

Почему программное обеспечение такое дорогое? Пояснение для разработчиков аппаратуры

Эдсгер Вибе Дейкстра

Недавно я получил приглашение от одной солидной (и продолжающей расти) компании, производящей вычислительное оборудование. В течение многих лет основной их продукцией было высококачественное ана-

логовое оборудование; впоследствии, однако, цифровые компоненты начали играть всё более значительную роль. Руководство корпорации осознало, что так или иначе компании неизбежно придётся столкнуться с областью программного обеспечения (ранее неизвестной для неё), 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 ещё 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 на-

чала 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 всё рав-

но не будет обладать исключительными правами на моё 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 дешёвого 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 всё же весьма трудно. Мне хотелось бы подчеркнуть с самого начала, что нынешние проблемы в разработке программно-

го обеспечения только частично могут быть объяснены явным недостатком компетентности вовлечённых в неё программистов. Я делаю это в самом начале, поскольку это объяснение, хотя и не является чем-то необычным, всё же слишком простое.

Вполне естественно, что для руководителя аппаратной части весьма досадно произвести то, что мы справедливо оцениваем как надёжную машину с превосходным соотношением цена/производительность, и впоследствии наблюдать, как к моменту отправки полной системы потребителю оказывается, что эта самая система нашпигована ошибками, а её 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 ещё вдобавок требуют сми-

риться с тем, что по мере того как он работает всё эффективнее год от года, группу программного обеспечения награждают за её 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 преимуще-

ственно 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 её вероятностной природе: не следует путать очень высокую вероятность с пол-

ной 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 всё ближе к своим пределам а это происходит постоянно и допуски становятся всё более жёсткими, контроль за допусками становится основной заботой производителей аппаратуры.

По сравнению с разработчиком аппаратуры, который постоянно борется с непокорной природой, разработчик программного обеспечения живёт в раю, так как он строит свои творения только из нулей и единиц. Ноль есть ноль и единица есть единица: в его строительных блоках нет места нечёткости, и общее инженерное понятие о чем-то, находящемся в пределах допусков, здесь просто неприме-

нимо. 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 всё 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 неизменны-

ми, мы можем повторно использовать старую разработку. На самом же деле мы вынуждены непрерывно заниматься разработкой, поскольку они всё 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 недостаёт 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 оста-

ётся практически неизменной: он всё 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 нынеш-

нее оборудование и в самом деле заслуживает название аппарата 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 разнообра-

разие 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 ещё 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 всё 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 операцион-

ные 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, донучный, 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 её 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 ещё 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 её намереваются использовать, и используется независимо от того, каким образом она доказана; подобным образом подпрограмма реализует-

ся независимо от того, каким образом её 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, её можно глубокомысленно обсуждать, но окончательная разработка может иметь структуру, никогда ранее не обсуждаемую. Поэтому команда разработчиков *должна* изобретать свой собственный язык, чтобы обсуждать её, *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 сло-

ва 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 та-

ким 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, святость была булевским атрибутом: страница либо была священной, либо нет. Постепенно оказалось, что страницы должны иметь счётчик святости, а первоначальный булевский атрибут стал отвечать на вопрос, положителен счётчик святости или нет.

Если бы во время этих дискуссий кто-то посторонний вошёл в нашу комнату и послушал нас пятнадцать минут, он сказал бы: «Не верится, что вы сами знаете, о чём говорите».

Наш ответ был бы таким: «Да, вы правы, и именно об этом мы и говорим: мы пытаемся выяснить, о чём именно нам следует говорить».

Я описал эту сцену столь подробно, поскольку хорошо помню её и потому что считаю её достаточно типичной. Постепенно вы достигаете весьма формального и хорошо определённого 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 святости. 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 спо-*

собом, которым я смог достичь корректного формального подхода, было введение *новых синтаксических категорий*! Эти новые синтаксические категории описывали последовательности символов, которые были *бесмысленными* с точки зрения разбираемого языка, но *необходимыми* для понимания и проверки алгоритма разбора при разработке. Моё 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 даёт 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 удо-

вольствием 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 освещённой 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 дисциплину, 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 обречённого довёл своё выступление до самого горестного конца.

Когда я закончил и загорелся свет, я был поражён шквалом аплодисментов из задних рядов, которые были для меня невидимы! Как оказалось, мне досталась весьма разно-

родная аудитория, восхищённые программисты в задних рядах, а на передних их начальники, которые были чрезвычайно раздражены моим выступлением: явно указав количество используемых изобретений, я представил задачу программирования ещё 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, ещё слишком молодых, чтобы уйти на пенсию). Од-

на из проблем, вызываемых непониманием со стороны руководителя над программистами, заключается в том, что он думает, что его подчинённые 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, произведённым 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 трудоёмко, 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 преуспеее в этом, ваша цена будет равна стоимости куска золота равного с вами веса. (Я не очень тяжёлый).

Тем не менее очевидно, что многие руководители наносят ущерб, запрещая думать и подталкивая своих подчинённых 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 всё 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 её, и если она не работала, он обследовал сигналы осциллографом и налаживал ёмкости. Когда она начинала работать, он начинал ва-

рыировать напряжение источника питания в пределах 10 процентов, подстраи́вал её и т.д., пока не получалась цепь, корректно работающая во всём диапазоне заданных условий. Он делал продукт, для которого мог убедиться, что он работает во всём диапазоне. Разумеется, он не испытывал её 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 выдаёт корректный результат, *нельзя* экстраполировать, что программа в полном порядке; поэтому он пытается математически доказать, что его программа удовлетворяет требованиям.

Даже простой намёк 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 осведомлёнными. Неудивительно, что в мире вычислительного оборудования понимание потенциальной неадекватности процедуры тестирования является всё ещё 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 неё. 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 введени-

ем 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 её важнейшей, поскольку, пока она не достигнута, разработчики оборудования не поймут, за что же именно ответственны программисты. Согласно инженерной тра-

диции, завершённая разработка это весь продукт разработчиков: вы построили систему и вот она работает! Если вы не верите этому, просто испытайте её, 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 все-

го 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 своём желании применять эти новые замечательные машины становится чересчур амбициозным;

многие программисты уступили давлению расширить доступные программные технологии за пределы их 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 доказатель-

ство редко является лучшим: последующие доказательства часто на порядок сильнее.

Когда С.А.Р. Ноаре 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-строчных решениях, состряпанных так называемыми профессиональными программистами — или даже учи-

телями 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 успе-

хом и провалом? Я ожидаю помощи от экономических соображений. В отличие от ситуации с оборудованием, где увеличение надёжности обычно влечёт увеличение цены, в случае с программным обеспечением ненадёжность обходится чрезвычайно дорого. Возможно, это парадоксально звучит, но надёжная (а следовательно, простая) программа обходится в разработке и использовании гораздо дешевле, чем (сложная и, следовательно) ненадёжная. 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