступа к БД не менее дорогостоящими операциями обмена сообщениями? Другими словами, хорошо, если 90% запросов осуществляется к локальным данным объекта и только ради 10% приходится посылать сообщения, а если наоборот? Разумеется, на этот вопрос нет однозначного ответа, но практический опыт показывает, что обычно удается распределить данные по объектам достаточно удачно, т.е. организовать своеобразный конвейер. Сначала управление получает один объект, который, пользуясь своими локальными данными, выполняет определенные действия и посылает сообщение следующему объекту с результатами своей работы в качестве параметра сообщения, т.е. сообщения играют не только информационную, но и управляющую роль.

Случаи, когда сообщение посылается только с целью получения данных, локализованных в другом объекте, нужно сводить к минимуму. Здесь также можно воспользоваться особенностями встроенных систем, в которых процессы чаще всего далеко не равноправны по приоритетам. Например, в телефонных станциях процессы установления/освобождения соединения критичны по времени, а процессы техобслуживания, которые по объему во много раз больше, — нет. Соответственно, при разбиении системы на объекты нужно учитывать только локализацию данных, необходимых для установления и разъединения соединений, а данные, которые требуются для техобслуживания, распределять, экономя только собственные силы.

В качестве примера рассмотрим упрощенную телефонную станцию.

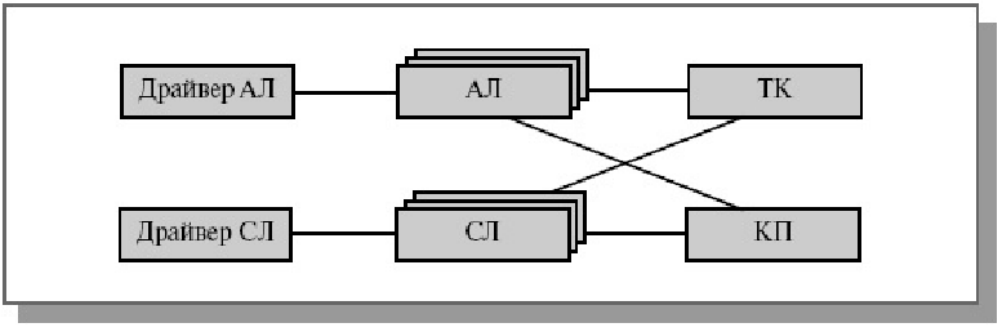


Рис. 7.1. Упрощенная телефонная станция

АЛ – Абонентская Линия;

СЛ – Соединительная Линия;

ТК – Телефонная Книга;

КП – Коммутационное Поле.

Здесь каждому абоненту соответствует объект типа АЛ, каждой линии, соединяющей данную станцию с другой, — объект типа СЛ. Объект типа ТК является справочником номеров абонентов данной телефонной станции и хранит данные, необходимые для маршрутизации. Объект типа КП соответствует реальному коммутационному полю. Драйверы играют связующую роль с внешним миром и обычно реализуются в виде ассемблерных программ, вызываемых по прерываниям.

Когда абонент поднимает трубку, соответствующий объект АЛ получает сообщение, принимает все цифры номера и посылает сообщение объекту ТК с принятым номером в качестве параметра. Заметим, что все это время АЛ работает только со своими локальными данными. ТК осуществляет поиск вызываемого абонента также только по своим локальным данным и посылает сообщение найденному объекту типа АЛ или СЛ. Вызываемый абонент или соединительная линия по своим локальным данным определяет, занят он или свободен, и если свободен, то посылает сообщение КП на подключение. КП по своим локальным данным находит свободную промлинию и т.д.

Технология RTST

Итак, создаваемую систему удалось разбить на объекты и зафиксировать организацию вычислительного процесса. На самом деле, удачное разбиение — весьма нетривиальный творческий процесс, и это заставило нас уже после нескольких выполненных проектов приступить к развитию технологии "вверх", с упором на начальные этапы разработки.

Уже с середины 1980-х годов нам было ясно, что разработка системы должна начинаться с описания на специальном языке схемы объектов (в языках программирования принято говорить "тип", а в базах данных — "схема"). Сначала объект описывается как черный ящик (точки его подключения к другим объектам, перечни входящих и исходящих сообщений и их параметры). Затем описываются внутренние атрибуты объекта. Они могут быть двух видов — те, которые не могут быть изменены самим объектом (статические), и обычные рабочие переменные (динамические). В нашем коллективе удобный для разработчиков язык описания схем объектов предложил В.В. Парфенов [30], примерно через 5 лет Object Management Group предложила для этих же целей свой язык IDL [31], на удивление похожий на наш.

Задать поведение объекта значительно сложнее, причем трудности носят скорее "человеческий", нежели технический характер, то есть проблемы возникают не из-за недостаточной квалификации кого-либо из участников разработки, а из-за взаимного недопонимания. Слишком много людей вовлечено в решение этой задачи – алгоритмисты, специалисты по протоколам, программисты, инженеры-электронщики и т.д.

Для преодоления этих трудностей Международный Консультационный Комитет по Телеграфии и Телефонии (ныне ITU-T) разработал серию графических способов описания, в частности, языки SDL (спецификация Z.100) и MSC (спецификация Z.120) [32].

SDL-диаграмма (Specification and Description Language) очень похожа на традиционные блок-схемы, но с несколькими важными отличиями: символ состояния, в котором процесс не занимает процессор, ожидая приема одного или нескольких сигналов; символ приема сигнала и символ посылки сигнала.

В примере, изображенном на рис. 7.2, объект в состоянии S0 ожидает

сообщения x или y. Получив сообщение y, он совершает переход: выполняет проверку некоторого условия C и, в зависимости от результата, выполняет определенное действие (Оператор Var:=0) или посылает сообщение z. После этого объект переходит в состояние S1. В RTST существует графический редактор для спецификации поведения объекта в таком виде, который, кроме того, осуществляет проверку синтаксической корректности спецификации и формирует ее внутреннее представление.

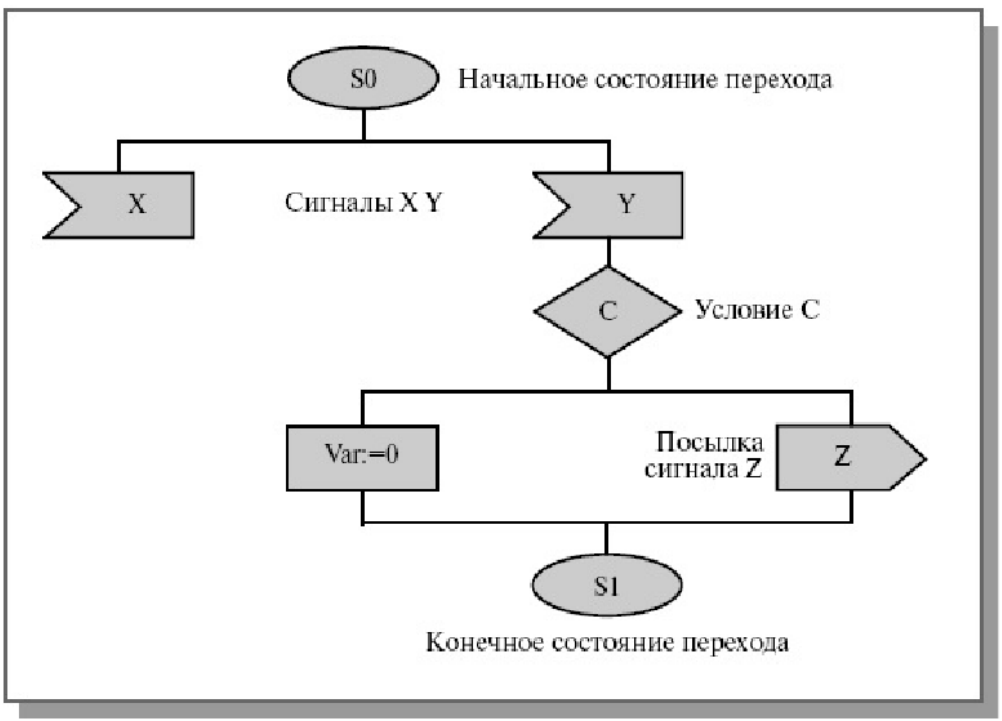
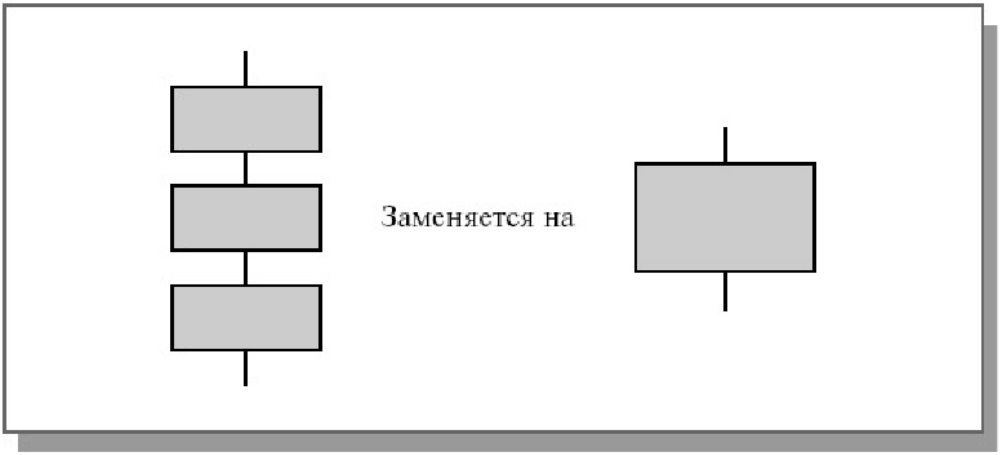


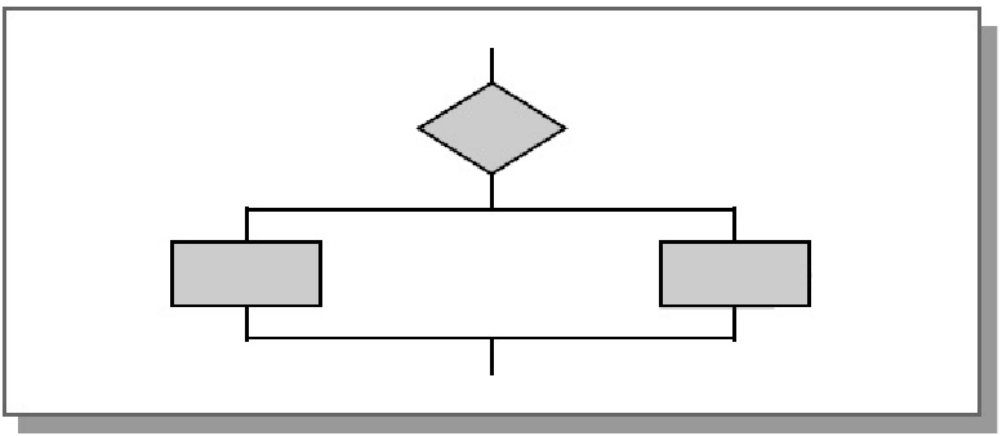
Рис. 7.2. Пример SDL-диаграммы

В дальнейшем конечно-автоматные диаграммы SDL мы будем называть просто SDL-диаграммами.

SDL-диаграммы очень удобны для наглядного описания параллельных процессов. На практике не принято графически детализировать процесс до отдельных мелких операторов, например:

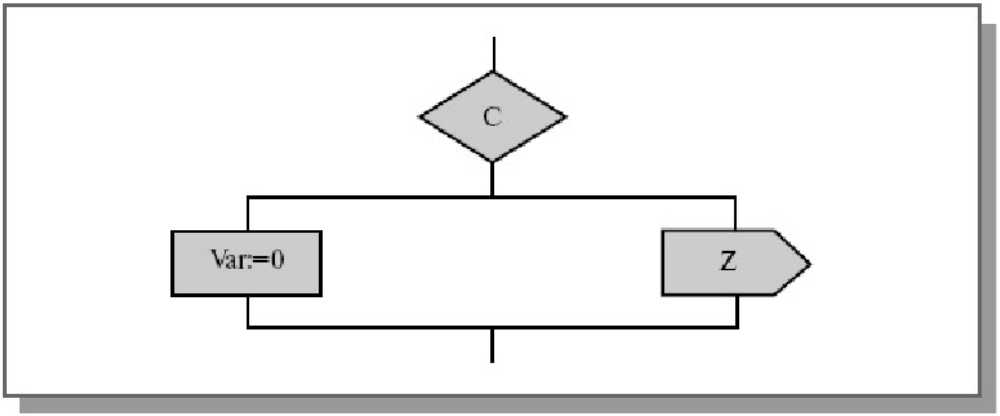


И даже



заменяется на один оператор с коротким условным оператором внутри.

Графическая детализация нужна до такой степени, чтобы была видна логика обмена сигналами, например,



не стоит сводить к одному оператору, так как при одном значении условия будет посылка сигнала, а при другом – нет.

При разработке нашего программного средства мы решили не изобретать велосипед и воспользоваться рекомендациями МККТТ. Первый редактор SDL-диаграмм был реализован на ЕС ЭВМ в 1984 году (тогда еще без всякой ориентации на объектно-ориентированное программирование). Конечно, графическим его можно было назвать лишь с некоторой натяжкой, поскольку и экраны, и устройства печати были только текстовыми. Однако если забыть про некоторую громоздкость SDL-диаграмм, они выглядели как настоящие.

Первые опыты по применению SDL-редактора на реальных телефонных станциях оказались неудачными — представьте себе две-три сотни больших печатных листов диаграмм, набранных с помощью SDL-редактора и напечатанных на АЦПУ (алфавитно-цифровое печатающее устройство), которые можно только читать и водить пальцем по связям.

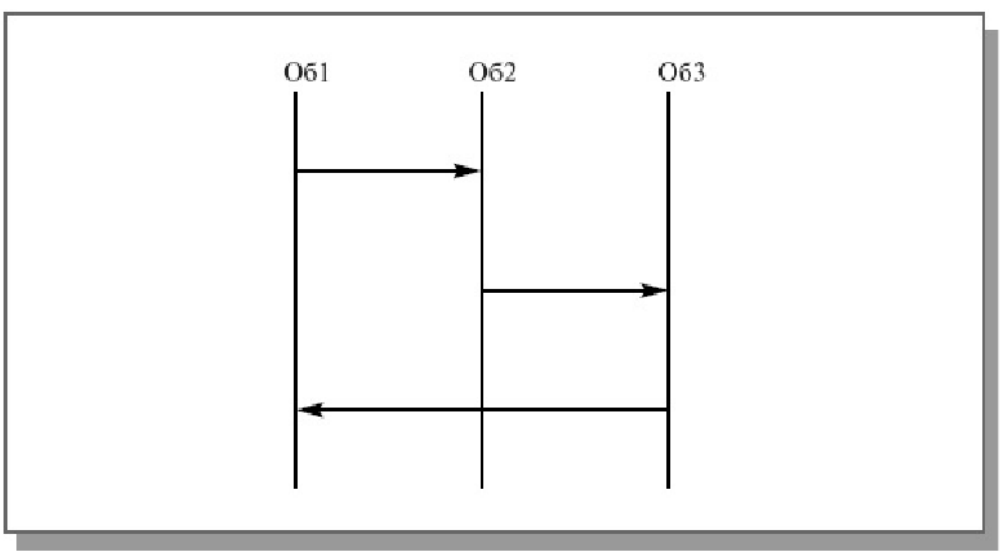
Поэтому мы разработали конвертор из SDL-диаграмм в код высокого уровня, систему имитационного моделирования, различные отладчики, средства для снятия и анализа трасс, другие инструменты, которые обеспечили первые возможности практического применения SDL-диаграмм.

В начале 90-х годов мы перевели все технологические средства на ПЭВМ, а заодно начали использовать объектно-ориентированный подход к проектированию ПО. Нам удалось найти эффективную

реализацию вычислительного процесса для систем реального времени; усилить различные статические проверки, например, если сигнал не упомянут в описании объекта, то его невозможно использовать в SDL-диаграммах; разработать средства автоматической генерации данных и так далее.

Под именем RTST (Real Time Software Technology) [33] эта технология просуществовала около 10 лет, с ее помощью было разработано более 10 типов различных телефонных станций и несколько других программно-аппаратных средств, причем 70-80% SDL-диаграмм легко переносились с одной платформы на другую. Существенно было облегчено сопровождение ПО, введение в коллектив новых специалистов, переиспользование фрагментов, так что в настоящее время мы являемся убежденными сторонниками графических средств проектирования.

MSC-диаграммы (Message Sequence Chart) позволяют описывать сценарии поведения системы во времени. Время течет сверху вниз, вертикальные линии представляют объекты системы, а между ними рисуются стрелки, обозначающие сигналы.



Если объект Об2, получив сигнал x, может не отправлять объекту Об3 сигнал y, а вместо него отправить объекту Об1 другой сигнал, например, свидетельствующий об ошибке, нужно нарисовать еще одну MSC-диаграмму для другого сценария.

MSC-диаграммы широко применяются различными международными организациями для описания протоколов, в том числе стандартов, разработанных организацией ITU-T, однако редко когда приводятся исчерпывающие описания поведения системы, включая обработку аварийных ситуаций, техобслуживания, тарификации и так далее.

На наш взгляд, это связано с недостаточностью выразительных возможностей стандарта MSC, в связи с чем получающиеся диаграммы слишком громоздки. Может быть, поэтому MSC-диаграммы раньше редко использовались для реального проектирования. В RTST MSC диаграммы практически не применялись.

С помощью транслятора схем спецификации объектов преобразуются во внутреннюю схему данных, которая является "сердцем" системы.

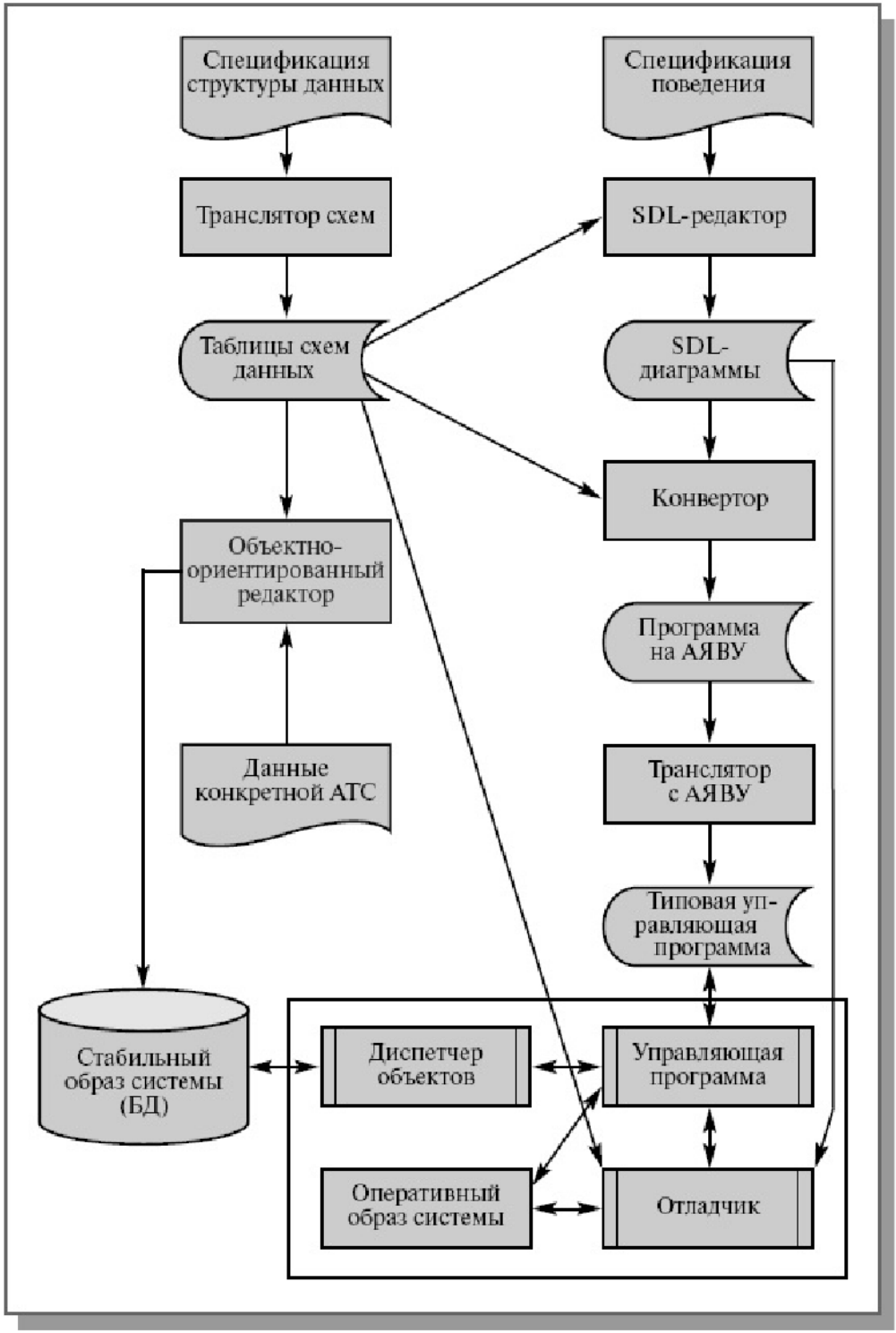
С помощью конвертора SDL-диаграммы превращаются в текст на алгоритмическом языке (ранее Алгол 68, теперь C или Java), при этом выполняются дополнительные проверки на соответствие схеме объектов, по ней же происходит генерация служебных процедур (отправка сообщения, генерация экземпляра объекта в оперативной памяти и т.д.).

Тексты, полученные в результате конвертации, транслируются в коды целевой управляющей ЭВМ (или, в целях отладки, в коды инструментальной ЭВМ) и собираются вместе со средствами динамической поддержки в загрузочный модуль.

Настройка типовой программы на конкретное приложение осуществляется посредством формирования статической базы данных, содержащей описание конфигурации аппаратных средств встроенной системы и ее внешней среды.

Управление объектно-ориентированной базой данных осуществляется системой управления базой данных (СУБД). В ведении СУБД находятся задачи создания, уничтожения, соединения и разъединения объектов, коррекции статических параметров объектов. Для повышения реактивности системы в случае рестартов СУБД реализована по методу "тщательного замещения", когда внутри транзакции ни один блок не пишется на старое место, поэтому после рестарта система мгновенно откатывается на начало транзакции.

Поддержка функционирования объектов осуществляется диспетчером, который производит последовательный запуск программ поведения готовых к исполнению объектов и управляет обменом сообщениями между связанными объектами. По запросам от СУБД диспетчер осуществляет генерацию и уничтожение объектов, запуск и остановку их программ поведения, соединение и разъединение.


Спецификация структуры данных
Спецификация поведения
Транслятор схем
SDL-редактор
Таблицы схем данных
SDL-диаграммы
Конвертор
Объектно-ориентированный редактор
Программа на АЯВУ
Данные конкретной АТС
Транслятор с АЯВУ
Типовая управляющая программа
Стабильный образ системы (БД)
Диспетчер объектов
Управляющая программа
Оперативный образ системы
Отладчик

Объектно-ориентированный редактор осуществляет управление процессом интерактивного создания и редактирования статической базы данных объектов. Он предоставляет возможность при настройке и эксплуатации системы записывать информацию в базу данных, вносить изменения и выполнять поиск. В процессе редактирования данных редактор осуществляет исчерпывающий контроль корректности и непротиворечивости данных, допуская построение базы данных лишь в точном соответствии со схемой данных, построенной по спецификации схемы объектов. Все формы ввода генерируются автоматически по базе данных схем.

В нашей технологии не различаются этапы первоначального ввода данных и их исправления в процессе функционирования системы. С помощью объектно-ориентированного редактора можно, не мешая работе системы, создать или уничтожить какие-то экземпляры объектов, изменить значения их атрибутов, изменить какие-то связи. Только если потребуется переопределить схему хотя бы одного объекта, придется перезапустить всю систему, но на практике это встречается очень редко.

Технология REAL

Многолетний опыт использования технологии RTST убедил нас в достоинствах графического проектирования ПО и сквозного контроля. Очень многие ошибки исчезли – например, нельзя послать сообщение объекту, если он этого сообщения не ожидает, в SDL- диаграммах нельзя использовать атрибут, если он не определен в описании схем объектов и т.д.

Однако постепенно пришло понимание и недостатков RTST. Прежде всего, оказалось, что трудно сразу создать описание классов, да еще в текстовом виде. Мы уже начали продумывать графические формы для описания проектируемой системы на этапах раннего моделирования, но тут появился UML. Мы были знакомы с подходами, которые до этого предлагали Буч, Рэмбо и Якобсон по отдельности, но каждый из них охватывал только отдельные фазы жизненного цикла проектирования ПО. Интеграция этих подходов в едином графическом языке UML коренным образом изменила ситуацию, особенно на начальных этапах проектирования. С другой стороны, в 1990-х годах UML не имел

практически никаких средств для определения детальных алгоритмов. Мы решили использовать наш опыт и объединить подходы UML и SDL в одной технологии [34]. Новая технология получила название REAL – с одной стороны, хотелось подчеркнуть преемственность с RTST и системами реального времени, с другой стороны, мы надеялись, что новая технология окажет реальную помощь создателям ПО.

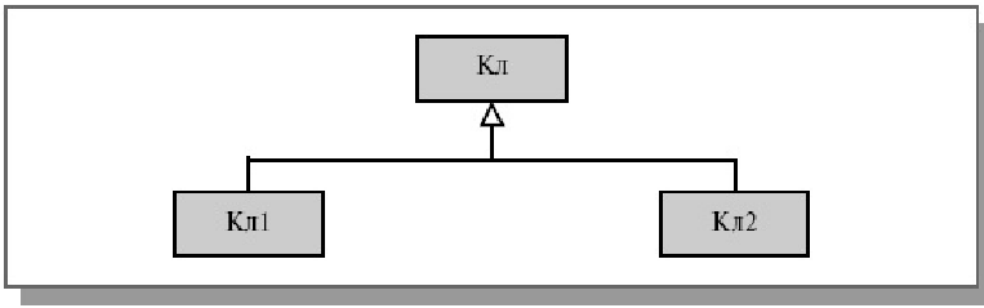
Как и прежде, мы приложили много усилий, чтобы обеспечить полноту и сквозной характер технологии. Если технология поддерживает все фазы жизненного цикла программирования, кроме какой-то одной (особенно где-то в середине жизненного цикла), то можно быть уверенным, что большинство ошибок будет именно там. Если разработчик изменил что-то на ранних фазах проектирования, а технология не отметила каким-то образом все места в более поздних фазах, затронутые этим изменением, развитие и сопровождение системы становится практически невозможным. Было очень непросто подобрать английский эквивалент понятию "сквозной характер технологии" при объяснении основных идей REAL иностранным заказчикам. В конце концов, один американец предложил перевод "drill down — просверленный". Кажется, это подходит.

Теперь приступим к изложению основных идей REAL, точнее, я поясню, как мы понимаем способы проектирования сложного ПО и то, как REAL поддерживает этапы проектирования. На наш взгляд, проектируемая система должна быть аккуратно описана с трех точек зрения.

Пользовательский уровень (иногда говорят "уровень требований"). Система описывается как "черный ящик", задаются все интерфейсы с внешним окружением, для каждого интерфейса дается его вербальное описание, т.е. описание на русском, английском или каком-то другом языке (человеческом, а не формальном). Интерфейсы задаются с помощью диаграмм случаев использования, для каждого случая использования строится диаграмма функций, которые система должна уметь исполнять.

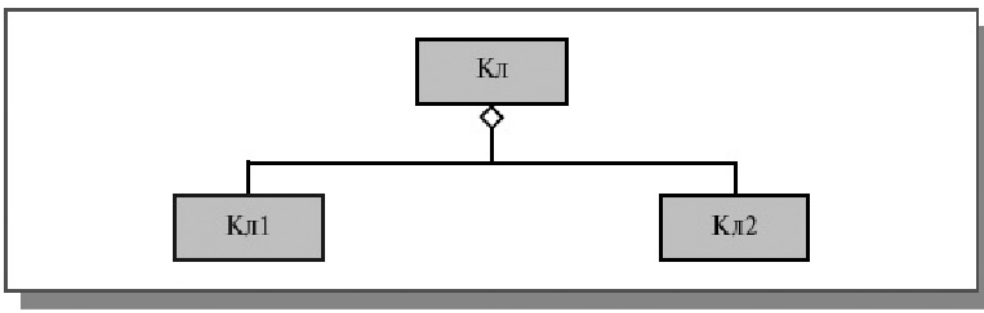
Структурный уровень. Для каждой функции рисуется диаграмма объектов, которые вместе могли бы реализовать эту функцию. Часто для разных функций удается использовать объекты одного и того же типа

или находятся объекты, которые лишь слегка отличаются друг от друга. Тогда приступают к созданию диаграммы классов. Класс — это тип (или схема в терминах баз данных), а объекты – это экземпляры значений какого-то типа (т.е. какого-то класса). Если два объекта Об1 и Об2 являются экземплярами похожих классов, то в диаграмме классов удобно использовать наследование:



Здесь класс Кл содержит атрибуты и методы, общие для двух классов, а классы Кл1 и Кл2 содержат описания, специфические для каждого класса. Объект Об1 содержит все атрибуты и методы, описанные в классах Кл1 и Кл, объект Об2 – все атрибуты и методы из Кл2 и Кл.

Если классы Кл1 и Кл2 не могут существовать без класса Кл, используется агрегирование:



Объекты агрегированных классов нельзя уничтожать по отдельности – только все вместе.

Поведенческий уровень. Здесь описывается динамическое поведение системы. Ранние версии UML слабо поддерживали этот уровень, поэтому мы решили использовать стандарты ITU-T (международного союза по телекоммуникациям). В 2004 году появились предварительные

версии UML 2.0, которые перекрывают возможности SDL, и мы уже приступили к их изучению и реализации.

Первым типом диаграмм на поведенческом уровне являются MSC-диаграммы (Message Sequence Chart – диаграмма последовательности сообщений), рекомендация ITU-T Z.120.

Исчерпывающее описание поведения системы на MSC, включающее в себя так называемые "обратные ветки" (обработка аварийных ситуаций и т.п.), оказалось слишком громоздким, поэтому мы расширили язык MSC, включив туда текстовые формы для ветвлений, процедуры-функции, возвращающие результат обработки, и некоторые другие возможности. Впрочем, при желании можно автоматически сгенерировать по расширенному MSC куда более громоздкий, зато стандартный вариант.

Если проектировщик не поленился и представил в MSC-диаграммах действительно полное описание поведения системы, появляется возможность впервые определить поведение каждого объекта по отдельности (до сих пор мы говорили о системе в целом). Перебираются все вертикальные линии (объекты) на MSC-диаграмме, каждая линия рассматривается во всех сценариях поведения. Работа объекта представляется в виде последовательных устойчивых состояний, в каждом состоянии объект ждет определенных сигналов. Получив один из возможных сигналов (сообщение), объект переходит в следующее состояние. Таким образом, получается конечно-автоматная модель объекта (STD – State Transition Diagram), которую можно рассматривать как скелет, основную схему поведения объекта. Здесь еще нет ветвлений, присваиваний, вызовов функций и т.п., но основные контуры поведения объекта уже есть. Ранее этот переход от MSC к STD рассматривался как самый трудный момент проектирования сложных систем. В 2004 году аспирант нашей кафедры системного программирования Владимир Соколов придумал и реализовал систему автоматической генерации STD по MSC. В отличие от других известных нам методов (собственно, более-менее существенный результат мы знаем только один [35]), В. Соколов предоставил проектировщику возможность влиять на структуру генерируемого автомата, что существенно повысило удобство работы и качество получаемого результата [36].

Имея качественную STD-модель, уже нетрудно преобразовывать ее в SDL-диаграмму (Specification and Description Language, рекомендация ITU-T Z.100). Прежде всего, определяется, когда и при каких условиях этим объектом посылаются сообщения, затем добавляется работа с базами данных, какие-то вычисления и т. д.

SDL-диаграмма является конечным продуктом проектирования, конвертация в выбранный промежуточный алгоритмический язык высокого уровня (АЯВУ) и трансляция в объектные коды происходят автоматически. Технология REAL предусматривает определенные правила работы проектировщиков ПО, например, нельзя вносить правки непосредственно в тексте на АЯВУ, все исправления нужно вносить только в SDL-диаграммы, контроль версий следует вести в терминах объектов, а не каких-то промежуточных файлов и т.д. Все попытки горе-проектировщиков "для скорости" нарушить правила и влезть куда-то "грязными руками" приводят к огромным трудностям в дальнейшем развитии системы и сопровождении, поэтому все развитие технологии REAL направлено на ужесточение контроля – делается все, чтобы корректные действия пользователя были легкими и удобными, а попытки совершить некорректные действия блокировались.

Постепенно общее развитие технологии REAL разделилось на два направления. Первое связано с традиционной областью применения – системы реального времени. Здесь главной является SDL-диаграмма, в технологию добавляются различные средства снятия и анализа трасс сигналов, имитаторы нагрузки, эффективные способы организации вычислительного процесса.

Второе направление связано с проектированием информационных систем. Лидером этого направления стал Александр Иванов, который придумал и реализовал средства автоматической генерации различных форм ввода и вывода информации, средства разграничения прав доступа, учета связей между различными атрибутами и т.д. [37]

В этом варианте технологии главной является диаграмма классов. По ней автоматически строится описание базы данных на языке DDL (data definition language), программы CRUD (Create, Read, Update, Delete, т.е. создать, прочитать, исправить, удалить), API (Aplication Program Interface – интерфейс программных приложений) для работы с реляционной

базой данных как с объектно-ориентированной. По этой же диаграмме генерируются программы работы с визуальными формами, а с помощью диаграммы объектов задаются связи между атрибутами и ограничения прав доступа.

Программа "Студент" [38], автоматизирующая деятельность деканатов и ректората нашего университета (а также многих других университетов) практически полностью автоматически сгенерирована, и лишь очень малый процент подпрограмм был написан вручную на Visual Basic.

Как готовить системных программистов

С каждым годом растет интерес выпускников средней школы к профессии программиста. Профессия программиста привлекает еще и потому, что считается высоко оплачиваемой. В этом году многие выпускники математических школ, мечтавшие стать "чистыми математиками", выбрали для себя профессию программиста. Тем не менее, далеко не все стремящиеся стать программистами представляют, что нужно знать тем, кто хочет стать высококвалифицированным специалистом в области программирования. Многие ребята считают, что знание одного или нескольких языков программирования уже делает их программистами. Редакция журнала обратилась к одному из ведущих специалистов в области программирования, являющемуся одновременно крупным ученым и руководителем созданной им же фирмы, заведующему кафедрой системного программирования Санкт-Петербургского государственного университета проф. А.Н. Терехову с просьбой высказать свое мнение о подготовке системных программистов вообще и в СПбГУ в частности.

Профессиональным преподавателем Университета я стал почти случайно. Я читал спецкурсы, будучи еще студентом математико-механического факультета, руководил дипломными работами, 9 человек защитили кандидатские диссертации под моим руководством. Еще в молодые годы я начал руководить лабораторией системного программирования НИИ математики и механики математико-механического факультета. Когда уехал работать во Францию заведующий кафедрой математического обеспечения А.О. Слисенко (сейчас он заведует кафедрой в университете Париж-12), наш декан решил, что я буду хорошей кандидатурой на этот пост. На собрании кафедры меня попросили рассказать о своей программе. Она была очень короткой, всего из двух пунктов. Первый тезис: каждый преподаватель должен быть сначала исследователем, а уж потом – преподавателем. Я готов простить некоторые недоработки, но не готов простить начетничества, когда преподаватель сегодня почитает книжку, а завтра расскажет. Надо, чтобы преподаватели в основном рассказывали о своих работах или о тех, в которых они принимали участие. Второй тезис состоял в том, что надо соответствовать международным программам.

За каждым из этих тезисов была моя выстраданная позиция. Не люблю я "начетчиков". Страдал от таких преподавателей, когда сам учился, и, естественно, не хочу поддерживать их сейчас. А с международными стандартами бывали ужасные истории. Например, мы по роду своей деятельности много контактировали с группой молодых людей из Академгородка из Новосибирска, которые под руководством доктора наук Котова делали новую машину "Кронос". Мы в это время делали свою машину "Самсон", поэтому очень интересно было поговорить, пообщаться, обменяться результатами, и ребята произвели на меня незабываемое впечатление. Они часто приезжали к нам, жили у меня дома. Многих из них я хорошо помню до сих пор. Двое из них в конце 1980-х годов пробовали поступить в аспирантуру в американском университете. Оба были очень умными, я мечтал бы иметь таких сотрудников.

И не поступили. Не потому, что плохо говорят по-английски, или по какой-либо другой формальной причине. Они просто на половину вопросов не знали ответов. Хотя у них была очень мощная поддержка. Руководителем их лаборатории был А. Марчук, а его отец был президентом Академии наук, поэтому сотрудники А. Марчука имели дополнительные возможности, получали доступ к материалам, связанным с аспирантурой в Америке, которых младшие научные сотрудники других организаций не имели.

На меня это произвело оглушающее впечатление, потому что я привык думать, что мы, по крайней мере, в области программирования "впереди планеты всей". В некоторых областях это действительно так. В области техники трансляции, в области теоретических вопросов программирования, теории оптимизации. Но оказалось, что программирование за это время разрослось, и мы в своих работах, в основном, на оборону, очень многие аспекты просто упустили. И в 1992 году, по моим подсчетам, мы не охватывали даже 40% международного стандарта по специальности "computer science and software engineering". Я сказал сотрудникам кафедры, что нечего почивать на лаврах, нужно засучить рукава и заниматься, догонять мировую цивилизацию. Была масса проблем, были дискуссии на Ученом Совете. Были неприятные и даже болезненные ситуации, но в результате сформировалась новая кафедра.

Так 6 лет назад я стал заведующим кафедрой системного программирования. Я начал честно воплощать собственную программу, развивать исследования, которых у нас раньше не было. Думаю, что сейчас мы охватываем примерно 80% международного стандарта, но не могу обещать, что скоро мы охватим все 100%. Именно сотрудники нашей кафедры руководят командами нашего Университета на международных соревнованиях. Мы дважды стали чемпионами, но для меня еще важнее, что в течение 5 лет подряд мы были в призовой "десятке" из 2500 команд. Заметна стабильность результата. Я считаю, что кафедра системного программирования, несмотря на свою молодость, развивается достаточно успешно.

Сейчас я хочу сосредоточиться не на успехах (я отчетливо понимаю – сегодня есть успех, а завтра тебя никто не вспомнит), а на проблемах, которые мешают нам развиваться дальше. Их несколько, и я не знаю, сумею ли я связно о них рассказать в этом интервью. Но попробую.

Начну я, как ни странно это, может показаться читателю, с практики. Студенты должны иметь практику. Программирование – это такая специальность, которой не научишь у доски с мелом в руках. Для того чтобы лучше понять возможные пути организации практики, мысленно перенесемся в Оксфордский университет, где мне доводилось читать лекции, и я специально изучал местную постановку образования. Конечно, там иногда отдает некоторой "замшелостью", но, тем не менее, сотрудники университета свято следуют традиции и с большой неохотой расстаются с чем-то старым. Иногда им это можно поставить в минус. Например, сейчас Оксфордский университет несколько отстал в области естественных наук от Кембриджа, и специалисты говорят, что одна из причин этого отставания в том, что в Оксфорде на 60 лет позже отменили обязательное обучение латыни. В Кембридже отменили в 1920-х годах, а в Оксфорде только в 1980-х. И на протяжении этих 60 лет многих молодых людей отпугивала необходимость учить мертвый язык только потому, что так делали 800 лет назад.

Верность традициям достойна уважения. Например, в Оксфорде для каждого предмета есть теоретический курс, то есть лекции; практический курс, когда преподаватель со студентами решает задачи в аудитории, у доски, у компьютера, но при этом стоит рядом, и практикум по каждому курсу – студент должен выполнить некоторую

работу самостоятельно. Причем не раз в полгода, как наша курсовая работа, одна на весь семестр, а по каждому предмету каждые две недели. Есть огромные аудитории, сотни свободных вычислительных машин. Приходишь, садишься, решаешь, и затем свой результат показываешь тьютору. Это еще одна особенность – персональное обучение. По каждому предмету час в неделю студент работает со своим тьютором – таким преподавателем, который за него отвечает. Было пять предметов в неделю – значит, пять часов студент с преподавателями проведет один на один. Причем тьютор составляет основу преподавания. Именно тьютор ведет вступительные экзамены, именно тьютор отбирает себе студентов, а не колледж или университет в целом. Именно тьютор может сказать своему коллеге: "Знаешь, я уже набрал себе нужное количество студентов, есть еще такой-то студент, возьми его себе в другой колледж".

Если читают, например, операционную систему реального времени, то каждый студент должен написать программу: управление памятью, управление процессорами, управление временем. Преподаватель смотрит не только на результат, но и на то, как написана программа. По всем предметам есть лекции, практика и практикум.

У нас с этим слабее. Курсовые работы – раз в семестр, и часто они превращаются в фикцию. Трудно изменить эту ситуацию к лучшему, потому что нет соответствующих материальных ресурсов. Как обязать всех студентов по каждому предмету сделать самостоятельную работу, если мы не можем им обеспечить полноценный доступ к вычислительным машинам? Классы всегда перегружены. У нас не принято, чтобы студенты занимались без преподавателя: и вирус занесут, и что-нибудь украдут, и что-нибудь сломают. Самостоятельную работу очень трудно наладить. О тьюторстве я даже не мечтаю. Это прежде всего вопрос денег. У нас наверняка нашлось бы много хороших преподавателей, но для того чтобы обеспечить индивидуальное обучение, сколько надо преподавателей и какое потребуется финансирование? Хотя еще со средних веков известно, что обучение – это всегда работа мастера с подмастерьем с непосредственной передачей опыта. Единственная "живая" практика у нас – на пятом курсе, полугодовая преддипломная практика. И здесь тоже есть свои проблемы. Хорошо если практика была в известной фирме, которая успешно и продуктивно на современном технологическом уровне

занимается программированием. Но так бывает не всегда.

Отсюда первый тезис – сегодня надо практику реально совместить с теорией. Формально говоря, все у нас есть. Студенты пятого курса полгода проходят преддипломную практику. И чем это кончается? Вот у меня толстая пачка отчетов по практике. Люди просто пристраиваются работать (например, программистами) в какую-нибудь малоизвестную контору, чаще всего я даже названия этой конторы не знаю. Они работают. С одной стороны, какие могут быть претензии? Человек полгода проработал, получал деньги, кому-то был полезен. Спрашиваю я как заведующий кафедрой: "Чему ты там научился?" "Вот, получил практический опыт программирования на Java или C++". "Как была организована работа?" "Никак. Начальник дал задание, я написал программу". "Как был организован коллектив? Какие были взаимоотношения? Как велось планирование, отчетность? Были ли еженедельные собрания? Была ли регулярная проверка качества? Были ли перекрестные чтения?" "Не было" "Так чему ты, милый, там научился?" "Программированию" "Почему тогда надо говорить, что ты заканчиваешь математико-механический факультет старейшего университета России?" Здесь явное несоответствие теории и практики.

Приведу другой пример. У нас несколько человек проходили практику в зарубежной фирме здесь, в Санкт-Петербурге. В этой фирме все хорошо организовано, но есть другая крайность: везде завеса секретности. Даже если дипломную работу студент написал там, ее в университете защитить нельзя. Надо защищать в фирме, организовывать ГЭК. Это целая проблема. Ладно, в конце концов, даже на военных работали, могли все организовать. Но ведь человек отрывается от коллектива, ничего не может обсудить. Самое главное в обучении – это беседа, разговор. А тут несколько человек проходят практику и даже со своими однокурсниками не общаются. Запрещено. Совершенно другая крайность. Очень высокий уровень работы, но слишком индивидуальный. Люди из этой фирмы, возможно, скажут, что я не прав, что у них есть и семинары, и регулярное обучение. Но я говорю о том, что вижу по результатам работы наших студентов.

Некоторые студенты проходят практику на предприятиях. Я требую, чтобы это была не только работа, но и обучение. Тоже возникают противоречия: "Мы – производственное предприятие, нам надо

зарабатывать деньги, приносить прибыль, поэтому заниматься чисто учебными делами как-то не с руки. Нет-нет, мы понимаем, что надо готовить кадры, но все должны заниматься своим делом".

Еще раз повторю, поскольку для меня это важно: практика – это не просто работа "от забора до обеда", "переделал готовую программу" или "спаял электронную схему". Практика подразумевает некоторое исследование, обучение организационным формам, современным методам. Практика должна быть разнообразной. Например, электронщик должен не только спаять, но и спроектировать схему, спроектировать кристаллы, которые внутри. Он должен участвовать в отладке – и не просто участвовать, а сделать тестовое окружение. Электронщики должны программировать. Это все тесно связано.

Вас не удивляет, что я все время вспоминаю электронику? У меня есть целый отдел электронщиков, у них и руководитель отдела – математик, и многие сотрудники – математики. Сейчас электроника такая, что все равно надо программировать. Но приходят инженеры, которые не знают, как простейший тест написать. А как написать не один тест, а систему тестов для исчерпывающего тестирования, им даже объяснить невозможно. Практика должна включать в себя организационные аспекты, элементы дизайна, элементы разработки, и самое главное – доводку до результата. За полгода всегда можно получить результат. На эту тему можно говорить долго.

Тезис второй – чему учим? Вот передо мной лежит программа 35.15. По этой программе учится отделение информатики математико-механического факультета. Мы с сотрудниками нашей кафедры принимали участие в ее разработке. Для сравнения скажу: у нас отделение прикладной математики учится по специальности 01.02. Математическая статистика, моделирование, теоретическая кибернетика – это все замечательно. В дипломе написано: "Математик. Системный программист". Я обращаюсь к авторам этой программы: "Покажите, где здесь программирование". На первом курсе учат программированию на языке С, и все. Я же не говорю, что в программу включили лишний материал. Все нужные вещи: и моделирование нужно, и кибернетика нужна, и распознавание образов, и вопросы оптимизации. Но зачем пишут в дипломе "Системный программист"? Вот я заведую кафедрой системного программирования. Надеюсь, что я знаю, что такое

системное программирование. Давайте я тоже буду учить программированию, а писать в дипломе "специалист по методу Монте-Карло". Кому это понравится? Конечно, все понимают, что надо привлечь людей, звучит название специальности хорошо, но не совсем соответствует содержанию курса.

Вернемся к тому, что я действительно считаю хорошим. Например, к специализации 35.15: математические основы информатики, информационные системы, технологии программирования, архитектуры вычислительных систем, сети. Далее: экспертные системы, теория оптимизации баз данных, Интернет и Интранет, инструментальные системы для C++, Java-технологии, инструментальные средства визуального программирования, инструментальные средства логического программирования, технология трансляции, языки и системы программирования, архитектура ЭВМ, программно-аппаратные комплексы, операционные системы реального времени, телекоммуникации и так далее.

Мы на кафедре подсчитали часы по этой программе, и все равно 50 % – это "чистая математика". Самый главный недостаток даже не в этом, а в том, что та половина времени, которая отведена на специальность, отнесена на конец. На первых трех семестрах – только 4 часа в неделю. Представьте себе: человек поступил на отделение информатики. Не на отделение "чистой математики", не на отделение астрономии или механики. И учится полтора года, три семестра, имея 4 часа программирования в неделю! В самых разных видах, все про все. Как можно его научить? Самое ценное время уходит. Я даже встречался с заместителем министра образования, обсуждал это все у нас в Университете, в УМО. Сценарий разговора всегда был таким: "На кого жалуетесь, вы же сами профессор, член УМО! Вот и вносите предложение, сократите то, добавьте это, для чего и создано УМО". Хорошо. Но как только я пытаюсь это делать, мне сразу говорят: "Как? Ты что? На факультете работают старые профессора, которые и тебя учили математическому анализу, алгебре, высшей геометрии... Если ты уменьшишь нагрузку, их надо будет сокращать. Неужели ты хочешь уволить старых профессоров-математиков?" Конечно, не хочу. Правила, установленные Министерством, предписывают, что количество преподавателей должно быть связано с количеством студентов. Если число студентов уменьшилось, соответственно уменьшается число

преподавателей. То есть такая простая вещь как перераспределение часов, сталкивается с министерскими правилами, и никакой Университет, никакое УМО изменить это не может.

В нашей программе есть федеральный компонент, вузовский компонент (региональный) и компонент по выбору. Федеральный компонент: математический анализ – 4 семестра, количество часов в неделю: 8, 4, 6, 6. Алгебра и теория чисел – 3 семестра, часы: 4, 4, 4. Геометрия и топология – 3 семестра, часы: 4, 4, 4. Дифференциальные уравнения – 2 семестра, по 4 часа. Функциональный анализ – один семестр, 4 часа. Когда я был студентом, было два семестра. Уравнения математической физики – один семестр, но 6 часов. Теория вероятностей и математическая статистика – два семестра, 7 часов.

Когда я учился, вся теория вероятностей ограничивалась изучением меры Лебега. О том, что вероятность находит применение в нашей науке, я узнал лет через 15: оказывается, отказы вычислительной техники распределены по закону Пуассона. Так можно оценить вероятность отказа, но узнал я об этом, только когда столкнулся на практике. Мы сделали новую вычислительную машину, от нас потребовали расчет, я взялся за книги и с удивлением узнал, что теория вероятностей – полезная наука. Мне было уже под сорок. Ничему такому – предсказывать отказы, считать их интенсивность, ничему этому нас не учили. Одни интегралы, интегралы, интегралы. Конечно, из этих интегралов потом следуют и закон Пуассона, и все остальное, но мостика между мерой Лебега и еще чем-нибудь полезным нет.

Есть вычислительный практикум – три семестра по 2 часа, и есть программирование – три семестра, 3, 2, 2 часа. А здесь должны быть общепрофессиональные дисциплины (федеральный компонент): архитектура вычислительных систем компьютерной сети, операционная система оболочки, структура алгоритма компьютерной обработки данных, базы данных и СУБД, компьютерное моделирование, компьютерная графика, теория формальных языков и трансляций, спецкурсы по выбору, спецсеминары. Но против всего этого – пустые клетки.

Посмотрим на третий курс (пятый-шестой семестр). Десять часов в неделю. Как можно за такое время научить чему-либо студентов? И

только на 4–5 курсе начинают учить "по специальности". Но на пятом курсе уже преддипломная практика, там только спецкурсы, и то понемножку. То есть мы можем учить практически только четвертый курс. Разве так можно? Вот где проблема. Причем не могу сказать, что у меня есть решение.

Тезис третий – необходимость теории. Один мой бывший однокурсник – сейчас профессор Западно-Берлинского технического университета. Я бывал у него, и он у нас бывал несколько раз. Я однажды его спросил, чему учат у них в университете. Выяснилось, что изучают и логику, и все остальное, но только формулировки теорем. Я его спросил: "Скажи честно, если я сейчас подойду к какому-нибудь вашему студенту и спрошу, что такое теорема Геделя о неполноте, он ответит?" "Нет, – говорит, – даже не вспомнит". "Тогда зачем так учить?" "Ну, положено. А зачем вам теорема Геделя?" "Хотя бы для того, – говорю, – чтобы молодой специалист имел представление о границах применимости теории. Теорема Геделя говорит о том, что корректность арифметики нельзя проверить средствами самой арифметики, и дает теоретические ограничения, предлагает искать какие-то метатеории, привлекать дополнительные возможности. Если человек об этом даже не подозревает, он будет в каких-то местах напрасно тратить время. У меня был случай, когда один выпускник кафедры математической физики, работающий у нас, должен был реализовать анализ потоков данных в программе. Он довольно быстро все реализовал. Самая мощная машина тогда была 486-я, и он на ней 4 часа тест из 20 строк гонял. Я посмотрел программу – простой перебор путей в графе. Я его спрашиваю: "Ты разве не знаешь, что число путей в графе растет экспоненциально относительно числа вершин?" "Не знаю. Подумаешь, экспонента! Машина железная, все посчитает". Я ему долго читал лекцию про актуальную бесконечность, о том, что если в программировании видишь экспоненту, то надо искать другое решение. Это не значит, что надо сдаваться. Я часто привожу студентам такой пример. На конференции, посвященной 1000-летию алгоритма, в Ургенче (на родине Аль-Хорезми), была представлена статья Ю.В. Матиясевича "Что нам делать с экспоненциально сложными задачами?" Вот это мне нравится, это конструктивный подход. Не просто "Все, сдаюсь, больше ничего сделать не можем". Всегда можно найти частные случаи. Есть и другая противоположность "теоретического" восприятия задачи. Другой не менее известный ученый меня мучил, когда я сдавал кандидатский

минимум: что значит теорема Геделя о неполноте? И заставлял меня на экзамене (за две недели до защиты диссертации!) признать, что из этого следует, что машина не все может. Но это не так! Нет общего подхода – найдем частные.

Например, знаете, на чем основана шифрация? Сводят задачу к какой-нибудь трудноразрешимой (например, разложению числа на множители). Мои коллеги, практики, сделали систему шифрации. Популярная система, продается хорошо. Они меня попросили показать кому-нибудь из коллег; чтобы услышать мнение о том, насколько их работа теоретически обоснована. Я попросил Ю.В. Матиясевича посмотреть их статьи. Он минут 10 смотрел, тут же указывает мне фразу: "Поскольку никакого другого способа вычислить, кроме простого перебора, нет, – это трудноразрешимая задача". Мне даже обидно стало, что сам не разглядел. А вдруг найдется какой-то другой метод, который для данного конкретного класса задач даст хороший алгоритм? Тогда все это рассыплется, как карточный домик. Возможно, такого метода и не найдется, но математика отличается тем, что все надо доказывать. Они не доказали, что другого метода, кроме прямого перебора, нет.

Итак, нужны теоретики, нужны исследования и нужны доказательства. И уж если человек говорит, что эта система шифрации стойкая – будь любезен, докажи, что никакого способа, кроме полного перебора, нет. Другое дело, что и в некоторых классических задачах криптографии этот вопрос до сих пор открыт.

Если у тебя нет теоретической подготовки, то так и будешь перебирать пути в графе, не задумываясь об экспоненциальной сложности алгоритма. На практике это означает, что алгоритм работать не будет. Теория дает определенные границы: за что браться, за что не браться, где искать, где не искать. Где с самого начала надо искать срез, подзадачу, специальный случай.

Все-таки это знание составляет малую часть нашей профессии 5–10%. Просто есть вещи, которые надо знать. Если ты их вообще не знаешь, можешь налететь на такие грабли, что лоб себе расшибешь.

... Однако, давайте поговорим о программировании. Я много раз читал лекции в Гамбурге в Классическом университете и в Техническом университете. Там даже поговорить о программировании часто не с

кем. Две крайности: или "коробочники", которые умеют пользоваться стандартными программами, или теоретики, которые занимаются чем-нибудь таким, что неизвестно когда на практике осуществится. А людей, занимающихся нормальным программированием, часто и не встретишь.

Я приведу еще один пример. Примерно в 1975 году мы получили первую ЕС ЭВМ 1030 среди гражданских организаций СССР, об этом даже в газетах писали. Первые ЕС ЭВМ шли только на оборону. И вот ленинградский математико-механический факультет получил самую первую машину за счет того, что мы делали много программ для ЕС ЭВМ. Машина часто ломалась, а мы сидели вечерами и даже ночами. Весь чай был выпит, все возможные темы обсуждены, И вот я начал одну девушку-оператора учить своему любимому языку АЛГОЛ-68. Такой сложный язык программирования, и редко какой студент мог его освоить в полном объеме. За несколько месяцев я научил ее так, что не каждый студент мог с ней сравниться. Говорю ей: "Теперь тебе надо переходить работать программистом. И зарплата выше, и работа интересная. Ведь что такое работа оператора? Поставить диск, загрузить машину". И тут я с ужасом понял, что она не знает, что программировать. Она не знает, как можно итеративно вычислить квадратный корень, она не знает, как устроен транслятор. Она знает язык программирования, экзамен сдать может, а программировать не может. На меня этот эпизод произвел очень сильное впечатление.

Прошло 25 лет, вроде бы многое изменилось. Но посмотрите, мы с вами здесь сидим, каждые 10 минут дверь открывается. Каждый третий приходит с вопросом: "Андрей Николаевич, я хочу у вас работать. Я слышал, что у Вас много людей занимается интересной работой". "Отлично, что ты умеешь?" "Я умею программировать на Паскале". "А что ты знаешь-то?" "Ну, как, я же научусь" "Посмотри на список спецкурсов кафедры. Что из этого ты знаешь?" "Ничего" "И как ты будешь работать? Я тебя определю в группу "Телефония". Ты знаешь, что как устроено? Что ты там будешь программировать? Ты умеешь писать a:=b, a:=b+c, но ведь это не программирование. Надо знать, что программировать".

Результат таких разговоров может быть двояким. Кто-то всю жизнь меня после этого ненавидит за то, что он хотел заниматься интересным делом, а злобный Терехов на него ушат холодной воды вылил, а бывают

такие упрямцы, которые говорят: "Ничего, я научусь". Хорошо, первые три месяца – стажировка, и, если выяснится, что человек работать не может, ему просто вежливо скажут: "Извини, друг, но нам надо двигаться дальше". Это не значит, что человек пропадет, может быть, он попадет в другую группу. Бывает, что люди с третьей попытки свое место находят. Бывает так, что человек научится в процессе работы, но это скорее исключение, чем правило. Это обходится большими усилиями, чем у студента в процессе учебы, но зато закрепление совсем другое и мотивация другая.

Итак, между собой взаимосвязаны теория, практика, границы применимости теории, некоторые личные знания. Я объясняю своим студентам, что системный программист – это сфера обслуживания. Мы не делаем конечных продуктов. Например, человек производит расчеты. У него есть какой-то результат. При этом он пользуется трансляторами, операционными системами, вычислительными машинами, которые придумали другие люди. И системщики – как раз те люди, которые делают трансляторы, инструментальные средства, разрабатывают методологии по их использованию. А потом уже прикладные программисты этими средствами и методологиями пользуются и получают конечный результат. Очень часто, кстати, люди видят только конечный результат, особенно когда идет речь о делении денег, и совсем не видят той дороги, тех трудностей, которые были преодолены, чтобы этот результат получить. Системное программирование – это довольно старое название. По-русски мы хорошо знаем, что такое системное программирование. Но были проблемы, как перевести это словосочетание на английский язык. Есть наука computer science, она более теоретическая. А есть наука software engineering. Вот я, заведующий кафедрой software engineering. Engineering – это разработка программного обеспечения. И мне кажется, что это довольно четко сейчас определилось.

Теперь следующий тезис, следующая проблема. Вот защищается мой аспирант у нас на Совете. Я сам – член Ученого Совета. И каждый раз попадается какой-нибудь, мягко сказать, недоброжелатель, который, поскольку на Ученом Совете может выступать кто угодно, говорит: "Вы знаете, я не понимаю, почему это математика. Нет теорем, нет доказательств сходимости, почему эта диссертация защищается на математическом факультете?" В реальной действительности самый

лучший ответ – сказать: "Вы ничего не понимаете в этой области". Времена, когда математика была только в теоремах, кончились как минимум 100 лет назад, а может быть, и больше. В середине XIX века Ч. Бэббидж придумал вычислительную машину, в которой были все основные элементы (процессор, память, программа, хранящаяся в памяти). Дочь Байрона Ада Августа леди Лавлейс пятистраничный доклад этого Ч. Бэббиджа на итальянском языке преобразовала в 100-страничный английский текст, где впервые ввела слова "Переадресация", "Процедура", "Цикл" – это что, не математика? Это было сделано более 150 лет назад.

Проблема решается просто, если говорить серьезно. Есть специальность "Математическое обеспечение вычислительных машин, сетей и систем". Это не математический анализ, не математическая физика. Есть критерии, предъявляемые к кандидатской диссертации: новые результаты с практическим внедрением, языки программирования, их реализация, операционные системы и, самое главное, модели. Математика начинается тогда, когда мы можем что-то формализовать, когда мы можем сформулировать задачу настолько точно, что можно построить алгоритм.

Почему возникли потребности в строгой формализации? Потому что пришло понимание, что некоторые задачи в принципе нельзя решить, потребовалась алгоритмическая формализация, и с помощью этих формальных методов удалось доказать неразрешимость нескольких проблем. Первые такие доказательства были получены в 30-х годах XX века. Почему, когда я доказал про что-то, что этого нет и быть не может, – это математика, а когда я построил формальную модель и по ней сделал алгоритм, который работает, – это уже не математика? Никто не сказал, что только отрицательные результаты являются математическими. В нашей стране такой консерватизм особенно силен. Граница между "наукой" и "не наукой" в данном случае, когда речь идет о программировании, достаточно понятная, но трудно формализуемая, поэтому вызывает массу проблем, профессионал их знает. И постоянно на Ученом Совете кто-нибудь начинает возражать, что это "не наука". Важно создать новую модель, новый язык, новый метод, новый алгоритм, показать, что он отличается от других. Я уже много раз был оппонентом, а не только руководителем диссертации. Для новичка, для человека со стороны это, может быть, будет даже удивительно. В задачу

оппонента входит не только оценить, хорошая диссертация или плохая. Главная задача – оценить соотношение этой работы с другими известными работами. Не забыл ли диссертант, что это уже сделано? Он сравнивал свою работу с другими? Не забыл ли он какой-то важный результат, который был получен другим, а у него даже не упомянут? Насколько корректно проведено сравнение собственной работы с другими известными работами? Есть такая наука, которая называется software engineering, в ней есть своя область деятельности, свои требования, свои критерии. Профессионалы знают, что сделать хороший транслятор – это очень трудная задача. Сделать технологию, которой будет пользоваться широкий круг людей, тоже очень трудно.

У меня защищалось в этом году 19 человек. Приходит студент V курса, проведший полгода где-то на практике. Показывает свою программу. Я говорю: "Хорошо, давай посмотрим, насколько научна твоя работа. Что здесь самое главное? Во-первых, насколько это новый результат? Писал ли кто-нибудь об этом раньше?" "Не знаю" "Ты делал обзор литературы?". "Нет"... Чем отличается исследователь от практика в худшем понимании этого слова? Исследователь поймет, что не надо изобретать велосипед. В наше время, когда есть Интернет и другие каналы получения информации, имеет смысл посмотреть, что сделали другие. С этого надо начинать. Далеко не всем это приходит в голову. Нашел, что такого результата нет, и ты его сам получаешь. Но есть похожие результаты. Проведи сравнительную характеристику. Мне не нравится такая ситуация: люди работали, писали дипломы, писали диссертации. Спрашиваешь: "Чем ваша технология лучше, чем остальные?" И следуют аргументы: "С одной стороны, нельзя не признать, с другой стороны, нельзя не согласиться..." Спрашиваю: "Ребята, вы потратили на эту работу столько сил и времени. Мы интуитивно понимаем, что это хорошо. Но разве так трудно все это четко сформулировать? У многих программ есть демо-версии, и сравнение их с вашим вариантом входит в работу". Нынешняя молодежь с трудом понимает, что просто сделать что-то, что работает, – это меньше половины дела. Настоящий исследователь должен смотреть, как работают другие, скачивать демо-версии, читать инструкции – как у них запускается программа, научиться запускать, посмотреть, поработать, понять, что хорошо, что плохо, может быть, заимствовать какие-то идеи. Необходимо четко сформулировать, в чем твоя заслуга. Что ты такого сделал, чего у других нет? Сейчас этим занимаются только бедные

диссертанты, и то, к сожалению, есть такая десятилетиями сложившаяся практика, что они занимаются этим в последний момент, когда уже "кирпич" пишут. Положено иметь обзор литературы по теме – они и сидят в библиотеке по 3 месяца в самом конце. Между прочим, на моих глазах была ситуация, когда аспирант защищается по отладке, а его на Совете спрашивают: "Как это соотносится с такими-то методами отладки?" Оказалось, что он в библиотеке Академии Наук сравнивал свой метод с американскими работами, а то, что в СССР есть такие работы, и не знал...

Сейчас я попытаюсь еще раз сформулировать эту идею. Дали тебе работу – не поленись, потрать время, посмотри, что есть у других. Какие есть методы. Не изобретай велосипед. И в то же время – никакого низкопоклонства перед Западом, даже если там есть какие-то публикации. Посмотри, насколько они хороши. Если хороши – используй, со ссылкой. Для этого и существует вся мировая наука, чтобы пользоваться ее результатами. Не надо ничего высасывать из пальца. Но плохо, когда человек не потратил времени на то, чтобы лишний раз посмотреть, а потом ему кто-то другой указывает на точно такие же, если не лучшие, результаты других исследователей.

Итак, важно умение сформулировать задачу, умение посмотреть на нее со стороны "что есть в мире" и умение четко объяснить, что у тебя нового.

Я в очередной раз отвлекся, но все-таки закончу. Приходит студент из мелкой фирмы с программой. Никакого анализа не сделано, сравнительных характеристик не сделано, никаких моделей не построено, ничему новому он не научился, то есть просто сидел и работал на зарплату. Что теперь делать? Я уверяю, что это не может быть дипломной работой. Дипломная работа обязательно должна нести элемент научного творчества.

Преддипломная практика – на первом семестре пятого курса. И когда человек нормально практику прошел, чему-то научился и пришел с результатом, у него еще есть время на то, чтобы превратить эту работу в дипломную. Бывают смешные случаи. Два студента сделали транслятор с Java. Никаких ссылок – будто они в безвоздушном пространстве. "Сколько вы знаете трансляторов с Java?" "Один или два". Поискали в

Интернете – оказалось, 20–30. "Дайте сравнительные характеристики". Выяснилось, что проигрывают в 500 раз одному из трансляторов. Говорю: "Ребята, я читал про унтер-офицерскую вдову, которая сама себя высекла, но не думал, что это может быть среди студентов мат-меха". Посмотрели программу – это оптимизирующий транслятор с языка C. Java – это интерпретатор по определению, поэтому их и сравнивать по эффективности нельзя – это задачи с совершенно разными целями. Умение грамотно сформулировать задачу здесь было необходимо. Хорошо, что я успел увидеть эту дипломную работу, иначе над нами бы еще долго смеялись.

Или другой пример. Приходит студент с работой по теме "распараллеливание". Я не был научным руководителем, но, прочитав работу, почувствовал, что я такое сто раз слышал. У меня в это время дочь в Англии была и как раз занималась распараллеливанием. Позвонил ей, она перечислила методы и средства, которые сейчас используются. На мальчика было жалко смотреть. Конечно, он получил нормальный результат, что-то сосчитал и думал, что изобрел новый метод. Он был уверен, что никто в мире этого не знает. Откуда такая уверенность? Конечно, мы учим своих студентов тому, что "мат-мех лучше всех", но всему есть свой предел. Так же нельзя, ты же не один в мире.

Итак, следующий тезис такой. Мы готовим университетских людей, которые должны быть способны – я на этом жестко настаиваю – не только написать программу и придумать эффективный алгоритм, но и сравнить, обосновать, понять теоретически возможные границы применимости. Это обязательно входит в университетское образование.

Но кто сказал, что только такие люди нужны? Мы переходим к следующему вопросу. Вот передо мной лежит проект программы "Федеральная целевая программа электронной России на 2002 – 2010 годы". Мне очень понравилась фраза: "В настоящее время наблюдается определенное перепроизводство "массовых" специалистов с высшим образованием, в частности, в области создания программного обеспечения и технической поддержки информационно-вычислительных и коммуникационных систем, которые часто вынуждены выполнять функции, не требующие широкой теоретической подготовки, которые могут и должны замещаться специалистами со

средним образованием". Еще год назад, в связи с конференцией по Наукограду, я изучал материалы про аналогичные зоны в Индии, в Ирландии. В Индии готовится в год 5000 master of science (аналог нашего высшего образования) и 27000 бакалавров. Мы пытались прикинуть, сколько у нас готовится. Получилось несколько тысяч программистов с высшим образованием очень разного профиля. У нас и железнодорожный институт, и институт водного транспорта готовит программистов, но все равно получается всего несколько тысяч. А бакалавров нет вообще. У меня, как видите, вся стена увешана досками с именами наших сотрудников, которые готовят команды на международные олимпиады по программированию. Пять лет подряд в десятке из 2500 команд, два года подряд – чемпионы. Это приятно. Но только за 5 лет ни один из них не стал профессиональным программистом. Они математики. Сейчас Андрей Лопатин перешел на нашу кафедру и переучивается. Нельзя сказать, что у него все так бравурно идет – он тратит силы, время, и надеюсь, что из него мы сделаем профессионала. Это первый такой пример за 5 лет. Я и раньше пытался сказать, что мы слишком сильно сконцентрировались на результатах олимпиад. Действительно, важно побеждать на олимпиадах, важно иметь исследователей. У меня на предприятии есть несколько человек, таких, что, если хотя бы один из них уйдет, мое предприятие лопнет. Есть люди, на которых держится предприятие, потому что они придумывают новые идеи, они все цементируют. У меня на предприятии больше – 200 человек, а таких, без которых все остановится – 2–3, может быть, 5, если молодых прибавить. Если уйдут другие, можно найти им замену. Но именно остальные 200 человек выполняют основной объем работы. Бакалавриат у нас как-то не прижился, но нужно найти какие-то упрощенные формы подготовки программистов.

Иногда говорят: "У нас такой отсев, у нас уезжают на Запад". У нас на кафедре немногие уезжают – за все годы человек 15 из 250–300 выпускников уехали за рубеж. Многие из этих 15 не просто так уехали, а по согласованию со мной перешли в компании в США, которые со мной же сотрудничают. То есть можно сказать, что эти выпускники со мной же и работают, только с другой стороны. Кстати, это очень помогает. Всегда приятно, когда там свои люди, говорящие по-русски, с нашим менталитетом. Дело не в английском языке, у нас по-английски все говорят, но, когда там сидит человек и может поговорить на "ты" с

кем-то из наших это всегда сильно помогает. Так что это даже нельзя назвать потерей. Бывает отсев, когда уходят в другие компании. Бывает так, что выпускник нашей кафедры уходит за зарплатой в 1000 долларов. Я столько платить не могу. А они уходят и часто страдают. Вот сейчас Artificial Life объявила, что закрывается, другие компании сокращают количество сотрудников. Я не то что мстительный, просто я слежу за своими учениками. Этот был вынужден уйти сюда, этот туда. Не все у них плохо, слава Богу, но за длинным рублем гнаться не стоит, особенно в молодые годы.

Еще немного отвлекусь, уж больно пример для меня интересный. Когда уезжали хоккеисты уровня Ларионова и Фетисова, это было одно. Люди, сделавшие имя, они за рубежом уже были хорошо известны. Когда уезжают 19-20-летние ребята, чтобы годами играть в фарм-клубе, то потом, даже когда пытаются вернуться, уже здесь не могут играть на хорошем уровне, потому что там они не играли у таких великих тренеров, как Тихонов, Тарасов, и имели мало игровой практики. Аналогия абсолютная. Одно дело – человек закончил математико-механический факультет, еще лучше – аспирантуру, защитился, получил 5-10 лет стажа, поработал, принес пользу своему предприятию, своему родному университету, научил молодых специалистов и поехал. Вперед, я даже помогу. Это профессиональный рост, это интересная работа. Я вовсе не к тому, что нельзя ездить. Он там получит новые знания, новые силы, новую информацию для себя и, возможно, для нас. Может нам принести заказы, интересные научные исследования. Все это понятно. Я считаю, что это нормально. Совсем другое дело, когда уезжает совсем молодой человек. У меня был случай, когда студент полгода недоучился, диплом не защитил и уехал. Ведь диплом Ленинградского – Петербургского университета во всем мире признается. "Вы извините, может быть, мне перейти на другую кафедру, чтобы вашу кафедру не позорить?" "Да, – говорю, – и так уезжай!" Но я считаю, что это уже перебор. Кому он там нужен? Сейчас там перепроизводство программистов, массовые сокращения. Опять-таки не подумайте, что я злорадствую. Хотя, конечно, тех, кто меня бросил, тем более "на полуслове", не передав материалы, не передав информацию, я тоже запоминаю. Были разные неприятные случаи, но мне не хотелось бы о них подробно рассказывать.

Должны быть какие-то законы. Я в начале разговора вспоминал про

федеральную составляющую. Я не могу этот закон обойти. Если я его не выполню, то диплом будет считаться недействительным. Эти вопросы надо решать на государственном уровне. Точно такая же ситуация и с отъездом молодых специалистов. И она, кстати, решается. Не решена, но решается. Если этим интересуетесь, поищите в Интернете. Во многих странах дается кредит на образование, который ты должен отработать. Если уезжаешь – будь любезен, отдай. Неужели мы настолько богаты, чтобы об этом не думать? Должны быть определенные жесткие государственные законы. Если студент или его родители платили по 5000 долларов за семестр – уезжай, куда хочешь. Но если на тебя государство истратило столько денег – почему мы должны отпускать человека, который у нас получил образование?

Я понимаю, что у человека, занимающегося классической математикой, могут быть проблемы. Один мой приятель, алгебраист, который недавно уехал, говорил: "Если бы наше государство хотя бы треть тех денег, что в США, платило, я работал бы здесь. Пойми меня, я не хочу заниматься программированием". Все понятно, я тоже не хотел бы заниматься алгеброй. А это – фундаментальная наука, и государство ее практически не поддерживает. А у него семья, двое детей, вот он и уехал. Я в него камнем не кину. У него выхода не было, его государство так подставило. И не он один. Но это "классическая наука". А люди, которые занимаются прикладными вещами, живут. Причем, не только программисты. Я сейчас общаюсь, например, с людьми, которые корабли строят. Есть заказы, есть интересная работа, есть зарплата.

Если программист бедный – значит, он плохо работает. У нас такая специальность, что на кусок хлеба заработать можно. Для меня отъезд за границу не представляется серьезной проблемой. Когда говорят про "утечку мозгов", это скорее лозунги для демонстрации. Создайте рабочие места, обеспечьте людей интересной работой, обеспечьте приличную зарплату (пусть не на уровне США, но такую, чтобы можно было содержать семью) – и почти никто не уедет. Поедут только те, кто любит приключения.

Что такое государственная поддержка? В Ирландии, если предприятие делает программное обеспечение, оно платит 10% налога с прибыли при норме 28%. Но это относится только к прибыли, полученной от программного обеспечения. Программное обеспечение отличается в

лучшую сторону от многих других видов продукции тем, что это возобновляемый ресурс. Индия, благодаря развитию телекоммуникаций и программного обеспечения, получает большие деньги с экспорта, в страну возвращаются те, кто ранее ее покинул. Ирландия – маленькая страна, но производит 40% программного обеспечения в Европе. Десять лет назад это было трудно предположить, но многое решает разумная государственная поддержка. Все должно быть в разумных пределах.

Государственная поддержка – важный момент, и есть положительные примеры – Индия, Ирландия, Израиль. Имеется в виду не только налоговая политика. Например, хорошо подобранная программа обучения – это государственная поддержка. Инфраструктура телекоммуникаций – это государственная поддержка. Удобный проезд до крупнейших учебных заведений – тоже государственная поддержка. Вспомните наш мат-мех! У нас мог бы быть совсем иной контингент студентов, если бы электричка ходила полчаса, а не час, как сейчас. В Сингапуре в Школе программирования на IV курсе занятия начинаются в 17 часов. Дело в том, что, если ты не работал на фирму полный рабочий день – диплом не выдадут. Какой ты программист, если никогда не работал профессионально? И это – на дневном отделении. Сначала поработай, и пусть отзыв дадут, что ты действительно программист.

На юге Швеции в некоторых маленьких городках, где по 30 тысяч жителей, есть свои IT-университеты. Например, пять факультетов: программирование, технологии, архитектура телекоммуникаций, экономика, юриспруденция. Отделения в Англии, в Техасе, много профессоров, и программы хорошие, я сам их смотрел. В Швеции каждый университет окружен технопарком, 200-300 мелких компаний. Государство предоставляет здание (побольше нашего матмеха). Аудитории, места для конференций, дешевые гостиницы. Получается такой симбиоз: университет готовит специалистов, которые работают в этих фирмах. Технопарку выгодно иметь ресурсы, а университету выгодно иметь финансовую поддержку и обратную связь, чему учить и как учить. Я уже давно говорю, что в нашем университетском городке тоже надо бы сделать технопарк. Пока никак не получается.

И еще одна тема – развитие собственно науки. Есть класс людей,

которые готовы работать за меньшие деньги, но при условии, что будут заниматься наукой. Это особенность психики человека, которому неинтересно делать программное обеспечение с заранее указанными свойствами в жесткие сроки. Но таких людей мало. И проблема в том, что в Петербурге есть сотни компаний, где платят приличные деньги, и программист может просто зарабатывать, но предприятия наукой заниматься не хотят.

Мы не можем все свести к работам "по заказу", даже если работа хорошо организована, поддержана мощными современными технологиями и налажены все организационные процессы. Кто будет создавать новые идеи, новые методологии, новые инструментальные средства? На мой взгляд, есть определенный процент выпускников университета, у которых в особенностях психики заложено, что они хотят сидеть в тиши лаборатории и заниматься наукой. Это люди, которые готовы получать, скажем, в два раза меньшую зарплату, чем люди на фирмах, но заниматься своей любимой наукой.

Кстати, первый раз я столкнулся с этой проблемой в Гамбурге. Это было в 1989 году. Хорошие преподаватели Гамбургского университета получают 7000 марок. Я навел справки и выяснил, что можно работать в программистской фирме и получать 20000 марок тут же, в Гамбурге. И места есть, и приглашают. Но преподаватели на такую работу не стремятся попасть. Я постарался узнать, почему. Ответ был следующим: "Во-первых, 7000 марок вполне достаточно, у меня есть дом, у меня есть дача, машина, что мне еще надо? В бочку, что ли, эти деньги складывать? А во-вторых, в программистской фирме будет жесткое управление, жесткие приказы. Я этого не люблю. Я работаю в университете. Этой зарплаты мне достаточно, чтобы прилично существовать, но я хочу заниматься именно наукой".

Я понимаю этого преподавателя. Есть такие люди. Если бы не было таких людей, то наша наука бы остановилась. Где им работать здесь, в Санкт-Петербурге? Какая фирма возьмет на себя обязанность материально поддерживать фундаментальные исследования?

Кстати, а что это такое – фундаментальные исследования? У моих прагматичных друзей-американцев есть забавный, но полезный критерий, как отличать фундаментальные исследования от

нефундаментальных. Если исследование 3 года не приносит прибыль, оно считается фундаментальным, и та часть прибыли, которая в него вложена, освобождается от налогов. И фирме это полезно. Если же, не дай Бог, через 2,5 года из этого исследования получился результат, который был применен на практике, продан, получены деньги за него – налоговый инспектор пересчитает налоги за него. Значит, исследование не было фундаментальным. Можно улыбаться, глядя на это правило, но оно существует, и оно работает. В США очень много забавных правил, но еще более забавно, что основная масса людей этим правилам следует.

Так, или похожим образом, или совсем по-другому, но можно придумать законы для поддержки фундаментальных исследований. Например, я директор предприятия. У меня есть свой директорский фонд, то есть сравнительно небольшая сумма денег, которую я отнимаю от других заказов и имею право тратить их по своему усмотрению. Практически все эти деньги я трачу на поддержку исследований.

Но те люди, которые получают деньги, чувствуют себя "людьми второго сорта", потому что время от времени я вынужден им напоминать, что мне нужны результаты, что я ради вас брал деньги у такой-то группы... Это не улучшает психологический климат. И те люди, у которых я отнял деньги, тоже не рады: "Лучше бы ты нам еще два компьютера купил, лучше бы ты нам новую мебель купил" и так далее. Всегда есть необходимость чтото купить, разумно истратив деньги для той группы, которая их заработала. У меня даже распадались группы, и одной из причин ухода сотрудников было их несогласие с моей политикой распределения денег. Они твердо настаивали на том, что они заработали деньги, поэтому эти деньги должны быть истрачены на их зарплату, в крайнем случае, на их инфраструктуру, на компьютеры. Даже, может быть, и на исследования, но только в этой узкой области, где они заработали деньги. И все мои попытки объяснить им, что так не бывает, что никто не может предсказать, какое направление в науке проявит себя через два-три года, ни к чему не приводят. Я уж не говорю о том, что понятие "они заработали" – совершенно неправильное. Зарабатывает предприятие, директор, маркетологи, завхоз, сетевые администраторы и т.д. Когда "уговаривают" заказчика, ссылаются на предыдущие работы предприятия. Заказчик с большим удовольствием работает с устоявшимся большим коллективом, а не с малой группой,

которая в любой момент может разбежаться.

Я не настаиваю на том, что мое решение – единственно правильное. Возможно, в чем-то и они были правы. Но, тем не менее, я искренне считаю, что большое предприятие, диверсифицированное, то есть имеющее много направлений работы и много направлений исследований, стабильнее и надежнее. Бывает, что некоторые направления, даже очень многообещающие, неожиданно коллапсируют. Нынешние резкие падения коэффициента высокотехнологичных производств в Америке, по-моему, как раз это и показали. Сколько было ожиданий от всех этих бесконечных интернет-приложений! А теперь идет массовое закрытие фирм, увольнение людей. Может быть, найдется 2-3 умных человека, которые скажут: "Я об этом и говорил", только я таких не знаю. Трудно предугадать, что разовьется, а что нет.

Я считаю, что мы (тем более мы, выпускники мат-меха!) должны работать в возможно более широком наборе направлений. Во-первых, это интересно, во-вторых, таковы законы науки, а в-третьих, это обеспечит определенную стабильность, если что-то все-таки по не зависящим от нас причинам или из-за нашей плохой работы не оправдает надежд. В этом случае маленький коллектив обанкротится и будет вынужден искать другие способы существования, а большие предприятия это спокойно переживут.

В конце концов, я пришел к пониманию, что поддержка науки не может быть выполнена на уровне спонсорской поддержки, на уровне отдельных пожертвований. Как и везде, нужна системность, нужна структура, нужны определенные "правила игры".

Сейчас я прилагаю много усилий для создания института, который так и будет называться: научно-исследовательский институт информационных технологий, входящий в структуру моего родного Санкт-Петербургского государственного университета. Я провел беседы со многими исследователями нашего факультета, в частности, и с такими, которые весьма далеки от практики, но продуктивно работали в нашей области и в computer science. Мы с ними наметили несколько направлений, в которых исследования особенно важны сейчас, даже без стопроцентной уверенности, что через два-три года они приведут к видимому результату, который можно будет использовать. Я получил

поддержку Ученого Совета факультета, хотя тоже не без вопросов. Вопросы были такие же, как несколько лет назад, когда создавалась новая кафедра: "Зачем нужно еще что-то новое? Нельзя ли обойтись старыми структурами?" Но, тем не менее, процесс идет.

Я надеюсь, что мне удастся этот институт создать. Более того, под этот пока не созданный институт я нашел западную компанию, которая готова материально поддержать проект. Причем я их не обманывал. Я говорил, что институт будет заниматься исследованиями и не обещал немедленной прибыли. Тем не менее, крупные компании понимают, что если поддержат исследования, то все равно получат выгоду. Если появится положительный результат, он, в первую очередь, будет применен в их интересах. Это нормальный мировой процесс хоть в Западной Европе, хоть в США. Любой ученый из любого университета до 40% своего времени тратит на поиск грантов, их обоснование. Не считается зазорным делать 3, 5, 10 попыток. В одном, в другом, в третьем фонде.

В первую очередь, нужна структура для такого поиска. Нужны специальные люди, которые следят, где объявлены гранты, где объявлены программы, какие к ним предъявляются требования. Именно эти люди должны оформлять бумаги. Сначала надо сообщить: "Коллеги! Есть такие-то предложения. Требования такие-то, сроки такие-то". Специальные люди должны оформлять и рассылать бумаги (ученые не всегда удачно их оформляют). Должна вестись база данных: на что получен положительный ответ, какие замечания были, как на них реагировать. Этим должны заниматься специально подготовленные люди, а не сами исследователи. И я думаю, что в наше время научно-исследовательский институт – это как раз та структура, которая сможет сконцентрировать внутри себя исследователей, обеспечить им определенную поддержку для того чтобы получить финансирование, отслеживать сроки и этапы исследований.

Короче говоря, все должно быть сделано для того, чтобы ученые максимум своего времени тратили на исследования, получали за это достойную зарплату, чтобы не надо было "халтурить", отвлекаться на различные работы. Мешки с картошкой и песком сейчас никто уже не грузит, но случаи, когда заслуженные ученые, профессора вынуждены заниматься неинтересной работой, далекой от науки, только для того,

чтобы получить зарплату, я знаю. Их много.

Кто знает, каким получится этот институт? Но я искренне верю, что институт с международным участием поможет тем ученым, которые предрасположены к научной деятельности, не только выжить, но и активно участвовать в развитии нашей любимой науки.

Индустриальная программа подготовки IT-кадров на базе кафедры системного программирования СПбГУ

Компания ЛАНИТ-ТЕРКОМ решает проблему нехватки кадров, ежегодно набирая студентов в так называемые "студенческие проекты", которые позволяют молодым специалистам получить бесценный опыт работы в реальных условиях, а компании – приобрести хорошо подготовленных сотрудников.

Проблема нехватки квалифицированных кадров в IT-индустрии давно стала острой и часто обсуждаемой темой. Сегодняшний программист – вчерашний студент, а то, что существующая сегодня система высшего образования не отвечает потребностям индустрии разработки программного обеспечения – факт известный и печальный. В вузах Петербурга выжило не более 5 признанных кафедр, готовящих 300-400 профессиональных программистов в год.

На протяжении последних трех лет компания ЛАНИТ-ТЕРКОМ совместно с кафедрой системного программирования математико-механического факультета СПбГУ проводит эксперимент с целью подготовки высококвалифицированных IT-специалистов. Используются самые разнообразные формы: дополнительные вечерние занятия, стипендии, стажировки, работа студентов в инвестиционных проектах, наконец, непосредственная работа на предприятии по темам, которые близки направлениям обучения на кафедре. Высокая востребованность выпускников кафедры доказывает правильность выбранного подхода. Многие сотрудники ЛАНИТ-ТЕРКОМ (основу коллектива составляют выпускники СПбГУ) преподают на кафедре, курируют студенческие проекты.

Таким образом, компания обеспечивает себе непрерывный приток кадров из числа выпускников университета. При этом молодой специалист сразу может приступать к работе без дополнительного обучения. Опыт участия в студенческих проектах на базе предприятия позволяет еще во время обучения в вузе сориентировать молодого программиста на интересующую его область IT, на актуальность того или иного направления, учит работе в команде и правилам производственного процесса.

В ЛАНИТ-ТЕРКОМ все департаменты принимают активное участие в подготовке студентов, что позволяет каждому студенту выбрать наиболее близкое ему направление. В каждом департаменте выделяются сотрудники, курирующие данную программу, все они являются выпускниками математико-механического факультета СПбГУ.

Для студентов цикл подготовки начинается на втором курсе. Обычно я провожу общее собрание для всех студентов, желающих принять участие в программе, и знакомлю их с деятельностью фирмы. Далее от каждого департамента выступают специалисты – потенциальные руководители проектов. Студенты выбирают одно из направлений деятельности компании и приступают к обучению. Обычно практическая работа предваряется вводным курсом лекций, в ходе которого студентов знакомят со специфичными для каждого направления темами. В результате этих лекций часть студентов отсеивается. Кому-то из студентов не понравилось выбранное направление, кто-то нашел работу и темы для исследований в других местах, кто-то просто ленится. Студенты хотят поскорее приступить к настоящей практической работе, за которой они собственно и приходят в компанию. Это понимают и кураторы проектов, поэтому такие лекции длятся не больше 2-3 недель. Затем в течение нескольких месяцев со студентами ведутся практические занятия, мало чем отличающиеся от обычных студенческих семинаров, но ориентированные, прежде всего, на практику.

В начале третьего курса студентам предлагаются исследовательские проекты. Никто не рассчитывает на создание продаваемых продуктов, основная цель – обучение, но часто результаты студенческих проектов являются хорошими прототипами будущих реальных проектов. Как правило, в проекте участвуют 10-12 студентов, из них 5-6 вносят основной вклад в проект. В течение семестра каждый департамент ведет 2-3 проекта в зависимости от потребностей предприятия и активности научных руководителей. Руководители – молодые энергичные специалисты, желающие получить опыт управления. Проекты длятся от 1 до 2-х семестров, в результате студенты получают опыт промышленной разработки, рекомендации на кафедру системного программирования, а также возможность работать в ЛАНИТ-ТЕРКОМ.

Департаментам предоставляется определенная свобода в разработке

программ обучения и стратегии привлечения студентов. Поскольку студентов на отделении ограниченное количество, между департаментами существует конкуренция, и каждый департамент имеет свои уникальные приемы работы со студентами и опыт привлечения их именно на свои направления.

Так, для департамента аутсорсинга это новейшие технологии, что, в первую очередь, обусловлено спецификой работы самого департамента. Особое внимание уделяется изучению именно тех технологий, которые планируется использовать в реализации проектов. Практикуются семинары внутри департамента, где сотрудники фирмы или студенты, участвующие в проектах, выступают с докладами, рассказывая коллегам и студентам о перспективных направлениях в отрасли. Как правило, студенческие проекты этого департамента находят применение внутри компании. Примером является ряд проектов по созданию web-интерфейсов: от проектирования web-интерфейса и обработки недельных отчетов сотрудников до "Системы с web-интерфейсом для подведения результатов турниров по настольному теннису", реализованных в разных популярных сегодня технологиях. В департаменте создана и совершенствуется база данных результатов всех студенческих проектов, система учета выведена на автоматический уровень, что позволяет руководителям проектов более четко планировать свою работу.

В департаменте реинжиниринга студентов привлекает перспектива работы со стабильными крупными зарубежными заказчиками, что является весомым аргументом для молодых специалистов. Многие студенты, участвующие в программе подготовки IT-специалистов, выбирают именно этот департамент. В качестве заданий здесь используются несложные части реальных проектов. Студенты занимаются разработкой и тестированием синтаксических анализаторов, средств генерации, анализа графов, оптимизаторов и т.п. По окончании работы студенты устраивают презентацию полученного продукта. Руководители проекта считают, что такая схема подготовки новых кадров для работы в департаменте весьма удачна. В результате предыдущего цикла подготовки приняли активное участие и успешно завершили проект 11 человек, из которых 7 остались работать в департаменте реинжиниринга.

В департаменте электронных систем в качестве задания студенты также работают с частью реального проекта или типичной небольшой задачей. Примером может служить проект "Аппаратное моделирование архитектур цифровых фильтров", работая над которым, студенты занимались VHDL-программированием специализированного процессора. Результаты этого проекта в дальнейшем использовались в работе департамента. Количество студентов, которых этот департамент может привлечь к работе в студенческих проектах, в значительной мере ограничено необходимостью определенного количества специального оборудования и приборов.

Специфической особенностью работы в департаменте телекоммуникаций является большой объем знаний по этой отрасли, не входящих в университетскую программу, что отпугивает часть студентов от данного направления, но зато привлекает других. Важно подчеркнуть, что нас интересуют "программистские" аспекты телекоммуникаций, но мы ни в коей мере не конкурируем с отраслевыми институтами.

В качестве основного проекта в этом году планируется развитие продукта ЛАНИТ-ТЕРКОМ – универсальной телефонной станции "Юнивер" с функциями многоканального радиоудлиннителя абонентских телефонных линий. В рамках студенческого проекта планируется реализация новых функций этой станции. Ведутся также такие интересные студенческие проекты, как: "Реализация протокола VXML" (оказание голосовых услуг через Internet), "Синтез речи", "Сервер компьютерной и IP-телефонии".

Прежде чем студенты приступят к практическим занятиям, им предстоит пройти значительный курс дополнительного обучения, в который входят: SDL/MSC (для понимания алгоритмов), RTST/REAL (эти технологии разработки ПО реального времени созданы в ЛАНИТ-ТЕРКОМ), курсы технологии сборки и отладки ФПО.

К сожалению, даже такое крупное предприятие, как ЛАНИТ-ТЕРКОМ, штат которого насчитывает 300 человек, не может обеспечить отдельного руководителя (тьютора) для каждого студента. Но даже с группой из 4-5 студентов практическая работа несравненно эффективней классического университетского подхода. Студенты

обучаются не только научным и техническим приемам, но и умению работать в группе, планировать работу, отвечать за сроки, результаты и бюджет. По итогам первых лет работы можно уверенно сказать, что такой симбиоз университетского и практического образования позволяет готовить действительно сильных специалистов. Однако возникли и проблемы:

1. Вечерняя работа преподавателей и сотрудников оплачивается предприятием, причем по достаточно высоким ставкам. Для студентов же это обучение бесплатно. Необходимо иметь возможность заключения определенного контракта с каждым обучаемым, в соответствии с которым получивший дополнительное образование студент был бы обязан отработать, скажем, 3 года на предприятии, которое финансировало его образование. На данном этапе неясно, как гарантируются такие контракты Российским законодательством. Иначе получается, что мы готовим элитные кадры для работы на западные компании, представленные в России, т.к. они имеют возможность платить гораздо больше (в Санкт-Петербурге – это Intel, Sun, HP, Motorola, Alcatel и т.п.). У нас есть некоторые неэкономические средства – аспирантура, отсрочка от армии, возможность заниматься наукой, поэтому процентов 70 обученных нами студентов остается работать на нашем предприятии, но понятно, что нужны и более прямые экономические рычаги.
2. В настоящее время многие толковые студенты приезжают с периферии России и из стран бывшего СССР. Хотелось бы иметь возможность оставить их работать без лишних бюрократических проблем.
3. Часть студентов – из малообеспеченных семей, и вместо того, чтобы совершенствовать свои профессиональные знания, они работают разнорабочими и т.п., в особенности на младших курсах. Многим из них мы предоставляем стипендии, но, естественно, наше предприятие не может заменить государство.
4. В таких вопросах как разработка новых программ обучения, подготовка преподавателей для других вузов, написание учебников и т.п., хотелось бы видеть более активное участие государства, например, в виде адресных программ или целевых грантов.

На наш взгляд, накопленный нами опыт вполне может использоваться и другими IT-предприятиями, имеющими тесные связи с университетами. Во всяком случае, мы всегда готовы поделиться методиками, примерами решаемых задач и любой другой информацией. Мы понимаем, что предлагаемый нами подход не может служить глобальным решением проблемы – на все IT-предприятия университетов не хватит. Мы уже начали прорабатывать вопрос расширения масштабов эксперимента путем создания технопарка в Петергофе вокруг естественных факультетов СПбГУ. На данный момент с кафедрой системного программирования взаимодействует 5-7 IT-предприятий, руководители которых когда-то закончили мат-мех под моим научным руководством. Кафедра предоставляет предприятиям возможность встреч со студентами, принимает в качестве курсовых и дипломных проектов работы, выполненные под руководством сотрудников этих предприятий. Сотрудники предприятий ведут занятия не только со "своими" студентами, но и с другими студентами кафедры, предоставляют свою технику, а самое главное, свои интересные задачи и опыт их решения.

В настоящее время планируется строительство большого IT-технопарка в непосредственной близости от мат-меха в Петергофе. Имеется договоренность с университетом о предоставлении инженерно обустроенной территории и с потенциальным инвестором. Мы стараемся держаться вне жарких споров различных министерств. Разумеется, предоставление налоговых скидок ускорит процесс строительства нашего и других технопарков. Мы пока не ставим вопрос строительства технопарка в зависимость от получения каких-то преференций, надеемся, что практически бесконечный источник квалифицированных кадров, возможность развития и использования новейших научных достижений в IT являются достаточными аргументами как для большинства предприятий, работающих в области IT, так и для инвесторов, ну а польза для университета представляется совершенно бесспорной.

Наукоград и технопарки в Петергофе

Много лет назад было замечено, что концентрация многих высокотехнологичных предприятий, НИИ различного профиля и исследовательских лабораторий на сравнительно небольшой территории дает большой эффект. Развитие науки, высоких технологий требует наличия большого количества ученых, различных специалистов в одном месте, определенной "критической массы" знаний и умений. Полностью этот феномен объяснить достаточно трудно, хотя основные идеи понятны: научное общение, доведение идей до решений и продуктов, воспитание молодежи на "живых" примерах и т.д. Именно поэтому в СССР были созданы академгородки на Урале, в Сибири, на Дальнем востоке. В США классическим примером является знаменитая Силиконовая долина. Сразу заметим, что все известные центры такого рода создавались вокруг крупных университетов, причем с обоюдной пользой – как для университета, так и для предприятий и исследовательских центров.

В современной России законодательно определено понятие "Наукоград". Если на территории муниципального образования валовый доход научных предприятий превышает половину дохода всего муниципального образования или количество научных работников составляет более 25% от всех работающих, то такое образование может быть объявлено "наукоградом". (На самом деле требований гораздо больше, но основная идея понятна). Наукограды имеют определенные привилегии (например, до половины федеральных налогов можно тратить прямо на месте), формируется специальный госзаказ, короче, многое делается для развития как территории в целом, так и отдельных предприятий.

Кроме академгородков и наукоградов, в мире активно развиваются технопарки. В принципе, большой разницы между этими понятиями нет. Технопарк часто ориентирован на одно научное направление, например, информационные технологии (ИТ) или биотехнологии, а наукоград обычно носит более комплексный характер; технопарк – это отдельное учреждение, а наукоград – это целое муниципальное образование. Хотя бывают и многоцелевые технопарки (например, в Финляндии), часто технопарки тоже строят жилье для своих сотрудников, постепенно превращаясь в целые городки.

В нашем случае будем для определенности считать, что Наукоград – это муниципальное образование Петергоф, а в составе Наукограда строятся технопарки с различными специализациями – ИТ, биотехнологии, нанотехнологии, экологии и т.д. Наукоград создается и управляется дирекцией, которая планирует бюджет и контролирует его исполнение. Стратегические цели Наукограда формулируются общественными советами. Одной из первоочередных задач дирекции является строительство и поддержание инфраструктуры (дороги, связь, транспорт, экологическая обстановка, досуг и т.п.). Развитие каждого технопарка или отдельного предприятия – задача их местного руководства, но дирекция Наукограда обязана координировать их деятельность и всемерно способствовать междисциплинарной кооперации.

Так что же такое "Технопарк"? Представьте себе небольшое (10-15 сотрудников) предприятие. Такому предприятию трудно содержать квалифицированного системного администратора локальной сети и сервера базы данных, юриста и т.д. На Западе в этой ситуации применяется outsourcing – использование ресурсов других организаций. Даже бухгалтерию для маленьких предприятий ведут крупные специализированные организации. У нас в России классический аутсорсинг не слишком развит, более того, это понятие "оккупировано" предприятиями, выполняющими заказы по программированию иностранных компаний. Тоже, разумеется, аутсорсинг, но только в одном из возможных смыслов. Технопарки призваны вернуть понятию "аутсорсинг" первоначальный смысл, т.е. брать на себя заботы многих организаций, работающих в технопарке.

Прежде всего, технопарк должен иметь большой дом или несколько домов.

Очень хорошо, если технопарк может предоставлять и жилье по выгодным для сотрудников ипотечным схемам; программисты обычно хорошо зарабатывают, поэтому оформление ипотеки может быть упрощено.

Должна быть обеспечена охрана, телефоны, Internet. ИТ-технопарк должен обеспечивать серверы, локальные сети, брандмауэры, хороших сисадминов. Один мой знакомый бизнесмен образно охарактеризовал

ИТ-технопарк так: "все помещения там должны иметь фальшпол". Он имел в виду просто легкость реконструирования локальных сетей, но в одной фразе выразил необходимость "заточенности" помещений именно на IT-индустрию.

Однако сдача в аренду площадей не является единственной деятельностью технопарка. Технопарк должен иметь мощную юридическую службу (иностранные контракты, трудовые соглашения, вопросы интеллектуальной собственности и т.п.). Большую пользу и прибыль может принести специальная служба по организации школ, выставок, конференций. Технопарк должен иметь большие и малые конференц-залы с соответствующим оборудованием. Очень важно иметь спортзал и открытые спортивные площадки. Нельзя забывать о буфетах, кафе и об организации досуга.

Многие технопарки мира имеют в своем составе венчурные фирмы, поддерживающие новые оригинальные идеи и помогающие довести их до продаваемого продукта. Неотъемлемой частью любого технопарка являются службы, занимающиеся поиском и переподготовкой кадров, сертифицированным и авторизованным обучением, сертификацией, поставками компьютеров, сетевого оборудования, расходных материалов и т.п.

Разумеется, перечисленный список предоставляемых услуг не является исчерпывающим. Администрация технопарка должна быть достаточно оперативной и гибкой: возникла какая-то новая потребность (например, обслуживание бухгалтерии, финансовый аудит, IP-телефония, call-центры) – нужно оперативно предоставить и эти услуги.

Технопарк, предоставляющий весь комплекс услуг (а не только аренду помещений), является прибыльным предприятием. В этом заключается интерес инвесторов, которые вкладывают деньги в строительство технопарка.

Предприятия "кучкуются" в технопарке, так как жизнь сильно упрощается – не нужно за каждой отдельной услугой бегать по всему городу. Однако главное, зачем предприятия (особенно молодые) стремятся в технопарки – это кадры. Обычно технопарки располагаются в непосредственной близости от университетов. Если между технопарком и университетом расстояние, которое можно преодолеть за

10-15 минут пешком, сотрудников промышленных предприятий можно привлечь к преподаванию в университете. Особенно важна такая помощь в организации практики, руководстве курсовыми и дипломными работами.

Многолетний опыт показывает, что если студент, начиная с третьего курса, пишет курсовые, работает, получая приличную зарплату, наконец, пишет дипломную работу на каком-то предприятии, то после окончании университета он с большой вероятностью останется работать на этом предприятии. Такие сотрудники с существенно большей вероятностью закрепляются на предприятии, чем сотрудники, пришедшие "с улицы". Не будем забывать, что даже для высокопрофессиональных специалистов со стороны требуется время для "доводки".

Не менее важной, чем приток кадров, причиной для многих IT-предприятий, работающих в технопарке, является возможность "держать руку на пульсе" новейших научных разработок, результатов апробации новых технологий, возможность заказать необходимое исследование, причем именно тем кафедрам и профессорам, которые хорошо известны предприятиям.

Национальные черты производства ПО

Понять свою уникальность можно только сравнивая себя с кем-то. Раньше нас учили, что отечественное – значит отличное. Мы видели, что сильно отличаемся от американцев, а про себя отмечали, что порой не в лучшую сторону...

Я хорошо помню, как начиналось сотрудничество с американцами в 1992 году. С самого начала они признавали наше преимущество в созидательности (модном сегодня понятии "creativity"), в уровне образования, общей математической культуре. Сразу же обнаружились и наши основные проблемы: плохо работающая почта (электронной почты тогда еще не было), резкое несоответствие технического оснащения у нас и в Америке, недоступность новых технологий и даже их описаний. Первый в нашей организации факс мне привезли итальянцы в 1993 году. Специальную рулонную бумагу к нему в России приобрести было очень трудно.

Мы осознавали, чему нам стоит поучиться у западных коллег. Что такое планирование, мы, разумеется, знали. Но были искренне убеждены(и обстоятельства того времени располагали к такому мнению) что детальное планирование на год – это шарлатанство. Сдвиги сроков исполнения квартального заказа на 1-2 недели были нормой. Понятия планирования рисков для нас не существовало. Пользу регулярных недельных отчетов мы осознали только через несколько лет. До этого мы к ним относились как к блажи заказчика, неизбежной в процедуре получения от него вознаграждения. То же можно сказать и о базе данных ошибок, где из года в год накапливаются все серьезные и мелкие недочеты. Именно американцы объяснили мне, что если какой-то сотрудник сделал много ошибок в течение недели, это еще ничего не значит. Такая нестабильность может быть обусловлена проблемами личного характера. Но если один программист в течение 5-10 недель допускает в 5 раз больше ошибок, чем другой, то по этому факту уже можно судить о способностях каждого из них при наличии достоверной статистики. Так, основной проблемой стало наше воспитание, склад ума. Нашему программисту ничего не стоило забыть ответить на запрос в тот же день. Более того, когда американцы устраивали по этому поводу скандал, я про себя думал: "чего прицепились...". Все эти недочеты мы исправляли, руководствуясь опытом западных коллег.

Постепенно нам удалось наладить процесс производства ПО в точности так, как это принято у американцев. Но вместе с тем стала очевидна и обратная сторона медали. Современному молодому сотруднику с двухлетним опытом работы в жесткой производственной структуре начинает казаться, что, кроме выполнения правил игры, от него ничего не требуется. Редко кто из молодежи проявляет инициативу, предлагает варианты рационализации процесса, хотя это было естественно для молодого поколения 30 лет назад. И уж никак нельзя опустить явный меркантилизм: очень часто предложение о небольшом повышении зарплаты является для молодых сотрудников достаточным поводом для того, чтобы сменить работу. Аргументом служит: "А что, в Америке принято раз в год менять работу и переезжать из штата в штат, почему у нас должно быть по-другому?" С другой стороны, такой борьбе за "рубль подлиннее" способствовала и общая нищета в стране. Отсутствие государственной поддержки индустрии в целом спровоцировало ситуацию, когда наши элитные вузы готовили специалистов высшей квалификации для приходящих на наш рынок западных компаний, способных предложить достойный уровень заработной платы. Российским производителям ПО при этом оставалось довольствоваться тем, что осталось, хотя немалый вклад в подготовку будущих специалистов внесли именно они.

На мой взгляд, корень проблемы – именно в слепом следовании американскому опыту. Если мы выберем этот путь – мы всегда гарантированно будем отставать в организации производства ПО на 10-15 лет. Нужно стараться выявить наши преимущества, четко их сформулировать, развивать коллективизм в работе, сделав основную ставку на образование и способность самостоятельно созидательно мыслить.

В Америке я много раз сталкивался с ситуацией, когда разработчик, сидящий за соседним столом, не хочет потратить 5 минут, чтобы объяснить тебе что-то. Там это называется job security. Сегодня я тебе что-то объясню, завтра ты станешь лучше, чем я, а послезавтра меня выгонят. Хочется верить, что эта американская черта не приживется в нашей действительности.

Я много лет играл в баскетбол. Тренеры учат, что основное внимание надо уделять шлифовке своих сильных сторон, а не тратить время на

освоение того, что ты не выучил в детстве. Я думаю, что эта мысль применима и в нашем случае. Неистовое стремление "стать похожими" только отдалит нас от возможности быть уникальными. Пытаясь равняться на индийские software house или довольно безликие американские программистские организации, мы потеряем свое лицо и потенциальную возможность занять достойное место в мировой индустрии ПО.

Говорят коллеги. Игорь Агамирзян:

Профессор Андрей Николаевич Терехов – удивительный человек, занимающий совершенно уникальное место на российском рынке информационных технологий. Судьба свела меня с ним 30 лет назад, когда я учился на математико-механическом факультете ЛГУ (Санкт-Петербургского, а в то время Ленинградского – государственного университета), а Андрей Николаевич (совсем молодой, недавний выпускник мат-меха и даже еще не кандидат наук) возглавлял небольшую группу в лаборатории системного программирования, где в это время шла работа над транслятором с Алгола-68 для ЕС ЭВМ. Уже тогда я почувствовал его отличие от других наших преподавателей (а преподавательский состав на мат-мехе того времени был очень сильным) – Терехов был твердо уверен, в том, что он программист-практик, и учил студентов (с которыми любил работать, и аккуратно отбирал лучших) не теории, а практическим навыкам и приемам. При этом именно под его влиянием теория, обильно вливаемая в наши головы на лекциях, оседала и кристаллизовывалась. Очень часто у меня складывалось впечатление, что это происходило только с нами, с той небольшой группой, которая работала с Тереховым – у многих других наших соучеников теория так и оставалась в "жидком виде".

Парадокс, однако, заключается в том, что на самом деле Андрей Николаевич всегда был скорее теоретиком и методологом, нежели настоящим практиком. Несмотря на то, что на протяжении всей своей творческой биографии Терехов работал в сугубо практических проектах, в советское время ориентированных в первую очередь на оборонную промышленность и программирование систем связи, а в пост-советское – на коммерческие программные продукты, он всегда привносил в них глубокий теоретический и методический базис. И зачастую оказывалось, что формируемая методика (почти никогда не формализованная, а существующая в виде историй, баек и анекдотов) имела большую ценность, чем результат проекта сам по себе.

Сегодня трудно говорить о программировании как науке. С другой стороны, проводимый многими с легкой руки Дональда Кнута подход к программированию как к искусству, тоже себя изжил. За последние 30 лет программирование прошло через стадию ремесла, и в последнее десятилетие в крупных софтверных компаниях стало выходить на

уровень фабричного производства. И именно на этом этапе теоретико-методологический подход к созданию программного продукта оказывается необыкновенно востребованным.

За плечами Андрея Николаевича Терехова – не только ряд успешных (а иногда и неуспешных) проектов, не только 30 лет преподавательской работы в университете и опыт руководства исследовательской лабораторией и кафедрой, но и сформировавшаяся за эти годы уникальная научно-практическая школа программирования, успешность которой подтверждается и выполненными проектами, и победами студенческих команд в олимпиадах по программированию, и просто теми компаниями, в которых работают ученики Терехова. В каком-то смысле предлагаемая вниманию читателя книга как раз и отражает дух этой школы – прагматичный, не очень формальный, со ставкой на сочетание индивидуального блеска и эффективной командной работы. А вдумчивый читатель увидит за интересными историями глубокие методологические принципы - как в свое время мы, первые ученики Андрея Николаевича, за байками и приемами находили те самые неочевидные связи между теоретическими результатами и потребностями практики.

Игорь Агамирзян, Директор по стратегии Microsoft в России и СНГ

Говорят коллеги. Валентин Макаров:

Дорогие читатели, перед вами книга, написанная замечательным человеком, сумевшим совместить в себе талант преподавателя, ученого, промышленного разработчика программного обеспечения и бизнесмена. И в каждой сфере своей деятельности Андрей Николаевич достигал невиданных успехов. Заведующий кафедрой на знаменитом мат-мехе Петербургского Государственного Университета, директор созданного им самим Института Информационных Технологий СПбГУ, создатель и главный архитектор целой серии блестящих аппаратных и программных комплексов и генеральный директор одной из самых известных петербургских компаний разработчиков программного обеспечения – все это один человек – Андрей Терехов. Если прибавить к этому поразительные успехи, достигнутые его детьми – Андреем и Кариной – в той же сфере информационных технологий, то можно прямо признать, что перед вами – легендарная фигура российской индустрии разработки программного обеспечения, настоящий классик и верный кандидат на звание "национального достояния".

Андрей Николаевич начал свою карьеру в советское время и был из тех, кто участвовал в формировании самого понятия "советской вычислительной техники". Затем в труднейшие годы перестройки он сумел создать при Университете жизнеспособную структуру для промышленного программирования, которая буквально спасла от отъезда с Родины в никуда сотни блестящих российских программистов, дав им возможность зарабатывать своей профессией, оттачивать знания и применять их для решения сложнейших математических задач.

Затем в конце 1990-х он сделал, пожалуй, самое главное дело для становления российской индустрии разработчиков программного обеспечения, отдав много сил и энергии созданию и взрослению Консорциума "Форт Росс". Его вклад в наше обще дело был оценен коллегами, избравшими его Председателем Правления общероссийской организации программистских компаний, в которую превратился "Форт Росс" – Некоммерческого Партнерства РУССОФТ.

Для чего я перечисляю заслуги Андрея Николаевича? Для того чтобы вы, любезные читатели, правильно подошли к прочтению этой книги. В ней вы найдете фантастическое соединение академического подхода к

построению изложения, выстраданный опыт организации процесса разработки программного обеспечения, аналитический научный подход к выявлению проблем и поиску их решений – и замечательный юмор, позволяющий читать книгу как увлекательную повесть, усваивая материал даже на подсознательном уровне.

Читайте эту книгу, работайте над собой, старайтесь взять на вооружение идеи автора и – идите дальше, ставьте новые задачи и ищите их решения. Не боги горшки обжигают. Предлагаемая вам книга создает уверенность в силах человека, крепко стоящего на своей земле и опирающегося на свою школу. И это, пожалуй, еще одно из замечательных качеств и предназначений этой книги.

Валентин Макаров, Президент Некоммерческого Партнерства разработчиков программного обеспечения РУССОФТ

Список литературы

1. Дж. Фокс, Программное обеспечение и его разработка, М.: Мир, 1982

2. Г.С. Цейтина, Алгол 68. Методы реализации, Л.: Изд. ЛГУ, 1976, 224 с

3. И.В. Вельбицкий, В.Н. Ходаковский, Л.И. Шолмов, Технологический комплекс производства программ на машинах ЕС ЭВМ и БЭСМ-6, М.: Статистика, 1980, 263 с

4. И. Соммервиль, Инженерия программного обеспечения, М.: Изд. Вильямс, 2002

5. Ф. Брукс, Мифический человеко-месяц или Как создаются программные системы, СПб.: Изд. Символ-плюс, 2000

6. Э. Дейкстра, Дисциплина программирования, М.: 1982

7. Обратный метод установления выводимости непрефиксных формул в исчислениях предикатов, Докл. Акад. Наук СССР 172, 1967,с. 22-25

8. М. Кантор, Управление программными проектами. Практическое руководство по разработке успешного программного обеспечения, М.: Вильямс, 2002

9. В.В. Липаев, Тестирование программ, М.: Радио и связь, 1986, 296 с

10. С.Н. Баранов, А.Н. Терехов, Г.С. Цейтин, Инструкции к программе DICO. Методические материалы по программному обеспечению ЭВМ, Серия 4. Выпуск 5. Изд. ЛГУ, 1974

11. Maslow A.A, Motivation and Personality, New York: Harper and Row, 1954

12. Сайт Microsoft Solutions Framework. Методология создания программных решений -- Книга Анализ требований и создание архитектуры решений на основе Microsoft .NET. Учебный курс MCSD, URL: http://www.microsoft.com/rus/msdn/msf/default.mspx , Русская редакция, 2004, 416 с

13. ГОСТ 19.001-77. Единая система программной документации. Общие положения

14. The Free Software Foundation, URL: http://www.fsf.org/

15. А.Н. Терехов, Л.А. Эрлих, А.А. Терехов, История и архитектура проекта RescueWare. Автоматизированный реинжиниринг программ, СПб.: Изд-во С.-Петербургского университета, 2000, с. 7-19, Под ред. А.Н. Терехова, А.А. Терехова

16. М.А. Бульонков, Д.Е. Бабурин, HyperCode – открытая система визуализации программ. Автоматизированный реинжиниринг программ, СПб.: Изд. С.-Петербургского университета, 2000, с. 165-183, Под ред. А.Н. Терехова, А.А. Терехова

17. Р. Фатрелл, Д. Шафер, Л. Шафер, Управление программными проектами: достижение оптимального качества при минимуме затрат, М., Вильямс, 2003

18. Международный стандарт ISO 9001-94. Системы качества. Модель обеспечения качества при проектировании, разработке, производстве, монтаже и обслуживании, М.: ИПК, Изд. стандартов, 1996, 19 с

19. M.C. Paulk, C.V. Weber, B. Curtis, M.B. Chrissis et al, The Capability Maturity Model: Guidelines for Improving the Software Process, Addison-Wesley, 1995

20. А.М. Вендров, CASE-технологии. Современные методы и средства проектирования информационных систем, М.: Финансы и статистика, 1998. – 176 с.: илл

21. E.F. Codd, A Relational Model of Data for Large Shared Data Banks. Communications of the ACM, Vol. 13, No. 6, June 1970, pp. 377-387. Copyright ” 1970, Association for Computing Machinery, Inc

22. OMG Unified Modeling Language Specification, Version 1.5, March 2003 formal/03-03-01//www.omg.org

23. В.Е. Карпов, К.А. Коньков, Основы операционных систем. Курс лекций. Учебное

пособие, М.: Интеренет-Университет Информационных Технологий, 2004, 628 с

24. Edsger W. Dijksta, Cooperating sequential processes, Programming Languages: NATO Advanced Study Institute. Academic Press, 1968. P. 43—112

25. C.A.R. Hoare, Monitors: an operating system structuring concept, Communications of the ACM, Volume 17 Issue 10. October 1974

26. М.Т. Хиллс, С. Кано, Программирование для электронных систем коммутации, М.: Связь, 1980, с. 248

27. RTST – технология программирования встроенных систем реального времени, СПб.; Изд. СПбГУ, 1998, (Сб. Записки семинара кафедры системного программирования CASE-средства RTST++. Вып. 1)

28. CCITT Recommendation Z.100: CCITT Specification and Description Language (SDL), COM X-R 26, ITU General Secretariat, Geneva, 1992

29. В.В. Парфенов, Объектно-ориентированный подход в проектировании программного обеспечения встроенных систем реального времени, Проблемы теоретического и экспериментального программирования. Новосибирск. 1992

30. CORBA 3.0, OMG IDL Syntax and Semantics chapter, URL: http://www.omg.org/cgi-bin/apps/doc?formal/02-06-39.pdf

31. ITU Recommendation Z.120: Message Sequence Chart, 11/1999. P. 138

32. А.Терехов, А. Иванов, Дм. Кознов, А. Лебедев, Т. Мурашова,В. Парфенов, Объектно-ориентированное расширение технологии RTST, СПб.; Изд. СПбГУ, 1998, Сб. Записки семинара кафедры системного программирования CASE-средства RTST++. Вып. 1

33. А.Н. Терехов, К.Ю. Романовский, Дм.В. Кознов, П.С. Долгов,А.Н. Иванов, REAL: методология и CASE-средство для разработки систем реального времени и информационных систем, Программирование, 1999, № 5. с. 44-52

34. N. Mansurov, Automatic synthesis of SDL from MSC and its application in forward and reverse engineering, Computer Languages. Vol. 27. N 1/3.2001. P. 115-136

35. В.В. Соколов, Проверка соответствия SDL-диаграмм MSC-документации при имеющихся отличиях. Системное программирование, Спб., 2004, с. 366-390, Сб. статей/ Под. ред. А.Н. Терехова иД.Ю. Булычева

36. А.Н. Иванов, Технологическое решение REAL-IT: создание информационных систем на основе визуального моделирования. Системное программирование, Спб., 2004, с. 89-100, Сб. статей/ Под. ред. А.Н. Терехова и Д.Ю. Булычева

37. С.А. Стригун, А.Н. Иванов, Д.И. Соболев, Технология REAL для создания информационных систем и ее применение на примере системы Картотека., Математические модели и информационные технологии в менеджменте. Выпуск 2. – СПб: Изд. СПбГУ, 2004. с 120-139, Под ред. проф. Казанцева А.К. и доц. Должикова В.В

Содержание