Эдсгер Дейкстра

Избранные статьи

Перевод выполнил Alf (alf63@list.ru).
Первоисточник: http://shelek.com/
Публикуется на OberonCore.ru
с любезного разрешения переводчика.

О переводах или От редактора перевода.

Уважаемые читатели!

Представляю на ваш суд серию переводов статей одного из величайших ученых и мыслителей в области информатики – Эдсгера Вайба Дейкстры. Думаю, у программистов со стажем не возникнет вопросов, зачем и кому это нужно. Но среди читателей наверняка найдутся новички, которым это имя, к сожалению, ничего не говорит. Поэтому я и решил написать несколько слов о том, что заставило меня приступить сначала к переводу, а затем, когда к этой работе присоединились единомышленники, то и к редактированию переводов его статей.

Почему именно Дейкстра

Как-то само собой так сложилось, что в России Дейкстру обходили и продолжали обходить вниманием.

Впервые я познакомился с его трудами много лет назад, благодаря моему университетскому научному руководителю Владиславу Вениаминовичу Кривицкому, которому я столь многим обязан в своем профессиональном становлении. Он доставал какими-то неведомыми мне путями (учитывая, что слова «персональный» и «компьютер» в то время были взаимоисключающими, а об Интернете никто и слыхом не слыхивал) распечатки наиболее интересных статей из зарубежных (столь труднодоступных тогда) источников. Сделаны они были на барабанном принтере кривыми волнистыми строками и были излохмачены и затерты до дыр многочисленными читателями. Некоторые были кем-то переведены на русский язык, другие оставались на английском и тем самым становились недоступными для меня (к сожалению, в школе и в Университете я изучал французский и тем самым был лишен возможности чтения англоязычной литературы). Названия некоторых из этих статей я помню до сих пор: «Смиренный программист», «Как быть, если правда колет глаза», «Почему программное обеспечение такое дорогое»…

Отечественное книгоиздание почему-то не баловало нас книгами Дейкстры. Из огромного множества его работ переведены и изданы в тогдашнем СССР были буквально считанные единицы, да и те достать было весьма трудно. Издавали Кнута, Вирта, Ульмана, Цикритзиса… Дейкстру упорно игнорировали.

Наступили другие времена, но положение существенно не изменилось. Посмотрим на полки в отделе компьютерной литературы книжного магазина: «HTML для чайников», «Освой C++ за 5 минут»… Концептуальных книг не прибавилось.

Окончательное решение взяться за самостоятельный перевод возникло, когда мне попалась статья "Ремесленник или ученый?". После ее прочтения захотелось собрать у себя побольше столь же мудрых статей, чтобы было что почитать между изучением весьма полезных, но столь же и скучных справочных мануалов по .NET или MFC.

Я был просто поражен тем, что поиск в Сети почти не принес результатов. Нашлись «Как быть, если правда колет глаза» и «Притча», добросовестно передираемые друг у друга неутомимыми создателями «хоумпагов» и используемые для заполнения скудного раздела «Юмор» (видимо, они решили, что Дейкстра – писатель-юморист сродни Марку Твену или Джерому). Нашлась та самая статья в «Компьютерре». Подключились другие форумчане, нашли еще с пяток крошечных статей. И все! За прошедшие годы утеряно даже то немногое, что передавалось друг другу на 1200-футовых бобинах магнитной ленты…

Единственный выход – браться за дело самим. Что после некоторой раскачки и было сделано.

К счастью, оказалось, что университет штата Техас бережно хранит наследие своего почетного профессора. Бумаги, которые остались после смерти Дейкстры в 2002 г., были отсканированы и опубликованы на сайте университета. Осталось только выбрать наиболее интересные из них, перевести и опубликовать на сайте Грома.

Этот труд, как оказалось, вовсе не тягостен. Во-первых, в процессе перевода приходится более тщательно следить за мыслью автора, чем при обычном чтении, так как перевод потом попадает на всеобщее обозрение, и просто стыдно допустить в нем серьезные промахи. Поэтому в голове остается значительно больше, и ради этого стоит потрудиться.

Во-вторых, многие статьи написаны Дейкстрой от руки. Человек, знающий о компьютерах все и сделавший для них, возможно, больше, чем кто-то иной, по старинке излагал свои мысли пером на бумаге, как много лет назад. Читая эти строки, выведенные аккуратным почерком почти без единой помарки, можно как бы прикоснуться к той эпохе, когда они были написаны… Наверное, то же самое испытывает музыкант, которому посчастливилось взять в руки настоящую скрипку Страдивари.

О стиле перевода

Разумеется, по возможности я старался придерживаться дословного перевода, не отступая от авторского текста. Тем не менее не всегда это удавалось.

Дело в том, что Дейкстра – голландец по происхождению, а его родной язык сродни немецкому. Немцы же известны своим пристрастием к построению сложнейших грамматических конструкций. Нередки случаи, когда одно предложение немецкого текста занимает целый параграф, который, в свою очередь, порой не умещается на странице. Когда мне попадались подобные фразы, я разбивал их на несколько фраз попроще (согласно рецептам самого Дейкстры по написанию текстов программ, которым он, к сожалению, не всегда следовал при написании текстов на английском). Так что дотошный читатель, сравнив оригинал и перевод, может найти некоторое несоответствие в построении предложений. Я позволял себе такие отступления при условии, что смысл повествования в целом никоим образом не будет искажен.

Кому и зачем это нужно сегодня

Многие, наверное, зададут этот вопрос. И неудивительно. Ведь уже не одно десятилетие действует закон Мура, согласно которому количество транзисторов на кристалле удваивается каждые полтора-два года, а следом за ним – и производительность процессоров на их основе. Вычислительная мощность, сконцентрированная сегодня в системном блоке персонального компьютера на рабочем столе секретарши, многократно перекрывает мощность крупного вычислительного центра 70-х годов XX века. Давно забыты перфокарты и магнитные барабаны. Программирование тоже прошло огромный путь от автокодов до сложнейших и в то же время весьма облегчающих жизнь программиста систем вроде Visual Studio .NET.

Казалось бы, чему может научить нас человек, большинство работ которого написаны 20-30 лет назад и более? Ведь изменилось все: оборудование, языки программирования, операционные системы…

Оказывается, как и в любой науке, в информатике есть истины, которые, будучи открыты, сохраняются неограниченно долго. И Дейкстра был один из немногих, кто подошел к новой дисциплине – программированию – не как ремесленник и даже не как инженер, а как настоящий ученый, виртуозно владеющий математическим аппаратом (сказалась подготовка физика-теоретика) и системным подходом.

Чему же учил нас Эдсгер Дейкстра? Прежде всего, он настаивал на том, что программист-профессионал обязан владеть математическим инструментарием. Особенно это относится к математической логике, поскольку помимо умения писать программы, нужно также при необходимости уметь и доказать их правильность. Разумеется, и сам он был в этом образцом, его математической эрудиции можно только позавидовать.

Почти через все статьи красной нитью проходит мысль о том, что программы должны быть не только (и не столько) эффективными, сколько изящными, по возможности краткими и легко читаемыми. Выражая свое мнение о том, каким быть будущим языкам программирования, он утверждал, что элегантность и выразительность языка не менее важна, чем возможность его эффективной реализации на существующем вычислительном оборудовании.

Наконец, Дейкстра всегда находил мужество говорить об острых и неудобных проблемах информатики, в то время как другие делали вид, что таковых не существует в природе. Его не пугало, что о нем подумают коллеги-ученые или промышленники – производители компьютерного оборудования. Истина – вот чему он служил и оставался верен до последних дней жизни…

Впрочем, не пожалейте немного времени и убедитесь в этом сами.

Alf

Программирование как вид человеческой деятельности

EWD117

Введение

В качестве введения мне хотелось бы начать разговор с истории и цитат. История эта – о физике Людвиге Больцмане, который хотел достичь своих результатов путем громоздких вычислений. Кто-то однажды пожаловался на то, что его методы ужасны, на что Больцман заявил, что «об элегантности должны заботиться портные и сапожники», дав тем самым понять, что его самого это никоим образом не беспокоит.

В противоположность ему, я хотел бы процитировать другого известного ученого XIX века, Джорджа Буля. В своей книге «Исследование законов мышления», в главе «Условия совершенного метода», он писал: «Я говорю здесь не только о том совершенстве, которое состоит в могуществе, но и о том, которое основывается на концепции изящества и красоты. Вполне возможно, что тщательное изучение этого вопроса приведет нас к выводу, что совершенный метод должен быть не только эффективным по отношению к объектам, для которых он разработан, но и демонстрировать определенное единство и гармонию всех своих частей и процессов». Вряд ли кто-то не заметит коренного различия в этих подходах.

Мы подсознательно ассоциируем элегантность с роскошью. Возможно, это одна из причин того, что для нас само собой разумеется, что элегантность должна дорого обходиться. Одна из моих основных целей – показать, что элегантность может быть также выгодна. Это даст нам ясное понимание истинной природы качества программ и пути, которым оно может быть достигнуто, а именно – языка программирования. Поняв это, мы попытаемся вывести некоторые ключевые моменты, например, какие особенности языков программирования являются наиболее предпочтительными. Наконец, мы надеемся убедить вас в том, что различные цели конфликтуют друг с другом меньше, чем это кажется на первый взгляд.

О качестве результатов

Даже полагая, что машины работают безупречно, мы должны задать себе вопрос: «Когда компьютер выдает результаты, почему мы должны им доверять, если только мы им действительно доверяем?», а затем: «Какие меры мы можем предпринять, чтобы повысить степень нашей уверенности в том, что выданные результаты – это то, что нам нужно на самом деле?».

Насколько важен первый вопрос, можно проиллюстрировать на простом, даже несколько упрощенном примере. Предположим, что математик, работающий в области теории чисел, имеет в своем распоряжении машину с программой факторизации чисел. Этот процесс может завершиться двумя способами: либо он выдает факторизацию данного числа, либо отвечает, что заданное число является простым. Предположим теперь, что наш математик хочет подставить в этот процесс, скажем, число с 20-ю десятичными знаками, для которого у него есть веские причины полагать, что оно простое. Если машина подтверждает это ожидание, он будет счастлив; если она находит факторизацию, математик будет разочарован, так как интуиция снова подвела его, но, если он сомневается, он может взять счетную машинку и перемножить полученные множители, чтобы проверить, получится ли в результате исходное число. Если же, наоборот, машина выдает ответ, что данное число, согласно его ожиданиям и горячему желанию, является простым, с чего бы ему верить этому?

Наш пример демонстрирует, что даже в полностью абстрактных задачах получение результата не является четко определенным процессом, четко определенным в том смысле, что можно сказать: «Я сделал это», не беспокоясь за убедительность результата, то есть его «качество».

Ситуация, в которой находятся программисты, очень похожа на ту, в которой находятся чистые математики, которые разрабатывают теорию и доказывают результаты. Долгое время чистые математики думали – а некоторые из них до сих пор думают – что теорема может быть доказана полностью, что вопрос о том, является ли предложенное доказательство теоремы адекватным или нет, допускает абсолютный ответ «да» или «нет». Но это иллюзия, потому что как только кто-то начинает думать, что доказал что-то, он должен доказать, что его доказательство безукоризненно, и так далее, до бесконечности! Никто не может гарантировать, что доказательство корректно, в лучшем случае он может сказать: «Я не нашел ни одной ошибки». Порой мы льстим себе мыслью о неопровержимом доказательстве, но на самом деле все, что мы делаем, это лишь придаем правдоподобный вид своим заключениям.

Несмотря на все свои недостатки, математические рассуждения представляют замечательную модель того, как можно охватить чрезвычайно сложные структуры посредством мозга, возможности которого ограничены. Кажется, стоит разобраться, до какой степени эти проверенные методы могут быть перенесены в искусство использования компьютеров. В разработке языков программирования можно позволить себе руководствоваться в первую очередь тем, «что машина может делать». Впрочем, учитывая, что язык программирования – это мост между пользователем и машиной и фактически может рассматриваться как его инструмент, кажется столь же важным рассматривать, «что Человек может придумать». Именно в этом русле мы продолжим наше исследование.

О структуре убедительных программ

Техника управления сложностью известна с древних времен: “Divide et impera” (разделяй и властвуй). Аналогия между построением доказательства и построением программы, пожалуй, просто поразительна. В обоих случаях даны отправные точки (аксиомы и существующая теория против примитивов и доступных библиотечных программ), в обоих случаях задана цель (доказанная теорема против желаемых результатов), в обоих случаях сложность преодолевается делением на части (леммы против подпрограмм и процедур).

Я полагаю, что гениальность программиста соответствует сложности решаемой задачи, а также полагаю, что он сумел добиться подходящего разделения задачи. Затем он продолжает действовать следующим образом:

1. Он разрабатывает полные спецификации отдельных частей.

2. Он убеждается, что проблема в целом решается при наличии в его распоряжении частей программы, удовлетворяющих этим спецификациям.

3. Он разрабатывает отдельные части в соответствии со спецификациями, но при этом делает их максимально независимыми друг от друга и от окружения, в котором они будут использоваться.

Очевидно, что построение каждой такой отдельной части может снова оказаться сложной задачей, так что для данной части задачи потребуется дальнейшее разбиение.

Некоторые могут счесть описанный метод разбиения на части недостаточно прямолинейным и слишком извилистым путем достижения конечной цели. Мое собственное мнение я лучше всего могу выразить так: я твердо уверен, что «царских дорог в математике нет», или, другими словами, что у меня очень маленькая голова, и я

вынужден обходиться ей. Поэтому я рассматриваю технику разбиения на части как базовый прием человеческого мышления и считаю, что стоит попробовать создать условия, в которых она может быть наиболее плодотворно применена.

Предположение о том, что программист сделал подходящее разбиение, находит подтверждение в том, что становится возможным выполнить первые два этапа: спецификацию частей и проверку, что они совместно решают задачу. Здесь элегантность, точность, ясность и тщательное понимание задачи являются необходимыми предпосылками. Но в целом техника разбиения основывается на том, что обсуждалось значительно меньше, а именно на том, я назвал бы «принципом невмешательства». На втором этапе подразумевается, что правильная работа целого может быть установлена путем рассмотрения только внешних спецификаций частей, без деталей их внутреннего строения. На третьем этапе принцип невмешательства всплывает снова; здесь подразумевается, что отдельные части могут быть поняты и построены независимо друг от друга.

Возможно, здесь уместно заметить, что если я правильно понял нынешнее отношение к проблеме определения языка, при несколько более формальном подходе состоятельность техники разбиения подвергается сомнению. Те, кто выдвигает возражения, аргументируют свою точку зрения следующим образом. Когда вы используете механизм, подобный описанному двухэтапному, во-первых, должны быть созданы спецификации, а во-вторых, описано, как все это работает. При этом вы вынуждены в лучшем случае сказать дважды одно и то же, но вероятнее всего, вы придете к противоречию. С точки зрения статистики, как ни грустно мне об этом говорить, последнее замечание достаточно серьезно. Единственный ясный путь к определению языка, возражают они, это просто определение механизмов, потому что все, что они будут делать, следует из этого. Мой вопрос: «А как оно следует?» мудро оставляют без ответа, и я боюсь, что это тонкое, но порой значительное различие между понятиями «определено» и «известно» сделают их работу интеллектуальным упражнением, которое ведет в тупик.
После этого отступления вернемся к собственно программированию. Каждый, кто знаком с ALGOL 60, согласится, что его концепция процедуры в высокой степени удовлетворяет нашей концепции невмешательства, как по своим статическим (например, в свободном выборе локальных идентификаторов), так и по динамическим свойствам (например, возможность вызова процедуры, прямо или косвенно, из себя самой).

Другой впечатляющий пример улучшения ясности посредством невмешательства, гарантированного структурой, представлен всеми языками программирования, в которых допустимы алгебраические выражения. Вычисление этих выражений последовательной машиной, имеющей арифметическое устройство ограниченной сложности, подразумевает использование временного хранилища для промежуточных результатов. Их анонимность в исходном языке гарантирует невозможность того, что один из них будет нечаянно разрушен до использования, что было бы возможно при программировании в кодах машины Фон Неймана.

Сравнение некоторых альтернатив

Развернутое сравнение кода машины Фон-неймановского типа – хорошо известное отсутствием ясности – и различных типов алгоритмических языков было бы не лишним.
Выполнение программы всегда состоит в периодическом взаимодействии двух информационных потоков, одного постоянного во времени («программы»), другого изменяющегося («данные»). Много лет одним из основных достоинств кода Фон-неймановского типа считалась возможность программы изменять свои собственные инструкции. Со временем мы обнаружили, что именно эта возможность в немалой степени ответственна за отсутствие ясности в программах на машинном коде. Тут же была

поставлена под вопрос необходимость этого: все алгебраические компиляторы, которые я знаю, производят объектные программы, остающиеся неизменными все время выполнения.

Это наблюдение приводит нас к рассмотрению статуса переменной информации. Давайте ограничимся рассмотрением языков программирования без операторов присваивания и перехода. При условии, что набор доступных функций достаточно широк и понятие условного выражения входит в число примитивов, можно описать вывод любой программы как значение большой (рекурсивной) функции. Для последовательной машины она может быть транслирована в постоянную объектную программу, в которой во время выполнения стек используется для отслеживания текущей иерархии вызовов и значений фактических параметров, передаваемых этим вызовам.

Несмотря на элегантность подобного языка программирования, имеется серьезный довод против него. Информация в стеке может рассматриваться как объекты с вложенными временами жизни и постоянными значениями в течение всего времени жизни. Нигде (за исключением явного наращивания счетчика команд, отражающего ход времени) значение уже существующего именованного объекта не заменяется другой величиной. В результате единственный способ сохранить только что полученный результат – выложить его на верхушку стека; у нас нет способа выразить, что прежнее значение устарело, и время его жизни будет продолжаться, хотя безо всякого интереса для нас. Второй довод против, возможно, является следствием первого: такую программу после некоторого, довольно быстро достижимого, уровня вложенности будет ужасно трудно читать.

Обычное средство от этого – совместное введение операторов присвоения и goto. Оператор goto позволяет нам посредством перехода назад повторить часть программы, тогда как оператор присвоения может создать необходимое различие в состоянии между последовательными повторениями.

Но у меня есть причины спросить: будучи примененным в качестве спасительного средства, оператор goto не хуже ли сам по себе, чем тот дефект, который он призван исправить? Например, два менеджера программных отделов из разных стран и разной квалификации – один в основном ученый, другой в основном коммерсант – сообщили мне независимо друг от друга и по собственной инициативе о своем наблюдении, что квалификация их программистов обратно пропорциональна частоте появления оператора goto в их программах. Это было стимулом к тому, чтобы попытаться обойтись без оператора goto.

Идея состоит в следующем: то, то нам известно как «передача управления», т.е. подмена счетчика инструкций, - это операция, обычно подразумеваемая как часть более содержательного действия. Я упомяну переход на следующий оператор, вызов процедуры и возврат из нее, условные конструкции и оператор for. Это еще вопрос, не собьет ли программиста с толку наличие отдельного средства управления передачей управления. Я поставил несколько программных экспериментов и сравнил текст на ALGOL с текстом, полученным мной на модифицированной версии ALGOL 60, в котором оператор goto был упразднен, а оператор for – будучи слишком помпезным и сложным – заменен более простой конструкцией повтора. Последние версии было потруднее создавать: мы так хорошо знакомы с командой перехода, что требовались некоторые усилия, чтобы забыть его! Во всех попытках, впрочем, программы без goto оказались короче и понятнее. Начальный шаг в повышении ясности достаточно понятен. Хорошо известно, что не существует алгоритма для определения, завершится данная программа или нет. Другими словами: каждый программист, который хочет создать безукоризненную программу, должен хотя бы убедиться сам, проверив, действительно ли завершается его программа. В программе, где goto применен в неограниченных количествах, этот анализ может оказаться очень трудным, учитывая большое разнообразие путей, по которым программа может зациклиться. После упразднения goto остаются только два способа, которыми

программа может зациклиться: либо бесконечная рекурсия – т.е. через механизм процедур – либо конструкция цикла. Это весьма упрощает проверку.

Понятие цикла, столь фундаментальное в программировании, имеет еще один аспект. Нет ничего необычного в том, что среди последовательности повторяющихся операторов появляется одно или более подвыражений, которые не изменяют значение в ходе цикла. Если такая последовательность должна повторяться много раз, было бы досадной потерей машинного времени перевычислять одни и те же значения снова и снова. Один из способов избежать этого – переложить на оптимизирующий транслятор выявление подобных константных подвыражений, чтобы он мог вынести вычисление их значений за пределы цикла. При отсутствии оптимизирующего транслятора очевидное решение – предложить программисту расписать их явно путем введения стольких дополнительных переменных, сколько константных подвыражений имеется в цикле, и присвоения им значений перед входом в цикл. Я должен подчеркнуть, что оба способа написания программ в равной степени сбивают с толку. В первом случае транслятор сталкивается с ненужной головоломкой по выявлению константности, во втором – мы вводим переменную, единственное назначение которой – обозначать константное значение. Последнее рассмотрение указывает выход из этого затруднения: помимо переменных, в распоряжении программиста должны быть «локальные константы», т.е. идентифицируемые величины с ограниченным временем жизни, в течение которого они сохраняют постоянное значение, заданное в момент появления величины. Эти величины не новы: формальные параметры процедур уже демонстрируют это свойство. Вышеупомянутое – это призыв осознать, что концепция «локальной константы» имеет право на существование. Если меня правильно информировали, это уже было заложено в CPL, язык программирования, разработанный совместными усилиями в математической лаборатории Кембриджского университета в Англии.

Двойная выгода от ясности

Я подробно рассуждал о том, что убедительность результата сильно зависит от ясности программы, от степени, в которой она отражает структуру выполняемого процесса. Для тех, кто в первую очередь озабочен эффективностью, измеряемой сомнительной мерой в виде занимаемой памяти и машинного времени, я хотел бы указать, что увеличение эффективности всегда ведет к использованию структуры. Для них я хотел бы особо подчеркнуть, что все упомянутые структурные свойства могут быть использованы для повышения эффективности реализации. Следовало бы снова пересмотреть их вкратце.

Аспекты, связанные со временем жизни, разрешаемые при помощи локальных величин процедуры, позволяют нам разместить их в стеке, тем самым используя доступную память весьма эффективно; анонимность промежуточных результатов позволяет нам динамически минимизировать обращения к памяти при помощи автоматически управляемого набора аккумуляторов; постоянство текста программы во время выполнения очень полезно на машинах с разными уровнями памяти и значительно уменьшает сложность управления; оператор цикла облегчает динамическое выявление бесконечного зацикливания, и, наконец, локальные константы – подходящие кандидаты для хранения в памяти с медленной записью и быстрым чтением, где она имеется.

Позвольте мне подвести итог. Когда я познакомился с понятием алгоритмических языков, я никогда не оспаривал господствующего тогда мнения, что проблемы разработки и реализации языка – в основном вопрос компромиссов: каждое новое удобство для пользователя должно быть оплачено при реализации, либо в виде дополнительных проблем при трансляции, либо при выполнении, либо при том и другом. Что ж, мы определенно живем не в раю, и я не собираюсь отрицать возможность конфликта между

удобством и эффективностью, но теперь я возражаю, когда этот конфликт подается как закономерный итог ситуации. Я придерживаюсь мнения, что стоило бы разобраться, до какой степени интересы Человека и Машины совпадают, и посмотреть, какие технологии мы можем изобрести на пользу всем нам. Я верю, что это исследование принесет плоды, и если этот рассказ пробудит в вас такую же надежду, он достиг своей цели.

Prof. Dr. E.W.Dijkstra
Department of Mathematics
Technological University
P.O.Box 513
EINDHOVEN
The Netherlands

Навстречу корректным программам

EWD241

Цель данного документа – отметить, какие вспомогательные средства для нашего интеллекта мы имеем в своем распоряжении для разработки и понимания алгоритмов, продемонстрировать некоторые приемы программирования, которые мы можем попытаться применить к своим задачам без ущерба для понимания, и подчеркнуть потребность в том, чтобы наши программы (т.е. окончательные и промежуточные версии) как можно точнее отражали наше понимание задачи и алгоритма ее решения.

Среди вспомогательных средств для интеллекта я особо хотел бы упомянуть три:

1. Перечисление.
2. Математическая индукция.
3. Абстракция.

Я рассматриваю как применение Перечисления умственные усилия, необходимые для понимания либо последовательной программы, выполняющей фиксированную временную последовательность действий, либо условного оператора, либо оператора выбора (так называемая «конструкция case»). Я считаю, что одно из основных свойств человеческого разума – плохая способность к перечислению. В частности, это означает:

1a) что текст, описывающий фрагмент последовательной программы, должен выполнять небольшое количество действий (то есть соответствующие вычисления могут быть сгруппированы и восприняты как небольшое количество действий, последовательных во времени);

1b) что количество альтернативных ветвей в операторе выбора должно быть невелико.

Математическая индукция упомянута явно, поскольку это стандартный шаблон рассуждений для понимания рекурсивных процедур и (намного чаще встречаемых) циклов; я ограничусь циклами, образованными некоторыми разновидностями оператора повторения.

Абстракция рассматривается как главный умственный инструмент, нужный для применения математической индукции – то есть для формирования концепций, в терминах которых действие шага индукции может быть описано – и важный для уменьшения применения перечисления: в случае оператора выбора он должен предоставить концепции, в терминах которых действие может быть описано независимо от выбранного пути.

Памятуя о вышесказанном, я приступлю к следующей задаче. Заданы 32 позиции, расположенные по кругу; нужно составить программу, находящую все способы (если они есть), которыми эти позиции могут быть заполнены нулями и единицами (по одной цифре в каждой позиции) таким образом, что 32 пятерки, составленные из смежных позиций, представляют 32 различных цепочки из пяти двоичных цифр. Заполнения, которые отличаются друг от друга только поворотом, рассматриваются как эквивалентные, все решения должны начинаться с пяти нулей. (Последнее условие уменьшает количество «независимых решений» в 32 раза).

Лайтманс показал, что эта циклическая задача эквивалентна следующей линейной задаче. Задан линейный массив на 36 позиций; нужно составить программу, находящую все способы (если они есть), которыми эти позиции могут быть заполнены нулями и единицами (по одной цифре в каждой позиции) таким образом, что 32 пятерки, составленные из смежных позиций, представляют 32 различных шаблона пяти двоичных цифр. Мы можем ограничиться последовательностями, начинающимися с пяти нулей, при этом 32 начальные цифры решения линейной задачи представляют решение циклической, и наоборот. [Доказательство Лайтманса выглядит следующим образом. Каждое решение линейной задачи начинается с 000001…, потому что последовательность 00000 может

встретиться только единожды; кроме того, последовательность 10000 также может встретиться только единожды. Поскольку за этими последовательностями могут идти только 00000 или 00001 (уже имеющиеся в первых двух пятерках), то последовательность 10000 не может встретиться в середине линейной последовательности и поэтому может появиться только в конце. В результате каждое решение линейной задачи заканчивается четырьмя нулями, и таким образом кольцо замыкается!]. Мы должны решить линейную задачу, наложив дополнительное условие, что решения (если их несколько) должны выдаваться в алфавитном порядке.

Самое грубое описание программы – это единственная инструкция.

Версия 0:

```
«выполни всю работу».
```

Это описание совершенно общее, но также и совершенно бесполезное, поскольку не отражает ни нашего понимания задачи, ни структуру алгоритма. Чтобы двигаться дальше, мы должны глубже проанализировать задачу.

Само собой разумеется, что для заданной последовательности из 36 двоичных цифр может быть вычислена булевская функция, устанавливающая, является ли эта последовательность решением, и что мы можем написать алгоритм ее вычисления. В принципе мы могли бы написать программу, генерирующую все 36-цифровые последовательности с пятью ведущими нулями в алфавитном порядке и подвергающую все эти последовательности такой проверке, отбирая таким образом те, которые выдержали тест. Этот метод дает весьма нереальную программу, и мы не должны ему следовать; заметим только, что генерация пробных последовательностей в алфавитном порядке обеспечит, что решения, если таковые будут найдены, будут упорядочены по алфавиту.

Примечание. Наша окончательная программа может рассматриваться как производная от программы, набросок которой приведен выше, путем добавления нескольких мелких, но важных деталей. Сейчас я не склонен подчеркивать эту преемственность, так как она кажется тесно связанной с этой специфической задачей.

В нашем следующем приближении мы снова будем генерировать наши решения путем (генерации и) сканирования больших наборов последовательностей, из которых будут выбираться все решения по подходящему критерию.

Определим «**длину последовательности**» как количество пятерок, которое она содержит (то есть **длина** = **число цифр** – 4). Будем называть последовательность «**приемлемой**», если никакие две различные пятерки в ней не представляют одинаковых цепочек цифр. С такими определениями решения – это подмножество множества приемлемых последовательностей, а именно те, у которых длина = 32.

Мы не знаем, существуют ли решения вообще, но зато знаем, что множество приемлемых последовательностей не пусто (например, «00000»); у нас нет готового критерия для распознавания «**последнего решения** », когда мы его достигнем в нашем наборе приемлемых последовательностей; впрочем, мы можем пометить «**виртуальное последнее решение**» (а именно «00001»): когда оно достигнуто, мы знаем, что все приемлемые последовательности с пятью ведущими нулями просмотрены и что больше решений найдено не будет. Подытожим, что мы знаем о множестве приемлемых последовательностей:

1. Оно непустое и конечное.
2. Мы знаем первый член («00000»).
3. Мы знаем виртуальный последний член («00001»)
4. Мы можем преобразовать **приемлемую последовательность** в следующую **приемлемую последовательность**

5. **Решения – это все** приемлемые последовательности (кроме виртуальной), удовлетворяющие условию «длина последовательности равна 32».

Переход от рассмотрения только набора решений к рассмотрению набора приемлемых последовательностей явился следующим шагом в нашем анализе, который является первым улучшением: программой, в которой инструкции оперируют объектом (все еще достаточно абстрактным), называемым «**последовательностью**».

Версия 1:

```
Установить последовательность на первую приемлемую;
repeat
  if длина последовательности = 32 do
    begin
      принять последовательность как решение;
      печатать решение
    end;
    преобразовать последовательность в следующую приемлемую
until последовательность – это (первый или) виртуальный последний член
```

Для пояснения этой программы уместно было бы сделать пару замечаний.

Замечание 1. Последняя проверка должна отличать последний член от всех, кроме первого, так как первый не проверяется. По существу, не должно быть возражений, если первый член будет равен последнему виртуальному (например: пустая последовательность).

Замечание 2. Оператор «**принять последовательность как решение**» может изрядно озадачить читателя. Он вполне может оказаться пустым. Он включен в тест затем, чтобы подчеркнуть, что за «следующей точкой с запятой» последовательность рассматривается как представляющая решение. (Можно представить его себе как вытаскивание первых 32 цифр).

Критерий «приемлемости» имеет важное свойство:

6. никакое расширение неприемлемой последовательности не является приемлемым

и это его важное свойство будет использовано при уточнении оператора «**преобразовать последовательность в следующую приемлемую**». Прямое следствие свойства 6: проверка приемлемости должна применяться только лишь к так называемым «**потенциально приемлемым последовательностям**», которые можно определить как приемлемую последовательность, к которой добавлена одна цифра. Это приводит к следующему улучшению программы:

Преобразовать последовательность в следующую приемлемую:

```
преобразовать приемлемую-последовательность в следующую потенциально-
приемлемую-последовательность ;
while потенциально-приемлемая-последовательность не является приемлемой do
  преобразовать потенциально-приемлемую-последовательность в следующую
потенциально-приемлемую-последовательность ;
принять последовательность как приемлемую
```

Когда мы рассматриваем «**преобразовать последовательность в следующую приемлемую»** как доступный примитив, значение «**последовательности**» всегда приемлемо; только лишь при рассмотрении его внутреннего строения – в данном случае возле точки с запятой в предыдущем фрагменте – текущее значение «**последовательности**» - это **потенциально-приемлемая-последовательность**.

Ввиду требования алфавитной упорядоченности «**преобразовать потенциально-приемлемую-последовательность в следующую потенциально-приемлемую-последовательность**» образует новую последовательность, которая равна старой, дополненной нулем, тогда как «**преобразовать потенциально-приемлемую-последовательность в следующую потенциально-приемлемую-последовательность**»

образует новую последовательность, равную старой, за исключением последнего нуля, дополненной единицей. Итак, очередное усовершенствование программы выглядит так:

преобразовать приемлемую-последовательность в следующую потенциально-приемлемую-последовательность:

```
while последовательность заканчивается единицей do
  удалить последнюю цифру из последовательности;
заменить последний нуль единицей
```

В приведенных выше пошаговых усовершенствованиях программы мы сосредоточили внимание на действиях программы с последовательностью. Теперь пора более явно представить решения, как в действительности должен быть представлен абстрактный объект под названием «**последовательность**». Введем integer k и положим k = **длина последовательности**. Затем введем integer array d[-3:33] для представления цифр, при этом d[-3] d[-2] ……. d[k] представляют последовательность. (1)

Замечание 1: с d[-3] по d[0] должны быть равны нулю.

Замечание 2: Максимальная длина **приемлемой-последовательности** = 32; так как алгоритм оперирует с **потенциально-приемлемыми-последовательностями** и **потенциально-приемлемая-последовательность** – это **приемлемая-последовательность**, дополненная одной цифрой, то максимальная длина **последовательности** – 33.

Представленные выше соглашения служат базой для более детального определения свойства «**приемлемости**». Мы должны характеризовать цифровые цепочки как представленные пятерками, содержащимися в текущем значении **последовательности**. Я предлагаю характеризовать такие цепочки цифр целым числом, которое получается при интерпретации такой пятерки цифр как двоичного числа. Другими словами, мы определяем функцию H(i) для 1 <= i <= k:

H(i) = d[i-4]*16 + d[i-3]*8 + d[i-2]*4 + d[i-1]*2 + d[i] (2)

Свойство «**приемлемости**» выглядит так:

для 1<=i,j<=k, i<>j означает H(i)<>H(j) (3)

В любой момент функция H(i) определена на текущем значении последовательности для1<=i<=k. Вместо того, чтобы вычислять значения этой функции каждый раз, когда они нам понадобятся (например, в проверке **приемлемости**), мы можем протабулировать их значения в integer array h[1:33].

По нашему соглашению для всех значений **последовательности**

h[i] = H(i) для 1<=i<=k. (4)

Это соглашение подразумевает, что изменение значения последовательности в общем случае включает изменение массива h, чтобы следовать отношению (4).

Проверка **приемлемости** теперь – по аналогии с (3) – такова:

для 1<=i,j<=k из i<>j следует h[i]<>h[j].

Теперь мы воспользуемся фактом, что тесту на **приемлемость** подвергаются только **потенциально-приемлемые последовательности**, т.е. **приемлемые последовательности**, дополненные одной цифрой. Для потенциально-приемлемой последовательности мы можем заключить, что она является приемлемой тогда и только тогда, когда

h[i]<>h[k] для 1<=i<k (5)

т.е. когда последняя пятерка представляет цепочку, отличную от всех других. Это снова предполагает сканирование, но мы можем повторить трюк и протабулировать, появлялась ли уже некоторая цепочка цифр. Наиболее элегантный способ – это ввести boolean array in[0:31], в котором для 0<=m<=31 in[m] означает:

для **приемлемой-последовательности**: m присутствует среди h[1]…h[k]

для **потенциально-приемлемой последовательности**: m присутствует среди h[1]… h[k-1] (6)

Примечание. Если последовательность **приемлема**, каждая цепочка может присутствовать только один раз, поэтому для того, чтобы отметить ее появление, достаточно булевой переменной. Для **потенциально-приемлемой** последовательности h[k] может быть равным h[i] для i<k; таким образом, соглашение (6) различно для **приемлемой** и для **потенциально-приемлемой** последовательностей.

Теперь дадим окончательную версию программы. То, что было именами примитивов, теперь становится метками (либо операторов, либо выражений)

```
begin
  integer k;
  integer array d[-3:33];
  integer array h[1:33];
  boolean array in[0:31];
  // установить последовательность на первую приемлемую
  begin
    d[-3] := d[-2] := d[-1] := d[0] :=d [1] := 0;
    k := 1;
    h[1]: = 0;
    in[0] := true;
    begin
      integer m;
      m := 1;
      repeat
        in[m] := false;
        m := m + 1
      until m = 32
    end
end;
repeat
  if (k = 32) then // длина последовательности равна 32
    begin
      // принять последовательность как решение:
      new line carriage return;
      // print solution:
      begin
        integer m;
        m := 0;
        repeat
          print(d[m - 3]);
          m := m + 1
        until m = 32
      end
    end;
  // преобразовать последовательность в следующую приемлемую:
  begin
    // преобразовать последовательность в следующую
потенциально-приемлемую:
    begin
      k := k + 1;
      d[k] := 0;
      h[k]:= 2 * h[k - 1] - 32 * d[k - 4]
    end;
    while (in[h[k]]) do // потенциально-приемлемая
последовательность не является приемлемой:
    // преобразовать потенциально-приемлемую последовательность
в следующую приемлемую:
      begin
        while (d[k] = 1) do //
последовательность заканчивается единицей
        // удалить последнюю цифру из
```

```
последовательности:
          begin
            k := k - 1;
            in[h[k]] :=
false
          end;
        // заменить последний нуль единицей:
        begin
        d[k] : = 1;
        h[k] := h[k] + 1
      end
    end;
    // принять последовательность как приемлемую:
    in[h[k]] := true
  end
until (k = 1) // последовательность - виртуальный последовательный член
end
```

Пояснения:

1. «**принять последовательность как решение**» могло бы быть пустым оператором, но вместо этого задает переход на новую строку;
2. «**принять последовательность как приемлемую**» получает содержание в соответствии с соглашением (6), где делается различие между **приемлемой** и **потенциально-приемлемой** последовательностями.

Заключение.

Мы продемонстрировали последовательность версий программы от начальной постановки задачи до окончательной программы. В окончательном варианте программы вручную было произведено слияние последовательных версий, при этом более абстрактные версии постепенно выродились в комментарии.

Для больших программ сам по себе подобный процесс слияния становится серьезной задачей обработки данных, и я ожидаю появления интерактивных технологий составления программ, в которых этот процесс будет упрощен посредством привлечения помощи компьютера.

Более того: в настоящий момент более абстрактные версии исполняют роль только пояснительных комментариев, включенных для облегчения понимания программы человеком. Источник этого – наше желание иметь программы, сформулированные на постоянном семантическом уровне, в данном случае – на уровне языка программирования. На сегодняшний день более абстрактные версии совершенно бесполезны для выполнения их на машине. Я ожидаю, что в будущем более абстрактные версии станут неотъемлемой частью программы.

Наконец, мы рассмотрели только одну линию программ от постановки задачи до рабочей программы, написанной на желаемом семантическом уровне. Я ожидаю, что в будущем эта единственная линия будет расширена до более или менее структурированного в виде дерева класса программ, в котором найдется также место и альтернативам, таким образом связывая вместе составление программ и их модификацию.

Edsger W. Dijkstra

Department of Mathematics Technological University

The Netherlands

(Эта статья была написана в связи с докладом, прочитанным в университете Гренобля в декабре 1967 г.).

Смиренный программист

EWD340

(Рукопись опубликована в Commun. ACM 15 (1972), 10: 859-866)

В результате долгой цепочки случайностей первым весенним утром 1952 года я официально стал программистом, и насколько я могу судить, я был первым голландцем, выбравшим эту стезю. Помнится, самой забавной вещью была та неторопливость, с которой, по крайней мере в той части мира, где я жил, появлялась профессия программиста, неторопливость, в которую теперь даже поверить трудно.

После того, как я прозанимался программированием около трех лет, у меня состоялась беседа с А. Ван Вейнгаарденом, который был в то время моим боссом в Математическом центре Амстердама, беседа, за которую я буду благодарен ему до конца моих дней. Дело в том, что предполагалось, что я буду параллельно изучать теоретическую физику в Лейденском университете, и так как мне становилось все труднее и труднее совмещать два этих занятия, я должен был определиться: либо бросить программирование и стать почтенным физиком-теоретиком, либо кое-как доучиться физике до формального выпуска с минимальными усилиями, и затем стать..., а кем, кстати? Программистом? Но является ли это достойной профессией? В конце концов, что это такое - программирование? Где та солидная теоретическая основа, которая должна поддерживать его как уважаемую интеллектуальную дисциплину? Я довольно отчетливо помню, как я завидовал своим коллегам, которые работали с оборудованием: когда у них спрашивали об их профессиональных навыках, они по крайней мере могли сказать, что знают все о вакуумных лампах, усилителях и тому подобных вещах, тогда как я чувствовал, что, когда столкнусь с этим вопросом, мне останется только лишь развести руками. Полный опасений, я постучал в дверь Ван Вейнгаардена, спросив, могу ли я "поговорить с ним минутку"; несколько часов спустя я покинул его офис другим человеком. Внимательно выслушав мои проблемы, он согласился, что до сих пор наука программирования не так уж развита, но затем принялся неторопливо объяснять, что автоматические компьютеры - это надолго, что мы находимся в самом начале, и почему бы мне не стать одним из тех, кто призван сделать программирование уважаемой дисциплиной? Это стало поворотной точкой в моей жизни, и я формально закончил изучение физики так быстро, как только мог. Мораль этой истории - в том, что, конечно, мы должны быть очень осторожны, давая советы молодежи: иногда они следуют им!

Еще два года спустя, в 1957, я женился, и голландский обряд регистрации брака требовал указать профессию; я указал "программист". Но городские власти Амстердама не приняли документы на том основании, что такой профессии не существует. Хотите - верьте, хотите - нет, но в графе "профессия" моего свидетельства о браке значится забавная запись "физик-теоретик"!

Стоит ли говорить о том, как медленно профессия программиста пробивала себе путь в моей стране. С тех пор я немало повидал во всем мире, и мое общее впечатление таково, что в других странах, если не считать возможного сдвига дат, ее продвижение в целом было таким же.

Позвольте мне попытаться передать события этих давно минувших дней немного детальнее, чтобы попытаться получше понять, что же происходит сегодня. В процессе нашего анализа мы увидим, как много недоразумений относительно истинной природы программирования восходит к тем временам.

Первые автоматические электронные компьютеры были уникальными машинами, построенными в единственном экземпляре, и они находились в восхитительном окружении экспериментальной лаборатории. Как только дух автоматического компьютера в ней побывал, его воплощение становилось потрясающим вызовом электронной технологии того времени, и одно несомненно: мы не можем отказать в мужестве группам, которые решили попытаться построить это фантастическое сооружение. Потому что эти

сооружения действительно были фантастическими: оглядываясь назад, мы можем только изумляться тому, что эти первые машины вообще работали, по крайней мере, иногда. Самой обременительной проблемой было заставить машину работать и поддерживать ее в этом состоянии. Озабоченность физической стороной автоматических вычислений до сих пор находит отражение в названиях старейших научных сообществ этого направления, таких как the Association for Computing Machinery или the British Computer Society, - в названиях, в которых прослеживается явный намек на физическое оборудование.

А как же бедный программист? Что же, честно признаться, о нем вряд ли даже вспоминали. Во-первых, первые машины были столь громоздки, что вы вряд ли смогли бы сдвинуть их с места, и кроме того, они требовали такого объема работ по обслуживанию, что было вполне естественным, что их пытались использовать в той же лаборатории, где они и разрабатывались. Во-вторых, его практически невидимый труд был полностью лишен волшебного ореола: вы могли показать машину посетителям, и это было несравнимо зрелищнее, чем несколько листов, покрытых кодами. Но самое важное во всем этом - сам программист очень скромно оценивал собственный труд: вся значимость его работы определялась существованием этой чудесной машины. Так как эта машина была уникальной, он слишком хорошо сознавал, что его программы имеют исключительно местное значение. Также, поскольку было совершенно очевидно, что время жизни данной машины ограничено, то он знал, что весьма малая часть его работы будет иметь долговременное значение. Наконец, есть еще одно обстоятельство, которое оказало глубокое влияние на отношение программиста к своей работе: с одной стороны, помимо своей ненадежности, его машина обычно была слишком медленной и имела слишком мало памяти, так что он постоянно находился в прокрустовом ложе, тогда как с другой стороны ее причудливая система команд приводила к весьма неожиданным конструкциям. В те времена смышленый программист получал немалое интеллектуальное удовлетворение от изощренных трюков, посредством которых он умудрялся втиснуть невозможное в тесные рамки своего оборудования.

В те времена сложилось два мнения о программировании. Я упомяну о них сейчас и вернусь к ним позже. Одно мнение - действительно компетентный программист должен обладать парадоксальным мышлением и обожать заумные трюки; второе - программирование представляет собой не более чем оптимизацию эффективности вычислительного процесса в том или ином направлении.

Последнее мнение являлось результатом того, что весьма часто мощности имеющегося оборудования катастрофически не хватало, и в то время часто встречались наивные предположения, что как только более мощные машины станут доступны, программирование перестанет быть проблемой, ибо закончится борьба с тесными рамками оборудования, а ведь именно в этом и заключается суть программирования, не так ли? Но в следующие десятилетия произошло кое-что совершенно иное: стали доступны более мощные машины, не на один, а на несколько порядков величины мощнее прежних. Но вместо того, чтобы оказаться в состоянии бесконечного блаженства от того, что все проблемы программирования решены, мы оказались по самое горло в кризисе программирования! Как же это случилось?

Вот второстепенная причина: в одном-двух аспектах современные компьютеры значительно сложнее содержать, чем старые. Во-первых, мы получили прерывания ввода/вывода, происходящие в непредсказуемые и невоспроизводимые моменты времени; в сравнении со старыми последовательными машинами, которые прикидывались полностью детерминированными автоматами, это разительное изменение, и преждевременная седина многих системных программистов служит свидетельством тому, что нам не стоит легкомысленно отзываться о логических проблемах, порожденных этой возможностью. Во-вторых, мы получили машины, оборудованные многоуровневыми запоминающими устройствами, что ставит перед нами проблемы стратегии управления ею, которые, несмотря на обилие литературы по этому предмету, остаются весьма

скользкими. Слишком дорогая плата за возрастание сложности, обусловленное изменениями в структуре современных машин.

Но я назвал это второстепенной причиной; первостепенная же кроется в том, что... машины стали на несколько порядков мощнее! Говоря прямолинейно: пока машин не было вовсе, не существовало проблемы программирования; когда появились немногочисленные слабые компьютеры, программирование стало малозаметной проблемой; а теперь, когда у нас есть гигантские компьютеры, программирование само превратилось в гигантскую проблему. В этом смысле электронная промышленность не решила ни единой проблемы, она только породила их, создав проблему использования своей продукции. Другими словами, по мере того как мощность доступных машин выросла более чем в тысячу раз, стремление общества найти им применение выросло пропорционально, и бедный программист вынужден метаться буквально по минному полю. Возросшая мощь оборудования совместно с, возможно, еще более возросшей надежностью, сделали возможными решения, о которых программист даже не отваживался мечтать несколько лет назад. А теперь, спустя несколько лет, он должен мечтать о них и, более того, он обязан воплощать эти мечты в реальность! Удивительно ли, что мы оказались в кризисе программирования? Конечно же, нет, и как вы можете догадаться, это даже было предсказано заранее; но трагедия пророков, предсказывающих неприятности, состоит в том, что только лет через пять вы действительно осознаете, что они были правы.

Затем в середине шестидесятых годов произошло нечто ужасное: появились так называемые "компьютеры третьего поколения". Официальная литература утверждает, что соотношение "цена/производительность" было одной из главных целей разработки. Но если рассматривать как производительность рабочий цикл различных компонентов машины, вряд ли что-то сможет удержать вас от разработки, в которой большая часть производительности достигается посредством внутренних действий сомнительной необходимости. А если под ценой вы понимаете ту цену, которую приходится платить за оборудование, мало что удержит вас от разработки устройство, которое будет дьявольски трудно программировать: например, коды команд могут потребовать ранней компоновки либо от программиста, либо от системы, что представляет в действительности неразрешимый конфликт. И в немалой степени эти неприятные свойства стали реальностью.

Когда про эти машины было заявлено и стали известны их функциональные спецификации, многие из нас стали несчастны; по крайней мере, я стал. Можно было только ожидать, что подобные машины заполонят компьютерное сообщество, и поэтому важнее всего было разработать их как можно более надежными. Но в разработку вкрались столь серьезные изъяны, что я почувствовал, что одним ударом компьютерная наука отброшена как минимум на десятилетие: это была самая черная неделя в моей профессиональной жизни. Пожалуй, самое грустное сейчас - это то, что несмотря на все эти годы обескураживающего опыта, так много людей до сих пор искренне верят, что эти машины развиваются согласно законам природы. Они молчат о своих сомнениях, видя, как много подобных машин уже продано, и это внушает им ложное чувство безопасности, что в конце концов разработка не могла быть так уж плоха. Но при ближайшем рассмотрении эта линия защиты столь же убедительна, сколь и аргумент, что курение полезно для здоровья, потому что так много людей курят.

В связи с этим я сожалею, что не в обычаях научных журналов компьютерной тематики публиковать обзоры недавно анонсированных компьютеров, подобно обзорам научных публикаций: обозревать машины по меньшей мере столь же важно. И здесь я должен признаться: в начале шестидесятых я написал такой обзор, намереваясь отправить его в CACM. Но, несмотря на то, что немногие коллеги, которым я разослал текст на рецензию, побуждали меня сделать это, я все же не рискнул, опасаясь, что трудности либо для меня, либо для редакции окажутся слишком велики. За это малодушие я виню себя все

больше и больше. Трудности, которые я предвидел, были следствием общепринятых критериев, и хотя меня убедили в правильности выбранных мной критериев, я опасался, что мой обзор будет отвергнут как "выражение личного мнения". Я до сих пор считаю, что подобные обзоры были бы чрезвычайно полезны, и мне бы очень хотелось, чтобы они появились, потому что их принятие было бы безусловным признаком зрелости компьютерного сообщества.

Причина, по которой я выше уделил столь много внимания оборудованию, кроется в моей уверенности, что один из важнейших аспектов любого вычислительного инструмента - это его влияние на образ мыслей тех, кто пытается его использовать, а также в том, что у меня есть причины считать, что это влияние во много раз сильнее, чем принято думать. Давайте теперь переключим наше внимание на область программного обеспечения.

Здесь разнообразие было столь велико, что я вынужден ограничиться несколькими путевыми вехами. Я убежден в своей предвзятости, поэтому умоляю не делать никаких выводов и помнить о том, что я упоминаю здесь отнюдь не все достижения, которые высоко ценю.

Вначале был EDSAC в Кембридже, Англия, и я нахожу весьма впечатляющим, что с самого начала понятие библиотеки подпрограмм играло центральную роль при проектировании этой машины и способа ее использования. Прошло почти 25 лет, и обстановка в компьютерном мире радикально изменилась, но понятие базового программного обеспечения до сих пор с нами, и понятие замкнутой подпрограммы остается одной из ключевых концепций программирования. Мы должны признать замкнутые подпрограммы одним из величайших изобретений в программировании; оно пережило три поколения компьютеров и будет жить еще значительно дольше, поскольку представляет основу для реализации наших базовых механизмов абстракции. К огромному сожалению, их важность была недооценена при разработке компьютеров третьего поколения, в которых большое количество явно именуемых регистров арифметического устройства подразумевает большие накладные расходы для механизма подпрограмм. Но даже это не убило концепцию подпрограммы, и мы можем только молиться о том, чтобы эта мутация не оказалась наследуемой.

Вторым большим шагом вперед в области программирования, который мне хотелось бы упомянуть, является рождение FORTRAN'a. В то время это был проект безрассудной смелости, и люди, ответственные за него, заслуживают нашего величайшего восхищения. Было бы черной неблагодарностью винить их за недостатки, которые стали очевидны только после десятилетия интенсивного использования: группы, обладающие даром предвидения на десятилетие, весьма редки! Оглядываясь назад, мы должны оценить FORTRAN как успешную технологию кодирования, но очень мало помогающую мышлению, а эта помощь так нужна сейчас, что пришло время признать эту технологию устаревшей. Чем скорее мы сможем забыть, что FORTRAN когда-то существовал, тем лучше, потому что как ускоритель мысли он уже не адекватен: он растрачивает попусту наши мыслительные способности, он слишком рискован и, таким образом, слишком дорог, чтобы его использовать. Трагедия FORTRAN'a состоит в его общепринятости, приковывании умов тысяч и тысяч программистов к нашим прошлым ошибкам. Я ежедневно молюсь о том, чтобы больше моих коллег-программистов смогли найти пути к освобождению от проклятия совместимости.

Третий проект, который не хотелось бы оставить в забвении, - это LISP, очаровательный проект совершенно иной природы. Имея в своей основе так мало базовых принципов, он продемонстрировал замечательную стабильность. Кроме этого, LISP послужил основой для значительного числа в некотором роде наших наиболее изощренных компьютерных приложений. LISP был шутливо охарактеризован как "наиболее интеллектуальный способ использования компьютера не по назначению". Я считаю эту характеристику лестным комплиментом, поскольку она в полной мере

передает дух освобождения (разума): он помог многим нашим наиболее одаренным коллегам в обдумывании идей, доселе немыслимых.

Четвертый проект, о котором следует упомянуть, - это ALGOL 60. В то время как до сих пор программисты на FORTRAN'е продолжают осознавать свой язык в рамках конкретной реализации, с которой они работают - отсюда и избыток восьмеричных или шестнадцатеричных дампов, - в то время как определение языка LISP остается причудливой смесью описаний, что означает язык и как работает его механизм, знаменитый "Отчет об алгоритмическом языке ALGOL 60" - это продукт усилий по продвижению абстракции на один жизненно важный шаг дальше и по определению языка программирования способом, независимым от реализации. Кое-кто может возразить, что в этом отношении его авторы столь преуспели, что им удалось посеять серьезные сомнения, может ли он вообще быть реализован! "Отчет" великолепно демонстрирует мощь формального метода BNF, теперь справедливо известного как Backus-Naur-Form (Формализм Бэкуса-Наура, или Форма Бэкуса-Наура - прим. перев), и силу тщательно сформулированного английского языка, по крайней мере в изложении тех, кто владеет им столь же блестяще, как Питер Наур. Я думаю, справедливо будет заметить, что очень немногие документы, столь же короткие, как этот, оказали такое глубокое влияние на компьютерное сообщество. Легкость, с которой в последующие годы слова "ALGOL" и "ALGOL-подобный" использовались как незащищенные торговые марки, чтобы придать часть своей славы незрелым проектам, порой имеющим весьма слабое отношение к предмету, - это порой обескураживающий комплимент его основам. Сила BNF как определяющего механизма в ответе за то, что я расцениваю как одну из слабостей языка: переработанный и не слишком систематический синтаксис мог бы теперь быть втиснут всего в несколько страниц. Располагая столь мощным инструментом, каким является BNF, "Отчет об алгоритмическом языке ALGOL 60" должен был бы быть значительно короче. Кроме того, я весьма усомнился в механизме параметров ALGOL'а 60: он предоставляет программисту столько комбинаторной свободы, что его надежное использование требует от программиста строгой дисциплины. Помимо того, что он дорог в реализации, он кажется также опасным в использовании.

Наконец, хотя этот предмет не из приятных, я должен упомянуть PL/1, язык программирования, документация которого обладает устрашающими размерами и сложностью. Использование PL/1 больше всего напоминает полет на самолете с 7000 кнопок, переключателей и рычагов в кабине. Я совершенно не представляю себе, как мы можем удерживать растущие программы в голове, когда из-за своей полнейшей вычурности язык программирования - наш основной инструмент, не так ли! - ускользает из-под контроля нашего интеллекта. И если мне понадобится описать влияние, которое PL/1 может оказывать на своих пользователей, ближайшее сравнение, которое приходит мне в голову, - это наркотик. Я помню лекцию в защиту PL/1, прочитанную на симпозиуме по языкам программирования высокого уровня человеком, который представился одним из его преданных пользователей. Но после похвал в адрес PL/1 в течение часа он умудрился попросить добавить к нему около пятидесяти новых "возможностей", не предполагая, что главный источник его проблем кроется в том, что в нем уже и так слишком уж много "возможностей". Выступающий продемонстрировал все неутешительные признаки пагубной привычки, сводящейся к тому, что он впал в состояние умственного застоя и может теперь только просить еще, еще, еще... Если FORTRAN называют детским расстройством, то PL/1, с его тенденциями роста подобно опасной опухоли, может оказаться смертельной болезнью.

Я немало рассказал о прошлом. Но ведь нет никакого смысла делать ошибки, если мы впоследствии не способны извлечь из них урок. В действительности я считаю, что мы научились столь многому, что через несколько лет программирование станет видом деятельности, в корне отличным от того, каким оно было до сих пор, настолько отличным, что нам, пожалуй, лучше приготовиться к шоку. Позвольте мне набросать для вас одну из

возможных картин будущего. На первый взгляд, такая точка зрения на программирование в, возможно, уже ближайшем будущем может поразить вас своей крайней фантастичностью. Поэтому позвольте мне привести доводы, из которых можно сделать заключение, что она может быть весьма реалистичной.

Эта точка зрения такова: еще до конца семидесятых мы сможем разрабатывать и реализовывать такие виды систем, которые нынче едва по силам нашим возможностям как программистов, ценой всего лишь несколько процентов тех человеко-лет, в которые они нам обходятся сегодня, и кроме того, эти системы будут практически свободны от ошибок. Эти два усовершенствования идут рука об руку. В последнем отношении программное обеспечение кажется отличным от множества других продуктов, для которых, как правило, более высокое качество подразумевает более высокую цену. Те, кто желает получить действительно надежное программное обеспечение, обнаружат, что они должны искать средства избежать большинство ошибок с самого начала, и в результате сам процесс программирования станет дешевле. Если вам нужны более эффективные программисты, вы обнаружите, что они не должны растрачивать свое время на отладку, - они с самого начала не должны вносить ошибки в программы. Другими словами: обе цели требуют одной и той же перемены.

Такая значительная перемена в столь короткий период времени была бы революцией, и для всех, кто строит свои надежды на будущее на осторожной экстраполяции недавнего прошлого, основываясь на неписанных законах социальной и культурной инертности, вероятность, что такая перемена произойдет, должна казаться пренебрежимо малой. Но мы знаем, что революции все же происходят время от времени! И какие же изменения она принесет нам?

Похоже, должны выполниться три основных условия. Во-первых, во всем мире должны признать необходимость перемен; во-вторых, для них должны быть достаточно сильны экономические потребности; и в-третьих, перемены должны быть осуществимы технически. Давайте обсудим эти три условия в этом же порядке.

В отношении признания необходимости более высокой надежности программного обеспечения, я надеюсь, больше нет возражений. Всего несколько лет назад это было не так: разговоры о кризисе программного обеспечения расценивались как кощунство. Поворотной точкой стала "Конференция по инженерии программного обеспечения" в октябре 1968 г. в Гармише, конференция, которая породила сенсацию, ибо именно на ней впервые открыто было признано существование кризиса программного обеспечения. К настоящему моменту стало общепризнанным, что разработка любой крупной и сложной системы будет очень трудной задачей, и каждый раз, встречаясь с людьми, ответственными за такое мероприятие, можно заметить их озабоченность проблемой надежности, и это действительно так. Короче говоря, кажется, наше первое условие выполнено.

Теперь об экономической потребности. Теперь часто встречается мнение, что в шестидесятые программирование было чересчур высокооплачиваемой профессией, и в ближайшие годы зарплаты программистов, вероятно, будут понижаться. Обычно это мнение высказывается в связи с экономическим спадом, но это могло бы быть и симптомом кое-чего другого, и весьма резонным, а именно - возможно, программисты истекшего десятилетия выполнили свою работу вовсе не так хорошо, как следовало бы. Общество разочаровывается в производительности труда программистов и в результатах их усилий. Но есть и другой фактор, куда более весомый. В сложившейся ситуации вполне естественно, что для заданной системы стоимость разработки программного обеспечения имеет примерно тот же порядок, что и стоимость необходимого оборудования, и общество более-менее смирилось с этим. Но производители оборудования уверяют нас, что в следующем десятилетии ожидается снижение стоимости оборудования раз в десять. Если разработка программного обеспечения будет и далее столь же дорогим и неуклюжим процессом, каким она является сейчас, баланс будет

окончательно нарушен. Нельзя надеяться, что общество смирится с этим, и таким образом мы должны научиться программировать на порядок эффективнее. Другими словами, пока на долю оборудования приходилась наиболее весомая часть бюджета, программированию сходили с рук неуклюжие технологии, но долго в тени этого зонтика прятаться не поучится. Короче говоря, наше второе условие также выполнено.

А теперь - третье условие: выполнимо ли все это технически? Я думаю, могло бы быть выполнимо, и приведу шесть аргументов в поддержку этого мнения.

Изучение структуры программ выявило, что программы - даже альтернативные варианты, которые решают те же самые задачи и оперируют теми же математическими константами - могут разительно отличаться по своей интеллектуальной управляемости. Было открыто несколько правил, нарушение которых либо существенно ослабляет, либо вовсе разрушает интеллектуальную управляемость программы. Эти правила делятся на два вида. Правила первого вида накладываются автоматически выбором адекватного языка программирования. Примеры - исключения операторов goto и процедур с более чем одним выходным параметром. Для правил второго вида я по меньшей мере - возможно, из-за недостаточной компетентности с моей стороны - не вижу способа механического применения, потому что, мне кажется, для этого потребуется некий автомат для доказательства теорем, для которого я не располагаю доказательством существования. Поэтому сейчас (и возможно, навсегда) правила второго вида представляют собой элементы дисциплины, требуемой от программиста. Некоторые из этих правил, которые я держу в голове, настолько ясны, что им можно обучить и что никогда не потребуется спорить, нарушает ли их данная программа или нет. Примеры: требования к циклам, которые не должны записываться без приведения доказательства их завершения, а также без установки соотношений, инвариантность которых не нарушается при циклическом выполнении оператора.

Я предлагаю отныне строго придерживаться разработки и реализации только интеллектуально-управляемых программ. Если кто-то опасается, что это ограничение столь строго, что мы не в состоянии жить с ним, я могу его уверить: класс интеллектуально-управляемых программ все еще остается достаточно обширным, чтобы в нем содержались многие весьма реалистичные программы для любых задач, имеющих алгоритмическое решение. Не следует забывать, что производство программ - это не наше дело, наше дело - разработка классов вычислений, которые демонстрируют желаемое поведение. Предложение ограничиться интеллектуально управляемыми программами - основа первых двух из шести объявленных аргументов.

Первый аргумент: так как программисту необходимо рассматривать только интеллектуально-управляемые программы, с выбором альтернатив гораздо легче справиться.

Второй аргумент: как только мы решили ограничиться подмножеством интеллектуально-управляемых программ, мы раз и навсегда достигли значительного сжатия пространства рассматриваемых решений. И этот аргумент отличен от первого.

Третий аргумент основывается на конструктивном подходе к проблеме корректности программ. Сегодня обычная технология - сделать программу, а затем оттестировать ее. Но тестирование программы может весьма эффективно продемонстрировать наличие ошибок, но безнадежно неадекватно для демонстрации их отсутствия. Единственный эффективный способ значительно поднять уровень доверия к программам - предоставить убедительное доказательство их корректности. Но не следует сначала писать программу, а затем доказывать ее корректность, потому что тогда требование предоставить доказательство только увеличит обузу бедного программиста. Наоборот, программист должен позволять расти доказательству корректности и программе совместно. Третий аргумент в основном основывается на следующем наблюдении. Если кто-то сначала спросит себя, какова будет структура убедительного доказательства и, найдя ее, затем построит программу, удовлетворяющую требованиям этого доказательства, тогда эти заботы о корректности

обернутся весьма эффективным эвристическим указателем пути. По определению этот подход применим только в случае, если мы ограничиваемся интеллектуально-управляемыми программами, но он дает нам эффективные средства для поиска удовлетворительного решения среди них.

Четвертый аргумент относится к образу, которым количество интеллектуальных усилий, необходимых для разработки программы, зависит от длины программы. Было высказано предположение, что имеется некий закон природы, согласно которому количество необходимых интеллектуальных усилий растет как квадрат длины программы. Но, слава богу, никто не смог доказать справедливость этого закона. А не смог потому, что это не должно быть правдой. Все мы знаем, что единственный умственный инструмент, посредством которого весьма ограниченный разум может охватить мириады различных вариантов, называется "абстракцией"; в результате эффективное использование мощи абстракции должно расцениваться как одно из наиболее жизненно важных действий компетентного программиста. В этой связи было бы полезно упомянуть, что цель абстракции - не напустить тумана, а создать новый семантический уровень, на котором она будет абсолютно точна. Разумеется, я попытался найти фундаментальную причину, которая помешала бы нашим абстрактным механизмам быть достаточно эффективными. Но как бы я ни старался, я не мог найти эту причину. В результате я склоняюсь к мнению - до сих пор не опровергнутому практикой - что при надлежащем применении наших способностей к абстракции интеллектуальные усилия, необходимые для постижения или понимания программы, должны расти не более чем пропорционально длине программы. Но побочный продукт этих исследований может иметь еще большее практическое значение и фактически является основой моего четвертого аргумента. Этим побочным продуктом является признание нескольких шаблонов абстракции, которые играют жизненно-важную роль во всем процессе составления программ. Сейчас о таких шаблонах абстракции известно достаточно много для того, чтобы посвятить каждому из них отдельную лекцию. То, сколько знаний и навыков хранят в себе эти шаблоны абстракции, поразило меня, когда я осознал, что если бы они были общеизвестны пятнадцать лет назад, переход от BNF к синтаксически-управляемым компиляторам, например, занял бы несколько минут вместо нескольких лет. Поэтому я выдвигаю наши последние познания в области жизненно-важных абстракций как четвертый аргумент.

Теперь переходим к пятому аргументу. Он относится к влиянию инструментов, которые мы пытаемся использовать, на наш образ мышления. Я наблюдаю культурную традицию, которая, по всей вероятности, уходит корнями в эпоху Возрождения: игнорировать это влияние и рассматривать человеческую мысль как первичную и главенствующую над тем, что ей создано. Но когда я начинаю анализировать свои собственные мыслительные привычки и привычки моих друзей, я прихожу, нравится мне это или нет, к совершенно иному заключению, а именно: инструменты, которые мы пытаемся использовать, и язык и обозначения, которые мы используем для выражения или записи наших мыслей, являются основным фактором, который определяет, о чем мы вообще можем мыслить и что можем выразить! Анализ влияния, которое язык программирования оказывает на своих пользователей, и признание факта, что к настоящему времени мощь нашего мозга является наиболее скудным ресурсом, совместно дают нам новый набор эталонов для сравнения относительных достоинств различных языков программирования. Компетентный программист хорошо знает о том, что объем его черепной коробки крайне ограничен; поэтому он подходит к задаче программирования с предельным смирением, и помимо прочего он избегает заумных трюков как чумы. В случае хорошо известных диалоговых языков мне часто толковали со всех сторон, что как только сообщество программистов обзавелось терминалами, появился специфический феномен, который даже получил широко известное название - "однострочники". Он принимает одну из двух форм: один программист кладет перед другим однострочную программу и либо гордо рассказывает, что она делает, а затем спрашивает: "А ты можешь

закодировать это меньшим количеством символов?" - как будто это имеет какую-то практическую ценность, - либо просто спрашивает: "Угадай, как это работает!". Из этого наблюдения мы можем заключить, что в качестве инструмента этот язык является открытым вызовом для хитроумных трюков; и пока именно это может быть объяснением его привлекательности для некоторых, которым нравится демонстрировать свою гениальность, простите меня, но я вынужден рассматривать его как самую предосудительную вещь, насколько это вообще может относиться к языку программирования. Еще один урок, который мы должны извлечь из недавнего прошлого, - это то, что разработка "более богатых возможностями" или "более мощных" языков программирования была ошибкой в том смысле, что эта причудливая монструозность, это нагромождение идиосинкразий в действительности неуправляемы как механически, так и ментально. Я предвижу великое будущее для очень систематических и очень умеренных языков. Говоря об умеренности, я имею в виду, к примеру, что не только выражение "for" из ALGOL 60, но и цикл "DO" из FORTRAN'a могут оказаться выкинутыми за свою причудливость. Я провел небольшой программный эксперимент с действительно опытными добровольцами, подсунув им что-то достаточно неожиданное и непредвиденное. Ни один из моих добровольцев не нашел очевидного и самого элегантного решения. При более внимательном рассмотрении оказалось, что это имеет общий для всех источник: их представление о повторении было так тесно связано с идеей связанной управляющей переменной, что их рассудок был слишком зашорен, чтобы заметить очевидное. Их решения были менее эффективны, беспричинно слишком трудны для понимания, и им потребовалось слишком много времени на их поиск. Это был весьма поучительный, но в то же время обескураживающий опыт. Наконец, можно надеяться, что языки программирования завтрашнего дня будут значительно отличаться от тех, к которым мы привыкли сегодня: в значительно большей степени, чем до сих пор, они будут подталкивать нас к отражению в структуре того, что мы пишем, всех абстракций, необходимых для концептуального охвата сложности предмета нашей разработки. Это все, что касается большей адекватности наших будущих инструментов, что лежит в основе пятого аргумента.

К слову, мне хотелось бы вставить предупреждение для тех, кто отождествляет сложность задачи программирования с борьбой против неадекватности наших нынешних инструментов, потому что они могли бы заключить, что, как только наши инструменты станут более адекватными, программирование перестанет быть проблемой. Программирование останется весьма трудным, потому что как только мы избавимся от обременительных деталей, мы окажемся свободны для того, чтобы приняться за задачи, которые лежат пока за пределами наших возможностей.

Вы можете оспаривать мой шестой аргумент, потому что не так-то легко собрать экспериментальные свидетельства в его поддержку, но это не изменит моей веры в его истинность. До сих пор я не упоминал слово "иерархия", но я считаю, что справедливо сказать, что это ключевая концепция всех систем, основанных на хорошо структурированном решении. Я мог бы даже пойти на шаг дальше и написать статью в подтверждение этого, а именно: мы можем в действительности удовлетворительно решить только те задачи, которые окончательно допускают хорошо структурированное решение. На первый взгляд, такая точка зрения на свойственные человеку ограничения может поразить вас своим пессимизмом по поводу свойственных нам затруднений, но я так не считаю, совсем наоборот! Лучший способ научиться жить, несмотря на наши ограничения, - это знать их. К тому времени, когда мы станем достаточно скромны, чтобы пытаться находить только структурированные решения, потому что другие усилия нам попросту не по зубам, мы будем делать все, что в наших силах, чтобы избежать всех интерфейсов, идущих во вред нашей способности структурировать систему подходящим образом. И я могу только надеяться, что это будет раз за разом приводить к открытию, что изначально неподдающаяся задача в конце концов может быть структурирована. Любой,

кто видел, как большинство проблем фазы компиляции, называемой "генерацией кода", может быть локализовано благодаря забавным особенностям машинного кода, знает простой пример того, что я имею в виду. Широкое применение хорошо структурированных решений - мой шестой и последний аргумент в пользу технической возможности революции, которая может разразиться в текущем десятилетии.

В принципе, я оставляю на ваше усмотрение решать для себя, насколько серьезно отнестись к моим доводам, хорошо зная, что я никого не могу заставить насильно разделить мои надежды. Как и каждая серьезная революция, она вызовет серьезное сопротивление, и каждый может спросить себя, где ожидать появления консервативных сил, противодействующих такому развитию. Я не ожидаю их появления в большом бизнесе и даже в компьютерном бизнесе; я скорее ожидаю их в образовательных учреждениях, которые сегодня занимаются обучением, и в консервативных группах пользователей компьютеров, которые считают свои старые программы столь важными, что не находят целесообразным переписать и улучшить их. В этой связи грустно наблюдать, как в некоторых университетских городках выбор центрального компьютера слишком часто определяется потребностями немногих, но дорогих приложений, игнорируя вопрос, сколько тысяч "малых пользователей", желающих писать собственные программы, пострадают от этого выбора. Например, слишком часто кажется, что ученые, занимающиеся физикой высоких энергий, шантажируют научное сообщество стоимостью экспериментального оборудования. Самый простой ответ, конечно, - это категорическое отрицание технической осуществимости, но боюсь, что вам понадобятся весьма состоятельные доводы для этого. Увы, замечание, что интеллектуальный потолок нынешнего среднего программиста не даст революции свершиться, не внушает уверенности: когда другие начнут программировать более эффективно, он непременно выйдет за эти рамки.

Могут быть также препятствия политического характера. Если даже мы знаем, как учить завтрашних профессиональных программистов, неочевидно, что общество, в котором мы живем, позволит нам делать это. Первый эффект от обучения методологии - а скорее, распространения знаний - состоит в том, что, расширяя возможности уже возможного в данный момент, тем самым усиливается различие в интеллектуальных способностях. В обществе, где система образования используется как инструмент для создания однородной культуры, в котором сливкам не дают подниматься к вершине, обучение компетентных программистов может оказаться политически некорректным.

Позвольте мне перейти к заключению. Автоматические компьютеры были с нами уже четверть века. Они оказали огромное влияние на наше общество как могучие инструменты, но при всей этой мощности их влияние будет только рябью на поверхности нашей культуры по сравнению с гораздо более глубоким влиянием, которое они окажут своей возможностью бросить беспрецедентный интеллектуальный вызов за всю культурную историю человечества. Похоже, что иерархические системы обладают тем свойством, что нечто, рассматриваемое как неделимое целое на одном уровне, рассматривается как составной объект на следующем, более низком уровне с большей детализацией; в результате дискретность пространства или времени, применимая на каждом уровне, уменьшается а порядок величины при переходе от одного уровня к другому, следующему за ним более низкому. Мы воспринимаем стену через понятие кирпичей, кирпичи - через понятие кристаллов, кристаллы - через понятие молекул и так далее. В результате количество уровней, которые могут быть осмысленно выделены в иерархической системе, в некотором роде пропорционально логарифму отношения между наибольшим и наименьшим дискретами, и поэтому, если только это соотношение не чрезмерно велико, мы можем ожидать появления не слишком большого числа уровней. В области компьютерного программирования наш базовый строительный блок имеет дискретность времени менее микросекунды, но наша программа может потребовать несколько часов вычислений. Я не знаю никакой другой технологии, перекрывающей

отношение 1010 и более: компьютер, благодаря его фантастической скорости, кажется, впервые предоставил нам среду, в которой артефакты с высокой степенью иерархичности одновременно и возможны, и необходимы. Этот вызов, а именно противостояние задаче программирования, столь уникален, что новый опыт может рассказать нам очень много нового о нас самих. Он может углубить наше понимание процессов разработки и созидания, он может дать нам лучший контроль над организацией нашего мышления. Если бы он не сделал этого, на мой взгляд, мы вообще не заслуживаем компьютеров!

Он уже преподал нам несколько уроков, и один из них, который я выбрал, чтобы акцентировать внимание в этом докладе, заключается в следующем. Мы будем программировать гораздо лучше, если только подойдем к задаче, полностью оценивая ее потрясающую сложность, если только мы будем придерживаться скромных и элегантных языков программирования, если только мы примем во внимание свойственные человеческому разуму ограничения и подойдем к задаче как Очень Смиренные Программисты.

Два взгляда на программирование

EWD540

В окружающем нас мире мы можем встретить два радикально противоположных взгляда на программирование:

- Взгляд A: Программирование в основном весьма просто.
- Взгляд B: Программирование – это очень сложно.

Это противоречие можно объяснить тем, что в этих двух взглядах одно и то же слово «программирование» употребляется в двух совершенно различных значениях, и на этом успокоиться. Тем не менее то, какой из взглядов преобладает, A или B, оказывает глубокое влияние не только на кадровую политику организаций, использующих компьютеры, и учебные программы высших учебных заведений, но даже и на направление развития и исследований в самой компьютерной науке. Таким образом, представляется важным исследовать природу различий между этими двумя смыслами и по возможности выявить базовые предположения, которые делают каждый их них применимым. В этом и есть назначение данного документа.

В этом исследовании у меня есть одно препятствие: в этой дискуссии я не являюсь нейтральной стороной. Я – убежденный сторонник взгляда B и рассматриваю взгляд A как основную причину многих печальных заблуждений. С другой стороны, я не считаю, что наличие собственного мнения дисквалифицирует меня как автора, особенно если я заранее предупрежу об этом своих читателей и не буду притворяться нейтральным. В процессе анализа мы раскроем, как эти различные взгляды на программирование (которое является человеческой деятельностью!) связаны с различными мнениями Человека. Это уже само по себе является весьма ценным пониманием, так как объясняет почти религиозное рвение, с которым разворачиваются сражения между сторонниками противоположных взглядов (догм?).

Ранняя история автоматических вычислений делает взгляд A очень понятным. До того, как у нас появились компьютеры, программирование вообще не являлось проблемой. Затем появились первые машины: по сравнению с нашими нынешними машинами они были просто игрушками, и по сравнению с тем, что мы пытаемся делать сейчас, они использовались лишь для «микро-приложений». Если на этом этапе программирование и было проблемой, то весьма незначительной. Добавьте к этому источники трудностей, которые в то время поглощали – или лучше сказать узурпировали? – большую часть нашего внимания:

1. Арифметические устройства были слишком медленные по отношению к тому, что мы хотели делать с их помощью: эти башмаки почти всегда оказывались слишком тесными, и ради эффективности программы допускались все возможные трюки кодирования (и очень немногие из них реально не использовались).
2. Разработка и конструирование арифметических устройств были настолько новой и, следовательно, трудной задачей, что, если очередная аномалия в коде инструкции могла избавить от каких-либо кульбитов, от них обычно избавлялись, – также, разумеется, потому что мы имели так мало опыта в программировании, что мы не могли так уж хорошо распознавать «аномалии в программном коде»; в результате, помимо необходимости использования трюков в коде, также была великолепная возможность их применять.
3. Памяти всегда было слишком мало, и это вместе с ненадежностью первого оборудования препятствовало более разумному использованию машин.

В это время программирование представлялось в первую очередь как битва с ограничениями машины, битва, которую нужно было выиграть хитростью. Это было

систематическое использование специфических особенностей каждой машины: это был расцвет виртуозного кодирования.

В течение следующих десяти-пятнадцати лет процессоры стали в тысячи раз быстрее, памяти стало в тысячи раз больше, и языки программирования высокого уровня вошли в обиход. И именно в это время с одной стороны программирование все еще прочно ассоциировалось с тесными башмаками, в то время как с другой – чувствовалось, что башмаки жмут все меньше и меньше, и ожидалось, что еще через пять лет технического прогресса проблемы программирования вовсе исчезнут. Именно в этот период появился взгляд А. Именно в конце этого периода, вдохновленный взглядом А, был разработан COBOL с провозглашенным намерением, что он должен сделать программирование, выполняемое профессиональными программистами, ненужным, позволив «пользователю» (не в это ли время слово «пользователь» стало общеупотребимым?) записывать то, что он хочет, на «простом английском», который любой может прочесть и понять.

Все мы знаем, что эта прекрасная мечта не воплотилась в жизнь. Следующие пять лет принесли нам вместо исчезновения всех проблем, связанных с программированием, кризис программного обеспечения, и COBOL, вместо того чтобы выжить профессиональных программистов, стал грандиозным механизмом программирования для еще большего их числа; и еще десять лет спустя мы все еще имеем машины, в которых ошибки базового программного обеспечения вызывают в среднем час простоя на каждые пятнадцать часов работы. Очевидно, что до сих пор остались серьезные проблемы программирования…

Забавная вещь: несмотря на полную очевидность обратного, взгляд А все еще жив. В объяснение этого странного факта некоторые с укоризной указывают пальцем на большие организации: либо на организации, применяющие компьютеры, которые, привлекая большие трудовые ресурсы, разделяющие взгляд А, тем самым потеряли возможность свободно расстаться с ним, либо на фирмы-производители компьютеров и образовательные институты, которые поддерживают широко распространенный взгляд А, который им полагается представлять как основной для их рынка. Даже если этот палец грозит поделом, тем не менее я не могу принять это как полное объяснение живучести взгляда А, и вынужден предположить, что взгляд А удовлетворяет более глубокие, физиологические потребности.

Откуда взялся взгляд В? Были люди, которые чувствовали, что появление больших и быстрых машин заменит тесные башмаки хотя бы на башмаки по размеру, и что несмотря на это эффективность выполнения программ останется серьезной заботой программиста, заботой, которая станет даже более важной по мере роста машин и приложений, и что более сложные установки поставят более трудные проблемы. Также было замечено, что переключение с машинного кода на языки высокого уровня вовсе не гарантирует тех преимуществ, на которые возлагали столько надежд. В частности, программисты продолжали столь же охотно выдавать большие куски непонятного кода, и единственное различие было в том, что теперь они делали это в более грандиозных масштабах, и что высокоуровневые ошибки пришли на смену низкоуровневым. Они также поняли, что появление языков программирования высокого уровня не уменьшает потребности в тщательности: избыточность языков высокого уровня лишь уменьшает вредный эффект от некоторых видов небрежности. И тогда появился взгляд В. (Взгляд В не является реакцией на кризис программного обеспечения, который всплыл в 1968, так как он на много лет старше. Фактически взгляд В предсказал этот кризис, но даже это подтверждение не убило взгляд А).

После этой интерлюдии по поводу появления взгляда В вернемся к нашему вопросу: как и почему, лицом к лицу с признанными проблемами программного обеспечения, взгляд А (а именно – программирование в основном весьма просто) все еще здравствует. Вот ответ: из-за веры, причем не веры в лучших программистов, а веры в лучшие языки

программирования или (диалоговые?) системы программирования, а также веры в лучшие технологии программного менеджмента.

Я придерживаюсь мнения, что программирование – один из наиболее сложных разделов прикладной математики, поскольку оно также является одним из наиболее сложных направлений инженерии, и наоборот. Когда я попытался разъяснить одному из моих коллег-математиков, почему я придерживаюсь этого мнения, он довольно бесцеремонно отказался выслушать мои доводы и вместо этого обвинил меня и моих единомышленников-компьютерщиков в том, что мы до сих пор не создали язык программирования, который сделал бы программирование настолько простым, насколько ему и подобает быть! Возможно, мне стоило бы спросить его, почему математики до сих пор не разработали нотацию, которая позволила бы любому, невзирая на отсутствие профессиональной подготовки, заниматься математикой?

Копнув чуть глубже, выясняется, что сторонники взгляда А не отрицают потенциальной сложности программ и их разработки, но верят, что жизнь программистов будет становиться все легче, поскольку все наиболее сложные части задачи будет брать на себя машина. Они указывают на появление языков программирования высокого уровня, которые уже сделали программирование гораздо легче, чем во времена старых машин, и опрометчиво экстраполируют, что в будущем программирование станет вовсе тривиальным. Но оправдана ли эта экстраполяция? Я много программировал, как в машинных кодах, так и на языках высокого уровня, и последние несомненно более удобны, поскольку в этом случае многие решения, относящиеся к внутренним деталям программы, такие, как распределение памяти, не приходится принимать явно, поскольку ими занимается алгоритм распределения памяти компилятора. Переход к языкам высокого уровня освобождает нас от многих тривиальных забот. Это сделало программирование деятельностью с небольшим количеством нудной работы, с преобладанием изобретательности: именно та часть работы, которая занимала целые дни, исчезла! Вывод, который следует из появления языков программирования высокого уровня, о потребности в программистах большего интеллектуального калибра, полностью подтвердился моими наблюдениями в Западной Европе (где я мог следить за разработками последнее время): в конце 60-х многие крупные организации, использующие компьютеры, испытывали проблемы в подборе подходящей работы для программистов, нанятых 50-е годы, поскольку профессия переросла их интеллектуальные способности.

Однако ни эти наблюдения, ни указания на провал попытки COBOL'a выжить профессиональных программистов не произвели никакого впечатления на правоверных. Они объяснят, что традиционные языки программирования высокого уровня потерпели крах из-за их «процедурности», и что провал COBOL'a очевиден, поскольку ввиду недостаточной интерактивности он на самом деле не является простым английским, но вот через пять или десять лет дальнейший прогресс в области Искусственного Интеллекта (для посвященных – AI, т.е. Artificial Intellect) позволит нам построить «контекстно-зависимые», «основанные на знаниях», «автоматизированные системы для мышления и понимания», такие, что «пользователю достаточно будет только побеседовать с ними».

Должно быть, я неизлечимый скептик, но мне весьма трудно поверить, что подобным надеждам суждено сбыться. Имеются некоторые видимые подтверждения таких ожиданий – я цитирую отрывок из недавно полученного письма: «…в общем, передача более совершенному компьютеру того, что мы сегодня называем человеческими навыками, знанием и разумом». Я не намерен повторять здесь фрагменты горячих дискуссий, которые мы проводили по поводу значимости Искусственного Интеллекта, да здесь это и ни к чему.

Во-первых, оглядываясь назад, приходим к неизбежному заключению: смешивание надежды на будущее AI с завтрашней реальностью было бы безрассудством, и весьма безответственно быть неподготовленными на случай, если мечты по поводу AI останутся

мечтами по крайней мере на протяжение нашей жизни. Или, говоря другими словами, глядя на серьезность сегодняшних проблем программирования, обычная осторожность заставляет нас не забывать о взгляде B.

Во-вторых, первоочередной задачей программиста, если он хочет, чтобы его творения заслуживали доверия, будет разрабатывать их настолько понятными, что он сможет нести за них ответственность, и, несмотря на ответ на вопрос, как много из его нынешней деятельности может быть переложено на машину, мы должны всегда помнить, что ни «понимание», ни «ответственность» не могут быть классифицированы как деятельность: это скорее «состояние разума» и в принципе не может быть передано машине.

Я считаю неразумным, в особенности для ученого-компьютерщика, недооценивать влияние психологической школы, которая, считая человеческий разум слишком сложным и трудно поддающимся изучению, занялись вместо этого изучением крыс, и даже ограничивает это изучение – как я слышал недавнюю формулировку – «наиболее механической формой поведения – зачастую настолько механической, что даже крысам не дают проявить свои высшие возможности». Представляя свои грубые, механические модели в качестве допустимого приближения к человеческому разуму, они опасно затуманили различие между человеком и машиной, и мы наблюдаем два взаимно дополняющих друг друга феномена: антропоморфный взгляд на машины и механический взгляд на людей.

Эта неразбериха, вне всякого сомнения, – плод усилий верховных жрецов AI. Преобладание в основном антропоморфной терминологии в компьютерной науке – «память», «интерпретатор», «язык программирования», «рукопожатие», «диалог», и это лишь немногие примеры, – это предупреждение, которое не следует игнорировать. Я не знаю, как думать и разговаривать, обходясь без метафор; я знаю также, что каждая метафора несет в себе опасность скрытого подтекста. В данном случае антропоморфной терминологии в компьютерной науке мы давно уже достигли стадии, когда опасность путаницы перевешивает достоинства аналогии.

Кроме того, кажется, что механический подход к людям среди компьютерных ученых (и их руководителей) распространен шире, чем представляется мне нормальным. Ибо я подозреваю, что именно этот механический подход ограничивает деятельность программистов механическими действиями по написанию кода, и затем измеряет «производительность труда программиста» количеством произведенных им строк кода. (Когда весьма широко известный и очень уважаемый ученый-компьютерщик использовал недавно эту меру производительности труда программиста в своей лекции, от слушателей поступило предложение говорить не о «количестве произведенных строк кода», а о «количестве израсходованных строк кода», и что лектор, следовательно, занес их не в ту графу учета баланса расходов и доходов. Лектор ответил, что он вынужден использовать эту меру производительности, поскольку не располагает никакой альтернативной, которая позволяет вести точный учет!) Это не может больше рассматриваться как безобидное заблуждение, поскольку принятие этой бессмысленной «меры производительности» профессиональными программистами гарантированно стимулирует их к написанию рыхлого кода.

Влияние психологии было рассмотрено здесь, поскольку оно объясняет цепкость, с которой так много людей склонны тяготеть ко взгляду A.

В основном это не вина производителей компьютеров, которые желают вести дела так, будто они продают простейшую продукцию; и не вина руководителей программных проектов, которые предпочитают рассматривать деятельность программистов как простой и предсказуемый процесс; и не вина учебных заведений, которые хотели бы подготовить студентов к достижению гарантированного успеха.

Это – следствие комфортной иллюзии, что Человек – это лишь сложный автомат, которая, подобно наркотику, приносит своим жертвам кажущееся освобождение от

бремени ответственности. Признание программирования серьезным вызовом интеллекту вернуло бы полный вес этой ноши обратно на их плечи.

prof. dr. Edsger W. Dijkstra

Burroughs Research Fellow

Почему программное обеспечение такое дорогое?

EWD648

Пояснение для разработчиков аппаратуры.

Недавно я получил приглашение от одной солидной (и продолжающей расти) компании, производящей вычислительное оборудование. В течение многих лет основной их продукцией было высококачественное аналоговое оборудование; впоследствии, однако, цифровые компоненты начали играть все более значительную роль. Руководство корпорации осознало, что так или иначе компании неизбежно придется столкнуться с областью программного обеспечения (ранее неизведанной для нее), а также осведомлено о существовании множества ловушек в этой области (не имея, впрочем, ясного понимания, что же именно они собой представляют). Меня пригласили объяснить руководству компании, что же такое эта самая разработка программного обеспечения, почему оно такое дорогое и т.д.

Имея массу других обязательств, я пока еще не знаю, смогу ли принять приглашение, но в любом случае я очень рад принять этот вызов. Я не просто занимался программированием более 25 лет, но и с самого начала по сей день я делал это в сотрудничестве, порой весьма тесном, с разработчиками оборудования, конструкторами машин, тестерами прототипов и т.п. Я думаю, что достаточно хорошо знаком со средним разработчиком оборудования и его проблемами, чтобы понять, почему он не в состоянии оценить всю сложность разработки программного обеспечения. Объяснить ему сложность разработки программного обеспечения достаточно трудно – почти так же трудно, как и объяснить это чистому математику, – объяснить же это группе разработчиков, с их профессиональной подготовкой и профессиональной гордостью за высококачественное аналоговое оборудование, добавляет брошенному вызову особую остроту! Размышляя о том, как принять его, и понимая, что, если даже я приму приглашение, пригласившая сторона все равно не будет обладать исключительными правами на мое объяснение, я решил взять перо и бумагу. В результате появился этот текст.

На экономический вопрос «Почему программное обеспечение такое дорогое» столь же экономическим ответом был бы такой: «Потому что его пытаются получить при помощи дешевого труда». А почему пытаются? Да потому, что присущие ему трудности повсеместно сильно недооцениваются. Поэтому давайте зададимся вопросом: «Почему разрабатывать программное обеспечение столь трудно?». Один из моих выводов – с недостаточно подготовленным персоналом это невозможно; с достаточно подготовленными разработчиками программного обеспечения это возможно, но все же весьма трудно. Мне хотелось бы подчеркнуть с самого начала, что нынешние проблемы в разработке программного обеспечения только частично могут быть объяснены явным недостатком компетентности вовлеченных в нее программистов. Я делаю это в самом начале, поскольку это объяснение, хотя и не является чем-то необычным, все же слишком простое.

Вполне естественно, что для руководителя аппаратной части весьма досадно произвести то, что мы справедливо оцениваем как надежную машину с превосходным соотношением цена/производительность, и впоследствии наблюдать, как к моменту отправки полной системы потребителю оказывается, что эта самая система нашпигована ошибками, а ее производительность упала ниже, чем в самых кошмарных снах разработчиков. И мало того, что он обязан стерпеть, что парни из программного отдела сгубили его детище. От него еще вдобавок требуют смириться с тем, что по мере того как он работает все эффективнее год от года, группу программного обеспечения награждают за ее некомпетентность увеличивающимся каждый год бюджетом. Без дальнейших объяснений с нашей стороны мы, программисты, должны простить ему периодически посещающую его обиду, ибо, обвиняя нас в некомпетентности, сам он грешит

невежеством… И пока мы не в состоянии ясно объяснить природу наших проблем, мы не вправе винить его в этом невежестве!

Сравнение между миром аппаратуры и миром программного обеспечения кажется хорошим введением для разработчика аппаратуры в проблемы его коллеги-программиста.

Разработчик аппаратуры вынужден симулировать дискретную машину преимущественно аналоговыми средствами. В результате он должен думать о задержках и искажениях сигналов, притоке и отводе воздуха, синхронизации, рассеивании тепла, охлаждении и источнике питания и подобных проблемах технологии и производства. Использование в качестве строительного материала преимущественно аналоговых компонентов подразумевает, что «допуски» являются весьма существенным аспектом спецификаций его компонентов; его контроль качества имеет в основном статистическую природу, и в результате гарантия качества – в основном вероятностное утверждение. Тот факт, что при нынешних стандартах качества вероятность правильного функционирования весьма высока, не должен позволять нам расслабиться и забыть о ее вероятностной природе: не следует путать очень высокую вероятность с полной уверенностью (в математическом смысле), поэтому вполне естественно, что ни один блок оборудования не поставляется, не будучи проверенным согласно программе тестирования. По мере того как технология подходит все ближе к своим пределам – а это происходит постоянно – и допуски становятся все более жесткими, контроль за допусками становится основной заботой производителей аппаратуры.

По сравнению с разработчиком аппаратуры, который постоянно борется с непокорной природой, разработчик программного обеспечения живет в раю, так как он строит свои творения только из нулей и единиц. Нуль есть нуль и единица есть единица: в его строительных блоках нет места нечеткости, и общее инженерное понятие о чем-то, «находящемся в пределах допусков», здесь просто неприменимо. В этом смысле программист и в самом деле работает в райской среде. Гипотетический стопроцентный разработчик электрических схем, который привык сводить проблемы разработки и реализации к проблемам контроля над допусками должен быть слеп к проблемам программирования: раз уж он сумел корректно симулировать дискретную машину, все действительно сложные проблемы уже решены, не так ли?

Чтобы разъяснить аппаратному миру, почему программирование все же представляет проблему, мы должны привлечь внимание к некоторым другим различиям. В самых общих словах мы можем рассматривать «разработку» как построение моста через пропасть, как построение сущности из заданных компонентов; до тех пор, пока «целевая сущность» и «исходные компоненты» остаются неизменными, мы можем повторно использовать старую разработку. На самом же деле мы вынуждены непрерывно заниматься разработкой, поскольку они все же меняются. Впрочем, здесь разработчики аппаратуры и программного обеспечения сталкиваются с различными, почти противоположными типами разнообразия, изменения и отличия.

Для разработчика аппаратуры наибольшее разнообразие кроется в «исходных компонентах»: в течение разработки машин ему приходилось быть постоянно в курсе новых технологий, ему постоянно недостает времени на то, чтобы полностью ознакомиться со своим исходным материалом, потому что прежде чем он достигает этой стадии, на сцену выходят новые компоненты, новые технологии. По сравнению с огромным разнообразием «исходных компонентов», его «целевая сущность» остается практически неизменной: он все время разрабатывает заново одни и те же несколько машин.

Для программиста изменение и разнообразие находятся по другую сторону: то, что для разработчика аппаратуры является целью, для программиста – начальная точка. «Исходные компоненты» программиста оставались поразительно стабильными – по мнению некоторых, даже гнетуще стабильными, - FORTRAN и COBOL, столь модные до сих пор, имеют возраст более четверти столетия! Программист находит разнообразие на

другой стороне пропасти, через которую строит мост: он сталкивается с целым набором самых разнообразных «целевых сущностей». Даже очень разнообразных; даже в основном весьма разнообразных, потому что здесь мы находим отражение того факта, что нынешнее оборудование и в самом деле заслуживает название «аппаратура общего назначения».

В течение последнего десятилетия разработчики программного обеспечения продолжали вести почти религиозные споры о противостоянии методов разработки «снизу-вверх» и «сверху-вниз». Хотя ранее привычной была методика «снизу-вверх», я полагаю, что религиозное течение «сверху-вниз» теперь привлекло большинство сторонников. Если мы придерживаемся строгого принципа, столкнувшись с многогранной проблемой, сначала исследовать наиболее неизведанную область (так как решение знакомых проблем может быть отложено с меньшим риском), то мы можем интерпретировать переход программистского сообщества от метода «снизу-вверх» к методу «сверху-вниз» как медленное осознание того обстоятельства, что наибольшее разнообразие ожидает программиста на другой стороне пропасти.

Помимо, что изменения и разнообразие, с которыми сталкивается программист, находятся на другой стороне пропасти, через которую строится мост, они еще и более расплывчаты. Для понимания своих исходных компонентов разработчик аппаратуры всегда может опереться на физику и электронику в качестве последнего прибежища; для понимания целевых проблем и разработки решающих их алгоритмов разработчик обнаруживает, что подходящая теория чаще всего отсутствует. Впрочем, понимание того, насколько может тормозить работу отсутствие адекватной теории, понемногу начинает приходить.

С первыми приложениями машин, которые были научно-техническими, не было подобных затруднений: решаемые проблемы были совершенно понятны с научной точки зрения, и математики-вычислители были способны предоставить алгоритмы и их проверку. Вспомогательное кодирование, вроде преобразований между десятичной и двоичной системами и загрузчиков программ, было столь тривиальным, что здравого смысла было вполне достаточно.

С тех пор мы многократно видели, что ввиду недостатка подходящей теории за решение задач все чаще брались исходя из соображений здравого смысла, и одного только здравого смысла оказывалось недостаточно. Первые компиляторы были сделаны в пятидесятых годах без сколь-либо приличной теории для определения языка, синтаксического разбора и т.д., и они изобиловали ошибками. Теория синтаксического разбора и подобные ей появились позже. Первые операционные системы делались без должного понимания синхронизации, «смертельных объятий» и т.п. опасностей, и они также страдали от дефектов, которые, глядя в прошлое с нынешних позиций, можно было бы предсказать. И опять необходимая теория появилась позже.

Понятно, что люди убедились на практике, что для некоторых проблем один только здравый смысл не является достаточным мыслительным инструментом. Проблема состоит в том, что к тому времени, как необходимые теории были разработаны, донаучный, интуитивный подход уже успел закрепиться, и несмотря на его явную недостаточность, его теперь трудно искоренить. Здесь я должен привести критический комментарий касательно управленческой практики, достаточно типичной среди производителей компьютеров, а именно – выбирать в качестве руководителя проекта кого-то, имеющего практический опыт предшествующих сходных проектов: если за предыдущий проект взялись с донаучными технологиями, скорее всего та же судьба постигнет и новый проект, даже если к этому времени уже доступна подходящая теория.

Второй вывод из такого положения дел: одной из наиболее важных способностей разработчика программного обеспечения, столкнувшегося с новой задачей, является способность оценить, достаточно ли для ее решения существующей теории и здравого смысла, либо сначала необходимо разработать какую-то новую интеллектуальную

дисциплину. В последнем случае чрезвычайно важно не впрягаться в кодирование, не располагая такой теорией. Прежде всего – думать! Я еще вернусь к этой теме впоследствии, при рассмотрении выводов для руководства.

Позвольте мне теперь поделиться с вами своими соображениями по поводу того, какой образ мышления является необходимым.

Так как IBM присвоила себе термин «структурное программирование», сам я больше им не пользуюсь, но я читал лекции по этому предмету в MIT в конце шестидесятых. Главной моей идеей было то, что (большие) программы – это объекты, не имеющие прецедентов в нашей культурной истории, и что наиболее близкой аналогией, которую я могу привести, является математическая теория. Я иллюстрировал это аналогией между леммой и подпрограммой: лемма доказывается независимо от того, каким образом ее намереваются использовать, и используется независимо от того, каким образом она доказана; подобным образом подпрограмма реализуется независимо от того, каким образом ее намереваются использовать, и используется независимо от того, каким образом она реализована. Обе они являются примерами правила «разделяй и властвуй»: математическое доказательство разделяется на теоремы и леммы, программа аналогичным образом делится на процессы, подпрограммы, кластеры и т.п.

Более того, я знаю, что аналогия простирается дальше, на способы, которыми разрабатываются математические теории и программы. Недавно я слышал, как Дана Скотт описывала разработку математической теории как экспериментальную науку, экспериментальную в том смысле, что адекватность и применимость новых нотаций и концепций определяется экспериментально, в попытках их использовать. Это весьма похоже на то, каким образом команда разработчиков пытается справиться с концептуальной проблемой, с которой они столкнулись.

Когда разработка завершена, ее можно глубокомысленно обсуждать, но окончательная разработка может иметь структуру, никогда ранее не обсуждаемую. Поэтому команда разработчиков должна изобретать свой собственный язык, чтобы обсуждать ее, должна найти проясняющие дело концепции и придумать для них подходящие имена. но они не могут ждать с этим до окончания разработки, поскольку язык им нужен для самой разработки! Это старая проблема курицы и яйца. Я знаю только один способ разорвать этот порочный круг: придумать язык, который кажется вам необходимым, что-то достаточно свободное, если вы не уверены полностью, и проверить его адекватность на деле, пытаясь применить его, поскольку новые слова обретут смысл при их использовании.

Позвольте привести пример. В первой половине шестидесятых я разрабатывал как часть мультипрограммной системы подсистему, назначением которой было абстрагироваться от различий между первичной и вторичной памятью: единица, которой обменивались между собой различные уровни памяти, называлась «страницей». когда мы изучили свою первую разработку, оказалось, что такой подход годится лишь в первом приближении, т.к. соображения эффективности вынуждали нас придать подмножеству страниц в первичной памяти особый статус. Мы назвали их «священные страницы», поскольку по идее гарантированное присутствие священных страниц в первичной памяти ускоряло доступ к ним. Было ли это хорошей идеей? Нам пришлось определить «священные страницы» таким образом, чтобы мы могли убедиться, что их количество фиксировано. В конце концов мы пришли к очень точному определению, какие страницы должны быть священными, которые удовлетворяли всем нашим требованиям логики и эффективности, но в ходе обсуждений понятие «священная» понемногу превращалось в нечто точное и полезное. Например, первоначально, помнится, «священность» была булевским атрибутом: страница либо была священной, либо нет. Постепенно оказалось, что страницы должны иметь «счетчик святости», а первоначальный булевский атрибут стал отвечать на вопрос, положителен счетчик святости или нет.

Если бы во время этих дискуссий кто-то посторонний вошел в нашу комнату и послушал нас пятнадцать минут, он сказал бы: «Не верится, что вы сами знаете, о чем говорите». Наш ответ был бы таким: «Да, вы правы, и именно об этом мы и говорим: мы пытаемся выяснить, о чем именно нам следует говорить».

Я описал эту сцену столь подробно, поскольку хорошо помню ее и потому что считаю ее достаточно типичной. Постепенно вы достигаете весьма формального и хорошо определенного результата, но этому постепенному зарождению предшествует период созревания, в течение которого новые идеи опробуются и либо отбрасываются, либо развиваются. Это единственный известный мне способ, которым разум может справиться с подобными концептуальными проблемами. Из опыта я уяснил, что в этот период созревания, когда должен быть создан новый жаргон, блестящее владение родным языком является абсолютным требованием для всех участников. Программист, изъясняющийся неряшливым языком, - это просто бедствие. Блестящее владение родным языком – это мой первый критерий отбора будущих программистов; хороший математический вкус – второй важный критерий. (К счастью, они часто сопутствуют друг другу).

У меня есть и третья причина столь подробно описывать рождение понятия «священности». Через несколько лет я узнал, что это не просто романтика, не просто приятные воспоминания о проекте, который всем нам нравился: наш эксперимент попал в самую точку. Я понял это, когда пожелал в качестве упражнения для себя самого дать полную формальную разработку рекурсивного парсера для простого языка программирования, заданного в терминах пяти или шести синтаксических категорий. Единственным способом, которым я смог достичь корректного формального подхода, было введение новых синтаксических категорий! Эти новые синтаксические категории описывали последовательности символов, которые были бессмысленными с точки зрения разбираемого языка, но необходимыми для понимания и проверки алгоритма разбора при разработке. Мое формальное упражнение многое прояснило не потому, что привело к созданию хорошего парсера, а потому, что в краткой, компактной форме продемонстрировало необходимость изобретений, которые требуются при разработке программного обеспечения: новые синтаксические категории были образцом понятий, которые постоянно приходится изобретать, понятий, которые не имеют смысла с точки зрения начальной постановки задачи, но необходимы для понимания решения.

Я надеюсь, вышесказанное дает вам некоторое понимание задачи, стоящей перед программистом. Коснувшись проблем разработки программного обеспечения, я должен также посвятить пару слов феномену плохого управления ей. Как это ни прискорбно, но плохие менеджеры в этой области есть и, что хуже того, имеют достаточно власти, чтобы провалить проект. Я читал лекции по всему миру программистам, работающим в организациях всевозможного рода, и у меня осталось ошеломляющее впечатление от дискуссий с ними, что плохой руководитель над программистами – почти повсеместное явление: одной из обычных реакций аудитории при дискуссии после лекции было «Как жаль, что здесь не было нашего руководителя! Мы не можем объяснить ему этого, но к вам, возможно, он бы прислушался. Мы бы с удовольствием работали так, как вы описали, но наш начальник, который не понимает этого, не позволит нам этого». Я так часто встречал такую реакцию, что могу заключить только, что в среднем ситуация сложилась действительно неважная. (С самым худшим случаем я столкнулся в банке, и некоторые правительственные организации оказались не лучше).

По поводу плохих руководителей я часто описывал мой опыт как лектора в IBM, Hursley, потому что он был таким показательным. Прямо перед моим приходом оформитель переоформил аудиторию, и в ходе переоформления он заменил старомодную доску на экран и проектор. В результате мне пришлось выступать в тускло освещенной комнате в солнечных очках, чтобы не ослепнуть полностью. Я мог видеть только людей в первых рядах.

Эта лекция была одним из худших экспериментов в моей жизни. Несколькими хорошо подобранными примерами я иллюстрировал технику решения проблем, которую мог сформулировать в то время как свободу разработчика с одной стороны и формальную дисциплин, необходимую для его контроля, с другой. Но видимая часть аудитории была совершенно безответной: я чувствовал себя, словно обращаюсь к фигуркам, вылепленным из жевательной резинки. Это было для меня настоящей пыткой, но я знал, что это была хорошая лекция, и с решимостью обреченного довел свое выступление до самого горестного конца.

Когда а закончил и загорелся свет, я был поражен шквалом аплодисментов… из задних рядов, которые были для меня невидимы! Как оказалось, мне досталась весьма разнородная аудитория, восхищенные программисты в задних рядах, а на передних – их начальники, которые были чрезвычайно раздражены моим выступлением: явно указав количество используемых «изобретений», я представил задачу программирования еще более «неуправляемой», чем они уже боялись. С их точки зрения, я оказал им медвежью услугу. Из этого случая я сделал для себя вывод, что плохие руководители программных проектов видят в программировании прежде всего управленческую проблему, потому что они не знают, как им управлять.

Эти проблемы не столь заметны в тех организациях – я знаю несколько таких фирм, – в которых руководство состоит из компетентных, опытных программистов (а не из банкиров с колониальным опытом, еще слишком молодых, чтобы уйти на пенсию). Одна из проблем, вызываемых непониманием со стороны руководителя над программистами, заключается в том, что он думает, что его подчиненные должны производить код: они должны решать проблемы, а для этого они должны использовать код. До настоящего времени остались организации, которые измеряют «производительность труда программиста» «количеством строк кода, произведенным в месяц»; в действительности это количество может быть подсчитано, но они смотрят на этот вопрос не под тем углом зрения, ибо следует говорить о «количестве затраченных строк кода».

Настоящее кодирование требует огромной тщательности и неизменного таланта к точности; оно трудоемко, и поэтому его следует откладывать до тех пор, пока вы не станете максимально уверены в том, что программа, к кодированию которой вы намерены приступить, - это та самая программа, к которой вы стремитесь. Я знаю одну весьма успешную фирму, в которой заведено правило, согласно которому для проекта, рассчитанного на один год, запрещено начинать кодировать ранее чем через девять месяцев! В этой организации знают, что окончательный код – не более чем залог вашего понимания. Когда я сказал их директору, что моя основная забота при обучении студентов программированию – это научить их сначала думать, а не поспешно бросаться кодировать, он сказал только: «Если вы преуспеете в этом, ваша цена будет равна стоимости куска золота равного с вами веса». (Я не очень тяжелый).

Тем не менее очевидно, что многие руководители наносят ущерб, запрещая думать и подталкивая своих подчиненных «производить» код. Потом они жалуются, что 80 процентов их рабочей силы завязаны в «службе поддержки программ», и винят технологии программного обеспечения в этом плачевном состоянии дел, вместо того чтобы винить самих себя. Это уж слишком для плохого руководителя в сфере программного обеспечения. (Это все хорошо известно, но время от времени следует повторять это снова и снова).

Другое глубокое различие между вычислительным оборудованием и программным обеспечением заключается в различной роли тестирования.

Когда 25 лет назад разработчик логики стряпал электрическую схему, следующим его действием было построить и проверить ее, и если она не работала, он обследовал сигналы осциллографом и налаживал емкости. Когда она начинала работать, он начинал варьировать напряжение источника питания в пределах 10 процентов, подстраивал ее и

т.д., пока не получалась цепь, корректно работающая во всем диапазоне заданных условий. Он делал продукт, для которого мог «убедиться, что он работает во всем диапазоне». Разумеется, он не испытывал ее во «всех» точках диапазона, но это и не было нужно, потому что многочисленные соображения непрерывности делали очевидным, что вполне достаточно проверить схему при весьма ограниченном количестве условий, совместно «покрывающих» весь диапазон.

Этот итеративный процесс проб и ошибок был принят как нечто столь неоспоримое, что его применяли в условиях, когда предположение о непрерывности процедуры проверки не является верным. В случае артефакта с дискретным «пространством выполнения», такого как программа, предположение о непрерывности не является верным, и в результате итеративный процесс разработки методом проб и ошибок становится неприменимым. Хороший разработчик программного обеспечения знает это; он знает, что из наблюдения, что при тестах программа выдает корректный результат, нельзя экстраполировать, что программа в полном порядке; поэтому он пытается математически доказать, что его программа удовлетворяет требованиям.

Даже простой намек на существование среды, в которой традиционный процесс разработки методом проб и ошибок не подходит и где, следовательно, требуется математическое доказательство, неприятен тем, для кого математическое доказательство лежит выше уровня понимания. Поэтому такое предложение встретило значительное сопротивление, даже среди программистов, которым следовало бы быть более осведомленными. Неудивительно, что в мире вычислительного оборудования понимание потенциальной неадекватности процедуры тестирования является все еще весьма редким.

Некоторые разработчики оборудования начинают беспокоиться, но не потому, что осознают фундаментальную неадекватность своего подхода к тестированию, а лишь потому, что «подстройка» становится слишком дорогой с момента появления технологии БИС. Но даже помимо этого финансового аспекта им следовало бы волноваться, потому что значительная часть их разработки приходится на дискретную среду.

Недавно я слышал историю об одной машине (я счастлив добавить, что это не была машина разработки фирмы Burroughs). Это была микропрограммируемая многопроцессорная установка, которую ускорили добавлением памяти, но разработчики плохо справились с этим добавлением: когда два процессора одновременно обрабатывали две половины одного слова, машина с добавленной памятью вела себя не так, так без нее. После нескольких месяцев эксплуатации крах системы был прослежен вплоть до этой самой ошибки разработчиков. При тестировании нельзя даже надеяться отловить подобные ошибки, которые выявляются только при случайном совпадении. Ясно, что машину разрабатывали люди, не имеющие даже смутного представления о программировании. Один-единственный компетентный программист в команде разработчиков предотвратил бы этот промах: как только вы усложняете многопроцессорную установку введением дополнительной памяти, для компетентного программиста очевидна необходимость доказать – вместо того, чтобы слепо верить без убедительных к тому оснований, – что после добавления памяти машина продолжает соответствовать оригинальным функциональным спецификациям. (Подобное доказательство не должно вызывать каких-либо фундаментальных или практических затруднений). Убедить разработчиков вычислительного оборудования в том, что они попали в ту область, в которой их обычная экспериментальная техника проектирования и контроля качества больше не адекватна, – одна из важнейших задач обучения.

Я называю ее «важнейшей», поскольку, пока она не достигнута, разработчики оборудования не поймут, за что же именно ответственны программисты. Согласно инженерной традиции, завершенная разработка – это весь продукт разработчиков: вы построили систему – и вот она работает! Если вы не верите этому, просто испытайте ее, и вы увидите, что «она работает». В случае системы с дискретной рабочей областью единственная ожидаемая реакция на замечание, что она «работала» в проверенных

случаях: «Ну и что?». Единственное убедительное подтверждение того, что подобное устройство с дискретной рабочей областью соответствует требованиям, включает математическое доказательство. Было бы ошибкой думать, что результат работы программиста – это те программы, которые он пишет; программист должен производить решения, заслуживающие доверия, и он должен производить и представлять их в виде убедительных аргументов. Эти аргументы образуют ядро его продукта, а текст программы – всего лишь сопроводительный материал, к которому эти аргументы применимы.

Многие программные проекты, выполненные в прошлом, были чересчур сложны и, следовательно, полны ошибок и заплат. Это обусловлено в основном двумя следующими обстоятельствами:

1. драматическим увеличением производительности процессора и размера памяти, из-за которого кажется, что мы ограничены только небесами; только после создания нескольких чудовищно сложных систем нам становится ясно, что наши умственные способности ограничены куда сильнее.
2. мир, который в своем желании применять эти новые замечательные машины становится чересчур амбициозным; многие программисты уступили давлению расширить доступные программные технологии за пределы из возможностей; это не очень научное поведение, но, возможно, необходимо переступить предел, чтобы понять, где же он находится.

Оглядываясь назад, мы можем добавить две другие причины: из-за недостатка опыта программисты не знали, как вредна сложность, и кроме того, они также не знали, как многих сложностей можно избежать, если подойти к делу с умом. Возможно, было бы полезно, если бы аналогия между разработкой программного обеспечения и математической теорией была широко признана ранее, поскольку общеизвестно, что даже для одной теоремы первое найденное доказательство редко является лучшим: последующие доказательства часто на порядок сильнее.

Когда C.A.R. Hoare писал в начале этого года: «…порог моей терпимости к сложности намного ниже, чем обычно», он имел в виду развитие по двум направлениям: осознание больших опасностей сложности, а также растущие стандарты элегантности. Осознание опасностей сложности делает большую простоту желаемой целью, но поначалу вопрос о том, достижима ли эта цель, оставался открытым. Некоторые проблемы бросают вызов элегантным решениям, но очевидно, что большая часть из того, что сделано в программировании (и в компьютерной науке в целом) может быть существенно упрощено. (Известны многочисленные истории о 30-строчых решениях, состряпанных так называемыми профессиональными программистами – или даже учителями программирования! – которые могли быть уменьшены до 4-5 строк).

Обучить новое поколение программистов с пониженным порогом терпимости к сложности и научить их, как искать действительно простое решение, – это вторая большая интеллектуальная проблема в нашей отрасли. Это технически сложно, поскольку вы должны передать немного дара убеждения и массу хорошего математического вкуса. Это психологически трудно в среде, где путают любовь к совершенству с претензией на совершенство и, порицая вас за первое, ставит вам в вину второе.

Как нам убедить людей, что в программировании простота и ясность – вкратце: то, что математики зовут элегантностью – это не чрезмерная роскошь, а жизненно важный вопрос, который определяет выбор между успехом и провалом? Я ожидаю помощи от экономических соображений. В отличие от ситуации с оборудованием, где увеличение надежности обычно влечет увеличение цены, в случае с программным обеспечением ненадежность обходится чрезвычайно дорого. Возможно, это парадоксально звучит, но надежная (а следовательно, простая) программа обходится в разработке и использовании гораздо дешевле, чем (сложная и, следовательно) ненадежная. Этот «парадокс» должен заставить нас усомниться в том, стоит ли так уж полагаться на возможную аналогию

между разработкой программного обеспечения и более традиционными инженерными дисциплинами.

prof.dr.Edsger W.Dijkstra
Burrought Research Fellow

О природе информатики

EWD896

Теперь, когда летние курсы подходят к концу(1), самое время посмотреть на их тему под другим углом зрения.

Официальная тема курсов: "Потоки управления и потоки данных: Концепции распределенного программирования" - всего лишь определила направление, обычно мы приходили к гораздо более общим вопросам, которые, вероятно, жизненно важны для информатики в целом. Итак, какова природа информатики или, что возможно более точно, какой она должна быть?

В этом вопросе присутствует большая путаница, и это не должно нас удивлять, потому что существует множество точек зрения на компьютерное настоящее:

- Вы можете воспринимать компьютеры как продукт производства, предназначенный для продажи и получения прибыли. Насущным вопросом тогда становится, рекламировать ли их с помощью Чарли Чаплина или Микки-Мауса, и плану обучения информатике следовало бы включать в себя курс "Продвижение товара" в качестве главной составляющей.
- Вы можете видеть в компьютерах главное поле международного соперничества, тогда обучение должно состоять преимущественно из курсов "Безопасность" и "Промышленный шпионаж".
- Сегодня вы можете считать автоматизацию двигателем экономики, завтра - основной угрозой рынку занятости; поэтому главными курсами в учебном плане станут "Экономика" и "Производственные отношения".
- Осознав, что новые технологии оказывают сильнейшее влияние на мышление, расширяя горизонты в вечных философских вопросах "Могут ли машины мыслить?" или "Что такое жизнь?", мы можем заключить, что главную роль в преподавании информатики должны играть факультеты философии, психологии и биологии.

Это не полный список: предлагаю вам самим сформулировать и обосновать причины, по которым ученые-управленцы, лингвисты, физики-экспериментаторы и педагоги должны играть главную роль в информатике. А теперь, когда источник путаницы разобран достаточно подробно, взглянем на другую сторону явления: науку.

Сначала я хотел бы напомнить, что Наука как единое целое потерпела горькое поражение в достижении своих основных целей. Все мы помним, что этих целей изначально было три:

- В первую очередь - получение Эликсира, который дал бы нам вечную молодость.
- От вечной жизни в нищете радости мало, поэтому второй целью стал поиск Философского Камня, с помощью которого можно получать столько золота, сколько нужно.
- Как вы понимаете, планирование этих двух грандиозных проектов - Эликсира и Камня требовали большей дальновидности, чем у пророков, и Точное Предсказание Будущего стало третьей актуальнейшей целью науки.

Все мы знаем, что, по прошествии веков, медицина отделилась от знахарства (хотелось бы верить!), что химия вышла из алхимии, и астрономия откололась от астрологии. Другими словами, первоначальные цели были тактично забыты.

Неужели? Ну, не совсем. Очевидно, осталось застарелое чувство вины, и, как только всплывает интересная наука или технология, неожиданно вспоминают старые задачи, и подразумевается, что новая наука должна их решить. Можно добавить, что чем меньше понимают новую науку, тем больше на нее возлагают надежд. Мы видим, как сейчас от

компьютеров ожидают исцеления всех болезней мира и как, пока эти ожидания находятся в центре внимания, даже небеса не выглядят пределом.

По аналогии возникает, например, вопрос: какие из сегодняшних компьютерных исследований позже будут признаны "алхимией вычислений" и можно ли выявить их пораньше, но я оставляю подобные вопросы вам для самостоятельных предположений и подойду к предмету по-другому.

Поскольку правила "академической игры" чрезвычайно строги и неизменны, мы можем рассмотреть вопрос о том, какие аспекты информатики гарантированно жизнеспособны в качестве научных дисциплин. Здесь я в намного лучшем положении с тех пор, как десять лет назад разработал хорошо обоснованную теорию о жизнеспособности научных направлений.

Эта теория гласит, что если информатике суждено развиться в жизнеспособную научную дисциплину, то она должна превратиться в необычный формальный раздел математики, в котором знание собственно фактов почти не важно, а методологические аспекты играют неожиданно большую роль. Как следствие, не станет принципиальных различий между "чистой" и "прикладной" информатиками. Сегодняшнее определение математики (которое давно устарело) как "науки о величинах и количествах"(2) через некоторое время будет изменено на "искусство и наука об эффективных доказательствах", и когда это случится (спустя столетие или около того), информатика будет иметь намного большее влияние на математику, чем имела когда-то физика.

Все это захватывающе и очень соблазнительно потому, что математическое изящество станет очень важным. Как ученые-информатики мы знаем, что в нашей области математическая элегантность не просто бесполезная роскошь, а главная причина успеха или провала.

Приятно обнаружить, что словарь определяет прилагательное "изящный" в значении "простой и неожиданно эффективный".

Все же, перед тем как наступит это будущее в розовых тонах, плеснем дегтя в бочку меда. Простота - замечательное свойство, но необходимы огромные усилия, чтобы ее достичь, и хорошее образование для того, чтобы оценить ее по достоинству. И чтобы совсем испортить вам настроение: сложность продается лучше. Компьютерщики не единственные, кто открыл эту горькую правду: то же самое творится и в научном мире. Если вы прочитаете лекцию, в которой все предельно ясно от начала и до конца, ваши слушатели почувствуют себя обманутыми и, выходя из аудитории, будут бормотать себе под нос: "Но ведь это же тривиально?". Один из наших научных журналов не принял одну мою замечательную статью потому, что представленное в ней решение было "слишком простым, чтобы иметь научный интерес" и я жду отклонения другой статьи на основании того, что она слишком короткая.

Ко всему этому, система поощрений в научном мире работает против нас. Можно заработать хорошую репутацию, выдвинув какие-то сложные идеи - трудно заслужить благодарность за открытие того, как наилучшим способом обойтись без некоторых признанных, но чрезмерно сложных концепций: те, кто с ними не сталкивался, не заметят вашей работы, а те, кто в них заинтересован, будут вас ненавидеть. Поэтому, вот вам мой настойчивый совет - отбросьте мораль общества бестселлеров и найдите награду для начала в собственном удовольствии. Это достижимо: проблема простоты настолько привлекательна, что мы (если будем делать свое дело правильно) получим самое большое удовольствие в мире.

Или, коротко: Трижды ура Изящности!

Марктобердорф, 10 августа 1984 года.

The Netherlands

Профессор, доктор наук Эдсгер Вайб Дийкстра Исследовательский центр компании "Бэрроуз"

(1) Ежегодные летние курсы для молодых ученых (информатиков и математиков) в Марктобердорфе (Германия). События того лета описаны в EWD895: "Trip report E.W. Dijkstra, Marktoberdorf, 30 July - 12 Aug. 1984".

(2) Определение взято по Далю; определение в оригинале очень похожее.

Научная фантастика и научная реальность в информатике

EWD952

Как среди практиков информатики, так и среди широкой публики бытует масса недоразумений касательно информатики, и цель данной беседы – прояснить эти недоразумения, поскольку они вредят нам всеми мыслимыми способами.

С одной стороны, достижения информатики досадно игнорируются. Многие проекты, в которых осознанное применение информатики необходимо, выполняются в совершенно ненаучной манере, как будто информатики не существует вовсе. Это не может не вызывать сожаления, поскольку приводит ко многим дорогостоящим ошибкам, которых можно было бы избежать, а информатика не получает должного признания своих достижений.

С другой стороны, ожидания относительно того, что информатика может нам дать, зачастую совершенно нереальны: ожидается как минимум череда чудес целыми дюжинами. И это тоже весьма досадно, поскольку ведет к неоправданным надеждам и нереальным планам, а когда эти чудесные планы проваливаются с треском, информатика теряет доверия как еще одна из форм шарлатанства.

Если желаете, можете рассматривать этот доклад как попытку изменить то, что кажется сейчас основным затруднением для ученого-информатика: прежде всего ему не дозволено иметь какого-либо влияния, а затем на него валят все шишки за любые неудачи.

На первый взгляд кажется, что разъяснение квинтэссенции науки и затем получение выводов о свойственной ей роли – чисто техническая задача, но, к сожалению, это не так. Проблема состоит в том, что это противоречит интересам промышленности и чересчур неудобно для многих, чтобы быть услышанным. В прошлом от этого бы просто избавились. Это вызывало дискомфорт на некоторое время, после чего публика осмеивала это, и все возвращалось на круги своя, а провозвестника идеи терпели, как терпят выходки шута. Но состояние дел изменилось, интересы промышленников стали обширнее и разнообразнее, и все больше и больше людей начинают понимать, что уже становится не до смеха. По мере того как разумное и бесстрастное обсуждение этих вопросов становилось все более неотложным, становилось все труднее и труднее говорить о них. За техническими деталями я отправляю вас к судьбе Галилео Галилея.

Ранее я говорил, что достижения информатики, к сожалению, игнорируются. Что же, если бы это было именно так, не было бы повода для тревоги: разрыв между полезными достижениями в лаборатории и их практическим применением неизбежен. В нашем случае, однако, есть повод бить тревогу, поскольку на протяжении последних десятилетий разрыв между информатикой и практическим использованием компьютеров только расширился. В то время как информатика делала большие шаги к превращению в строгую, точную науку, компьютерная практика в основном пребывала в застое. Я не преувеличиваю: физики до сих пор убеждены, что FORTRAN – последнее слово в информатике, химики продолжают использовать BASIC, и COBOL стал для бакалавра тем же, что и APL для инженера-электронщика. Человеческая склонность привязываться к источнику своих бед зарекомендовала себя как важный стабилизирующий фактор во многих браках и религиях, однако из-за своей болезненной склонности к неадекватным инструментам эти дисциплины переступили черту, за которой им уже не помочь.

Эти наблюдения обычно относятся к «случайному пользователю», но в последнее время проблема усилилась из-за того, что она повлияла на многие университетские учебные курсы таким образом, что вместо первоклассных ученых они теперь готовят третьесортных программистов.

Что же, вы можете возразить, что эти люди сами себя считают не программистами, а физиками, химиками и т.д., но и в профессиональной области картина столь же

безрадостна. Мы все слышали о «Шаттле», который не был запущен из-за ошибки в программном обеспечении синхронизации, поэтому я должен привести вам другой пример. Недавно Британская железная дорога установила свою первую компьютерную систему сигнализации на одной из веток, и они разрекламировали в надежде продать ее другим железнодорожным компаниям, что во избежание риска от использования компилятора в целях безопасности система была написана в машинных кодах. Очевидно, даже не ставилось под сомнение, что промышленные компиляторы сыграют с вами злую шутку. Другая область, полная страшных историй, - разработка VLSI (СБИС, сверхбольшие интегральные схемы – прим. перев.). В арсенале разработчиков среди прочих инструментов имеется программа – очень дорогая в эксплуатации, – которая пытается восстановить схему по фотошаблону, поскольку программа, производящая фотошаблон, ненадежна. Но алхимия коммуникационных протоколов превосходит их всех, вместе взятых; первоначально разработанные телекоммуникационными инженерами для компенсации случайных аппаратных сбоев, они стали настолько громоздкими, что вызывают гораздо больше ошибок передачи, чем те, которые они должны были бы исправлять, и проверка их реализации стала сложной задачей: распутывание этого клубка может стать темой диссертации. И всех этих несуразностей можно было бы избежать, поскольку информатика начала углубленное и успешное изучение проблем компиляции и синхронизации двадцать с лишним лет назад. Слишком много для разрыва между наукой и практикой.

Мы уже видели, что программы учебных курсов меняются под действием необоснованного предположения, что в наши дни использование компьютера является основным, чтобы стать хорошим ученым. Сейчас это представляется весьма странным, поскольку этому выводу противоречит факт, что существует большое количество чрезвычайно успешных ученых, которые никогда не использовали компьютер, так как в их время компьютеров попросту не было. Верно, что микрокомпьютеры продаются столь же широко, как и энциклопедии, компрометируя родителей будущим их чад. А если вы читаете между строк, призыв звучит еще более заманчиво: дайте вашему ребенку наш домашний компьютер, и он вырастет гением. Но нам не следует винить промышленность за это недоразумение, как бы настойчиво она не пыталась запихнуть свои леденцы в наши глотки: в подобных рекламных ухищрениях они лишь используют нашу скрытую, но не покидающую нас мечту о «философском камне», способном делать золото, и «Эликсире», дарующем вечные молодость и здоровье. Промышленность ничем не хуже обычного проходимца.

Насколько мне известно, не промышленность, а общество придумало термин «компьютерная грамотность». В самом начале игры в человеческое сознание прочно вбили мысль, что компьютеры появились, чтобы спасти нашу экономику (а позже – также и экономику стран третьего мира). Некоторое время никто не знал, каким именно образом, но, к счастью, кого-то посетила блестящая идея, что информация – это «ресурс», и это решило проблему, тем более что с этого момента, впервые в истории человеческого разума, мы наконец-то получили ценный ресурс, запасы которого неисчерпаемы. Затем люди вспомнили, как нефть принесла сказочные богатства немногим счастливцам; на этот раз подобного не должно было произойти, поэтому для решения этой проблемы была изобретена «компьютерная грамотность» для миллионов. Что этот термин должен означать, до сих пор является открытым вопросом, однако это не мешает нашим оракулам продолжать грезить о постиндустриальном обществе, в котором люди, когда им надоест наслаждаться телеконференциями, будут посвящать свой обширный досуг созидательным видеоиграм.

Нет ничего нового в том, что люди ожидают рая на земле в ближайшем будущем; новым, однако, является то, что это будет сотворено при помощи микроэлектроники, а не милостию Божьей. Я не преувеличиваю: ожидания эти столь нереальны, что вещи, которые следовало бы разумно обсудить, все чаще преподносятся так, словно люди

участвуют в странном ритуале, истово повторяя слова жрецов на языке, которого они не понимают. Позвольте привести лишь несколько свежих примеров.

Директор крупной исследовательской лаборатории во время интервью по поводу его ухода на пенсию преподносил достоинства домашнего компьютера, подчеркивая, что он позволит людям «упорядочить их творческие мысли». Только подумайте на минуту, какая бессмыслица! Между нами, когда вас в последний раз посещала творческая мысль? Пару лет назад? Неплохо. Гораздо больше смысла было бы, если бы он сказал «привести в порядок вашу коллекцию марок». Когда Бертран Рассел обронил свое знаменитое изречение «Многие люди предпочтут скорее умереть, чем подумать. Это в самом деле так», он высказал больше реализма. Лично я сделал для себя вывод, что этот директор ушел на пенсию не слишком рано.

На международной конференции, посвященной компьютерам в образовании, все сошлись на мысли, что маленьких детей следует знакомить с компьютером при первой возможности. Возникла некоторая озабоченность, что столь интенсивная механизация образовательного процесса могла бы повредить их эмоциональному развитию и образованию навыков общения, пока кто-то не заметил, что «разговор с программой не причинит вреда, поскольку, будучи продуктом человеческой мысли, программа обладает в основном человеческими чертами». Теперь я задумываюсь, не было ли окончательное решение Гитлера результатом человеческой мысли?

На последней конференции относительно осуществимости или неосуществимости разработки программного обеспечения для проекта «звездных войн» одним из аргументов в пользу осуществимости, как мне сказали, было замечание, что сегодня почти все банковские операции успешно компьютеризованы, не так ли? Ну что ж, по правде говоря, это не так: не далее как в 1986 году попытка выплачивать зарплату электронными средствами в крупной (компьютерной!) компании провалилась. Но если бы даже это было и так, это ложный аргумент: эти два проекта настолько различны, что аналогия на несколько порядков величины мельче, чем нужно для принятия ее во внимание. Хуже всего, я считаю, то, что этот аргумент был выдвинут человеком, занимающим видное положение в академической среде.

В шестидесятых годах сэр Герман Бонди написал статью о том, кому следует заниматься академическими исследованиями. Вопрос был поднят весьма важный. До второй мировой войны ученых было пренебрежимо мало, но после признания того, что война была выиграна наукой и технологиями, академические учреждения начали получать поддержку. Неудивительно, что Бонди пришел к выводу, что многим из его современников, привлеченных к научным исследованиям, лучше было бы подыскать себе другое занятие. Поскольку взрывное развитие университетов было напрямую связано с ожиданием большой отдачи от науки, Бонди также посвятил раздел своей статьи обсуждению пользы от науки в целом. Его вывод отрезвляет.

Он указывает, что среди задач, поступающих в университеты извне, из «реального мира», около 80% тривиальны и около 20% явно неразрешимы, и что академические исследования потенциально могут повлиять на тонкий граничный слой между ними, так как только там знания, талант и упорный труд могут достичь чего-то, что не может быть достигнуто другими средствами. В нынешней научно-технической эйфории столь отрезвляющее предупреждение более чем необходимо; оно, впрочем, не было ни услышано, ни замечено, но это уже другая история.

По несчастному историческому совпадению, компьютеры появились как раз в десятилетия безграничной веры в всемогущество науки и технологии. Тогда как более устоявшиеся науки обзавелись корнями и традициями в более благоразумные времена, мы ввязались в информатику, когда наши ожидания ничем не сдерживались. С исторической точки зрения этот наивный оптимизм вполне понятен. И этому даже можно найти кое-

какие оправдания. Видите ли, Бонди физик, и его мнение 80/20% обусловлено его знанием, каким образом естественные науки справляются со сложностями, с которыми мы сталкиваемся в окружающем нас мире. Напротив, информатика имеет дело с миром артефактов, в котором сложность имеет рукотворное происхождение. В этом заключается большая разница между ними, и вполне возможно, что в области информатики «пограничный слой» Бонди вовсе не так уж исчезающе тонок. Но чтобы сделать грамотный вывод о том, что данный слой мог бы и должен был бы содержать, нам придется понять, как возникла и как развивалась информатика.

Впрочем, перед тем, как обратиться к информатике как таковой, я должен разделаться еще с одной мечтой, которой часто грезили в эти десятилетия, а именно – что возможно «планировать» исследования таким образом, чтобы в должное время производить то, что требует мир. Наука не может работать в таком режиме (что, кстати говоря, не так уж плохо, поскольку зачастую существует коренное различие между тем, что мир требует, и тем, что ему в действительности нужно). Е.Т. Белл справедливо хвалил некоторых руководителей за то, что они были «достаточно дальновидны, чтобы понять, что самый простой путь избавить математику от математика – это оплачивать его содержание», и это справедливо и для других наук. Никогда еще не наблюдался значительный прогресс в науке из-за того, что какому-то благодетелю понадобился результат. Существенный прогресс в науке возникает лишь тогда, когда после поисков и размышлений образованный и оригинальный ум приходит к выводу, что что-то загадочное наконец-то созрело для понимания или что-то очень трудное может теперь быть сделано. Успешный научный поиск – это искусство делать действительно возможное, и следовательно, развитие науки гораздо лучше рассматривать как автономный процесс со своими собственными правилами, а не как плановую деятельность с заранее установленными внешними целями. (И с этим парадоксом сталкиваются все директора промышленных исследовательских лабораторий: привлекая нужных людей, они не могут служить своей компании лучше, чем просто оставив их в покое).

Сторонники междисциплинарных исследований, кажется, иногда верят в то, что границы между различными науками – не более, чем каприз истории. Но пути, которыми научное знание расходилось по разным дисциплинам, вовсе не случайны: то, что может образовать область жизнеспособной научной дисциплины, должно иметь количественные и качественные ограничения.

Среди количественных ограничений я упомяну, что область должна быть достаточно мала, так чтобы основные положения умещались в одной человеческой голове; с другой стороны, она должна быть достаточно обширна, чтобы предоставлять пищу для размышлений хотя бы на некоторое время.

Среди качественных ограничений я упомяну, что с одной стороны эти проблемы должны быть достаточно независимы от остальных, чтобы их можно было изучать изолированно от других, тогда как с другой стороны область должна обладать внутренней согласованностью.

Последнее требование полностью проясняет, каким образом раньше отделения информатики предшествовали самой информатике: поначалу они представляли собой не более чем малопонятный коктейль всевозможных дисциплин, имеющих отношение к компьютерам, которые посчастливилось наскрести по университету, например: немного электроники, немного численного анализа, немного статистики и экономики, немного бизнес-администрирования и в США в довершение всего – немного искусственного интеллекта. Коктейль, составленный как попало, не может быть хорош на вкус. Образование связей – одна из первостепенных задач для тех, кто пробивает нишу, в которой многообещающая наука информатика станет жизнеспособной.

Ради этой согласованности они оставили все специфические области потенциальных компьютерных приложений и попытались сконцентрироваться на том, что все эти

приложения должны были иметь общего; они сделали это ради согласованности и общности, которая необходима для надежности.

Другой аспект этой надежности – сознательная попытка избежать подготовки ученых в течение половины от пятилетнего срока. Это включает, в частности, воздержание от всего, что только имеет отношение к компьютерам, доступным нынче на рынке. Например, то, как обходиться с идиосинкразией OS/360, рассматривалось как временная цель, не относящаяся к предмету информатики.

В общем: ниша была подготовлена вне специфических приложений и вне специфических машин. С течением времени в том же духе продолжается уклонение от особенностей специфических языков программирования и операционных систем. Сначала это делалось для защиты зарождающейся науки от изменчивости рыночных продуктов, а когда некоторые из этих продуктов стали стандартом де-факто, это стало делаться для защиты расцветающей науки от рыночного застоя. Независимо от того, выражать ли сожаление или восторг по поводу этого разделения, я хочу, чтобы вы поняли, что для становления информатики как жизнеспособной науки это разделение было и остается conditio sine qua non.

Что же информатика делала в этой блистательной изоляции? Точнее говоря, что она делала, не теряя претензий на прикладную значимость? Что же, сделала она немало; фактически, куда больше, чем я могу разъяснить в рамках данной лекции, но я могу хотя бы передать вам общие моменты.

В шестидесятые годы она разработала теорию синтаксического разбора, необходимую для поднятия уровня компиляторов выше уровня поделок, напичканных ошибками, и превратив ее в предмет, пригодный для обучения. Это было главное достижение: я, например, помню весьма отчетливо, как в 1962 те из нас, кто действительно написал компилятор, выглядели в глазах остальных как некие полубоги. В связи с этим достижением я бы хотел подчеркнуть, что этого никогда бы не произошло, если бы мы со временем не научились давать формальное определение синтаксиса компилируемого языка: без этого формального определения слишком сложно было бы определить существование проблемы компиляции. Теория конечных автоматов и теория сложности были разработаны, чтобы задать основные количественные границы того, что в принципе может быть вычислено; опять же, в основе этих теорий лежит очень формальный постулат относительно того, что есть вычисление, постулат, без которого эти теории не могут существовать. Для разработки операционных систем проблема синхронизации процессов была поставлена и решена, и первые теоремы об отсутствии «смертельных объятий» были доказаны; также формальное определение явления, интуитивно известного как «смертельное объятие», было первой предпосылкой этого достижения.

В семидесятые годы центр внимания сместился от синтаксиса к семантике, сначала к детерминированным последовательным программам, но вскоре впоследствии охватил также недетерминированность и параллельность. Я не буду подробно описывать различные отрасли: они простираются от модели типизированного лямбда-счисления до разработки преобразований программ с сохранением семантики. В этом десятилетии программы стали самостоятельными математическими объектами. Кратчайший способ уловить изменение направления внимания – это, пожалуй, отметить, что если раньше задачей программ было управлять поведением машины, то теперь задачей машины стало выполнение наших программ. Верификация и разработка программ развились в разделы формальной математики до такой степени, что теперь уже не считается безответственностью опубликовать программу, не испробовав ее на компьютере.

Что ж, это далеко не полный обзор того, как информатика стала наукой, и я приношу свои извинения всем неупомянутым мной, кто внес свой вклад в ее становление. Но я надеюсь, что он все же достаточно полон, чтобы донести до вас аромат квинтэссенции дисциплины. Она стала замечательной дисциплиной, поскольку разделение между «чистой» и «прикладной», столь традиционное для многих других дисциплин,

совершенно поблекло и существенно утратило свое значение. Роскошь работы в окружении, в котором различие между чистой и прикладной наукой лишено смысла, - это, пожалуй, еще одно признание факта, что компьютер общего назначения действительно заслуживает эпитета «общего назначения».

Она обладает всей пикантностью чистой математики, будучи более формальной, чем многие другие отрасли математики. Она не может избежать такой формальности, поскольку любой язык программирования, будучи интерпретируемым механически, представляет своего рода формальную систему. В то же время она обладает всей прелестью прикладной математики, поскольку огромная мощность современных компьютеров дает такие возможности для создания хаоса, что ее методы необходимы, если мы не намерены попасться в ловушку сложности, которую сами же и создали.

Научиться не попадаться в собственноручно произведенную ловушку сложности, сохранять вещи достаточно простыми и научиться достаточно эффективно мыслить о своих разработках – вот центральная задача информатики. Это также было осознано больше десятка лет назад, когда «разделение задач» стало находкой программной терминологии.

Заметьте, пожалуйста, что путь, которым информатика пробивала свою нишу, был сам по себе примером успешного «разделения проблем». Каждый раз, когда мы вкладываем уйму умственной энергии в тщательную разработку любой дискретной системы, мы делаем это не просто для удовольствия: мы всегда надеемся, что результат наших усилий будет использован на благо другим. Мы надеемся, что он будет удовлетворять нужды, соответствовать ожиданиям и доставлять удовольствие своим пользователям. В донаучный период разработки систем неформальное понятие «удовлетворения пользователя» было единственным приемлемым критерием качества программного обеспечения.

Недостаток «удовлетворения пользователя» как критерия качества состоит в том, что это не техническое понятие: он не задает технического направления разработчику и, кроме того, может быть достигнут другими средствами, помимо технических, например, агрессивной рекламой или промывкой мозгов. К науке это не имеет ни малейшего отношения. Задачи должны быть разделены, и тут на сцене появляются функциональные спецификации.

Роль формальной функциональной спецификации – просто служить логическим барьером между двумя совершенно разными проблемами, известными как «проблема удовлетворенности» и «проблема корректности». Проблема удовлетворенности касается вопроса, соответствует ли система, отвечающая таким-то и таким-то формальным спецификациям, вашим ожиданиям и надеждам. Проблема корректности касается вопроса, соответствует ли данная разработка таким-то и таким-то формальным функциональным спецификациям.

Логический барьер был необходим, чтобы поставить проблему корректности перед информатикой: он изолирует уютную нишу информатики от проблемы удовлетворенности, решению которой наука мало чем способна помочь. Заметьте, пожалуйста, что я не утверждаю, что одна из проблем важнее другой; в конце концов, цепь не прочнее самого слабого звена. Однако я утверждаю, что проблема корректности представляет ту самую часть, которую нам удалось втиснуть в «тонкий пограничный слой» Бонди, где разумное применение научной мысли может принести пользу, тогда как неформализованная проблема удовлетворенности в действительности лежит на пределами компетенции науки.

У вас могут быть самые разные проблемы, начиная с опасного перекрестка и заканчивая такими, которые угрожают самому существованию целых поколений. Однако наука никогда не решает ваши проблемы, она решает лишь свои собственные, и заключение, примете ли вы решение формальной научной проблемы как таковой в качестве приемлемого решения своей задачи, лежит целиком на вас. Другими словами:

наука никогда не предлагает моделей реальности, она лишь строит свою теорию, и вопрос, примете ли вы свое восприятие действительности в качестве достаточно достоверной модели для этой теории, полностью ваша проблема. Достоверность и реалистичность больше не являются научными понятиями, и ученый уступает право разглагольствования о них философам, пророкам и поэтам.

С этой точки зрения роль науки довольно ограничена, на самом деле до такой неутешительной степени, что многие предпочитают закрывать глаза на ограниченность науки. Акцентировать внимание на этих ограничениях не принято в научной среде, поскольку помимо прочего это вызывает вопрос, а зачем тогда общество должно терпеть ученых. Это не шутка: все мы знаем, что если бы сегодня общество вдруг решило изгнать своих ученых, это было бы не впервые.

И теперь, когда наука «вышла в народ», так сказать, она также не столь популярна среди публики в целом. Люди всегда питали противоречивые чувства по отношению к технологии, и с чем более мощными технологиями они сталкиваются, тем драматичнее становится двойственность этого отношения. Они чувствуют угрозу со стороны технологии сильнее, чем когда-либо прежде; в то же время их надежда на спасительную мощь науки и технологии все больше крепнет. В старые добрые временя традиционного шаманства от него требовались только снадобья от всех недугов, Эликсир для вечной юности и Философский Камень для сотворения золота. Так было в старые добры времена, когда поиск Философского Камня только начинался.

В наши же дни передовой электроники Философский Камень обзавелся совершенно новыми измерениями. Достижения в традиционном стиле сотворения золота благодаря робототехнике постепенно во всех странах приведут к положительному балансу в торговле. Они решат проблемы производства и безработицы. Они дадут правительству силы победить преступность и коррупцию, тогда как вездесущие микрокомпьютеры станут на страже демократии. Обучающие машины обновят образовательный процесс, в то время как калькуляторы, автоматические корректоры правописания и в целом мыслящие машины сделают большую часть обучения излишней. Все государственные секреты будут абсолютно надежно защищены неприступным шифрованием; мощные схемы декодирования позволят нам взламывать любые коды. Оружейные и защитные системы будут одинаково эффективны. И, что важнее всего, если мы не знаем, что нам делать, мы получим «поддержку принятия решений», наше руководство будет информировано, а наша информация – управляема, нашим разработкам будет оказываться столь же значительная помощь, что и усилению наших мыслительных способностей, и безо всякой специальной подготовки любой, в самом деле любой, даже руководство и генералы, будут иметь под рукой все нужные им знания экспертов. Замечательное новшество сегодняшнего Философского Камня состоит в том, что вы можете перевалить на него свою ответственность.

Слишком много для вопиющей бессмыслицы. Теперь можно было бы поспорить, что в своей несклонности к компромиссам я занял непримиримую позицию; я знаю, что немало из моих благоразумных и уважаемых коллег критикуют бы пределы, которые я обрисовал им для информатики непрактично узкими. Они указывают на всевозможные системы, обладающие большой потенциальной полезностью, лежащие за пределами очерченных мной строгих границ. Они возражают, прежде всего, что нынче такие системы разрабатываются без особых формальных функциональных спецификаций, как бы нам этого ни хотелось. Один из примеров предоставляется библиотекой расчетных программ, для которой не границы применимости, ни точность результатов не были четко установлены. Другим примером могла бы быть система оптического распознавания символов для использования в сортировке почты, по крайней мере будучи поставленной без точного определения, какие символы будут безошибочно распознаваться. Верно, однако позвольте мне отметить несколько моментов.

Прежде всего, позвольте напомнить, что я отказался разделять проблемы удовлетворенности и корректности по их относительной важности, другими словами, что нет никакой внутренней порочности в ненаучном проекте: он всего лишь ненаучен. Во-вторых, существуют проекты, для которых очевидно, что ниша информатики не подходит; если это тот самый случай, информатика не в состоянии ничем им помочь, и лучше ей не вмешиваться. Почему компьютерные приложения, которым наука не способна помочь, должны быть отброшены? Я не думаю, что нам следует беспокоиться об этом. Я твердо убежден в том, что мы сильно недооцениваем культурное значение компьютеров, судя о них в первую очередь как об инструментах, поскольку я ожидаю от них гораздо большего влияния на их способность интеллектуального вызова.

Мое растущее беспокойство, однако, вызывает то, что подобные проекты используют лишь часть компьютерных характеристик, и то, что они ведут к достаточно неспецифицированным продуктам, в которых исчезают другие характеристики компьютера. Расчетные программы используют способность компьютера к перемалыванию чисел, оптическое распознавание символов использует емкость памяти и гибкость, но в обоих случаях конечный продукт больше не является автоматом с точно установленными свойствами. Следовательно, результирующий продукт уже не является машиной в смысле информатики, и его использование сродни занятиям математикой без аксиом. Подобные расчетные программы могут использоваться лишь в контексте, в котором результат не имеет столь большого значения, либо аналитик, использующий их, располагает другими средствами для проверки ответа; подобная система оптического распознавания может применяться лишь в обстоятельствах, в которых не имеет большого значения, если, скажем, часть писем сначала отправится в неверном направлении. Мы больше не можем полагаться на такие системы, как если бы они были автоматами, и нам придется удалять их из замкнутых циклов.

Для многих такое заключение неприемлемо, и как результат существует школа мышления – или, если вам так больше нравится, школа недомыслия, – которая утверждает, что ситуация не столь серьезна, что нам не следует быть столь строгими, что инженеры всегда допускали периодические сбои своих компонентов, что стремление к совершенству быстро начинает препятствовать производительности, и что лучше бы нам научиться жить в реальном мире с системами и подсистемами, которые обычно делают то, что мы от них ожидаем. Это соблазнительное предложение. Разве не замечательно – достичь превосходства без гонки за совершенством? (Тем более что в крайне популистском обществе последнее социально неприемлемо). Такова позиция, ныне принятая под знаменем «программной инженерии».

Однако то, что сторонники называют «прогрессом программной инженерии», страдает несколькими непреодолимыми противоречиями.

Одно из них – это то, что стремление к совершенству находится в противоречии с производительностью в том смысле, что сделает разработку программного обеспечения чересчур дорогостоящей. Но в чем основная причина заоблачных цен на нее? Дороже всего обходится как по людским ресурсам, так и по непредвиденным задержкам отладка, и можно немало сэкономить, вложив больше средств в предотвращение будущих ошибок, поставив разработку на первое место. Поскольку ошибки обходятся столь дорого, в конечном счете высококачественный дизайн обходится куда дешевле. Другая важная причина состоит в том, что многие системы построены на зыбком фундаменте в том смысле, что базовое программное обеспечение в виде операционных систем и компиляторов слишком неустойчивы, поэтому каждый новый релиз этого базового программного обеспечения требует возможно дорогой адаптации базирующейся на нем прикладной части. Наконец, многие инструменты, с которыми собирается работать программист, настолько плохо документированы, что вынуждают его выяснять экспериментальным путем, чем же они могут оказаться для него полезными. Поскольку эти эксперименты могут быть весьма дорогостоящими и длительными, бедный

программист оказывается в поистине незавидном положении, поскольку вынужден полагаться лишь на свои догадки. Так что вы видите здесь три основных источника роста цены, след от которых ведет к чьему-то предположению, что стремление к совершенству противоречит производительности!

Другое противоречие в развитии инженерии программного обеспечения всплывает, как только инженер-информатик задается вопросом, каким образом стать лучше, как искусство, мастерство или практика инженерии программного обеспечения могут быть улучшены. Он немедленно обнаружит, что вынужден обратиться к дисциплине, которую отверг. Приняв в качестве хартии «Как программировать, если не можешь этого делать», инженерия программного обеспечения загнала себя в неприемлемую позицию, в позицию, которой информатика может избежать, лишь отказавшись от компромисса, придерживаясь своей собственной формальной дисциплины и не прикидываясь чем-то б*о*льшим.

Чуть раньше я упомянул плохую документацию системы как внутреннее ограничение надежности, с которой система может быть использована механически в более широком контексте. Теперь самое время указать, что привлечение технического писателя редко является выходом из положения; в сущности, это не более как признание того, что разработчики системы в некотором роде функционально безграмотны. Обычно даже целая армия технических писателей не может справиться с задачей, поскольку система становится столь сложной, что не поддается точному описанию.

Выдающийся пример этого явления недавно продемонстрировала Ada. Если Ada собирается выдать стандарт, желательно, чтобы он был недвусмысленно документирован. По меньшей мере две группы попытались сделать это; в результате обе выдали около 600 страниц формального текста. Это гораздо больше, чем необходимо, чтобы удостовериться в невозможности хотя бы твердо установить, что оба документа определяют один и тот же язык. Ошибка очевидной неуправляемости этих двух документов кроется ни в двух группах, составивших их, ни в принятом ими формализме, а лишь в самом языке: сами не обеспечив формального определения, могут ли его разработчики скрыть, что они предлагают неуправляемого монстра. То, что Ada уменьшит проблемы программирования и увеличит надежность наших разработок до приемлемых границ, – это лишь одна из тех сказок, в которые могут поверить только люди с военным образованием. Самое лучшее, что я могу сказать об этом, – это упорные слухи, что даже военного образования недостаточно, чтобы поддерживать веру в этот Философский Камень. Я упомянул про Ada, поскольку это замечательный пример того, на что я указывал в начале доклада: ее принятие – это политический процесс, в котором информатике, предостережения которой рассматривались как досадная помеха, не было дозволено оказывать никакого влияния. Следовательно, даже простое согласие с чьим-то обоснованным сомнением на этот счет, даже без вмешательства в контроль и намерения, становится действием с политическим душком.

Я еще не говорил об искусственном интеллекте. Что ж, этот предмет связан с другими политическими осложнениями, поскольку он стал частью заокеанской полемики: этот вопрос никогда не поднимался в Европе. В течение первых двух послевоенных десятилетий этому существовало простое финансовое объяснение. Искусственный интеллект был дорог, а Европа была бедна; к тому же искусственный интеллект финансировался почти исключительно министерством обороны, которое направило свои усилия на субсидирование – не могу сказать «поддержку» – американских исследований. Но одна лишь финансовая сторона не объясняет всего, поскольку, когда Европа стала достаточно богатой, чтобы финансировать собственные исследования в области искусственного интеллекта, об этом по-прежнему не велось и речи. На самом деле подобная участь постигла и другие отрасли науки программирования.

Мое заключение таково: это лишь один из аспектов намного более существенных культурных различий. Европейский разум поддерживает большее различие между

Человеком и Машиной и ждет меньшего от обоих. Он менее склонен описывать человеческую психику в механистических терминах; он также менее склонен описывать бездушные машины антропоморфной терминологией; следовательно, он считает вопрос, может ли машина мыслить, столь же уместным, сколь вопрос, может ли субмарина плавать. Это общество очевидно менее тяготеет к разным техническим штучкам, отчасти потому, что не ждет от них слишком многого, тем более спасения. Наоборот, попытка имитировать человеческий разум вызывает у них лишь комментарий: «А может, попробуете скопировать что-нибудь получше?».

Обычно мне не нужно говорить об искусственном интеллекте, поскольку это всего лишь специфическая область потенциального применения машин и поскольку она находится за пределами собственно информатики. Впрочем, я вынужден делать это, как только высказывается мнение, что с применением техники искусственного интеллекта машины справятся с проблемами программного обеспечения, которые нам самим не по зубам. Нашей первой реакцией на Проект Пятого Поколения был вздох облегчения в таком духе: «Что ж, если японская промышленность пытается вложиться в искусственный интеллект, уж он-то позаботится о японской конкурентоспособности». Примерно через неделю пришло грустное понимание, что западному миру, похоже, не хватит силы духа удержаться и не примкнуть к этому поветрию.

В самом деле, он примкнул к этому поветрию, и, следовательно, опять входит в моду мысль: «не правда ли, как было бы мило, если бы наши машины были настолько умны, чтобы их можно было программировать на естественном языке?».Что же, естественные языки лучше всего подходят для своей основной цели, а именно быть неоднозначным, передавать шутки и признания в любви, но совершенно не подходят для хотя бы сколь0нибудь немного разумной точности. И если вы не верите этому, попытайтесь прочесть современный правовой документ, и вы тут же увидите, как необходимость в точности создает самые неестественные языковые обороты; либо попытайтесь почитать одно из первоначальных словесных доказательств Евклида (желательно на греческом). Это исцелит вас и заставит понять, что формализмы вводятся не для того, чтобы усложнить вещи, а для того, чтобы сделать их возможными. И если вы все еще верите, что нам легче всего изъясняться на наших родных языках, вас следует приговорить к прочтению пяти студенческих рефератов. Проблема с «умными» машинами остается той же, что и со всеми «опциями» языков программирования: каждый последующий слой «дружественности к пользователю» затуманивает спецификацию и тем самым делает систему более рискованной для использования.

И в заключение. Это не предмет моего доклада, однако в эти политические дни если я сам не подниму этот вопрос, он будет поднят в обсуждении: как насчет программного обеспечения для Стратегической Оборонной Инициативы, более известной как «Звездные войны»? Что ж, я уверен, что не смог бы его разработать, к своему удовлетворению.

Благодарю за внимание.

Почему американская информатика кажется неизлечимой

EWD1209

Обсуждая, как чувствует себя академическая информатика в США, мы должны уделять больше внимания компьютерной индустрии, чем в других странах, и должны мы это делать по двум причинам. Во-первых, в сравнении с другими странами, здесь нет барьера, отделяющего американский студенческий городок от окружающего общества, во-вторых, большая часть, если не вся, мировой компьютерной индустрии сосредоточена в США. Следовательно, мнения и предрассудки, господствующие в американской компьютерной индустрии, для академической информатики здесь неоспоримы, невзирая на то, являются ли они движущими вперед или парализующими.

Ради достижения стабильности предприятия идеал для менеджера – организация, максимально независимая от индивидуальных способностей служащих. Преобладание этого идеала – это хорошо документированный международный феномен. Этот идеал возник в индустрии высоких технологий, в которой он чрезвычайно неуместен и отталкивает от сотрудничества промышленников с учеными, особенно с блестящими и оригинальными учеными.

Американская ситуация усугубляется общим неверием в ее образовательную систему и глубоко укоренившимся недоверием к интеллектуалам.

Кстати. Насколько типично американским является это недоверие, хорошо иллюстрируется фактом, что американское слово «яйцеголовый» не переводится на голландский язык. Я поискал его в моем «Webster's New Collegiate Dictionary» (1973), выдержка из которого относится к непереводимому слову: "яйцеголовый сущ.: интеллектуал, далекий от жизни ученый *<с точки зрения практичных людей, которые свысока относятся к планам и мечтам яйцеголовых – W.L.Miller>*".

По голландским меркам, цитата из Миллера является однозначной дискредитацией «практичных людей». (Конец Кстати).

В соответствии с этим, повышенное внимание со стороны индустрии оказывает сильное давление на университеты – не поощрять увлечения вроде академического образования, а углубляться в профессиональную подготовку того или иного рода.

Но в случае информатики индустриальные предрассудки влияют не только на преподавание, исследования страдают от них ничуть не меньше. Например, я припоминаю одного так называемого «кандидата в преподаватели», который докладывал о своей диссертации на звание доктора философии – так было принято, – темой которой являлась система «распараллеливания» весьма скромного класса Fortran-программ. Докладчик не претендовал на какое-либо новое видение предмета или на то, что вы могли бы научиться чему-то интересному, изучая его тезисы. Единственное обоснование его работы заключалось в ее конечном продукте, а именно – программном обеспечении для распараллеливания Fortran-программ. И это программное обеспечение было весьма важно, поскольку (i) имеются тысячи Fortran-программ и (ii) система является полностью автоматической – абсолютное требование для ее принятия промышленностью. (Я не выдумал все это!)

Я думал, что основной критерий, по которому должны оцениваться наши академические исследования, – это насколько они улучшают учебные материалы, но этот бедняга принял в качестве критерия качества «применимость в промышленности», и в результате его творение работало в таких узких рамках условий, что его основным достоинством стала бездумная легкость применения.

Есть и другие примеры того, насколько исследования страдают от давления промышленности.

Быть лучшим программистом – означает быть способным разрабатывать более эффективные и достоверные программы и знать, как делать это эффективно. Это относится к экономному использованию ячеек памяти или машинных циклов, а также к

избеганию сложностей, которые увеличивают количество рассуждений, необходимых для удержания строгого интеллектуального контроля над разработкой. То, что, по моему мнению, для этого нужно, - это совершенствование математических навыков, при этом я употребляю слово «математика» в смысле «искусство и наука эффективных рассуждений». В действительности задачи разработки высококачественных программ и построения высококачественных доказательств весьма сходны, настолько сходны, что я уже не могу их различать: я не вижу осмысленных различий между методологией программирования и математической методологией в общих чертах. Короче говоря, повсеместное вторжение компьютеров сделала способность к применению математического метода важнее, чем когда-либо.

По жестокой шутке истории, впрочем, американское общество выбрало именно двадцатое столетие для того, чтобы становиться все более и более нематематическим (кстати, явление, рассмотренное Моррисом Клайном и вызвавшее у него глубочайшее сожаление). Мы достигли парадоксального состояния, когда из всех так называемых «развитых наций» именно США сильнее всех зависят от программируемых компьютеров и хуже всех интеллектуально оснащены в данном направлении. Предположение о том, что проблема программирования может быть вылечена математическими средствами, мгновенно отвергается как совершенно нереалистичное.

В результате Разработка Программ ограждена от возможности стать поддисциплиной Информатики. Имеет место значительная озабоченность корректностью, но она почти полностью направлена на верификацию программ a posteriori, потому что опять же это легче укладывается в мечту о полной автоматизации. Но, разумеется, многие рассматривают верификацию a posteriori как установку телеги впереди лошади, потому что процедура «сначала программирование, потом проверка» поднимает насущный вопрос, откуда берется программа, подвергающаяся проверке. Ели же она выведена, то верификация сводится к простой проверке вывода. А между тем методология программирования, переименованная в «программную инженерию», стала настоящим раем для гуру и знахарей.

Будучи лишенной того, что обычно рассматривается как основное направление компьютерной науки, американская информатика превратилась в большого неудачника. И мы не вправе винить в этом университеты, поскольку, когда промышленность, наиболее нуждающаяся в их научной помощи, неспособна понять, что это высокотехнологичный бизнес, лучшие университеты оказываются бессильны. Университетам следует быть более просвещенными, чем их окружение, и они способны на это, но не очень (точнее, стараются этого не показывать). В нынешней политической ситуации непохоже, чтобы что-то улучшилось вскоре; в ближайшем будущем нам придется жить среди предрассудков, что программирование – это «настолько просто, что им могут заниматься даже члены республиканской партии».

Остин, 26 августа 1995

Prof. dr. Edsger W. Dijkstra
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-1188
USA

Конец Информатики?

«Цена надежности – это погоня за предельной простотой.
Такая цена по карману не каждому богатому»
Сэр Энтони Хоар, 1980..

В академической науке, промышленности и в мире коммерции широко распространена уверенность, что Информатика стала совершенной наукой и что, следовательно, она «выросла» от теоретической научной проблемы до практической дисциплины для инженеров, менеджеров и промышленников, т.е. в основном людей – и таких немало! – которые готовы применять научные приложения в случае очевидной выгоды, но которые чувствуют себя довольно неуверенно в процессе их разработки, потому что они не понимают, к чему приводит проведение исследований с непонятными целями и негарантированными результатами. Эта всеобщая уверенность, впрочем, оправдана только в том случае, если мы акцентируем внимание на решенных задачах Информатики и забываем о тех задачах, при решении которых потерпели неудачу, даже если они слишком важны, чтобы их игнорировать.

Все же я замечу, что центральная проблема Информатики, а именно «Как во всем этом не запутаться», до сих пор не решена. Напротив, большинство наших систем гораздо сложнее, чем может считаться разумным, и слишком запутанны и хаотичны, чтобы ими было удобно и надежно пользоваться. Среднего пользователя компьютерной индустрии обслуживают так скверно, что он в любой момент ожидает сбоя своей системы, и мы наблюдали массовое распространение программного обеспечения, нашпигованного ошибками, по всему миру, за что нам должно было бы быть весьма стыдно.

Для нас, ученых, слишком велико искушение переложить вину за печальное состояние дел на недостаток образования среднего инженера, недальновидность менеджеров и злой умысел промышленников, но это никуда не годится. Видите ли, в то время, как мы все знаем, что причиной всех бед является неуправляемая сложность, мы не знаем, ни какой степени простоты можно достичь, ни до какой степени внутренняя сложность разработки в целом должна отражаться на видимых интерфейсах. Мы попросту до сих пор не знаем, насколько сможем выпутаться из этого. Мы до сих пор не знаем, можно ли отличить сложность, присущую самой задаче, от случайной сложности. Мы до сих пор не знаем, будут ли возможны компромиссы. Мы до сих пор не знаем, сумеем ли разработать для сложности осмысленную концепцию, на базе которой сможем построить полезную теорию. Откровенно говоря, мы просто до сих пор не знаем, о чем нам следует говорить, но это не должно нас беспокоить, поскольку это только иллюстрирует, что имеется в виду под «непонятными целями и негарантированными результатами».

Это всего лишь один из примеров. Мораль: ответ на вопрос, является ли Информатика совершенной наукой или нет, зависит только от нашего мужества и нашего воображения.

Остин, 19 ноября 2000 г.

[Написано для Communications of the ACM]

Ответы на вопросы студентов отделения программного обеспечения

EWD1305

Примерно восстановить вопросы оставлено читателю в качестве упражнения.

> Красивые безделушки - не всегда усовершенствование, возьмите, к примеру, последовательность Классная доска => Диапроектор => PowerPoint

- Мне не следует тратить время на компьютер лишь потому, что я ученый-компьютерщик. [Медикам-исследователям вовсе не обязательно самим страдать от болезней, которые они изучают.]
- Это не задача информатики – продвигать «компьютеризацию», скажем, разрабатывая прожорливые приложения и тем самым создавая рынок для следующего поколения вычислительного оборудования. [Медикам-исследователям не следует изобретать новые болезни, чтобы создавать рынок для новых лекарственных средств.]
- Задача Университета – не предлагать то, что просит общество, а давать то, что обществу необходимо. [Те вещи, что общество просит, в основном хорошо понятны, и для них не нужен Университет; Университет же должен предлагать то, что никто больше предоставить не в состоянии.]
- На наше формирование большое влияние оказывают инструменты, которые мы используем, в частности: формализмы, которые мы используем, формируют наш образ мышления лучшим или худшим образом, и это значит, что мы должны быть чрезвычайно осторожны в выборе того, чему учить и чему учиться, потому что разучиться потом совершенно невозможно.
  [Много лет назад, когда мне нужен был новый ассистент, одним из требований было «Не иметь даже понятия о FORTRAN'е», а в высшей школе в Сибири запрещено преподавание BASIC'а.]
- Программист должен уметь демонстрировать, что его программа обладает требуемыми свойствами. Если эта мысль приходит к нему слишком поздно, он наверняка не сможет справиться с этой задачей: только если он позволяет этой цели влиять на разработку, есть надежда, что он справится с ней. Окончательная проверка не обеспечивает этого влияния и по сути дела является телегой, запряженной впереди лошади, но именно это и происходит в программных фирмах, в которых «программирование» и «контроль качества» осуществляют разные группы. [Вряд ли стоит говорить, что эти фирмы не дают никакой гарантии качества.]
- Необходимые приемы эффективного доказательства достаточно формальны, но до тех пор, пока программированием занимаются люди, не владеющие ими, кризис программного обеспечения будет продолжать пребывать с нами и будет рассматриваться как неизлечимая болезнь. А вы знаете, что делают неизлечимые болезни: они приводят к появлению знахарей и шарлатанов, которые в данном случае принимают личину Гуру Программирования.
- Кое-кто из вас сомневается, что упомянутые ранее «приемы эффективного доказательства», столь изящные для маленьких программ, способны масштабироваться, я цитирую, «применимо к устрашающим размерам и явной сложности большинства программ». Что ж, они окажутся бессильны, если вы попытаетесь использовать их для распутывания хаоса, созданного группой некомпетентных, неорганизованных программистов. Их сила проявляется в фазе конструирования, когда (i) они приводят к значительно более коротким исходным текстам, чем созданные без их помощи, и (ii) длина вывода программы растет не

быстрее, чем линейно, с ростом самой программы. Наконец, программы, произведенные таким способом, получаются бесконечно лучшими, чем обычный программный хлам. Мы не должны забывать, что программисты живут в мире искусственно созданных сущностей, это отличает их от большинства других ученых. Программист не должен спрашивать, насколько применимы технологии надежного программирования, он должен создать мир, в котором они применимы; это единственный путь обеспечить высокое качество разработки. Добавлю цитату из EWD898 (1984): «Возможности машины дают нам теперь достаточный простор для создания хаоса. Неограниченные возможности для запутывания всего на свете! Выработка строгой интеллектуальной дисциплины сохранять вещи достаточно простыми – это настоящий вызов в этой среде, как технический, так и образовательный».

- В ответ на вопрос, зачем мы учим бесполезным вещам, которые промышленность игнорирует, я отправлю вас к документу EWD920 (1985). Позвольте процитировать один параграф оттуда: «Вернемся к нашему первоначальному вопросу: может ли наука о компьютерах спасти компьютерную промышленность? Мой ответ таков: "Если компьютерную промышленность вообще можно спасти, только наука о компьютерах способна сделать это". Но может пройти немало времени, пока компьютерная промышленность – в особенности компании, крепко стоящие на ногах – согласятся с этой точкой зрения. Почти наверняка это потребует больше времени, чем тот ограниченный период, на который они строят свои будущие планы. Тем временем академический мир – который традиционно строит гораздо более далеко идущие планы – не имеет выбора. Он вынужден совершенствовать лучшие навыки, которые может выработать информатика, и обучать им; чем поддаваться внешнему давлению и распространять сегодняшние заблуждения, лучше прекратить эту деятельность вовсе». Но чтобы подчеркнуть, как много терпения нам понадобится, позвольте привести еще одну старую цитату (1988): «Слишком мало людей осознает, что высокие технологии, столь знаменитые сегодня, - это в основном математические технологии». (Выдержка из 2nd David-report, названного так в честь председателя комитета Dr. E.E. David Jr).
- Нет, я боюсь, что Информатика страдает от популярности Интернета. Он привлекает все возрастающую – если не сказать, сметающую все на своем пути – массу студентов с весьма слабыми научными склонностями, и при более близком знакомстве он только укрепляет господствующую (и несколько вульгарную) одержимость своей скоростью и объемами.
- Да, я разделяю ваше беспокойство: вряд ли можно научиться хорошо программировать, пройдя соответствующий курс. Ситуация сходна с аналогичной в математике, где программа обучения ориентирована на математические результаты; как заниматься самой математикой – студент должен впитать самостоятельно, так сказать. Один из доводов в пользу манипуляции символами и вычислительных доказательств – им гораздо легче научить, чем словесно-графическим доказательствам. Широкое введение курсов подобной вычислительной методологии, впрочем, столкнется с непреодолимыми политическими проблемами.
- В программном бизнесе имеется множество предприятий, которым неясно, что наука может помочь им; им также неясно, что она должна попытаться сделать это.

Остин, 28 ноября 2000 г.